

# Real-Time Algebraic Surface Visualization

Johan Simon Seland<sup>1</sup> and Tor Dokken<sup>2</sup>

<sup>1</sup> Center of Mathematics for Applications, University of Oslo [johans@cma.uio.no](mailto:johans@cma.uio.no)

<sup>2</sup> SINTEF ICT [tor.dokken@sintef.no](mailto:tor.dokken@sintef.no)

**Summary.** We demonstrate a ray tracing type technique for rendering algebraic surfaces using programmable graphics hardware (GPUs). Our approach allows for real-time exploration and manipulation of arbitrary real algebraic surfaces, with no pre-processing step, except that of a possible change of polynomial basis.

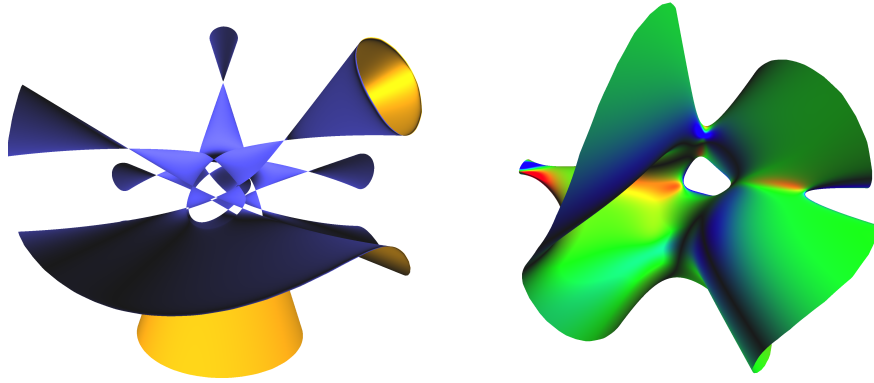
The algorithm is based on the blossoming principle of trivariate Bernstein-Bézier functions over a tetrahedron. By computing the blossom of the function describing the surface with respect to each ray, we obtain the coefficients of a univariate Bernstein polynomial, describing the surface's value along each ray. We then use Bézier subdivision to find the first root of the curve along each ray to display the surface. These computations are performed in parallel for all rays and executed on a GPU.

**Key words:** GPU, algebraic surface, ray tracing, root finding, blossoming

## 1.1 Introduction

Visualization of 3D shapes is a computationally intensive task and modern GPUs have been designed with lots of computational horsepower to improve performance and quality of 3D shape visualization. However, GPUs were designed to only process shapes represented as collections of discrete polygons. Consequently all other representations have to be converted to polygons, a process known as *tessellation*, in order to be processed by GPUs. Conversion of shapes represented in formats other than polygons will often give satisfactory visualization quality. However, the tessellation process can easily miss details and consequently provide false information.

The qualitatively best shape visualization is currently achieved by *ray tracing*; for each pixel in the image plane, straight lines from the center of projection through the pixel are followed until the shape is intersected. In these points shading is calculated and possibly combined with shading information calculated from refracted and reflected continuations of the line. This process is computationally intensive and time consuming, and in general it is not possible to use ray tracing to achieve interactive frame rates.



**Fig. 1.1.** A quintic surface with 31-double points, also known as a Dervish (left), and a randomly generated degree 4 surface with curvature visualization (right). Both surfaces above are rendered at interactive frame rates. We emphasize that no tessellation has taken place.

For a function  $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ , an implicit surface can be defined by the level set of the equation  $f(x, y, z) = c$ , where  $x, y, z, c \in \mathbb{R}$ . By reordering the terms, we can, without loss of generality, just consider the zero-set of the function, e.g.  $f(x, y, z) - c = 0$ . For this work, all functions and surfaces are considered to be real-valued. If the function  $f$  is an algebraic polynomial, the resulting surface is called an *algebraic surface*. Such surfaces can easily describe intricate, smooth shapes with varying topology, and also allow for the interpolation and blending of such surfaces. Their polynomial nature also makes it easy to find analytic directional derivatives, normals and curvature. The standard example of an algebraic surface is that of a sphere with radius  $r$  centered at the origin:

$$x^2 + y^2 + z^2 - r^2 = 0.$$

This paper proposes a method for high quality, high performance, ray tracing type rendering of algebraic surfaces using GPUs. The approach guarantees that the topology of the visualized surface is correct within pixel accuracy. The current prototype can be used for real-time visualization of algebraic surfaces of total degree five, the limitation being the current number of registers in each fragment processor and not the floating point capabilities of the GPU.

The approach aims at algebraic surfaces of degree 3 and higher, as conversion of these to high quality tessellations is a very challenging task. Such conversion requires detailed knowledge of the topology of the algebraic surface. For quadratic algebraic surfaces the traditional classification into spheres, cylinders, cones, ellipsoids, hyperboloids, as well as the degenerate cases aids the process of generating tessellations. Cubic algebraic surfaces can be classified into 45 classes [19], but for higher degrees no such useful classification exists. However, our method is applicable also for quadratic and cubic algebraic surfaces.

This paper is organized as follows: We begin with an overview of related work in Sect. 1.2. Thereafter we give a short, informal description of our method, focusing on how the GPU pipeline can be used for ray tracing. In Sect. 1.4 we describe details of our GPU implementation, as well as performance benchmarks of our method compared to state-of-the-art CPU ray tracing for algebraic surfaces. To avoid adding too much mathematical detail to Sects. 1.3 and 1.4, we present the basic ideas first, using forward references to equations, which are presented in Sect. 1.5 together with more mathematical background. Finally we conclude and present some thoughts on future work in Sect. 1.6.

## 1.2 Related Work

Using the GPU for non-graphical simulations is an emerging field, called GPGPU (General-Purpose Computing on Graphics Processing Units). A survey of recent activity is given by Owens et. al. [30]. Chapter [8] in this book details activity at Sintef and programming techniques for GPUs.

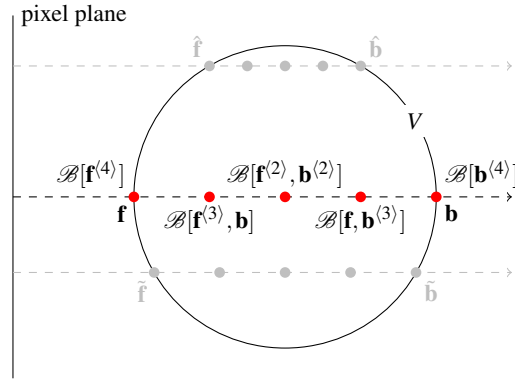
Since programming GPUs is rather complex, several systems have been developed to use the CPU for automatic generation of shader code. A group at the University of Waterloo has developed Sh [26] which is a metaprogramming language which uses the syntax of C++ for the development of shader programs for different generations of GPUs. A group at Stanford have developed Brook for GPUs [3] which is designed for using the GPU as a general stream processor.

