Real-Time Texture-Mapped Vector Glyphs

Zheng Qin   Michael D. McCool   Craig S. Kaplan
Computer Graphics Lab, School of Computer Science, University of Waterloo

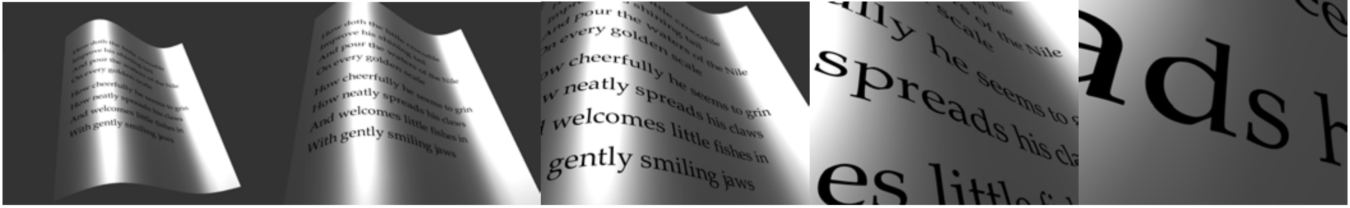

Figure 1: *Vector texture demonstrating resolution-independent antialiased edges at over 90fps (on a NVIDIA 7800GT at* 512×512*).*

Abstract

We present a vector graphics representation suitable for real-time rendering on GPUs. Our representation can be used in place of a texture map, and renders precise antialiased edges at any magnification. A combination of texture data and procedural computation is used to evaluate an exact signed distance to a contour and its gradient. An optimized uniform grid accelerator is created using Voronoi analysis and redundancy elimination, so only the distances to a small constant number of features need be computed at every access. Contours and sharp features can be exactly reconstructed using a constant amount of computation per pixel. Our representation supports inexpensive high-quality anisotropic antialiasing as well as special effects such as outlining (with both rounded and sharp miters) and embossing.

We have applied our representation to the important application of glyph rendering. Variations in glyph complexity are handled by storing different glyphs at different grid resolutions. Large blocks of glyphs can be rendered efficiently with a single indirection through an index texture.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Antialiasing I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: vector graphics, texture mapping, antialiasing, distance function, font rendering

1 Introduction

Vector graphics image representations are a useful alternative to raster image representations for images that contain precise geometric features with sharp intensity boundaries. Examples include logos and other symbols; user-interface buttons and indicators; cartoons, graphs, and line drawings; and text rendered using outline fonts. A vector graphics image is described with a collection of geometric primitives. A raster image, on the other hand, approximates a conceptually continuous function by interpolating a regular grid of samples.

We would like to use vector graphics images in the same way we use raster images in real-time rendering: as textures. Although real-time graphics accelerators do process geometric descriptions of scenes, texture maps and programmable shaders are designed around the manipulation and generation of raster images. This is because raster images can be easily point-sampled at random locations in constant time.

Vector graphics can be used in a real-time 3D rendering context by converting them into 2D raster images, a process which (of course!) can be accelerated by rasterization hardware. However, when magnified, the raster nature of sampled images is apparent. Furthermore, due to perspective distortion and texture map parameterization, the magnification of an image can vary spatially. What is needed is an image representation that can be sampled in constant time, but supports an accurate representation of sharp features at all magnifications. It is also important that this representation support efficient anisotropic antialiasing over continuously varying scales.

While we eventually intend to extend our approach to general vector graphics, in this paper we have focused on one specific problem: the rendering of text. Rendering text is a classic problem in computer graphics, and one of the original motivations for antialiasing [Crow 1977; Crow 1978; Warnock 1980]. The standard approach to rendering text is to prerender antialiased glyphs into a table and then composite together images drawn from this table. However, this means that the glyphs are stored at a fixed (and usually low) resolution. Mapping them onto a 3D surface then requires resampling, which can degrade quality and introduce artifacts. Direct rendering of the glyphs on the GPU as we propose avoids these problems, since the antialiasing can be adapted procedurally to the local spatial distortion.

In this paper we consider only the problem of rendering *display text*, not *body text*. For maximum readability, body text needs to be aligned with and fitted to the display grid, a process supported by hinting, or subtle variations in the shape of the characters. Hinting assumes a fixed-scale, orthographic projection. Hinting cannot be supported in a texture map representation since the characters will be dynamically deformed by surface parameterization and perspective. For display text, the shape of the glyph is represented independently of the sampling grid.

Glyphs in scalable fonts are represented using geometric contours consisting of line segments, quadratic splines, and cubic splines. To render text, *many* of these glyphs have to be placed in an image with precise positioning relative to one another. The representations of the glyphs themselves can be precomputed, but

Copyright © 2006 by the Association for Computing Machinery, Inc.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212) 869-0481 or e-mail permissions@acm.org.
I3D 2006, Redwood City, California, 14–17 March 2006.
© 2006 ACM 1-59593-295-X/06/0003 $5.00

it must be possible to quickly lay out and modify an arrangement of glyphs interactively. However, the problem of text rendering is simpler than that of general vector graphics: although glyphs must be instanced many times, glyphs in normal text rendering do not exceed a certain density in any one area and usually do not overlap. In the rare cases that glyphs do overlap, for instance for accents or strikethroughs, it is with a constant offset and we can generate new glyphs for these cases. Bounding box overlap must be supported for kerning, but this is only in the horizontal direction in normal text layout. We can handle more general cases such as support for glyph rotation or more complex overlap at some additional cost in memory and shader execution time.

