Ray Casting Algebraic Surfaces using the Frustum Form

Martin Reimers[1] and Johan Seland[1,2]
[1]Centre of Mathematics for Applications, University of Oslo, Norway
[2]SINTEF ICT, Norway

EARLY DRAFT
Final version to appear in Computer Graphics Forum

Abstract

We propose an algorithm for interactive ray-casting of algebraic surfaces of high degree. A key point of our approach is a polynomial form adapted to the view frustum. This so called frustum form yields simple expressions for the Bernstein form of the ray equations, which can be computed efficiently using matrix products and pre-computed quantities. Numerical root-finding is performed using B-spline and Bézier techniques, and we compare the performances of recent and classical algorithms. Furthermore, we propose a simple and fairly efficient anti-aliasing scheme, based on a combination of screen space and object space techniques. We show how our algorithms can be implemented on streaming architectures with single precision, and demonstrate interactive frame-rates for degrees up to 16.

Introduction

An algebraic surface is the zero set of a polynomial,

$$f(x,y,z) = \sum_{0\le i+j+k\le d} f_{ijk} x^i y^j z^k = 0, \tag{1}$$

where f_{ijk} are real coefficients. Algebraic geometry is a classical mathematical discipline, and today computer visualization of algebraic surfaces is important in several fields, such as biology, physics and mathematics. Ray casting is a standard approach for visualizing a 3D surface on a 2D screen, and amounts to computing intersections with a number of "rays" inside a view frustum, i.e. the region of space that appear on the screen. Ray casting for algebraic surfaces is conceptually simple. For each ray $\mathbf{r}_{pq} : [0,1] \to \mathbb{R}^3$, corresponding to a pixel (p,q), one computes the zero of a univariate polynomial equation on the form

$$f(\mathbf{r}_{pq}(t)) = 0 \tag{2}$$

that corresponds to an intersection point closest to a view point. If such a root t is found, the corresponding point $\mathbf{r}_{pq}(t)$ is used for lighting calculations to determine a pixel color. Ray casting is a simplification of ray tracing which amounts to reflecting the ray off of the surface to obtain more advanced lighting models. Although the method is simple in principle, surfaces of higher degree requires numerical root-finding. This is a challenging task, both with respect to efficiency, stability and robustness.

We present a new approach for efficient ray casting of algebraic surfaces, based on a parameterization of the view frustum. The resulting *frustum form* (FF) yields simple expressions for the ray equations (2) and allows the use of pre-evaluated basis functions for efficiency. Furthermore, we use B-spline and Bézier based numerical root finders which allows for fast convergence and robustness. Another contribution is a new anti-aliasing method, using a combination of screen and object space techniques. Ray casting is embarrassingly parallel and therefore well suited for implementation on a modern computer equipped with a programmable stream processor, such as a Graphics Processing Unit (GPU). To achieve maximum performance, calculations critical for accuracy are performed in double precision on the CPU and those critical for performance in single precision on the GPU.

In the following section we review related work before we describe our algorithm in Sections 2 through 4. We discuss details of our implementation in Section 5 and present numerical examples in Section 6, before we conclude.

1 Related Work

Rendering of algebraic surfaces goes back to the beginning of computer graphics, see e.g. [Han83]. Recently, Loop and Blinn [LB06] demonstrated real time rendering of quartic surfaces on GPUs, using blossoming to derive closed form ray equations and computing roots analytically. Seland and Dokken [SD07] proposed a similar approach and render algebraic surfaces up to degree 5 interactively on GPUs using numerical root-finding. Sphere tracing [Har96] is another approach that has been adapted [Don05] for execution on GPUs.

Lipschitz bounds and interval arithmetic have been proposed [Mit90, KB89] as alternatives to point sampling for ray tracing in order to resolve thin features better. Recently, Knoll et. al. [KHH*07] used SSE SIMD extensions to perform intersection test on multiple rays at a time, achieving interactive (but less than 24 frames per second) for arbitrary implicit surfaces up to degree 8.

Sederberg and Zundel [SZ89] compute silhouette points along scan lines using discriminants and resultants, techniques that are rather expensive even for moderate degrees. They use a Bernstein Pyramidal Polynomial (BPP) basis which is somewhat similar to our frustum form.

An alternative to viewpoint dependent methods, is to generate a polygonal model of the surface, typically using marching cube-type methods [LC87]. Although used extensively in practice, such methods have difficulties with complex topology, singularities and thin features.

Numerical root finding is a classical field and standard methods are covered in most references on numerical methods, e.g. [IK66]. For polynomials in Bernstein form, several methods are based on subdivision, e.g. Lane and Riesenfeld [LR81], Rockwood et al. [RHD89] and Schneider [Sch90], see [Spe94] for an overview. Recently, Mørken and Reimers [MR07, MR08] proposed to use B-spline knot insertion for root-finding.

See e.g. [Far02] for an introduction to Bézier and B-Spline topics and [AMH02] for an overview of anti-aliasing and real time rendering techniques in general. For an introduction to implicit surfaces in general, see [Blo97].

2 Algorithm overview

We consider an algebraic surface of total degree d, i.e one that can be expressed in power form as (1). In this case, the ray equation (2) is in fact a univariate polynomial equation of degree at most d. Its solutions can be found numerically, a critical operation that must be implemented with care. One could in principle work directly on (2). However, this would require many costly evaluations of f. We would rather first express (2) as a univariate polynomial equation on the Bernstein form,

$$f(\mathbf{r}_{pq}(t)) = \sum_{k=0}^{d} c_{pqk} B_k^d(t) = 0, \quad t \in [0,1], \tag{3}$$

where $B_k^d(t) = \binom{d}{k} t^k (1-t)^{d-k}$ are the Bernstein basis functions. It is shown in [FR87] that this form is superior to other representations of polynomials, in particular for computing roots. We then seek the smallest root w_{pq} of (3) in $[0,1]$, using a univariate root-finder. These two steps are bottlenecks in our approach and we therefore focus on their efficient implementation. Our algorithm, which is parallel of nature, consists of the following steps:

1. Compute all ray coefficients $C = (c_{pqk})$.
2. For each pixel (p,q), seek the smallest root w_{pq} of (3).
3. For each pixel (p,q), compute a pixel color.
4. Optionally, perform anti-aliasing.

In the following section we describe the frustum form, which simplifies the coefficient computations significantly.

