Real-Time Ray-Tracing of Implicit Surfaces on the GPU

Jag Mohan Singh and P J Narayanan, *Member, IEEE*
Centre for Visual Information Technology,
International Institute of Information Technology(IIIT), Hyderabad, India

***Abstract*— Compact representation of geometry using a suitable procedural or mathematical model and a ray-tracing mode of rendering fit the programmable graphics processor units (GPUs) well. Several such representations including parametric and subdivision surfaces have been explored in recent research. The important and widely applicable category of the general implicit surface has received less attention. In this paper, we present a ray-tracing procedure to render general implicit surfaces efficiently on the GPU. Though only the fourth or lower order surfaces can be rendered using analytical roots, our *adaptive marching points* algorithm can ray-trace arbitrary implicit surfaces without multiple roots, 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 sign-test can handle any surface without multiple roots. The Taylor-test that uses ideas from interval analysis can ray-trace many surfaces with complex roots. Overall, a simple algorithm that fits the SIMD architecture of the GPU results in high performance. We demonstrate the ray-tracing of algebraic surfaces up to order 50 and non-algebraic surfaces including a Blinn's blobby with 75 spheres at better than interactive frame rates.**

***Index Terms*— Ray-Tracing, Implicit Surfaces, GPU Rendering**

I. Introduction

Current Graphics Processor Units (GPUs) are optimized to render polygons. Programmable vertex, geometry, and pixel stages have made it widely applicable beyond polygon rendering. Ray-tracing is of particular interest as each fragment effectively handles 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.

Implicit and procedural geometry are important in computer graphics. They are compact and can be evaluated on the fly. Implicit geometry is defined by an equation $S(x, y, z) = 0$. Different forms of $S(\cdot)$ are possible. 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 $\max_m(i_m + j_m + k_m)$. Non-algebraic surfaces can be of different functional forms. Implicit surfaces are popular in fluid simulation, scientific computing, weather modeling, etc. They are often used to visualize high-dimensional data after fitting them with a suitable implicit function.

Polygonization is the most common method of rendering implicit surfaces [1]. Dynamic implicit surfaces with changing topology poses great challenges to this process. The implicit form allows compact and exact definition 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 small triangles, but at the loss of compactness. Direct rendering using ray tracing performed on the GPU can retain both. The computing power of the GPUs grows 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. 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. 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.

Ray-tracing is an application ideally suited to the high computing and low memory performance of multicore and manycore architectures [2]. Woop et al. argue for a programmable ray-tracing unit much like the GPUs and show an implementation using FPGAs for real time rendering [3]. 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 [4]. Our work strongly endorses this line of thinking by extending exact and high-quality ray-tracing to a large class of arbitrary implicit surfaces on the GPUs. Modeling using procedural or implicit techniques and rendering using ray-tracing is likely to be important components of high performance graphics in the future.

In this paper, 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 for simple surfaces. Solution to the equation $F_f(t) = 0$ depends on its form. Interactive ray-tracing has been achieved only for lower order implicit surfaces. These include algebraic surfaces up to order 4 using analytical roots on the GPU [5] and selected algebraic surfaces and some non-algebraic surfaces using interval-analysis and affine-arithmetic on the GPU [6]. We introduce the *adaptive marching points* (AMP) algorithm which samples each ray in t to find the first solution of the equation $S(x, y, z) = S(p(t)) = 0$. The sampling step size adapts to the distance to the surface and the closeness to a silhouette. This method matches the SIMD architecture of the GPUs and can handle arbitrary implicit surfaces. We show that simple and seemingly non-promising algorithms that suit the

Email: jagmohan@research.iiit.ac.in, pjn@iiit.ac.in

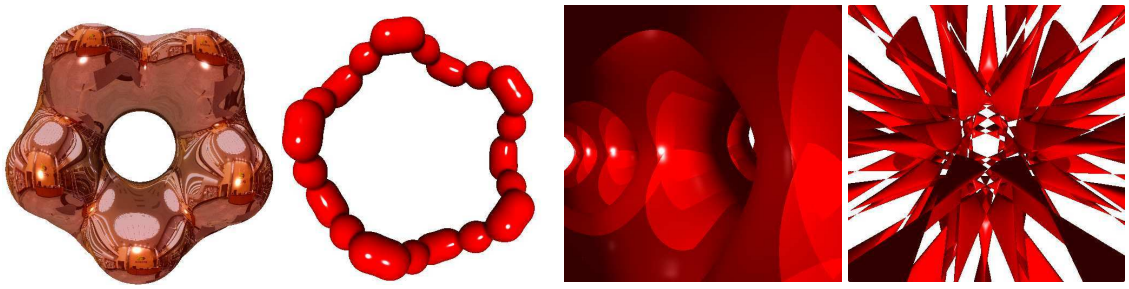

Fig. 1. Ray-traced Blinn's blobby with 75 spheres with environment-mapping and shading (35 fps), Chmutov dodecic with four light sources (215 fps), and Barth decic with one light source (573 fps).

architecture well can deliver very high performance on the GPUs. Our method finds the exact intersections on algebraic and general implicit surfaces when multiple roots are not present. A root-test inspired by interval analysis can find the correct intersections for multiple roots on several surfaces. We show results on several algebraic surfaces of order up to 50 and non-algebraic surfaces like superquadrics, sinusoids, and blobbies with exact lighting and shadowing at significantly better than real-time rates. Our technique can handle dynamic surfaces also since it evaluates the equation directly in each frame without any preprocessing. Figure 1 presents some of the surfaces ray-traced using our method.

The AMP algorithm delivers high performance on the GPUs, but can work on the CPU also. We ray-trace many of these surfaces on the CPU, but the rendering time varies from 1 second for cubic surfaces to tens of seconds for algebraic surfaces of order 20 and above. While this is the only reported rendering of such high order surfaces on the CPU, the far from real-time speeds make it unattractive as an option. The simplicity of the algorithm, on the other hand, fits the architecture of the GPU well, and extracts up to 70% of its peak FLOPS.

Section II reviews the previous work related to the topics of this paper. Section III presents the adapting marching points method. Results of our algorithm on different algebraic and non-algebraic surfaces is presented in Section IV. Conclusions and directions for future work are presented in Section V. Appendix presents the equations of the implicit surfaces used in the paper along with simple screenshots.

