REAL TIME RENDERING OF IMPLICIT SURFACES ON THE GPU

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science (by Research)
in
Computer Science

by

Jag Mohan Singh
200507013
jagmohan@research.iiit.ac.in

International Institute of Information Technology
Hyderabad, India
July 2008

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Real time Rendering of Implicit Surfaces on the GPU" by Jag Mohan Singh, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Advisor: Dr. P. J. Narayanan

To my parents

Acknowledgements

I would like to thank Professor P. J. Narayanan for his support and guidance during the past three years. I gratefully acknowledge Prof. P. J. Narayanan for long hours of discussion on the problems presented in the thesis.

I am also grateful to fellow lab mates at the CVIT, IIIT Hyderabad for their stimulating company during the past years. These include technical discussion on problems with Vishesh, Paresh, Pulkit, Sunil Mohan Ranta, Suryakant, Shiben, Avinash Kumar and Tarun. Also, I would like to thank other lab mates Pooja, Pradhee, Sachin, Avinash Sharma, Santosh, Jyotirmoy, Vibhav, Prachi, Chayya, Mihir and Dileep.

Abstract

Generating visually realistic looking models is one of the core problems of Computer Graphics. Rasterization or scan converting the primitives used such as triangles is one method to render them. This method suffers from problems of an inexact representation as triangles themselves are an approximation of the underlying geometry. Ray tracing primitives is another method of rendering the objects. This method delivers exact representation of the underlying geometry and looks visually realistic. We thus use ray tracing of implicit surfaces rather than polygonizing them. The programmable graphics processor units (GPUs) have high computation capabilities but relatively limited bandwidth for data access. Compact representation of geometry using a suitable procedural or mathematical model and a ray-tracing mode of rendering fit the GPUs well, consequently. An implicit surface can be represented as $S(x, y, z) = 0$ and the ray dependent equation is $F_f(t) = 0$. Ray tracing $S(x, y, z) = 0$ is root computation of $F_f(t) = 0$ for all the pixels on the screen. Analytical methods can be used in surfaces up to order 4. We compute interval extension of functions exactly by computing the function at points of maxima and minima and end points. Since, we can compute roots of functions up to order 4 we can compute points of maxima and minima of functions up to order 5. We use interval arithmetic for surfaces up to order 5 using Mitchell's algorithm. Interval methods provide a robust way for root isolation. Marching points algorithm marches in equal stepsizes until the root is found which is detected by a sign change in the function. Marching points wastes computation by computing the function values at many points. *Adaptive marching points algorithm* marches adaptively to find the root. Though only fourth or lower order surfaces can be rendered using analytical roots, our *adaptive marching points* algorithm can ray-trace arbitrary implicit surfaces exactly, by sampling the ray at selected points till a root is found. Adapting the sampling step size based on a proximity measure and a horizon measure delivers high speed. The horizon measure helps in silhouette adaptation and provides good quality silhouettes. We also provide a *taylor test* which has flavours of interval arithmetic and helps in robust rendering of surfaces using *adaptive marching points algorithm*. While computing the function $S(x, y, z) = 0$ we never compute the ray dependent $F_f(t) = 0$ by using coefficients of t. We save lot of computational

overhead by computing $S(x, y, z) = 0$ directly instead as there are $O(d^3)$ coefficients for t where d is the degree of the surface. In our method we don't need coefficients of t which are expensive to compute we only need the value $S(x, y, z) = 0$. The derivative $F'_f(t)$ can also be calculated efficiently using the gradient of $S()$ as $\vec{\nabla}S(x, y, z) \cdot D_f$. The Barth decic can be evaluated using about 30 terms as $S(x, y, z)$ but needs to evaluate 1373 terms to compute all 11 coefficients of the tenth order polynomial $F_f(t)$. We render Dynamic Implicit Surfaces which vary with time. Overall, a simple algorithm that fits the SIMD architecture of the GPU results in high performance. We ray-trace algebraic surfaces up to order 18 and non-algebraic surfaces including a Blinn's blobby with 30 spheres at better than interactive frame rates. Our *adaptive marching points* is an ideal match for the SIMD model of GPU due to low computational cost required per operation.

We use analytical methods for ray tracing surfaces up to order 4. We achieve fps of 3750 on a cubic surface and 1400 on a quartic surface. We use the robust Mitchell method on surfaces up to order 5 and achieve fps up to 400 on a torus quartic and 85 on a quintic surface. Our adaptive marching points method renders high order implicit surfaces at interactive frame rates. We render surface of order 18 at an fps of 158. These experiments used NVIDIA 8800 GTX at a resolution of 512×512. Our *GPU Objects* renders Bunny with 35,947 spheres at 57 fps, 99,130 spheres is rendered at 30 fps and Hyperboloid with reflection and refraction at 300 fps. NVIDIA 6600 GTX was used in experiments related to *GPU Objects* and the viewport was of the size 512×512.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Current Graphics Processor Units (GPUs) are optimized to render polygons. Programmability in the vertex, geometry, and pixel stages has made it possible for them to go beyond polygon rendering. They have been used for graphics effects like per-pixel lighting, ray-tracing, toon rendering as well as to general purpose computing. Ray-tracing is of particular interest as each fragment can be considered to be dealing with an imaging ray. Surfaces defined procedurally or implicitly can be rendered directly using ray-tracing on the GPUs, if the resulting functional form can be solved on the fragment processor. Procedural geometry is evaluated on the fly and has many benefits over polygonal geometry. Computationally, the overhead of tessellation is removed and visually, the geometry is rendered exactly at all resolutions. Procedural geometry is also compact in representation needing less bandwidth to send to the GPU. General, recursive ray-tracing is difficult on the GPUs. Simple algorithms that fit their restricted architecture will have higher performance than those that are efficient on a general purpose processor.

Implicit and procedural geometry has been studied well by the graphics community, in spite of the dominance of polygonal geometry. Implicit geometry is defined by an equation $S(x, y, z) = 0$. Different forms of $S(\cdot)$ are possible including polynomial, sinusoidal, transcendental, etc. An algebraic surface is defined as the roots of the polynomial $S(x, y, z) = \sum_m a_m x^{i_m} y^{j_m} z^{k_m} = 0$ and its order is $\max_m(i_m + j_m + k_m)$ over all terms. Non-algebraic surfaces can be of different functional forms. Implicit surfaces are popular in fluid simulation, scientific computing, weather modelling, etc. They are often used to visualize high-dimensional

data after fitting them with a suitable implicit function. Lower-order algebraic and simple non-algebraic functions have been used for visualization previously, but the use of higher-order surfaces is not common due to the difficulty in rendering them.

The fragment processors do most of the work in GPU-based ray-tracing. The ray parameters and the surface equation are needed at each fragment shader program. The points on the ray for a fragment f are given in the parametric form by $P = O + tD_f$, where t is the ray parameter, O the camera center, and D_f the direction of the ray (Figure 1.1). Substituting for x, y, z from the ray equation into the surface equation $S(x, y, z) = 0$, we get

$$F_f(t) = 0. \tag{1.1}$$

The smallest, real, positive solution for t gives the point of intersection of the ray with the object if there are multiple roots. Each fragment shader can independently find the root using a suitable method. The normal of the surface at the point of intersection can also be computed as the gradient $\vec{\nabla} S(x, y, z)$ for exact lighting and shadows. Figure 1.1 shows eye ray intersecting with a surface, both the roots (P_1, P_2) are positive in this case and smaller (P_1) is the solution in this case where the normal is computed for shading.

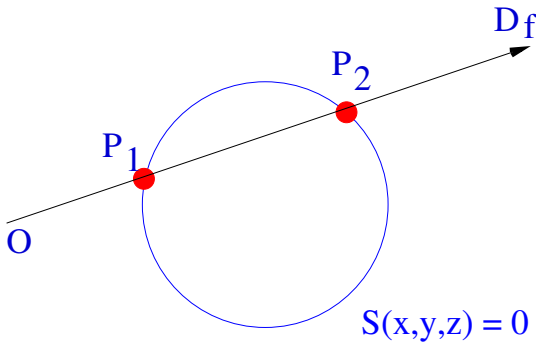


Figure 1.1: Ray Intersection with an Implicit Surface $S(x, y, z) = 0$ Red dots show the real roots.

Polygonization is the most common method of rendering implicit surfaces [5]. Particle-rendering have also been used for this task [53]. Dynamic implicit surfaces with changing topology pose great challenges to this process. The implicit form allows compact and exact defintion of surfaces. Converting them to triangles or particles compromises on both

compactness and exactness. Exactness can be retained by the use of large numbers of micro-triangles, but at the total loss of compactness. Direct rendering using ray tracing performed on the GPU can retain both advantages. The computing power of the GPUs is growing at over double the rate predicted by Moore's law, while the bandwidth from the CPU to the GPU is lagging behind seriously. Thus, compact representations that are light on communications and ray-tracing like techniques that are heavy on computations will suit them ideally. Computationally simple methods for ray-tracing are needed for today's GPUs due to their constrained architecture and SIMD (Single Instruction, Multiple Data) programming model.

Multicore and manycore computing is the current mechanism to take advantage of the continued developments in chip technology. Ray-tracing is projected as an application ideally suited to the high computing and low memory performance of such architectures [39]. Woop et al. argue for a programmable ray-tracing unit much like the GPUs and show an implementation using FPGAs [54]. Whitted and Kajiya propose using only procedural elements in a graphics pipeline to match the high computation power and the low external bandwidth of the GPUs [51]. They suggest extensions to the graphics hardware including a programmable sampler to replace the rasterizer. Our work strongly endorses this line of thinking by extending exact and high-quality ray-tracing to arbitrary implicit surfaces on the GPUs. The future of high-performance graphics is likely to be in modelling using procedural or implicit techniques and rendering using ray-tracing.

We explore real-time ray-tracing of arbitrary implicit surfaces on a modern GPU, beyond the low-order algebraic and simple non-algebraic surfaces reported in the literature. The basic idea is to reduce the surface $S(x, y, z) = 0$ to the form $F_f(t) = 0$ using the ray equation for the fragment f, where t is the ray-parameter. Each fragment can then solve for t and perform per-pixel lighting, shadowing, etc., based on the exact intersection. Solution to the equation $F_f(t) = 0$ depends on its form. Interactive ray-tracing has been achieved only for simpler implicit forms. These include algebraic surfaces up to order 4 using analytical roots on the GPU [26] and selected algebraic surfaces up to order 6 and some non-algebraic surfaces using interval-analysis on the SSE hardware [22].

1.1 Contributions of the thesis

We present fast ray tracing of algebraic surfaces of order less than five using closed-form, analytic solutions at higher frame rates – exceeding 1000 per second on an Nvidia 8800 GTX – than reported before. We also present the first adaptation of the Mitchell's interval-based method to the GPU using exact interval extension and ray trace algebraic surfaces of order less than 6 at frame rates that are at least an order of magnitude more than reported before. We introduce ray-tracing algorithms that sample the ray to find an interval containing the intersection with the surface. The *marching points* algorithm that samples each ray uniformly in t to find solutions of the equation $S(x, y, z) = S(p(t)) = 0$ and the *adaptive marching points* that samples each ray non-uniformly based on the distance to the surface and the closeness to a silhouette. These methods have the flavour of brute-force linear searching but are fast due to the low computational requirements and better match with the SIMD architecture of the GPUs. They can also handle arbitrary implicit surfaces easily, needing only to evaluate $S(x, y, z)$ at the sample points. In fact, a key finding of this thesis is that simple and seemingly non-promising algorithms that suit the architecture well can deliver very high performance on the GPUs. We also present a root-containment test that combines the simplicity of ray sampling with the robustness of interval analysis using a first-order Taylor expansion of the function. This results in a fast, versatile, and robust ray-tracing scheme. We ray-trace algebraic surfaces of order up to 18 and non-algebraic surfaces like super-quadrics, sinusoids, and blobbies with exact lighting and shadowing at significantly better than real-time rates. Figure 1.2 presents some of the surfaces ray-traced using our method.

1.2 Organization of thesis

Chapter 2 reviews the previous work related to the topic of this thesis. GPU implementations of analytic root finding is dealt in Chapter 3. Interval-based root-finding is presented in Chapter 4. Chapter 5 presents our sampling based methods that provide speed and versatility. Results of our algorithm on different algebraic and non-algebraic surfaces is presented

in Chapter 6. Chapter 7 presents a comparison and a discussion on ray-tracing on the GPUs and the CPU. Conclusions and directions for future work are presented in Chapter 8. Appendix A presents the equations of the implicit surfaces used in the thesis along with simple screenshots.

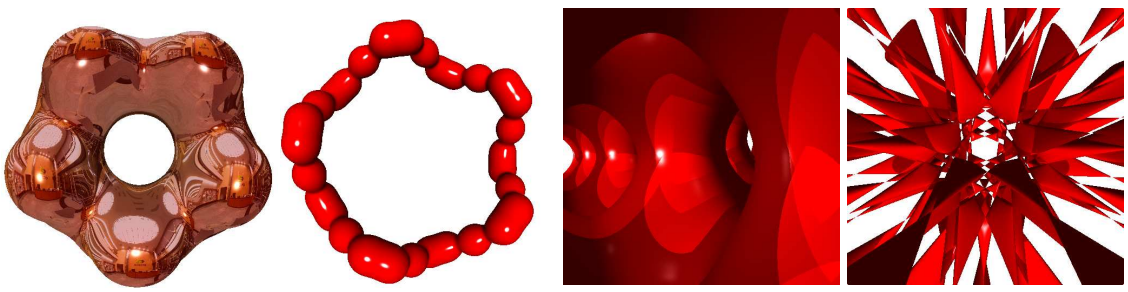


Figure 1.2: Ray-traced Blinn's blobby with 30 spheres with environment-mapping and shading (34 fps), Chmutov dodecic with four light sources (90 fps), and Barth decic with one light source (100 fps).

Chapter 2

Related Work

We review the related work divided into four sections: root finding for general functions, rendering implicit surfaces, ray-tracing implicit surfaces, and ray-tracing on the GPU.

2.1 Root Finding

Iterative root finding methods are used widely to solve general implicit equations in one variable. Analytical solutions exist for polynomials of order four or lower; only iterative solutions exist for higher order polynomials [3, 4] and other implicit forms. Loop and Blinn compute the analytical roots of equations up to fourth order [26]. We describe their method in detail. It mainly consists of the following steps: (a) The polynomial coefficients are computed in Bernstein bases as they allow for more robust root computation. (b) The real roots are computed to determine if the surface is visible. (c) Then they determine whether the visible surface is inside tetrahedron. (d) Then they compute surface normal and apply shading.

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

$$\sum_{i+j+k+l=d} b_{ijkl} \begin{pmatrix} d \\ ijkl \end{pmatrix} r^i s^j t^k u^l = 0 \tag{2.1}$$

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

$$r^{\alpha_1} \cdots r^{\alpha_d} B_{\alpha_1 \cdots \alpha_d} = 0 \tag{2.2}$$

where B is a symmetric rank d tensor containing Bezier weights, and $r = [r\ s\ t\ u]$. They transform both the Bezier tetrahedron vertices T and weights B to screen space for rendering. Transforming the bounding tetrahedron T and weight tensor B to screen space for rendering has the advantage that all viewing rays become parallel to z axis. They take clever advantage of the linear interpolation of vertex attributes at each pixel to interpolated coefficients of B. Thus, they arrive at the equation whose roots are required at each pixel in Bernstein bases. They then depress the polynomial by translating it in parameter space to make a new polynomial with a_i coefficients but a_{d-1} is zero. This is done by a simple matrix transformation. After this the quartic roots are found out by the Ferrari technique and cubic roots are found out using discriminant which are described in Chapter 3.

