Real-Time GPU Rendering of Piecewise Algebraic Surfaces

Charles Loop*
Microsoft Research

Jim Blinn†
Microsoft Research

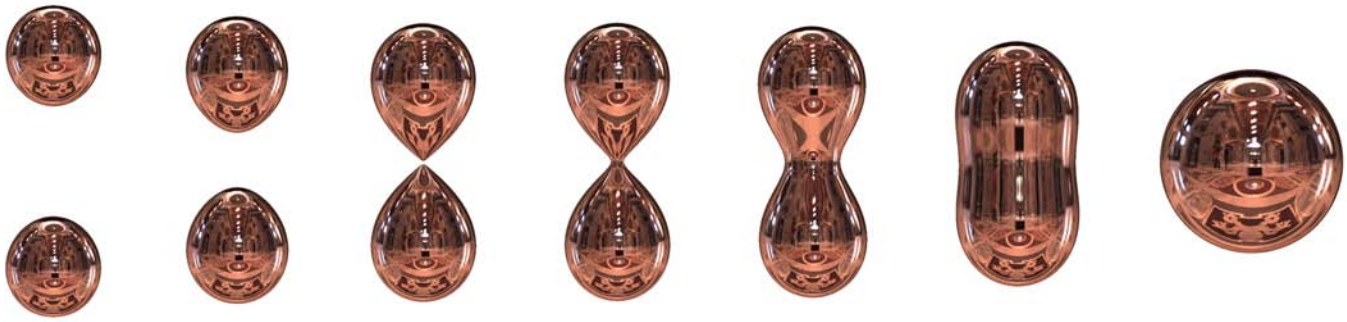

Figure 1: Several states of an animated fourth order algebraic surface rendered in real-time using our technique.

Abstract

We consider the problem of real-time GPU rendering of algebraic surfaces defined by Bézier tetrahedra. These surfaces are rendered directly in terms of their polynomial representations, as opposed to a collection of approximating triangles, thereby eliminating tessellation artifacts and reducing memory usage. A key step in such algorithms is the computation of univariate polynomial coefficients at each pixel; real roots of this polynomial correspond to possibly visible points on the surface. Our approach leverages the strengths of GPU computation and is highly efficient. Furthermore, we compute these coefficients in Bernstein form to maximize the stability of root finding, and to provide shader instances with an early exit test based on the sign of these coefficients. Solving for roots is done using analytic techniques that map well to a SIMD architecture, but limits us to fourth order algebraic surfaces. The general framework could be extended to higher order with numerical root finding.

Keywords: algebraic surface rendering, implicit surface rendering, Bézier tetrahedra, GPU algorithms

1 Introduction

Graphics hardware has been optimized to rasterize triangles, storing color and depth information in pixel memory. In recent years, the computation of color and depth information per pixel has come under programmer control [Gray 2003]. Such pixel processing maps well to a SIMD architecture; allowing hardware vendors to exploit parallelism and achieve high performance. These pixel programs, or *shaders*, have evolved to support sophisticated appearance models to compute the color, or shade of a pixel. Ever larger and more complex shaders, combined with rapid gains in triangle throughput, have lead to stunning realism in real-time synthetic imagery. However, these advances have done little to aid in level-of-detail (LOD) management. LOD management is needed to avoid under sampling, or tessellation, artifacts when a curved surface is viewed up close; and to avoid over sampling, wasting resources, both temporal and spatial, when a densely triangulated surface is viewed from afar.

In this paper, we leverage the processing power of a Graphics Processing Unit (GPU) to directly render curved primitives. By rendering curved surfaces directly, as opposed to an approximating triangle mesh, we avoid tessellation artifacts and the need for LOD management. Since we only need to store the (resolution-independent) polynomial coefficients of a each surface element, this approach requires less memory and bus bandwidth in general then a corresponding (resolution-dependent) triangle mesh. Furthermore, true surface normals can be used for lighting at each pixel, leading to better surface renderings than one gets with interpolated Phong normals.

Our curved primitives are algebraic surfaces defined by trivariate *Bézier tetrahedra* [de Boor 1987; Hoschek and Lasser 1989]. An algebraic surface is an implicit surface defined as the solution of a polynomial equation. We only render the portion of surface inside a bounding tetrahedron; this is analogous to curve segments and surface patches that are the image of a bounded domain. The restriction to a tetrahedron combined with simple continuity conditions between adjacent tetrahedra enables the modeling of piecewise smooth surfaces.

In this paper, we consider analytic techniques for finding the zeros of polynomials. This limits us to a maximum of degree 4 surfaces, but this space is quite rich. Higher order surfaces could be rendered using this framework if robust root finding with bounded iterations were available on a GPU.

1.1 Previous Work

Piecewise smooth algebraic surfaces were introduced in [Sederberg 1985]. [Dahman 1989] proposed a scheme for interpolating position and normal data based on piecewise smooth second order Bézier tetrahedra. Several similar schemes have appeared [Guo

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

1995; Bajaj 1997; Bangert and Prautzsch 1999]. These papers establish Bézier tetrahedra as a geometric modeling primitive, but not as a rendering primitive.

There is extensive literature on rendering algebraic surfaces that goes back several decades. An early scan-line system was developed at the University of Utah by Mahl [Mahl 1972]. A very good overview of work in this area, with an extensive bibliography, can be found in [Heckbert 1984]. Sederberg and Zundel [Sederberg and Zundel 1989] have developed a scan-line algorithm for rendering algebraic surfaces. They find scan-line/silhouette intersections to determine and fill visible intervals via interpolation. A GPU algorithm for rendering implicit surface data has appeared [Hadwiger et al. 2005], but this work is concerned with surfaces over a voxel grid, not surfaces described by polynomials.

