Resolution Independent Curve Rendering using Programmable Graphics Hardware

Charles Loop[*]
Microsoft Research

Jim Blinn[†]
Microsoft Research

Abstract

We present a method for resolution independent rendering of paths and bounded regions, defined by quadratic and cubic spline curves, that leverages the parallelism of programmable graphics hardware to achieve high performance. A simple implicit equation for a parametric curve is found in a space that can be thought of as an analog to texture space. The image of a curve's Bézier control points are found in this space and assigned to the control points as texture coordinates. When the triangle(s) corresponding to the Bézier curve control hull are rendered, a pixel shader program evaluates the implicit equation for a pixel's interpolated texture coordinates to determine an inside/outside test for the curve. We extend our technique to handle anti-aliasing of boundaries. We also construct a vector image from mosaics of triangulated Bézier control points and show how to deform such images to create resolution independent texture on three dimensional objects.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation;

Keywords: curve rendering, resolution independence, vector representations, graphics hardware algorithms

1 Introduction

The main ideas of this paper are extensions of our solution to the following simple problem. Suppose we want to render a triangle, whose vertices are the control points of a quadratic Bézier curve, such that the parts *inside* and *outside* the curve are shaded differently. Furthermore, we assume that our triangle might be embedded in a three dimensional space and viewed in perspective.

One solution might be to densely sample the curve and form many more smaller triangles and shade the inside and outside triangles accordingly. Alternatively, we might create a texture image of the untransformed triangle and mark texels as inside or outside, and then render the triangle with this texture. While both of these approaches will work, they are both plagued by sampling artifacts. If we zoom in on the triangle using the sampled curve, we will see the facets of the piecewise linear approximation to the curve. If we zoom in on the textured triangle, we will see the texels of the underlying texture image.

---
[*]e-mail: cloop@microsoft.com
[†]e-mail: blinn@microsoft.com

If we had an implicit curve, we could solve the problem by transforming this curve to screen space and evaluating the new implicit curve at pixel locations. Values less than zero are inside, values greater than or equal to zero are outside (by arbitrary choice). With a bit of computation one can readily *implicitize* a parametric curve. However, the screen space projection of the curve, and hence its implicit equation, might change at every frame. Finding the implicit form of the curve every frame could be fairly expensive.

Fortunately, there is a better way. We observe that the implicit form of any rational parametric quadratic curve is a conic section; and that any conic section is the projected image of single canonical parabola. This leads to a simple solution to our original problem.

Suppose the vertices of our triangle are the quadratic Bézier control points $\mathbf{b}_0$, $\mathbf{b}_1$, and $\mathbf{b}_2$. We associate with these vertices the $[u\ v]$ texture coordinates $[0\ 0]$, $[\frac{1}{2}\ 0]$ and $[1\ 1]$, see Figure 1. During rasterization, the Graphics Processing Unit (GPU) will calculate a texture coordinate for each pixel on the interior of the triangle by interpolating the texture coordinates of the triangle vertices. We determine if the pixel is inside or outside the curve by evaluating

$$f(u,v) = u^2 - v$$

in a pixel shader program. If $f(u,v) < 0$ then the pixel is inside the curve, otherwise it is outside.
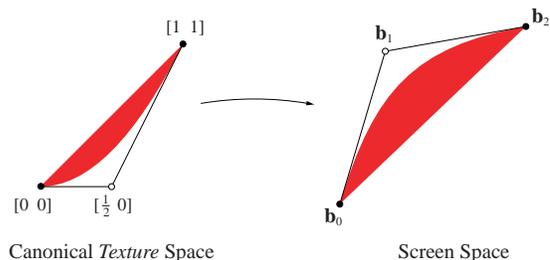
Figure 1: The canonical quadratic curve element (left), a triangle formed by the control points of a quadratic Bézier curve (right).

We are interested in rendering more than a single triangle that contains a quadratic curve. Our goal is to construct mosaics of triangles that contain quadratic and cubic curves. Our result is a mechanism for rendering vector geometry that has the following properties:

- *resolution independence*,

- *compact geometric representation*,

- *high performance*.

Resolution independence means that the curved elements of a vector based image are always curved, independent of viewpoint. Our representation consists of a collection of triangles that is proportional to the design time complexity of the vector image; this is often much smaller than a corresponding raster image of comparable quality. Our scheme is fast since our shader programs are small and

run in parallel on programmable graphics hardware with multiple pixel pipelines.

Previous work on implicit curve rendering concentrates on interval based searches for pixels containing the curve, and on the topology of extreme points [Arnon 1983; Taubin 1994; Tupper 2001]. The rendering of ellipsoids using a GPU has been considered in [Gumhold 2003] where the algebraic form is used to determine if a pixel-ray intersects the surface. Embedding sharp linear features into images to obtain resolution independence while leveraging GPU pixel processing has been done in [Tumblin and Choudhury 2004] and [Sen 2004]. Curved elements have been embedded into texture images at the texel level in [Ramanarayanan et al. 2004]. However, the curves are parametric, requiring a complex winding rule test (performed on the CPU) to determine if a pixel is inside or outside of a closed region. We consider a new class of algorithms that are made practical by the speed of modern GPU's.

This paper is organized as follows. Basic properties of programmable GPUs, and parametric and implicit curves are covered in Section 2. In Section 3, we present our algorithm for rendering quadratic curves, and applied it to the problem of font rendering. The more complex cubic case is considered in Section 4. The important issue of anti-aliasing is addressed in Section 5, and we show how to deal with degenerate viewpoints in Section 6. We discuss aspects of this work in Section 7, and make concluding remarks in Section 8.

2 Preliminaries

2.1 Programmable Graphics Hardware

Graphics hardware has evolved from a *fixed function* to a *programmable* pipeline in recent years. The programmable pipeline is based on vertex and pixel shaders. A vertex shader program executes on each vertex of a graphics primitive, while a pixel shader program executes on every pixel of a rasterized triangle. The data encapsulated in a vertex is a user defined collection of floating point numbers, much like a C struct. The vertex shader program can modify this, or invent new data, and pass the result along to a pixel shader. The input to a pixel shader is an interpolation of the vertex data on the vertices of a triangle. This interpolation is non-linear, involving the projective transform that maps a triangle from model to screen space. The pixel shader can output a color value that is written to the frame buffer.

2.2 Parametric Curves

We assume familiarity with basic concepts of affine and projective geometry. We work in projective 2D space where points are represented by a homogeneous 3-tuple $[x \ y \ w]$; and the position of a point in the plane is $[x/w \ y/w]$.

A parametric curve is a vector valued function of a single variable. Points on the curve are found by sampling the function at parameter values t. We write a rational parametric curve of degree n as the product

$$C(t) = \mathbf{t} \cdot \mathbf{C},$$

where

$$\mathbf{t} = [\, 1 \quad t \quad \cdots \quad t^n \,], \quad \text{and} \quad \mathbf{C} = \begin{bmatrix} x_0 & y_0 & w_0 \\ x_1 & y_1 & w_1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & w_n \end{bmatrix}.$$

The vector $\mathbf{t}$ contains *power basis functions* and $\mathbf{C}$ is the coefficient matrix that determines the shape of the curve. The *rational* curve $C(t)$ has components $[x(t) \ y(t) \ w(t)]$. In the special case where $w(t) = 1$, we refer to $C(t)$ as an *integral* curve. Commonly, the parameter t is restricted to the interval $[0,1]$ and we think of $C(t)$ as defining a curve *segment*.