Iterative methods critically depend on good initialization of the roots, which is difficulty for complex equations. An alternative to initialization is to bracket the roots to an interval in t [36, 40] and then solve it using an iterative technique. Newton-Raphson, Newton-Bisection, and Laguerre's method are popular for ray tracing [36, 20, 29, 55]. Newton Raphson iterations approximate root as $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. The convergence of this method for the root is generally quadratic. However, if the initial approximation is too far from the actual root it will fail to converge. Also, if the derivative is discontinuous at the root it will fail to converge. Newton-Bisection starts with an interval $[a, b]$ in which root exists and creates two new intervals $[a, (a+b)/2]$ and $[(a+b)/2, b]$ and recurses into the one where function changes sign. Newton-Bisection method has slower convergence to the root than Newton Raphson method. However, it is guaranteed to converge to the root if it exists in the initial interval.

Laguerre methods is as follows: Compute $G = \frac{f'(x_k)}{f(x_k)}$. Compute $H = G^2 - \frac{f"(x_k)}{f(x_k)}$. Compute $a = \frac{n}{G \pm \sqrt{(n-1)(nH-G^2)}}$ where sign is chosen such that it results in maximum absolute value of the denominator. Compute $x_{k+1} = x_k - a$. For a single root, Laguerre's method converges cubically whereas for multiple roots the convergence is only linear. One advantage of Laguerre's method is that it is guaranteed to converge to the root irrespective of the initial value given. Extensions of these to use interval arithmetic have also been used [16, 24], which are more robust at critical regions.

Interval analysis has been used for robust root finding [16]. Consider an interval X that

contains the root x^* of f, where f is continuously differentiable. Let F' be interval extension of f' and assuming F'(X) does not contain zero. $x^* \in N(x,X) \equiv x - \frac{f(x)}{F'(X)}$ $x^* \in X \cap N(x,X)$. Thus, interval version of Newton's method is : Start with $X_{(0)}$ containing x^* and compute a nested set of intervals $X_{(1)}, X_{(2)}, \cdots$ by the formula: $X_{(s+1)} = X_{(s)} \cap N(x_{(s)}, X_{(s)})$ with $x_{(s)} \in X_{(s)}, s = 0, 1, \cdots$. Krawczyk's method is similar and the operator is: $K(x,X) \equiv x - Hf(x) + (I - HJ(X))(X - x)$ where H is an arbitrary matrix in $R^{n \times n}$. H is generally set to $1/f'(x)$. If we start with $X_{(0)}$ containing the root x^*, then the formula: $X_{(s+1)} = X_{(s)} \cap K(x_{(s)}, X_{(s)})$ with $x_{(s)} \in X_{(s)}, s = 0, 1, \cdots$. Interval version of Newton's method demands that $0 \notin F'(X_{(s)})$ whereas Krawczyk method only demands that $f'(x_{(s)}) \neq 0$. Krawczyk's method has better convergence properties than the interval version of Newton's method. Auxiliary polynomials [42] and Sturm sequences [52, 32] have also been used to find roots of polynomials of various degrees. Most of these methods cannot be implemented easily on the SIMD architecture of the GPUs, however.

2.2 Rendering Implicit Surfaces

Polygonization can convert implicit surfaces into triangulated model prior to rendering them using traditional graphics [5]. The marching cubes algorithm can be used to create polygonal models from implicit functions [28]. Marching cubes algorithm proceeds through the scalar field , taking eight neighbor locations at a time (thus forming an imaginary cube), then determining the polygons needed to represent the part of the isosurface that passes through this cube. This is done by creating an index to a precalculated array of 256 possible polygon configurations ($2^8 = 256$) within the cube, by treating each of the 8 scalar values as a bit in an 8-bit integer. The precalculated array of 256 cube configurations can be obtained by reflections and symmetrical rotations of 15 unique cases. The gradient of the scalar field at each grid point is also the normal vector of a hypothetical isosurface passing from that point. Therefore, we may interpolate these normals along the edges of each cube to find the normals of the generated vertices which are essential for shading the resulting mesh with some illumination model. However, marching cubes fails to polygonize on many pathological surfaces on which our method works. This is particularly true for surfaces having

discontinuous or complex gradients. Green et al. released a high-performance, marching tetrahedra package on the GPU recently [14], which can be used to polygonalize and render arbitrary surfaces. In practice, this method is much slower than our approach and does not produce good results on complex surfaces due to severe resolution problems (Chapter 6.4). Twinned meshes were introduced recently to triangulate dynamic implicit surfaces with changing topology using a mechanical mesh and a geometric mesh [6]. Point or particle-based sampling and rendering of implicit surfaces have also been popular [53, 49]. They distribute particles on the surface and apply attractive and repulsive forces to distribute them evenly on the surface. These methods are typically demonstrated on metaball or blobby surfaces used widely for fluid simulations and do not extend to arbitrary implicit surfaces well. Triangulation and point-sampling go against the strengths of the GPU by increasing the size and the bandwidth needs of the representation.

2.3 Ray-Tracing Implicit Surfaces

Ray-tracing of implicit surfaces is about finding the smallest positive root of an appropriate equation in the ray-parameter t. Hanrahan demonstrated ray tracing of algebraic surfaces up to the fourth order using Descartes rule of signs for root isolation and Newton's bisection for root refinement [15]. Kajiya reduces ray tracing of spline surfaces to a globally convergent method resulting in root finding of an 18th degree polynomial using the Laguerre's method [20]. Kajiya's method treats the ray as intersection of two planes. Thus, intersection of ray and a spline reduces to intersection of intersection of two spline curves which are formed by intersecting the spline surface with the planes forming the ray. These two spline curves are intersected using Bezout determinant form which reduces to finding roots of a univariate 18th degree polynomial. The roots are found out by intersecting two cubic curves. Once all the points of intersection of two curves are found out the nearest point of intersection is reported. This point corresponds to the intersection of ray with the bicubic surface.

Interval arithmetic treats numbers as within some range thus t_0 is represented as $[t_0 - \delta, t_0 + \delta]$. It has specific rules for addition, subtraction, multiplication and division of two interval based numbers. Let $A = [\underline{a}, \overline{a}]$ and $B = [\underline{b}, \overline{b}]$ be two intervals. Then the arithmetic

operations between them are as follows:

$$A + B = [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \tag{2.3}$$

$$A - B = [\underline{a} - \overline{b}, \overline{a} - \underline{b}] \tag{2.4}$$

$$A * B = [min(\underline{a}\overline{b}, \overline{a}\underline{b}, \underline{ab}, \overline{a}\overline{b}), max(\underline{a}\overline{b}, \overline{a}\underline{b}, \underline{ab}, \overline{a}\overline{b})] \tag{2.5}$$

$$A/B = A * [1/\overline{b}, 1/\underline{b}] \tag{2.6}$$

It is more robust than normal arithmetic. There are different types of methods by which one can *interval extend* normal functions these include natural interval extension and more robust taylor series based interval extensions. Interval-analysis has also been used for robust root isolation by many [30, 11, 7, 41, 12, 22]. Caprani et al. use robust methods which are obtained with interval inclusions in a variant of Alefeld-Hansens globally convergent method for computing and bounding all the roots of a single equation [10]. Alefeld-Hansens method has been modified so instead of searching for all roots, a recursive depth-first search is carried out to obtain the smallest non-negative root. When compared to other methods suggested, it is found that this variant of Alefeld-Hansens method is not only robust but also an efficient method for finding the ray intersections [7].

Florez et al. use trimming algorithm which uses interval analysis to perform rejection tests in a set of pixels simultaneously, instead of individual pixels at each time. With this approach, the presented algorithm by them runs faster than the traditional interval ray tracing algorithm. Also, an interval algorithm to remove aliasing in the rendering of implicit surfaces is introduced by them. Their algorithm obtains better visualizations than the traditional point sampling [12].

The interval extension of a function gives a bound in its range given an interval in its domain. Mitchell isolates the root using repeated bisections till the interval in t contains a single root [30]. Mitchell algorithm computes the interval extension of the function and it's dervitaive. The condition for single root which can be found out using different root refinement methods according to Mitchell is that interval extension of the function contains a zero whereas the interval extension of the derivative does not contain a zero. The condition for multiple roots is that interval extension of the function and interval extension of the

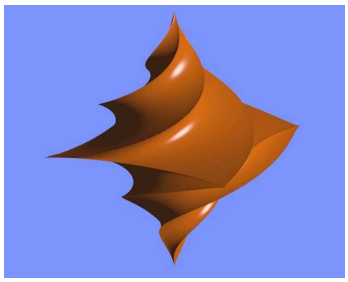


Figure 2.1: Twisted Superquadric rendered using Mitchell's algorithm [30]

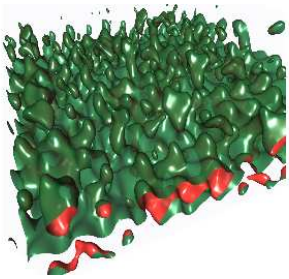


Figure 2.2: An animated sinusoid-kernel surface [23]

derivative both contain a zero. In case of multiple roots the interval is recursively subdivided into left subinterval and right subinterval. The right subinterval is chosen only in the case interval extension of the function in left subinterval does not contain a zero. The recursive subdivision proceeds until width of subinterval crosses some lower limit. This scheme gives robust roots as it uses interval arithmetic as compared to normal arithmetic. Figure 2.1 shows robust rendering of twisted superquadric.

Reliable interval-extensions, however, are difficult to compute for large intervals in the domain of complex functions. Subintervals, branch and bound schemes, octree grids, etc., have been used to increase the reliability of interval-based methods. Sherstyuk ray traces implicit surfaces by approximating $F_f(t)$ using piecewise Hermite polynomials of order 4 or lower and solving them using analytical methods [44]. Snyder poses problems including ray tracing implicit surfaces and CSG as constraint minimization and performs a numerically stable global minimization using an interval analysis based method [46]. These results have been on the CPU mostly and involve low-order algebraic surfaces or small blobbies. Knoll et al. achieve 30 fps on a superquadric and 6 fps on a few sextic surfaces using the CPU and the SSE hardware [22]. The interval-based methods can be adapted to the GPU for faster

ray tracing, as we show in Section 4, but are limited in scope due to the difficulty of reliable interval extensions for higher order algebraic and non-algebraic surfaces. Simple methods that suit the GPU deliver high performance on them.

Affine arithmetic provides tighter inclusions on the function than their interval arithmetic counterparts. Knoll et al. achieve 121 fps on teardrop quintic surface, 88 fps on Barth sextic and 15.6 fps on Barth decic surface [23](2.2. They use combination of rejection test and bisection to arrive at the correct root. They compute the affine arithmetic based extension of the function (F). If $0 \in F$ the rejection test succeeds and there is a root in this interval. In case the rejection test succeeds they compute maximum depth of bisection (d_max) based on user specified precision ϵ. If the test succeeds they hit the surface if they have reached maximum depth of bisection $d = d_{max}$ or recurse to the next level by setting the stepsize $t_{incr} = t_{incr}/2$ and incrementing the depth d. The back recursion step decrements the depth and goes to other unvisited nodes of the bisection tree. In the worst case their scheme can lead to visiting all the nodes in the bisection tree and the algorithm can be very slow.

Sampling points along the ray and looking for intersections is a simple and intuitive way to isolate the smallest positive root. This approach has been used for procedural hypertextures [33] and other implicit surfaces [21, 17]. Kalra and Barr ray-traced LG-implicit surfaces using Lipschitz constants, L for the function $S(x, y, z)$ and G for its derivative along the ray $F'(t)$, for efficient sampling of the rays [21]. A positive real number L is called a Lipschitz constant on a function $f(x)$ in a region R, if given any two points x_1 and x_2 the following condition holds

$$||f(x_1) - f(x_2)|| < L||x_1 - x_2|| \tag{2.7}$$

L is equal to greater than the maximum rate of change of $f(x)$ in R i.e. $L \geq max_R|\nabla f(x)|$ Given a one dimensional closed interval $\tau \in [t_1, t_2]$, Lipschitz constant G for $g(t)$ in τ to be equal to or greater than the maximum rate of change of $g(t)$ in τ i.e. $G \geq max_\tau|\frac{dg}{dt}|$. For surfaces with known L and G the algorithm has no problem rendering the finest features Figure 2.3.

Hart used variable step sizes in sphere tracing based on a geometric distance function evaluated at the current point [17]. It is able to handle surfaces defined by functions with discontinuous or undefined derivatives. In order to achieve this it requires bounds on the

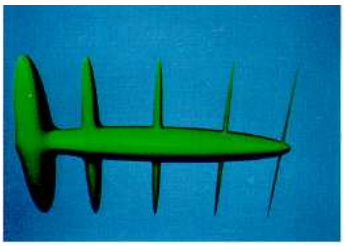


Figure 2.3: A surface with fine features rendered using LG technique

Figure 2.4: A logo for zeno using sphere tracing technique

magnitude of the derivative. Figure 2.4 shows logo for zeno rendered using sphere tracing technique. The Lipschitz theory or geometric distances do not extend easily to complex surfaces, however. Our algorithms also follow the point sampling approach, but change the step size using simpler measures that suit the GPU than optimal ones from Lipschitz theory.

2.4 Ray-Tracing on the GPU

The traditional ray-tracing technique has been adapted to the GPU for general polygonal models. Purcell et al. performed multipass ray tracing [37] and Carr et al. combined CPU and GPU computations for a variety of tasks including recursive ray tracing [8]. These methods work for general objects but are slow. Non-linear beam tracing on the GPU was demonstrated by Liu et al. for regular geometry [25]. Spheres and other quadric primitives were ray-traced

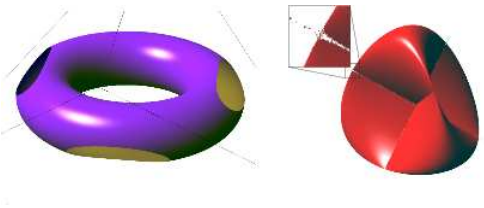


Figure 2.5: Torus (Left) and Steiner(Right) visualized using Loop and Blinn's technique. Steiner shows some problems which are shown in the inset.

on the GPU using per-fragment ray-quadric intersection and optimized bounding boxes [48, 45, 38]. Loop and Blinn showed resolution independent rendering of quadratic and cubic-spline curves on the GPU [27]. They evaluate quadratic and cubic curves in a space similar to the texture space. They also apply operations for antialiasing the curves using a signed distance expression. Incase, of overlapping curves they subdivide the the control point triangle of the one having larger area and get rid of the overlapping regions. They extended it to render piecewise algebraic surfaces up to fourth order defined over tetrahedral bases using analytical roots [26]. Figure 2.5 shows the rendering of torus and steiner quartics. The steiner quartic has some problems due to self intersections.

Bajaj et al. used cubic A-patches using Bernstein-Bezier bases within a tetrahedron as a way to approximate scattered points [2]. Seland and Dokken rendered algebraic surfaces up to order five on the GPU [43] by computing the blossom of the function with respect to each ray as a univariate Bernstein polynomial. This will not extend easily to higher order surfaces as the complexity of computing coefficients of the univariate polynomial increases rapidly with its degree. Computing coefficients occupies the majority of work done in root finding in their case. Our method keeps the process simple to match the GPU by not evaluating the coefficients of the complex univariate polynomials. Our method does not require this expensive coefficient computation.

