Texel Programs for Random-Access Antialiased Vector Graphics

Diego Nehab Princeton University
Hugues Hoppe Microsoft Research

Figure 1: Within each cell of a coarse grid, a texel program encodes several layers of locally specialized graphics primitives, to enable rendering of general vector graphics over arbitrary surfaces, as well as efficient antialiasing.

Abstract

We encode a broad class of vector graphics in a randomly accessible format. Our approach is to create a coarse grid in which each cell contains a texel program — a locally specialized description of the graphics primitives overlapping the cell. These texel programs are interpreted at runtime within a programmable pixel shader. Advantages include coherent low-bandwidth memory access, efficient inter-primitive antialiasing, and the ability to map general vector graphics (including strokes) onto arbitrary surfaces. We present a fast construction algorithm, and demonstrate the space and time efficiency of the representation on many practical examples.

Keywords: scalable vector graphics, texture mapping.

1. Introduction

Vector graphics including filled shapes and stroke outlines are often used for symbolic information such as text, illustrations, and maps. Common formats include Adobe Acrobat (PDF), Windows Metafile (WMF), and scalable vector graphics (SVG). Because evaluating an individual pixel within a vector graphics object requires traversing all its primitives, the object is usually rendered by rasterizing each primitive in a framebuffer, e.g. with a scanline algorithm.

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

As reviewed in Section 2, recent *vector image* schemes support sharp outlines within raster images by adding to each pixel a local approximation of the outline geometry. In effect, the image domain is partitioned into regions, and conventional bilinear interpolation is disabled across these encoded region outlines.

Instead, our aim is to directly model general vector graphics, composed of layered overlapping primitives. Such primitives often include thin strokes that cannot be conveniently antialiased when represented as regions within vector images.

Our approach. We develop a random-access representation for general vector graphics with arbitrary colored layers of filled and outlined shapes. Our key idea is to construct a coarse image grid in which each grid cell contains a local graphics description *specialized* to that cell. This local description is stored as a variable-length string of tokens — a *texel program* — that is evaluated within a programmable pixel shader at rendering time. The complexity of the texel program 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 uniform color are rendered quickly.

We show that traditional vector graphics can be converted into locally specialized texel programs using an efficient rasterization-like clipping algorithm followed by a simple planar sweep. A unique aspect of the algorithm is that it makes a single traversal of the graphics primitives to update all affected cells.

Benefits. One key advantage of texel programs is that they coherently encapsulate all the data involved in rendering a region of the image. Whereas conventional vector graphics rasterization would read a stream of primitives and perform overlapping scattered updates to the framebuffer, we instead cache a small string of specialized primitives, combine the primitives within the shader (albeit with significant computation), and perform a *single* write per pixel. Such low-bandwidth coherent memory access should become advantageous in many-core architectures. (An interesting analogy is the transition from multi-pass simple shading [Peercy et al. 2000] to single-pass complex shading.)

Another benefit is efficient antialiasing. Because texel programs provide a (conservative) list of primitives overlapping the pixel, we directly evaluate an antialiased pixel color for the current surface in a single rendering pass, without resorting to A-buffer

fragment lists [Carpenter 1984]. Although inter-primitive antialiasing adds computational cost, it involves no extra memory access. Moreover, there is opportunity to spatially adapt the supersampling density to local geometric complexity, e.g. falling back to a single sample per pixel in regions of constant color.

Texel programs also inherit advantages demonstrated previously in vector images. The geometric primitives stored in texel programs can use lower-precision (e.g. 8-bit) cell-local coordinates. And, since the vector graphics are evaluated entirely in the shader, they can be mapped onto general surfaces just like texture images.

Limitations:

- Cell-based specialization assumes a static layout of graphics primitives in the domain, although the construction is fast;
- The description of each vector path segment is replicated in all cells over which it overlaps, but this storage overhead is often small since segments typically have small spatial extent;
- 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 triangulation of the shape;
- The current implementation supports only a subset of SVG, e.g. no stylized strokes, gradient fills, or instancing of glyphs;
- Filtered minification requires an ordinary mipmap, but this is true of all other approaches;
- Antialiasing considers all vector graphics primitives rendered onto the current surface, but does not address inter-surface antialiasing such as silhouettes.

Because the cell descriptions have variable sizes, we use an indirection scheme to compact the data. We explore compaction strategies based on indirection tables and perfect spatial hashes.

Main contributions:

- Texel programs for spatially specialized vector graphics;
- Fast construction using path rasterization and a simple sweep;
- Fast computation of approximate distance to a quadratic curve;
- Single-pass, spatially adaptive inter-primitive antialiasing.

2. Related work

Vector images incorporate sharp outlines within a color image by encoding extra information in its pixels. Most schemes enforce a bound on the outline complexity within each image cell, such as two 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], or 2–4 corner features [Qin et al. 2006]. A drawback of fixed-complexity cells is that small areas of high detail (such as serifs on a font glyph or cities on a map) require fine cell grids, which globally increases storage cost. A quadtree structure can provide nice adaptivity [Frisken et al. 2000], but still limits the number of primitives at the leaf nodes. In contrast, the feature-based textures of Ramanarayanan et al. [2004] can store arbitrary lists of path discontinuities in each texel. These discontinuities partition the texels into regions, and traditional bilinear interpolation is overridden based on precomputed region masks.