1.2 Algorithm Overview

We render an algebraic surface corresponding to a Bézier tetrahedron by encoding coefficient data as vertex attributes and rasterizing the projected triangle faces. A pixel shader program is executed that performs the following at each pixel

1. Compute coefficients of a univariate polynomial in z.
2. Find real roots to determine if the surface is visible.
3. Determine if the visible surface is inside tetrahedron.
4. Calculate the surface normal.
5. Compute shading.

Our contributions address steps 1 through 4, to take advantage of the strengths of the GPU to efficiently and accurately perform these calculations at high framerate, producing artifact-free images.

2 Mathematics of Algebraic Surfaces

An algebraic surface of degree d can be defined by a *Bézier tetrahedron* as follows

$$\sum_{i+j+k+l=d} b_{ijkl}\binom{d}{ijkl} r^i s^j t^k u^l = 0, \tag{1}$$

where b_{ijkl} are scalar valued *weights* that control the shape of the surface, and r,s,t,and u are barycentric coordinates ($r+s+t+u=1$) of a point in space with respect to some world space domain tetrahedron $\mathbf{T} = [\mathbf{v}_0\ \mathbf{v}_1\ \mathbf{v}_2\ \mathbf{v}_3]^T$ (each row of $\mathbf{T}$ corresponds to a vertex). When r,s,t, and u satisfy Equation (1) and are all positive, then the surface lies inside the tetrahedron. The weights have a geometric interpretation reminiscent of parametric Bézier curves and surfaces. Each weight b_{ijkl} is associated with position $(i\mathbf{v}_0 + j\mathbf{v}_1 + k\mathbf{v}_2 + l\mathbf{v}_3)/d$, forming a tetrahedral array as illustrated in Figure 2. Positive and negative weights act as attractive or repellent forces in the vicinity of the position of the weight.

As with parametric Bézier curves and surfaces, there are affine relationships that must be satisfied between the weights of adjacent tetrahedral elements for derivative continuity up to any order. This makes it possible to build piecewise smooth collections of Bézier tetrahedra to represent complex shapes.

For every Bézier tetrahedron there is a unique symmetric multiaffine map known as its *blossom* or *polar form* [Ramshaw 1989]. It is well known that the blossom of a polynomial in Bézier form can be evaluated using deCastlejau's algorithm. For Bézier tetrahedra

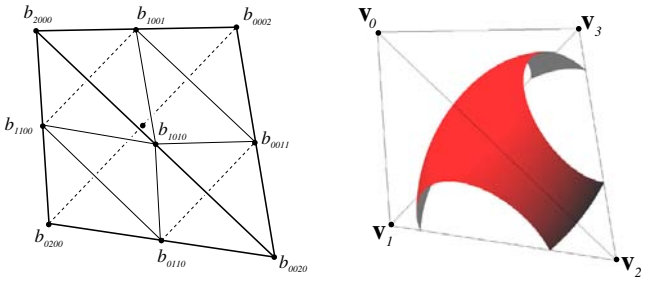

Figure 2: A Quadratic Bézier tetrahedron. On the left, the layout of Bézier weights within a tetrahedron. On the right, vertex labeling together with an algebraic surface restricted to the tetrahedron.

this requires the implementation of arrays with tetrahedral indexing. As an alternative but equivalent approach, we evaluate a blossom using a symmetric tensor. A tensor is a higher dimensional analog of a matrix, where the number of indices is referred to as the *rank* of the tensor. Using tensors allows us to express and evaluate blossoms in terms of dot products that are native operations on a GPU.

Tensor algebra generalizes the notion of dot product and matrix multiplication to *tensor contraction*. We represent this using Einstein index notation. This notation places contravariant indices as superscripts and covariant indices as subscripts. An expression that has the same symbol, typically a Greek letter, in a superscript and a subscript means that an implied summation is performed over that index. (A detailed tutorial is given in Chapter 20 of [Blinn 2003])

In tensor notation, a degree d Bézier tetrahedron is defined as d contractions

$$\mathbf{r}^{\alpha_1} \cdots \mathbf{r}^{\alpha_d} \mathbf{B}_{\alpha_1 \cdots \alpha_d} = 0, \tag{2}$$

where $\mathbf{B}$ is a symmetric rank d tensor containing the Bézier weights, and $\mathbf{r} = [r\ s\ t\ u]$. The elements of tensor $\mathbf{B}$ are assigned Bézier weights by

$$\mathbf{B}_{\alpha_1 \cdots \alpha_d} = b_{e_{\alpha_1} + \cdots + e_{\alpha_d}},$$

where e_α is 4-tuple with a 1 at position α, and zeros elsewhere.

We prefer tensor notation as it maps directly to a dot product implementation. For example in the case $d = 2$, $\mathbf{B}$ is a 4×4 matrix and we can write Equation (2) as $\mathbf{r} \cdot (\mathbf{r} \cdot \mathbf{B})^T = 0$. For arbitrary degree d Einstein index notion avoids nested parenthesis. Using tensors has greater computational complexity than tetrahedral arrays; evaluation of (2) is $O(4^d)$ while evaluation of (1) is only $O(d^3)$. In practice however, the shader compiler (Microsoft's High Level Shader Language - HLSL) is able to take advantage of symmetries in $\mathbf{B}$ and eliminate the redundant computations inherent in the evaluation of (2).

