Scientitfic Computing and Imaging Institute, University of Utah. Technical Report No UUSCI-2007-002

Interactive Ray Tracing of Arbitrary Implicit Functions

Aaron Knoll[*] — SCI Institute, University of Utah, IRTG
Younis Hijazi[†] — University of Kaiserslautern, IRTG
Charles Hansen[‡] — SCI Institute, University of Utah
Ingo Wald[§] — SCI Institute, University of Utah
Hans Hagen[¶] — University of Kaiserslautern

Abstract

We present a practical and efficient algorithm for interactively ray tracing arbitrary implicit surfaces. We use interval arithmetic both for reliable numerical computation and guaranteed detection of topological features. In conjunction with ray tracing, this allows for rendering literally any implicit surface simply from its definition. Interactive ray tracing facilitates flexible shading and visualization techniques, and allows dynamic rendering of higher-dimensional surfaces. Our method requires neither special hardware, nor preprocessing or storage of any data structure. Efficiency is achieved through SIMD optimization of both the interval arithmetic computation and coherent ray traversal algorithm, delivering interactive results even for complex implicit functions.

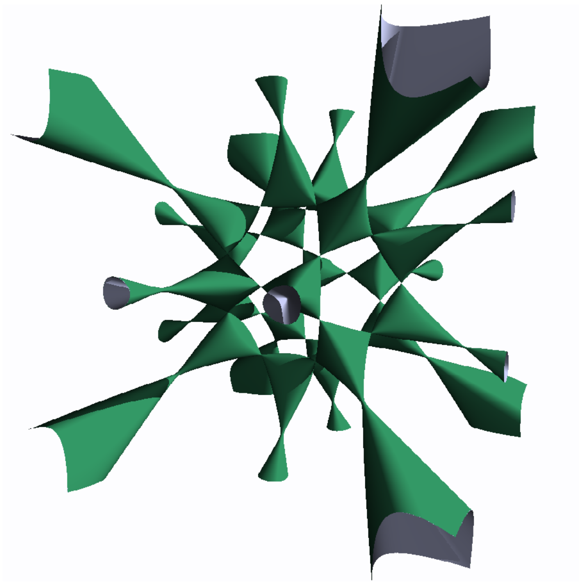


Figure 1: *The SuSE Linux 1995 Cover Manual Implicit* rendered roughly interactively at 9.0 fps (6.1 fps with shadows) with a 512^2 frame buffer on an Intel Core Duo 2.16 GHz, purely on the CPU. This image used $d_{stop} = 9$.

1 Introduction

Rendering implicit functions in 3D is not a new research area. In the past two decades, numerous techniques have been developed to ray trace arbitrary implicits robustly, with optimal convergence and minimal artifacts. At the same time, geometry processing methods have sought to improve surface extraction to handle non-differentiable and non-manifold surfaces, generating meshes that can be rendered trivially on GPU hardware. However, ray tracing methods have proven too slow to interactively render arbitrary implicits. While topologically intelligent extraction metrics exist, they often require numerous refinement iterations, and are impractical to pair with dynamic, real-time rendering.

Recently, coherent traversal techniques and SIMD optimizations have enabled interactive ray tracing. The most immediately practical applications have addressed rendering of large models and volumes, exploiting the logarithmic complexity and scalability of ray tracing as an alternative to out-of-core methods on the GPU. Nonetheless, despite the limits of GPU memory and bus speed, rasterization techniques excel at rendering both meshes and volume data. In contrast, unrefined implicit surfaces are geometries for which coherent ray tracing could potentially perform better than rasterization, in both feature-correctness and in rendering speed.

While implicits have not experienced as widespread adoption as parametric surfaces in graphics, they are common in other fields, such as mathematics, physics and biological modeling. These applications often do not demand the same real-time performance on large frame buffers as games, and are more interested in accurate representation of topological features such as singularities. However, interactivity allows better exploration of 3D shapes due to motion cues. Coherent ray tracing has not been applied to this problem before, and conventional ray tracing methods are slow largely due to the high computational cost of interval evaluation. By optimizing interval arithmetic with SSE, and pairing this with a fast coherent traversal algorithm, we find that interactive performance is possible on current laptop hardware, within a system that accurately visualizes any implicit surface that can be programmed.