2.5 Our Work in Context

The analytical root-finding and Mitchell's interval-based method are adapted to the GPU (Chapter 3). The analytical roots are computed directly on the GPU without subdivisions and achieve faster ray-tracing framerates on quartics and cubics than reported before. We present the first interval-based ray-tracing on the GPU with an exact interval-extension, which can ray-trace algebraic surfaces up to order 5 without any subdivisions or octrees and achieves frame-rates that are at least an order of magnitude higher than reported previously. The ray sampling methods we present (Chapter 5) steps along each ray till a step covers a root. Step size is adapted using the algebraic distance to the surface and the angle the ray makes with the local normal so that smaller steps are taken in regions that need greater

attention. For robustness, we use simple interval analysis to test root-containment. Evaluating the multivariate $S(x, y, z)$ polynomials instead of the complex and ray-dependent univariate $F(t)$ polynomials reduces the computation effort per pixel. The methods also work on arbitrary implicit surfaces since only samples of it are needed. The simplicity of the methods is the key to achieving high performance on the restricted parallel architecture of the GPU. We can ray-trace algebraic surfaces up to order 18 and several non-algebraic surfaces at framerates upwards of 100. The method works on dynamic implicit surfaces also as the equation is evaluated directly in each frame with no precomputations.

Chapter 3

GPU Ray-Tracing using Analytic Roots

In this section, we present the GPU implementation of analytic root-finding method and use it ray-trace quadratic, cubic and quartic surfaces.

3.1 Ray Tracing Quadrics

Quadratic surfaces are defined by $p^TQp = 0$ where Q is a 4×4 matrix. Substituting $p = O + tD_f$, the $F_f(t) = 0$ reduces to $At^2 + Bt + C = 0$. Each fragment or ray has a different equation, which can be solved independently. We solve the quadratic equation $At^2+Bt+C = 0$ using discriminant $D = B^2 - 4AC$ to get the roots as $\frac{-B\pm\sqrt{D}}{2A}$. The smaller positive of these two roots gives the rays intersection with surface. All quadrics can be ray-casted by solving a quadratic equation. We demonstrate results using ray casting large number of quadrics. We also demonstrate results on ray casting quadrics with reflection and refraction and environment mapping[38].

3.2 CSG of Quadric Objects

A CSG object is defined by a set of solid primitives and an expression tree in which leaf nodes are the primitives and non leaf nodes are set operations. The set operations which we

use in our CSG tree are intersection ($\cap$), union ($\cup$) and difference ($\setminus$). The CSG tree defines a sequence in which these set operations must be performed to get the desired CSG object. An arbitrary CSG tree can be transformed to produce a new tree which is the normal form of the tree.

The normalized tree produces an expression which is in *sum-of-products* form. This can be expressed as union of products $P_1 \cup \cdots \cup P_m$ where each P_i is a product of primitives. Each product term can be expressed as $X_0 op_1 X_1 op_2 \cdots op_k X_k$ where each X_i is a primitive and op_i is either $\setminus$ or $\cap$. We use Goldfeather et. al's approach [13] to normalize the CSG tree. A single product can rewritten as intersection of primitives where each can be uncomplemented or complemented as for primitives A, B and $A \setminus B$ is same as $A \cap \overline{B}$. Thus, the rendering of CSG expression reduces to rendering of union of products. The CPU normalizes the CSG expression using Goldfeather et. al's approach into union of products. Thus, we have a sum of products which has to be rendered. This is done in two steps first each product term which is passed from CPU is sent to GPU to be rendered. Then union of these product terms has to be computed. Render CSG Object procedure is described in Algorithm 1.

Algorithm 1 Render CSG Object

1: Convert the CSG expression to normalized form which is in *sum-of-products* using Goldfeather et. al's [13] approach.
2: Render each product term using Algorithm 2.
3: Combine them using z-buffering. This is done by using the depth test which will provide us with nearest product term for each pixel with correct depth and colour which were computed during rendering of each individual product.

We render procedural objects in such a way that bounding box is computed in vertex shader. The intersection is computed in the pixel shader for each pixel, given the parameters of the object being rendered. This step is essentially the conventional ray casting implemented on the pixel shader. The points on the ray in parametric form can be represented as $P = O + tD$, where t is the parameter along the ray, O the camera center and D the direction of the ray. If the current pixel is accepted then its 3D position and normal are computed for per-pixel lighting. The primitives include 3D Shapes such as include Quadric,

Parallelepiped and Tetrahedron. The main advantage of procedural rendering is resolution independence which means that curved objects look curved at all resolutions independent of zooming.

3.2.1 Rendering of a Product Term

Now consider the product containing i terms where some may be in complemented form. We initialize the *membership vector* of the product by a bit vector of length i. We assume that the origin is outside all the objects being considered. Render Product Term procedure is described in Algorithm 2.

Algorithm 2 Render Product Term

1: Initialize *membership vector* to "outside" which is assumed to be 0 for the uncomplemented primitives and 1 for complemented primitives. This is stored in a boolean vector Int. The length of *membership vector* is equal to number of complemented terms+1.
2: Compute front and back intersection with each primitive procedurally t_f, t_b.
3: Now for the uncomplemented primitives find t_f^u, t_b^u where $t_f^u = max(t_f)$ and $t_b^u = min(t_b)$.
4: If $t_f^u > t_b^u$ exit product term is not true for the given ray.
5: Sort ${t_f}^i, {t_b}^i$ for all complemented primitives i and t_f^u, t_b^u.
6: From front, if $Int[j]$ i.e. intersection is with $j'th$ object, toggle $j'th$ bit.
7: **if** *membership vector* is all $1's$ **then**
8: Use normal of j for lighting. Light using material and other surface properties. Set the colour and depth
9: **else**
10: Proceed to the next intersection in the front to back order.
11: **end if**

Algorithm 2 is illustrated for the product term $AB\overline{C}$ Figure 3.1 then the *membership vector* is initialized to 01 this is by assuming the terms in *membership vector* to be 0 for all uncomplemented primitives and 1 for the complemented primitives. We denote the t parameters for intersection with each object by ${t_f}^1, {t_b}^1$, ${t_f}^2, {t_b}^2$ and ${t_f}^3, {t_b}^3$ where first is the smaller of the two parameters for each object. Now, for the uncomplemented primitives we

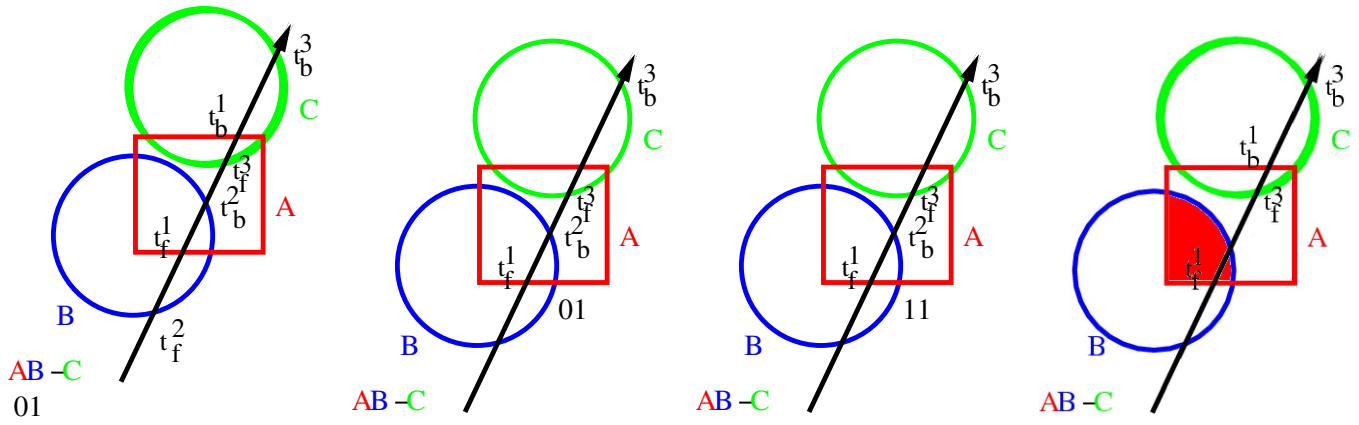


Figure 3.1: Ray Intersection with a product term(a) Initial Membership Vector (b) Membership Vector after combining uncomplemented terms (c) Membership Vector after intersection with front face of A (d) Final Product Term which shows the product surface filled in red

take the maximum of all entering intersections which in this case is ${t_f}^1$ and minimum of all exit intersections ${t_b}^2$. Thus after sorting the itersections for the ray we get the objects in the following order $ABCC$ and the intersection parameters ${t_f}^1, {t_b}^2, {t_f}^3, {t_b}^3$. Then the *membership vector* goes through following transitions $01, 11$ thus the ray intersects the product. The intersection point is given by the parameter ${t_f}^1$ and normal is the normal of A. Thus, the *membership vector* is toggled at most $2i$ times where i is the number of terms in the product which is number of uncomplemented primitives $+1$.

3.3 Ray Tracing Cubics and Quartics

Analytic solutions to the ray dependent equation exists only for simple forms of $F_f()$, such as polynomials of order less than five. Root-finding for quadric surfaces have been used for ray-tracing [48, 45, 38]. Results on cubics and quartics have been reported using tetrahedral bases to limit the search range for the roots [26].

We solve the cubic equation using the method given by Blinn [3, 4]. For a cubic equation $Ax^3 + 3Bx^2w + 3Cxw^2 + Dw^3 = 0$, compute δ $\delta_1 = AC - B^2$, $\delta_2 = AD - BC$, and $\delta_3 = BD - C^2$. The discriminant is defined as $\delta = 4\delta_1\delta_3 - \delta_2^2$. The sign of the discriminant

and the values of δ_is determine if it has one triple root, one double and a single real root, three distinct real roots, or one real root and one complex conjugate pair as roots. These can be worked out for each fragment independently for fast rendering of the shapes. We are able to achieve over 3000 fps on cubics as shown in Table 3.1.

We use the Ferrari method described by Herbison-Evans for the fourth order polynomials [18]. The equation is first depressed by removing the cubic term to the form $t^4 + pt^2 + qt + r = 0$. If r is zero, the roots are 0 and the roots of the cubic equation. If r is non-zero, the equation can be written as a product of two quadric equations. This is done by rewriting it as $(t^2 + p)^2 + qt + r = pt^2 + p^2$. This is followed by a substitution y such that the right hand side (RHS) becomes a perfect square. The equation then transforms to $(t^2 + p + y)^2 = (p + 2y)t^2 - qt + (y^2 + 2yp + p^2 - r)$. Now, for the RHS to be a perfect square its discriminant must be zero which leads to a cubic equation in y whose root is found as described before. Table 3.1 shows the frame rates for representative cubic and quartic surfaces for a resolution of 512×512 on an Nvidia 8800 GTX. Techniques specialized for spheres or ellipsoids achieve 15-30 fps on a scene with 99K tiny spheres but do not give the rendering time of a single primitive [45, 38]. Loop and Blinn report an fps of 1200 on single quadratics and about 500 on single quartics. Our results are 2-4 times faster than theirs on a 8800 GTX compared to the 7800 GTX they used. While most quartics work perfectly, surfaces with self intersections or multiple roots like Steiner, Cross-Cap, and Miter, have small holes from a few viewpoints. Multiple roots are hard for root-finding methods

Figure 3.2: Left: Ray-traced Bunny model containing 36k spheres at 57 fps Middle: Hyperboloid ray traced with reflection and refraction at 300 fps Right: Four primitive CSG at 22 fps.

Surface	FPS	Surface	FPS
Steiner Quartic	1400	Miter Quartic	1045
Torus Quartic	1200	Cross Cap Quartic	1025
Tooth Quartic	1100	Clebsch Cubic	3400
Goursat Quartic	1175	Cayley Cubic	3300
Cassini Quartic	1103	Ding-Dong Cubic	3750
Piriform Quartic	1082	Sphere Quadric	5821
Cylinder Quadric	4358		

Table 3.1: Frame rates for different surfaces using analytic root-finding for a 512×512 window on an Nvidia 8800 GTX.

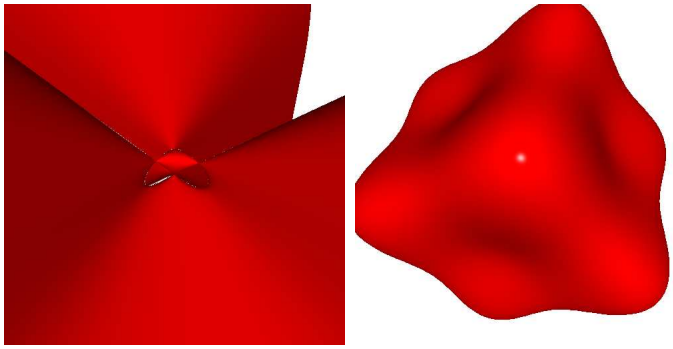


Figure 3.3: Left: Cayley cubic at 3300 fps Right: Tooth quartic at 1100 fps

in general. The analytical methods may be able to detect degenerate situations based on the determinants and take suitable action. Figure 3.3 shows a quartic and cubic rendered analytically.

Chapter 4

Interval-Based Root Isolation

Interval-based methods have been used for robust root-finding [1, 31]. The basic idea is to extend a function to an interval in its domain, with the result being an interval in its range. If $\tilde{x}$ is an interval $[a, b]$, the interval extension $\tilde{f}(\tilde{x})$ is an interval $[p, q]$ that encloses the minimum and maximum values of $f(x)$ for $x \in [a, b]$. Interval extensions of iterative root-finding methods produce more robust roots because they don't deal with possibly singular values.

Mitchell's algorithm for root finding recursively isolates an interval that contains the root by halving it, using interval arithmetic [30]. The interval extension of the function $F_f(t)$ and its derivative $F'_f(t)$ are used for this. If the interval extension of the function does not contain 0, then the corresponding interval does not have a root inside the interval. The interval contains one or more roots otherwise. If the interval extension of its derivative does not contain 0, then the function is monotonic in the interval and contains a single root. Otherwise (i.e., the interval extensions of the function and its derivative contain 0), the function has multiple roots in the interval. The interval is then bisected and the procedure is applied on both intervals recursively, starting with the lower half.

We adapted this algorithm for the GPU for root isolation (Algorithm 3). Since we are interested in the smallest positive real root, we check the second half of the interval only if the first does not contain a root. Algorithm 3 is executed on the GPU independently for each ray. We exploit the vector operations of the GPUs to implement interval arithmetic operations. The bisection at each step results in a running time that is logarithmic in the

length of the starting interval. We render several algebraic surfaces of order up to 5 and a few non-algebraic surfaces using the above algorithm. Table 4.1 shows the frame rate achieved on different surfaces. The average number of iterations vary from 18 for the Ding Dong cubic to 85 for the Dervish quintic using this method. The rendering speed is affected by this, but better than interactive rates is achieved on all these surfaces. The previous interval based methods on the CPU were limited to 4th order algebraic surfaces, superquadrics, Steiner surface, and blobbies. Recently, a sixth order surfaces was ray-traced at 6 fps and a superquadric at 30 fps using the CPU plus the SSE hardware [22]. Ours is the first reported attempt at implementing the interval-based ray tracing on the GPUs.