Parametric curves may be represented in any linearly independent basis. We will also represent curves in terms of the Bernstein (a.k.a Bézier) basis

$$B(t) = [\, B_0^n(t) \quad B_1^n(t) \quad \cdots \quad B_n^n(t) \,] \cdot \mathbf{B},$$

where the Bernstein basis functions are defined as

$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}\, t^i, \quad \text{and} \quad \mathbf{B} = [\, \mathbf{b}_0 \quad \mathbf{b}_1 \quad \cdots \quad \mathbf{b}_n \,]^T,$$

is an $n \times 3$ matrix of Bézier control points $\mathbf{b}_i$. Changing from power basis to Bernstein basis is an invertible linear operation implemented as a multiplication of the coefficient matrix by an $n \times n$ change of basis matrix. For quadratics and cubics, these matrices are

$$\mathbf{M}_2 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 2 & 0 \\ 1 & -2 & 1 \end{bmatrix}, \qquad \mathbf{M}_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & \frac{1}{2} & 0 \\ 1 & 1 & 1 \end{bmatrix},$$

$$\mathbf{M}_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}, \quad \mathbf{M}_3^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & \frac{1}{3} & 0 & 0 \\ 1 & \frac{2}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}.$$

Therefore, the power basis coefficients of a Bézier curve are

$$\mathbf{C} = \mathbf{M}_i \cdot \mathbf{B}.$$

2.3 Implicit Curves

An implicit curve is the zero set of a function of two variables

$$c(x,y) = 0.$$

That is, the set of points $[x \ y]$ in the plane where c evaluates to zero. The relationship between implicit and parametric forms was the topic of Sederberg's Ph.D. thesis [Sederberg 1983]. He points out that many of the tools needed to convert between parametric and implicit forms have been known from elimination theory for well over a century. Adopting the notation $c^n(\cdot)$ to mean a polynomial of maximum degree n, we repeat the result noted by Sederberg:

Any curve which is defined parametrically by the equation

$$x = \frac{x^n(t)}{w^n(t)}, \quad y = \frac{y^n(t)}{w^n(t)}$$

will have an implicit equation of the form

$$c^n(x,y) = 0.$$

It is worth noting that the reverse is not true in general; that is, not all implicit curves can be parameterized by rational polynomials. For our purposes (assuming no degree lowering degeneracies) quadratic parametric curves will have degree 2 implicit equations (conic sections); and cubic parametric curves will have degree 3 implicit equations. Note that many parametric curves might have the same implicit form. This is because parametric curves represent a *particular* parametrization of the underlying algebraic curve. A given algebraic curve may have infinitely many parameterizations.

3 Rendering Quadratic Curves

After noting the basic fact about quadratic curves that makes our algorithm possible, we apply the technique to the problem of rendering two dimensional text embedded in a three dimensional space.

Restating the claim made in the introduction, we have

Claim 1 *Any rational quadratic parametric curve has an implicit form that is a projected image of the algebraic curve*

$$f(u,v) = u^2 - v. \tag{1}$$

Proof : Begin with an arbitrary rational quadratic parametric curve

$$C(t) = \mathbf{t} \cdot \mathbf{C}$$

where

$$\mathbf{t} = \begin{bmatrix} 1 & t & t^2 \end{bmatrix}, \quad \text{and} \quad \mathbf{C} = \begin{bmatrix} x_0 & y_0 & w_0 \\ x_1 & y_1 & w_1 \\ x_2 & y_2 & w_2 \end{bmatrix}.$$

Compare this to the curve

$$F(t) = \mathbf{t} \cdot \mathbf{F} = \begin{bmatrix} t & t^2 & 1 \end{bmatrix}, \quad \text{where} \quad \mathbf{F} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

The curve F has $u(t) = t$ and $v(t) = t^2$ and so has the desired implicit equation

$$f(u,v) = u^2 - v = 0.$$

We seek a transform Ψ such that

$$\mathbf{C} = \mathbf{F} \cdot \Psi^{-1}.$$

Clearly $\Psi^{-1} = \mathbf{F}^{-1} \cdot \mathbf{C}$, so we may write $C(t) = \mathbf{t} \cdot (\mathbf{F} \cdot \Psi^{-1})$. That is, $C(t)$ belongs to the transformed image of the curve $f(u,v)$. $\square$

We assign the Bézier control points of our canonical curve as texture coordinates on the vertices of a triangle that correspond to the control points of an arbitrary quadratic curve. These are found by applying the change of basis matrix to the coefficients of the canonical curve

$$\mathbf{M}_2^{-1} \cdot \mathbf{F} = \begin{bmatrix} 0 & 0 & 1 \\ \frac{1}{2} & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

We can ignore the constant $w = 1$ component and use $[0\ 0], [\frac{1}{2}\ 0]$, and $[1\ 1]$ as $[u\ v]$ texture coordinates corresponding to $\mathbf{b}_0$, $\mathbf{b}_1$, and $\mathbf{b}_2$ of any triangle containing a quadratic curve. We render this triangle using a pixel shader program that evaluates Equation (1).

The color of a pixel depends on the sign of $f(u,v)$. This implicit curve will partition the plane into two sets. In texture coordinates, points above the parabola will be associated with negative values of $f(u,v)$, and below with positive values of $f(u,v)$. We arbitrarily choose to associate the *inside* with negative sign. This implies that all triangles containing curves will treat the convex region of the curve as inside. However, we want to have shapes containing both convex and concave curve pieces, see Figure 2.

There are several ways to handle this. Two different shaders could be used, one that evaluates $u^2 - v$ and another that evaluates $v - u^2$; however, shader context switching can hurt performance. In practice, we add an extra floating point number with value ± 1 to each triangle vertex; this is used to change the sign of $f(u,v)$ to get both shapes.

We now apply our technique to the problem of rendering arbitrarily projected 2D text.

Figure 2: The quadratic curve contained in a triangle can have two possible orientations: a) convex, and b) concave orientation.

3.1 Font Rendering

Our simple pixel shader for rendering triangles that contain quadratic curves is sufficient for rendering TrueType font outlines, see Figure 3. Each character, or *glyph*, outline consists of a set of contours, corresponding to closed curves. Each curve is defined by an ordered set of *on-curve* and *off-curve* control points. The on-curve points are used to create straight edges and discontinuities. The off-curve points are control points of quadratic B-spline curves. By convention, the region of the glyph to the right of each oriented outline is considered to be on the inside of the shape.

First, we need to convert the B-spline curves into Bézier form by inserting a new point between each pair of adjacent off-curve points. These newly inserted points will lie on the curve and can be thought of as implied on-curve points. Once these points have been inserted into the boundary, every off-curve point will correspond to the *middle* point of a quadratic Bézier control point triangle $\mathbf{b}_0 \mathbf{b}_1 \mathbf{b}_2$.