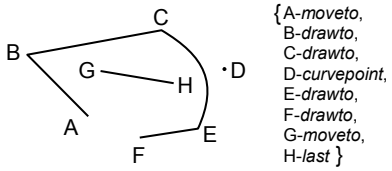
Most prior schemes consider only a single layer of non-overlapping vector primitives. A generalization explored by Ray et al. [2005] is to construct a compositing hierarchy of several vector images. However, on general graphics models such as the examples in this paper, a global compositing hierarchy becomes overly complex and its cost applies uniformly to all pixels. In essence, our scheme adaptively simplifies this hierarchy per cell.

Like the approach of Ramanarayanan et al. [2004], we also store variable-length descriptions within each grid cell. However, rather than maintaining compatibility with a raster image, we instead focus on encoding general vector graphics, including thin stroke primitives that cannot be represented as regions. We also model multiple semitransparent graphics layers, and compute distances for antialiasing. Finally, we design our data structure to encapsulate each cell description into a concise token string that can be efficiently parsed within a shader.

Similar to Frisken et al. [2000] and Qin et al. [2006], our representation lets us compute signed distance to the shape primitives, which permits screen-space antialiasing. While Frisken et al. use a multilinear implicit approximant, and Qin et al. record oriented corner features, we recover signed distance by storing paths of segments and performing ray intersection testing. A unique aspect of our scheme is the runtime computation of a winding number to allow filling of self-intersecting paths. Also, paths let us represent corners accurately without any special processing.

3. Vector graphics representation

Our vector graphics representation is very similar to that of Postscript or SVG. The basic shape primitive is a path consisting of linear and/or quadratic segments specified by a sequence of 2D points. Each point has one of four possible tags: *moveto*, *drawto*, *curvepoint*, or *last*, as shown by this example:



For vector graphics containing cubic curve segments, we adaptively approximate these using one or more quadratic segments.

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 a winding rule, which is even-odd by default. The overall vector graphics consists of a back-to-front ordered list of such layers.

4. Pixel-based rendering evaluation

Most rendering algorithms for vector graphics objects traverse the graphics primitives sequentially, rasterizing each one over a framebuffer. Instead, for our random-access strategy, we must design an algorithm to directly evaluate color at any given point, as requested by a pixel shader program.

The basic approach is to compute the color (including alpha) contributed by each graphics layer at the current pixel and to composite these colors in back-to-front order. A more advanced algorithm for antialiasing primitives across layers is described in Section 6.

For each layer, we seek (1) the absolute distance from the pixel to the path, and (2) for filled shapes whether the pixel is in the path interior. The distance to the path is found simply as the minimum distance to the segments of the path. To determine if the pixel lies in the path interior, we compute the winding number by shooting a ray from the pixel to the right ($+x$) in texture space and summing the signed intersections with the oriented path segments, as illustrated in Figure 2 (see [Foley et al. 1990]). We convert the absolute distance to a signed distance using the even-odd winding number rule.

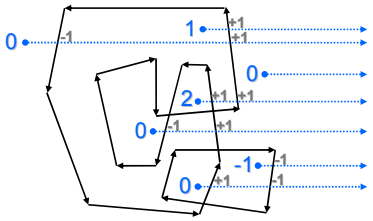


Figure 2: Winding numbers as sums of oriented intersections.

The overall rendering algorithm can be summarized as follows:

```
Initialize the pixel color.  // e.g. white or transparent
for (each layer) {  // ordered back-to-front
  Initialize the cell winding number to 0.
  for (each segment in layer) {
    Shoot ray from pixel through segment,
      and update running sum of winding number.
    Compute absolute distance to segment,
      and update mininum absolute distance.
  }
  Assign sign to distance using winding number.
  if (fill) Blend fill color based on signed distance.
  if (stroke) Blend stroke color based on distance.
}
```

4.1 Linear segments

Given a pixel p and linear segments $\{(b_1, b_2), (b_2, b_3), \ldots\}$, we compute the signed distance d as follows (refer to Figure 3a). For each segment (b_i, b_{i+1}), we find the signed intersection w_i (± 1 or zero if none) and the distance vector v_i:

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

$$w_i = \begin{cases} \mathrm{sign}(b_{i+1,y} - b_{i,y}) & , 0 \le t_i \le 1 \text{ and } q_{i,x} > p_x \\ 0 & , \text{otherwise}, \end{cases}$$

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

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

In principle one should handle the degenerate case of the ray passing through a vertex, but we have found this unnecessary; it occurs so infrequently that we have never noticed any artifacts.

Finally, we combine the results of all segments as:

$$w = \sum_i w_i,$$
$$d = (-1)^{\mathrm{windingrule}(w)} \min_i \|v_i\|.$$

4.2 Quadratic segments

A quadratic segment (b_{i-1}, b_i, b_{i+1}), where point b_i is tagged *curvepoint*, defines a Bezier curve $b(t) = (1-t)^2 b_{i-1} + 2(1-t)t b_i + t^2 b_{i+1}$ over $0 \le t \le 1$. We compute the signed distance from pixel p as follows (Figure 3b).