Ray tracing on GPUs has seen a flurry of activity in recent years. A CPU simulation of GPU ray tracing was implemented by Purcell et. al. [31] when programmable graphics hardware was starting to appear. Later the master theses by Karlsson and Ljungsted [21] and Christen [6] describe concrete implementations running on production hardware. Thrane and Simonsen [40] compare acceleration structures for GPU based ray tracing, and conclude that bounding volume hierarchies (BVH) is the best technique. Carr et. al. [5] accelerate the BVH approach by storing it on the GPU as a *geometry image* [17].

Displacement mapping on GPUs has also been implemented by using ray tracing like techniques. Hirche et. al. [18] displace a triangle mesh along the normal direction by replacing base mesh triangles with triangles that cover the area on the screen that would be affected by the displaced base mesh. Donnelly [10] uses sphere tracing to effectively implement bounded displacement mapping using fragment shaders.

There has also been a few recent publications about dedicated hardware for ray tracing: Carr et. al. [4] describe the necessary modifications to existing GPU designs to support ray tracing. Schmittler et. al. [35] describe a dedicated FPGA chip. However none of these designs are yet in commercial production. Using dedicated graphics hardware for ray tracing is however not a new idea, Olano et. al. [28, 29] used the PixelFlow system developed at the University of North Carolina, to render spherical primitives in the mid nineties.

GPU assisted rendering of implicit surfaces (metaballs) has been demonstrated by Uralsky [41], using the geometry shader which will be made available on next



**Fig. 1.2.** For each ray segment inside the view volume  $V$ , we find the front ( $\mathbf{f}$ ) and back ( $\mathbf{b}$ ) vectors in barycentric coordinates. Then we use blossoming to find the coefficients of the univariate Bernstein-Bézier function along each ray. Since each ray corresponds to a pixel, and there is no interdependency between rays we can calculate all rays in parallel on a GPU.

generation (DX10) GPUs. However, his approach uses the GPU to tessellate the surface to discrete geometry in the classical way, by using a marching-cubes like algorithm, whereas our method avoids the tessellation step completely. Rendering of implicit curves using GPUs has been demonstrated by Loop and Blinn [22]. Their approach was based on rendering the inside of a curve defined over a triangle for use in font rendering. Our work can be seen as an extension of their approach to surfaces. During the later stages of our research, Loop and Blinn [23] published their extension to surfaces, which is very similar to our approach. However, they use analytic root finding algorithms, thereby limiting the maximum degree of surfaces they can visualize to general quartic surfaces, as it is not possible to analytically solve general equations of degree 5 and higher.

One existing comparable software package is also worth mentioning, namely the package Surfex [20], which uses the (CPU-based) ray tracer SURF [12] to visualize algebraic surfaces. We will provide some run-time comparisons with the SURF ray tracer in Sect. 1.4.5.

### 1.3 Overview of the Algorithm

Our algorithm is based on the classical ray tracing approach, where rays pass from the eye, and we wish to find the location of each ray's first intersection with the surface. For a given algebraic surface  $f(x, y, z) = 0$  of total degree  $d$ , we are only interested in intersections inside a view volume  $V$ . Hence, it is natural to only consider the segment of each ray inside  $V$ . Let  $\mathbf{f}$  and  $\mathbf{b}$  represent the front and back coordinate of the ray as it enters and leaves  $V$ . The segment is then given by the parametrization  $\mathbf{p}(t) = (1-t)\mathbf{f} + t\mathbf{b}$ ,  $t \in [0, 1]$ . We combine this parametrization with the equation of the algebraic surface, yielding a univariate polynomial  $g(t) = f(\mathbf{p}(t))$  of degree  $d$ .

Now the problem of finding the closest intersection to the eye point, can be reformulated to finding the smallest  $t \in [0, 1]$  such that  $g(t) = 0$ .

Our algorithm is run fully on a GPU, and heavily exploits the floating point power and programmable capabilities of modern GPUs. Although we use the GPU to render graphics, our approach does not use the GPU pipeline in the classical way. In fact, our use of the graphics pipeline is more closely related to GPGPU techniques than to classical computer graphics. In the following, we give a short summary of the rendering pipeline of a GPU, highlighting just the stages relevant for our method.

- To initiate rendering, the CPU sends geometric primitives to the GPU as a set of triangles.
- Each vertex is thereafter processed in the *vertex processor*. The vertex processor may change the position of a vertex and also set up other variables that will be interpolated across the triangle (e.g., normals, colors etc.).
- Thereafter the *rasterizer*-stage generates a set of fragments (meta-pixels) from the associated geometry.
- The final stage is the *fragment processor*, which calculates the final color and depth of the pixel. The fragment processor cannot change the position of the pixel, but it may discard the pixel if it is not of interest.

Both the vertex and fragment processors can be programmed by the use of *shading languages*, and such programs are called *shaders*. High end GPUs are highly parallel and typically have 6-8 *vertex pipelines* and 16-48 *fragment pipelines* operating in parallel.

Our use of the GPU can then be summarized as follows

1. Issue *spanning geometry* to the GPU. The spanning geometry defines our view volume  $V$ , and initiates rendering.
2. Let the GPU rasterize the spanning geometry into a set of fragments, yielding the coordinates  $\mathbf{f}$  and  $\mathbf{b}$  for each ray.
3. Find the univariate polynomial,  $g(t)$ , representing the value of the algebraic surface along each ray.
4. Find the smallest  $t$  such that  $g(t) = 0$ . If no zeros are found, we discard the fragment.
5. For the surviving fragments, calculate the final depth and color by using information from the algebraic surface.

Steps 3 to 5 above are executed in the fragment processor of the GPU, and require the activation of a special shader program, which is outlined in Listing 1.1, and discussed further in Sect. 1.4.