This paper makes the following contributions:

1. An efficient anisotropic antialiasing technique for texture-mapped vector graphics (Section 4);

2. A GPU-based representation of contours, including an inside/outside test, based on *exact* evaluation of distance-to-feature functions (Sections 3 and 5);

3. A packed grid accelerator structure based on Voronoi analysis supporting efficient constant-time evaluation of signed distance-to-contour functions (Section 6);

4. A sprite mapping technique that supports the dynamic composition of large numbers of glyphs in a single procedural texture map (Section 7); and

5. Techniques for supporting special effects such as outlining, miter rules, and embossing (Figures 6 and 8).

2 Previous Work

Other researchers have developed approaches for representing vector graphics using sampled representations. Some of these representions are aimed at or can be adapted to procedural texture mapping, real-time rendering, and GPU computation.

Adaptive distance fields [Frisken et al. 2000] can represent polygonal boundaries of 2D and 3D shapes to arbitrary precision, but they only approximate corner features and require a $O(\log(1/\varepsilon))$ random-access lookup cost for precision ε, since more levels of a quadtree (or octtree) need to be traversed near corners. The ADF has been applied in the Saffron text rendering engine, after having been extended with a biquadratic approximation of the sampled distance field [Frisken et al. 2005] and an explicit representation of the distance field around sharp corners [Perry and Frisken 2005; Frisken and Perry 2004, among several other patents]. Combining explicit corners and biquadratic interpolation reduces the lookup and storage cost but requires use of different specialized cell types in different parts of the image. A method using procedural shading and texture mapping to compute and combine distance fields at every pixel is also described: The distance field is decomposed into cells and a region containing the support of each cell is rendered with geometric primitives; the distance field in each cell is then evaluated using shaders. They also mention the use of Voronoi analysis to find an optimal cell decomposition. However, unlike our approach, their representation does not include a shader-based approach to select cells and so does not create a unified representation that can be evaluated as an encapsulated procedural texture.

Silhouette maps [Sen et al. 2003; Sen 2004] give constant time lookup for random access, and can be implemented as procedural textures, but can only represent a limited class of edge structures in a single texel. For instance, two edges cannot both go through the same side of a texel. Feature-based textures [Ramanarayanan et al. 2004] can represent an unbounded number of boundaries in a single texel and use the notion of reachability to generalise bilinear interpolation in the presence of edges. However, their representation is complex and does not guarantee constant-time lookup, since an arbitrary number of features are stored per texel. Neither approach explicitly handles anisotropic antialiasing or mitering rules. Bixels [Tumblin and Choudhury 2004] are similar to silhouette maps, but with generalized interpolation rules similar to those of feature-based textures. We do not consider interpolation of colours in the present work. Instead we focus on the accurate and efficient representation of closed contours and bilevel images.

Recent work uses graphics hardware to evaluate an implicit representation of glyph contours [Loop and Blinn 2005] for real-time rendering, the same goal as this paper. However, their approach requires segmenting the contour and embedding each convex region in a triangular element. To represent the entire contour, a mesh covering each glyph is generated. The representation is not completely embedded in a procedural texture, and requires a rendering mechanism similar to that described in [Perry and Frisken 2005], so “texture mapping” a surface requires computing the intersection of two meshes and modification of the base geometry. In contrast, our approach completely separates the mesh and the texture representation. They also only approximate the distance field very close to the contour. Unfortunately, advanced features like mitering and embossing require an accurate distance field in a relatively wide region around the contour.

We extend our vector representation with a mechanism to dynamically combine the precomputed representations of a large number of glyphs on a single page. Our approach is similar to that of texture bombing [Glanville 2004], texture sprites [Lefebvre et al. 2005], and pattern based procedural textures [Lefebvre and Neyret 2003]. We combine the idea of sprites with that of multiresolution texture maps [Kraus and Ertl 2002] to permit different amounts of storage to be used for glyphs with different levels of complexity.

3 Outline of Representation

Our goal was to design a semi-procedural representation of vector graphics images that could be used as if it were a random-access texture map in a shader. The representation must support arbitrary contours and efficient antialiasing. Also, as GPU fragment shaders do not support control flow without a loss of efficiency, and since such support is not yet universal, we decided to avoid a dependency on control flow in our representation.

Our representation will consist both of data structures stored in texture maps and shader code to interpret that data. However, our system is implemented using the object-oriented Sh GPU programming system [McCool et al. 2004; McCool and Du Toit 2004], which allows us to strongly encapsulate our representation and use it in any context that a raster texture could be used.

We begin with signed distance functions. Given a 2D shape, a signed distance function f measures the distance to the closest point on the shape's boundary contour. The distance function is negative inside the shape, positive outside, and zero on the contour. Thresholding these functions at zero reproduces the original contours, and it is easy to antialias edges by using a smooth transition function [Gupta and Sproull 1981; McNamara et al. 2000], or to support special effects like outlining or embossing.

Signed distance functions can be precomputed and sampled. Interpolation of these samples can reproduce linear edges at any orientation, but interpolation tends to round off corner features. To address this problem, distance fields can be adaptively sampled [Frisken et al. 2000]. This could be implemented on the GPU using multiresolution textures [Kraus and Ertl 2002], but would require an extra level of indirection and, to support hardware bilinear interpolation, redundant storage. It also does not completely solve the problem: at *some* magnification, the corners will still be rounded.

Instead, we compute the distance function procedurally and *exactly*, using geometric feature information rather than interpolation of distance samples. We use a Voronoi diagram to build an accelerator, so at every point we only have to consider the minimal number of features required to evaluate the distance field. We overlay a Voronoi diagram with a grid, and in each grid cell we make a list of the contour features that contribute to the distance field in that cell. The resolution of the grid is adapted to the complexity of each glyph, so we can use less storage for simple glyphs and more for complex glyphs. Our accelerator structure also takes advantage of spatial coherence to reduce redundancy by searching neighbouring texels for features. During preprocessing, an optimization process assigns features to texels in anticipation of this runtime local search.