The intersection points of the ray from pixel p with the (infinite) quadratic curve are found at the roots t_1, t_2 of the quadratic equation $b_y(t_j) = p_y$. (Let us assume $t_1 \le t_2$.) A root t_j corresponds to an intersection point on the curve if $0 \le t_j \le 1$. The orientation of the curve at an intersection point $b(t_j)$ is determined from the vertical direction $b_y'(t_j)$ of its tangent vector

$$b'(t_j) = \left.\frac{db(t)}{dt}\right|_{t=t_j} = 2(b_i - b_{i-1})(1-t) + 2(b_{i+1} - b_i)t.$$

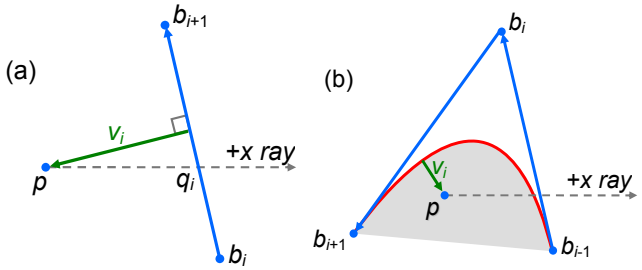


Figure 3: Ray intersection testing and distance computation, for (a) linear and (b) quadratic path segments.

For each root t_j we check if $0 \le t_j \le 1$ and $b_x(t_j) > p_x$, and if so report an intersection with orientation $w_i = \mathrm{sign}(b_y'(t_j))$.

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

Computing the absolute distance to the quadratic Bezier curve involves finding the roots of a cubic polynomial [Qin et al. 2006]. (Unlike in [Loop and Blinn 2005], we must address distance to a bounded curve, i.e. with two endpoints.) Analytic roots are quite expensive to evaluate [Blinn 2006], so an alternative is to use an iterative solver, as discussed by Qin et al.

Instead, we develop a fast approximate technique based on *implicitization*. Accordingly, we convert the Bezier curve to its implicit quadratic representation, given by the Sylvester form of its resultant [Goldman et al. 1984]:

$$f(p) = \beta(p)\delta(p) - \alpha^2(p) = 0, \text{ with}$$
$$\beta(p) = 2\,\mathrm{Det}(p, b_{i+1}, b_i),\ \delta(p) = 2\,\mathrm{Det}(p, b_i, b_{i-1}),$$
$$\alpha(p) = -\mathrm{Det}(p, b_{i+1}, b_{i-1}), \text{ and } \mathrm{Det}(p, q, r) = \begin{vmatrix} p & q & r \\ 1 & 1 & 1 \end{vmatrix}.$$

The first-order Taylor expansion of $f(p) = 0$ at p defines a line, and the closest point p' to p on this line is then 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 Bezier case, Goldman et al. [1984] show that inversion can be obtained by

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

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 preferred choice is

$$\bar{u} = \frac{\alpha + \delta}{\beta + \alpha} \text{ which gives } \bar{t} = \frac{\alpha + \delta}{(\beta + \alpha) + (\alpha + \delta)}.$$

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 u_1' or u_2' become undefined).

We then compute the distance vector

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

(Implementation of these formulas is much simpler if the Bezier points b are translated so as to place p (and resp. p') at the origin.)

Although there are cases in which the clamping produces the wrong endpoint, in practice we have found it to be a sufficiently

good approximation, especially when p is a small distance away from the curve, as demonstrated in Figure 4. Stroke paths whose widths are poorly approximated are converted to filled primitives.

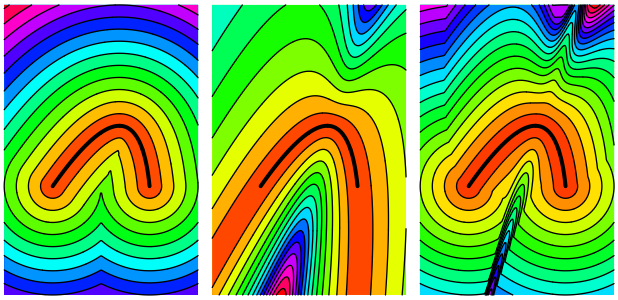


Figure 4: Distance to quadratic curve. The ground truth (left) involves cubic polynomial roots. A first-order implicit approximation (middle) fails to capture the segment endpoints. Our approximation (right) is fast, and visual analysis reveals that it is sufficiently accurate in the vicinity of the curve.

4.3 Intra-primitive antialiasing

From the texture-space signed distance d, we know if the pixel lies within the interior of a filled path and/or within half a stroke width of a stroked path. We use the precise magnitude of d for subpixel antialiasing as in [Loop and Blinn 2005].