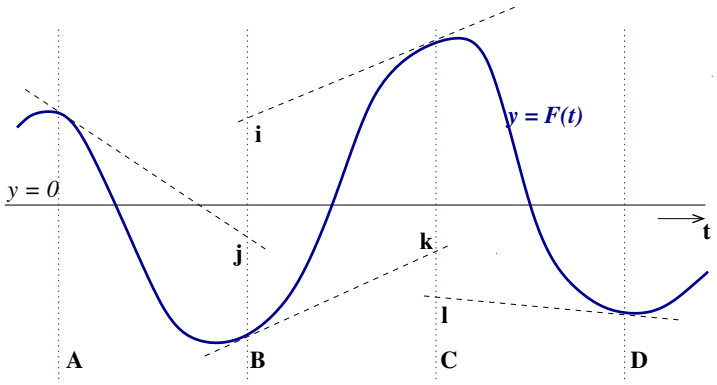


Figure 4.1: Interval extension methods. Roots in ranges [A, B], [B, C], and [C, D] can be found using the natural extension involving the function values at the end points only, which will not work for [A, C] or [B, D]. The first-order Taylor expansion based extension can work for [A, C]. All critical points of the function need to be evaluated to detect the root in the range [B, D]. The first-order extensions for [A, C] are i and j and for [B,D] are k and l.

The effectiveness of interval-based methods depends critically on the interval extension used. Finding the bounds in range of a function given an interval in its domain is a hard problem for arbitrary functions. Figure 4.1 illustrates the difficulty involved. Common methods like the natural and the centered interval extension [19] use the values of the function and/or its derivative at both ends of the interval. These can miss the root if an even number of roots are in the interval. Another option is to interval extend each argument independently and evaluate the function using interval-based addition, subtraction, multiplication, etc. The bounds generated by this method tend to overestimate the true bounds and can result in

Surface [order]	**No. of iterations**	**Frames per second**	**Surface [order]**	**No. of iterations**	**Frames per second**
Dervish [5]	86	60	Nordstrands [4]	47	220
Kiss [5]	65	77	Kummer [4]	45	225
Peninsula [5]	60	85	Steiner [4]	45	235
Cushion [4]	53	170	Piriform [4]	42	230
Cross-Cap [4]	52	195	Torus [4]	32	400
Miter [4]	52	186	Cayley [3]	27	580
Tooth [4]	50	195	Clebsch [3]	25	590
Cassini [4]	48	215	Ding-Dong [3]	18	965
Goursat [4]	47	210			

Table 4.1: Number of iterations and the frame rate using the interval-based method for a 512×512 window.

false roots when the domain interval is large, as is well known [22].

The bounds in the range of a function can be computed exactly only if all critical points in the interval are known. For algebraic surfaces, this reduces to finding *all* roots of a polynomial of order less by one. Consequently, we can only render algebraic surfaces of order 5 or less (Table 4.1). It should be noted that rendering a quintic surface involves solving for all roots of a 4th order polynomial to compute the interval extension of the function and solving for all roots of a 3rd order polynomial to do the same for its derivative, per fragment. Interval-based methods, thus, incur heavy computations and achieve lower framerates as a result.

The total interval $[t_s, t_e]$ is typically too large in practice for bisections to work using an inexact interval extension for complex surfaces. Subdivision of the interval can enable the reliable use of a simpler interval extension method. However, for arbitrary functions, such subdivision based on a proper analysis is difficult. The considerations are similar to those used in setting the optimal step-size of the marching points method. The Taylor series expansion based interval extension (Section 5.4) is one that works on smaller sub-intervals. Figure 4.2 shows a quintic and a quartic rendered using Mitchell's algorithm.

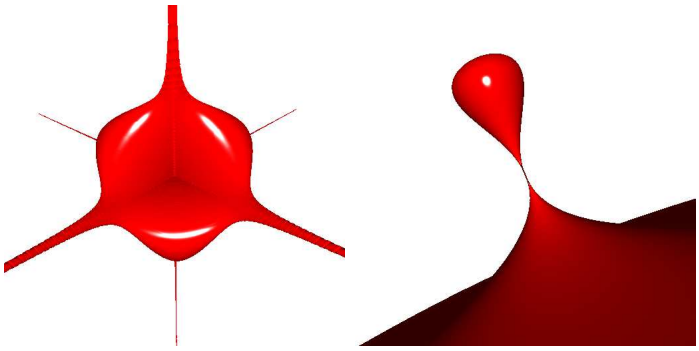


Figure 4.2: Left: Steiner quartic at 235 fps Right: Kiss quintic at 77 fps

Algorithm 3 Interval-Based Root Isolation (f, a, b)

Compute the interval extensions $\tilde{F}_f([a,b])$ and $\tilde{F}'_f[a,b]$.
if $0 \notin \tilde{F}_f([a,b])$ **then**
    Declare no root
else if $0 \in \tilde{F}_f([a,b])$ and $0 \notin \tilde{F}'_f([a,b])$ **then**
    Single root. Return $[a,b]$
else if $0 \in \tilde{F}_f([a,b])$ and $0 \in \tilde{F}'_f([a,b])$ **then**
    Multiple roots. Invoke the algorithm for the interval $[a, \frac{a+b}{2}]$.
    If no root, invoke algorithm for $[\frac{a+b}{2}, b]$.
    Until a root is isolated or the width of interval is less than ϵ.
end if

Chapter 5

Marching Points and Adaptive Marching Points

In this chapter, we describe our method to ray-trace arbitrary implicit surfaces, beyond those described earlier in this thesis. Our approach is ideally suited to the SIMD architecture of the GPU.

5.1 Iterative Root-Finding: Outline

Most interesting surfaces do not admit closed-form solutions and must be solved iteratively. Iterative methods need to be used with caution on implicit surfaces as the equation $F_f(t)$ may have many solutions. Standard iterative methods like Newton-Raphson, Laguerre, etc., need good initialization for convergence. A two-step process with root isolation followed by root refinement works better in the general case. Root isolation returns a bracket or interval in the domain in which a root is present. Root refinement isolates the root within that bracket. The total search range is $[t_s, t_e]$, the intersection of the ray with the near and far planes.

We present two root isolation methods in this section based on sampling the ray and another method in the next section based on interval analysis. A simple bisection method is used to refine the isolated root in all cases. The bisection method divides the given bracket $[t_1, t_2]$ into two sub-intervals $[t_1, t_m]$ and $[t_m, t_2]$ using the midpoint t_m. The smaller

Algorithm 4 Root-Finding: Overview

1: Isolate the first interval $[t_1, t_2]$ that contains a root for the ray corresponding to each fragment f.

2: Refine the root in $[t_1, t_2]$ using repeated bisections.

half that contains the root is identified and explored further recursively. We perform 10 bisections after root isolation, but a condition based on the value of $|F_f(t)|$ can be used instead. Bisection method is robust and succeeds in all cases, if the bracketing is correct [36]. It also guarantees that the solution gets closer to a real root with more iterations. Other iterative methods (like the Newton-Raphson method) that base the next estimate on the secant or the gradient converge faster in theory than the bisection method. However, they are computationally more expensive due to need for derivatives and do not suit the SIMD computation model of the GPUs well. In practice, the root refinement step is less critical for all surfaces. In our experience, only about 15% of the total time is spent on the second step for all surfaces with the percentage dropping below 10% for higher order algebraic surfaces.

5.2 Computing $S(x, y, z)$ vs $F_f(t)$

Root finding may need the values of the function $F_f(t)$ and possibly the first derivative $F'_f(t)$ and higher order ones at several points. The function can be evaluated for a given t using the univariate polynomial $F_f(t)$ directly or using the multivariate polynomial $S(x, y, z) = S(p(t))$ after computing $(x, y, z) = p(t)$ from t using the ray equation. The computational implications of each could be very different. The expression $F_f(t)$ typically has many terms for higher order polynomials. Note that the coefficients of $F_f()$ depend on the viewpoint and the pixel coordinates and cannot be precomputed. For example, a single sixth order expression x^3y^3 of $S()$ maps to $(a + bt)^3(c + dt)^3$ in $F_f(t)$ and expands to 16 terms for the 7 coefficients of the sixth order polynomial in t, requiring 44 multiplications and 9 additions to evaluate. On the other hand, x and y can be computed using 2 multiplications and 2 additions and x^3y^3 can be evaluated using 5 more multiplications.

The Barth decic (Section 6) can be evaluated using about 30 terms as $S(x, y, z)$ but needs to evaluate 1373 terms to compute all 11 coefficients of the tenth order polynomial $F_f(t)$. The derivative $F'_f(t)$ can also be calculated efficiently using the gradient of $S()$ as $\vec{\nabla}S(x, y, z) \cdot D_f$. The univariate expression for the derivative is as cumbersome as the expression for $F_f(t)$.

Loop and Blinn use GPU's interpolation hardware to evaluate the coefficients of the polynomial [26]. They evaluate the polynomial using a tensor contraction in a Bezier-Bernstein tetrahedral basis. This is achieved by sending a symmetric tensor of rank $d-1$ with $\binom{d+2}{d-1}$ unique elements from the vertex shader for each of the 4 vertices of the tetrahedra. The rasterization hardware on the GPU interpolates the tensor values after which, the fragment shader computes the coefficients efficiently using dot products. While this method is very clever, it will be computationally expensive for higher-order polynomials as $O(d^3)$ elements need to be sent for an algebraic surface of order d. This will also not extend easily for other non-algebraic surfaces. Since their goal was to render piecewise low order algebraic objects, streaming the coefficients down the pipeline was essential and delivered decent speed.

A balanced computation load is critical to good performance on the GPUs, given their SIMD model and limits on the length of the shader programs. Methods that use the $S(x, y, z)$ values are likely to be faster than those that use $F_f(t)$ values. Root finding schemes that use Bezier-Bernstein bases, Sturm sequences, singular value decomposition of the companion matrix, etc., operate primarily in the space of the $F_f(t)$ polynomial and will be quite inefficient on the GPU for higher order surfaces.

5.3 Marching Points Algorithm

Short and simple computations achieve the best performance on GPUs. An exceedingly simple root-isolation scheme is to march a point along the ray till the function $F_f(t)$ crosses zero between two samples. The point needs to march between bounds in t given by the view frustum or the bounding volume of the object. The computation complexity is low as only $F_f(t) = S(p(t))$ needs to be evaluated at the sample points. This suits the SIMD model of the GPU and can exploit its high computing power. We call this the *marching points* (MP) algorithm (Algorithm 5). This algorithm can be used for arbitrary implicit surfaces, even

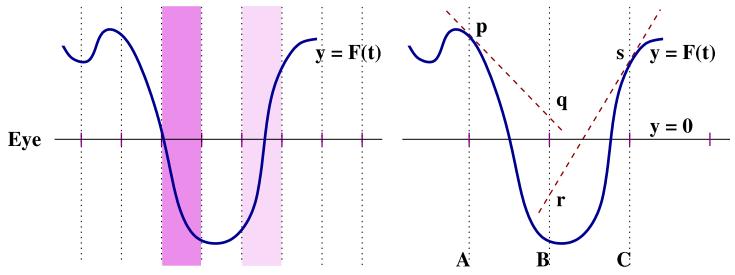


Figure 5.1: Marching points algorithm samples uniformly in the ray parameter t. The sign test identifies the first interval where the function changes sign at the endpoints (darker shaded region on the left). Sign test will fail as the step size increases (right). Roots will be isolated in intervals [A, B] and [B, C]. If the step size doubles again, the roots in [A, C] will be missed by the sign test. Taylor test detects the root in [A, C] by including points q and r into the calculations.

those with difficult derivatives or for general piecewise algebraic surfaces without derivatives at boundaries. The performance of the algorithm depends on the marching or sampling step-size. The optimal step-size may differ from one surface to another.

Algorithm 5 Marching Points (f, N)

1: Find the bounds t_s and t_e of the ray for fragment f.
2: Divide $[t_s, t_e]$ into N equal intervals
3: **for** $i = 0$ to $N - 1$ **do**
4: **if** rootExistsIn (t_i, t_{i+1}) **then**
5: Return $[t_i, t_{i+1}]$ as containing a root
6: **end if**
7: **end for**

The root-containment test used in step 4 is the critical operation in the above procedure. The test can be implemented in different ways. Two promising root-containment tests are the following.

1. *Sign test:* rootExistsIn $(t_i, t_{i+1}) = (S(p(t_i)) * S(p(t_{i+1})) < 0)$. Root exists if the function changes sign between the end points of the step. This test is simple to implement as only the function values at the sample points are needed. It is also a strict test that

does not produce false roots. It may, however, miss roots if an even number of roots are in the step.

2. *Taylor test:* rootExistsIn $(t_i, t_{i+1}) = (0 \in \tilde{F}([t_i, t_{i+1}]))$, a test for containment of a zero within a step. Interval arithmetic gives bounds of functions for a range in its domain as seen earlier. Exact interval extension is impractical in general, but acceptable ones can be found if the interval is small enough. We use an interval extension employing the function values at the endpoints as well as the first order Taylor series approximation of the function at the middle of the interval computed from both endpoints (Figures 5.1 and 4.1). This works adequately for moderate lengths of intervals. The interval extension in the interval $[t_i, t_{i+1}]$ is defined as

$$\tilde{F}([t_i, t_{i+1}]) = [\min\{p, q, r, s\},\ \max\{p, q, r, s\}] \quad \text{where}$$
$$p = F(t_i),\ q = F(t_i) + F'(t_i)\frac{(t_{i+1} - t_i)}{2},\ r = F(t_{i+1}) - F'(t_{i+1})\frac{(t_{i+1} - t_i)}{2},\ s = F(t_{i+1}). \tag{5.1}$$

This test is slower than the sign test because of the derivatives but larger step-sizes can be used. In practice, the running time doesn't change much on the average though the higher order surfaces suffer more due to the derivative calculations. This test can produce false roots, but works robustly in practice and can handle multiple roots well.

The worst case running time of marching points is linear in the size of the total range in t. However, it is fast in practice and can render a large number and type of surfaces. Table 5.1 gives the running time performance for algebraic surfaces up to order 18 and for several non-algebraic surfaces using both tests. The step-size is chosen so as to not miss any root. The interval $[t_s, t_e]$ is divided equally into the number of steps shown the table, ranging from 25 to 400 steps. Algebraic and non-algebraic implicit surfaces as complex as these have never been ray-traced at interactive rates before to the best of our knowledge.

5.4 Adaptive Marching Points Algorithm

The marching points algorithm takes fixed size steps in empty space as well as near the surface. The step-size has to be small enough to handle the worst-case, which occurs near

the silhouette of the surface. Larger step-sizes suffice in empty space if small step sizes can be used close to the surface. The *adaptive marching points* (AMP) algorithm uses a step size that depends on the closeness of the point to the surface and to its silhouette. We need a *proximity measure* to determine how close the current point on the ray is to the surface and a *horizon measure* to determine how close it is to a silhouette of the surface. The step-size should be small when near the surface and smaller near silhouettes.

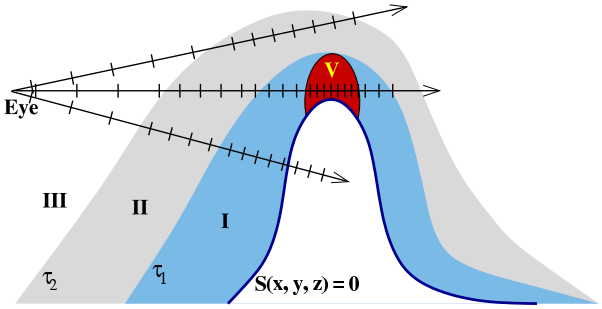


Figure 5.2: Adapting the step size to the distance to the surface. Region IV will have the largest step size and the region I will have the smallest, based on the proximity measure $|S(x, y, z)|$. The step size is further reduced for when the horizon condition is true (the darkened region V) as the surface normal is nearly perpendicular to the viewing direction.