II. Related Work

Implicit surfaces can be converted to triangulated models prior to rendering them using traditional graphics [1]. The marching cubes algorithm can be used to create polygonal models from implicit functions [7]. A high-performance marching tetrahedron package was released on the GPU recently [8]. Triangulation increases the size and the bandwidth needs of the representation. and goes against the strengths of the GPU.

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 [9]. Wald et al. achieved interactive ray tracing of RBF implicits using a specialized intersection algorithm [10]. Kajiya reduces ray tracing of spline surfaces to a globally convergent method [11]. Interval-analysis has also been used for robust root isolation by many [12], [13], [14], [15], [16]. Mitchell isolates the root using repeated bisections till the interval in t contains a single root [12]. Reliable interval-extensions, however, are difficult to compute for large intervals in the domain of complex functions. Our implementation of Mitchell's method ray-traced surfaces up to order 5 at interactive frame rates on the GPU using the exact interval extension [17]. Surfaces beyond order 5 could not be ray-traced robustly due to the difficulty in the interval extension. Sub-intervals, branch and bound schemes, octree grids, etc., have been used to increase the reliability of interval-based methods. Knoll et al. achieve 30 fps on a superquadric and 6 fps on a few sextic surfaces using the CPU and the SSE hardware [16], 121 fps on a quartic surface, 88 fps on a sextic surface and 16 fps on a decic surface using affine arithmetic based extension on GPU [6].

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 [18] and other implicit forms. Iterative methods critically depend on good initialization of the roots, which is difficult for complex equations. An alternative is to bracket the roots to an interval in t and then solve it using an iterative technique [19], [20]. Interval arithmetic have also been used [21], [22], which are more robust at critical regions. Most of these methods cannot be implemented easily on the SIMD architecture of the GPUs, however. Ray-tracing has been adapted to the GPU for general polygonal models. Purcell et al. performed multipass ray tracing [23] and Carr et al. combined CPU and GPU computations for recursive ray tracing [24]. These methods work for general objects but are slow. Spheres and other quadric primitives were ray-traced on the GPU using per-fragment ray-quadric intersection and optimized bounding boxes [25], [26], [27]. Adamson et al. performed ray intersections with local polynomial approximation inside a sphere for large polygonal models [28]. Hadwiger et al. ray-cast implicit surfaces defined on a regular volume grid using adaptive sampling and iterative refinement [29]. Loop and Blinn showed resolution independent rendering of quadratic and cubic-spline curves on the GPU [30] and extended it to render piecewise algebraic surfaces up to fourth order [5]. Seland and Dokken rendered algebraic surfaces up to order five on the GPU [31] 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. Our method keeps the process simple to match the GPU by not evaluating the complex univariate polynomials.

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 [32] and other implicit surfaces [33], [34]. Kalra and Barr ray-traced

LG-implicit surfaces using Lipschitz constants [33]. Hart used variable step sizes in sphere tracing based on a geometric distance function evaluated at the current point [34]. The Lipschitz theory or geometric distances do not extend easily to complex surfaces, however. We follow the point sampling approach, but change the step size using simpler measures that suit the GPU.

Our method samples or searches along each ray till a step covers a root. Step size is adapted using the algebraic distance to the surface and proximity to a local silhouette. We use a simple interval-based test for root-containment for robustness. Our method works on arbitrary implicit surfaces with simple roots since only samples of it are needed. While AMP can work on the CPU as well as the GPU, its simplicity achieves high performance on the restricted parallel architecture of the GPU. Methods involving interval analysis do not adopt to higher order surfaces easily due to the unavailability of robust interval extensions. Such methods are also slower due to the conditionalities in the program that do not suit the GPU. We can ray-trace algebraic surfaces of very high order – we show an order 50 surface – and several non-algebraic surfaces at framerates upwards of 100. We also handle dynamic implicit surfaces with no loss in performance as the surface is evaluated directly in each frame with no precomputations.

III. Adaptive Marching Points Algorithm

The points on the ray for a pixel or fragment f are given in the parametric form by $p(t) = O + tD_f$, where t is the ray parameter, O the camera center, and D_f the direction of the ray. Substituting for x, y, z from the ray equation into the surface equation $S(x, y, z) = 0$, we get

$$F_f(t) = 0. \tag{1}$$

The smallest, real, positive solution for t gives the point of intersection of the ray with the object. 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 of simple implicit surfaces.

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

Root finding may need the values of the function $F_f(t)$ and its derivatives $F'_f(t), F''_f(t)$, etc. 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) 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 with coefficients depending on the viewpoint and the ray. For example, a single sixth order expression x^3y^3 of $S(\cdot)$ 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 using 5 more multiplications. The Barth decic (Section IV) can be evaluated using about 30 terms as $S(p(t))$ 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 be calculated using the gradient as the dot-product $\vec{\nabla} S(x, y, z) \cdot D_f$. The situation is the same for the univariate expressions of the derivatives. Loop and Blinn use GPU's interpolation hardware to evaluate the coefficients of the polynomial by sending a symmetric tensor of rank $d - 1$ with $\binom{d+2}{d-1}$ unique elements from the vertex shader for each vertex of the tetrahedron [5]. 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 each vertex for an algebraic surface of order d.

B. Adaptive Sampling of the Ray

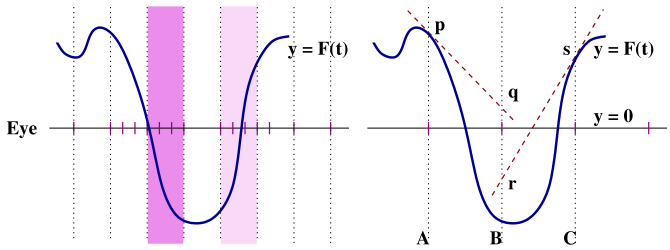

Fig. 2. Marching points algorithm samples uniformly in the ray parameter t. The sign test identifies the first interval where the function changes sign at the endpoints (darker shaded region on the left). Sign test will fail as the step size increases (right). Roots will be isolated in intervals [A, B] and [B, C] but not [A, C]

A balanced computation load and short and simple computations are critical to good performance on the GPUs, given their SIMD model. Methods that use $S(x, y, z)$ values are likely to be faster than those that use $F_f(t)$ values. An exceedingly simple root-isolation scheme is to sample regularly along the ray till the function $F_f(t)$ crosses zero between successive samples. The computation is low as only $F_f(t) = S(p(t))$ needs to be evaluated at the sample points. This *marching points* scheme can be used for arbitrary implicit surfaces, even those with difficult derivatives or for general piecewise algebraic surfaces without derivatives at boundaries [17]. The performance of the algorithm depends on the marching or sampling step-size. The optimal step-size may differ from one surface to another.