For **filled** paths, we transform the texture-space distance d into a screen-space distance d' for isotropic antialiasing using $d' = d/\sigma$ where $\sigma = \max(\|J_x\|, \|J_y\|)$ and the 2×2 matrix J is the screen-to-texture Jacobian reported by the pixel shader derivative instructions. Therefore, each graphics layer successively modifies the color c computed at the pixel using the blend operation:

$$c = \text{lerp}(c, \mathit{fillcolor}, \alpha) \text{ with } \alpha = \text{clamp}(d' + 0.5, 0, 1)\,.$$

Alternatively, we could apply anisotropic antialiasing using the technique of Qin et al. [2006], by transforming each texture-space distance vector v_i to screen space (using the inverse of the same Jacobian matrix J) and measuring its magnitude there:

$$v_i' = J^{-1}\, v_i\,,$$
$$d' = (-1)^{\text{windingrule}(w)} \min_i \|v_i'\|\,.$$

For **stroked** paths, the situation is slightly more complicated. Because the stroke width w is expressed in texture units, we must consider the distance d before its transformation to screen coordinates. Also, thin strokes may have width less than one pixel (Figure 5). For isotropic antialiasing, the blending operation is:

$$c = \text{lerp}(c, \mathit{strokecolor}, \alpha)\,, \text{ with}$$
$$\alpha = \text{clamp}\left(\left(|d| + \tfrac{w}{2}\right)/\sigma\,, -0.5, 0.5\right) - \text{clamp}\left(\left(|d| - \tfrac{w}{2}\right)/\sigma\,, -0.5, 0.5\right)$$
$$= \text{clamp}\left(\left(|d| + \tfrac{w}{2}\right)/\sigma + 0.5, 0, 1\right) - \text{clamp}\left(\left(|d| - \tfrac{w}{2}\right)/\sigma + 0.5, 0, 1\right).$$

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

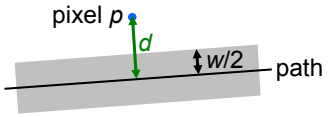


Figure 5: Antialiasing of a thin stroke primitive.

5. Cell-specialized vector graphics

Our strategy is to specialize the vector graphics definition in each cell to obtain compact storage and efficient runtime evaluation.

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, a vector primitive must be included if:

(1) it is filled and its interior overlaps the cell, or

(2) it is stroked and the width of the stroked path overlaps the cell.

However, a primitive should also be included if its screen-space distance d' (see Section 4.3) is less than 0.5 pixels from the cell boundary, because it could then contribute a blended antialiased color. Unfortunately, this screen-space distance varies with the runtime viewing parameters – specifically, the size of cells in screen space. As the rendered cells shrink to the size of a screen pixel, the width of the antialiasing ramp becomes as large as a whole cell. At extreme minification levels where several cells are mapped into individual pixels, antialiasing breaks down, and one must instead transition to a conventional image mipmap pyramid.

To allow good 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 6.

Growing this extended cell allows a coarser mipmap pyramid, but increases the size of the texel programs 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

Conceptually, we consider each cell independently, and clip the entire set of graphics primitives to the boundary of the extended cell. For stroked paths, the clipping is extremely simple. For filled paths, it involves polygon clipping [Sutherland-Hodgman 1974] and its straightforward extension to quadratic Bezier segments.

To achieve the most effective specialization, we exploit knowledge of our particular rendering algorithm. Specifically, because interior testing is based on shooting a ray in the $+x$ direction, we omit any segment reported by polygon clipping if the segment lies on the top, left, or bottom boundaries of the extended cell, as it has no influence on the signed distance within the cell. This is best shown with the example of Figure 7.

We express all point coordinates in a $[0 \ldots 1]^2$ coordinate system over the extended cell. In the rare case that a middle *curvepoint* Bezier control point lies outside the extended cell, we recursively subdivide the curve.

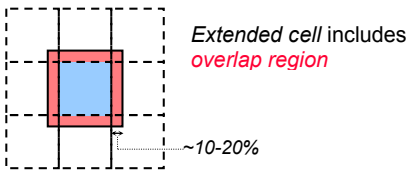


Figure 6: For correct antialiasing, each cell stores all primitives that affect an extended cell region.

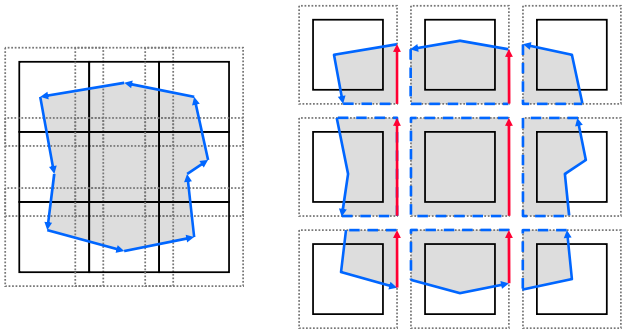


Figure 7: For a filled shape, polygon clipping to each extended cell gives the paths on the right; the dashed segments have no effect on our winding rule computation and are therefore removed, whereas the red segments are kept.

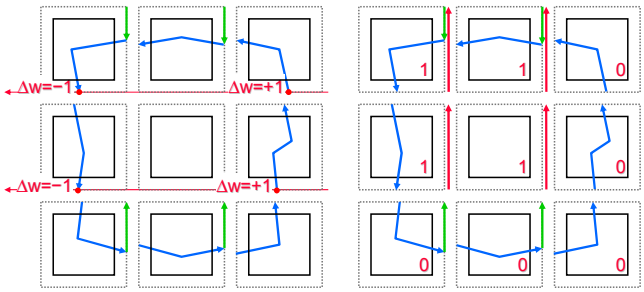


Figure 8: To recover correct winding numbers within the cells, our fast algorithm inserts segments (green) on right boundaries, and makes a right-to-left sum of bottom-boundary intersections to appropriately insert right-boundary edges (red).