As GPUs are limited to single precision floating point arithmetic, extreme care has to be taken with respect to representation formats and algorithmic components to guarantee the quality of the ray tracing. Our approach uses Bernstein bases both for representing the (trivariate) algebraic surfaces, see Sect. 1.5.1, and for representing the (univariate) polynomials describing the intersection of the ray segments of interest with the algebraic surface, see Sect. 1.5.2. The choice of the Bernstein bases

**Listing 1.1.** The main steps of our fragment shader. This code is executed in parallel for all rays, and run completely on a GPU.

```

uniform int degree ;

void main() {
    vec4 f = barycentric_front ();
    vec4 b = barycentric_back ();
    float c[degree+1]; // Control points
    for(int i = 0; i < degree; ++i)
        c[i] = blossom(f, b, i);
    if (number_of_sign_changes(c) == 0)
        discard ;
    float t = find_leftmost_zero(c);
    vec4 p = (1-t)*f + t*b;
    gl_FragColor = shade_and_light(p);
    gl_FragDepth = to_cartesian(p).z;
}

```

is based on its numerical superiority compared to other polynomial bases for algorithms run in single precision arithmetic. It can be shown that the Bernstein basis is optimal both for univariate and multivariate polynomials, see[14, 24]. Some more details and numerical examples are given in Chapter [9] of this book.

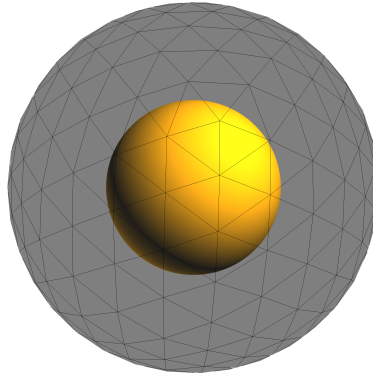
Another advantage is the possibility of representing coordinates in *barycentric coordinates* (see Sect. 1.5). This allows us to use a technique know as *blossoming* (see Sect. 1.5.3) to find an expression for the surface values along each ray (Item 3 above). Blossoming can be seen as a generalization of the *de Casteljau algorithm*, which is used for evaluating Bézier functions. Both of these algorithms are numerically very stable and effective. Barycentric coordinates over a domain in  $\mathbb{R}^3$  are represented as 4-component vectors, which fit neatly on GPUs, as they are designed to operate on 4-component vectors.

The de Casteljau algorithm is however not optimal for evaluating a polynomial, as it uses  $\mathcal{O}(d^2)$  floating point operations, whereas Horner's scheme just uses  $\mathcal{O}(d)$  operations for an evaluation. However, numerical errors are much more likely to occur using Horner's scheme which might lead to visual artifacts.

## 1.4 GPU Implementation

We will now examine in detail how we have implemented our algorithm on programmable GPUs, using the OpenGL Shading Language. Other shading languages such as Cg or HLSL are equally well suited, since there are only minor differences between them.

Some more terminology from shader programming is necessary to describe our method. Shader programs are written in a C-like programming language, and com-



**Fig. 1.3.** The view volume (black edges) is all that is sent to the GPU, and the rasterization of the view volume geometry initiates our ray tracing type rendering.

piled at run-time by the graphics driver. The host CPU may set some variables of the shader program at run-time. These are shared by all the pipelines of the GPU and are called *uniform variables*, since they have a uniform value over each primitive. Communication between the vertex and fragment shader programs takes the form of *varying variables*, which are interpolated (and thus varying) across geometric primitives. Uniform variables can only be read by the shader program and can be seen as global constants. The vertex processor may only write varying variables, and the fragment processor can only read varying variables. There can also be local variables for each vertex/fragment which the shader program can read and write at will. However their number is limited and there exist no stack or heap memories, so recursion and pointers are not available. For an in-depth overview of GPU programming see any book on shading languages, e.g. Rost [34] or Fernando and Kilgard [15].

This section uses forward references to equations that will be detailed in Sect. 1.5, as all mathematical prerequisites are collected there.

#### 1.4.1 Issue spanning geometry

To initiate our algorithm, we must send geometry encompassing the barycentric domain to the GPU. Theoretically any convex shape will suffice, but in practice shapes for which it is trivial to calculate the barycentric front and back coordinates for each ray are best suited.

The easiest spanning geometry of all is the tetrahedron, where the barycentric coordinates can be encoded as the color property of each vertex. Linear interpolation of color across each face will then yield the correct coordinate. However, visualizing an algebraic surface inside a tetrahedron clips the surface in an unnatural way, and gives a poor impression of the surface to a human observer. Instead, we have

found the sphere to be a good spanning geometry. By issuing a moderately tessellated sphere and linearly interpolating the analytic normal at each vertex over each face, and thereafter using (1.3) to get the geometric normal (scaled by the radius) yields a good approximation to the barycentric coordinate at a pixel. Figure 1.3 above illustrates how the spanning geometry corresponds to the view volume.

To calculate the back coordinate  $\mathbf{b}$ , several methods can be used. We usually render the same bounding geometry, using front face culling in a separate render pass, and store the value in a texture. For simple shapes, like a sphere, it is possible to use standard intersection equations to calculate  $\mathbf{b}$ . It is also possible to set up tetrahedrons such that both coordinates may be calculated in the same pass, as demonstrated by Weiler et. al. [42] or Wylie et. al. [43].

### 1.4.2 Multivariate blossoming on GPUs

The most computationally demanding subtask of the visualization is the calculation of the coefficients of the curve representing the value of  $f$  along each ray. We find these coefficients by blossoming and form an algorithm based on (1.9).

If recursion had been supported by the GPU, we could have implemented (1.9) directly. However, current generation GPUs do not support recursion, so for a given degree  $d$ , we use the CPU to unroll the recursion and end up with a long expression (of shader source code) representing the blossom, and use uniform variables to represent the coefficients of the surface. Since the blossom is recalculated for every frame, we can easily modify the surface by altering the coefficients, allowing for real-time shape manipulation.

As the degree of the surface increases, the complexity of the generated shader source code increases dramatically, and we have triggered several bugs in the shader compilers while developing this algorithm. Higher degrees also require more registers for intermediate calculations, and this is the limiting factor on the degree of surfaces we are able to visualize.

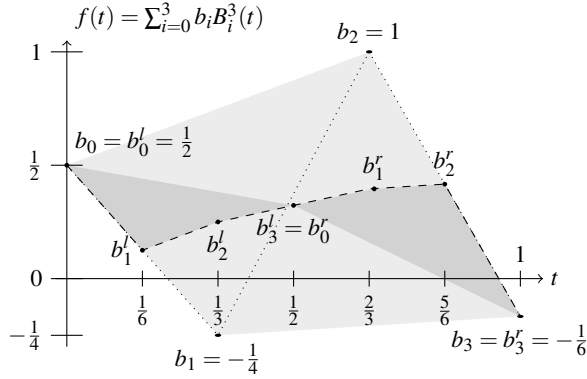
DeRose et. al. [7] describe optimizations to the recursive approach for calculating the blossom. Relevant to our method is to reuse partial blossom calculations, as we calculate the control points. We implemented this on the GPU, but the added bookkeeping actually made our algorithm slower, even if we were able to reduce the amount of floating point computations, demonstrating the performance penalty of executing branching code on GPUs.