3 The Frustum Form

The key to our construction is the *frustum form* which results from a parameterization of the view frustum, composed with the polynomial. With this we achieve simple expressions for the ray coefficients and reduce

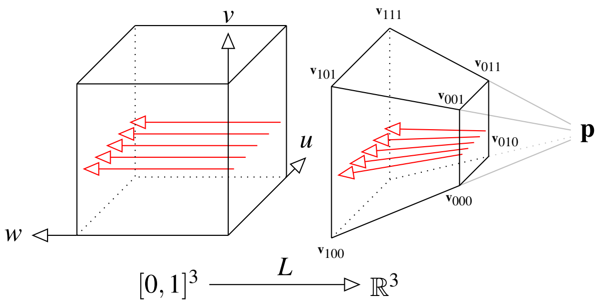


Figure 1: The view frustum is parameterized over the unit cube by a tri-linear map L, taking lines parallel to the w-axis to rays.

the algorithmic complexity of the computations. In addition it allows us to pre-compute basis functions and increase the efficiency even further.

The view frustum is the part of $\mathbb{R}^3$ that appear on the screen, see e.g. [AMH02]. It has the shape of the frustum of a pyramid, with apex in a view point $\mathbf{p}$, and can be defined by eight points $(\mathbf{v}_{ijk})$ in $\mathbb{R}^3$. Four of the points constitutes a rectangle in the far clipping plane, and the remaining four a rectangle in the near clipping plane, see Figure 1. We associate the rectangle in the near clipping plane with the screen with coordinates in $[0,1]\times[0,1]$ and $(m+1)\times(n+1)$ pixels. Each pixel (p,q) corresponds to a ray through the view point and the pixel center with screen coordinates $(p/m, q/n)$.

The view frustum can be parameterized over the unit cube by a tri-linear mapping $L : [0,1]^3 \to \mathbb{R}^3$ on the form

$$L(u,v,w) = \sum_{i,j,k=0}^{1,1,1} \mathbf{v}_{ijk} B_i^1(u) B_j^1(v) B_k^1(w). \tag{4}$$

The corners and faces of $[0,1]^3$ are mapped to corners and faces of the view frustum, respectively. A ray is thus linearly parameterized as $\mathbf{r}_{pq}(w) = L(p/m, q/n, w)$.

Given a view frustum in $\mathbb{R}^3$ and a polynomial f, we define the corresponding Frustum Form (FF) to be the composition

$$g = f \circ L : [0,1]^3 \to \mathbb{R}. \tag{5}$$

It is easy to show that $g(u,v,w)$ is a polynomial for which u,v and w occurs in powers of at most d. It can therefore be expressed as a Bernstein tensor product of tri-degree d,

$$g(u,v,w) = \sum_{i,j,k=0}^{d,d,d} g_{ijk} B_i^d(u) B_j^d(v) B_k^d(w), \tag{6}$$

where $g_{ijk} \in \mathbb{R}$ are the *frustum form coefficients*. Note that although the degree of g appears to be $3d$, its algebraic degree can be shown to be d.

As the mapping L is injective, the part of the algebraic surface that lies in the view frustum is in one-to-one correspondence with the part of the algebraic surface given by $g = 0$ in the projective parameter domain $[0,1]^3$. The normal of the algebraic surface $f = 0$ at a non-singular point $\mathbf{x}$ is parallel to the gradient $\nabla f(\mathbf{x})$. The corresponding gradient (and hence normal) of $g = 0$ is related by

$$\nabla g(\mathbf{u}) = \nabla f(L(\mathbf{u})) \nabla L(\mathbf{u}), \tag{7}$$

where $\nabla L(\mathbf{u})$ is the Jacobian matrix of L evaluated in $\mathbf{u}$. Note that $\nabla L(\mathbf{u})$ is invertible for any $\mathbf{u}$. A silhouette point of $f = 0$ is a point for which $f(\mathbf{x}) = 0$ and $\nabla f(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{p}) = 0$. The latter condition says that the normal at $\mathbf{x}$ should be perpendicular to the ray through $\mathbf{p}$ and $\mathbf{x}$. The corresponding conditions for g are

$$g(u,v,w) = 0, \qquad g_w(u,v,w) = 0, \tag{8}$$

Our use of the frustum form is primarily as an intermediate representation of the algebraic surface, aligned with the view frustum. The key observation is that a ray equation takes the form

$$f(\mathbf{r}_{pq}(w)) = f(L(\frac{p}{m},\frac{q}{n},w)) = g(\frac{p}{m},\frac{q}{n},w)$$
$$= \sum_{k=0}^{d} \left(\sum_{i,j=0}^{d,d} g_{ijk} B_i^d(\frac{p}{m}) B_j^d(\frac{q}{n}) \right) B_k^d(w),$$

i.e. a degree d Bernstein polynomial in w. Consequently, the k'th coefficient of the ray equation for (p,q) is of the form

$$c_{pqk} = \sum_{0\le i,j\le d} g_{ijk} B_i^d(p/m) B_j^d(q/n), \tag{9}$$

which can be recognized as the evaluation of a bivariate tensor product with coefficients equal to the k'th "slice" of the three dimensional array $G = (g_{ijk})$ at the pixel coordinates.

The frustum form is somewhat similar to the BPP of [SZ89]. Both use a pyramid to describe a projective coordinate system. BPP positions the near plane at the view point, whereas in FF it can lie at an arbitrary distance from **p**. This allows the near and far planes to encompass the surface more closely. In our experience this yields better numerical stability and clips away geometry in the foreground.

4 Algorithm details

We next describe the four steps of our algorithm in detail, focusing on the algorithms. The implementation details are deferred to the subsequent section.

4.1 Ray coefficient computation

In order to compute the ray coefficients, we first compute the frustum form of f, relative to the current view frustum. The FF coefficients G can be computed in a number of ways, e.g. by the use of blossoms as described in [DGHM93], by a recursion similar to the one in [SZ89] or by interpolation. The latter does not depend on a particular representation of f, is easy to implement and efficient, and is therefore our preferred choice.