Before proceeding we must check for, and remove, triangle overlaps. Although somewhat rare, this is done to avoid overdraw and possible visual artifacts. If an overlap is detected, we subdivide the triangle with the larger area and repeat until no more overlaps exist, see Figure 4. This process will terminate provided the original boundary curves do not (self) intersect or osculate. Dealing with these conditions is future work.
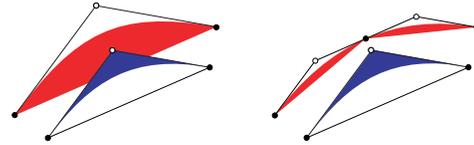
Figure 4: To avoid overdraw and visual artifacts, we do not allow overlapping triangles. We remove overlaps by subdividing the triangle with the larger area.

Next, we apply a constrained Delaunay triangulation to the control points, where we require each contour edge and triangle containing a Bézier curve to be preserved in the triangulation. After removal of those triangles not on the inside of the shape, we are left with three types of triangles. Interior triangles that do not contain a curve, and both orientations of triangles that contain a quadratic curve. We use our quadratic curve shader program to render the triangles containing a curve; the triangles that do not contain curves may be rasterized in a conventional way, or given texture coordinates that ensure the triangle is entirely inside (e.g. $[0\ 1], [0\ 1], [0\ 1]$).

The overlap removal and triangulation is a one-time preprocess that takes place on the CPU; the number of triangles is proportional to the complexity of the font outline and remains fixed. The triangulated font outlines could conceivably be stored on disk for reuse without any need for the tessellation framework. Once the triangles have been uploaded to GPU memory, they may be transformed by an arbitrary projective transform to yield a rendered image that is resolution independent.
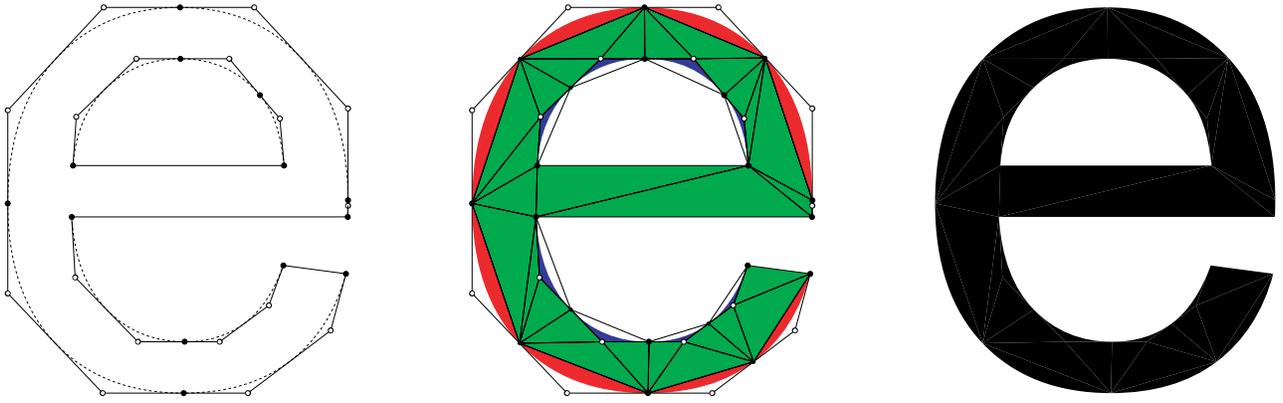
Figure 3: A two contour TrueType font outline on the left; filled dots represent on-curve points, hollow dots represent off-curve points. The outline is triangulated together with implied *on-curve* points as shown in the middle. The green triangles are interior to the shape and are entirely filled. The red (convex) and blue (concave) curves within triangles are rendered using our pixel shader program. The resulting shape on the right, can be arbitrarily transformed projectively and remains resolution independent.

3.2 Rendering Rational Quadratic Curves

While integral quadratic curves are sufficient for many applications, such as font rendering, it is sometime useful to be able to render rational quadratic curves. The algorithm presented Section 3 correctly renders integral quadratic curves under projective transformations, such that the image curves are rational, but the underlying curves in design space are not rational.

We can extend our rendering technique to handle the more general class of quadratic curves that are rational in design space. We can write an implicit quadratic curve as

$$c(x, y, w) = k^2 - lm \qquad (2)$$

where l, and m are the homogeneous equations of any two lines tangent to the curve, and k is the line connecting these points of tangency. More specifically, $k = ax + by + cw$ for some a, b, c, with similar expressions for l, m. With appropriate choice of functionals k, l, and m, any conic section can be represented in this way. In the special case where $m = 1$, the curve is a parabola. Furthermore, if $k = u$ and $l = v$ then Equation (1) simplifies to Equation (2).

The three linear functionals k, l, and m are exactly the functions that graphics hardware specializes in evaluating (with appropriate perspective correction) for each pixel in a triangle, so the expression in Equation (2) fits well with the hardware. The idea of reducing an implicit equation to a sum of products of a few linear functionals is the key to extending our rendering approach to handle cubic curves.

4 Rendering Cubic Curves

Rendering cubic curves is important since many commercial drawing packages produce vector art representations that contain cubics. The observation that made shading of quadratic curves efficient was that all quadratics are the projected image of a *single* canonical quadratic that has a simple algebraic form. For cubics, any parametric curve is the projected image of one of *three* canonical curves illustrated in Figure 5 [Stone and DeRose 1989; Blinn 2003], (plus a quadratic, straight line, and point as degenerate cases). It has been known for some time [Salmon 1852] that all three of these curves

can be expressed implicitly in one simple homogeneous algebraic form:

$$c(x, y, w) = k^3 - lmn, \qquad (3)$$

Where k, l, m, and n are the homogeneous equations of the lines **k**, **l**, **m** and **n** respectively, in Figure 5.
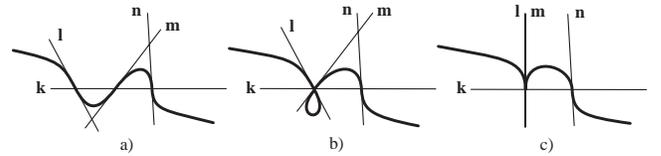
Figure 5: a) Serpentine curve. This curve has three collinear inflection points (on line **k**) with tangent lines **l**, **m** and **n** at those inflections. b) Loop curve. This curve has one inflection and one double point with **k** the line through them. The lines **l** and **m** are the tangents to the curve at the double point and **n** is the tangent at the inflection. c) Cusp curve. This curve has one inflection point and one cusp, with **k** the line through them. The line **l** = **m** is the tangent at the cusp and **n** is the tangent at the inflection.

4.1 Algorithm Overview

Our algorithm for rendering parametric cubics is as follows. For each cubic curve we start with four Bézier control points. We then calculate values for the four quantities k, l, m, and n at each control point to use as texture coordinates. These values are equal to the dot product of the control point coordinates with the lines **k**, **l**, **m** and **n**. We tessellate each Bézier curve segment as shown in Figure 7 and pass the triangles to the hardware. The graphics hardware will properly interpolate the k, l, m, n values so that a simple pixel shader needs only evaluate Equation 3 using these interpolated values.