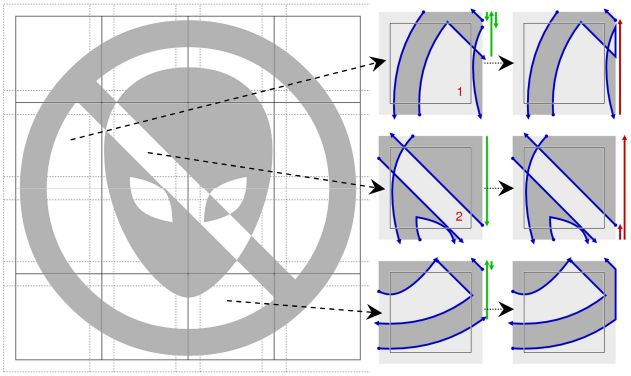


Figure 9: Demonstration of our fast cell-specialization algorithm on a complex shape with self-intersections.

5.3 Fast construction algorithm

Of course, evaluating polygon clipping of all primitives against all cells is too inefficient, especially for large filled shapes. One acceleration technique would be a recursive subdivision algorithm over a quadtree of cells, similar to [Warnock 1969].

We develop a more efficient algorithm that streams over the primitives *just once*, and directly enters each path segment into all cells that the segment overlaps.

The main difficulty is to introduce segments on the right sides of the cells (red in Figure 7) to define the correct winding number for a filled shape. In effect, adding an upward segment on the right cell boundary uniformly increments the winding number, while a downward segment decrements it. Unfortunately this is a nonlocal problem, since for instance a cell may lie in the interior of the shape and yet not contain any path segments.

We have found a simple robust scheme as follows (see Figure 8). We extend the path segments crossing the right boundary of a cell to the top right corner of the cell (green edges in the figure), and for each path segment crossing the bottom of a cell, we record a change Δw in winding number affecting all cells to the left in that row ($+1$ for upward segment and -1 for downward segment). Then in a fast second pass, we traverse each row right-to-left, integrating the winding number changes, and for each cell with a nonzero winding number we add the appropriate number of upward or downward right-boundary segments (red edges in the figure). The resulting green and red edges are merged together to exactly reproduce the earlier result in Figure 7.

For completeness we provide here a more detailed algorithm:

```
for (each layer) {
  for (each segment in layer) {
    Enter the segment into the image cell(s) in which 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 Δw_c on the cell.
    If the segment leaves a through its lower boundary,
      decrement Δw_c on the cell.
  }
  for (each row of modified cells) {
    Initialize the winding number w=0 (associated with the row).
    for (each span of cells in right-to-left order) {
      Add |w| vertical segments on the right boundary of the cell,
        pointing up if w>0, or down otherwise.
      Merge the cell segments if possible.
      Update the winding number as the running sum w=w+Δw_c.
    }
  }
  Clear modified Δw_c. // efficiently, using linked lists.
}
```

The algorithm just described is designed to exactly preserve winding numbers. For the case of the even-odd fill rule, it can be simplified to preserve just the parity of the winding number. Also, for efficiency we preserve connected sets of segments to avoid extraneous *moveto* points.

Overall the algorithm is extremely fast; it requires less than a second even on our most complicated example with ~100K segments. Figure 9 shows an example with an intricate self-intersecting shape.

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, effectively performing "dead code removal" on the texel program. 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 filled layer fully occludes the cell, and if so remove all the layer behind it. As an example, for the tiger model in Figure 16, the average texel program length is reduced from 9.8 to 7.3 tokens.

5.5 Compilation into texel program

To represent the cell-specialized graphics as a texel program, we define a token stream with a simple grammar. We use the least-significant bit of the color and point coordinates to encode simple states including the termination of the stream itself:

```
{TokenStream} = {Layer}+    // ordered back-to-front
{Layer} = {Color} {Point}*
{Color} = [RGBA]            // (4 bytes)
  // R,G,B,A color channels for the layer
  // the lsb of R,G,B encode:
  // - LastLayer:   is this the last layer?
  // - Stroke:      should the path be stroked?
  // - Fill:        should the path be filled?
  // !Fill && !Stroke : empty layer containing no {Point} records.
  // Stroke: color channel A encodes strokewidth.
  // Fill && Stroke: strokecolor is assumed black.
{Point} = [X Y]             // (2 bytes)
  // X,Y in extended-cell coordinates, quantized to 7 bits.
  // the lsb of X,Y encode the 4 possible
  //    point tags: moveto, drawto, curvepoint, last.
```

The two-byte *Point* records are packed two-at-a-time, so the texel program is a simple stream of 32-bit words. Our encoding lets the same path be simultaneously filled and stroked in the common case that its stroke color is black and fill color is opaque. Table 1 shows the storage size of different path configurations per layer.

As an optimization, we assume that each layer in the texel program 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. first two cells in top row of Figure 7).

Cell contents	Size in words
Empty (white or transparent)	0
Constant color	1
1 linear segment	2
2 linear segments	3
3 linear segments	3
1 quadratic Bezier curve	3
2 quadratic Bezier curves	4
3 quadratic Bezier curves	5

Table 1: Number of 32-bit words required per cell layer as a function of path complexity.