Geometric distances are reliable measures of proximity to a surface. They are, however, surface dependent and not available for arbitrary implicit surfaces. Lipschitz bounds can be used to estimate the optimum step size for efficient ray-tracing [21, 17]. The Lipschitz constant can be defined as the maximum absolute value of a the derivative of the function in an interval. Unfortunately, it is hard to compute for the higher order algebraic and general implicit surfaces. Taubin suggests the use of the ratio $\frac{F(t)}{|F'(t)|}$ as a measure for signed geometric distance to the function $F(t)$ [47]. However, the measure is useful only for low-order algebraic surfaces and for points close to the surface. Most areas of the surface are missed on most higher-order surfaces with this distance function, in practice. Extending the definitions for geometric distance and Lipschitz bounds to arbitrary algebraic and non-algebraic surfaces will be a fruitful research direction for the future.

The magnitude of $S(x, y, z)$ gives the algebraic distance from a point to the surface. We use it as the proximity measure and vary the step-size as a monotonic function of it. In

practice, we use a piecewise constant approximation of this function and vary the step size piecewise constant approximation of this function and vary the step size in octaves, starting with a base step size of δ. The base step size is multiplied by 2 if the current point is far away from the surface and halved if close to it, using two thresholds τ_1 and τ_2. Thus, different step sizes can be used in regions of different colour/shade shown in Figure 5.2. The same base step as the marching points algorithm is used and the thresholds are set based on the coefficients of $S()$.

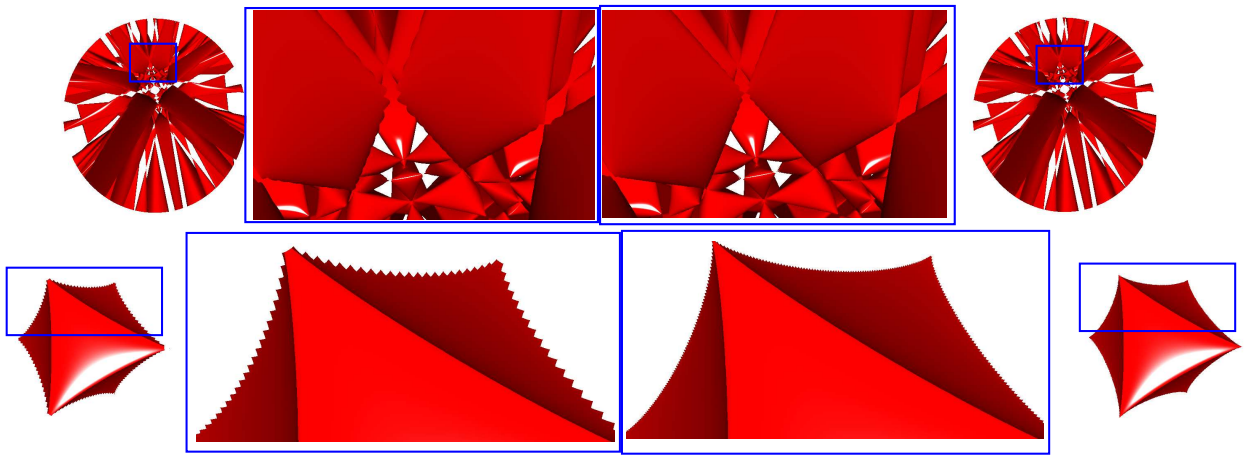


Figure 5.3: Top row: Barth tenth order surface without silhouette adaptation (left) and with it (right). The zoomed views in the middle show great reduction in the aliasing for the internal silhouettes. Bottom row: Superquadric surface without (left) and with (right) silhouette adaptation with zoomed views in the middle.

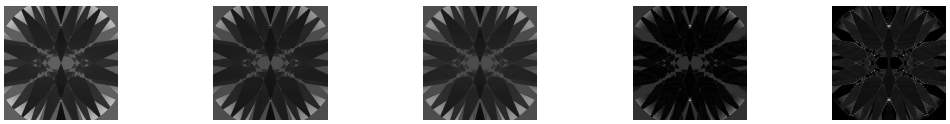


Figure 5.4: Number of steps taken along each ray for a Barth tenth order surface darker colour indicates less number of steps. Left to Right: Marching Points(mp), Adaptive Marching Points(amp), Adaptive Marching Points with silhouette adaptation Scaled difference image between mp and amp with silhouette adaptation, and Scaled difference image between amp and amp with silhouette adaptation.

It is also important to adapt the step-size to the view-dependent silhouettes. We decrease

the step-size near the silhouettes of the surface, using $|F'_f(t)|$ as a horizon measure. Note that the complex implicit surfaces may have many internal silhouettes that need to be handled carefully. We use $|F'_f(t)| \leq \epsilon$ as a *horizon condition*, which works fine for surfaces with continuous gradients. This is a reliable horizon measure if $p(t)$ is close to the surface, being the cosine of the angle between the ray and the local surface gradient. The step size is reduced near the silhouettes based on $|F'_f(t)|$. In practice, we halve the step-size further when the horizon condition is met (Algorithm 6). Thus, the darker, oval region of Figure 5.2 will have further reduced step sizes in order to render silhouettes well. The use of multiple thresholds ϵ_1, ϵ_2, and additional piecewise constant levels can provide greater adaptation to difficult silhouettes, but the single threshold suffices in practice for the kind of surfaces we dealt with. Olievera et al. also used the angle between the viewing direction and the surface normal to control the step size while ray-tracing height-fields on the GPU [35, 34].

Marching points and adaptive marching points can, however, miss multiple roots or produce false roots based on the specific test used, as explained earlier. A comparison of different tests for multiple roots is shown in Figure 5.5. The sign-change test can miss the root when the interval contains multiple roots. We can offset the surface by a small value to render $S(x, y, z) = \epsilon$ to alleviate proble (Figure 5.5). Strictly speaking, we are rendering a different surface, but the results can be close enough. Offsetting is similar to the $S(x, y, z) \leq \epsilon$ test for roots used by sphere tracing [17]. The Taylor test usign the first order interval extension produces robust results similar to the interval-based method given in Section 4 (Figure 5.5), confirming that it is a decent interval extension method in small intervals. The adaptive marching points method (Algorithm 6) achieves better rendering speeds without losing the versatility or quality of the basic marching points method (Table 5.2). Figure 5.3 shows the effect of silhouette adaptation. The aliasing at the silhouettes reduces sharply with silhouette adaptation. The superquadrics have the most challenging silhouettes as the surface is not C^1 continuous. The aliasing effects can be seen occasionally on these surfaces on the video. Figure 5.4 shows the number of iterations used for each pixel as a measure of the work done for the Barth decic surface. Adaptive marching points does less work than marching points almost everywhere. The extra effort near the silhouettes can be observed when silhouette adaptation is used. Figure 5.6 shows a quaddecic and a dodecic rendered at interactive

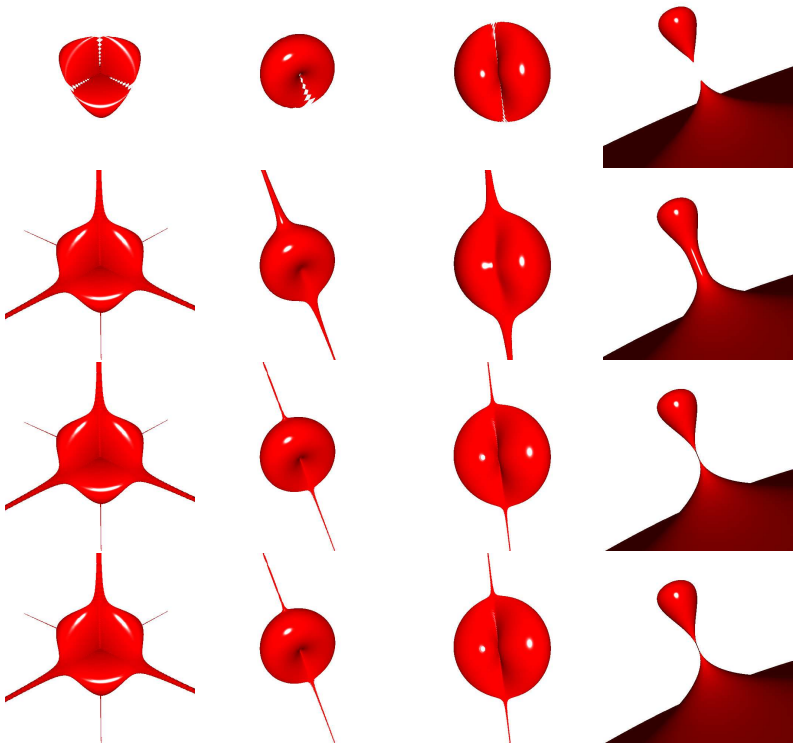


Figure 5.5: Top row: Steiner, Cross Cap, Miter and Kiss surfaces ray-traced using the adaptive marching points method with the sign test. Multiple roots are missed by it. Second row: Surfaces shifted by 0.01 using AMP and sign test. Region of multiple roots tend to be fattened. Third row: Same surfaces rendered using the AMP algorithm and the Taylor test for root containment. The performance is more robust for multiple roots. Bottom row: Same surfaces rendered using Mitchell's interval-based method (Section 4) which also produces robust roots.

frame rates.

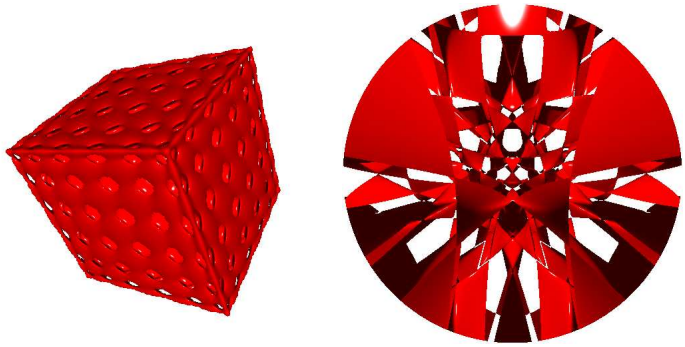


Figure 5.6: Left: Chmutovquaddecic at 125 fps Right:Sartidodecic at 86 fps

Algorithm 6 Adaptive Marching Points (f, b)

1: Find the intersections t_s and t_e of the ray for fragment f with the near and far planes.

2: Initialize s to the basic step size b; t to starting point t_s

3: **while** $t < t_e$ **do**

4: $$s = \begin{cases} b/4 & \text{if} \quad |S(p(t))| \leq \tau_1 \text{ and } |\vec{\nabla} S(p(t)) \cdot D_f| \leq \epsilon \\ b/2 & \text{if} \quad |S(p(t))| \leq \tau_1 \\ 2b & \text{if} \quad |S(p(t))| > \tau_2 \\ b & \text{otherwise} \end{cases}$$

5: **if** rootExistsIn $(t, t+s)$ **then**

6: Return $[t, t+s]$ as containing a root

7: **end if**

8: $t = t + s$

9: **end while**

Surface [order]	**Max steps**	**Frames per second**		**Surface [order]**	**Max steps**	**Frames per second**	
		Sign	Taylor			Sign	Taylor
Algebraic Surfaces							
Chmutov [18]	400	85	38	Kleine [6]	400	285	290
Chmutov [14]	400	55	48	Dervish [5]	300	285	275
Sarti [12]	300	60	53	Peninsula [5]	85	370	447
Barth [10]	300	92	105	Piriform [4]	55	520	305
Chmutov [9]	200	125	135	Cushion [4]	75	390	305
Endrass [8]	300	140	179	Torus [4]	50	410	430
Chmutov [8]	250	185	195	Cassini [4]	55	405	375
Chmutov [7]	175	138	206	Cross-Cap [4]	50	400	465
Labs [7]	200	115	120	Goursat [4]	50	580	515
Barth [6]	125	300	310	Cayley [3]	60	460	475
Heart [6]	120	265	260	Clebsch [3]	55	470	500
Hunt [6]	400	230	225	Ding-Dong [3]	25	825	560
Non-Algebraic Surfaces							
Superquadric	150	105	125	Scherk's	250	200	315
Blobby	250	160	305	Diamond	250	260	306
Blinn's Blobby	75	380	460				

Table 5.1: Maximum number of steps and the frame rate using the marching points method for a 512×512 window.

Surface [**order**]	**Max steps**	**Frames per second**		**Surface** [**order**]	**Max steps**	**Frames per second**	
		Sign	Taylor			Sign	Taylor
Algebraic Surfaces							
Chmutov [18]	100	98	60	Kleine [6]	48	435	385
Chmutov [14]	100	125	95	Dervish [5]	45	285	280
Sarti [12]	100	86	75	Peninsula[5]	35	512	435
Barth [10]	100	150	115	Piriform [4]	32	552	315
Chmutov [9]	81	185	165	Cushion [4]	32	420	335
Endrass [8]	96	190	208	Cassini [4]	32	525	506
Chmutov [8]	64	215	216	Cross-Cap[4]	32	530	540
Chmutov [7]	63	242	233	Torus [4]	24	555	525
Labs [7]	77	232	310	Goursat [4]	24	635	605
Hunt [6]	84	240	325	Cayley [3]	33	600	652
Barth [6]	60	325	310	Clebsch [3]	21	585	555
Heart [6]	48	420	325	Ding-Dong[3]	15	920	665
Non-Algebraic Surfaces							
Superquadric	100	185	155	Scherk's	100	358	322
Blobby	50	329	300	Diamond	100	360	330
Blinn's Blobby	40	456	545				

Table 5.2: Maximum number of steps and the frame rate using the adaptive marching points method for a 512×512 window.

Chapter 6

Experimental Results

In this section, we present the complete rendering results on several algebraic and non-algebraic surfaces. We could render algebraic surfaces up to order 18 including all surfaces shown in the MathWorld site and several non-algebraic and transcendental objects. Screen-shots of these surfaces appear in Figures 8.1 and 8.2. The equations of the corresponding surfaces is given in the Appendix A. First, we display the overall ray-tracing algorithm and some of its implementation issues.

6.1 Overall Algorithm: Implementation Issues

The overall ray-tracing program for implicit surfaces is given in Algorithm 7. The implementation is in OpenGL/GLSL for the SM4.0 architecture of the Nvidia 8800 GTX GPU. Here we give a few points to be kept in mind for efficiency and applicability.

1. Analytical root-finding (Section 3.3) can be used for surfaces of order less than 5. Adaptive marching points (Section 5.4) performs the best otherwise.

2. The shaders are compiled on the fly under current programming environments including GLSL and Cg. The function to evaluate the expression $S(p(t))$ and its gradient (if necessary) can be synthesized on the fly by the CPU for the specific surface.

3. Computing $S(p(t))$ and its gradient $\vec{\nabla}S(p(t))$ together in one function is more efficient than evaluating them separately as many calculations can be shared. The gradient is

Algorithm 7 ImplicitSurface Render (f)

CPU:

1: Setup equations in the shader program.

2: Send a dummy quad to the OpenGL pipeline.

Vertex Shader:

1: Pass through vertices without any modification and camera center to the geometry shader.

Geometry Shader:

1: Transform the dummy quad to screen-facing quad and pass parameters like the ray direction to the vertices, the camera center, near, and far plane distances to the pixel shader.

Fragment Shader:

1: Intersect each ray with the near and far planes to get the range $[t_s, t_e]$.

2: Isolate the root using one of the described algorithms.

3: Refine the root using 10 steps of Newton's Bisection method.

4: Shoot a ray from the intersection point towards the light source(s). Perform root-isolation for it. If root is found, the point is under shadow.

5: Compute per-pixel colour and depth using the position, normal, and shadowing at the intersection point.

needed only in some cases, like the interval-based methods and for silhouette adaptation.