The contribution of our work is the combination of a SIMD interval arithmetic library with a novel coherent ray tracing algorithm for implicits that performs coherent spatial bisection without the need for an explicit acceleration structure. Additionally, we obviate the need for user-defined partial derivative gradients, requiring only the implicit function and desired domain as inputs to our system. We will demonstrate our method on various implicits that are difficult for non-interactive or extraction-based systems, such as singularities and time-variant 4D hyper-surfaces.

2 Related work

2.1 Mesh Extraction

Naïve application of marching cubes [11, 23] on implicit functions can generate meshes interactively. However, topological features, particularly singularities, are easily lost. Paiva et al. [15] detailed a robust meshing algorithm based on dual marching cubes from an octree, using topological and geometric oracles. Other topologically-guided mesh extraction methods exist, e.g. Schreiner et al. [19], but have not specifically been evaluated on implicit functions with known thin regions and singularities. Both methods rely on iterative refinement of a mesh as an offline process.

[*]e-mail: knolla@sci.utah.edu
[†]e-mail: hijazi@informatik.uni-kl.de
[‡]e-mail: hansen@sci.utah.edu
[§]e-mail: wald@sci.utah.edu
[¶]e-mail: hagen@informatik.uni-kl.de

2.2 Ray Tracing

Much has been done in ray tracing of implicit functions; comparatively little of it is recent, and none except the piecewise GPU methods are interactive, even after accounting for Moore's Law.

2.2.1 Implicits

The blobby surfaces of Blinn [1] provided modeling interest in an efficient method of rendering implicits. Kalra & Barr [7] devised a class of L-G surfaces, which could be robustly isolated within a bounding region given a known Lipschitz-condition bound. Stolte & Caubet [20] applied discrete ray tracing to voxelized representations of implicits. Hart [5] proposed evaluating signed distance functions along a ray, considering balls of diminishing radi separating the ray and a surface. Recently on the GPU, Loop & Blinn [9] implemented an extremely fast ray caster by decomposing implicits into piecewise Bézier tetrahedra. Romeiro et al. [17] proposed a hybrid GPU/CPU technique for casting rays through CSG trees of implicits.

2.2.2 Interval Arithmetic

Without an explicit equation for ray-surface intersection, or knowledge of surface gradient properties, "point sampling" evaluation of the implicit cannot determine where a non-monotonic surface exists (Figure 2a). Mitchell [13] was the first to recognize that evaluating the implicit through interval arithmetic operations guarantees a convex hull around the function over a given domain, hence a correct method of isolating the surface when ray tracing. De Cusatis Junior et al. [3] used affine arithmetic on parallelepiped bounds for improved root isolation. Sanjuan-Estrada et al. [18] extended the Mitchell implementation with additional simplifying steps to traverse a disjoint set of intervals. Florez et al. [4] proposed a ray tracer that antialiases surfaces by adaptive sampling during interval subdivision.

2.2.3 Ray Coherence

The notion of a group of rays marching in a single direction is simple yet critical to the performance of coherent ray tracing systems. Coherent methods have delivered real-time performance for polygonal scenes [22, 16], and SIMD has been used in optimized intersection algorithms for trilinear voxel interpolant surfaces [12].

Our work was heavily inspired by optimized traversals for coherent SIMD ray tracing, particularly the frustum grid traversal proposed by Wald et al. [21], and the hierarchical extension of that algorithm to large octree volume data by Knoll et al. [8].

3 Background

3.1 Interval Arithmetic

Interval arithmetic (IA) reliably provides bounds for the global range of a function over a given domain, as opposed to simply evaluating the interval bounds as with point sampling. Thus, it correctly captures the behavior of a function, regardless of whether or not it is monotonic on a domain interval (Figure 2).