4.2 Curve Categorization

As a first step in finding k, l, m, n we determine which of the cases of Figure 5 we have. This requires finding out how many inflection points the curve has. An inflection point is a point on the curve where the curvature vanishes; this occurs for values of t such that

the cross product of the first and second derivatives $\dot{C}(t) \times \ddot{C}(t) = 0$. As shown in [Blinn 2003]) the problem of finding the parameter values of the inflection points of a given cubic curve $C(t)$ can be converted to finding a special scalar valued cubic polynomial $I(t)$, whose roots are also at the inflection points parameters. We find the explicit form of $I(t)$ as follows.

First convert the Bézier control points, $\mathbf{b}_0, \ldots, \mathbf{b}_3$ to the power basis by the product $\mathbf{C} = \mathbf{M}_3 \cdot \mathbf{B}$. Then compute the vector $\mathbf{d} = [d_0 \ d_1 \ d_2 \ d_3]$ where

$$d_0 = \det \begin{bmatrix} x_3 & y_3 & w_3 \\ x_2 & y_2 & w_2 \\ x_1 & y_1 & w_1 \end{bmatrix}, \quad d_1 = -\det \begin{bmatrix} x_3 & y_3 & w_3 \\ x_2 & y_2 & w_2 \\ x_0 & y_0 & w_0 \end{bmatrix},$$

$$d_2 = \det \begin{bmatrix} x_3 & y_3 & w_3 \\ x_1 & y_1 & w_1 \\ x_0 & y_0 & w_0 \end{bmatrix}, \quad d_3 = -\det \begin{bmatrix} x_2 & y_2 & w_2 \\ x_1 & y_1 & w_1 \\ x_0 & y_0 & w_0 \end{bmatrix}.$$

Note that $\mathbf{d}$ is (up to a scalar multiple) the unique vector that is perpendicular to the columns of the coefficient matrix $\mathbf{C}$, that is $\mathbf{d} \cdot \mathbf{C} = [0 \ 0 \ 0]$. Clearly, a projective transform of the curve will not change $\mathbf{d}$, since $\mathbf{d} \cdot (\mathbf{C} \cdot \mathbf{P}) = (\mathbf{d} \cdot \mathbf{C}) \cdot \mathbf{P} = [0 \ 0 \ 0]$.

Given the vector $\mathbf{d}$, the inflection point polynomial is then

$$I(t,s) = d_0 t^3 - 3d_1 t^2 s + 3d_2 t s^2 - d_3 s^3$$

(see [Blinn 2003] for the derivation). Note that, since an inflection or other singularity may occur at a parametric value of infinity, we have expressed this polynomial in terms of the homogeneous parameter pair (t,s). The number and multiplicity of real roots to this polynomial will determine the curve type as follows

1. 3 distinct real roots - Serpentine

2. 1 real root and 2 complex roots - Loop

3. 1 double root and a distinct single root - Cusp

4. 1 triple root - Curve is a quadratic.

5. $I(t,s)$ is identically zero – Curve is a line or point.

To form a numerical test for the root count we evaluate the discriminant of $I(t,s)$ by the calculations

$$\delta_1 = d_0 d_2 - d_1^2,$$
$$\delta_2 = d_1 d_2 - d_0 d_3,$$
$$\delta_3 = d_1 d_3 - d_2^2,$$
$$discr(I) = 4\delta_1 \delta_3 - \delta_2^2.$$

If the discriminant is positive we have case 1, if negative we have case 2, and if zero we have case 3, or if additionally $\delta_1 = \delta_2 = \delta_3 = 0$ we have case 4, or if all $d_0 = \cdots = d_3 = 0$ we have case 5.

The loop (case 2) has a double point at the intersection of lines $\mathbf{k}, \mathbf{l}$, and $\mathbf{m}$. The two parameter values at this double point can be found as the solutions of the quadratic polynomial $H(t,s)$, the Hessian of $I(t,s)$ defined as

$$H(t,s) = I_{ss}I_{tt} - I_{st}^2,$$
$$= 36 \left(\delta_1 t^2 + \delta_2 ts + \delta_3 s^2 \right).$$

4.3 Finding klmn

In this section we show the calculations that will find the values of k, l, m, and n to use as texture coordinates at the control vertices of

a rational Bézier curve. Our aim is to give a high level overview of the procedure in the most general setting; in the next section we specialize to the integral cubic case and work out more of the details.

We assume that a given cubic curve $C(t,s)$ has been classified according to the method just given, and the three roots $(t_l, s_l), (t_m, s_m), (t_n, s_n)$ of the cubic inflection point polynomial $I(t,s)$ are known. If the curve is a loop, we assume that the parameter values (t_d, s_d) and (t_e, s_e) of the double point have been found as the roots of the quadratic polynomial $H(t,s)$. In the case of a cusp, the double point parameters coincide at the parameter value $(t_d, s_d) = (t_e, s_e)$.

Our strategy is to find cubic polynomials $k(t,s), l(t,s), m(t,s)$, and $n(t,s)$ that represent the values of the four linear functionals k, l, m, n evaluated at points on the curve $C(t,s)$, for example $k(t,s) = C(t,s) \cdot \mathbf{k}$. These polynomials are constructed differently for each of the three cubic curve types by considering how $C(t,s)$ behaves as it passes through the intersection points of line $\mathbf{k}$ with lines $\mathbf{l}, \mathbf{m}$, and $\mathbf{n}$. These points are all zeroes of $k(t,s), l(t,s), m(t,s)$, and $n(t,s)$, so we can construct these polynomials as products of known linear factors. We label these linear factors with the upper case letter corresponding the subscript of the parameter value from which is was constructed, that is

$$L = (st_l - ts_l), \quad M = (st_m - ts_m), \quad N = (st_n - ts_n),$$
$$D = (st_d - ts_d), \quad E = (st_e - ts_e).$$

The following table shows the factored forms of $k(t,s)$, $l(t,s)$, $m(t,s)$, and $n(t,s)$ for each of the cubic curve types:

	serpentine	loop	cusp
$k(t,s)$	LMN	DEN	D^2N
$l(t,s)$	L^3	D^2E	D^3
$m(t,s)$	M^3	DE^2	D^3
$n(t,s)$	N^3	N^3	N^3

Table 1: Factored forms of *klmn* polynomials.

Note that, for each curve type above, the relation $k^3 - lmn = 0$ is satisfied. The 4D texture coordinates to be assigned to the Bézier control points correspond to the Bézier coefficients of $k(t,s), l(t,s), m(t,s)$, and $n(t,s)$. The Bézier coefficients are found by expanding the factored forms of these polynomials, and collecting power basis coefficients into a 4×4 matrix $\mathbf{F}$, then taking the product $\mathbf{M}_3^{-1}\mathbf{F}$.

Note that $\mathbf{F}$ is not unique; any homogeneous scale of one or more of the roots (t_i, s_i) will generate a different $\mathbf{F}$. In particular, a sign change will flip the orientation of an implicit curve, reversing the roles of inside and outside. In order to resolve this ambiguity, we apply an orientation test to make sure the inside of the curve is to the right (by convention) of the direction of parametric travel as (t/s) increases.