Our second major challenge was to support dynamic layout of large numbers of glyphs. First, the feature data for a set of desired glyphs are packed into a "font" texture. Then, at every texel of a "sprite" texture we store the offset, scale, and extent of a glyph that covers that texel's cell. During rendering, we access (using dependent texturing) a set of features from the glyph referenced from the current cell and from any neighbours whose glyphs might overlap the current cell. We then compute the overall minimum distance function to these features. This approach requires only one level of texture indirection, is spatially coherent, and takes a constant amount of time per rendered pixel.

4 Antialiasing

One of the major advantages of an implicit representation of contours is that it can be antialiased easily. For an infinite straight edge, it has been shown that using the signed distance function (normalized plane equation) as the implicit representation and using a smooth transition function is equivalent to convolution with a radial filter [Gupta and Sproull 1981]. For corners, in order for a smooth thresholding to be equivalent to convolution, theoretically we have to do more work, using the distance to the two closest edges and a 2D lookup table. However, this is not necessary for good results [Gupta and Sproull 1981]: in practice simpler rules with approximately the right behaviour suffice. For example, we can use the minimum distance as the implicit representation of a corner [Turkowski 1982] (which is suggested by the usual implementation of CSG operations on implicit representations), or the thresholded edge functions can be combined with multiplication [McNamara et al. 2000]. The important thing about antialiasing is to bound the bandwidth of a signal *before* sampling it, not to simulate convolution exactly. It is also not necessary to use the distance function, a fact that can be exploited to simplify the representation or compute various special effects, such as mitering. Any implicit function with the same zero set can be used, although its gradient should be normalized to unity if we plan to control the width of the transition region [Loop and Blinn 2005].

The simplest smooth transition function is a clamped linear ramp, which in 1D would be equivalent to convolution by a box filter. A better choice is the smooth cubic ease curve given by the formula

$$\text{smoothstep}(g) \quad = \quad 1/2 + g_c(3/2 - 2g_c^2), \tag{1}$$

where g_c is g clamped to the interval $[-1,1]$. All figures in this paper use this cubic transition function.

Let f be a signed distance function. Relative to the texture coordinates (u,v), the gradient $\nabla f(u,v)$ of the signed distance function has magnitude 1. It points towards the closest point on the contour on the inside (negative distances), and away from the closest point on the outside (positive distances). For implicit representations that are not true distance functions, the gradient can be explicitly normalized to unit length.

Figure 2: *Isotropic filtering (left) vs. anisotropic filtering (right).*

The normalized texture-space gradient can be used to implement inexpensive anisotropic antialiasing. The texture-space gradient can be transformed into screen space coordinates (x,y) as follows:

$$\nabla f(x,y) \quad = \quad \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \end{bmatrix} \tag{2}$$

$$= \quad \begin{bmatrix} \partial u/\partial x & \partial v/\partial x \\ \partial u/\partial y & \partial v/\partial y \end{bmatrix} \begin{bmatrix} \partial f/\partial u \\ \partial f/\partial v \end{bmatrix} \tag{3}$$

$$= \quad \mathsf{J}_{x,y}(u,v)\nabla f(u,v). \tag{4}$$

where J is the Jacobian of the (possibly procedural and usually nonlinear) transformation from texture space to screen space. GPUs provide the ability to approximate these derivatives using differencing in fragment shaders, although they could also be computed exactly using automatic differentiation.

The gradient vector gives the direction of the maximum rate of change of an implicit function. We can therefore compute the width of the transition region based on the magnitude of ∇f in screen space, then reparameterize a smooth step function to get the desired final intensity value I,

$$s \quad = \quad \sqrt{(\partial f/\partial x)^2 + (\partial f/\partial y)^2}, \tag{5}$$

$$I \quad = \quad \text{smoothstep}(f/2sw), \tag{6}$$

where w is the desired width of the transition region in pixels.

By contrast, isotropic antialiasing, as is typically used in MIP-mapping [Williams 1983], computes the scale factor using

$$m_x \quad = \quad \sqrt{(\partial u/\partial x)^2 + (\partial v/\partial x)^2}, \tag{7}$$

$$m_y \quad = \quad \sqrt{(\partial u/\partial y)^2 + (\partial v/\partial y)^2}, \tag{8}$$

$$s \quad = \quad \max(m_x, m_y). \tag{9}$$

The isotropic computation of s estimates the width of a worst case filter. Isotropic antialiasing is often too conservative and can cause severe blurring when the transformation is anisotropic, for instance under perspective foreshortening for oblique views near silhouette edges, or for highly nonlinear texture deformations (see Figure 2). Isotropic antialiasing can severely degrade the readability of text in particular. Fortunately, when the gradient of the distance function is available, anisotropic antialiasing is inexpensive, and produces a filter of constant screen-space width.

5 Features and Distance Functions

Each contour in a glyph is a piecewise path. The pieces of the path are line, quadratic curve, and cubic curve segments. In this work, we consider primarily line segments. Quadratic and cubic curves are adaptively approximated by line segments. While we have some preliminary results based on computing distances to quadratic polynomial curves, and plan to follow up on this in future work, computation of distances to line segments is simpler and faster.

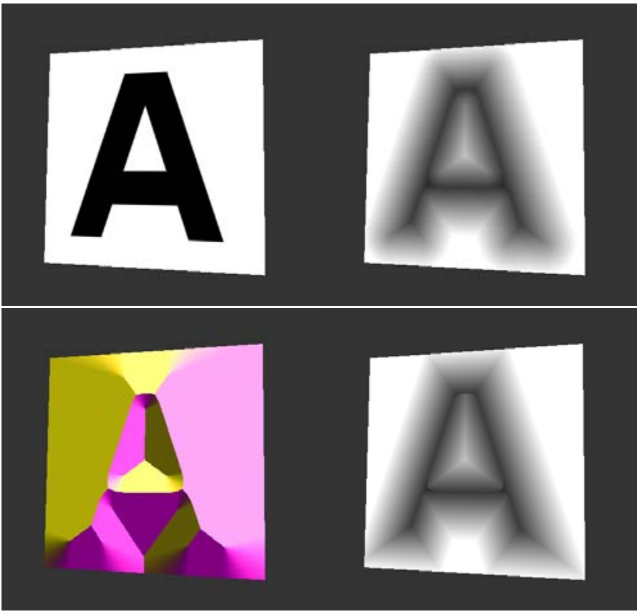

Figure 3: *A glyph with 11 line segments, its distance field, the gradient of its distance field, and its pseudodistance.*

To construct a signed distance function for a contour, we break it down into geometric *features*, in such a way that the distance function for the contour can be found as the distance to the closest feature. We primarily considered two features: individual line segments, and "corners" consisting of two half-segments meeting at a vertex.