We choose a grid of $(d+1)^3$ distinct interpolation points (u_p, v_q, w_r) in $[0,1]^3$, and seek FF coefficients $G = (g_{ijk})$ such that for each $0 \le p,q,r \le d$,

$$\sum_{i,j,k=0}^{d,d,d} g_{ijk} B_i^d(u_p) B_j^d(v_q) B_k^d(w_r) = f(L(u_p, v_q, w_r)). \tag{10}$$

This tensor product interpolation problem has a unique solution which can be found by solving a sequence of univariate problems. Using the Einstein summation convention, we can formulate the system of equations compactly as tensor products as

$$\Omega^u_{ip}\Omega^v_{jq}\Omega^w_{kr} G_{ijk} = F_{pqr}, \tag{11}$$

where $\Omega^u = (B_i^d(u_j))$ denotes the degree d Bernstein collocation matrix etc. and $F := (f(L(u_p, v_q, w_r)))$ is a rank three tensor (cubical grid). Its solution can be formulated similarily as

$$G_{ijk} = (\Omega^w)^{-1}_{kr} (\Omega^v)^{-1}_{jq} (\Omega^u)^{-1}_{ip} F_{pqr}. \tag{12}$$

In other words, the frustum form coefficients can be found by first forming F by evaluating $f \circ L$ at the interpolation points, and then performing three tensor products. Each of these can be implemented as a sequence of $d+1$ matrix products, one for each slice of F. By choosing fixed interpolation points, we can invert the collocation matrices Ω^u etc. once for fixed degree d and store them.

The interpolation points (u_p, v_q, w_r) should be chosen so that the interpolation problem (10) is as numerically stable as possible. To that end we choose the degree d Chebychev points for (u_p), (u_q) and (w_r) as they minimize the condition number Ω and hence the numerical error introduced in the interpolation process, see e.g. [Mør96].

With the frustum form at hand, the ray-coefficients of ray (p,q) can be computed from (9). The coordinates $(p/m, q/n)$, as well as the degree d, can be regarded as fixed and so we can pre-compute the matrices $M = (B_i^d(p/m))$ and $N = (B_j^d(q/n))$, and store them for later lookup. The computation of c_{pqk} thus requires $(d+1)^2$

multiplications/additions. By contrast, the evaluation of f requires in general at least $(d+1)(d+2)(d+3)/6$ such operations, after evaluation of the basis functions. Therefore, the alternative approach based on interpolation for each ray independently requires significantly more computations than our approach. The computation of the ray-coefficients can be formulated as another tensor product, in Einstein notation as $C_{pqk} = M_{ip}N_{jq}G_{ijk}$. This can be implemented as $d+1$ matrix products on the form

$$C_k = MG_kN^T, \tag{13}$$

where $C_k = (c_{pqk})$ is the k'th slice of the cubical array C.

4.2 B-spline based root finding

Our next step is to compute for each pixel the smallest solution in $[0,1]$ of the corresponding ray-equation (3). The roots of these equations can be badly conditioned and with higher multiplicity, typically for rays intersecting the silhouettes of the surface. This requires robust and efficient root-finders. We take the approach proposed in [MR07], using B-spline techniques to approximate the roots. We show how several previous root finding algorithms based on Bézier subdivision can be implemented using similar techniques instead of recursions since such algorithms are not currently well supported on GPUs.

The Bernstein form of a polynomial is particularly well suited for numerical root-finding, see [FR87]. It has the variation diminishing property: the number of zeros, counting multiplicities, exceeds the number of strict sign changes in the coefficients. Moreover, a polynomial is approximated by its control polygon and it can be subdivided using the de Casteljau algorithm. These properties have been exploited in many previous root isolation and root approximation algorithms, providing a high degree of robustness.

B-splines can be considered a generalization of Bernstein polynomials, as a polynomial in Bernstein form has the same coefficients as a spline on a Bernstein knot-vector, i.e. with $d+1$ knots at either end of the interval. B-splines have similar properties and can be refined through knot insertion. Bézier subdivision is a special case and corresponds to inserting d equal knots into a B-spline knot-vector. Moreover, the B-spline control polygon approximates the spline to second order with respect to the knot spacing. Loosely speaking, inserting knots refines the representation of a polynomial locally and draws the control polygon closer to the function around the new knot. This suggests the following simple root finding approach: repeatedly insert knots at the zeros of the control polygon of the B-spline form of the polynomial. This approach was recently in [MR07] shown to converge unconditionally to the smallest zero of a B-spline, with second order rate to simple roots, i.e. with $p'(t) \neq 0$. A similar algorithm based on estimating the root multiplicity was proposed in [MR08], yielding second order convergence even to roots with higher multiplicity.

In addition to the B-spline algorithms, we implemented the methods suggested by Lane and Riesenfeld [LR81], see also [RHD89], which is based on Bézier subdivision at the zeros of the control polygon. We also extended the multiplicity estimation method of [MR08] to yield a similar Bézier method. The method proposed in [Sch90] is based on Bézier subdivision at the midpoint of the interval. These algorithms where originally proposed as recursive algorithms, based on Bézier subdivision. Recursive algorithms are inconvenient on current GPUs and we therefore implemented them sequentially using a combination of B-spline and Bézier algorithms. As the knots in the resulting knot vectors are repeated d times, we can use a simplified knot vector by storing only one copy of each knot. We can further identify each piece with a Bernstein polynomial and use Bézier subdivision instead of knot insertion. Moreover, there is no need to store a tree of successively subdivided polynomials which would require more registers. Finally, we implemented a version of *Bézier clipping* [NSK90], which is based on Bézier subdivision at the smallest intersection between the t-axis and the convex hull of the control points. This method is algorithmically simpler than the others since it operates sequentially on only one Bernstein polynomial.

Stopping criteria are crucial for the efficiency and accuracy of iterative root finders. The methods based on Bézier subdivision are usually stopped when the length of the interval is smaller than ε, or $|f(x)| < \varepsilon^2$ for some user defined constant ε. For the spline methods we used the criterion proposed in [MR08], which is based on the density of the knots around the zero of the control polygon. More precisely, it was shown that if the control polygon is zero at the point x in the k'th segment of the control polygon, then $f(x) = \mathcal{O}(h(x)^2)$ where $h(x) = \max(|x - t_{k+1}|, |x - t_{k+d-1}|)$ and the t_i's are knots. Hence, it is reasonable to stop the iterations when $h(x) < \varepsilon$ since $f(x)$ is dominated by ε^2 asymptotically. This criterion is cheap to evaluate and easy to adapt to the Bézier based methods for comparison.

The B-spline form allows a simple and practical optimization strategy, based on the coherency of data available to us at a given time. If we have a good guess for the root of a given ray equation, we might insert d

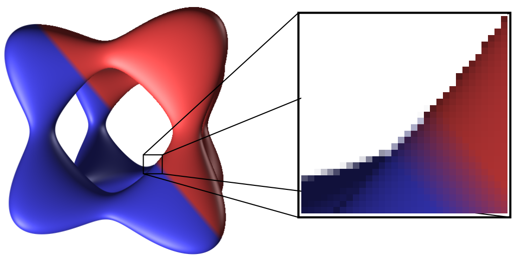


Figure 2: Our anti-aliasing algorithm is applied to the blue parts of the “Tangle” surface. Pixels near silhouettes are blended between the surface color and the background.