The worst case running time of this scheme is linear in the number of steps in the total range in t. The step-size needs to be chosen so as to not miss any root. We can observe that large step-sizes suffice in empty space, but small steps are necessary near the surface and near the silhouettes. The *adaptive marching points* (AMP) algorithm varies the step size based on the closeness of the point to the surface and to a silhouette. The step-size should be small near the surface and smaller near silhouettes (Figure 3).

Geometric distances are reliable measures of proximity to a surface but are surface dependent and are not available for arbitrary implicit surfaces. Lipschitz bounds have been used to estimate the optimum step size for efficient ray-tracing [33], [34]. Taubin used the ratio $\frac{F(t)}{|F'(t)|}$ as a measure for signed geometric distance to the function $F(t)$ [35]. However, it is useful only for low-order algebraic surfaces and for points close to the surface. Defining geometric distance and Lipschitz bounds for arbitrary algebraic and non-algebraic surfaces is hard and will be a fruitful research direction for the future.

Distance Adaptation: The magnitude of $S(x, y, z)$ gives the algebraic distance from a point to the surface. We normalize $S(x, y, z)$ such that the highest coefficient of the top-order term is unity and use $|S(x, y, z)|$ as a *proximity measure* that is zero close

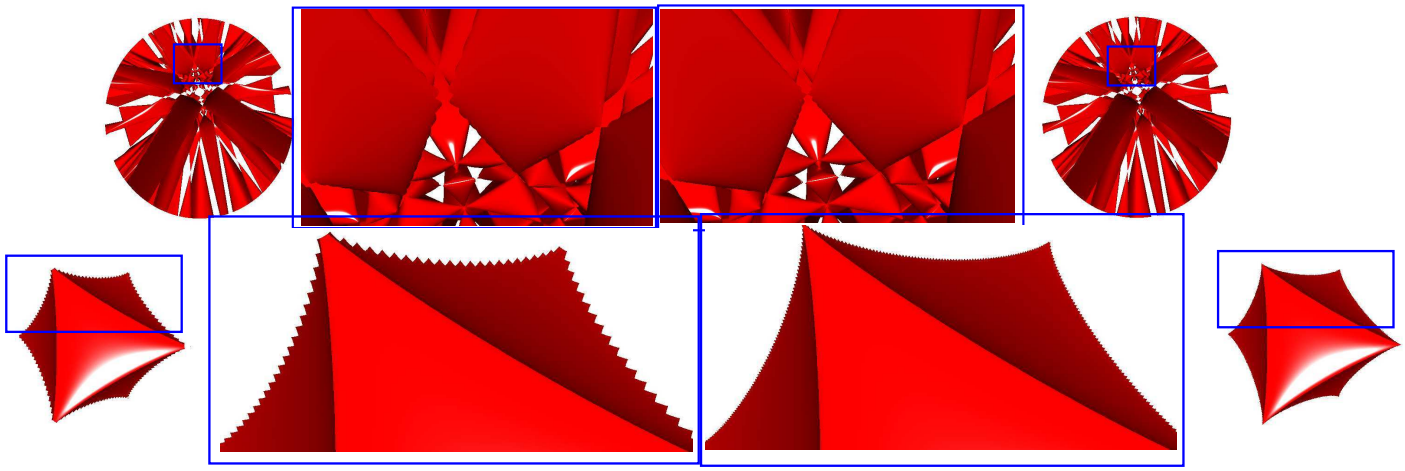

Fig. 4. Top row: Barth tenth order surface without silhouette adaptation (left) and with it (right). The zoomed views in the middle show great reduction in aliasing for the internal silhouettes. Bottom row: Superquadric surface without (left) and with (right) silhouette adaptation with zoomed views in the middle.

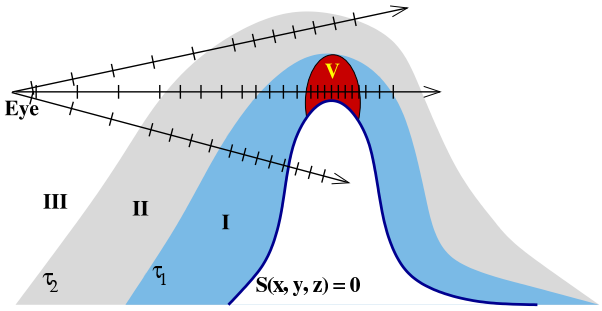

Fig. 3. The step size is adapted to the distance to the surface and the proximity to a silhouette. Region III will have the largest step size and the region I will have the smallest, based on the proximity measure $|S(x, y, z)|$. The step size is further reduced when the horizon condition is true (the darkened region V) as the surface normal is nearly perpendicular to the viewing direction.

to the surface. The step-size can varies as a monotonic function of it. In practice, we use a piecewise constant approximation and vary the step size in octaves, starting with a base step size of b. The base step size is doubled if the current point is far away from the surface and halved if close to it, using two thresholds τ_1 and τ_2. Different step sizes are used in regions of different colour/shade shown in Figure 3. The thresholds are set based on the coefficients of $S()$.

Algorithm 1 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: Set the stepsize s using Equation 2.
5: **if** rootExistsIn $(t, t + s)$ **then**
6: Goto step 11 with $[t, t + s]$ as the isolated interval
7: **end if**
8: $t = t + s$
9: **end while**
10: No isolated interval. Discard pixel
11: Perform 10 bisections of the isolated interval, keeping the half with the root in each.

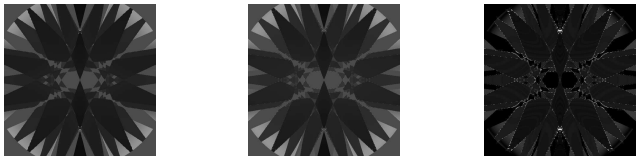

Fig. 5. Number of steps taken along each ray for a Barth tenth order surface for the AMP algorithm without silhouette adaptation (left), with it (middle) and difference image scaled by 2 for legibility (right). Darker colour indicates fewer steps.