3 Per Frame Processing

3.1 Transformation to Screen Space

We transform both the Bézier tetrahedron vertices $\mathbf{T}$ and weights $\mathbf{B}$ to screen space for rendering. We use a standard 4×4 matrix $\mathbf{M}$, formed as the product of *world*, *view*, and *perspective* matrices, that takes world space points to screen space. For our purposes, screen space is a 4D projective space where pixel coordinates $[x\ y] \in [-1,1] \times [-1,1]$ and depth z correspond to $w = 1$. The

symmetric rank d tensor $\mathbf{B}$ is defined with respect to barycentric coordinates $\mathbf{r} = [r\ s\ t\ u]$. By definition of barycentric coordinates

$$\mathbf{x} \;=\; \mathbf{r} \cdot \mathbf{T},$$

where $\mathbf{x}$ is a point in world space. The composite transform from barycentric coordinates to screen space, $\tilde{\mathbf{x}}$, is written

$$\tilde{\mathbf{x}} \;=\; \mathbf{r} \cdot (\mathbf{T} \cdot \mathbf{M}).$$

Therefore, the barycentric coordinates of a screen space point are

$$\mathbf{r} \;=\; \tilde{\mathbf{x}} \cdot \left(\mathbf{M}^{-1} \cdot \mathbf{T}^{-1}\right) \;=\; \tilde{\mathbf{x}} \cdot \mathbf{W}.$$

Note that due to the projective nature of $\mathbf{W}$, the components $\mathbf{r}$ may not sum to 1; though they may be normalized to do so. From this, we can transform tensor weights $\mathbf{B}$ to screen space by

$$\tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d} \;=\; \mathbf{W}^{\alpha_1}_{\beta_1} \cdots \mathbf{W}^{\alpha_d}_{\beta_d} \mathbf{B}_{\alpha_1 \cdots \alpha_d}.$$

Note that only the unique elements of $\tilde{\mathbf{B}}$ need be computed. In screen space our original algebraic surface can be written

$$\tilde{\mathbf{x}}^{\beta_1} \cdots \tilde{\mathbf{x}}^{\beta_d} \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d} \;=\; 0. \tag{3}$$

Transforming the bounding tetrahedron $\mathbf{T}$ and weight tensor $\mathbf{B}$ to screen space for rendering has the advantage that all viewing rays become parallel to the z axis. This simplification leads to more compact formulae for the coefficients of ray/surface intersection polynomials. An alternative approach is to transform the viewing rays into the barycentric coordinate system of each Bézier tetrahedron, and solve a univariate equation in this space. This approach has the advantage that the weight tensor $\mathbf{B}$ need not be transformed, avoiding possible precision loss. We take the former approach as the details are more straightforward and fewer interpolated vertex attributes are needed.

3.2 Rendering Primitives

We have experimented with two different approaches to rendering tetrahedra. The obvious solution is to render the front facing triangular faces. Using this approach we can take advantage of GPU vertex processing by creating a vertex shader program to transform data, as outlined in the previous section. There are two disadvantages of this however. First, much of the computation being done per tetrahedron must be duplicated at each of the four tetrahedral vertices. Second, an important aspect of our computation is determining visibility within a tetrahedron; having to consider all four bounding planes (the faces) on input to a pixel shader creates a complicated interval that z values must be tested against, resulting in instruction sequences with many branches.

Instead, we perform vertex processing on the CPU and submit post transformed triangles to the GPU. We have not done a rigorous analysis of this design tradeoff on current (circa 2006) graphics hardware. However, we anticipate this will be the better approach on next generation GPUs that will support per triangle shader calls [Blythe 2006], and these computations can migrate back to the GPU.

Our approach to rendering a tetrahedron is to render the non overlapping triangles that cover the convex hull in screen space. This is similar to the approach taken in [Shirley and Tuchman 1990]. There are two cases, illustrated in Figure 3. We can determine if a projected tetrahedron $\tilde{\mathbf{T}}$ is case *a)* or *b)* by examining the inverse matrix $\tilde{\mathbf{T}}^{-1}$. The columns of this matrix represent the four planes

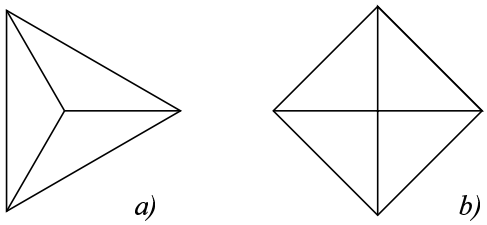

Figure 3: Two case for the screen space projection of a tetrahedron.

containing the faces of $\tilde{\mathbf{T}}$. We count the number of negative z values (third component) among these planes; case *a)* occurs when there are 1 or 3, case *b)* when there are 2. We don't worry about degenerate faces as these are not drawn by the hardware.

In case *a)* we draw 3 triangles and in case *b)* we draw 4 triangles. We store as vertex attributes z_{min} and z_{max}, representing the minimum and maximum z values that occur along a z-axis ray through each vertex. When interpolated, these attributes provide the z extent of the tetrahedron at each pixel. If a root does not occur in this range, then the surface will not be visible at the corresponding pixel.

4 Per Pixel Processing

For a given pixel $[x\ y]$, we must determine if a real z exists so that $\tilde{\mathbf{x}} = [x\ y\ z\ 1]$ satisfies Equation (3). In other words, we must find the roots of a degree d polynomial in z whose coefficients vary from pixel to pixel. At each pixel, we would like to compute the Bézier form of this univariate polynomial restricted to the interval $\left[z_p, z_q\right]$, where z_p and z_q are the interpolated values of z_{min} and z_{max} (from the previous section) representing the z extent of the interior of the tetrahedron at the current pixel. We use Bézier form to get optimal numerical conditioning[Farouki and Rajan 1987], and to allow a pixel shader an early exist test if all coefficients have the same sign.