Our orientation test is based on comparing the tangent line formula calculated from the parametric form, $C(t,s) \times \dot{C}(t,s)$, with that calculated by taking the gradient of the implicit form, $\nabla c(x,y,w)$ and evaluating it at the same point, $\nabla c(C(t,s))$. These will be the same up to a scale factor

$$\alpha(t,s) \left(C(t,s) \times \dot{C}(t,s) \right) = \nabla c(C(t,s)).$$

By our convention, if $\alpha(t,s) > 0$ we must flip the signs of k and l to properly orient the implicit curve.

To calculate the gradient of $c(x,y,w)$ we apply the chain rule to $f = k^3 - lmn$

$$\nabla c(x,y,w) = \frac{\partial f}{\partial k}\nabla k + \frac{\partial f}{\partial l}\nabla l + \frac{\partial f}{\partial m}\nabla m + \frac{\partial f}{\partial n}\nabla n,$$
$$= 3k^2\nabla k - mn\nabla l - ln\nabla m - lm\nabla n.$$

The gradients of $klmn$ are constants; they are just the columns of the matrix $\Psi = [\mathbf{k}\ \mathbf{l}\ \mathbf{m}\ \mathbf{n}]$, a 3×4 matrix whose columns represent the lines in Figure 5. So we can write $\nabla c(x,y,w)$ as the matrix product

$$\nabla c(x,y,w) = \Psi \cdot \begin{bmatrix} 3k^2 & -mn & -ln & -lm \end{bmatrix}^T.$$

We can compute the matrix Ψ by noting that

$$\mathbf{C}\Psi = \mathbf{F}.$$

Since $\mathbf{C}$ is not square, we use the *pseudo inverse* to compute Ψ, that is

$$\Psi = \left(\mathbf{C}^T\mathbf{C}\right)^{-1}\mathbf{C}^T\mathbf{F}.$$

In the next section we show a simplified situation and the above calculations are carried out explicitly.

4.4 Integral Cubics

Solving for the roots of $I(t,s)$ will, for arbitrary rational cubic curves, require solving a cubic equation. For simplicity, we have restricted our input curves to be integral, meaning that the w value of each control point equals 1. This makes the coefficients $w_1 = w_2 = w_3 = 0$, and makes the value $d_0 = 0$. The inflection point polynomial and its discriminant reduce to

$$I_{\text{integral}}(t,s) = s\left(-3d_1t^2 + 3d_2ts - d_3s^2\right),$$
$$discr\left(I_{\text{integral}}\right) = d_1^2\left(3d_2^2 - 4d_3d_1\right).$$

In this case one inflection point is always at $(t,s) = (1,0)$ and the other two require solving only a quadratic equation. We will identify the root at $(1,0)$ with the intersection of line $\mathbf{n}$ with line $\mathbf{k}$. The line $\mathbf{n}$ will be the line-at-infinity, $[0\ 0\ 1]^T$. Since the control points $\mathbf{b}_i$ of integral curves always have $w = 1$, $\mathbf{b}_i \cdot \mathbf{n} = 1$ so the interpolated value of n will always be 1. This allows us to eliminate one of our texture coordinate slots and simplify our shader equation further to

$$c(x,y) = k^3 - lm. \tag{4}$$

Note that this does not preclude perspective projection of the integral curve. This will work correctly and still maintain $n = 1$. Only if the original model had non-unit values for their w components (making them no longer be integral curves) will the interpolated n be other than 1.

In the integral case where $d_0 = 0$, the formula for the Hessian simplifies to

$$H(t,s) = 36\left(\left(d_3d_1 - d_2^2\right)s^2 + d_1d_2st - d_1^2t^2\right)$$

Note that if $H(t,s)$ has no real roots (meaning that we have a serpentine curve) we will have $H(t,s) < 0$ for all (t,s).

To handle all the possible geometric situations we must consider the following six cases:

---

1. The Serpentine $\qquad d_1 \neq 0,\ 3d_2^2 - 4d_1d_3 > 0$

For integral curves the three inflection points occur at the following (t,s) parameter values (the homogeneous roots of $I_{\text{integral}}(t,s)$):

$$(t_l,s_l) = \left(d_2 + \frac{1}{\sqrt{3}}\sqrt{3d_2^2 - 4d_1d_3},\ 2d_1\right),$$
$$(t_m,s_m) = \left(d_2 - \frac{1}{\sqrt{3}}\sqrt{3d_2^2 - 4d_1d_3},\ 2d_1\right),$$
$$(t_n,s_n) = (1,0)$$

(Actually a somewhat more stable quadratic solution technique should be used [Press et al. 1992], but the above is shown for simplicity). Since any scalar multiple of (t_l, s_l) or (t_m, s_m) represent the same roots, it is a good idea to scale these homogeneous 2D vectors to be unit length to avoid possible exponent overflows.

Putting these root values into the factor forms of the polynomials shown in Table 1 and multiplying them out to get the power basis coefficients of $klmn$ gives

$$\mathbf{F} = \begin{bmatrix} t_lt_m & t_l^3 & t_m^3 & 1 \\ -s_mt_l - s_lt_m & -3s_lt_l^2 & -3s_mt_m^2 & 0 \\ s_ls_m & 3s_l^2t_l & 3s_m^2t_m & 0 \\ 0 & -s_l^3 & -s_m^3 & 0 \end{bmatrix}.$$

The ith row of $\mathbf{M}_3^{-1}\mathbf{F}$ is assigned to point $\mathbf{b}_i$ as a texture coordinate.

Evaluation of our orientation test for the case $d_0 = 0,\ d_1 \neq 0$ shows

$$\alpha(t,s) = \tfrac{32}{3}d_1^3 H(t,s)$$

(This formula also applies to cases 2 and 3a below). Since $H(t,s)$ is negative for the serpentine curve, this means that we must flip the sign of k and l if d_1 is negative.

---

2. The Loop $\qquad d_1 \neq 0,\ 3d_2^2 - 4d_1d_3 < 0$

The double point occurs at the roots of $H(t,s)$, which are:

$$(t_d,s_d) = \left(d_2 + \sqrt{4d_1d_3 - 3d_2^2},\ 2d_1\right),$$
$$(t_e,s_e) = \left(d_2 - \sqrt{4d_1d_3 - 3d_2^2},\ 2d_1\right).$$

These roots are used to compute the $k(t,s), l(t,s)$ and $m(t,s)$ for the loop case using Table 1; resulting in the power basis coefficients matrix

$$\mathbf{F} = \begin{bmatrix} t_dt_e & t_d^2t_e & t_dt_e^2 & 1 \\ -s_et_d - s_dt_e & -s_et_d^2 - 2s_dt_et_d & -s_dt_e^2 - 2s_et_dt_e & 0 \\ s_ds_e & t_es_d^2 + 2s_et_ds_d & t_ds_e^2 + 2s_dt_es_e & 0 \\ 0 & -s_d^2s_e & -s_ds_e^2 & 0 \end{bmatrix}.$$

If one of the parameter values (t_d/s_d) or (t_e/s_e) should lie in the interval $[0,1]$, then the double point will cause a shading anomaly. We see this in the left side of Figure 6 where the desired inside/outside decision for the function changes when passing over the double point. We eliminate this possibility by subdividing the curve at the offending parameter value. This will move the double point to $t/s = 0$ and $t/s = 1$ for the two subcurves, and guarantee that $\alpha(t,s) = 32/3\,d_1^3 H(t,s)$ will not change signs over the parameter interval $[0,1]$ of each curve. Due to the subdivision of a curve, either $H(0)$ or $H(1)$ will be zero for each subcurve. Our orientation test for loop curves is based on the larger (in absolute value) of $H(0)$ and $H(1)$ to handle the case where one of these is zero. If $d_1H(\cdot)$ is positive, then we flip the signs of k and l.
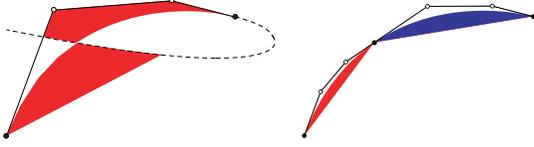
Figure 6: If a cubic curve has a double point in the interval $[0, 1]$, then the curve is subdivided at the double point and treated as two curves.