5.6 Texel program evaluation in pixel shader

Within the pixel shader, we interpret the token stream of the texel program and evaluate the rendering algorithm of Section 4. This interpretation involves a set of three nested loops: a loop over layers, a loop over point-pair tokens, and a loop over each point in the point pair. Moreover, within the inner loop there is branching based on the 4 possible point tags.

Due to the SIMD parallelism in current GPU architectures, shader program execution is more efficient if nearby pixels follow the same dynamic branching path. This is true in our context if the pixels evaluate the same texel program, i.e. if the pixels lie in the same texture cell or if adjacent cells have the same texel program (such as in areas of constant color).

Our implementation uses Microsoft DirectX 9. The pixel shader has a total of 167 assembly instructions if the graphics has only linear segments, and 273 instructions when including quadratic paths. The bottleneck is the evaluation of distance to the segments for both antialiasing and strokes; if this is omitted, the shader simplifies to 123 and 179 instructions, respectively.

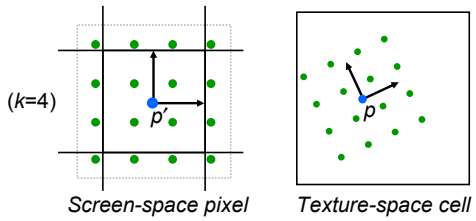


Figure 10: Inter-primitive antialiasing evaluates the texel program at multiple samples (shown in green).

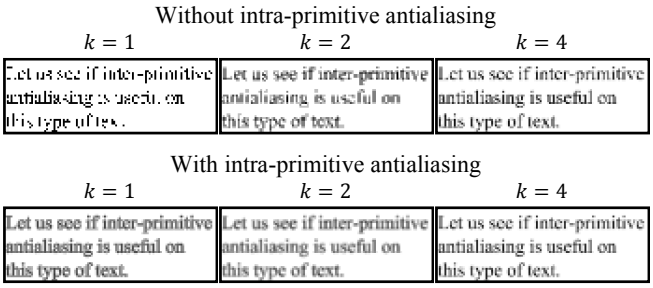


Figure 11: Effect of inter-primitive ($k > 1$) antialiasing, with and without the intra-primitive antialiasing of Section 4.3.

6. Inter-primitive antialiasing

The antialiasing of Section 4.3 (using signed distance to the path) is inexact because the resulting linear blend corresponds to an infinite line at the closest point of the path, when of course the path can have more complex local geometry including corners.

Moreover, such antialiasing is computed per shape primitive, so it ignores inter-primitive interactions. For example, if two filled paths cross at a pixel, the simple inter-layer blending corresponds to the assumption that the paths are always perpendicular.

Correct antialiasing requires considering all the shapes overlapping each pixel. A common approach in traditional rasterization is the A-buffer [Carpenter 1984], which maintains per-pixel lists of fragments, with each fragment containing a subpixel bitmask. This general solution is challenging to implement efficiently in hardware [Winner et al. 1997].

Because texel programs encode the 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 texel programs. Fortunately, our extended cells (Section 5.1) provide the desired margin so we need only consider the current cell. When a pixel footprint grows larger than the overlap region, we transition to a conventional mipmap.

Our system evaluates a $k \times k$ grid of samples within a parallelogram footprint $\{p + Jv \mid -\frac{3}{4} \le v_x, v_y \le \frac{3}{4}\}$ (see Figure 10), and blends them using a cubic filter. We parse the texel program just once, updating all samples as each primitive is decoded. This requires allocating a few temporary registers per sample (accumulated color c, accumulated winding w, and shortest distance vector v_i). For each subpixel sample, we still evaluate the intra-primitive antialiasing of Section 4.3, but with a modified Jacobian matrix $J' = \frac{3}{2k}J$ to account for the modified inter-sample spacing.

Unfortunately, some difficulties with the current shader compilers prevented us from implementing this functionality within a GPU pixel program, so for now we resort to a software emulation.

Figure 11 compares the rendering quality with different antialiasing settings. Of course, the computation has a cost that scales as $O(k^2)$, but it is performed entirely on local data, and is therefore amenable to additional parallelism.

The overall rendering algorithm with inter-primitive antialiasing can be summarized as follows:

```
for (each sample) {
  Initialize the sample color (e.g. to white or transparent).
}
for (each layer) {  // ordered back-to-front
  for (each sample) {
    Initialize the sample winding number to 0.
    Initialize the sample minimum absolute distance to ∞.
  }
  for (each segment in layer) {
    for (each sample) {
      Shoot ray from sample through segment,
        and update running sum of winding number for sample.
      Compute absolute distance to segment,
        and update mininum absolute distance for sample.
    }
  }
  for (each sample) {
    Assign sign to sample distance using sample winding.
    if (fill) Blend fill color over sample color
            based on sample signed distance.
    if (stroke) Blend stroke color over sample color
            based on sample absolute distance.
  }
}
Combine the resulting sample colors to obtain the pixel color.
```

Several improvements could be explored in future work:

- Both the number and distribution of samples could be adapted [Laine and Aila 2006].
- The sampling density (e.g. k) could adapt to the complexity of each cell; it could even be encoded within the texel program.
- Letting multiple samples share the same texture y value would allow reuse of horizontal intersection points.
- Because the footprints overlap in screen space, some samples could be shared between pixels if hardware would permit it.