Our goal is to find a univariate polynomial in Bézier form

$$\sum_{i=0}^{d} \binom{d}{i} (1-v)^{d-i} v^i \, a_i \;=\; 0, \tag{4}$$

that corresponds to (3) with x and y held constant and reparameterized so that $v \in [0,1]$ corresponds to $z \in \left[z_p, z_q\right]$. Note that each Bézier coefficient a_i will depend on pixel coordinates x and y. To find the coefficients a_i, we define

$$\mathbf{p} = \left[x\ y\ z_p\ 1\right] \quad \text{and} \quad \mathbf{q} = \left[x\ y\ z_q\ 1\right]. \tag{5}$$

so that a point in screen space can be written

$$\tilde{\mathbf{x}} \;=\; (1-v)\,\mathbf{p} \;+\; v\,\mathbf{q}.$$

If we plug this into Equation (3) the coefficients a_i can be written

$$\begin{aligned}
a_0 \;&=\; \mathbf{p}^{\beta_1} \cdots \mathbf{p}^{\beta_d} \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d}, \\
&\vdots \\
a_i \;&=\; \mathbf{q}^{\beta_1} \cdots \mathbf{q}^{\beta_i}\, \mathbf{p}^{\beta_{i+1}} \cdots \mathbf{p}^{\beta_d}\, \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d}, \\
&\vdots \\
a_d \;&=\; \mathbf{q}^{\beta_1} \cdots \mathbf{q}^{\beta_d}\, \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d}.
\end{aligned} \tag{6}$$

Next, we show how to compute these Bézier coefficients on the GPU in an efficient way.

4.1 Interpolating Coefficients

At each pixel we must evaluate Equations (6) given $\mathbf{p}$ and $\mathbf{q}$ derived from that pixel. The problem is how to pass the tensor $\tilde{\mathbf{B}}$ to the pixel shader. This data will be different for every tetrahedron rendered. That is, it will change for every 3 or 4 triangles that are drawn. The mechanism we use is to store a variant of the tensor $\tilde{\mathbf{B}}$ as attributes on the vertices of triangles.

The GPU will interpolate vertex attributes when rasterizing a triangle. The interpolation is done in a *perspectively correct* way. That is, vertex attributes are interpolated in non-linear fashion consistent with any projective foreshortening induced by the viewing transformation. However, if $w = 1$ for each vertex, then the interpolation will be linear. We take advantage of this linear interpolation to do useful work and to reduce the amount of data stored in vertex attribute memory.

This idea is based on the following observation. Each of the coefficients $a_0, \ldots, a_{d-1}$ (all but a_d) in equations (6) have a common factor of

$$\mathbf{p}^{\beta_d}\, \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d} \equiv \mathbf{p} \cdot \tilde{\mathbf{B}}. \tag{7}$$

This is a symmetric tensor of rank $(d-1)$ with $\binom{d+2}{d-1}$ unique elements. Suppose that $\mathbf{p} = r\mathbf{w}_0 + s\mathbf{w}_1 + t\mathbf{w}_2$, where each $\mathbf{w}_j = \begin{bmatrix} x_j & y_j & z_{minj} & 1 \end{bmatrix}$, are the vertices of a screen space triangle. It follows by linearity of dot products that

$$\mathbf{p} \cdot \tilde{\mathbf{B}} = r\, \left(\mathbf{w}_0 \cdot \tilde{\mathbf{B}}\right) + s\, \left(\mathbf{w}_1 \cdot \tilde{\mathbf{B}}\right) + t\, \left(\mathbf{w}_2 \cdot \tilde{\mathbf{B}}\right).$$

Therefore, we can determine the *partially collapsed* rank $(d-1)$ tensor $\mathbf{p} \cdot \tilde{\mathbf{B}}$ at each pixel by evaluating $\mathbf{w}_j \cdot \tilde{\mathbf{B}}$ at the vertices of a triangle and assigning the $\binom{d+2}{d-1}$ unique elements as vertex attribute data. The GPU will interpolate these values when rasterizing the triangle, giving us the unique elements of $\mathbf{p} \cdot \tilde{\mathbf{B}}$ at the current pixel. The advantages of this scheme are 1) we save space by passing a tensor of one less rank to the shader, and 2) we utilize the GPU interpolator do the initial tensor collapse computation in the process.

The pixel shader program uses the interpolated data to reconstruct the symmetric rank $(d-1)$ tensor $\mathbf{p} \cdot \tilde{\mathbf{B}}$ from its unique elements, as well as the points $\mathbf{p}$ and $\mathbf{q}$, see equation (5). Pixel coordinates x and y are determined by reverse mapping the contents of a hardware register (Shader Model 3.0 [Microsoft Corp 2006]) through the viewport transform, and the values z_p and z_q have been interpolated as vertex attribute data (Section 3). The coefficients $a_0, \ldots a_{d-1}$ are computed using Equations (6). These are implemented as a sequence of GPU native 4 element dot products.

It remains to compute the final coefficient a_d. To do this, we write point $\mathbf{q}$ as

$$\mathbf{q} \;=\; \mathbf{p} \;+\; \delta\, \mathbf{z},$$

where $\delta = \left(z_q - z_p\right)$, and $\mathbf{z} = [0\ 0\ 1\ 0]$. The coefficient a_d becomes

$$\begin{aligned} a_d \;&=\; (\mathbf{p} \;+\; \delta\, \mathbf{z})^{\beta_1} \cdots (\mathbf{p} \;+\; \delta\, \mathbf{z})^{\beta_d}\; \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d}, \\ &= \sum_{i=0}^{d} \binom{d}{i} \delta^i\, \mathbf{z}^{\beta_1} \cdots \mathbf{z}^{\beta_i}\; \mathbf{p}^{\beta_{i+1}} \cdots \mathbf{p}^{\beta_d}\; \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d}, \end{aligned} \tag{8}$$

a polynomial in δ (*The first and last terms of this summation are well-formed if we adopt the notational convention that, for a sequence of identical tensors, when the first summation subindex is greater than the second, those tensors are eliminated from the expression*). Note that the dot product of $\mathbf{z}$ with a tensor simply selects the third element. So the expression for a_d is a weighted sum of elements of tensors that we have already evaluated to compute $a_0 \ldots a_{d-1}$; with the exception of

$$\mathbf{z}^{\beta_1} \cdots \mathbf{z}^{\beta_d}\; \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d} \;=\; \tilde{\mathbf{B}}_{\underbrace{3, \ldots, 3}_{d}}.$$

This constant value must be included in the set of interpolated vertex attribute data. But it turns out that this value, along with three others are also needed to calculate the tangent plane to a point on the surface, described in Section 4.3.