Silhouette Adaptation: The view-dependent silhouettes represent regions of close and multiple roots. It is important to sample the ray finely near them. We do that by decreasing the step-size near the silhouettes. The derivative magnitude $|F'_f(t)|$ serves as a *horizon measure* which is close to zero near internal and external silhouettes of even complex implicit surfaces. As described earlier, $F'_f(t) = \vec{\nabla} S(x, y, z) \cdot D_f$ and can be computed efficiently. The step size can be a monotonic function of $|F'_f(t)|$. In practice, we halve the step-size when the horizon condition $|F'_f(t)| \leq \epsilon$ is satisfied (Algorithm 1). Thus, region V of Figure 3 will have reduced step sizes in order to render silhouettes well. Olievera et al. used the angle between the viewing direction and the surface normal to control the step size while ray-tracing height-fields on the GPU [36]. Hadwiger et al. used a multiple of base sampling rate for better quality near silhouettes for better quality [29].

Combining distance and silhouette adaptation, we fix the step-size in each iteration using the following formula

$$s = \begin{cases} b/4 & \text{if } |S(p(t))| \leq \tau_1 \text{ and } |\vec{\nabla} S(p(t)) \cdot D_f| \leq \tau_3 \\ b/2 & \text{if } |S(p(t))| \leq \tau_1 \\ 2b & \text{if } |S(p(t))| > \tau_2 \\ b & \text{otherwise} \end{cases} \quad (2)$$

where b is the base stepsize and τ_1, τ_2 and τ_3 are thresholds. The root-containment test (Step 5, Algorithm 1) is also critical to isolating roots and can be implemented in different ways. Two promising ones are the sign test and the Taylor test described below.

Sign test: Root exists if the function changes sign between the end points of the step, i.e., if $(S(p(t_i)) * S(p(t_{i+1})) < 0)$. This test is simple to implement as only the function values at the sample points are needed. It is a strict test that does not produce

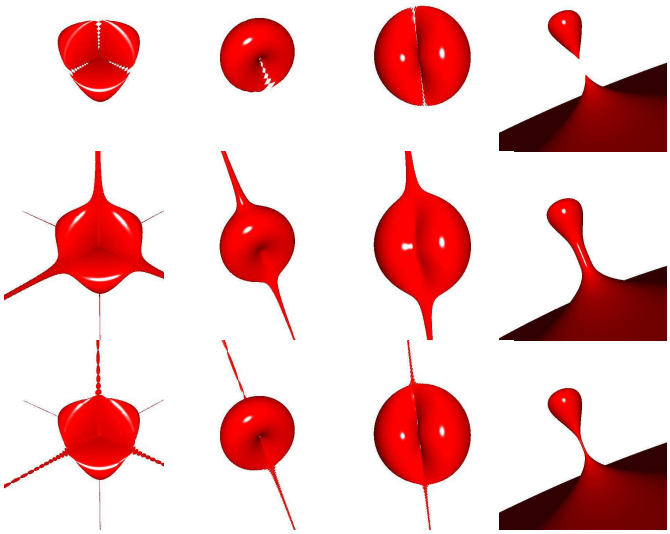

Fig. 6. Top row: Steiner, Cross Cap, Miter, Kiss and High Silhouette surfaces ray-traced using the AMP method with sign test. Multiple roots are missed by it. Middle row: Surfaces shifted by 0.01 using AMP and sign test. Region of multiple roots tend to be fattened. Bottom row: Same surfaces rendered using AMP and Taylor test. The performance is more robust for multiple roots.

false roots. It misses roots if an even number of roots are in the step, however.

Taylor test: This test checks if the function values and linear extensions of them enclose a zero. Interval arithmetic has been used for robust root finding and this test is inspired by it. 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 (Figure 2). This works adequately for moderate lengths of intervals. The extension of F in the interval $[t_i, t_{i+1}]$ is defined as $\tilde{F}([t_i, t_{i+1}]) = [\min\ \{p, q, r, s\},\ \ \max\ \{p, q, r, s\}]$, where

$$\begin{aligned} q &= F(t_i) + F'(t_i)\tfrac{(t_{i+1}-t_i)}{2}, & p &= F(t_i), \\ r &= F(t_{i+1}) - F'(t_{i+1})\tfrac{(t_{i+1}-t_i)}{2}, & s &= F(t_{i+1}) \end{aligned} \tag{3}$$

This test is slower than the sign test because of the derivatives but larger step-sizes can be used. This test can produce false roots, but works robustly in practice and can handle multiple roots well.

The AMP scheme can, however, miss multiple roots or produce false roots based on the specific test used and the step size. A comparison of different tests for multiple roots is shown in Figure 6. The sign 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 problem (Figure 6). Though we are rendering a different surface, the results are close. Offsetting is similar to the $S(x, y, z) \leq \epsilon$ test for roots used by sphere tracing [34]. The Taylor test imitates interval extension and produces robust results similar to the interval-based method (Figure 6). Figure 4 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 shows the number of iterations used for each pixel as a measure of the work done for the Barth decic surface. The extra effort near the silhouettes can be observed when silhouette adaptation is used.

Algorithm 2 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 and the camera center to the geometry shader.

Geometry Shader:

1: Transform the quad to a screen-facing one and pass 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 and find the root using the AMP algorithm.
3: Shoot rays to light source and perform root-isolation for it. If root is found, the point is under shadow.
4: Compute colour and depth using the position, normal, and shadowing at the intersection point.

The most work is done for non-intersecting rays as the entire range of t values need to be sampled.

IV. Results

We could render algebraic surfaces up to order 50 robustly including all surfaces shown in the MathWorld site and several non-algebraic and transcendental objects. Screenshots of these surfaces appear in Appendix. The equations of the corresponding surfaces is given in the Appendix. First, we display the overall ray-tracing algorithm and some of its implementation issues.

A. Overall Algorithm and Implementation Issues

The overall pseudo code is given in Algorithm 2. The implementation is in OpenGL/GLSL for the SM4.0 architecture of the Nvidia 280 GTX GPU. The following steps were used for efficiency. (a) The shaders for the function to evaluate the expression $S(p(t))$ and its gradient (if necessary) are synthesized on the fly by the CPU. The shaders are compiled on the fly in any case. We also compute $S(p(t))$ and its gradient together in one function for faster evaluation as this will help in minimizing redundant computations where they exist. (b) 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. (c) Tight screen-space and depth bounds improves the timing greatly. We couldn't take much advantage of this as the complex surfaces we used cannot be bounded easily.