7. Storage of nonuniform cells

Texel programs are variable-length, so we need a data structure to pack them in memory. Note that the token strings are self-terminating, so it is unnecessary to store their sizes explicitly.

7.1 Indirection table

An elegant solution is to simply concatenate the token strings in raster-scan order into a memory buffer (Figure 12a), letting a 2D indirection table contain pointers to the start of the strings. Cells with identical strings share use the same string instance, although we only perform this instancing for row-adjacent cells to preserve memory coherence. For larger datasets, we introduce a two-level indirection scheme: one 32-bit pointer for the start of each image row, and a second 16-bit offset for each string within the row.

This simple solution is possible with the DirectX 10 API, but unfortunately all the necessary elements (OS, drivers, etc.) did not arrive in time for us to demonstrate it.

Instead we had to resort to storing data in 2D textures under DirectX 9. The complication is that 2D textures are actually stored using an internal tiling structure (optimized for 2D coherence), so our 1D token strips (Figure 12b) become fragmented in memory. Although we are able to achieve excellent packing of the strips into the 2D texture using a greedy best-fit heuristic, this packing requires modifying the order of the strips, which results in further loss of memory coherence.

7.2 Variable-rate perfect spatial hashing

Some vector graphics objects are quite sparse, so we have also explored replacing the indirection table with a perfect spatial hash function (Figure 12c) similar to [Lefebvre and Hoppe 2006]. Thus, the undefined cells of the domain are identified using a domain bit image which requires only a single bit per cell. An important difference is that the data records (token strings) are variable-sized and therefore stored as strips in the hash table. Let $s(c)$ denote the strip size of cell c. As in [Lefebvre and Hoppe 2006], the hash is defined using a 2D offset table Φ:

$$h(c) = (c + \Phi[c \bmod \bar{r}]) \bmod \bar{m}\,,$$

where $\bar{r}\times\bar{r}$ and $\bar{m}\times\bar{m}$ are the dimensions of the offset and hash tables respectively.

The construction of the offset table Φ follows the same heuristic strategy as in [Lefebvre and Hoppe 2006]: offset vectors $\Phi[q]$ are assigned in order of decreasing number of dependent data elements. However, rather than counting the number $|h_1^{-1}(q)|$ of dependent data records, where $h_1(c) = c \bmod \bar{r}$, we find that is better to count the total data size $\sum_{c\in h_1^{-1}(q)} s(c)$ of these dependent records.

To assign $\Phi[q]$, we consider all possible offset vectors (starting from a random one) until finding one that does not create any hash collisions. An improvement is to encourage placing strips adjacent to each other by first finding an invalid offset vector and then looking for the first valid offset.

The indirection table better preserves data coherence and allows instancing of texel programs, while the hash is more concise in the case of sparse data.

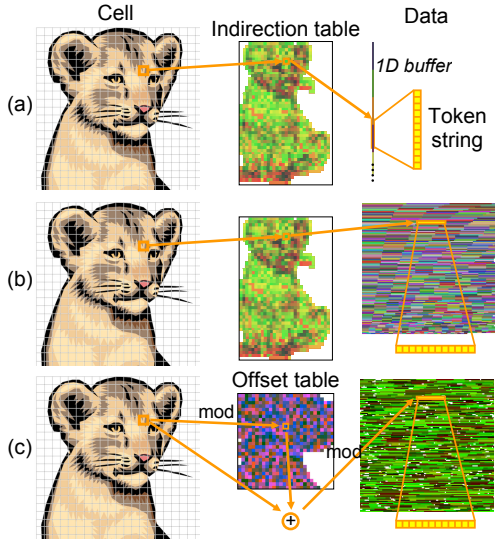


Figure 12: Our data packing approaches: indirection tables and variable-rate perfect spatial hashing.

8. Results and discussion

All results are obtained using Microsoft DirectX 9 and an NVIDIA GeForce 8800 GTX with 768MB, in an 800^2 window. The examples in this section use an overlap region of size 10-20% (depending on the maximum stroke width), isotropic intra-primitive antialiasing, and no inter-primitive antialiasing.

Timing analysis. As discussed in Section 5.3, construction takes less than a second even on our most complex example. Figure 13 plots rendering rate of the texel program representation as a function of cell grid resolution, for the lion in Figure 1. At coarse grid resolutions, the large cells increase the number of primitives per cell, which in turn increases the per-pixel rendering cost, thus leading to slower rendering rates. At the other end, as the grid resolution becomes very fine, the majority of cells contain a single constant color, so rendering speed reaches a plateau.

Our shading evaluation 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 the full texel program but avoid nearly all computation using the parsed primitives, and the frame rates increase by a factor of 3-4, thus indicating that we are presently compute-bound. Therefore, performance will benefit greatly from additional ALU cores in future hardware.

The pixel shader makes several coarse-grain branching decisions, based on the number and types of primitives in the texel program. Fortunately, these decisions are identical for nearby pixels accessing the same program, so the SIMD branching penalty is reduced.