For any feature, we want to compute not only the distance, but the gradient of the distance function, a sign to be used in an inside/outside test, and optionally a "pseudodistance" to be used for mitering (see Figures 3 and 6). The pseudodistance is the distance to the closest point on the infinite line containing the closest segment; the true distance takes the segment endpoints into account.

The choice of features and distance function evaluation techniques for them is a separate decision from the rest of our system. However the distance field is generated, it can be used for antialiasing and glyph placement as described later.

Any 2D curve can be described parametrically as a function $\mathbf{P} : \mathbb{R} \mapsto \mathbb{R}^2$. Given a test point $\mathbf{Q}$, we can solve for the parameter t^* that minimizes $|\mathbf{Q} - \mathbf{P}(t^*)|$. For a curve *segment*, we have to consider the endpoints as well. Without loss of generality, let the endpoints of a curve segment be $\mathbf{P}_0 = \mathbf{P}(0)$ and $\mathbf{P}_1 = \mathbf{P}(1)$. If $t^* \in [0,1]$, then the closest distance is given by $|\mathbf{Q} - \mathbf{P}(t^*)|$, otherwise it is given by $\min(|\mathbf{Q} - \mathbf{P}_0|, |\mathbf{Q} - \mathbf{P}_1|)$.

For line segments, by clamping the value of t^* to $[0,1]$ we can reconstruct the true distance function and gradients, taking endpoints into account. If we don't clamp, we compute the pseudodistance. It is also convenient to compute only the squares of the distances and compare these, taking a single square root of the minimum distance to all features under consideration. Likewise we can defer certain operations, such as normalization of the gradient vector, until after we have found the closest of a set of features.

5.1 Line Segments

Figure 3 shows a glyph represented with line segments and the various fields associated with it.

It is relatively inexpensive to compute the distance to line segment features. However, we also have to compute the sign of the distance to the contour, which depends on whether our test point is on the inside or the outside of the contour. Unfortunately, certain difficulties arise at corners.

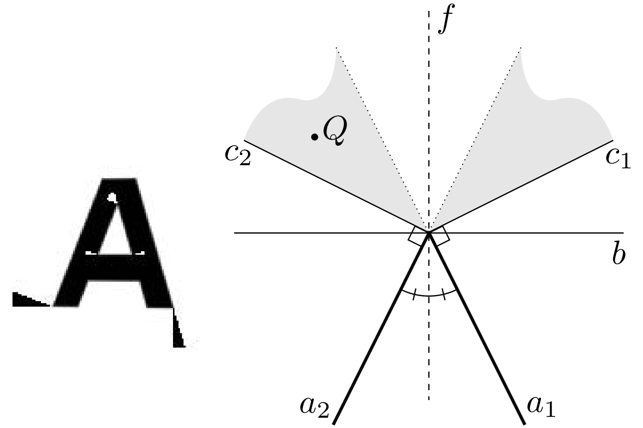

Figure 4: *Errors caused by vertex ambiguity, and the relevant regions and boundaries around a corner where two lines meet.*

A naive approach to computing the sign is to determine the closest line segment, and then use the sign of the plane equation of that segment. However, this approach may fail at corners since the distance to a shared vertex will be the same in the region bounded by lines perpendicular to each segment, shown as c_1 to c_2 in Figure 4. This may result in the wrong sign being computed in the shaded regions shown in that figure.

One inexpensive (but inelegant) solution to this problem is to "shrink" the line segments by a small amount, putting small gaps in the contour. In Figure 4, the gap induces a separating plane at f that bisects the corner. Unfortunately, if the gap is too large, it can introduce artifacts in the rendering, and if it is too small, the angle of the separating planes between the endpoints can be inaccurate (especially if the endpoints are stored at low precision). To get the correct bisection angle (important in mitering), the shrink distance has to be the same on both sides of a vertex. Care also has to be taken not to reverse or eliminate very short line segments, which often arise as the result of curve subdivision.

Another approach is to use the pseudodistance to break ties: above line b, the *maximum* absolute pseudodistance gives the *closest* line segment (consider that the pseudodistance is the distance to extensions of the segments a). Below c, the true distance gives the correct answer, and below b, the maximum absolute pseudodistance rule is wrong, so we need to *not* use it. To avoid pixel dropout, we need to switch between the two rules midway between b and c, where they are both correct. Using this disambiguating rule unfortunately results in a distance computation that requires about twice as many arithmetic operations as using shrunken line segments.