Once we have determined the Bézier coefficients of Equation (4), we see if any zeroes that lie in the range $[0,1]$. The first thing we can do is an early-out test. If all of the a_i have the same sign then, by the convex hull property, there can be no roots in $[0,1]$, and the pixel shader can exit early. Terminating the pixel shader in this way can boost performance since the expense of unnecessary root finding can be avoided, see Figure 4. If we cannot exit the pixel shader, we must look for real roots. Our analytic root finding scheme is outlined in the next section.

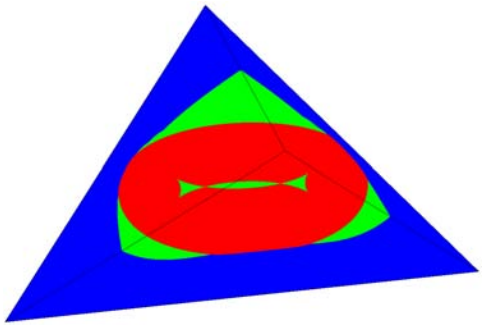

Figure 4: Torus model shaded by: blue) pixel culled by Bézier coefficient sign test, red) positive quartic discriminant, green) negative quartic discriminant.

4.2 Root Finding

There are basically two approaches to any root finding problem: iterative and analytic. Most of the literature on Bernstein-based root finding discusses the iterative technique [Farouki and Rajan 1988]. Analytic techniques, however, are better for pixel shader code as they have bounded computation. For polynomials of degree 2, 3 and 4 expressed in the power basis there are relatively simple closed form solutions. We will now see how to adapt these to the Bernstein basis.

To see this most clearly let us consider a degree 3 polynomial and compare it with a general homogeneous cubic polynomial in (x,w) (This use of the variables x and w is local to this section, and are not related to earlier uses.)

$$\begin{aligned} f(v) &= a_0 (1-v)^3 + 3a_1 (1-v)^2 v + 3a_2 (1-v) v^2 + a_3 v^3, \\ f(x,w) &= a_0 w^3 + 3a_1 w^2 x + 3a_2 w x^2 + a_3 x^3. \end{aligned}$$

So any roots (x,w) to the homogeneous polynomial (using the Bernstein basis coefficient values) will give roots of the original polynomial according to the formula $v = x/(x+w)$. Another way to look at this is to realize that the conversion between the power and Bernstein bases is simply the result of a homogeneous projective transformation in parameter space.

$$\begin{bmatrix} x & w \end{bmatrix} = \begin{bmatrix} v & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

This explains the initially surprising fact that the formulas for discriminants of polynomials are the same in terms of either Bernstein and power coefficients: the discriminant is simply an invariant under parameter space transformation.

We now turn our attention to finding stable solutions to homogeneous polynomials. The first step in conventional polynomial root finding is to "depress" the polynomial by translating it in parameter space to make a new polynomial in $[\tilde{x}\ \tilde{w}]$ with coefficients $\tilde{a}_i$ but where $\tilde{a}_{d-1}$ is zero. In our notation the translation amount for each degree d is the same: a_{d-1}/a_d. We can express the translation between the original $[x\ w]$ space and the depressed $[\tilde{x}\ \tilde{w}]$ space as a matrix product

$$\begin{bmatrix} x & w \end{bmatrix} = \begin{bmatrix} \tilde{x} & \tilde{w} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -a_{d-1} & a_d \end{bmatrix}$$

This works most of the time, but we can see that the matrix is very ill conditioned when a_d is small compared with a_{d-1}. There are, however, many other 2×2 matrices that we can use that will make the new coefficient $\tilde{a}_{d-1} = 0$. These may be projective transformations of parameter space rather than simple translations, but as long as their matrix is nonsingular (and hopefully very nonsingular) the process works. A little algebra will show that, given an arbitrary first row $[p\ q]$, a matrix that will depress the polynomial will be:

$$\begin{bmatrix} x & w \end{bmatrix} = \begin{bmatrix} \tilde{x} & \tilde{w} \end{bmatrix} \begin{bmatrix} p & q \\ -f_w(p,q) & f_x(p,q) \end{bmatrix}$$

Where the subscripts on the f denote partial derivatives. The determinant of this transformation is simply d times $f(p,q)$. So to find a transformation that is not singular we need to find a $[p\ q]$ that is *not* a root of the polynomial f. While finding a completely optimal $[p\ q]$ may be possible, we have had good results by the following simple choices: $[p\ q] = [1\ 0]$ (this generates the original translation), $[0\ 1]$ (this effectively reverses the order of the coefficients, solving for w/x instead of x/w, and then inverts the results), $[1\ 1]$ and $[1\ -1]$ (these do a 45 degree rotation in parameter space and rescue us when both a_d and a_0 are extremely small). The choice of which of these to use is based on which will give the largest value of $|f(p,q)|$. For the simple $[p\ q]$ values used here this leads to fairly simple decision code. (Of course a large determinant does not guarantee that the transformation is nicely conditioned, but our experience has shown that it is adequate.) To avoid possible singularities we must use at least one more $[p\ q]$ choice than the number of roots of the polynomial. For example, a cubic polynomial that has its three roots near $[1\ 0], [0\ 1]$ and $[1\ 1]$ could still be nicely depressed by the choice $[p\ q] = [1\ -1]$. (In fact, for quartics we really need five choices, although we haven't needed to implement that as yet.) Once the polynomial has been pre-conditioned by application of one of these transforms we then solve it by techniques based on [Blinn 2005] and the Ferrari technique described in [Herbison-Evans 1995]. The results are post transformed back to the original $[x\ w]$ space by the 2×2 transform. The conversion back to $[v\ 1]$ space (another 2×2 transform) can be merged with this.