3a. Cusp with inflection at infinity $d_1 \neq 0$, $3d_2^2 - 4d_1 d_3 = 0$

This is the boundary case between the above two situations. The two roots of the quadratic portion of $I(t, s)$ are equal

$$(t_d, s_d) = (d_2, 2d_1)$$

so lines **l** and **m** are the same. This case can actually be merged with case 1, since that case does the right thing if $3d_2^2 - 4d_1 d_3 = 0$.

3b. Cusp with cusp at infinity $d_1 = 0$, $d_2 \neq 0$

The inflection point polynomial is

$$I(t, s) = s^2 (3d_2 t - d_3 s)$$

which has a double root at parametric infinity $(s = 0)$. This means that the cusp is at infinity and the inflection point is local. This is a perfectly possible situation for integral curves. We assign the roots of I as follows, exchanging the roles of lines **l** and **n** from Figure 5.

$$(t, s)_l = (d_3, 3d_2)$$
$$(t, s)_m = (1, 0)$$
$$(t, s)_n = (1, 0)$$

The resulting power basis coefficient matrix for this case is

$$\mathbf{F} = \begin{bmatrix} t_l & t_l^3 & 1 & 1 \\ -s_l & -3s_l t_l^2 & 0 & 0 \\ 0 & 3s_l^2 t_l & 0 & 0 \\ 0 & -s_l^3 & 0 & 0 \end{bmatrix}.$$

Conversion to Bézier form reveals that the interpolated values of both m and n are constant at 1. To keep the shader uniform for all curves we retain the m interpolation and just plug the value 1 into the m slot of all four control points.

To construct an orientation test, we first note that when $d_1 = 0$ the Hessian simplifies to

$$H(t, s) = -36 d_2^2 s^2.$$

Further derivations show that

$$\alpha(t, s) = \tfrac{9}{2} H(t, s).$$

Since this is negative for all (t, s) we never have to flip the signs of k and l.

---

4. The curve is really a quadratic $d_1 = d_2 = 0$, $d_3 \neq 0$

In this case the curve could be rendered using the quadratic techniques of Section 3. However this requires a different pixel shader equation to be evaluated. We have seen that this is the function $c(x, y) = k^2 - lm$. For integral curves, the line **m** is the line-at-infinity and its interpolated value is constant at 1, so the quadratic

shader function only needs to be $c(x, y) = k^2 - l$ as derived in section 3. We can therefore cause a quadratic curve to masquerade as a cubic curve (and use the existing cubic machinery) by simply multiplying through by k yielding

$$c(x, y) = k^3 - lk$$

We simply put a copy of the texture coordinates for k into the m slot of the tessellated Bézier control mesh. This actually evaluates an implicit function that is the product of the desired curve with the line **k**. This is generally not a problem since line **k** does not intersect the Bézier control mesh except at just the single point $\mathbf{b}_0$.

---

5. The curve is really a line or point $d_1 = d_2 = d_3 = 0$

This final degeneration of the curve into a line (point) takes place only if all four control points are collinear (coincident). In that case the tessellated triangles have zero area and can be eliminated from the mesh.

One potential difficulty with our categorization is the possibility of numerical problems encountered when, say d_1 is almost (but not exactly) zero. While this has not been a problem in our experience, it can be addressed by finding a threshold for d_1 where the exact $d_1 = 0$ curve is within subpixel accuracy of the approximation. However, if curve control points are defined in a fixed point design space (as is the case in most graphic design scenarios) then the d_i's are computed exactly and the categorization is unambiguous.

4.5 Tessellation of Cubics

We assign texture coordinates from the product $\mathbf{M}_3^{-1} \mathbf{F}$, to the control points of a cubic Bézier curve and locally triangulate these 4 points. We constrain the edges of these triangles and globally triangulate all of the control points, honoring the constrained edges and enforcing the Deluanay condition elsewhere. We remove the triangles that do not belong to the interior of the shape, see Figure 7. We upload the triangles to the GPU for rendering using a shader program that evaluates Equation (4).

5 Anti-Aliasing

The GPU will sample the implicit equations given in Sections 3 and 4 at pixel centers, producing images with *aliasing* artifacts. We can reduce these artifacts by performing anti-aliasing calculations in our pixel shader code. If we know the distance from a pixel to a curve, we can estimate a filtered alpha value for the pixel by either a 1D texture lookup or by evaluating a simple blending function [Gupta and Sproull 1981; Turkowski 1982].

All of our previous calculations have taken place in curve design space $[x\ y]$, we now move to pixel space $[X\ Y]$. We approximate the distance from a pixel to a curve $g(X, Y) = 0$ as follows. The gradient

$$\nabla g(X, Y) = \begin{bmatrix} \dfrac{\partial g}{\partial X}(X, Y) & \dfrac{\partial g}{\partial Y}(X, Y) \end{bmatrix}$$

is a vector perpendicular to the curve when $[X\ Y]$ is on the curve. For pixels $[X\ Y]$ that are close to the curve, the vector ∇g is still *nearly* perpendicular to the curve. We use the gradient to define an approximate *signed distance function* from a pixel to a curve to be

$$d(X, Y) = \frac{g(X, Y)}{\|\nabla g(X, Y)\|}.$$
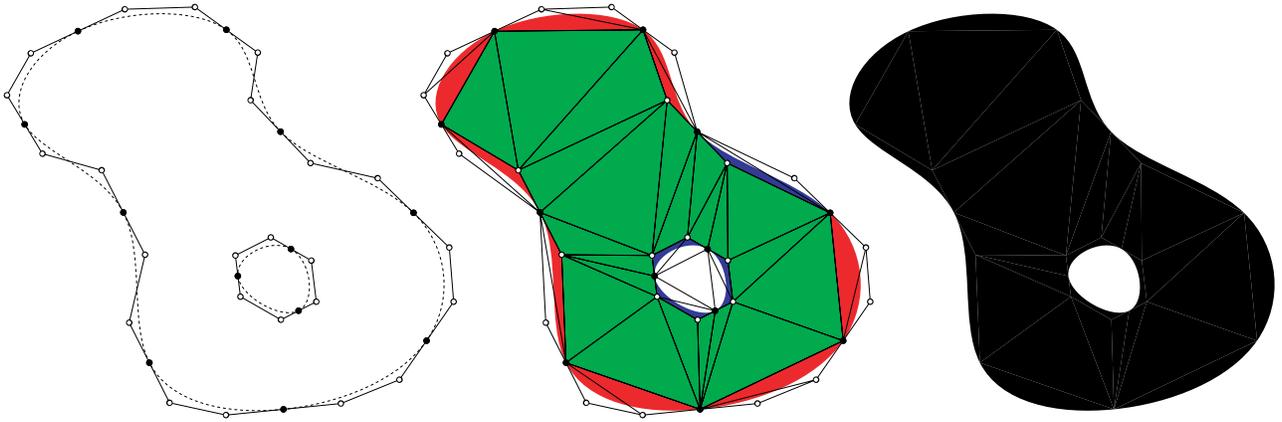
Figure 7: A region bounded by cubic Bézier curves (left). Control points are assigned texture coordinates and the shape is triangulated (middle). The triangles are rendered using our pixel shader, resulting in a resolution independent projectively transformed shape (right).