A line segment is described by its endpoints $\mathbf{P}_0$ and $\mathbf{P}_1$. Parametrically, points along the line can be generated by linear interpolation between these points:

$$\mathbf{P}(t) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1. \tag{10}$$

For points on the line segment, t must lie in the range $[0,1]$. Given a test point $\mathbf{Q} = (u,v)$, the parameter of the closest point on the (extrapolated) line can be found using the following computations:

$$\vec{\mathbf{d}} = \mathbf{P}_1 - \mathbf{P}_0, \tag{11}$$

$$\vec{\mathbf{q}} = \mathbf{Q} - \mathbf{P}_0, \tag{12}$$

$$t^* = \vec{\mathbf{d}} \cdot \vec{\mathbf{q}} / \vec{\mathbf{d}} \cdot \vec{\mathbf{d}}. \tag{13}$$

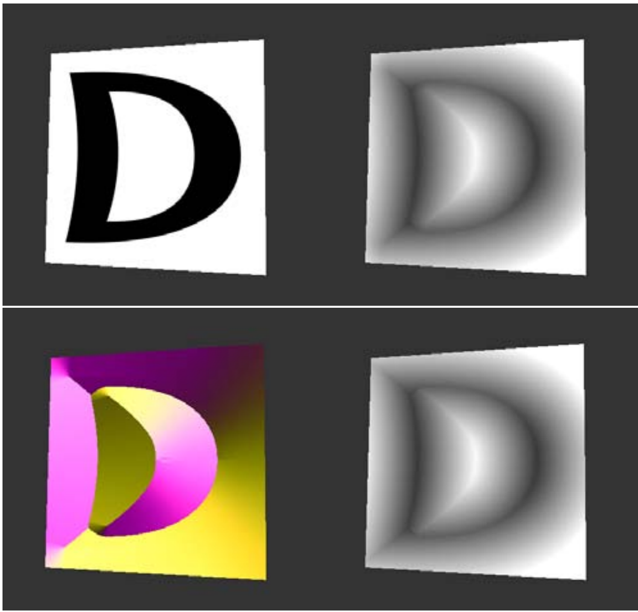

Figure 5: *A glyph with 6 quadratic segments, its distance field, the gradient of its distance field, and its pseudodistance.*

Note that no square root is required, only a division. However, t^* may not lie in the interval $[0,1]$, so a clamped value should be generated: $t_c^* = \min(1, \max(0, t^*))$.

Then, the squares of the distance g and pseudodistance h can be calculated as

$$g^2 = (\mathbf{Q} - \mathbf{P}(t_c^*)) \cdot (\mathbf{Q} - \mathbf{P}(t_c^*)) \quad (14)$$
$$h^2 = (\mathbf{Q} - \mathbf{P}(t^*)) \cdot (\mathbf{Q} - \mathbf{P}(t^*)) \quad (15)$$

To compute the sign (that is, determine which side of the line the point $\mathbf{Q}$ is on), we can take the dot product of $\vec{\mathbf{q}}$ with the normal:

$$\vec{\mathbf{n}} = (d_y, -d_x), \quad (16)$$
$$s = \text{sign}(\vec{\mathbf{n}} \cdot \vec{\mathbf{q}}), \quad (17)$$

where

$$\text{sign}(a) = \begin{cases} -1 & : \ a < 0 \\ 0 & : \ a = 0 \\ 1 & : \ a > 0 \end{cases} \quad (18)$$

Note that we do not have to normalize $\vec{\mathbf{n}}$ to unit length to get the correct sign. However, if we are willing to do so, an alternative way to compute the pseudodistance along with the sign is

$$\hat{\mathbf{n}} = \vec{\mathbf{n}}/|\vec{\mathbf{n}}|, \quad (19)$$
$$h = \vec{\mathbf{q}} \cdot \hat{\mathbf{n}}. \quad (20)$$

All the dot products in these computations are on two-tuples, but GPUs use four-tuple registers. However, we often compute distances to several line segments at once before comparing their magnitudes. A useful optimization in practice is to use "vertical" as well as "horizontal" SIMD computations. For instance, we can use four-tuple operations to compute two two-tuple operations in parallel, or four scalar operations to compute the distance to four line segments in parallel. Some GPUs can also co-issue two two-tuple instructions in a single cycle.

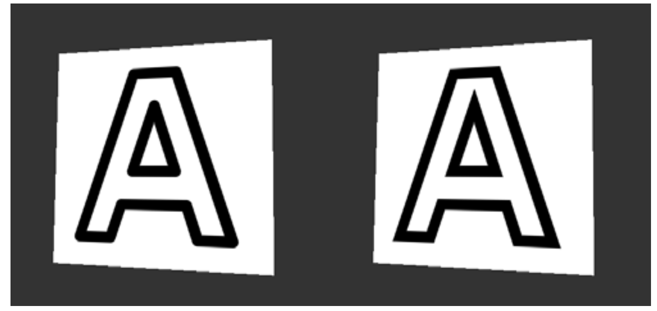

Figure 6: *Using a smooth pulse rather than a smooth step gives antialiased outlines. Using the true distance gives rounded outlines; using the pseudodistance gives sharp miters.*

5.2 Corners

Corners are pairs of line segments meeting at a common point. A closed contour expressed as a sequence of N line segments can also be expressed as a sequence of N corners by dividing all line segments at their midpoints. Although it is slightly more expensive to compute the distance to a corner, and corners take more parameters to represent, corners do not suffer from ambiguity about which one is closer, since they always meet with derivative continuity.

Corners are specified with three points: endpoints $\mathbf{P}_0$ and $\mathbf{P}_2$ and vertex $\mathbf{P}_1$. We compute the direction tangents, then use these to compute the normal of the bisecting line:

$$\vec{\mathbf{d}}_0 = \mathbf{P}_0 - \mathbf{P}_1, \quad (21)$$
$$\vec{\mathbf{n}}_0 = (d_{y,0}, -d_{x,0}), \quad (22)$$
$$\hat{\mathbf{n}}_0 = \vec{\mathbf{n}}_0/|\vec{\mathbf{n}}_0|, \quad (23)$$
$$\vec{\mathbf{d}}_1 = \mathbf{P}_2 - \mathbf{P}_1, \quad (24)$$
$$\vec{\mathbf{n}}_1 = (d_{y,1}, -d_{x,1}), \quad (25)$$
$$\hat{\mathbf{n}}_1 = \vec{\mathbf{n}}_1/|\vec{\mathbf{n}}_1|, \quad (26)$$
$$\vec{\mathbf{d}} = \hat{\mathbf{n}}_0 + \hat{\mathbf{n}}_1, \quad (27)$$
$$\vec{\mathbf{n}} = (d_y, -d_x). \quad (28)$$

We make both tangent vectors point away from the corner point. Averaging the perpendiculars of the direction tangents rather than the tangent vectors themselves avoids a degeneracy when the corner's vertices are colinear. Two normalizations are required to get the actual bisector. Once we have the normal of the bisector, though, it does not have to be normalized.