B. Rendering Times

Table I presents the frame rates of our algorithm on several algebraic and non-algebraic surfaces with and without shadow rays on an Nvidia 280 GTX for a resolution of 512×512. (See our technical report for timings on Nvidia 8800 GTX [17]). Results are given for ray-tracing with and without shadow-rays. Shadow rays start from each point and perform the root-isolation using exactly the same algorithm. The bisection to get the exact root is not necessary as we only need to know if there is an intersection. Our rendering times are better than work reported in the literature

for higher order surfaces and very competitive for lower order ones. We also obtain real-time results on surfaces much more complicated than have been reported before. The best reported effort by Knoll et al. achieve a frame rate of 121 on a quartic surface, 88 on a sextic surface, and up to 108 on superquadric-like surfaces on the GPU [6]. They use interval or affine arithmetic, which may not easily extend to complex algebraic surfaces. Table II shows comparison with the surfaces they use. Our method is faster than their scheme and allows for simple extension to even higher order surfaces than used by them. Table III shows the rendering of different surfaces on the CPU. The rendering times on CPU is slower by an orders of magnitude of that on GPU due to lack of parallelism. This shows that the AMP algorithm is suited for GPU. Figure 7 shows screenshots of some of the surfaces.

C. Performance: Discussion

The performance of any GPU ray-tracing algorithm depends on three aspects: the algorithmic complexity, the per-pixel computational load, and the match with SIMD architecture of the GPUs. The adaptive marching points algorithm has a linear complexity in the distance to the surface as the ray is sampled till a hit. Rays that do not intersect with the surface are the most expensive. Mitchell's method [12] and others inspired by it use a recursive formulation and have a logarithmic complexity, but suffer due to the non-availability of robust interval extensions for arbitrary surfaces. The AMP method has low per-pixel computations as only $S(x, y, z)$ and its gradient are evaluated. Root finding algorithms that use higher derivatives and the $F_f(t)$ formulation have higher computations per pixel. The sampling approach also fits the SIMD architecture of the GPU; Algorithm 1 has minimal divergence in the shader code between fragments compared to interval based methods [6], [17], which is key to good performance on the GPUs. The higher complexity is thus more than compensated by the lighter computation load and good match with the architecture.

We explore the performance of our algorithm on the Chmutov 14th order surface as a typical case. A normal view of the surface from outside evaluated the surface equation on an average 70 times per ray when the sign test is used without shadows. Maximum number was for the rays with no intersection. Over 75 floating point operations are needed per evaluation of the $S(x, y, z)$, giving a computation load of 1.38 billion operations per frame. The surface renders at about 460 fps on an Nvidia GTX 280. This translates to 635 GFLOPS of sustained computations, which is about 65% of the peak floating point performance of the GPU. Though AMP algorithm works on the CPU also (see Table III), the 1.38 GFLOP per frame makes its performance non-attractive. Each frame is rendered in about 9 seconds, giving a performance of about 150 MFLOPS or about 3% of the peak rating of the CPU.

D. Dynamic Implicit Objects

A dynamic implicit object changes its form over time and are challenging to render. The rayskip algorithm ray-traces dynamic implicits by exploiting the temporal and spatial coherence of ray-implicit intersections [37]. Knoll et al. render dynamic implicits as 4D implicits in an (x, y, z, t) space [16]. Our scheme ray-traces the surface independently in each frame without any precomputations or subdivisions. Thus, the equation can change each frame without affecting the performance in any way. Temporal coherence can be used but the additional book-keeping slows down the process in practice on the GPUs. Figure 8 shows a few views a Blinn's Blobby with 75 blobbies and a twisting superquadric. The topology of the blobby changes from each being independent spheres to a single fused object as seen in the video. We render the blobby at a framerate of 35 fps or more and the superquadric at over 900 fps.

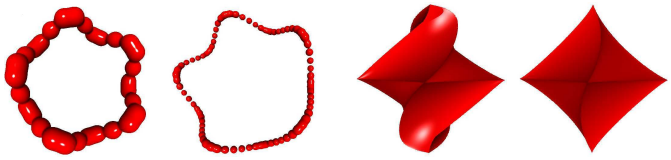

Fig. 8. Dynamic objects: Two views of an evolving object with 75 Blinn's blobbies rendered at over 35 fps (left) and of twisting superquadric rendered at over 900 fps (right).

E. Varying step-size and thresholds

The base step-size and thresholds for adaptation are selected based on the surface and has an impact on the correctness of the image. Figure 9 shows the effect of varying the step-size on Barth Decic. Many intersections are missed when using large step sizes and the effect appears like toothed edges of the surface. The quality gets better as the step-size decreases. Simultaneously, the frame rate also decreases as more intervals need to be tested. The effect of varying the thresholds used in Algorithm 1 is also show in Figure 9 in a Chmutov Octic surface. We set $\tau_2 = 2\tau_1$ and $\tau_3 = \frac{1}{2}\tau_1$ for this exploration. Larger steps are taken until very close to the surface when the threshold is small and several intersections are missed, resulting in artifacts. As the threshold increases, smaller steps are used and the quality improves, but at the cost of rendering speed. There is also a trade-off between the step-size and the threshold. If the step-size is small, the threshold has less impact on the results. The accompanying video shows the effect of varying the step-size and threshold more clearly.

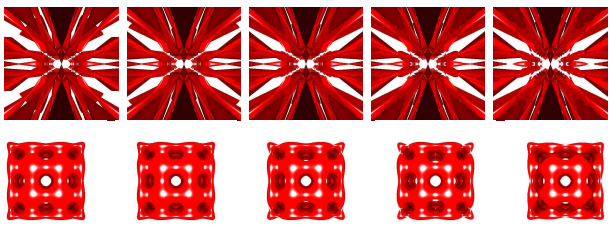

Fig. 9. Top row: Barth Decic with stepsizes 0.015 (573), 0.025 (601), 0.045 (663) and 0.085 (806). Bottom row: Chmutov Octic with τ_1 0.01 (1053), 0.009 (1160), 0.007 (1411), 0.0065 (1481) and 0.006 (1555) for each $\tau_2 = 2\tau_1$ and $\tau_3 = \frac{1}{2}\tau_1$. Numbers in parenthesis give the FPS.