### 1.4.3 Polynomial root finding on GPUs

Root finding for polynomials in Bernstein form has been explored in detail by Spencer [39]. Recent work by Mørken and Reimers [27] demonstrates a totally convergent root finding algorithm with quadratic convergence based on knot insertion of B-splines. Unfortunately, modern root finders require data structures such as linked lists that are not yet suitable for GPUs.

Because of its simple data structure, we decided to use a heuristic method based on recursive subdivision of the coefficients, as described by Schneider [36]. The idea





**Fig. 1.4.** A cubic Bézier curve with its original control polygon (dotted) and the control polygon after one iteration of Bézier-subdivision (dashes). The results of the subdivision are two new sets of control points, each covering half of the interval of the original control polygon. Observe that the dotted control polygon changes sign more often than  $f(t)$ . This false crossing vanishes after one iteration of subdivision. Also note that the curve is completely contained within the convex hull of the control points.

is to recursively subdivide the polynomial in half by using the de Casteljau algorithm (see Sect. 1.5.2). At each level of recursion we count the number of sign changes in the coefficients, and if both sets of coefficients contain sign changes we choose to continue to subdivide the leftmost of them, while storing the other one as we might need to unwind the recursion if the first one turns out to be a false zero. If only one set of coefficients has sign changes, we of course choose this one for further subdivision. This effectively uses the Bézier convex hull property to ensure that a zero is within an interval given by the coefficient endpoints. Interval arithmetic has been used successfully for ray tracing in the past, as demonstrated by Snyder[38] and Duff [11].

Rockwood et. al. [33] present a variant of the above, where the Bézier curve is subdivided at the leftmost sign change of the control polygon. While marginally faster than Schneider’s method, it has some problems near singularities when executed on a GPU, leading to visual artifacts.

Since it could happen that there is a false zero (as in Fig 1.4), we must be able to unwind the recursion and choose another interval to subdivide. On a regular CPU this would be trivial, but as the GPU lacks a stack we store the other potential coefficients and the recursion level, and when we detect a false zero we unroll to the last set of potential coefficients. The number of possible false zeros is bounded by the degree of the surface, so we know in advance how many sets of coefficients we might need to store. We terminate the recursion when we can decide that there is no root (no sign changes of the coefficients), or when we reach a fixed level of subdivisions.

When implementing CPU-based root finders, it is usually effective to have early termination criteria, such as the “flatness” test described by Schneider. Since such

tests lead to more complex branching structures, we have found that such tests actually hamper performance on current generation GPUs.

Since we only need to find the roots of univariate polynomials, where the derivatives can easily be computed, one might believe that Newton-like methods would perform extremely well. However, we are plagued with the classical problem of such methods; a good initial value must be found in order for the method to converge. Using the control polygon's first intersection as the initial value for a naive Newton iteration yields correct zeros for the majority of cases, and more than triples the FPS quoted in Table 1.1. However, the relatively few incorrect zeros yield glaring visual artifacts and a wrong topology of the surface. We believe that significant speed increases are possible by using hybrid Newton-subdivision methods when dynamic branching of GPUs improves.

The convergence of the coefficients of a Bézier curve is  $\mathcal{O}(h^2)$ , where  $h$  is the interval length. If the original Bézier curve was defined over the interval  $[0, 1]$ , ten levels of subdivision will give an interval length of  $2^{-10} = \frac{1}{1024}$ , with the maximal distance between the control polygon and the curve reduced by a factor of  $2(\frac{1}{1024})^2 \approx 0.95367 \times 10^{-6}$ . This is approximately the accuracy of numbers in single precision floating point format used by the GPU. Consequently the control polygon will be near indistinguishable from the curve. A typical displayed image is currently around  $1000 \times 1000$  pixels, thus ten levels of subdivision will give the same resolution orthogonal to the image plane as in the image plane.

#### 1.4.4 Curvature and lighting calculations

When the intersection has been found by the root finder, we know its position  $t$  within the univariate parameter domain. The resulting position in the trivariate domain can then easily be found by calculating  $\mathbf{u} = (1-t)\mathbf{f} + t\mathbf{b}$ , where  $\mathbf{f}$  and  $\mathbf{b}$  are the front and back vectors, respectively. Furthermore we can use the blossom (see (1.11)) to calculate the directional derivatives and obtain the (barycentric) normal vector of the surface. Various per-pixel lighting models based on the normal (or higher order derivatives) can then be trivially implemented. This generally works very well for most surfaces, however, when all the directional derivatives vanish the equations break down, as illustrated in Fig. 1.5.

Calculating the mean curvature of an implicit surface is possible, although the formulas become quite complex as they involve all second derivatives of the surface. However, the blossom can be used to find all these terms, and repeated use of (1.12) gives us the terms necessary to calculate the curvature, as illustrated by the right surface in Fig. 1.1. The performance impact of this curvature calculation is negligible in comparison to regular shading.

#### 1.4.5 Performance

We have implemented our algorithm using C++ and the OpenGL Shading Language using Nvidia 6000 and 7000-series hardware. Current ATI drivers have problems compiling our shader-programs, but we believe that once their drivers improve, our



**Fig. 1.5.** A quartic surface where all first order derivatives vanish, leading to a breakdown in the lighting equation (which is based on calculating the normal by taking the cross product of derivatives). This breakdown is clearly visible in the left image, where the shape looks flat and uninteresting. The right image shows the curvature field, clearly depicting the singularity which is invisible in the first image.

algorithm will be equally well (or better) suited for execution on ATI hardware, as present day ATI GPUs are better at dynamic branching.

For speed comparisons, we have compared rendering speeds of the same surface with the SURF ray tracer and our GPU implementation. We have benchmarked using surfaces with the highest known number of ordinary double points (singularities) for a given degree[1], thus stressing the root finder. We have also tested with random surfaces, where coefficients are in the range  $(-1, 1)$ . All benchmarks have been executed on an AMD X2 4400 CPU with a Nvidia Geforce 7800 GT GPU, running Windows XP and Nvidia driver version 91.31. Moreover, both ray tracers were set up to use a single light source, the same resolution ( $966 \times 892$ , the viewport of our prototype implementation when running in full-screen) and no anti-aliasing. The rendered surfaces also fill approximately the same number of pixels and the surfaces are oriented in the same direction. We emphasize that there is a strict correspondence between the resolution and the frame rate, as performance is directly affected by the number of pixels rendered. Since Loop and Blinn [23] does not provide any table of performance, a direct comparison is not possible. However, they use a tensor product formulation of the blossom, which has high computational complexity ( $\mathcal{O}(d^4)$ ), but leads to very effective shader source code (a series of nested multiply-and-add (MAD) operations). The recursive approach has computational complexity  $\mathcal{O}(d^3)$ , but is not as GPU friendly as the tensor formulation. For lower degrees it is apparent that using the tensor product formulation is more effective, the crossing point is probably around degree 5, for which [23] is no longer applicable as they use an analytic root finder, which only exist in general for degree 4 and lower.