Then, we compute a vector from the center vertex of the corner to the test point $\mathbf{Q} = (u, v)$, and test this against the bisector normal:

$$\vec{\mathbf{q}} = \mathbf{Q} - \mathbf{P}_1, \quad (29)$$
$$s = \vec{\mathbf{q}} \cdot \vec{\mathbf{n}}. \quad (30)$$

If s is positive, then we compute the distance to the line segment given by $\mathbf{P}_0$ and $\mathbf{P}_1$, otherwise we compute the distance to the line segment given by $\mathbf{P}_1$ and $\mathbf{P}_2$, reusing the vector $\vec{\mathbf{q}}$. If $\vec{\mathbf{n}}$ is precomputed, the extra cost of computing the distance to a corner relative to a line segment is one dot product and a conditional assignment—plus the cost of storing the additional center point and the precomputed bisection normal. A corner feature therefore requires between 1.5 and 2 times as many stored values as a line segment feature. However, since there is no ambiguity problem, we can potentially use lower precision for storage compared to shrunken line segments, which can result in an equivalent storage cost.

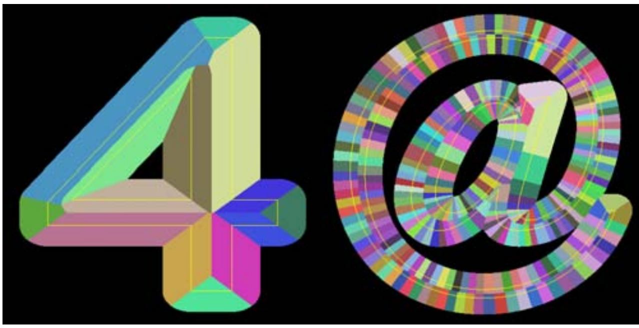

Figure 7: *Voronoi analysis of some TrueType glyphs.*

5.3 Quadratics

For higher quality, we can consider features based directly on polynomial curves. Glyphs in standard font formats use both quadratic and cubic segments. We only considered distances to quadratic segments in our preliminary research; a test glyph is shown in Figure 5. Unfortunately, solving for the nearest distance to a quadratic polynomial curve requires finding the roots of a cubic polynomial. We have found that it is most convenient to do this with an iterative root solver, since the analytic solution to a cubic equation requires cube roots (which, on GPUs, would be solved with *two* iterative solvers for logarithms and exponentiation anyway) and a case analysis (which we can avoid with suitable starting conditions on our own iterative solver). Quadratic splines, like corners, require a minimum of six coordinates to specify. It is also possible to speed up their computation by precomputation, but then twelve numbers are required for every quadratic, which requires three texture elements to store.

Quadratics suffer from the same problem as line segments: distances to endpoints can be ambiguous. As with line segments, either endpoint shrinking or midpoint subdivision and grouping into corners can be used to resolve this.

5.4 Performance

At the resolution of the results in this paper, roughly 512×512, and on our test machine (Pentium 4 2.6GHz and an NVIDIA 7800GT GPU), a glyph with 11 line segments (the A glyph shown in the figures) runs at 150fps without explicit disambiguation (using shrunken line segments) and 80fps with pseudodistance disambiguation. Using 11 corners and precomputed separation planes, the same glyph runs at 60fps. Surprisingly, the 6 quadratic polynomial features used in the curved test glyph (the D glyph shown in the figure) runs at 70fps at the same resolution. In practice we found the quality of representations computed with linear features to be adequate. For the rest of the results in this paper, we used only line segment features and the "shrinking" approach for disambiguation.

6 Voronoi Grid Accelerator

The test renderings and performance numbers given so far were generated using a brute-force approach, comparing the distances to all features in all contours of a glyph. This is not practical for real glyphs, and certainly not for a page of text. We therefore have to accelerate the computation by limiting the number of features considered at each texel.

6.1 Voronoi Analysis

In order to build an accelerated representation of the glyphs in a font, we read in the contour information using the FreeType library. We adaptively approximate quadratics and cubics by line segments, subdividing the splines recursively until a given error bound is met. We then compute the Voronoi diagram of each glyph using hardware acceleration [Hoff III et al. 1999]. The hardware acceleration approach to drawing a Voronoi diagram can also suffer from endpoint ambiguity, so we shrink endpoints to resolve it (this problem would not arise if corners were used as features). Some example analyses are shown in Figure 7. We compute the Voronoi diagram only within some finite distance d of the contour. We also stretch each character non-uniformly to fit a square with a minimum margin of d. The Voronoi cones we draw are non-uniformly scaled so the distance computation is still correct even though nonuniform scales are non-Euclidean: we get a scaled image of the Voronoi diagram of the unscaled contours. It would also be possible to use a brute-force shader computation to compute this diagram, to use a geometric (CPU only) Voronoi analysis, or to compute the diagram for corner or polynomial features.