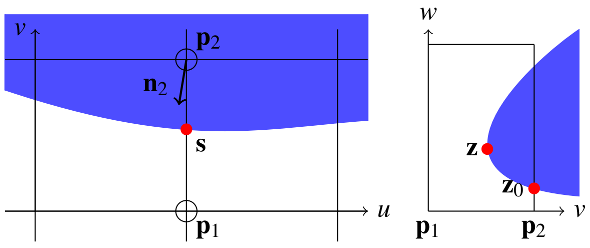


Figure 3: For two neighboring rays with a clear difference in color, we compute a point $\mathbf{s}$ on a separating curve such as a silhouette, and blend the colors using Wu’s method.

knots (or subdivide) at this location. Such a guess could be the root of a neighboring pixel (ray coherency) or the root at the pixel from the previous frame (frame coherency), e.g. when rotating a surface. Inserting d knots initially using Bézier subdivision is a cheap operation that in many cases improves the efficiency significantly, see Section 6.

The B-spline form also allows clipping of the parameter domain in a straightforward fashion. Parts of the view frustum can be excluded by specifying a clipping region, e.g. a sphere or a clipping plane. This can be incorporated in the root-finder by clipping from a ray parts outside the clipping region by knot insertion or Bézier subdivision.

4.3 Lighting Calculations

Once a root w_{pq} of a ray equation has been found, we compute a color of the corresponding pixel using a Phong lighting model, but others are certainly possible, see e.g. [AMH02].

Phong shading requires the surface normal, which is parallel to ∇f. In our case, we can instead calculate the gradient of g in parameter space, and apply (7) to map it to object space where the light sources and camera are defined. It is straightforward to evaluate the gradient of g in $(p/m, q/n, w_{pq})$. To accelerate these computations we can pre-evaluate the derivatives of the basis functions at each ray, forming matrices $DM = (DB_i^d(p/m))$ and $DN = (DB_i^d(q/n))$. We can then compute the coefficients of g_u and g_v along the ray using the analogue of (13). We then use de Casteljau in one variable to evaluate $g_u(p/m, q/n, w_{pq})$ and $g_v(p/m, q/n, w_{pq})$, together with the derivative of the ray equation $g_w(p/m, q/n, w_{pq})$.

Although straightforward, the above computations are still rather expensive and does have an impact on the overall performance. As an alternative, we propose to approximate the normal by instead using the surface samples computed in the ray casting step. We choose in each coordinate direction the pixel neighbor with a root w_{kl} that is closest to w_{pq}. In the rare case that two such neighbors cannot be found, one can fall back to computing the gradient. Otherwise, we can form a triangle in $\mathbb{R}^3$ from $(p/m, q/n, w_{pq})$ and its two neighbors. We can then compute the triangle normal and use it as an approximation to the surface normal of $g = 0$. To preserve the orientation of the surface, the normal is multiplied by the sign of the first ray coefficient.

4.4 Anti-aliasing

After the initial coloring of the pixels, aliasing effects occur between different regions of the surface and the background, see Figure 2. The regions are separated by silhouette- or boundary curves, i.e. the intersection of the surface with the boundary of the the parameter domain. We next propose an anti-aliasing strategy based on estimating the location of such separating curves. It is inspired by Wu's method [Wu91] and blends the color of a pixel with that of a neighboring pixel across such a curve.

Suppose the color of a pixel $\mathbf{p}_1 = (u_1, v_1)$ differs more than some user defined threshold from the color of a neighbor $\mathbf{p}_2 = (u_2, v_2)$, and such that the roots satisfy $w_1 \geq w_2$, taking misses to have $w > 1$. Our assumption is that the two pixels belong to different parts of the image, e.g. $\mathbf{p}_1$ could be a miss and $\mathbf{p}_2$ a hit like in Figure 3. If this is so, there must be a point between them separating the two regions, i.e. a screen point $\mathbf{s}$ corresponding either to a surface point $\mathbf{x}$ on the boundary of the view frustum, or a silhouette point. The point $\mathbf{s}$ can be written as $\mathbf{s} = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2$ for some $\alpha \in [0,1]$. Wu argues in [Wu91] that in this situation, setting the color of $\mathbf{p}_1$ to

$$(1-\alpha)\,\mathrm{color}(\mathbf{p}_1) + \alpha\,\mathrm{color}(\mathbf{p}_2) \tag{14}$$

yields good anti-aliasing. We adapt this approach by seeking silhouette- or boundary points separating two regions.

We first search for a silhouette point $\mathbf{z}$ near the parameter point $\mathbf{u}_2 = (u_2, v_2, w_2)$, by seeking solutions of (8) in the square $Q = [\mathbf{p}_1, \mathbf{p}_2] \times [0,1]$. Clearly, these are two equations in two unknowns which can be put on the form

$$\mathbf{h}(v,w) := (g(u,v,w), g_w(u,v,w)) = (0,0), \tag{15}$$

in case $\mathbf{p}_1$ and $\mathbf{p}_2$ are vertical neighbors as in Figure 3, and otherwise as $\mathbf{h}(u,w)$. We use Newton's method to search for silhouette points with the neighboring hit $\mathbf{z}_0 = (u_2, w_2)$ as a starting point. The iterations reads

$$\mathbf{z}_{i+1} = \mathbf{z}_i - (\nabla\mathbf{h}(\mathbf{z}_i))^{-1}\mathbf{h}(\mathbf{z}_i). \tag{16}$$

Here the Jacobian $\nabla\mathbf{h}(\mathbf{z}_i)$ can be evaluated by computing the partial derivatives of g with respect to the two variables. We check in each iteration for convergence and that $\mathbf{z}_i$ is in Q. If a silhouette point is found we set $\mathbf{s}$ to be the corresponding screen point and α as described above.

If $\mathbf{p}_1$ missed the surface and we could not find a silhouette point, we seek instead a point on the boundary curve between the two rays. We first determine the w coordinate of the clipping plane in which the boundary curve lies, checking w_2 and the normal $\mathbf{n}_2$. We then seek a solution to $g(\mathbf{u}) = 0$ restricted to the line $[\mathbf{p}_1, \mathbf{p}_2] \times \{w\}$. Since this is effectively a univariate problem, we use Newton's method with a starting value derived from $\mathbf{u}_2$.