**Table 1.1.** Speed comparisons of the CPU based ray tracer SURF and our GPU implementation. Resolution is  $966 \times 892$  for all benchmarks. The quoted GPU model is Nvidia 7800 GT. The named surfaces are the surfaces with the highest known number of singularities for their degree.

Surface	Degree	SURF (FPS)	GPU (FPS)	Speedup	Dual GPU (FPS)	Speedup
Sphere	2	2.6	60.0	23.1	108.0	41.5
Cayley	3	1.3	22.1	16.9	38.0	29.2
random	3	1.3	27.6	21.2	45.0	34.6
Kummer	4	1.1	9.2	8.3	18.2	16.5
random	4	1.0	12.2	12.2	22.3	22.3
Dervish	5	1.2	3.2	2.7	6.4	5.3
random	5	0.9	3.9	4.3	7.6	8.4

SURF is specially designed for rendering algebraic surfaces, and is significantly faster for such surfaces than the better known ray tracer POV-Ray. We have chosen to only benchmark against the fastest CPU solution available.

Table 1.1 summarizes our findings. We observe that the GPU based approach is significantly faster than SURF, but the difference decreases as the degree increases. The quintic Dervish surface (illustrated in Fig 1.1 (left)) is just 2.7 times faster on the GPU, while our algorithm is an order of magnitude faster for quartic surfaces. Since the computational load increases with each degree, one should expect the difference to be larger. However, the problem of a highly parallel GPU, is that the pixel pipelines must be synchronized, thus making early termination and adaptivity in the root finder almost useless. If one pixel requires recursion to the bottom, all pixels in the pipeline must wait until it is finished. Since higher degrees require more computations, this penalizes higher degrees more than the lower ones, as more computations must be executed while the rest of the pixels are waiting. Future GPUs are expected to improve at dynamic branching, and we expect huge speed increases when more adaptive root finding algorithms become available.

SURF is also able to exploit sparse polynomials, which our GPU approach fails to do. This is especially apparent for random surfaces, where all coefficients are non-zero. SURF is actually slower on a random quartic surface than on the relatively sparse quintic Dervish surface. In general, for our GPU approach, we observe little difference in rendering speed between surfaces of the same degree, but random surfaces are in general a little bit faster than the ones quoted in Table 1.1, as the surfaces in the table are chosen for their complexity. While sparse polynomials are important for studying algebraic surfaces, industrial use of algebraic surfaces will most often address surfaces with most coefficients non-zero. Consequently for industrial users the timings for sparse polynomials have little interest, while they are relevant within algebraic surfaces research.

As there is no interdependency between render passes or neighboring pixels, our algorithms are very well suited for execution on multi-GPU based systems, such as ATI CrossFire or Nvidia SLI. We effectively double the performance when using two Nvidia 7800 GT cards in *Split Frame Rendering* (SFR) mode.

In conclusion we emphasize that our GPU approach is still more than twice as fast as SURF for a quintic surface. For lower degrees the GPU based approach is almost an order of magnitude faster than the CPU based approach. In addition the CPU is left free to do other tasks while rendering. However, to effectively render models represented by a large number of piecewise algebraic surfaces, one would need to render a large number of primitives, probably including efficient culling strategies, and rendering speed must be increased further for this to be feasible in real time. The current rendering speeds for quartic and quintic surfaces can however hardly be called real time, but it is still possible to interactively alter and explore the surface, and future GPUs are expected to increase speeds even further.

### 1.5 Bernstein-Bézier Techniques

This section presents the necessary mathematical background material to understand our method. The material on blossoming is quite abstract, and comes from spline and approximation theory. However, a deep understanding of this material is not crucial to understand our method. The interested reader may consult Ramshaw [32] or Farin [13] for good and thorough introductions to the material. Statements are given without proofs, as they can be found in the above texts.

For compactness, we use standard multi-index notation. Thus for tuples  $\mathbf{j} = (j_1, \dots, j_n)$ , we let  $\mathbf{j} \pm \mathbf{k} = (j_1 \pm k_1, \dots, j_n \pm k_n)$ ,  $|\mathbf{j}| = \sum_{i=1}^n j_i$ ,  $\mathbf{j}! = \prod_{i=1}^n j_i!$  and  $\mathbf{x}^{\mathbf{j}} = \prod_{i=1}^n x_i^{j_i}$ . In the following,  $\mathbf{x}$  will always denote a point in Cartesian coordinates, while  $\boldsymbol{\beta}$  denotes a point in barycentric coordinates.

#### 1.5.1 The Bernstein basis

For the space of real polynomials of total degree  $d$ ,

$$P_{d,s} := P_d(\mathbb{R}^s) = \left\{ p(\mathbf{x}) = \sum_{|\mathbf{j}| \leq d} c_{\mathbf{j}} \mathbf{x}^{\mathbf{j}} : c_{\mathbf{j}} \in \mathbb{R} \right\}, \tag{1.1}$$

it is well known that they can be expressed in the Bernstein-Bézier (BB) basis, which is preferable for use in computations since it is more numerically stable.

The Bernstein basis functions  $B_{\mathbf{i}}^d$  of degree  $d$  can be expressed as:

$$B_{\mathbf{i}}^d(\boldsymbol{\beta}) = \frac{d!}{\mathbf{i}!} \boldsymbol{\beta}^{\mathbf{i}}, \quad |\mathbf{i}| = d, \quad |\boldsymbol{\beta}| = 1. \tag{1.2}$$

Here,  $\boldsymbol{\beta} = \boldsymbol{\beta}(\mathbf{x}) = (\beta_1(\mathbf{x}), \dots, \beta_{s+1}(\mathbf{x}))$  denotes the barycentric coordinate of a point  $\mathbf{x} \in \mathbb{R}^s$  with respect to a set of base points,  $(\mathbf{v}_1, \dots, \mathbf{v}_{s+1}) \in \mathbb{R}^s$ , which form a non-degenerate simplex  $\Sigma_s = \text{conv}(\mathbf{v}_1, \dots, \mathbf{v}_{s+1})$ . For instance, the 0-simplex is a point, the 1-simplex a line segment, the 2-simplex is a triangle and the 3-simplex a tetrahedron.