Interval arithmetic was introduced by R. E. Moore [14] as an approach to putting bounds on rounding errors in mathematical computation. The same way classical arithmetic operates on real numbers, interval arithmetic defines a set of operations on intervals. Let $X = [a,\ b]$ and $Y = [c,\ d]$ be intervals. Then, if $op \in \{+,-,*,/\}$, we define $X\ op\ Y = \{x\ op\ y \text{ where } x \in X \text{ and } y \in Y\}$. For example,

$$X + Y = [a,\ b] + [c,\ d] = [a + c,\ b + d]$$

$$X - Y = [a,\ b] - [c,\ d] = [a - d,\ b - c]$$

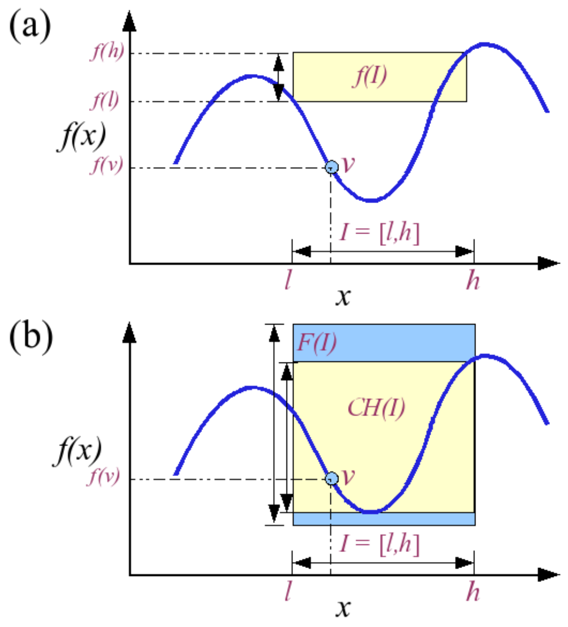


Figure 2: *Inclusion property of interval arithmetic.* (a) When a function is non-monotonic, simply evaluating the lower and upper components of a domain interval is insufficient to guarantee a convex hull over the range. This is not the case with interval arithmetic (b), which, when evaluated, will encompass all minima and maxima of the function within that interval. Thus, an IA representation F of a function f can definitively determine if f *possibly* passes through v on an interval I, by testing if $v \in F(I)$. Ideally, $F(I)$ is equal or close to the bounds of the convex hull, $CH(I)$.

$$X \times Y = [min(ac,ad,bc,bd), max(ac,ad,bc,bd)]$$

Then, for a function $f : \Omega \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$ (where Ω is an open subset of $\mathbb{R}^3$) and a box $B = X \times Y \times Z \subseteq \Omega$, we seek an *inclusion function* $F : B \rightarrow F(B)$ of f on intervals such that:

$$F(B) \supseteq f(B) = \{f(x,y,z) \mid (x,y,z) \in B\}$$

Moore's fundamental theorem of interval arithmetic [14] states that for any function f defined by an arithmetical expression, the according interval evaluation function F is an inclusion function of f. Effectively, it suffices to implement a library of these IA operators, and subsitute them for the real operators in producing an IA expression F.

An *implicit surface* S is defined as the set of solutions of an equation $f(x,y,z) = 0$, where $f : \Omega \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$. The strength of using interval arithmetic for evaluating an implicit surface S is that it provides a very simple and reliable rejection test for the box B not intersecting S,

$$0 \notin F(B) \Rightarrow 0 \notin f(B)$$

This property can be used in ray tracing for skipping empty space. The criterion proves that the box and the surface do not intersect, but its converse doesn't necessarily hold. Indeed, we only have an implication and not an equivalence: we can have $0 \in F(B)$ without B intersecting S, because of loose intervals. A well known aspect of IA, when not adequately used, is the overestimation problem which can happen with iterative function evaluations of increasingly large intervals. However, when interval domains are subdivided, as with implicit curve approximation [10] or intersections [6], IA will guarantee convergence to the true solution. In practice, subdivision usually requires a termination criterion, which depends on the application. Moreover, given sufficiently fine subdivision, techniques using IA will never miss features of the true surface.