This distance function assumes that $g(X,Y)$ is known in screen space. The implicit curves evaluated in our pixel shader programs are defined in texture space, not screen space.

We can find $g(X,Y)$ as a composition. If all curve control points lie in the $z = 0$ plane, then the viewing transform is a mapping from this plane to the screen plane represented by the 3×3 matrix $\mathbf{V}$, formed by removing the third row and column from the usual 4×4 composite viewing matrix. If $c(x,y)$ is a design space curve, we can write its screen space image as

$$g(X,Y) = c(\mathbf{V}^{-1}(X,\ Y)),$$
$$= f(\Psi(\mathbf{V}^{-1}(X,\ Y))),$$
$$= f(\Phi(X,\ Y,)),$$

where Ψ is the mapping from curve design space to texture space, and $\Phi = \Psi \circ \mathbf{V}^{-1}$ is the mapping from screen space to texture space. We do not have direct access to Φ; rather, it is implemented by the GPU as it interpolates the texture coordinates assigned to triangle vertices. The GPU evaluates f using pixel shader code.

There are several ways to compute $\nabla g(X,Y)$. Newer hardware implementations support *gradient instructions*. By arranging for 2×2 pixel blocks to execute in parallel, the hardware is able to take differences of values local to adjacent pixels. These differences are used to approximate gradients with respect to pixel coordinates in pixel shaders. If gradient instructions are available, ∇g can be approximated from the values of $g(X,Y)$ computed at adjacent pixels. Or, we can apply the chain rule to get

$$\nabla g(X,Y) = \nabla f(\Phi(X,Y)) \circ J(\Phi),$$

and compute $\nabla g(X,Y)$ *exactly* since gradient instructions applied to $\Phi(X,Y)$ (the interpolated texture coordinates) compute the Jacobian $J(\Phi)$ exactly, since Φ is linear in projective coordinates. Finally, if gradient instructions are not available, we must encode the matrix for Ψ in each vertex, so that Φ is available to compute ∇g.

5.1 Tessellation for Anti-Aliasing

Our signed distance function can be used to determine a filtered color value when a pixel is close to a curve boundary. This works well for pixels on the interior of a triangle containing a curve. However, pixels that need to be affected by anti-aliasing calculations often belong to interior triangles that do not contain curves, or may be outside the boundary of the shape.

The only way to affect pixels whose centers lie outside the shape boundary is to add geometry to cover these pixels. We have tried various ways of doing this. One possibility, is to add a thin layer of triangles around the perimeter of a shape. The exact thickness of this layer will depend on the screen space projection of the shape. This can be worked out in a vertex shader by encoding the boundary as a 2D line, whose transformed image can be used to measure perpendicular distance from the boundary. This approach gets somewhat tricky as the *bloated* boundaries of adjacent shapes overlap, requiring alpha blending that may not yield correct results.

A much simpler approach is to include some of the *negative space* of a shape by enclosing it in a slightly enlarged bounding box and triangulating. This will create triangles that will (in general) cover pixel centers that are adjacent to line segments or points of tangency on the curves. It is still possible that for highly oblique viewpoints some pixel centers may be missed. Our experience with a bounding box enlargement of 10% has not shown this to be a problem.

The other difficulty relating to preparing geometry for anti-aliasing is handling triangles that do not contain curves. For triangles that have one edge on a boundary, we can assign texture coordinates so that the edge will be treated as an implicit line. For triangles with two edges on a boundary, we can assign texture coordinates to treat the shape as a pair of intersecting lines. If a triangle has all three edges on the boundary, then we cannot find a quadratic that will interpolate this data. In such cases we can subdivide the triangle to isolate the boundaries.

In practice, we take a somewhat *brute force* approach by subdividing all interior triangles to compute a variant of the *chordal axis* [Prasad 1997] of a shape. The chordal axis is similar to the medial axis, and easily computed from our triangulation, as shown in Figure 8. By subdividing triangles along the chordal axis, each new triangle will have at most two vertices incident on a boundary, greatly simplifying texture coordinate assignment for producing a signed distance function.
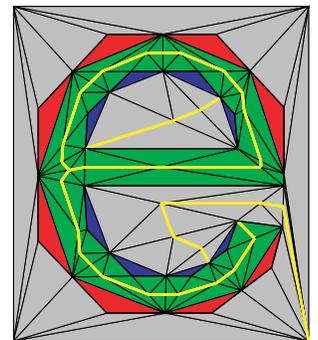
Figure 8: Chordal axis of letter 'e' shown in yellow.

The results of our anti-aliasing scheme are shown in Figure 9. On the left is the rendering of TrueType data using only the in/out test for triangles containing a curve (the image has been enlarged somewhat to make pixels more visible). On the right is the same set of triangles that has been split along the chordal axis and rendered using our anti-aliased technique.

Figure 9: On the left is a vector image of text using just our in/out pixel shader. On the right is the result of our anti-aliasing scheme.

If the image in Figure 9 were rotated a bit more, the image will begin to collapse to a line. This degenerate situation will result in image artifacts as our curves are poorly sampled at such extreme viewpoints. While these artifacts are confined to a narrow band only a few pixels in width, they create a highly pixelated look that we strive to eliminate in the next section.

6 Degenerate Transforms

Careful treatment of extreme viewpoints, such as when a planar image is viewed (nearly) edge-on, can greatly enhance overall image quality. Recall that $\mathbf{V}$ is the 3×3 matrix that represents the transform from curve design space to screen space. We know that a triangle will degenerate when $\det(\mathbf{V}) = 0$. Visually, this will mean that a triangle is being viewed edge on. When this happens, or is *close* to happening, entire triangles may fall between pixel centers resulting in undersampling artifacts.

One way of avoiding this is to monitor $\det(\mathbf{V})$ and do something as it nears zero. However, this metric is meaningless since the projective matrix $\mathbf{V}$ is scale invariant. What is needed is a metric in meaningful units that will tell us when a triangle is approaching the *edge-on* state.