Conversion between barycentric and Cartesian coordinates (and vice versa) is uniquely given by the relation:

$$\boldsymbol{\beta} = \boldsymbol{\beta}(\mathbf{x}) = A^{-1} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix}, \quad A = \begin{pmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_{s+1} \\ 1 & \cdots & 1 \end{pmatrix}. \quad (1.3)$$

The matrix  $A$  above does not depend on a particular point, and can thus be calculated (along with its inverse) once for a given domain,  $\Sigma_s$ .

The barycentric coordinates form a partition of unity,  $\sum_{i=1}^{s+1} \beta_i = 1$ , so even if we have increased the number of variables by one, we do not increase the dimension of the polynomial space. For a point in  $\Sigma_s$ , its barycentric coordinates are non-negative, which ensures that all computations are done by true convex combinations, yielding high numerical stability.

For illustration purposes, we expand the above notation for the trivariate case and use the standard unit simplex ( $\mathbf{v}_1 = (1, 0, 0)^T$ ,  $\mathbf{v}_2 = (0, 1, 0)^T$ ,  $\mathbf{v}_3 = (0, 0, 1)^T$  and  $\mathbf{v}_4 = (0, 0, 0)^T$ ). This gives the barycentric coordinate  $\boldsymbol{\beta} = (\beta_1 = x, \beta_2 = y, \beta_3 = z, \beta_4 = w = (1 - x - y - z))$ .

$$B_{i,j,k,\ell}^d(\boldsymbol{\beta}(x,y,z)) = B_{i,j,k,\ell}^d(x,y,z,w) = \frac{d!}{i!j!k!\ell!} x^i y^j z^k w^\ell, \quad (1.4)$$

$$\ell = d - i - j - k.$$

For an algebraic polynomial  $p$  of total degree  $d$ , we now have two alternative formulations, using either the power or the Bernstein basis:

$$p(\mathbf{x}) = \sum_{|j| \leq d} c_j \mathbf{x}^j = \sum_{|\mathbf{i}|=d} b_{\mathbf{i}} B_{\mathbf{i}}(\boldsymbol{\beta}(\mathbf{x})) = f(\boldsymbol{\beta}), \quad \mathbf{x} \in \Sigma_s, \quad (1.5)$$

where  $b_{\mathbf{i}}$  are called the BB-coefficients (Bernstein-Bézier) of the function. In the following we let  $f(\boldsymbol{\beta})$  be any polynomial in Bernstein form.

We also summarize some key properties of the Bernstein basis:

**Lemma 1.** *On the unit simplex, the following hold:*

1.  $B_{\mathbf{i}}^d(\boldsymbol{\beta}(\mathbf{x})) \geq 0$  for all  $\mathbf{x} \in \Sigma_s$ .
2.  $\sum_{|\mathbf{i}|=d} B_{\mathbf{i}}^d(\boldsymbol{\beta}(\mathbf{x})) = 1$  for all  $\mathbf{x} \in \mathbb{R}^s$ .
3.  $\frac{d!}{(d-|\mathbf{r}|)!} \mathbf{x}^{\mathbf{r}} = \sum_{|\mathbf{i}|=d} \frac{\hat{\mathbf{i}}!}{(\hat{\mathbf{i}}-\mathbf{r})!} B_{\mathbf{i}}^d(\boldsymbol{\beta}(\mathbf{x})), \quad |\mathbf{r}| \leq d, \hat{\mathbf{i}} = (i_1, \dots, i_s)$ .

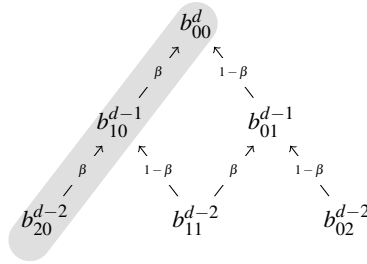
For a proof see [25]. The above lemma provides us with the foundations for a change of basis algorithm between the power and Bernstein bases.

### 1.5.2 Polynomials in Bernstein form

Given an algebraic polynomial in Bernstein-Bézier form, the de Casteljau algorithm provides a fast and numerically stable way of evaluating the function. Let  $b_{\mathbf{i}}$  denote the coefficients of an algebraic function in BB-form and compute repeated convex combinations of the base points by setting:

$$b_{\mathbf{i}}^r(\boldsymbol{\beta}) = \sum_{j=1}^{s+1} \beta_j b_{\mathbf{i}+\mathbf{e}_j}^{r-1}(\boldsymbol{\beta}), \quad b_{\mathbf{i}}^0(\boldsymbol{\beta}) = b_{\mathbf{i}}, \quad (1.6)$$

$$\mathbf{e}_1 = (1, 0, \dots, 0), \dots, \mathbf{e}_{s+1} = (0, \dots, 1).$$



**Fig. 1.6.** The de Casteljau algorithm for a univariate BB-polynomial. Each value is formed by a convex combination of the values on a lower level. At the lowest level we find the BB-coefficients of the polynomial. The shaded area indicates the coefficients for the subdivided polynomial in the interval  $[0, \beta]$ .

Then  $b_0^d(\beta)$  is the value of the polynomial at the parameter value  $\beta$  in a non-degenerate simplex. The de Casteljau algorithm can be seen as a pyramid-like structure, where each level is a convex combination of  $s + 1$  values at the level below, starting with the coefficients. This process is illustrated for a univariate polynomial in Fig 1.6.

The de Casteljau algorithm also admits a *subdivision formula* for univariate BB-polynomials, and hence allows us to find a formula for a polynomial in BB-form over a sub-interval of the original polynomial. To find the coefficients of a BB-polynomial over the interval  $[0, \beta]$ , we carry out (1.6) and store the leftmost intermediate value at each iteration:

$$c_i = b_0^j(\beta). \tag{1.7}$$

Conversely, the rightmost values give the coefficients for the BB-polynomial over the interval  $[\beta, 1]$ . For multivariate polynomials, blossoming (see Sect. 1.5.3 below), can be used to find the coefficients over a subset of the domain.

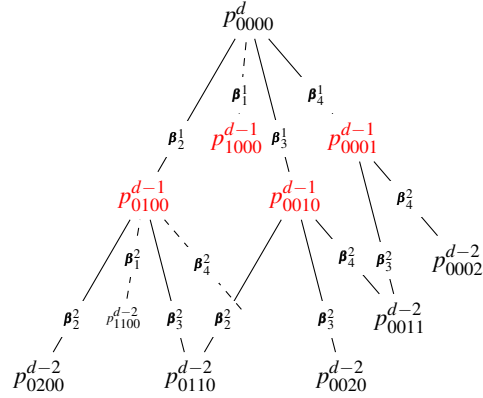
If we use the BB-coefficients of a univariate BB-polynomial  $f$  to form the points  $\mathbf{b}_i = \mathbf{b}_{i,j} = (j/d, b_i)$  we find the *control points* of a Bézier parametrization of  $f$ , often called a Bézier curve. The polygon formed by the control points is called the control polygon of the curve.