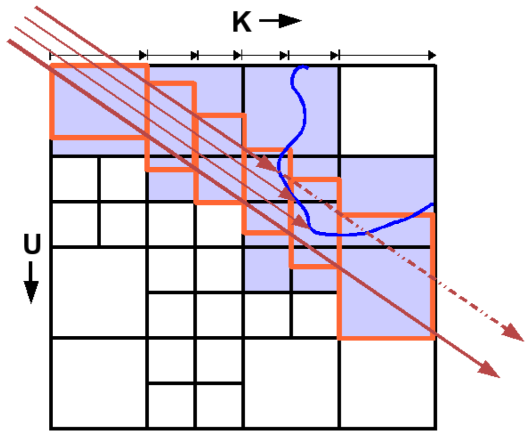


Figure 3: *An example of coherent traversal* using an octree as in [8], effectively the hierarchical extension of [21]. The packet is defined by a bounding frustum; nodes of an acceleration structure are queried when they contain the $\vec{U}$ (and $\vec{V}$ in 3D) extents of that frustum along an interval on $\vec{K}$. Marching from one slice to the next simply entails addition. Unlike acceleration structures, however, we do not explicitly store *any* data; we instead evaluate the IA expression of the implicit function.

3.2 Coherent Ray Tracing

The principal idea of coherent ray tracing is to perform traversal and intersection on groups, or *packets*, of rays. In this way, the costs associated with ray tracing are amortized over that group. Aggressive coherent methods often compute traversal steps over a bounding frustum of the packet as opposed to individual rays themselves, e.g. [21, 16]. While these methods can dramatically improve cache-coherence and performance with larger frame buffers, they can degenerate to single-ray performance or worse when the rays are incoherent or the scene is complex [8].

More conservative methods exploit coherence on a smaller scale, specifically when encouraged by hardware. SIMD instruction sets such as SSE effectively perform four floating point operations in parallel for the cost of one. By generating packets of 2x2 rays, traversal and intersection are performed on four rays for roughly the price of a single ray. While the potential gains are more modest, degenerate scenes and incoherent rays generally perform at least as well as a single-ray system, and code remains relatively simple.

In coherent acceleration structure traversal, *every* node touched by *any* ray in the packet must be explored. To perform the fewest number of traversal steps, rays should march in lockstep, ideally beginning and terminating traversal at the same time. A common device for traversing rectilinear space is choosing a major march direction, denoted by $\vec{K}$, and examining slices of the other dimensions along fixed $\vec{K}$ intervals [21, 8]. Such traversal is effectively a unidimensional algorithm, in which spatial marching can be accomplished iteratively with simple addition (Figure 3).

4 Coherent Ray Tracing of Implicits with Interval Arithmetic

In many ways, ray-implicit intersection with interval arithmetic more closely resembles acceleration structure traversal than explicit ray-primitive intersection. This similarity is even more pronounced in our coherent application, where again we seek to keep rays marching simultaneously through a spatial region. The difference is that we store no actual structure in memory, and instead evaluate an IA expression to determine if a region contains the implicit function.