Our solution to this problem is to map the line-at-infinity in the curve design plane $z = 0$ to the screen space line $\mathbf{h}$. When a triangle is viewed edge-on, it will coincide with this line. We transform this line to the screen by

$$\begin{aligned} \mathbf{h} &= \mathbf{V}^* \cdot [\,0\ \ 0\ \ 1\,]^T, \\ &= \mathbf{v}_0 \times \mathbf{v}_1, \end{aligned}$$

where $\mathbf{V}^*$ is the adjoint of $\mathbf{V}$, and $\mathbf{v}_0 \times \mathbf{v}_1$ is the cross product of the first two rows of $\mathbf{V}$. If $(\mathbf{v}_0 \times \mathbf{v}_1) \propto [\,0\ \ 0\ \ 1\,]^T$, then the line-at-infinity maps onto itself, and is not visible on screen; otherwise, we *normalize* $\mathbf{h}$ so that $\mathbf{h}_x^2 + \mathbf{h}_y^2 = 1$. When the resulting linear functional is evaluated at a pixel, the value corresponds to the distance (in pixels) from the given pixel to the projected image of the line-at-infinity. We choose an arbitrary tolerance of 50 pixels to the line $\mathbf{h}$ and smoothly reduce opacity in this region.

The result is that planar images can be freely transformed with no pixelation artifacts. As the plane containing the image approaches being viewed edge on, it will fade out and fade in as the plane is rotated. The effect is subtle, but its absence is noticeable. The cost of this effect is a dot product in the vertex shader, and an extra field of vertex data that needs to be interpolated by the rasterizer, and the computation or lookup of, and multiplication by, the fade coefficient.

7 Discussion

Our scheme consists of two distinct phases. In the first phase, we analyze the constituent curve segments, looking for overlap and double points and subdividing as necessary. We assign texture coordinates for subsequent shader evaluation and triangulate the plane together with the Bézier control points of the curves. All of this work is done as a preprocess on the CPU. In phase two, the triangles are transferred to GPU memory and rendered using the programmable pipeline. Once resident in GPU memory, the CPU is free to do other work and need only issue new transformation matrices for each new frame.

The basic in/out test uses a very small number of instructions. The following is a Microsoft DirectX high-level shader language (HLSL) listing of our integral cubic shader:

```
float4x4 WVP : WORLDVIEWPROJECTION;

float4 VertexShader (
    float3 pos        : POSITION,
    inout float3 klm  : TEXCOORD0 ) :  POSITION
{
    return mul(float4(pos, 1), WVP);
}

float4 PixelShader(float3 klm :  TEXCOORD0) :  COLOR
{
    clip(pow(klm.x,3) - klm.y*klm.z);
    return (float4)0;
}
```

We omit the anti-aliasing instructions for simplicity. Our `VertexShader()` transforms a vertex from world space to device coordinates using the matrix `WVP`. The texture coordinate slots, stored in the `xyz` components of the vector `klm`, are passed to the pixel shader untouched. Our `PixelShader()` evaluates the cubic function of Equation (4). If the resulting argument to `clip()` is negative, the pixel is *killed* and no further processing is done; otherwise the pixel shader returns the color black.

Another application of our method is to rendering curves as paths instead of filling closed regions. Since we have a signed distance

Figure 10: We use our curve rendering technique to embed vector based geometry on the surface of a 3 dimensional object. A triangular mosaic *vector image* is (possibly subdivided and) deformed to become a part of the existing surface geometry.

function for determining a pixel's distance to a curve boundary, we alter our shader slightly to render just the boundary with varying thickness. Our anti-aliasing scheme works in this context as well resulting in a high quality, high performance curve rendering algorithm.

Aside from rendering planar images of vector based geometry, we can use our algorithm to texture objects in a resolution independent way. The idea is to embed one of our planar images on a surface. Once we have tessellated and assigned texture coordinates we can deform the image plane by an arbitrary function, or project it onto a mesh. This simple approach might reveal the faceted nature of the vector image if the local curvature of the under lying surface is too high. We can deal with this by subdividing our vector image. Each triangle of the vector image can be split 1 to 4, recursively, to create a flexible image. The position and texture coordinates of the subtriangles will yield the expected subimage. Similarly, we could dice the vector image at intersections with a regular grid. If we maintain the original vertices within the grid, then the added geometric complexity will not alter the rendered image. It will however, lead to fewer faceting artifacts when deformed. These are sampling artifacts, but come from sampling the geometry, not the shading. The vector image as it appears on a curved surface will be resolution independent, see Figure 10.

8   Conclusions

We have presented a simple, high performance curve rendering algorithm whose efficiency is predicated on the design of modern programmable graphics hardware. We compute a set of texture coordinates that are attached to the vertices of quadratic and cubic Bézier curve control points. These texture coordinates encode a corresponding implicit function for these parametric curves that is evaluated by a pixel shader program to determine if the pixel is inside or outside of the curve, relative to the Bézier control hull. Our pixel shader program is extremely simple resulting in very high performance.

Our algorithm has the property that once the CPU has preprocessed the curve data, the resulting geometry is sent to the GPU for rendering. However, this model does not allow for dynamic geometry without CPU involvement. While the analysis code for classifying curves and assigning texture coordinates is well within the scope of GPU execution, the global nature of triangulation and overlapping triangle avoidance is not. We expect to render some dynamic geometry, but at reduced performance.

Our examples have only shown filled areas with solid color. We envision that shaders can be written to provide a wide range of fill tools, such as gradients and textures.

References

ARNON, D. 1983. Topologically reliable display of algebraic curves. *Siggraph 1983 Conference Proceeding 17*, 3 (July), 219–227.

BLINN, J. 2003. *Jim Blinn's Corner Notation, notation, notation*. Morgan Kaufmann. Chap. 14,15,16, and 19.

GUMHOLD, S. 2003. Splatting illuminated ellipsoids with depth correction. In *Proceedings of 8th International Fall Workshop on Vision, Modelling and Visualization 2003*, 245–252.

GUPTA, S., AND SPROULL, R. 1981. Filtering edges for gray-scale displays. *Siggraph 1981 Conference Proceeding 15*, 3, 1–5.

PRASAD, L. 1997. Morphological analysis of shapes. *CNLS Newsletter 139* (July), 1–18.

PRESS, W., TEUKOLSKY, S., VETTERLING, W., AND FLANNERY, B. 1992. *Numerical Recipes in C*. Cambridge Press.

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

SALMON, G. 1852. *A Treatise on the Higher Plane Curves*. Dublin, Hodges & Smith. available online at http://name.umdl.umich.edu/ABQ9497.

SEDERBERG, T. 1983. *Implicit and Parametric Curves and Surfaces for Computer Aided Geometric Design*. PhD thesis, Purdue University. Mechanical Engineering Department.

SEN, P. 2004. Silhouette maps for improved texture magnification. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, Eurographics Association.

STONE, M., AND DEROSE, T. 1989. A geometric characterization of parametric cubic curves. *ACM Transactions on Graphics 8*, 4 (July), 147–163.

TAUBIN, G. 1994. Distance approximations for rasterizing implicit curves. *ACM Transactions on Graphics 13*, 1 (January), 3–42.

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

TUPPER, J. 2001. Reliable two-dimensional graphing methods for mathematical formulae with two free variables. *Siggraph 2001 Conference Proceeding*, 77–86.

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