Such a parametrization has a number of useful properties:

**Lemma 2.**

1. Bézier curves have no more sign changes than their control polygons. This is called the variation diminishing property.
2. The control points generated by repeated subdivision of the control polygon converge to the original Bézier curve.
3. The entire Bézier curve is contained in the convex hull of the control points.

The above properties are used to form the root finding algorithm described in Sect. 1.4.3.



**Fig. 1.7.** The pyramidal structure of trivariate blossoming. Each value is formed as a combination of four values at the previous level, using  $\beta_i$  as weights. (Some nodes and weights have been removed for clarity.)

### 1.5.3 The blossom

If we modify the de Casteljau algorithm by introducing a sequence of parameter points,  $(\beta_1, \dots, \beta_d)$  and use a distinct point at each level of recursion, we arrive at the blossom of the polynomial, which we denote as  $\mathcal{B}$ . An intermediate value of the blossom can be expressed as

$$p_i^r = \sum_{j=1}^{s+1} \beta_{r_j} p_{i+e_j}^{r-1}, \quad p_i^0 = b_i, \tag{1.8}$$

and the blossom itself is given by

$$\mathcal{B}(f)[\beta_1, \dots, \beta_d] = p_0^d. \tag{1.9}$$

An illustration of this process for a trivariate polynomial is given in Fig 1.7.

The blossom has traditionally been used as an analytic tool for Bézier and spline curves and surfaces. In our context we are interested in using it as a computational algorithm to calculate coefficients and derivatives of an algebraic surface. Some of the properties of the blossom are summarized below:

**Lemma 3.** *Every BB-polynomial  $f$  of degree  $d$ , has an associated unique functional, called the blossom, which we will denote as  $\mathcal{B}$ .*

*The blossom has the following properties:*

1. *It is symmetric in its arguments;  $\mathcal{B}(f)[\beta_1, \dots, \beta_d] = \mathcal{B}(f)[\pi(\beta_1, \dots, \beta_d)]$ , (here  $\pi(\cdot)$  means any permutation of its arguments).*
2. *It is multi-affine in each of its variables;  $\mathcal{B}(f)[s\alpha + t\beta, \dots] = s\mathcal{B}(f)[\alpha, \dots] + t\mathcal{B}(f)[\beta, \dots]$ ,  $s, t \in \mathbb{R}$ .*
3. *It has the diagonal property;  $\mathcal{B}(f)[\beta, \dots, \beta] = f(\beta)$ .*



It is common to define the blossom as a functional with the above properties. For details see [32, 13].

We use notation in the style of Farin and let repetition of arguments be denoted  $\mathcal{B}(f)[\boldsymbol{\beta}, \dots, \boldsymbol{\beta}, \dots] = \mathcal{B}(f)[\boldsymbol{\beta}^{(n)}, \dots]$  if the argument  $\boldsymbol{\beta}$  is repeated  $n$  times.

The key to finding a curve representing the value of  $f$  along the line segment  $\mathbf{xy}$  is a surprising property of the blossom, which we summarize in the following lemma:

**Lemma 4.** *Given two points  $\boldsymbol{\alpha} = \boldsymbol{\beta}(\mathbf{x})$  and  $\boldsymbol{\gamma} = \boldsymbol{\beta}(\mathbf{y})$ ,  $\mathbf{x}, \mathbf{y} \in \Sigma_s$  and a multivariate BB-polynomial  $f(\boldsymbol{\beta})$ . Then the straight line segment from  $\mathbf{x}$  to  $\mathbf{y}$  is mapped to a univariate BB-polynomial of degree  $d$ , and its coefficients are given by repeated blossoming of the endpoints,  $\mathcal{B}(f)[\boldsymbol{\alpha}^{(d)}], \mathcal{B}(f)[\boldsymbol{\alpha}^{(d-1)}, \boldsymbol{\gamma}], \dots, \mathcal{B}(f)[\boldsymbol{\gamma}^{(d)}]$ .*

For a proof, see [13, 37, 16]. This property then leads to an algorithm for finding the coefficients  $(c_{d,0}, \dots, c_{0,d})$  of the univariate polynomial along the ray:

$$c_{d-i,i} = \mathcal{B}(f)[\boldsymbol{\alpha}^{(d-i)}, \boldsymbol{\gamma}^{(i)}], \quad i \in (0, \dots, d), \quad (1.10)$$

as illustrated in Fig 1.2.

For a point  $\boldsymbol{\beta}$  in the domain, we may also use the blossom to find derivatives with respect to the (barycentric) direction vector  $\mathbf{d}$ ,

$$D_{\mathbf{d}}^r f(\boldsymbol{\beta}) = \frac{d!}{(d-r)!} \mathcal{B}(f)[\mathbf{d}^{(r)}, \boldsymbol{\beta}^{(d-r)}], \quad r \leq d. \quad (1.11)$$

In the same manner we can also calculate mixed directional derivatives. If  $\mathbf{d}_1, \mathbf{d}_2$  represent two distinct vectors their mixed directional derivatives are:

$$D_{\mathbf{d}_1, \mathbf{d}_2}^{r,s} f(\boldsymbol{\beta}) = \frac{d!}{(d-r-s)!} \mathcal{B}(f)[\mathbf{d}_1^{(r)}, \mathbf{d}_2^{(s)}, \boldsymbol{\beta}^{(d-r-s)}], \quad r+s \leq d. \quad (1.12)$$

To summarize, the blossom provides us with the algorithms needed to convert a trivariate polynomial into a set of univariate polynomials with respect to each ray. Furthermore it allows us to find all derivatives of the surface at an arbitrary point.

## 1.6 Conclusion and Further Work

Our work presents a real-time rendering strategy for algebraic surfaces up to total degree five. However, as GPU technologies advance we believe it can be improved further. The two most obvious issues, and thus prime candidates for further work, are to increase rendering speed and the degree of surfaces we can visualize.

With regards to speed, we believe that better root finders than our current implementation can dramatically increase the performance. As GPUs get better at dynamic branching, we believe that hybrid methods will be very well suited for use on GPUs. Higher speeds will also allow a large number of primitives to be rendered simultaneously, which is necessary for displaying piecewise algebraic surfaces.