F. Correctness and Robustness

Interval Arithmetic approximates the convex hull of the surface, which gets refined to the actual surface when the interval is smaller [6]. The AMP method searches for intersection with the surface in discrete steps and can isolate any single root with a small-enough step size. Multiple roots pose challenges to all root finding techniques, but the interval-based methods show more robustness. The Taylor test is equivalent to central approximation

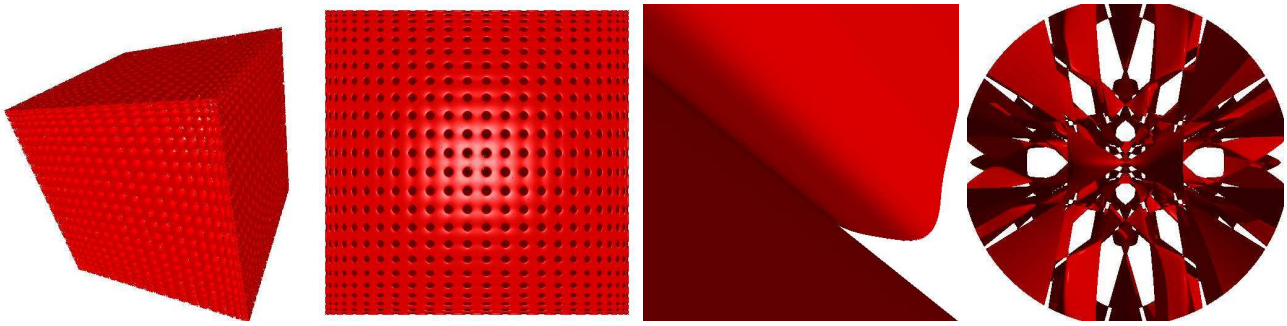

Fig. 7. Screenshots of (from left to right): Chmutov (50th order), Chmutov 50 (zoomed), NonIsol [8] and Sarti [12] surfaces

of Interval Arithmetic (IA) when step-size is small and will inherit the correctness of IA methods.

G. Limitations

The performance of the AMP algorithm depends on the step size. The optimum step-size is surface dependent; the Endrass and Hunt surfaces needed very small steps to work correctly as shown in Table I. A conservative step-size can produce correct results, but with a drop in speed.

The GPUs primarily support only single precision arithmetic. This has not been a problem in the class of algebraic and non-algebraic surfaces we explored. The computation of high order polynomials, however, needs to be done carefully as numerical instabilities can produce wrong results. The Chmutov surfaces of orders greater than 18 have serious artifacts due to false roots near ± 1 when the Chebyshev polynomial equations are evaluated directly using the powers of x, y, and z [17]. This was not due to lower precision as the double precision implementation on the CPU exhibits the same behavior. However, the problems disappear when the iterative or the sinusoidal definitions of the Chebyshev polynomial is used. (See the Appendix for the iterative and sinusoidal definitions of the Chmutov surfaces.) The problem is thus with the computation of higher powers of polynomials, which is a fact to be kept in mind when rendering complex surfaces. 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 I. The direct evaluation of the polynomial is the quickest for orders up to 18.

Our algorithm had difficulty rendering some of extremal implicit surfaces. Such surfaces have zero mean curvature and are very challenging to handle. We could render most other implicit surfaces, though complex surfaces could exhibit numerical instabilities as described above.

V. Conclusions & Future Work

We presented a scheme to ray-trace arbitrary algebraic and non-algebraic implicit surfaces on the GPU at very high frame rates using the adaptive marching points method in this paper. The sign-test based root isolation suffices for surfaces with simple roots; the first-order interval based Taylor-test for root isolation extends this to many surfaces with multiple roots. The simplicity of the method is the key factor behind the high performance on current GPUs. The limited-precision of current GPUs was not a constraint even on higher order surfaces. 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 [4]. We believe this will be a direction in 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 for simple implicit surfaces 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.

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.

Acknowledgments: We thank the Naval Research Board of India for the partial financial support for this research and Nvidia for generous equipment donations. We also acknowledge the helpful discussions with Charles Loop and Li-Yi Wei of Microsoft Research. We also acknowledge the helpful discussions with Prof. C. N. Kaul of IIIT. We thank the anonymous TVCG reviewers for their suggestions that improved the paper.

References

[1] J. Bloomenthal and K. Ferguson, "Polygonization of non-manifold implicit surfaces," in *SIGGRAPH '95*, 1995, pp. 309–316.

[2] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1176–1185, 2005.

[3] S. Woop, J. Schmittler, and P. Slusallek, "RPU: a programmable ray processing unit for realtime ray tracing," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 434–444, 2005.

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

[5] C. Loop and J. Blinn, "Real-time GPU rendering of piecewise algebraic surfaces," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 664–670, 2006.

[6] A. Knoll, Y. Hijazi, A. Kensler, M. Schott, C. D. Hansen, and H. Hagen, "Fast and robust ray tracing of general implicits on the GPU," University of Utah (To appear Computer Graphics Forum), Tech. Rep. UUSCI-2007-014, 2007.

[7] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *SIGGRAPH '87*. ACM Press, 1987, pp. 163–169.

[8] S. Green, Y. Urlasky, and E. Hart, "Nvidia OpenGL SDK isosurface extraction using marching tetrahedra," http://developer.nvidia.com/, 2007.

[9] P. Hanrahan, "Ray tracing algebraic surfaces," in *SIGGRAPH '83*, 1983, pp. 83–90.

[10] I. Wald and H.-P. Seidel, "Interactive Ray Tracing of Point Based Models," in *Proceedings of 2005 Symposium on Point Based Graphics*, 2005.

[11] J. T. Kajiya, "Ray tracing parametric patches," in *SIGGRAPH '82*, 1982, pp. 245–254.

[12] D. P. Mitchell, "Robust ray intersection with interval arithmetic," in *Proceedings on Graphics interface '90*, 1990, pp. 68–74.

[13] T. Duff, "Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry," *SIGGRAPH Comput. Graph.*, vol. 26, no. 2, pp. 131–138, 1992.