4. Products of vectors are used to compute $x^2, y^2, z^2, x^3, y^3, z^3$, etc., simultaneously within the shaders. Dot products are used wherever possible.

5. All timing given in this thesis are for ray-tracing all 512×512 rays of the screen. Simple optimizations involving bounding boxes of the objects, octree partitioning, etc., can increase the timing performance greatly as reported earlier [21, 17]. Most of the complex surfaces are not bounded easily, however.

6. The vertex and geometry shaders jointly perform initializations of the common parameters and drawing a screen-sized quad. The root finding is performed on the fragment shader using single precision floating point arithmetic.

6.2 Self Shadowing

Though the single-bounce ray-tracing was described so far, secondary rays can be spawned from points of intersection for shadowing, reflections, transparency, etc. General recursive ray-tracing will require extensive book-keeping to perform on the shader and will be difficult on the GPUs. Shooting secondary rays to the light sources to compute per-pixel shadowing is practical, however. A point is shadowed if the ray from it to the light source hits the surface before the light source. We only need to know if a point is shadowed; we needn't know the intersection of the secondary ray with the surface. Thus, only root isolation (step 2 of Algorithm 4) is needed. Figures 1.2 and 6.1 show the shadowing effects on a few surfaces, including from multiple light sources. The video contains more interactive shots. Table 6.1 shows the performance of each algorithm with and without shadows. The rendering rate suffers a little due to shadowing, while comfortably being above the real-time rates in all cases.

Figure 6.1: Chmutov octdecic, Chmutov quaddecic, Sarti dodecic, Kiss quintic, and a Blobby surface with self shadows and highlights.

6.3 Rendering Times

Table 6.1 presents the comprehensive frame rates for all methods discussed in the thesis on several algebraic and non-algebraic surfaces for both with shadow rays and without them

on an Nvidia 8800 GTX for a resolution of 512×512. Table 6.2 presents results of AMP method on Nvidia 280 GTX for a resolution of 512×512. The interval-based method has low applicability but shows the least drop in fps due to shadowing. Marching points and its adaptive versions are versatile and fast. They have difficulty with surfaces with a long locus of multiple roots as explained earlier. The rendering times are orders of magnitude better than anything reported in the literature. We also report real-time results on surfaces much more complicated than have been reported before.

6.4 Comparison with Affine Arithmetic Ray-tracing and Marching Tetrahedra

Complex implicit surfaces like the ones we used have not been ray-traced before on the GPUs to provide comparison. The best reported effort by Knoll et al. achieve a frame rate of 101 on a sextic surface, 16 on a decic surface and up to 108 on superquadric-like surfaces on a CPU with the SIMD extensions [23]. Our method is faster as seen in Table 6.3. Their method involves an affine arithmetic on GPU and is slower than our AMP method based on adaptive point sampling.

Implicit surfaces can, however, be polygonized using an algorithm like the Marching Cubes algorithm [28] as the iso-surface of $S(x, y, z)$. An efficient implementation of the marching tetrahedra method – a variation on the marching cubes – was made available recently for the GPUs by Nvidia [14]. It works in a voxelized space of resolution $m \times n \times p$ and finds the triangles on the iso-surface. We adapted the code to render the triangles with per-pixel lighting and tried it on different implicit surfaces. Table 6.4 lists the running times of marching tetrahedra and our method on comparable resolutions and Figure 6.2 provides the visual rendering results. The $64 \times 64 \times 64$ configuration of the marching tetrahedra produces nearly the same visually quality as the $128 \times 128 \times 64$ configuration of marching points when rendering to a 128×128 window. We had difficulty getting higher resolutions of marching tetrahedra to work on GPU. Our method is 20 to 50 times faster than marching tetrahedra. The marching tetrahedra algorithm is also not able to handle more complex surfaces than

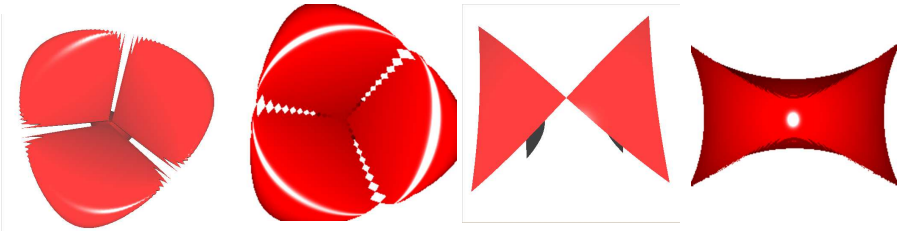


Figure 6.2: Left to right: Steiner quartic surface rendered with marching tetrahedra and with adaptive marching points. Hunt's sextic surface with marching tetrahedra and adaptive marching points. Marching tetrahedra was computed for a 64^3 voxel grid and AMP for a $256^2 \times 64$ grid and rendered at a 256×256 resolution.

those shown in the table. The coarse sampling of marching tetrahedra also produces more artifacts on difficult areas as can be seen in Figure 6.2.

6.5 Dynamic Implicit Objects

A dynamic implicit object changes its form over time. The rayskip algorithm ray-traces dynamic implicits by exploiting the temporal and spatial coherence of ray-implicit intersections [9]. Knoll et al. render dynamic implicits as 4D implicits in an (x, y, z, t) space [22]. The algorithms presented in this thesis ray-trace the implicit surface independently in each frame without any precomputations or subdivisions. Thus, the equation can change each frame without affecting the performance in anyway. Temporal coherence can be used but the additional book-keeping slows down the process in practice on the GPUs. Figure 6.3 shows a few views from two dynamic objects. The first is a Blinn's Blobby object with 30 spheres. As spheres fuse together when proximate, the topology of the object changes from each being independent spheres to a single fused object. The positions of the spheres change with time to create the segment in the video. The AMP method renders this object at a framerate of 34 fps or more. The second is a superquadric with a twist term that changes with frame. The video shows the twist being changed continuously, resulting in a continuous deformation of the object. The AMP method renders the object at over 100 fps.

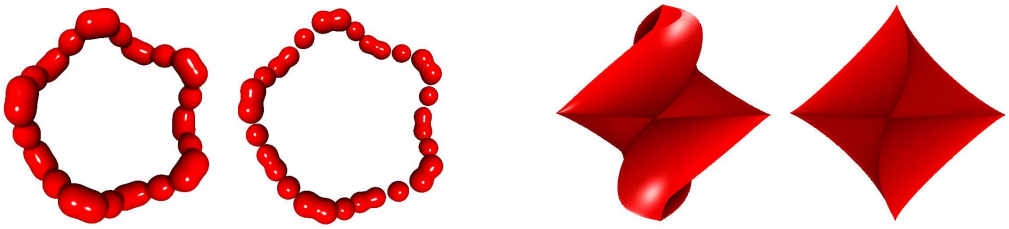


Figure 6.3: Dynamic objects: Two views of an evolving object with 30 Blinn's blobbies rendered at over 34 fps (left) and of twisting superquadric rendered at over 100 fps (right).

6.6 Limitations

We ray-traced algebraic and non-algebraic surfaces of high complexity using different methods in this thesis. However, each has its limitations. The interval-based method is robust but difficult on more complex surfaces due to the lack of good interval extensions. The marching points and its variations are versatile and fast but the performance depends on the sampling rate, which was set by hand for each surface for most of the results in this thesis. A conservative sampling step-size can produce correct results with some drop in the frame rate. The last four columns of Table 6.1 present results for the adaptive marching points algorithm when the maximum number of steps for the range $[t_s, t_e]$ is set to be $\max(100, 15 + 2k^2)$, where k is the order of the algebraic surface.

Multiple roots create difficulty for all methods. The interval-inspired Taylor test reduced the problem for some of the surfaces, though the false-root problem makes it slower than sign-test for some surfaces (e.g., Cushion, Piriform). All methods fail on surfaces with extreme self-intersections such as the minimal surfaces [50]. Among the non-algebraic minimal surfaces, our method was able to render only Scherk's surface correctly (Table 6.1, Figure 8.2).

The GPUs primarily support only single precision arithmetic. This has not been a problem in the wide class of algebraic and non-algebraic surfaces we tried. The computation of high order polynomials, however, needs to be done carefully as numerical instabilities can produce wrong results. The Chmutov surfaces of higher orders have serious artifacts due to false roots near ± 1 when the Chebyshev polynomial equations were evaluated directly, using the powers of x, y, and z. This was not due to lower precision as the CPU implementation using double

precision had the same artifacts. However, the problems disappear when the recursive or the sinusoidal definitions of the Chebyshev polynomial is used. The rendering speed suffers as the iterative evaluation is computationally expensive for higher orders. The computational load is nearly a constant independent of the order using the sinusoidal formulation as seen in Table 6.2. The direct evaluation of the polynomial is the quickest for orders upto 18. See the Appendix A for the iterative and sinusoidal definitions of the Chmutov surfaces.

Since ray-tracing relies on sampling the space in terms of the rays and we further sample the surface along the ray, aliasing is an important issue. Adapting the sampling rate based on the proximity to the surface and the view-dependent curvature of the surface provides good results as shown earlier. The problem is very hard for surfaces without continuous gradients like superquadrics with fractional powers (Figure 5.3). The straightforward approach we employed provide adequate quality in practice but some view-dependent aliasing effects are visible at places in the video. Extending these techniques to general GPU ray-tracing is a challenging topic of research by itself.

Surface [order]	Interval Based Algorithm		Adaptive Marching Points Algorithm							
			#Steps: Hand-Tuned				#Steps: Fixed Formula			
			Sign test		Taylor test		Sign test		Taylor test	
	A	**B**	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**
Algebraic Surfaces										
Chmutov [18]	-	-	98	70	60	45	98	70	45	38
Chmutov [14]	-	-	125	95	95	75	125	95	66	59
Sarti [12]	-	-	86	78	75	49	86	78	69	60
Barth [10]	-	-	150	110	115	79	150	110	82	75
Chmutov [9]	-	-	185	150	165	120	168	129	106	90
Endrass [8]	-	-	190	140	208	140	185	135	133	105
Chmutov [8]	-	-	215	175	216	161	165	160	145	110
Chmutov [7]	-	-	242	180	233	175	200	164	158	125
Labs [7]	-	-	232	165	310	155	220	150	125	105
Chmutov [6]	-	-	418	280	325	235	265	225	165	133
Hunt [6]	-	-	240	182	310	155	240	172	139	102
Barth [6]	-	-	325	270	325	230	316	225	175	139
Heart [6]	-	-	420	280	300	225	275	235	185	133
Kleine [6]	-	-	435	267	385	245	285	240	195	165
Dervish [5]	60	54	285	250	280	175	215	195	178	135
Kiss [5]	77	71	428	325	435	265	405	295	270	230
Peninsula [5]	85	78	520	326	535	295	380	285	306	235
Steiner [4]	235	215	645	420	516	315	405	365	270	240
Cassini [4]	215	208	510	320	506	285	396	305	285	240
Tooth [4]	195	190	617	425	542	287	440	375	345	260
Piriform [4]	230	222	552	450	315	246	345	275	291	240
Cross-Cap[4]	195	188	530	305	540	285	325	265	251	215
Miter [4]	186	176	535	325	528	285	345	285	245	225
Kummer [4]	225	219	555	405	302	245	318	245	195	155
Goursat [4]	210	198	635	420	605	325	420	365	315	265
Cushion [4]	170	163	420	320	335	225	329	250	186	145
Nordstrands [4]	220	211	458	305	324	235	336	285	245	170
Cayley [3]	580	572	600	365	652	315	452	345	391	200
Clebsch [3]	590	588	585	355	555	235	425	335	340	205
Ding-Dong [3]	965	960	918	482	665	475	605	485	450	325
Non-Algebraic Surfaces										
Torus	-	-	540	350	515	295	540	350	515	295
Superquadric	-	-	185	145	155	105	185	145	155	105
Blobby	-	-	329	265	300	195	329	265	300	195
Blinn's Blobby	-	-	456	344	545	325	456	344	545	325
Scherk	-	-	358	222	322	185	358	222	322	185
Diamond	-	-	360	208	330	199	360	208	330	199

Table 6.1: Rendering time results for several algebraic and non-algebraic surfaces for different algorithms. Frame rates without shadows is given in **A** columns and with shadows is shown in **B** columns for a 512×512 window on an Nvidia 8800 GTX. The order of each algebraic surface appears within square brackets. The timings on the left half for AMP are for step-sizes adjusted manually. The right half uses a conservative formula (see Section 6.6) for the maximum number of iterations.

Surface [order]	Stepsize	Sign Test		Taylor Test	
		NS	S	NS	S
Algebraic Surfaces					
Chmutov [50]	0.01	65	41	41	30
Chmutov [22]	0.01	160	71	113	49
Chmutov [18]	0.01	344	170	194	144
Chmutov [14]	0.01	457	287	262	224
Chmutov [12]	0.01	462	295	280	232
Sarti [12]	0.015	380	130	200	104
Barth [10]	0.015	573	252	286	176
Chmutov [9]	0.015	521	234	334	187
Endrass [8]	0.01	244	167	215	102
Nonisol [8]	0.01	1035	578	548	386
Chmutov [8]	0.02	1053	575	622	413
Chmutov [7]	0.02	787	256	545	228
Labs [7]	0.015	595	235	311	157
Chmutov [6]	0.02	1289	712	717	539
Hunt [6]	0.01	152	134	78	62
Barth [6]	0.02	836	377	557	247
Heart [6]	0.02	840	703	477	436
Kleine [6]	0.02	1024	356	633	287
High Silhouette[6]	0.02	1143	814	655	523
Dervish [5]	0.02	814	288	466	229
Kiss [5]	0.02	1297	644	774	573
Peninsula [5]	0.03	1349	590	832	489
Steiner [4]	0.02	1210	953	678	540
Cassini [4]	0.02	1147	517	699	371
Tooth [4]	0.03	1500	684	871	521
Piriform [4]	0.02	1450	1349	831	781
Cross-Cap[4]	0.02	1200	948	606	530
Miter [4]	0.02	1660	1140	891	645
Kummer [4]	0.02	1400	436	823	371
Goursat [4]	0.04	1700	930	1014	700
Cushion [4]	0.02	880	587	460	358
Nordstrands [4]	0.03	945	281	596	249
Cayley [3]	0.03	1519	481	844	405
Clebsch [3]	0.03	940	264	633	212
Ding-Dong [3]	0.03	1924	943	1188	739
Non-Algebraic Surfaces					
Chmutov [50]	0.01	207	124	145	87
Chmutov [22]	0.01	217	130	152	91
Chmutov [18]	0.01	244	147	171	102
Chmutov [14]	0.01	254	152	178	106
Torus	0.04	1650	915	922	628
Superquadric	0.02	900	751	394	366
Blobby	0.05	1226	509	780	418
Blinn's Blobby	0.05	1304	1022	691	620
Scherk	0.03	1112	449	718	390
Diamond	0.04	1300	309	983	272

Table 6.2: Frame rates for several algebraic and non-algebraic surfaces using our algorithm

Surface	**FPS using [23]**	**FPS using our method**
Steiner	38	212
Teardrop	121	178
Tangle	71	196
Barth Sextic	88	120
Kleine	101	170
Mitchell	60	176
Barth Decic	16	94
Superquadric	108	544

Table 6.3: Comparison of frame rates for different surfaces using Knoll's affine arithmetic method on a GPU and our AMP method on the GPU on common surfaces.