A pixel $\mathbf{p}_1$ may have more than one candidate neighbor to blend with. In this case we choose the one for which the angle between the vector $(\mathbf{p}_1 - \mathbf{p}_2)$ and the projection of the normal $\mathbf{n}_2$ to the screen is smallest. This is a heuristic that is consistent with Wu's method, and that seeks to blend $\mathbf{p}_1$ across a separating curve and not along one.

Having found a blending factor α as described above, the last step is to give $\mathbf{p}_1$ its final color according to (14).

5 Implementation

Our algorithm is designed for efficiency on modern hardware architectures, typically with several cores and with streaming processors such as GPUs. A trait of GPUs is that they are limited to single precision floating point computations, or perform them at twice the speed of double precision. It is therefore preferable to split computations between the CPU and GPU, in order to balance accuracy and performance.

We represented the polynomial f in the tensor product Bernstein form, $f = \sum_{ijk} b_{ijk} B_i^d(x) B_i^d(y) B_k^d(z)$. According to [FR88], this form is preferred for numerical reasons and can be evaluated with better numerical stability than the power form. Note however that although this works for a general surface, many surfaces have simple power form expressions which could be evaluated much faster by optimizing the evaluation.

As many of the computations can be formulated as matrix products, we can utilize standard (and often vendor optimized) BLAS libraries. It should therefore be straightforward to implement our method as new hardware architectures emerge.

Our reference implementation is based on C++ and uses the Boost Ublas library for CPU computations. For GPU computations we have used the Nvidia CUDA API. This allows programming an Nvidia GPU as a stream processor using the C language, bypassing the need to use traditional graphics libraries such as DirectX or OpenGL. Furthermore, CUDA allows us to use the exact same source code for both CPUs and GPUs, thus simplifying verification and comparison of the two. To help the compiler unroll loops, keep register usage at a minimum, and allocate the correct size of arrays, we use the template mechanism of CUDA to specialize the compiled object code for each degree, up to a maximal degree decided at compile time.

A key property of our algorithm is that for a given degree and screen resolution, we can pre-compute the matrices M, N and the Ω-matrices in double precision on the CPU, using de Casteljau algorithm. Thereafter, M and N can be transferred to the GPU. The collocation matrices Ω^u, Ω^v and Ω^w are inverted using LU-decomposition in double precision and stored in CPU memory.

The per frame computations are organized in a number of "passes", corresponding to the steps outlined in Section 2. In our experience, this organization is faster than grouping them all together as it makes CUDA thread scheduling easier.

The calculation of a new frame starts when the view frustum or f changes, typically because of user input. We compute FF coefficients by evaluating $f \circ L$ at our chosen interpolation points and solve the interpolation problem (10) by repeated tensor multiplication with the inverse Ω-matrices. As these operations are critical for the accuracy of the overall method they are performed in double precision. For reasonable degrees with $d \ll m, n$, these computations are negligible and G can be uploaded to a GPU without saturating the graphics bus. Otherwise, it is possible to trade precision for performance and move these computations to the GPU as well.

To find the ray-coefficients in (3), we multiply the FF coefficients with the pre-evaluated matrices M and N, using a dedicated CUDA kernel for matrix-tensor products. These matrices are stored in *texture memory* on the GPU to facilitate spatial cache locality. The tensor G is stored in *constant memory* on the GPU, and we fetch four components of G at a time to utilize coalesced memory reads. The output coefficients are organized and aligned, so that the root-finding kernel can perform coalesced reads in the next pass.

For the root finding, we handle each ray in a separate CUDA thread, independent of the other threads. We store the ray coefficients and the knots in separate, contiguous arrays of fixed size, dependent on the maximal number of iterations we allow. For knot insertions, we use Boehms algorithm, carefully overwriting the contents of the arrays.

6 Numerical results

We next present numerical results from a number of tests on the nine algebraic surfaces depicted in Figure 4, with degrees from 2 and up to 16. We used a Athlon 4200+ with an Nvidia GeForce 8800 GTX running Linux. We have used a screen resolution of 512×512 and a stopping criterion for the root-finders at $\varepsilon = 5 \times 10^{-4}$, which in our experience give a good trade-off between accuracy, topological correctness and efficiency. All tests were performed with the origin in the middle of the view frustum, a field of view of 45 degrees and a distance between 1 and 2 between the near and far clipping planes and a view direction depending on the surface.

In order to validate our method we computed for each polynomial f a "global" error as follows. We first find the maximal absolute Bernstein coefficient value $K := \max |b_{ijk}|$. We then rendered the algebraic surface and evaluated the "normalized" errors $|f \circ L(p/m, q/n, w_{pq})|/K$ for each found root w_{pq}. Table 1, which shows the mean and max of these numbers, indicates that our ray casting method hits the surface accurately, with global errors typically smaller than 10^{-6}. Our algorithm displayed a robust behavior for reasonable surfaces and view frustum configurations. However, choosing a large view frustum leads to instabilities and numerical problems. This is particularly pronounced in surfaces with singularities, such as the heart surface. Performing the computations in double precision on the CPU appears to eliminate these effects to some extent.

6.1 Root finding

The most critical component in our ray caster is the root-finder. We therefore benchmarked our root-finders in order to compare their accuracy and efficiency. In the following, the basic LR and MR refers to the methods in [LR81] (see also [RHD89]) and [MR07], while LR-Mult and MR-Mult stands for the method proposed in [MR08], adapted to the LR and MR methods respectively.

In the first two columns of Table 2 we report the mean and max errors $|w_{pq} - w^r_{pq}|$ of these methods, where w^r_{pq} is a reference solution computed in double precision on the CPU using the MR method with $\varepsilon = 10^{-8}$. The third column contains the number of knot-insertions performed, where one de Casteljau step corresponds to d knot insertions. The last column is the number of milliseconds per frame, averaged over 100 frames. Note that our comparison with a reference solution is not entirely robust since the reference solution might miss the surface or hit a different part of the surface. This effect is particularly pronounced for Mult methods which in our experience are better near silhouettes and hence might give different results than the reference solution. We nevertheless feel that the numbers reflect the relative performance of the methods.

The basic LR and MR root-finders appears to have comparable run times for lower degrees, while the LR method is faster for higher degrees. The number of steps required for the LR method is typically between two and three times the corresponding number of knot insertions for the MR method, although the latter seems to be slightly more accurate using the same stopping conditions. We believe the performance advantage of the LR methods lies in the simplicity of the de Casteljau steps which requires fewer registers and computations and therefore executes very efficiently.