4.3 Computing Surface Normals

If no real root $v \in [0,1]$ is found then the current pixel shader instance is terminated. Otherwise we take the smallest root v and shade the pixel by first computing the normal to a point on the surface $\tilde{\mathbf{x}} = (1-v)\,\mathbf{p} + v\mathbf{q}$ by collapsing the tensor $\tilde{\mathbf{B}}$ down to the tangent plane using

$$\mathbf{l}_{\beta_1} = \tilde{\mathbf{x}}^{\beta_2} \cdots \tilde{\mathbf{x}}^{\beta_d} \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d}.$$

Note that this expression is identical to Equation (3), with one less factor of $\tilde{\mathbf{x}}$, resulting in a covariant vector (a plane). By making the substitution $\tilde{\mathbf{x}} = \mathbf{p} + (v\,\delta)\,\mathbf{z}$, we can write the tangent plane as

$$\begin{aligned} \mathbf{l}_{\beta_1} &= (\mathbf{p} + (v\,\delta)\,\mathbf{z})^{\beta_2} \cdots (\mathbf{p} + (v\,\delta)\,\mathbf{z})^{\beta_d} \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d}, \\ &= \sum_{i=0}^{d-1} \binom{d-1}{i} (v\,\delta)^i\, \mathbf{z}^{\beta_2} \cdots \mathbf{z}^{\beta_{i+1}}\, \mathbf{p}^{\beta_{i+2}} \cdots \mathbf{p}^{\beta_d}\, \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d}, \end{aligned} \quad (9)$$

a polynomial in $(v\delta)$. Just as in (8), this is a weighted sum of previously computed tensors; with the exception of

$$\mathbf{z}^{\beta_2} \cdots \mathbf{z}^{\beta_d}\, \tilde{\mathbf{B}}_{\beta_1 \cdots \beta_d} = \tilde{\mathbf{B}}_{\beta_1, \underbrace{3, \ldots, 3}_{d-1}}. \quad (10)$$

This constant 4-tuple is added to the set of interpolated vertex attributes; note that the third element is the lone value needed to compute a_d from equation (8). Equations (8) and (9) are closely related and share many common factors that an implementation should take advantage of.

Example *We illustrate the procedure for the cubic surface case $d = 3$. We can think of the symmetric rank 3 tensor $\tilde{\mathbf{B}}$ as a 4-tuple of 4×4 matrices*

$$\tilde{\mathbf{B}} = \begin{bmatrix} \tilde{\mathbf{B}}_1 & \tilde{\mathbf{B}}_2 & \tilde{\mathbf{B}}_3 & \tilde{\mathbf{B}}_4 \end{bmatrix}.$$

For each of the three vertices $\mathbf{w}_j$, $j = 0,1,2$, of a screen space triangle we form a symmetric 4×4 matrix

$$\mathbf{G}_j = \begin{bmatrix} \mathbf{w}_j \cdot \tilde{\mathbf{B}}_1 & \mathbf{w}_j \cdot \tilde{\mathbf{B}}_2 & \mathbf{w}_j \cdot \tilde{\mathbf{B}}_3 & \mathbf{w}_j \cdot \tilde{\mathbf{B}}_4 \end{bmatrix}.$$

The following values are assigned as attributes of each vertex $\mathbf{w}_j$; the 10 scalar values on and above the diagonal of $\mathbf{G}_j$ that we label $t_0, \ldots, t_9$; the vector $[t_{10}\ t_{11}\ t_{12}\ t_{13}] = \tilde{\mathbf{B}}_{3,3}$ (the third row of $\tilde{\mathbf{B}}_3$, see Equation (10)); and the bounds $(t_{14}, t_{15}) = (z_{minj}, z_{maxj})$. The GPU will interpolate $t_0 \ldots t_{15}$ over the triangle, calling our pixel shader program with this input data.

The pixel shader then reconstructs, from the input data, the symmetric 4×4 matrix

$$\mathbf{H} = \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ t_1 & t_4 & t_5 & t_6 \\ t_2 & t_5 & t_7 & t_8 \\ t_3 & t_6 & t_8 & t_9 \end{bmatrix}$$

that represents the product $\mathbf{p} \cdot \tilde{\mathbf{B}}$ for the current pixel $[x\,y]$. We set $\mathbf{p} = [x\,y\,t_{14}\ 1]$, $\mathbf{q} = [x\,y\,t_{15}\ 1]$, $\mathbf{h} = [t_{10}\ t_{11}\ t_{12}\ t_{13}]$ and $\delta = (t_{15} - t_{14})$. The univariate cubic coefficients are computed

$$a_0 = \mathbf{p} \cdot (\mathbf{p} \cdot \mathbf{H}), \quad a_1 = \mathbf{q} \cdot (\mathbf{p} \cdot \mathbf{H}), \quad a_2 = \mathbf{q} \cdot (\mathbf{q} \cdot \mathbf{H}),$$

and

$$a_3 = \mathbf{p} \cdot (\mathbf{p} \cdot \mathbf{H}) + 3\,\delta\,(\mathbf{p} \cdot \mathbf{H})_3 + 3\,\delta^2 \mathbf{H}_{3,3} + \delta^3 \mathbf{h}_3.$$

If $a_0, \ldots, a_3$ have the same sign, we terminate processing. Otherwise, we find the roots of the polynomial

$$a_0\,(1-v)^3 + 3\,a_1\,(1-v)^2\,v + 3\,a_2\,(1-v)\,v^2 + a_3\,v^3,$$

in closed form. If no real root v is found in $[0,1]$, we terminate processing. Otherwise, taking the smallest root v, we compute the tangent plane as follows.