Space analysis. Figure 13 also plots memory size as a function of cell grid resolution. In comparison, the original SVG text description is 12 KB, and its traditional parametric encoding as vertex and index buffers is about 30KB. At coarse grid resolutions, the storage overhead with respect to this traditional parametric encoding is small, because most vertices of the vector shape appear in just one image cell. There are newly introduced vertices at the boundaries of the cell, and this cost diminishes as the cells are made larger. As the cell grid resolution increases, storage increases quadratically (just as in an ordinary image) due to the indirection table.

To get good rendering performance, we have (manually) selected grid sizes finer than we would have desired. Table 2 shows these for all datasets. Note that the memory sizes are on the same order as a typical image representation (but of course texel programs allow resolution-independent rendering).

Figure 14 shows a histogram of texel program sizes again for the lion in Figure 1. The most common type of cell is one containing a single token indicating a constant color. Figure 15 shows an example where a perfect spatial hash function is used to access the texel programs on a vector graphics with a sparse set of strokes.

Examples. Figure 16 presents a collection of vector graphics examples of various types, and Table 2 reports on their complexities, sizes, and rendering rates. Our representation is trivial to map onto surfaces as demonstrated in Figure 1.

9. Summary and future work

Texel programs are constructed by locally specializing a vector graphics description to the cells of a grid using a fast algorithm. The texel programs provide efficient random-access evaluation of composited layers of filled and stroked shapes, complete with antialiasing and transparency.

Avenues for future work include:

- Extension of texel programs to allow more rendering attributes (e.g. gradient and texture fills), as well as instancing of sprites.
- Improvements to the inter-primitive antialiasing algorithm.
- Generalization of the concept of cell-based specialization to other applications besides vector graphics.

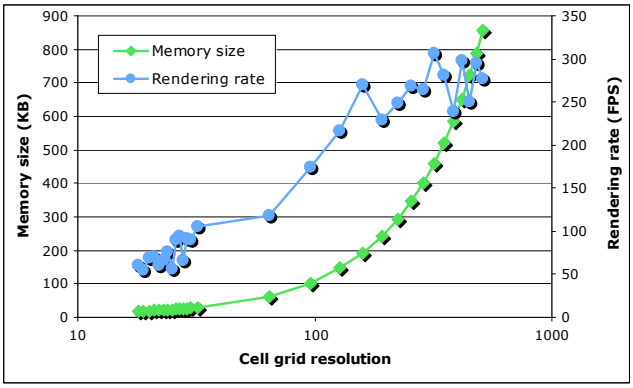


Figure 13: Memory usage and rendering rate as a function of cell grid resolution for the lion in Figure 1.

Dataset	Input #Verts	Conv. time (sec)	Texel program representation				Render (fps)
			Size (KB)	Cell grid	Cell size Avg.	Cell size Max.	
Lion	2080	0.052	60	41×64	5.2	22	320
Boston	140399	0.901	617	512×462	4.5	100	76
Siggr. logo	2420	0.018	99	64×41	9.9	35	226
Hygieia	9922	0.085	214	109×256	4.4	12	570
Tiger	21278	0.159	267	120×128	7.3	80	107
Butterfly	5669	0.013	85	32×20	15.7	60	77
Picasso	7717	0.083	199	53×64	11.2	63	81
Denmark	101386	0.387	406	256×198	4.1	67	97
Floor plan	91887	0.368	305	512×176	6.7	60	96
CAD	22393	0.168	258	256×199	5.6	54	179
Rollerblader	4122	0.036	134	78×128	3.6	26	292

Table 2: Quantitative results, including input vertices, construction times, and texel program statistics. (Cell sizes are in 32-bit words, and average cell size considers only nonempty cells.)

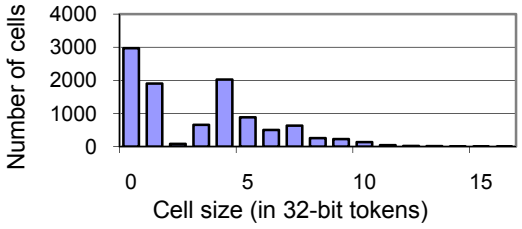


Figure 14: Histogram of texel program complexity.

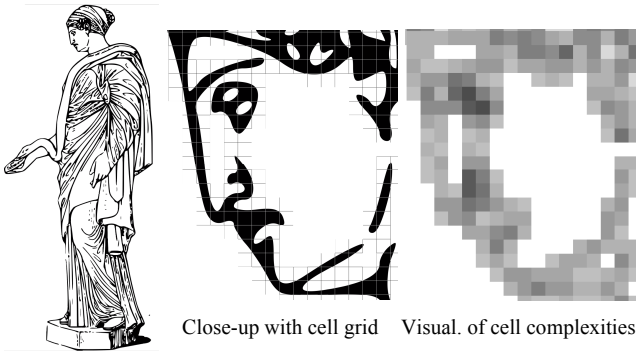


Figure 15: Example using a perfect spatial hash function.

Figure 16: Example results, and close-ups showing good antialiasing. Inset images reveal cell grid and magnified graphics.

References

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.

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.

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.

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

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*.

PEERCY, M., OLANO, M., AIREY, J., AND UNGAR, J. 2000. Interactive multi-pass programmable shading. *ACM SIGGRAPH*, 425-432.

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

RAMANARAYANAN, G., BALA, K., AND WALTER, B. 2004. Feature-based textures. *Eurographics 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.

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.

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.