6.2 Grid Packing

Once the Voronoi diagram is computed, it is overlaid with a regular grid. Within each cell of the grid, we make a list of all the features associated with regions in that cell. These are the only features which need be considered when computing the minimum distance to any point in this cell. Also, there are cells that are more than d from any contour. These cells are black in the diagram, but we perform an additional test on the CPU to determine if they are completely inside or completely outside. In a `flag` texture, we store (in a suitably biased fashion) -1 for cells completely inside, 0 for cells on the boundary, and 1 for cells completely outside. Then, in a `feature` texture, we store the parameters of features. These can be clipped and quantized, since we don't care about features or parts of features more than distance d away. Note that only one feature can fit in each cell of the `feature` texture. However, adjacent cells often refer to the same features. In our shader, we will look at not only the features stored in the cell containing the test point, *but also at a number of neighbours* (four is reasonably fast, but nine can also be used). We use a simulated annealing process to assign features to texels of the `feature` texture so that all the required features for each cell will be accessed by the shader. Note that it is acceptable for a feature to be accessed and evaluated even if it is not necessary, a fact we exploit to avoid conditionals. The `flag` texture marks texels that are completely inside or outside, and more than distance d from the boundary. We multiply the value in `flag` by a large number and add it to the computed minimum distance. Thus, we can store "extra" features in these texels as well. The distance computation from these cells will be incorrect, but we are going to swamp it with a value of the correct sign (and then clean up the distance field with a clamp), so it doesn't matter. This provides extra storage for features that won't fit near a contour. We also make sure that the border of the glyph is bounded by cells marked as being outside the contour. At a resolution of approximately 512×512, using four features per sample, a single magnified glyph renders at about 100fps on our test machine.

6.3 Multiresolution

It is possible that the features for complex glyphs cannot be "packed" at a given resolution. We therefore start at the lowest possible resolution and, if we cannot successfully pack the features at that resolution, increase the resolution of the accelerator in power-of-two steps until we find a packing. Due to subdivision, curved edges can result in a large number of features, whereas glyphs with

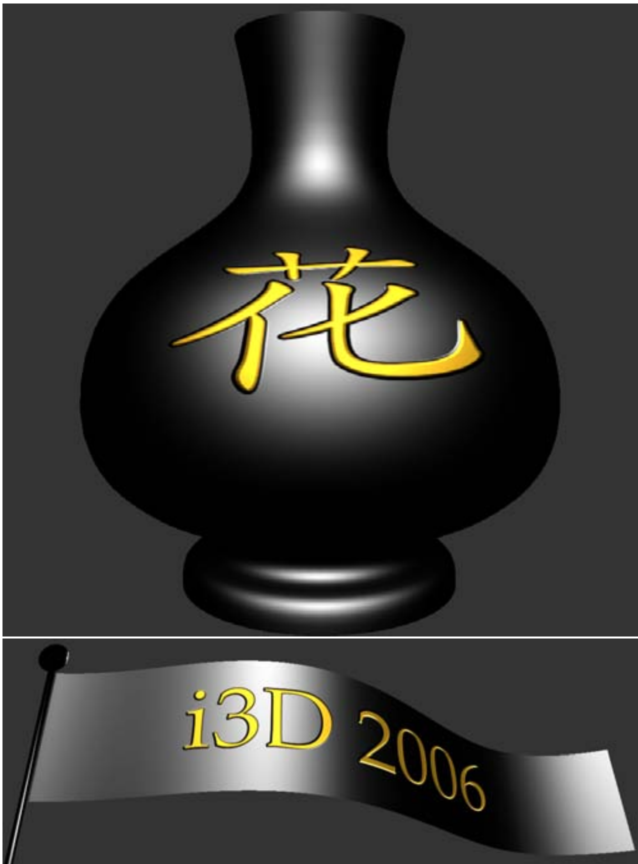

Figure 8: *Some examples showing embossed glyphs. The gradient of the distance field is used to perturb the normal.*

straight edges can be stored at a relatively low resolution. If we wanted to use curved features, the process would be the same but the accelerator resolution required would be smaller, since there would be a smaller number of features required for similar quality.

Different glyphs may now require accelerator structures of different resolutions. We pack glyphs together into a quadtree, taking note of the scale factor and offset of each glyph. This latter information is not stored in a texture; rather, it is retained in a CPU data structure and used later by the CPU when placing sprites, as discussed in the next section.

The analysis process takes about one second per glyph. This is not fast enough to be done in real time, but is fast enough that a font subset can be computed for individual textures during a preprocessed encoding of a given vector texture.

It is possible but rare that the analysis process will fail if the Voronoi diagram contains a vertex of high outdegree, where more regions are adjacent than the number of features accessed per pixel. In this case, we can

1. consider a larger number of features per pixel (globally);
2. accept an approximation of the distance field around that vertex, since they are usually far from the contour; or
3. on newer GPUs, use conditional execution to consider more features only around this point.

Use of conditional execution maintains high performance on average since extra computation will only be required in small regions.

7 Glyph Sprite Mapping

The output of the previous step is a multiresolution font table, which is visualized on the right of Figure 9. Now we wish to instance multiple glyphs on a document. We use a sprite table for this (Figure 9, left). The size of a cell in a sprite table must be less than the minimum distance possible between glyphs. For text, this can be computed from the advance distance for each glyph and the kerning information containined in the font file, as well as the desired minimum text size.

Figure 9: *Sprite and font tables.*

In each sprite cell, we store the 2D offset of a sprite that (partially) covers that cell, the 2D location of the origin of that sprite in the font table, the two factors by which we wish to scale the glyph in the vertical and horizontal directions (the ratio of these should be calculated to return a glyph to its original aspect ratio so that distance calculations are correct), and the total scale factor of the glyph relative to its absolute size in a canonical coordinate system (so we can correct the distance computations for each glyph and make them comparable). We must store this information in every cell covered by a sprite's bounding box, except for the cells on the right that are only partially covered. Two textures of four components each are required.