The limiting factor for the maximum degree of surface visualization is the number of registers available in each fragment pipeline. Our Nvidia 7800 GPU has 32 four-wide temporary registers, in comparison next generation GPUs (DX10) are expected to have 4096 registers[2], which will allow us to visualize surfaces of much higher total degree using our current approach.

Using the depth buffer and blending functionality of GPUs to do constructive solid geometry with algebraic surfaces, as well as integration with mesh based ray tracing techniques is also possible. More complex lighting models, including reflection, refraction and self-shadows could be possible, but these aspects will also rely on more registers and better dynamic branching.

With regards to applications, we believe that CAD-systems could benefit significantly from modeling shapes by using general algebraic surfaces. However, as algebraic surfaces often have a non-trivial topology, it has until now been necessary to analyze and establish the topology of the algebraic surface before tessellation and visualization can take place. The approach of this paper shows that such a topology analysis is not necessary and consequently allows efficient visualization of general algebraic surfaces in CAD-systems.

*Acknowledgement.* This work was supported by contract number 158911/I30 of The Research Council of Norway.

## References

1. Ordinary double point. <http://mathworld.wolfram.com/OrdinaryDoublePoint.html>. Checked as of September 2006.
2. D. Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.
3. BrookGPU. <http://graphics.stanford.edu/projects/brookgpu/>.
4. N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.
5. N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, pages 203–209, Toronto, Ont., Canada, 2006. Canadian Information Processing Society.
6. M. Christen. Ray tracing on GPU. Master's thesis, University of Applied Sciences, Basel, 2005. <http://www.clockworkcoders.com/ogls1/rt/>.
7. T. D. DeRose, R. N. Goldman, H. Hagen, and S. Mann. Functional composition algorithms via blossoming. *ACM Trans. Graph.*, 12(2):113–135, 1993.
8. T. Dokken, T. R. Hagen, and J. M. Hjelmervik. *this book*, chapter An Introduction to General-Purpose Computing on Programmable Graphics Hardware.
9. T. Dokken and V. Skytt. *this book*, chapter Intersection Algorithms for CAGD.
10. W. Donnelly. *GPU Gems 2*, chapter Per-Pixel Displacement Mapping with Distance Functions, pages 123–136. Addison-Wesley, 2005.
11. T. Duff. Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 131–138, New York, NY, USA, 1992. ACM Press.

12. S. Endraß. SURF 1.0.5. <http://surf.sourceforge.net>.
13. G. Farin. *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
14. R. T. Farouki and T. N. T. Goodman. On the optimal stability of Bernstein basis. *Mathematics of Computation*, 65:1553–1556, 1996.
15. R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
16. R. Goldman. Using degenerate Bézier triangles and tetrahedra to subdivide Bézier curves. *Computer Aided Design*, 14(6):307–311, November 1982.
17. X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361, New York, NY, USA, 2002. ACM Press.
18. J. Hirche, A. Ehlert, S. Guthe, and M. Doggett. Hardware accelerated per-pixel displacement mapping. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 153–158, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.
19. S. Holzer and O. Labs. *Algebraic Geometry and Geometric Modelling*, chapter Illustrating the classification of real cubic surfaces, pages 119–134. Mathematics and Visualization. Springer-Verlag, 2006.
20. S. Holzer and O. Labs. SURFEX 0.89. Technical report, University of Mainz, University of Saarbrücken, 2006. <http://www.surfex.AlgebraicSurface.net>.
21. F. Karlsson and C. J. Ljungstedt. Ray tracing fully implemented on programmable graphics hardware. Master's thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, 2004.
22. C. Loop and J. Blinn. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph.*, 24(3):1000–1009, 2005.
23. C. Loop and J. Blinn. Real-time GPU rendering of piecewise algebraic surfaces. *ACM Trans. Graph.*, 25(3):664–670, 2006.
24. T. Lyche and J. M. Pěna. Optimally stable multivariate bases. *Advances in Computational Mathematics*, 20:149–159, 2004.
25. T. Lyche and K. Scherer. On the p-norm condition number of the multivariate triangular Bernstein basis. *J. Comput. Appl. Math.*, 119(1-2):259–273, 2000.
26. M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *SIGGRAPH/Eurographics Graphics Hardware Workshop*, pages 57–68, September 2002. revised.
27. K. Mørken and M. Reimers. An unconditionally convergent method for computing zeros of splines and polynomials. *Mathematics of Computation*, To appear.
28. M. Olano. *A Programmable Pipeline for Graphics Hardware*. PhD thesis, Department of Computer Science, University of North Carolina, Chapel Hill, <http://www.cs.unc.edu/olano/papers/dissertation/>, April 1998.
29. M. Olano, A. Lastra, and J. Leech. Procedural primitives in a high performance, hardware accelerated, Z-Buffer renderer. Technical Report Tr97-040, UNC Computer Science Technical Report, 1997.
30. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.
31. T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, New York, NY, USA, 2002. ACM Press.

32. L. Ramshaw. Blossoming: A connect-the-dots approach to splines. Technical report, Digital Systems Research, 1987.
33. A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. *Computer Graphics*, 23(3):107–116, July 1989.
34. R. J. Rost. *OpenGL(R) Shading Language*. Addison Wesley Longman Publishing Co., Inc., 2004.
35. J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek. Realtime ray tracing of dynamic scenes on an FPGA chip. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 95–106, New York, NY, USA, 2004. ACM Press.
36. P. J. Schneider. *Graphics gems*, chapter A Bézier curve-based root-finder, pages 408–415. Academic Press Professional, Inc., San Diego, CA, USA, 1990.
37. H. P. Seidel. *A general subdivision theorem for Bézier triangles*, pages 573–581. Academic Press Professional, Inc., San Diego, CA, USA, 1989.
38. J. M. Snyder. Interval analysis for computer graphics. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 121–130, New York, NY, USA, 1992. ACM Press.
39. M. R. Spencer. *Polynomial real root finding in Bernstein form*. PhD thesis, Brigham Young University, Provo, UT, USA, 1994.
40. N. Thrane and L. O. Simonsen. A comparison of acceleration structures for GPU assisted ray tracing. Master's thesis, Department of Computer Science, University of Aarhus, August 2005.
41. Y. Uralsky. Practical metaballs and implicit surfaces. Game Developers Conference 2006. <http://developer.nvidia.com>.
42. M. Weiler, M. Kraus, and T. Ertl. Hardware-based view-independent cell projection. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 13–22, Piscataway, NJ, USA, 2002. IEEE Press.
43. B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral projection using vertex shaders. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 7–12, Piscataway, NJ, USA, 2002. IEEE Press.