In a typical scene there will be rays near silhouettes with roots that are close to being multiple. The Mult methods typically converge in less iterations in such situations, and the LR-Mult method appears to offer a slight performance increase over the basic LR method. The MR-Mult method on the other hand is slower than MR, probably because of the more elaborate computations.

The stopping criterion based on the density of the knots appeared to be significantly more efficient than the criterion based on $|f(x)|$ used in e.g. [RHD89]. It appears to yield higher efficiency while maintaining accuracy, also for the LR methods. One reason for this can be that we compare with ε which is typically far from machine accuracy, in contrast to critaria based on $|f(x)| < \varepsilon^2$.

Our implementation of one-sided Bézier clipping was significantly slower than the LR and MR methods for degrees $d > 4$. The convergence was particularly slow near silhouettes where multiple roots often occur. Although the method is robust and simple in terms of computations, it is rather conservative and does not appear to compete with methods based on more accurate root estimates. The method in [Sch90] is in effect an interval halving method with less than quadratic convergence rate and was inferior to the other methods in all our experiments.

6.2 Rendering performance

Finally we discuss the rendering performance of our algorithm. In Figure 4 we have depicted our test surfaces with frame-rates without anti-aliasing. We also report the percentage of the rendering time spent respectively for: CPU computations, coefficient computations, root finding and shading.

The numbers imply that root-finding is the bottleneck of our method for moderate degrees. For high degrees the CPU computations become more influential and in fact dominate the computations for the degree 16 super-sphere. This behavior is perhaps not surprising as the evaluation of f and the matrix multiplications have high complexity. Although our implementation could be optimized in these respects, this nevertheless indicates that it could be worthwhile to trade accuracy for efficiency and move these computations to the GPU for higher degree. Note that an optimized evaluation of the super-sphere in power form roughly triples the framerate reported in Figure 4. The super-sphere is extreme and can only be displayed at a less than real time frame-rate, provided the view frustum is chosen to closely encompass the surface. Since our method is based on interpolation, it is important that the view frustum is relatively close to the surface so that computations stay within floating point range. The spline based root-finders yields robust results and also seems capable of providing some topological robustness. The “Kiss” surface for example has a very thin component that is well resolved with our method, provided the stopping criterion is not too strict.

We found that in a typical scene rendered with gradient based normals, it was more efficient to compute the coefficients of the partial derivatives g_u and g_w along with the ray coefficients in the first step of our algorithm. The approximated normal proposed in Section 4.3 yields shading results that are indistinguishable from those based on the gradient, and at a fraction of the computational cost. The former computations do not depend on the degree, while computing the gradient does.

Our anti-aliasing approach yields visually pleasing results and comes at a performance loss in the range 25% to 50%, depending on the complexity of the scene. By contrast, we believe a antialiasing method based on $2\times$ supersampling would incur a performance loss of about 75%.

We have already mentioned coherency as one optimization of the ray casting method, exploiting similarities in neighboring pixels or in previous frames. We have experimented with neighbor coherency, but found it difficult to get a substantial increase in performance. This is likely due to more complex thread scheduling. Frame coherency on the other hand yields moderate to significant speedups, typically with an increase in total frame-rate in the range 10% to 100%, depending on the complexity of the scene and the rate at which the scene is navigated. In our experience these are simple and very efficient optimizations for which our root-finders are well suited. In fact, the MR methods appears to benefit the most from frame coherency.

It is difficult to compare the performance of our method with previous work since this requires working implementations. The GPU method in [LB06] is very efficient due to its use of analytic methods, but is

Surface	Mean	Max
Cayley	$1.2e-7$	$1.5e-6$
Klebsch	$2.3e-7$	$4.7e-6$
Tangle	$2.3e-7$	$4.7e-6$
Kiss	$2.7e-7$	$1.2e-5$
Klein	$2.4e-6$	$1.9e-5$
Heart	$4.3e-9$	$5.6e-8$
Linked-Tori	$3.4e-8$	$2.9e-7$
Decic	$3.5e-9$	$7.3e-7$
Super-sphere	$7.2e-9$	$2.9e-7$

Table 1: The average and maximal global error $|f \circ L(w_{pq})|$.

	LR				LR-Mult				MR				MR-Mult			
Surface	mean	max	its	ms	mean	max	its	ms	mean	max	its	ms	mean	max	its	ms
Cayley	$9.7e-8$	$2.0e-4$	11.2	5.1	$8.8e-8$	$1.2e-4$	11.1	5.1	$5.2e-8$	$1.5e-4$	4.8	5.3	$6.3e-8$	$3.4e-5$	4.7	5.4
Klebsch	$1.1e-7$	$2.7e-4$	11.7	5.4	$5.3e-6$	$1.5e-1$	12.2	5.7	$6.6e-8$	$1.8e-4$	5.1	5.3	$5.5e-7$	$6.2e-2$	5.3	6.0
Tangle	$2.5e-7$	$2.3e-4$	21.8	9.4	$1.5e-4$	$5.1e-1$	21.5	10.7	$1.3e-7$	$1.2e-4$	9.7	12.6	$5.9e-5$	$4.9e-1$	9.4	14.7
Kiss	$2.7e-6$	$3.4e-1$	21.2	10.0	$2.7e-6$	$3.4e-1$	19.9	8.7	$1.3e-6$	$3.3e-1$	9.7	15.4	$1.7e-5$	$4.8e-1$	9.0	14.5
Klein	$2.1e-7$	$1.8e-4$	32.4	16.0	$3.7e-5$	$2.3e-1$	29.8	14.3	$9.6e-8$	$4.7e-5$	14.5	22.7	$1.6e-5$	$2.4e-1$	12.6	23.8
Heart	$1.6e-5$	$7.0e-2$	34.6	17.6	$1.1e-4$	$3.6e-2$	33.8	16.2	$6.3e-6$	$2.9e-2$	15.6	26.6	$5.5e-5$	$2.9e-2$	14.4	28.2
Linked Tori	$3.7e-7$	$1.4e-4$	53.7	29.0	$2.2e-5$	$3.2e-1$	51.6	28.1	$2.0e-7$	$1.2e-4$	23.1	48.7	$1.8e-5$	$3.0e-1$	21.1	51.4
Decic	$7.8e-5$	$7.3e-1$	69.5	55.5	$2.9e-3$	$8.5e-1$	62.9	49.1	$2.2e-5$	$7.3e-1$	35.6	124.2	$2.3e-3$	$8.7e-1$	31.3	139.5
Super-sphere	$1.5e-5$	$3.6e-3$	103.5	77.0	$1.0e-5$	$5.4e-3$	98.9	79.0	$9.9e-6$	$3.6e-3$	51.9	185.6	$9.9e-6$	$3.6e-3$	48.7	231.4