Now, to sample this "virtual" texture, we look up the sprite information in the cell containing the sample point *and the cell to the immediate left* of that cell. For each of pair of sprites, we access the appropriate set of features in the font table and compute the minimum distance to them, then compute the minimum distance of these two distances. By clamping the offsets in the font table to normalized coordinates for each cell, we can avoid picking up features from adjacent glyphs. The boundary of empty cells around each glyph in the font table gives us white space around each glyph. Sprite table cells that are not used must refer to the "blank" glyph in the font.

This process lets two sprites overlap horizontally. In our implementation, we consider four features per glyph, requiring the computation of eight point to feature distances. Example renderings are shown in Figure 1; performance of this scene at a resolution of 512×512 is about 60fps on our test machine.

If we look at four sprites at once, one to the right, one below, and one to the right and below, and omit the top row of partially covered cells when placing sprites, we can also handle vertical overlap. However, this is not (usually) needed in normal text rendering, and doubles the cost. Likewise, we could add extra information to the sprites, such as rotations, to handle more general cases. The ability to rotate glyphs and permit four-way overlap might be useful in the rendering of map labels, for instance.

Figure 10: *A map using a vector texture for labels, with an enlargement of a distorted region to demonstrate anisotropic antialiasing.*

8 Conclusions

We have presented a technique for rendering outline font glyphs directly on the GPU as encapsulated procedural textures, and for placing multiple instances of these glyphs anywhere in a texture, subject to some constraints on overlap. Our approach is based on *exact* computation of distance fields and supports efficient anisotropic antialiasing. We have introduced a new representation for vector graphics in general, in which features of a vector graphics image are analysed using a Voronoi diagram and then packed into a grid under the assumption of lookup over a neighbourhood to avoid redundancy. Our approach can be thought of as a generalization of texture sprites to implicit function generators.

For many purposes, font glyphs can be prerendered and placed in a texture map. Even then, the sprite approach may be useful for rendering pages of text using compositing rather than minimum distance computations. It is also possible to prerender sampled distance fields, which would still support efficient anisotropic antialiasing. However, the exact procedural approach provides the assurance that no matter what 3D distortion or magnification is applied, the edges will always be crisp and the corners sharp. Since our representation gives the exact distance field rather than an approximation, applications such as outlining and embossing, which rely on an accurate computation of the distance field far from the contour, are also possible.

We have demonstrated that the storage and computational costs of vector textures are feasible. In fact, at high magnifications the bandwidth requirements of vector textures are lower than raster images of equivalent resolution would be, although of course the computational requirements are higher. However, since GPU computational performance is scaling much faster than bandwidth, ultimately image representations using more computation but less bandwidth will prove superior.

References

CROW, F. C. 1977. The Aliasing Problem in Computer-Generated Shaded Images. *Communications of the ACM 20*, 11, 799–805.

CROW, F. C. 1978. The Use of Grayscale for Improved Raster Display of Vectors and Characters. *SIGGRAPH*, 1–5.

FEIBUSH, E. A., LEVOY, M., AND COOK, R. L. 1980. Synthetic texturing using digital filters. vol. 14, 294–301.

FRISKEN, S., AND PERRY, R. 2004. Method for generating an adaptively sampled distance field of an object with specialized cells. *U.S. Patent Application 20040189642*.

FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., AND JONES, T. R. 2000. Adaptively sampled distance fields: a general representation of shape for computer graphics. *SIGGRAPH* (July), 249–254.

FRISKEN, S., PERRY, R., AND JONES, T. 2005. Detail-directed hierarchical distance fields. *U.S. Patent 6,396,492* (July 12).

GLANVILLE, R. S. 2004. Texture Bombing. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, Addison-Wesley, F. Randima, Ed.

GUPTA, S., AND SPROULL, R. F. 1981. Filtering edges for grayscale displays. In *SIGGRAPH*, 1–5.

HOFF III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast Computation of Generalized Voronoi Diagrams using Graphics Hardware. In *SIGGRAPH*, 277–286.

KRAUS, M., AND ERTL, T. 2002. Adaptive Texture Maps. In *Proc. Graphics Hardware*, 7–15.

LEFEBVRE, S., AND NEYRET, F. 2003. Pattern Based Procedural Textures. In *ACM Symposium on Interactive 3D Graphics*.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2005. Texture Sprites: Texture Elements Splatted on Surfaces. In *ACM Symposium on Interactive 3D Graphics*.

LOOP, C., AND BLINN, J. 2005. Resolution independent curve rendering using programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2005*, 1000–1010.

MCCOOL, M., AND DU TOIT, S. 2004. *Metaprogramming GPUs with Sh*. AK Peters.

MCCOOL, M., DU TOIT, S., POPA, T., CHAN, B., AND MOULE, K. 2004. Shader Algebra. In *ACM Trans. on Graphics (Proc. SIGGRAPH)*, vol. 23, 787–795.

MCNAMARA, R., MCCORMACK, J., AND JOUPPI, N. P. 2000. Prefiltered Antialiased Lines using Half-Plane Distance Functions. In *SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 77–85.

PERRY, R., AND FRISKEN, S. 2005. Method and apparatus for rendering cell-based distance fields using texture mapping. *U.S. Patent 6,917,369* (July 12).

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

SEN, P., CAMMARANO, M., AND HANRAHAN, P. 2003. Shadow silhouette maps. *ACM Trans. on Graphics (Proc. SIGGRAPH) 22*, 3 (July), 521–526.

SEN, P. 2004. Silhouette maps for improved texture magnification. In *Proc. Graphics Hardware*, 65–74,147.

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

TURKOWSKI, K. 1982. Anti-aliasing through the use of coordinate transformations. *ACM Trans. on Graphics 1*, 3 (July), 215–234.

WANG, S. W., AND KAUFMAN, A. E. 1996. Volume sampled voxelization of geometric primitives. In *IEEE Visualization 96*, 78–84.

WARNOCK, J. E. 1980. The Display of Characters Using Gray Level Sample Arrays. 302–307.

WILLIAMS, L. 1983. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, 1–11.