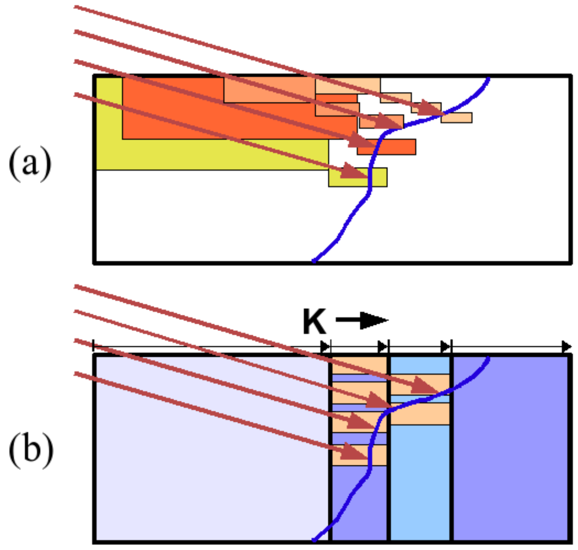


Figure 4: *Interval bisection methods for root isolation.* The conventional Mitchell [13] single-ray method (a) recursively bisects each ray until a surface is located to the satisfaction of a termination criterion. Unfortunately, even neighboring rays may diverge spatially in behavior, making this algorithm impractical for coherent implementation. Our spatial subdivision technique (b) remedies the problem by marching rays along a common axis in lockstep. Both methods are shown terminating after two subdivisions; after which the classic method has performed one more iteration than ours, while sampling nearly at twice the rate from one ray to another.

Existing IA implicit ray casters bisect rays into segments, as originally proposed by Mitchell [13]. Although neighboring rays are nearly identical, the fine subdivision of the interval search along with different entry and exit points causes them to fall out of lockstep. In coherent applications, adjacent rays using this method exhibit divergent behavior after even a few steps. When combined with packets, more traversal steps are performed than when rays march across common spatial regions concurrently. Finally, if a common depth is used to decide the sampling rate of our function, some rays will sample the image far more than others (Figure 4a).

Rather than bisecting rays, we bisect a single $\vec{X}, \vec{Y}$ or $\vec{Z}$ axis similar to the octree grid implementation of Knoll et al. [8]. Rays then march in the same world space region, and behave similarly with regards to their respective intervals (Figure 4b). This is equivalent to bisecting rays into equal segments from the $\vec{K}_{enter}$ to $\vec{K}_{exit}$ planes of the domain. Rather than subdivide and march t intervals, we march actual world space coordinates and use these as interval bounds for $F(X,Y,Z)$. This way, the input function need not be reparameterized, and costly IA multiplications for ray evaluation are avoided.

The process of evaluating intervals is then simple. Given an interval box B, our function f and its corresponding IA evaluation F, we evaluate $F(B)$ and finally check if $0 \in F(B)$. If it does, then we bisect this space along the major march axis, and register a hit when a maximum depth threshold is reached.

5 IMPLEMENTATION

As the contribution of this paper is largely algorithmic in nature, we include pseudocode for critical components of our implementation. We abbreviate the 4-vector packet floating point datatype as “simd”; specific code for SSE, Altivec or GPU hardware is left to the reader. Implicit functions were hardcoded in a header file; we envision inputs to our program would compile implicits into a dynamic library, as expression parsing makes for costly evaluation. Rather than employ C++ arithmetic operators, implicits call the associated SSE and IA library function calls, for example “mul4” for a SIMD multiplication, and “mul_i4” for a SIMD interval multiplication, as opposed to “*”. We maintain these conventions in this pseudocode.

Algorithm 1 SIMD Interval Arithmetic

```
struct interval4 {
  simd lo, hi;
};
interval4 add_i4(interval4 a, interval4 b) {
  return interval4( add4(a.lo, b.lo),
                    add4(a.hi, b.hi) );
}
interval4 mul_i4(interval4 a, interval4 b) {
  simd lolo = mul4(a.lo, b.lo);
  simd lohi = mul4(a.lo, b.hi);
  simd hilo = mul4(a.hi, b.lo);
  simd hihi = mul4(a.hi, b.hi);
  return interval4( min4(lolo, min4(lohi,
                       min4(hilo, hihi))),
                    max4(lolo, max4(lohi,
                       max4(hilo, hihi))) );
}
```

5.1 SSE Interval Arithmetic

The foundation of our implicit ray tracing system is the SSE IA library, which allows us to quickly evaluate intervals in SIMD. Implementation is straightforward; interval multiplication is particularly efficient as SSE itself is relatively fast for both multiplication and minimum/maximum operation. Transcendental functions such as sine are somewhat tricky; these require modulus of the domain over $[0, 2\pi]$, and an SSE implementation of the function. Some pseudocode examples are given in Algorithm 1.