Surface Name	**Marching Tetrahedra** 64^3	**Marching Points**		**Adaptive March. Pts**	
		$128^2 \times 64$	$256^2 \times 64$	$128^2 \times 64$	$256^2 \times 64$
DingDong [3]	14	0.468	0.810	0.445	0.751
Steiner [4]	14	0.586	0.924	0.513	0.797
Peninsula[5]	16	0.537	0.890	0.539	0.738
Chmutov [7]	16	0.634	1.154	0.610	1.109
Chmutov [8]	32	0.689	1.323	0.653	1.353
Chmutov [9]	46	0.757	1.343	0.730	1.398

Table 6.4: Rendering times in milliseconds for the GPU marching tetrahedra for 64^3 resolution and marching points and adaptive marching points for $128^2 \times 64$ and $256^2 \times 64$ resolutions on an Nvidia 8800 GTX.

Chapter 7

Ray-Tracing on the GPU: Discussion

The performance of a GPU ray-tracing algorithm depends on three of its aspects: the algorithmic complexity, the per-pixel computational load, and the match with the SIMD architecture of the GPUs.

1. *Algorithmic Complexity:* The convergence rate determines the computational complexity for iterative root-finders. Among the methods we presented, the marching points methods have a linear behaviour due to its front-to-back sampling. The interval-based method has a logarithmic behaviour due to the repeated subdivision of the interval. Table 7.1 presents the ray-tracing times of a few surfaces for different depths or distances from the camera. The timings of the CPU version and the GPU version of the AMP method and the interval-based method are given. Clearly, the interval-based method is not affected by the distance to the camera while the AMP algorithm needs more time for farther surfaces. The behaviour for CPU and GPU versions are essentially same, suggesting that the interval-based method has lower algorithmic complexity as expected.

2. *Computational Load:* The running time depends on the the number of shader computations needed for each ray. The marching points methods need to evaluate only the function at each sample point and have a lower computation load than the interval-based method. The interval-based method uses exact interval-extension that needs to find all critical points of the function and its derivative as explained earlier for each

Method	Rendering time in milliseconds for different distances to surface			
	z = 0	**z = 1**	**z = 4**	**z = 5**
Sphere (Quadratic)				
Adaptive Marching Points (GPU)	0.770	0.931	1.144	1.195
Adaptive Marching Points (CPU)	760.4	791.7	1082.2	1195.2
GPU Speedup	*987.5*	*850.4*	*946.0*	*1000.2*
Interval Based (GPU)	0.519	0.519	0.520	0.520
Interval Based (CPU)	463.8	463.9	464.0	464.2
GPU Speedup	*893.6*	*893.8*	*892.3*	*892.7*
Ding Dong (Cubic)				
Adaptive Marching Points (GPU)	1.092	1.169	1.844	1.979
Adaptive Marching Points (CPU)	986.9	994.2	1289.0	1410.3
GPU Speedup	*903.8*	*850.5*	*699.0*	*712.6*
Interval Based (GPU)	1.035	1.038	1.039	1.040
Interval Based (CPU)	878.5	878.9	879.4	879.9
GPU Speedup	*848.8*	*846.7*	*846.4*	*846.1*
Nordstrands (Quartic)				
Adaptive Marching Points (GPU)	2.183	2.448	3.072	3.345
Marching Points on (CPU)	3159.5	3361.5	4092.1	4385.2
GPU Speedup	*1447.3*	*1373.2*	*1332.1*	*1311.0*
Interval Based (GPU)	4.544	4.545	4.546	4.548
Interval Based (CPU)	6210.2	6211.3	6211.9	6212.4
GPU Speedup	*1366.7*	*1366.6*	*1366.5*	*1366.0*
Dervish (Quintic)				
Adaptive Marching Points (GPU)	3.355	3.593	3.995	4.115
Adaptive Marching Points on (CPU)	4152.3	4495.7	5365.0	5874.6
GPU Speedup	*1237.6*	*1251.2*	*1342.9*	*1427.6*
Interval Based (GPU)	16.528	16.529	16.530	16.530
Interval Based (CPU)	8320.4	8321.6	8322.4	8322.7
GPU Speedup	*503.4*	*503.5*	*503.5*	*503.5*

Table 7.1: Ray-tracing times for different surfaces on CPU and GPU for the AMP method and the interval-based methond and for different distances to the camera. The speedup of the GPU for each of the method is also shown for each surface.

sample that it evaluates. Table 7.1 compares the ray-tracing times of the two methods for a few surfaces. The interval-based method is faster than AMP for lower order surfaces. The advantage is nullified for higher order surfaces due to the greater computational load incurred on them, even for larger distances to the surface. The behaviour is the same on the CPU and the GPU as it is based on the inherent computational load.

3. *SIMD Architecture of the GPU:* The constraints of the SIMD architecture can also affect the running time beyond the above factors. Conditional branching is inefficient under SIMD if different pixels have to take different branches. The interval-based method has conditional branching during the computation of the interval extensions. It also branches based on the change in sign of the function and its derivative. The AMP algorithm has no conditional branching and suits the SIMD architecture of the GPU well. The effect of branching may be less pronounced if all rays of the group that are scheduled together take the same branches. Table 7.1 also gives the speedup of the GPU version over the CPU version for each algorithm. The GPU is able to achive higher speedups for the AMP algorithm than the interval-based one due to the better match with the SIMD architecture. The difference is more pronounced at higher order surfaces as there is more scope for divergence on them.

Overall, the modern GPU is well suited for single-bounce ray-tracing using the ray sampling methods presented in this thesis. The algebraic surfaces shown in this thesis are rich enough for most tasks. The single precision of the GPUs is quite adequate for most surfaces, though higher numerical precision could help with the ray-tracing of more complex ones. The impact of the match with the SIMD architecture seems to grow with the order or complexity of the surface. The simple and seemingly non-promising methods can achieve very high performance on the GPUs as a result. This is in conformance with the experience with algorithms like sorting on the GPU, where the simpler bitonic sort came out faster on the GPU than more optimal sequential algorithms.

Recursive ray-tracing is essential to produce complex inter-reflections and other effects. The current generation of GPUs provide no support for it. Some support for stacks within

the GPU memory can enable limited recursive ray-tracing and other similar applications on the GPU.

Chapter 8

Conclusions & Future Work

We presented a few schemes to ray-trace complex algebraic and non-algebraic implicit surfaces on the GPU at very high frame rates. The analytic and interval-based methods give robust and fast solutions for algebraic surfaces, but are limited in its applicability. The adaptive marching points method provides the best performance for arbitrary algebraic or non-algebraic surfaces. Sign-test based root isolation will suffice for most surfaces in practice but the first-order interval based root isolation can be used if multiple roots are likely. The simplicity of the method is the key factor behind the high performance on current GPUs. The single-precision of current GPUs prevents easy adaption of this method to algebraic surfaces of order greater than twenty. The very high-order algebraic surfaces may not occur in practice today in graphics and visualization. However, the ability to ray-trace such surfaces provides scientists and other practitioners the freedom to choose whatever model they want for their data and use a uniform method for rendering. The performance of the lower order surfaces is significantly better than the higher order ones.

Whitted and Kajiya propose the use of fully procedural graphics to exploit the high compute power of the GPUs using the low external bandwidth they possess [51]. We believe this will be the direction of the high-performance graphics of tomorrow. General implicit surfaces are expressive and can be ray-traced fast on the GPUs using our scheme. Ray-tracing can produce exact images independent of the resolution and can exploit the high compute power of the GPUs effectively. Implicit or procedural description of geometry provides high quality without increasing the representational complexity. Simplicity of the

underlying algorithm is critical to extracting high performance from the SIMD architecture of the current GPUs.

8.1 Directions for Future Research

The following lines of research can help push the ray-tracing of implicit surfaces on GPUs and other multi-core platforms based on the experience gained by this work.

1. Arbitrary implicit surfaces lack good geometric distance measures. This prevents the adoption of efficient techniques like sphere tracing [17] to them. Lipschitz continuity theory can provide insights into this problem thought the measures suggested in the literature do not easily apply to arbitrary surfaces. More study is needed to arrive at reasonable bounds based on a normalized form of the implicit equation.

2. The effects of limited numerical precision on higher order surfaces need to be explored further. Are there normalizing methods that can exploit the precision to the maximum? Double precision arithmetic is likely to be available on future GPUs, but perhaps at a premium on computation time. Mixing single and double precision operations in the right proportion for a single ray and across adjacent rays will be critical to get high performance.

3. The inherent coherence between adjacent rays can also be exploited for faster and more accurate results. The best direction may be to bring in the ideas from beam-tracing into this problem. GPUs schedule execution of several pixels together based on image-space adjacency and the availability of fragment hardware. Thus, beam tracing that combines image-space and object-space coherence can give maximum performance on the GPUs.

4. Anti-aliasing may be an important issue for such complex surfaces which could have internal silhouettes in addition to external ones (Figure 5.3). The adaptive marching points algorithm already differentiates points near the silhouettes from other points. This can form the basis for selective oversampling, i.e., increasing the number of rays

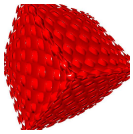
traced near the silhouettes. These rays can be produced on the fly by the shader programs or marked for rendering in a separate pass. A good geometric distance measure and a good horizon condition are essential for high performance.

5. Beam-tracing and sharing results of the computation across rays require more general-purpose communication between adjacent rays of the rendered image. A programming model that treats the GPU as a general parallel programming device is likely to provide better performance than the traditional graphics-pipeline approach. The CUDA programming model provided on the Nvidia GPUs and others similar to that may then achieve higher performance for ray-tracing on the same hardware.

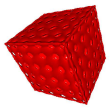
6. Texture mapping general implicit surfaces is an interesting problem on which very little work has gone on. The surfaces used in this thesis can have multiple parts and arbitrary genus. Parameterizing them consistently as a single piece is challenging. They could be paramterized locally using Monge patches which are either parabolic, ellipsoidal or hyperbolic in shape depending on the curvature of the point. Another approach could be to locally parametrize parts of the surface having genus zero.

7. Ray-tracing parametric surfaces is largely an open problem today. The 18th degree polynomial that results from applying Kajiya's technique [20] to bicubic surfaces is dense and cannot be solved easily using an extension of our method as all roots are needed. A solution using an extension of bracketing to two-dimensional intervals in the (u, v) parameter space and bisection within the parameter rectangle needs to be explored.

Overall, handling procedural and implicit geometry directly on fast GPUs will be more common in the future. The GPUs themselves may need to provide additional features in hardware to make this easy. This can include higher precision arithmetic, programmable rasterizers, etc. Procedural elements can also be applied to other aspects such as textures, normals, shading, etc. We expect the GPUs will evolve to support this natively and efficiently.



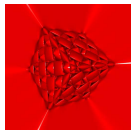
Chmutov [18]
(98)



Chmutov [14]
(125)

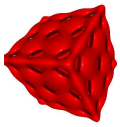
Sarti [12]
(86)

Barth [10]
(150)

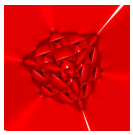


Chmutov [9]
(185)

Endreass [8]
(190)

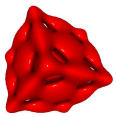


Chmutov [8]
(215)

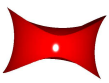


Chmutov [7]
(242)

Labs [7]
(232)

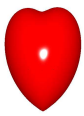


Chmutov [6]
(418)



Hunt [6]
(240)

Barth [6]
(325)

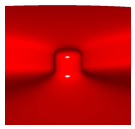


Heart [6]
(420)

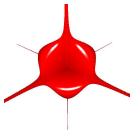
Kleine [6]
(435)

Dervish [5]
(285)

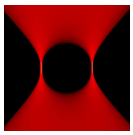
Kiss [5]
(428)



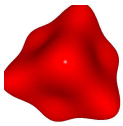
Peninsula [5]
(520)



Steiner [4]
(645)



Cassini [4]
(510)

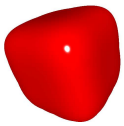


Tooth [4]
(617)

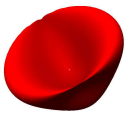
Cross Cap [4]
(530)

Miter [4]
(535)

Kummer [4]
(555)



Goursat [4]
(635)

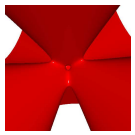


Cushion [4]
(420)

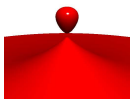
Nordstrands [4]
(458)

Piriform [4]
(552)

Cayley [3]
(600)

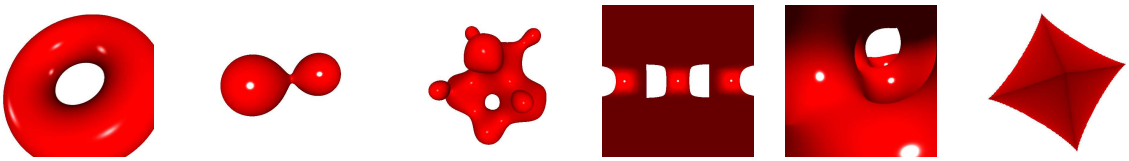


Clebsch [3]
(585)



Ding-Dong [3]
(918)

Figure 8.1: Pictures of various algebraic surfaces with the order of the surface shown within square brackets and the FPS using the adaptive marching points algorithm shown within parenthesis for a 512×512 window.



Torus	Blinn's Blobby	Blobby	Scherk	Diamond	Superquadric
(540)	(456)	(329)	(358)	(360)	(185)

Figure 8.2: Pictures of the non-algebraic surfaces rendered by us with the FPS using the adaptive point sampling algorithm given in parenthesis for a 512×512 window.

Appendix A

Implicit Equations

Cubic Surfaces

1. *Ding-Dong:* $x^2 + y^2 = z(1 - z^2)$.

2. *Clebsch:* $81(x^3 + y^3 + z^3) - 189(x^2(y + z) + y^2(x + z) + z^2(x + y)) + 54xyz + 126(xy + yx + xz) - 9(x^2 + y^2 + z^2) - 9(x + y + z) + 1 = 0$.

3. *Cayley:* $-5(x^2(y + z) + y^2(x + z) + z^2(x + y)) + 2(xy + yx + xz) = 0$.

Quartic Surfaces

1. *Torus:* $(x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(x^2 + y^2) = 0$.

2. *Nordstrands:* $25(x^3(y+z)+y^3(x+z)+z^3(x+y))+50(x^2y^2+y^2x^2+x^2z^2)-125(x^2yz+y^2xz+xyz^2)-4(xy+yx+xz)+60xyz=0$.

3. *Cushion:* $z^2x^2-z^4-2zx^2+2z^3+x^2-z^2-(x^2-z)^2-y^4-2x^2y^2-y^2z^2+2y^2z+y^2=0$.

4. *Goursat:* $x^4 + y^4 + z^4 - 1 = 0$.

5. *Kummer:* $x^4 + y^4 + z^4 - x^2 - y^2 - z^2 - x^2y^2 - y^2z^2 - z^2x^2 + 1 = 0$.

6. *Miter:* $4x^2(x^2 + y^2 + z^2) - y^2(1 - y^2 - z^2) = 0$.

7. *Cross Cap:* $4x^2(x^2 + y^2 + z^2 + z) + y^2(y^2 + z^2 - 1) = 0$.

8. *Piriform:* $x^4 - x^3 + y^2 + z^2 = 0$.

9. *Tooth:* $x^4 + y^4 + z^4 - x^2 - y^2 - z^2 = 0$.

10. *Cassini:* $((x+a)^2 + y^2)((x-a)^2 + y^2) = z^2$ where a is the radius of the circle.

11. *Steiner:* $x^2y^2 + x^2z^2 + y^2z^2 - 2xyz = 0$.

Quintic Surfaces

1. *Peninsula:* $x^2 + y^3 + z^5 - 1 = 0$.

2. *Kiss:* $x^2 + y^2 = z(1 - z^4)$.