$$\mathbf{l} = \mathbf{p} \cdot \mathbf{H} + 2\,(v\delta)\,\mathbf{H}_3 + (v\delta)^2\,\mathbf{h}.$$

Note that $\mathbf{H}_3$ *in the above expression refers to the* 3^{rd} *row (or column) of* $\mathbf{H}$. *We map v from the local pixel interval* $[0,1]$ *the global interval* $[t_{14}, t_{15}]$ ($\equiv [z_p, z_q]$) *to get the output depth*

$$z = v\delta + t_{14}.$$

The tangent plane $\mathbf{l}$ *is transformed to world space (by* $\mathbf{M} \cdot \mathbf{l}$*), where the leading* 3*-tuple is normalized and the resulting surface normal is used for lighting calculations to determine the output color.*

In order to make the preceding example more clear, we have shown several redundant calculations. A good implementation would optimize these away. The actual encoding of the preceding example into shader code is straightforward due to the expressive power of HLSL.

5 Results

Figure 5 shows a collection of curved surfaces rendered in real time using our algorithm. The top row shows a collection of piecewise quadratic shapes; the cone, cylinder and sphere examples are defined by a single equation, but modeled here in a piecewise fashion. The extruded @ symbol was constructed using a technique similar to that of [Loop and Blinn 2005]. The bottom row shows a single cubic and two quartic shapes defined by Bézier tetrahedra.

The HLSL compiler aggressively eliminates common subexpressions and emits GPU assembler instructions (a device independent intermediate form that the GPU driver translates into machine code). The following table gives assembler instruction counts for various shader tasks.

Degree	*Coefficients*	*Roots*	*Shading*	*Total*
2	15	65	37	117
3	37	168	37	242
4	66	193	37	296

The meaning of the column headings are as follows: *Degree* is the polynomial degree of the Bézier tetrahedra being rendered; *Coefficients* refers to the calculation of the univariate Bézier coefficients at each pixel (step 1 of Section 1.2); *Roots* refers to finding the roots of the univariate polynomial (steps 2, 3, and 4); *Shading* refers to lighting calculations (step 5); and *Total* is the total number of assembler instructions in the shader.

Performance on a particular GPU is related to the number of pixel shader instances executed per frame. This will depend on screen resolution, and the number, size, and degree of the projected tetrahedra. Our tests were run on an nVidia GeForce 7800 GTX at a resolution of 640×480 where the objects depicted in Figure 5 roughly fill the screen, we see frames rates in excess of 1200 fps for a single quadratic tetrahedron, and approximately 500 fps for a quartic tetrahedron. As depth complexity increases, frame rate decreases. The quadratic bowling pins model in Figure 5 was designed to test moderate depth complexity, and we see roughly 75 fps for screen filling views. The animated blobs shown in Figure 1 runs at close to 300 fps.

6 Limitations and Future Work

The images shown here illustrate the utility of this algorithm. There are some concerns that we now discuss.

6.1 Numerical Precision

A significant portion of our computations are performed during vertex attribute interpolation. In Shader Model 3.0, vertex attributes used as *texture coordinates* are represented and operated upon as 32 bit floating point numbers. As a result, we have not experienced major problems with numerical precision induced by the hardware.

Significant precision loss in the determination of the univariate polynomial coefficients at each pixel can occur when the distribution of z values is skewed by the choice of *near* and *far* clipping planes used in the projection matrix. This problem can be avoided by using a unique projection matrix for each tetrahedron whose near and far planes match the z extent of the tetrahedron.

6.2 Self Intersections

The described technique works well for the surfaces illustrated here. The root finding is stable for all pixels where a single root represents the visible portion of the surface. It does experience some minor problems where there are double (or more) roots in the v polynomial. Sometimes numerical noise will cause this to be reported as a complex conjugate pair and be rejected in the real-root-finding process. This can occur in two situations: near silhouette edges (where a missed pixel exactly on the edge will not make an obvious artifact) and near self intersections (where a missed pixel will leave a visible hole in the object). The bottom right of Figure 5 shows a particularly difficult case. While it does have some artifacts, they are noticeable only in a small region. We regard the problem with self intersecting surfaces as a nuisance but not as a fatal flaw. We envision the typical use of this algorithm would be to render piecewise smooth surfaces that model some real-world smooth object. A designer is unlikely to make such a surface out of self-intersecting pathological pieces. Never the less, we are investigating better root finding algorithms and more precise interpolation techniques. These will be reported in [Blinn 2006]

6.3 Anti Aliasing

While we have not addressed anti-aliasing in this paper there are several techniques that we are experimenting with. Aliased edges can be of two types. First, the boundary intersection curve of the surface with a tetrahedral face could be left open. This can be handled by the anti-aliasing technique described for 2D curves in space described in [Loop and Blinn 2005]. Second, the silhouette curve of the surface must be anti-aliased. This curve is defined as the locus where the discriminant of the v polynomial is zero. Simple formulas for this discriminant for orders 2, 3, and 4 are given in [Blinn 2003], see pages 259, 262, and 267. Given that we can evaluate this discriminant and use the graphics hardware to find its gradient over pixel coordinates, a pixel unit distance function for the silhouette can be easily computed.

The real challenge of anti-aliasing is z ordering for α-blending. We could sort the tetrahedra by smallest z value, and present them to the GPU in back to front order. But this simple sorting approach does not guarantee correct results for overlapping tetrahedra.