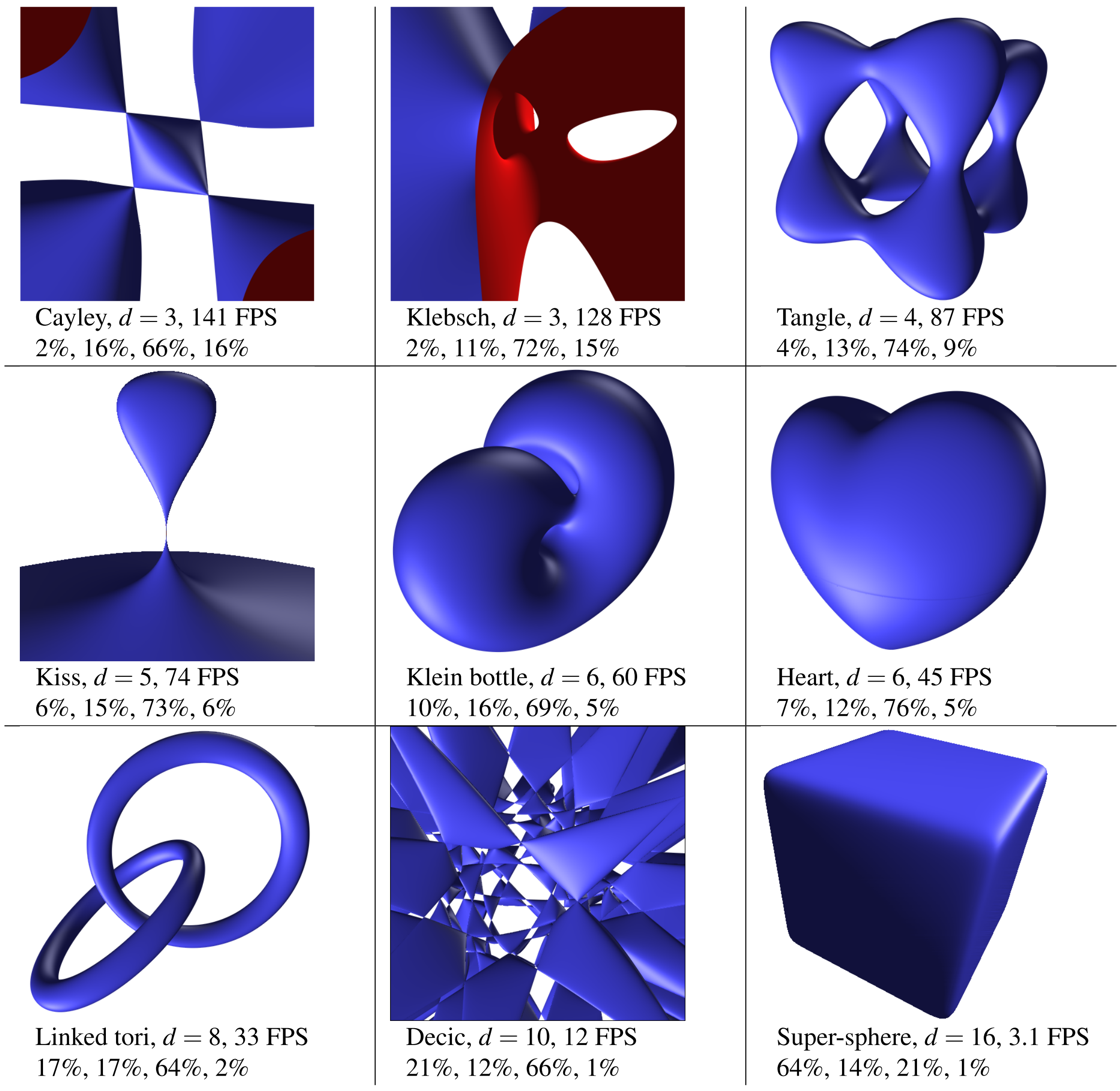
Table 2: The accuracy and performance of root-finders. "mean" and "max" refers to the errors $|w_{pq} - w^r_{pq}|$, "its" denotes the average number of knot insertions and "ms" the average number of milliseconds per frame.

restricted to degree $d \leq 4$. We estimate that for these particular degrees, their method would yield frame-rates roughly one order of magnitude faster than ours on the same hardware. The CPU approach of [KHH*07], which appears to be by far the most efficient algorithm for general degrees, appears to be about one order of magnitude slower than ours. However, a GPU implementation of their method would most likely result in a much more efficient algorithm.

7 Conclusion and Further work

The method described in this work allows for interactive rendering of algebraic surfaces on GPUs of a higher degree than previously demonstrated. The frustum form simplifies the computations significantly and is well suited for implementation on a GPU. Our numerical examples show that our root finding approach robustly and quickly finds the first root of each ray. We found that the Bézier based methods, implemented in a spline like fashion, performs better than the methods based on knot insertion for higher degree. Moreover, the stopping criterion we used appears to improve the performance of the LR method as presented in [RHD89] significantly. The simplified normal estimation yields fast and fair shading. We have also proposed a relatively simple anti-aliasing scheme, based on estimating the sub-pixel distance to the surface from a ray, with pleasing results for both interior and exterior silhouettes, as well as for boundary clipping curves.

We see several topics for future research. As future GPUs will most likely handle double precision, we believe our approach can be applied for even higher degrees, as more computations can be performed on the GPU. Handling more than one surface or piecewise surfaces should be straightforward. Non-polynomial implicit surfaces could be approximated directly since our method is based on interpolation. As ray casting is based on point-sampling, thin features might fall between rays. Although our method can handle thin features to some extent, one could consider methods similar to ours, but based on interval arithmetic, which would be more robust in this respect. Such methods could benefit from the simplified expressions derived from the frustum form. This robustness could prove useful for visualizing lower dimensional varieties, such as surface intersections and singularities. The frustum form yields no advantage for arbitrary rays as is needed in a full ray tracer. However, the first intersection, which is often the most critical, can be found efficiently by our method.