3. *Dervish:* $64(x-1)(x^4 - 4x^3 - 10x^2y^2 - 4x^2 + 16x - 20xy^2 + 5y^4 + 16 - 20y^2) - 5a(2z - a)(4(x^2 + y^2 + z^2) + (1 + 3\sqrt{5}))^2 = 0$ where $a = \sqrt{5 - \sqrt{5}}$.

Sextic Surfaces

1. *Barth:* $4(\phi^2x^2 - y^2)(\phi^2y^2 - z^2)(\phi^2z^2 - x^2) - (1 + 2\phi)(x^2 + y^2 + z^2 - 1)^2 = 0$ where $\phi = (1 + \sqrt{5})/2$ is the golden ratio.

2. *Hunt:* $4(x^2 + y^2 + z^2 - 13)^3 + 27(3x^2 + y^2 - 4z^2 - 12)^2 = 0$

3. Kleine: $(x^2+y^2+z^2+2y-1)(x^2+y^2+z^2-2y-1)^2-8z^2]+16xz(x^2+y^2+z^2-2y-1) = 0$ represents a 3D impression of the Kleine bottle.

4. *Chmutov:* $T_6(x) + T_6(y) + T_6(z) = 0$ where $T_6(x) = 2x^2(3 - 4x^2)^2 - 1 = 32x^6 - 48x^4 + 18x^2 - 1$ is the Chebyshev polynomial of the first kind of degree 6.

5. *Heart:* $(2x^2 + 2y^2 + z^2 - 1)^3 - 0.1x^2z^3 - y^2z^3 = 0$.

6. *High Silhouette:* $x^6 - y^5 - 2x^3y + y^2 = 0$

Septic Surfaces

1. *Chmutov:* $T_7(x) + T_7(y) + T_7(z) + 1 = 0$ where $T_7(x) = 64x^7 - 112x^5 + 56x^3 - 7x$ is the Chebyshev polynomial of the first kind of degree 7.

2. *Labs* $P - U_\alpha = 0$ where $P = x^7 - 21x^5y^2 + 35x^3y^4 - 7xy^6 + 7z((x^2+y^2)^3 - 8z^2(x^2+y^2)^2 + 16z^4(x^2+y^2)) - 64z^7$, $U_\alpha = (z+a_5)((z+1)(x^2+y^2) + a_1z^3 + a_2z^2 + a_3z + a_4)^2$,$a_1 = (-12/7)\alpha^2 - 384/49\alpha - 8/7$,$a_2 = (-32/7)\alpha^2 + 24/49\alpha - 4$,$a_3 = (-4)\alpha^2 + 24/49\alpha - 4$,$a_4 = (-8/7)\alpha^2 + 8/49\alpha - 8/7$,$a_5 = 49\alpha^2 - 7\alpha + 50$ and $\alpha = -0.14010685$

Octic Surfaces

1. *Nonisol:* $x^8 - y^8 - 2x^4y + y^2 = 0$

2. *Chmutov:* $T_8(x) + T_8(y) + T_8(z) = 0$ where $T_8(x) = 128x^8 - 256x^6 + 160x^4 - 32x^2 + 1$.

3. *Endrass:* $64(x^2-1)(y^2-1)(ab) - (c+d+e)^2 = 0$ where $a = (x+y)^2 - 2$, $b = (x-y)^2 - 2$, $c = -4(1-\sqrt{2})(x^2+y^2)^2$, $d = 8(2-\sqrt{2})z^2 + 2(2-7\sqrt{2})(x^2+y^2)$ and $e = -16z^4 + 8(1+2\sqrt{2})z^2 - (1-12\sqrt{2})$. Like many higher order algebraic surfaces, the Endrass octic appears like a collection of surfaces.

Nonic Surfaces

1. *Chmutov:* $T_9(x)+T_9(y)+T_9(z)+1 = 0$ where $T_9(x) = 256x^9 - 576x^7 + 432x^5 - 120x^3 + 9x$ is the Chebyshev polynomial of the first kind of degree 9.

Surfaces of order more than 10

1. *Barth Decic:* Barth decic is a tenth order surface with the equation $8(x^2 - \phi^4y^2)(y^2 - \phi^4z^2)(z^2 - \phi^4x^2)(x^4+y^4+z^4-2x^2y^2-2x^2z^2-2y^2z^2) + (3+5\phi)(x^2+y^2+z^2-1)^2(x^2+y^2+z^2-2+\phi)^2 = 0$ where $\phi = (1+\sqrt{5})/2$ is the golden ratio.

2. *Sarti Dodecic:* This surface which is of order twelve with the equation $243\,S - 22\,Q = 0$, where $Q = (x^2+y^2+z^2+1)^6$ and $S = 33\sqrt{5}(s^-_{2,3} + s^-_{3,4} + s^-_{4,2}) + 19(s^+_{2,3} + s^+_{3,4} + s^+_{4,2}) + 10s_{2,3,4} - 14s_{1,0} + 2s_{1,1} - 6s_{1,2} - 352s_{5,1} + 336l_5^2l_1 + 48l_2l_3l_4$ with $l_1 = x^4+y^4+z^4+1, l_2 = x^2y^2+z^2, l_3 = x^2z^2+y^2, l_4 = x^2+y^2z^2, l_5 = xyz$, $s_{1,0} = l_1(l_2l_3 + l_2l_4 + l_3l_4)$, $s_{1,1} = l_1^2(l_2+l_3+l_4)$, $s_{1,2} = l_1(l_2^2+l_3^2+l_4^2)$, $s_{2,3,4} = l_2^3+l_3^3+l_4^3$, $s^\pm_{2,3} = l_2^2l_3 \pm l_2l_3^2$, $s^\pm_{3,4} = l_3^2l_4 \pm l_3l_4^2$, $s^\pm_{4,2} = l_4^2l_2 \pm l_4l_2^2$, and $s_{5,1} = l_5^2(l_2+l_3+l_4)$.

3. *Chmutov of Higher Orders:* $T_n(x) + T_n(y) + T_n(z) = 0$ for $n = 14, 18, 22$ and 50 and T_n is Chebyshev polynomial of first kind of degree n.

Recursive Definition: $T_0(x) = 1$, $T_1(x) = x$ and $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$

Sinusoidal Definition:

$$T_n(x) = \begin{cases} \cos(n \arccos(x)) & x \in [-1, 1] \\ \cosh(n\cosh^{-1}(x)) & x > 1 \\ (-1)^n \cosh(n\cosh^{-1}(-x)) & x < -1 \end{cases}$$

A.0.1 Non-Algebraic Surfaces Equation

1. *Torus:* $(c - \sqrt{x^2 + y^2})^2 + z^2 = a^2$.

2. *Blinn's Blobby:* $\sum_{i=1}^{N} \frac{r_i^2}{||x-c_i||^2+\epsilon} - 1.0 = 0$.

3. *Blobby:* $x^2 + y^2 + z^2 + \sin(4x) - \cos(4y) + \sin(4z) - 1.0 = 0$.

4. *Scherk's Minimal:* $\exp(z) * \cos(y) - \cos(x) = 0$.

5. *Diamond:* $\sin(x) * \sin(y) * \sin(z) + \sin(x) * \cos(y) * \cos(z) + \cos(x) * \sin(y) * \cos(z) + \cos(x) * \cos(y) * \sin(z) = 0$.

6. *Superquadrics:* Superquadric surfaces are given by the equation $|x|^m + |y|^m + |z|^m - 1.0 = 0$ for different values of m. Fractional values produces concave sides. The shape approximates a cube with rounded edges for high values of m.

Bibliography

[1] D. W. Arthur. The use of interval arithmetic to bound the zeros of real polynomials. In *J. Inst. Math. Appl.*, volume 10, pages 231–237, 1972.

[2] Chandrajit L. Bajaj, Jindon Chen, and Guoliang Xu. Modeling with cubic a-patches. *ACM Trans. Graph.*, 14(2):103–133, 1995.

[3] James F. Blinn. How to solve a cubic equation, part 1: The shape of the discriminant. *IEEE Comput. Graph. Appl.*, 26(3):84–93, 2006.

[4] James F. Blinn. How to solve a cubic equation, part 3: General depression and a new covariant. *IEEE Comput. Graph. Appl.*, 26(6):92–102, 2006.

[5] Jules Bloomenthal and Keith Ferguson. Polygonization of non-manifold implicit surfaces. In *SIGGRAPH '95*, pages 309–316, 1995.

[6] Antoine Bouthors and Matthieu Nesme. Twinned meshes for dynamic triangulation of implicit surfaces. In *Graphics Interface*, Montréal, may 2007.

[7] O Caprani, L Hvidegaard, M Mortensen, and T Schneider. Robust and efficient ray intersection of implicit surfaces. *Reliable Computing*, 6(1):9–21, 2000.

[8] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46. Eurographics Association, 2002.

[9] Erwin de Groot and Brian Wyvill. Rayskip: faster ray tracing of implicit surface animations. In *GRAPHITE '05: Proceedings of the 3rd international conference on*

Computer graphics and interactive techniques in Australasia and South East Asia, pages 31–36. ACM Press, 2005.

[10] Olivier Didrit, Luc Jaulin, Michel Kieffer, and Eric Walter. *Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics*. 2001, Springer-Verlag.

[11] Tom Duff. Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry. *SIGGRAPH Comput. Graph.*, 26(2):131–138, 1992.

[12] Jorge Florez, Mateu Sbert, Miguel Angel Sainz, and Josep Vehi. Improving the interval ray tracing of implicit surfaces. In *Computer Graphics International*, pages 655–664, 2006.

[13] Jack Goldfeather, Jeff P M Hultquist, and Henry Fuchs. Fast constructive-solid geometry display in the pixel-powers graphics system. In *SIGGRAPH '86*, pages 107–116, 1986.

[14] Simon Green, Yury Urlasky, and Evan Hart. Nvidia opengl sdk isosurface extraction using marching tetrahedra. http://developer.nvidia.com/, 2007.

[15] Pat Hanrahan. Ray tracing algebraic surfaces. In *SIGGRAPH '83*, pages 83–90, 1983.

[16] Eldon Hansen and William Walster. *Global Optimization Using Interval Analysis*. Marcel Dekker, 2003.

[17] John C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[18] D. Herbison-Evans. Solving quartics and cubics for graphics. In *Graphics Gems V*, pages 3–15. Academic Press, 1995.

[19] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied Interval Analysis*. Springer, 2001.

[20] James T. Kajiya. Ray tracing parametric patches. In *SIGGRAPH '82*, pages 245–254, 1982.

[21] D. Kalra and A. H. Barr. Guaranteed ray intersections with implicit surfaces. In *SIGGRAPH '89*, pages 297–306. ACM Press, 1989.

[22] Aaron Knoll, Younis Hijazi, Charles D Hansen, Ingo Wald, and Hans Hagen. Interactive ray tracing of arbitrary implicit functions. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007.

[23] Aaron Knoll, Younis Hijazi, Andrew Kensler, Mathias Schott, Charles D Hansen, and Hans Hagen. Fast and robust ray tracing of general implicits on the gpu. Technical Report 2007-014, University of Utah, 2007.

[24] R. Krawczyk. Newton-algorithmen zur bcstimmung yon nullstellen mit fehlerschranken. *Computing*, 4:187–201, 1969.

[25] Baoquan Liu, Li-Yi Wei, Xu Yang, Ying-Qing Xu, and Baining Guo. Nonlinear beam tracing on a gpu. Microsoft Research Technical Report MSR-TR-2007-34, March 2007.

[26] Charles Loop and Jim Blinn. Real-time GPU rendering of piecewise algebraic surfaces. In *SIGGRAPH*, pages 664–670, 2006.

[27] Charles T. Loop and James F. Blinn. Resolution independent curve rendering using programmable graphics hardware. In *SIGGRAPH*, pages 1000–1009, 2005.

[28] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87*, pages 163–169. ACM Press, 1987.

[29] William Martin, Elaine Cohen, Russell Fish, and Peter Shirley. Practical ray tracing of trimmed NURBS surfaces. *J. Graph. Tools*, 5(1):27–52, 2000.

[30] D. P. Mitchell. Robust ray intersection with interval arithmetic. In *Proceedings on Graphics interface '90*, pages 68–74, 1990.

[31] R. E. Moore and S. T. Jones. Safe starting regions for iterative methods. In *SIAM J. Numer. Anal.*, volume 14, pages 1051–1065, 1988.

[32] David Nister. An efficient solution to the five-point relative pose problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(6):756–777, 2004.

[33] K. Perlin and E. M. Hoffert. Hypertexture. In *SIGGRAPH '89*, pages 253–262, 1989.

[34] Fabio Policarpo and Manuel M. Oliveira. Relaxed cone stepping for relief mapping. chapter 18. *GPU Gems 3*, pages 409–428, 2007.

[35] Fabio Policarpo, Manuel M. Oliveira, and Joao L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. *ACM Trans. Graph.*, 24(3):935–935, 2005.

[36] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[37] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02*, pages 703–712, 2002.

[38] Sunil Mohan Ranta, Jag Mohan Singh, and P. J. Narayanan. GPU Objects. In *Proceedings of ICVGIP*, volume 4338 of *Lecture Notes in Computer Science*, pages 352–363. Springer, 2006.

[39] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005.

[40] Fabrice Rouillier and Paul Zimmermann. Efficient isolation of polynomial's real roots. *J. Comput. Appl. Math.*, 162(1):33–50, 2004.

[41] J. F. Sanjuan-Estrada, Leocadio G. Casado, and Inmaculada García. Reliable algorithms for ray intersection in computer graphics based on interval arithmetic. In *Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, pages 35–44, 2003.

[42] T. W. Sederberg and Geng-Zhe Chang. Isolating the real roots of polynomials using isolator polynomials. In *Algebraic Geometry and Applications*. Springer Verlag, 1993.

[43] J.S. Seland and Tor Dokken. Real time algebraic surface visualization. In *Supercomputing '06 Workshop: General-Purpose GPU Computing: Practice And Experience*, 2006.

[44] Andrei Sherstyuk. Fast ray tracing of implicit surfaces. *Computer Graphics Forum*, 18(2):139–147, 1999.

[45] Christian Sigg, Tim Weyrich, Mario Botsch, and Markus Gross. Gpu-based ray-casting of quadratic surfaces. In *Proceedings of Eurographics Symposium on Point-Based Graphics 2006*, pages 59–65, 2006.

[46] John M. Snyder. Interval analysis for computer graphics. In *SIGGRAPH '92*, pages 121–130, 1992.

[47] Gabriel Taubin. Distance approximations for rasterizing implicit curves. *ACM Trans. Graph.*, 13(1):3–42, 1994.

[48] Rodrigo Toledo and Bruno Levy. Extending the graphic pipeline with new gpu-accelerated primitives. Technical report, INRIA, 2004.

[49] Kees van Kooten, Gino van den Bergen, and Alex Telea. Point-based visualization of metaballs on a gpu. chapter 7. *GPU Gems 3*, pages 123–148, 2007.

[50] Eric Weisstein. Wolfram research. http://mathworld.wolfram.com/.

[51] T. Whitted and J. Kajiya. Fully procedural graphics. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 81–90, 2005.

[52] J. J. VAN WIJK. Ray tracing objects defined by sweeping a sphere. In *Eurographics' 84 Conference*, pages 73–82, 1984.

[53] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In *SIGGRAPH*, pages 269–277, 1994.

[54] Sven Woop, Jörg Schmittler, and Philipp Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.

[55] G. Wyvill and A. Trotman. Ray-tracing soft objects. In *CG International '90*, pages 469–476, New York, NY, USA, 1990. Springer-Verlag New York, Inc.