[14] O. Caprani, L. Hvidegaard, M. Mortensen, and T. Schneider, "Robust and ecient ray intersection of implicit surfaces," *Reliable Computing*, vol. 6, no. 1, pp. 9–21, 2000.

[15] J. Florez, M. Sbert, M. A. Sainz, and J. Vehi, "Improving the interval ray tracing of implicit surfaces," in *Computer Graphics International*, 2006, pp. 655–664.

[16] A. Knoll, Y. Hijazi, C. D. Hansen, I. Wald, and H. Hagen, "Interactive ray tracing of arbitrary implicit functions," in *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007.

[17] J. M. Singh and P. J. Narayanan, "Real-time ray-tracing of implicit surfaces on the GPU," International Institute of Information Technology, Tech. Rep. 2007-72, 2007.

[18] J. F. Blinn, "How to solve a cubic equation, part 1: The shape of the discriminant," *IEEE Comput. Graph. Appl.*, vol. 26, no. 3, pp. 84–93, 2006.

[19] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[20] F. Rouillier and P. Zimmermann, "Efficient isolation of polynomial's real roots," *J. Comput. Appl. Math.*, vol. 162, no. 1, pp. 33–50, 2004.

[21] E. Hansen and W. Walster, *Global Optimization Using Interval Analysis*. Marcel Dekker, 2003.

[22] R. Krawczyk, "Newton-algorithmen zur bcstimmung yon nullstellen mit fehlerschranken," *Computing*, vol. 4, pp. 187–201, 1969.

[23] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware." in *SIGGRAPH '02*, 2002, pp. 703–712.

[24] N. A. Carr, J. D. Hall, and J. C. Hart, "The ray engine," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association, 2002, pp. 37–46.

[25] R. Toledo and B. Levy, "Extending the graphic pipeline with new gpu-accelerated primitives," INRIA, Tech. Rep., 2004.

[26] C. Sigg, T. Weyrich, M. Botsch, and M. Gross, "GPU-based ray-casting of quadratic surfaces," in *Proceedings of Eurographics Symposium on Point-Based Graphics 2006*, 2006, pp. 59–65.

[27] S. M. Ranta, J. M. Singh, and P. J. Narayanan, "GPU Objects," in *Proceedings of ICVGIP*, ser. Lecture Notes in Computer Science, vol. 4338. Springer, 2006, pp. 352–363.

[28] A. Adamson and M. Alexa, "Ray tracing point set surfaces," in *SMI '03: Proceedings of the Shape Modeling International 2003*. Washington, DC, USA: IEEE Computer Society, 2003, p. 272.

[29] M. Hadwiger, C. Sigg, H. Scharsach, K. Buhler, and M. Gross, "Real-time ray-casting and advanced shading of discrete isosurfaces," in *Eurographics 2005*, 2005, pp. 303–312.

[30] C. T. Loop and J. F. Blinn, "Resolution independent curve rendering using programmable graphics hardware." *ACM Transaction on Graphics*, vol. 24, no. 3, pp. 1000–1009, 2005.

[31] J. Seland and T. Dokken, "Real time algebraic surface visualization," in *Supercomputing '06 Workshop: General-Purpose GPU Computing: Practice And Experience*, 2006.

[32] K. Perlin and E. M. Hoffert, "Hypertexture," in *SIGGRAPH '89*, 1989, pp. 253–262.

[33] D. Kalra and A. H. Barr, "Guaranteed ray intersections with implicit surfaces," in *SIGGRAPH '89*. ACM Press, 1989, pp. 297–306.

[34] J. C. Hart, "Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces." *The Visual Computer*, vol. 12, no. 10, pp. 527–545, 1996.

[35] G. Taubin, "Distance approximations for rasterizing implicit curves," *ACM Trans. Graph.*, vol. 13, no. 1, pp. 3–42, 1994.

[36] F. Policarpo, M. M. Oliveira, and J. L. D. Comba, "Real-time relief mapping on arbitrary polygonal surfaces," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 935–935, 2005.

[37] E. de Groot and B. 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*. ACM Press, 2005, pp. 31–36.

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 I

FRAME RATES FOR ALGEBRAIC AND NON-ALGEBRAIC SURFACES FOR OUR ALGORITHM FOR A 512×512 WINDOW ON AN NVIDIA 280 GTX, WITHOUT SHADOW (**NS** COLUMNS) AND WITH SHADOW (**S** COLUMNS). THE ORDER OF ALGEBRAIC SURFACES APPEARS WITHIN SQUARE BRACKETS. THE STEPSIZE IS ALSO SHOWN.

Surface	Knoll et al. [6] 8800 GTX	AMP using Sign Test	
		8800 GTX	280 GTX
Steiner	38	233	487
Teardrop	121	195	408
Tangle	71	216	451
Barth Sextic	88	132	249
Kleine	101	187	350
Mitchell	60	194	363
Barth Decic	16	103	225
Superquadric	108	170	282

TABLE II

COMPARISON OF FRAME RATES FOR DIFFERENT SURFACES USING KNOLL'S METHOD AND THE AMP METHOD ON COMMON SURFACES FOR A 1024×1024 WINDOW ON THE SAME GPU NVIDIA 8800 GTX AND ON NVIDIA 280 GTX.

Surface [Order]	AMP using sign test	
	CPU	280 GTX
Ding Dong [3]	987	0.52
Nordstrands[4]	3160	1.05
Torus[4]	2674	0.60
Dervish[5]	4152	1.23
Kiss [5]	3647	0.77
Chmutov [6]	5371	0.77
Heart [6]	5027	1.19
Chmutov [8]	6853	0.95
Barth [10]	7969	1.75
Chmutov [18]	9562	2.90
Chmutov 50 (Non Algebraic)	27875	4.83

TABLE III

RENDERING TIMES IN MILLISECONDS FOR A RESOLUTION OF 512×512 ON THE CPU USING AMP WITH SIGN TEST

APPENDIX I
SURFACES: EQUATIONS AND SCREEN-SHOTS

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.
$T_n(x)$ can be defined *iteratively* as:

$$T_n(x)=\begin{cases}1 & \text{if } n=0\\ x & \text{if } n=1\\ 2xT_n(x)-T_{n-1}(x) & \text{otherwise}\end{cases}$$

$T_n(x)$ can be defined *sinusoidally* as:

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

A. 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.

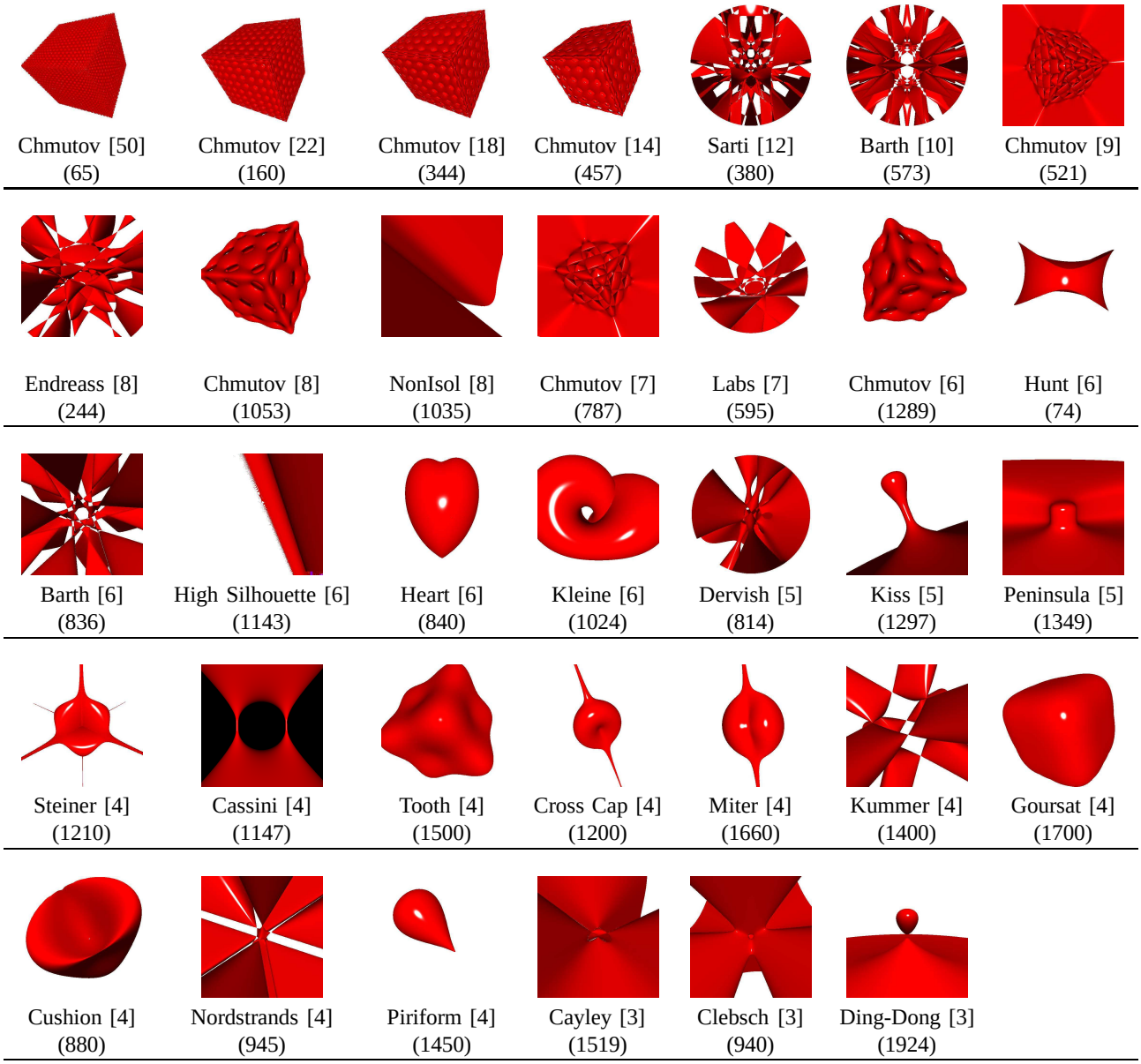

Fig. 10. Pictures of various algebraic surfaces with the order of the surface shown within square brackets and the FPS using the adaptive marching points algorithm shown within parenthesis for a 512×512 window.

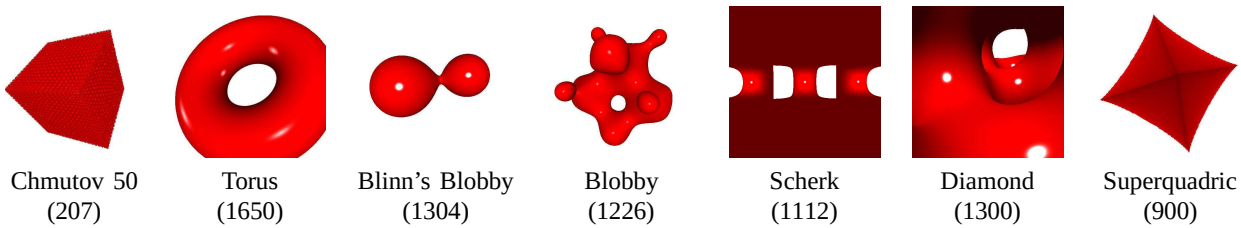

Fig. 11. Pictures of the non-algebraic surfaces rendered by us with the FPS using the adaptive point sampling algorithm given in parenthesis for a 512×512 window.

Jag Mohan Singh recieved his B.Tech (Honours) and MS by Research in Computer Science from the International Institute of Information Technology (IIIT), Hyderabad in 2005 and 2008 respectively. He was a member of Centre of Visual Information Technology (CVIT) from 2003 till 2008. His research interests include Computer Graphics and Computer Vision. He is currently a Research Associate at Technische Universitate, Darmstadt.

P J Narayanan is a Professor at the International Institute of Information Technology (IIIT), Hyderabad and heads the Centre for Visual Information Technology. He got his B.Tech from IIT, Kharagpur in 1984 and his PhD from the University of Maryland, College Park in 1992. He was a research faculty member at the Carnegie Mellon University from 1992-1996 where he worked on the Virtualized Reality project. He headed the Computer Vision and Virtual Reality group at the Centre for Artificial Intelligence and Robotics (CAIR), Bangalore from 1996 to 2000. He joined IIIT, Hyderabad in 2000 and is currently its Dean of Research & Development. Prof. Narayanan's research interests include Computer Vision, Computer Graphics, and GPU processing. He was made the first CUDA Fellow by Nvidia in November 2008.