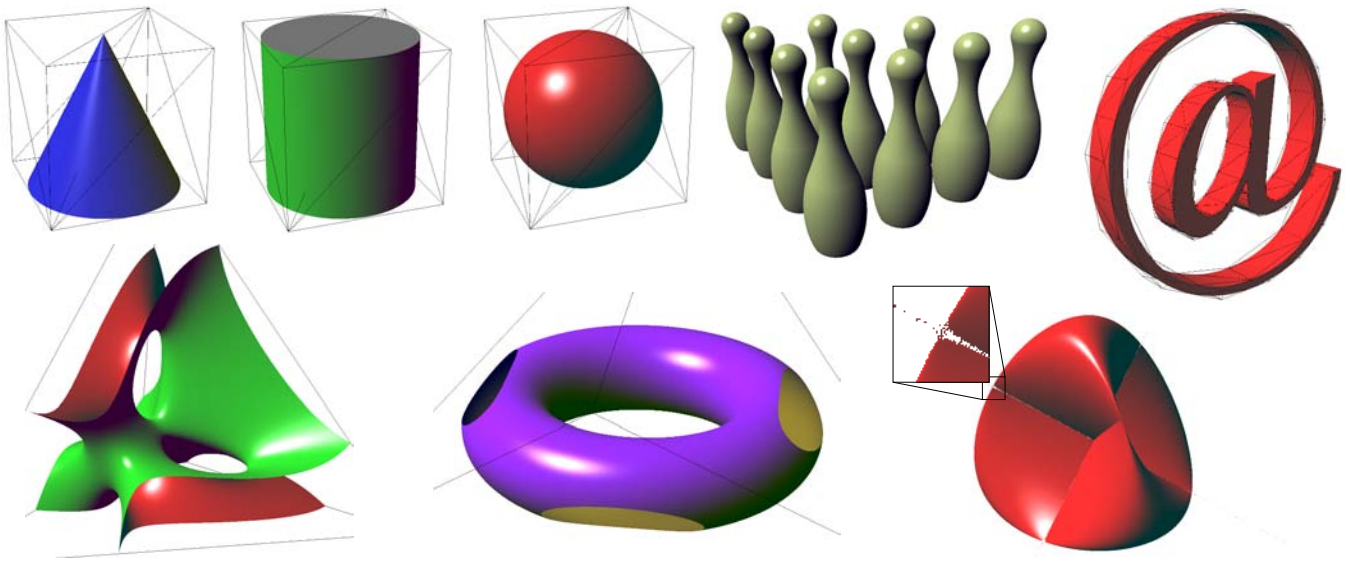

Figure 5: Some sample piecewise algebraic surfaces composed of Bézier tetrahedra and rendered using our technique.

References

BAJAJ, C. 1997. Implicit surface patches. In *Introduction to Implicit Surfaces*, Morgan Kaufman, J. Bloomenthal, Ed., 98–125.

BANGERT, C., AND PRAUTZSCH, H. 1999. Quadric splines. *Computer Aided Geometric Design 16*, 6, 497–525.

BLINN, J. 2003. *Jim Blinn's Corner Notation, notation, notation*. Morgan Kaufmann.

BLINN, J. 2005. How to solve a quadratic equation. *IEEE CG & A 25*, 6 (November/December), 76.

BLINN, J. 2006. Jim Blinn's Corner. *IEEE CG & A*.

BLYTHE, D. 2006. The Direct3D 10 System. *ACM Transactions on Graphics*. Siggraph Conference Proceedings.

DAHMAN, W. 1989. Smooth peicewise quadric surfaces. In *Mathmatical Methods in Computer Aided Geometric Design*, Acadmemic Press, T. Lyche and L. Schumaker, Eds., 181–194.

DE BOOR, C. 1987. B-form basics. In *Geometric Modeling: Algorithms and New Trends*, G. Farin, Ed. SIAM, 131–148.

FAROUKI, R. T., AND RAJAN, V. T. 1987. On the numerical condition of polynomials in bernstein form. *Computer Aided Geometric Design 4*, 191–216.

FAROUKI, R. T., AND RAJAN, V. T. 1988. Algorithms for polynomials in bernstein form. *Computer Aided Geometric Design 5*, 1–26.

GRAY, K. 2003. *The Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press.

GUO, B. 1995. Quadric and cubic bitetrahedral patches. *The Visual Computer 11*, 5, 253–262.

HADWIGER, M., SIGG, C., SCHARSACH, H., BÜHLER, K., AND GROSS, M. 2005. Real-time ray-casting and advanced shading of discrete isosurfaces. In *Eurographics*, Blackwell Publishing, M. Alexa and J. Marks, Eds., vol. 24.

HECKBERT, P. 1984. The mathematics of quadric surface rendering and *soid*. Tech. Rep. 3-D Technical Memo 4, New York Institute of Technology. available from: `www-2.cs.cmu.edu/~ph`.

HERBISON-EVANS, D. 1995. Solving quartics and cubics for graphics. In *Graphics Gems V.* , AP Professional, Chestnut Hill MA.

HOSCHEK, J., AND LASSER, D. 1989. *Fundamentals of Computer Aided Geometric Design*. A K Peters. English translation by L Schumaker, 1993.

LOOP, C., AND BLINN, J. 2005. Resolution independent curve rendering using programmable graphics hardware. *ACM Transactions on Graphics 24*, 3, 1000–1009. Siggraph Conference Proceedings.

MAHL, R. 1972. Visible surface algorithms for quadric patches. *IEEE Trans. Computers* (Jan.), 1–4.

MICROSOFT CORP. 2006. Direct3d 9 reference. In *In Direct3D 9 graphics*. `http://msdn.microsoft.com/directx`.

RAMSHAW, L. 1989. Blossoms are polar forms. *Computer Aided Geometric Design 6*, 4 (November), 323–358.

SEDERBERG, T. W., AND ZUNDEL, A. K. 1989. Scan line display of algebraic surfaces. *Siggraph 1998 Conference Proceeding 23*, 3.

SEDERBERG, T. 1985. Piecewise algebraic surface patches. *Computer Aided Geometric Design 2*, 1-3, 53–60.

SHIRLEY, P., AND TUCHMAN, A. A. 1990. Polygonal approximation to direct scalar volume rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics*, vol. 24, 63–70.