Cayley, $d = 3$, 141 FPS
2%, 16%, 66%, 16%

Klebsch, $d = 3$, 128 FPS
2%, 11%, 72%, 15%

Tangle, $d = 4$, 87 FPS
4%, 13%, 74%, 9%

Kiss, $d = 5$, 74 FPS
6%, 15%, 73%, 6%

Klein bottle, $d = 6$, 60 FPS
10%, 16%, 69%, 5%

Heart, $d = 6$, 45 FPS
7%, 12%, 76%, 5%

Linked tori, $d = 8$, 33 FPS
17%, 17%, 64%, 2%

Decic, $d = 10$, 12 FPS
21%, 12%, 66%, 1%

Super-sphere, $d = 16$, 3.1 FPS
64%, 14%, 21%, 1%

Figure 4: Some sample surfaces rendered with our algorithm. The quoted frame-rates are representative when navigating the surfaces, with frame coherency and anti-aliasing disabled. The second line of numbers indicates the percentage of the render time used for CPU computations, ray coefficient calculation, root finding and shading, respectively.

Acknowledgment

We wish to thank Christopher Dyken for many fruitful discussions.

References

[AMH02] AKENINE-MØLLER T., HAINES E.: *Real-Time Rendering (2nd Edition)*. AK Peters, Ltd., July 2002.

[Blo97] BLOOMENTHAL J. (Ed.): *Introduction to Implicit Surfaces*. Morgan Kaufmann, 1997.

[DGHM93] DEROSE T. D., GOLDMAN R. N., HAGEN H., MANN S.: Functional composition algorithms via blossoming. *ACM Trans. Graph. 12*, 2 (1993), 113–135.

[Don05] DONNELLY W.: *GPU Gems 2*. Addison-Wesley, 2005, ch. Per-Pixel Displacement Mapping with Distance Functions.

Surface	Polynomial f
Cayley	$4(x^2+y^2+z^2)-16xyz$
Klebsch	$81(x^3+y^3+z^3)-189(x^2y+x^2z+y^2z+z^2x+z^2y)+54xyz+126(xy+xz+yz)-9(x^2+y^2+z^2)-9(x+y+z)+1$
Tangle	$x^4-5x^2+y^4-5y^2+z^4-5z^2+11-8$
Kiss	$x^2+y^2-z^4+z^5$
Klein Bottle	$(x^2+y^2+z^2+2y-1)\left((x^2+y^2+z^2-2y-1)^2-8z^2\right)+16xz(x^2+y^2+z^2-2y-1)$
Heart	$(x^2+\frac{9}{4}y^2+z^2-1)^3-x^2z^3-\frac{9}{80}y^2z^3$
Linked Tori	$g(10x,10y-2,10z,13)g(10z,10y+2,10x,13)+1000, \quad g(x,y,z,c)=(x^2+y^2+z^2+c)^2-53(x^2+y^2)$
Barth Decic	$8(x^2-\phi^4y^2)(y^2-\phi^4z^2)(z^2-\phi^4x^2)(x^4+y^4+z^4-2x^2y^2-2x^2z^2-2y^2z^2)+(3+5\phi)(x^2+y^2+z^2-1)^2(x^2+y^2+z^2-(2-\phi))^2, \quad \phi=\frac{1}{2}(1+\sqrt{5})$
Super-sphere	$x^{16}+y^{16}+z^{16}-0.0001$

Table 3: Our test surfaces.

[Far02] FARIN G.: *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann Publ. Inc., 2002.

[FR87] FAROUKI R., RAJAN V.: On the numerical condition of polynomials in Bernstein form. *Comp. Aided Geom. Des. 4*, 3 (1987), 191–216.

[FR88] FAROUKI R., RAJAN V.: On the numerical condition of algebraic curves and surfaces 1. implicit equations. *Comp. Aided Geom. Des. 5*, 3 (1988), 215–252.

[Han83] HANRAHAN P.: Ray tracing algebraic surfaces. In *SIGGRAPH '83* (1983), ACM Press, pp. 83–90.

[Har96] HART J. C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 10 (1996), 527–545.

[IK66] ISAACSON E., KELLER H.: *Analysis of Numerical Methods*. John Wiley and Sons, 1966.

[KB89] KALRA D., BARR A. H.: Guaranteed ray intersections with implicit surfaces. In *SIGGRAPH '89* (1989), ACM Press, pp. 297–306.

[KHH*07] KNOLL A., HIJAZI Y., HANSEN C. D., WALD I., HAGEN H.: Interactive Ray Tracing of Arbitrary Implicit Functions. In *Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 11–17.

[LB06] LOOP C., BLINN J.: Real-time GPU rendering of piecewise algebraic surfaces. In *SIGGRAPH '06* (2006), ACM Press, pp. 664–670.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87* (1987), ACM Press, pp. 163–169.

[LR81] LANE J. M., RIESENFELD R. F.: Bounds on a polynomial. *BIT 21* (1981), 112–117.

[Mit90] MITCHELL D. P.: Robust ray intersection with interval arithmetic. In *Proc. on Graph. interface '90* (1990), Canadian Information Processing Society, pp. 68–74.

[Mør96] MØRKEN K.: *Total Positivity and its Applications*. Kluwer Academic Publishers, 1996, ch. Total Positivity and Splines, pp. 47–84.

[MR07] MØRKEN K., REIMERS M.: An unconditionally convergent method for computing zeros of splines and polynomials. *Math. of Comp. 76* (2007), 845–865.

[MR08] MØRKEN K., REIMERS M.: Second order multiple root finding. In progress, 2008.

[NSK90] NISHITA T., SEDERBERG T. W., KAKIMOTO M.: Ray tracing trimmed rational surface patches. In *SIGGRAPH '90* (1990), ACM, pp. 337–345.

[RHD89] ROCKWOOD A., HEATON K., DAVIS T.: Real-time rendering of trimmed surfaces. In *SIGGRAPH '89* (1989), ACM Press, pp. 107–116.

[Sch90] SCHNEIDER P. J.: *Graphics gems*. Academic Press Professional, Inc., 1990, ch. A Bézier curve-based root-finder, pp. 408–415.

[SD07] SELAND J., DOKKEN T.: *Geometrical Modeling, Numerical Simulation and Optimization*. Springer, 2007, ch. Real-Time Algebraic Surface Visualization.

[Spe94] SPENCER M. R.: *Polynomial real root finding in Bernstein form*. PhD thesis, Brigham Young University, Provo, UT, USA, 1994.

[SZ89] SEDERBERG T. W., ZUNDEL A. K.: Scan line display of algebraic surfaces. In *SIGGRAPH '89* (1989), ACM Press, pp. 147–156.

[Wu91] WU X.: An efficient antialiasing technique. In *SIGGRAPH '91* (1991), ACM Press, pp. 143–152.