5.2 Ray Packet Structure

We chose conservative 2x2 packets for our implementation. Above all, we wish to evaluate baseline performance with optimized ray tracing. Though not as potentially fast as wide packet coherent methods, 2x2 packets are more resistant to divergent ray behavior, and encourage simple algorithms that can more easily be ported to the GPU. In this ray tracing architecture, origin and direction are stored for each $\vec{X}, \vec{Y}, \vec{Z}$ axis in SSE packed floats. We also store the t parameter of the rays, and a mask indicating which rays have hit. This structure is detailed in Algorithm 2. Ray generation employs a simple pinhole camera; this optimizes our algorithm as all rays possess a constant origin.

Algorithm 2 Ray Packet Structure

```
struct RayPacket {
  simd org[3];
  simd dir[3];
  simd inv_dir[3];
  simd t_hit;
  simd p_hit[3];
  simd normal[3];
  simd hitmask;
};
```

5.3 Traversal

Once the user has supplied a function, a domain on $\mathbb{R}^3$, and a maximum depth d_{stop}, we are ready to perform traversal. In our implementation, d_{stop} determines a global precision for rendering the implicit; this is preferable in evaluating image quality and performance at various uniform sampling rates, though view-dependent adaptive subdivision could be desirable in the future. A crucial aspect of our method is that though rays traverse $\vec{K}$ together, they march across independent – and thus tight – intervals. This principle is illustrated in Figure 4b, and pseudocode for traversal is detailed in Appendix A.

As in coherent grid traversal [21], we first find $\vec{K}$, the dominant axis of the first ray in the packet, and denote the remaining two axes $\vec{U}$ and $\vec{V}$. We then perform a standard ray bounding-box test on our domain. We store the actual t_{exit} and t_{enter} parameters as well as the intersections with the $\vec{K}$ entry and exit planes, t_{Kenter} and t_{Kexit}. Now, we consider the total increment along $\vec{K}$, $t_{Kexit} - t_{Kenter}$, and compute the total $\vec{U}$ and $\vec{V}$ increments over the entire domain. As our implementation is iterative, not recursive, we store an array containing a traversal “stack” for each depth $\{0..d_{stop} - 1\}$, containing the t, $\vec{K}, \vec{U}$ and $\vec{V}$ increments bisected at each level.

The algorithm then simply marches from one $\vec{K}$ slice to the next, incrementing the t, $\vec{K}, \vec{U}$ and $\vec{V}$ positions once per step and keeping track of current and next values, orthogonally for each ray using SSE. It constructs intervals from the $\vec{K}, \vec{U}$ and $\vec{V}$ current and next values. This enables us to iteratively increment domain intervals simply with three SIMD additions, as opposed to three IA SIMD multiplications and additions using the Mitchell t-marching method. Branching is only used to omit intervals when $t < t_{enter}$, and exit when all rays hit successfully or have $t > t_{exit}$. We store and check a flag for each depth, which indicates when both sides of a $\vec{K}$-subtree have been traversed. When this happens, we decrement the depth, and exit traversal when $depth == -1$.

At each march iteration, we evaluate the IA function expression on this domain interval. If $0 \in F(x, y, z)$, we “recurse” by incrementing d and using the bisected increments one level deeper. We register a hit on the surface when $d == d_{stop} - 1$, and mask rays that successfully hit.

5.4 Handling Division

Traversing with such naïve IA evaluation will robustly handle any implicit with adequate sampling, with the exception of functions performing division. In theory, IA division by intervals containing zero is ill-defined, similar to division of real numbers by zero. Fortunately, we can easily detect and handle these cases. For two intervals A and B, when $0 \in B$, we define $A/B = [-\infty, \infty]$. We then approximate ∞ with a large number, such as FLT_MAX/2. When rays traverse these intervals, they will *always* find a surface within and recurse to maximum depth. Thus, without modification to the traversal, asymptotes will rendered. The final implementation detail of traversal is choosing not to render asymptotes by neglecting to register a hit when $F_{hi} - F_{lo} = \infty$. This principle is illustrated in Figure 5.

With division correctly handled, our IA library and renderer will work for any function whose components are implemented in SSE.

5.5 Shading

5.5.1 Central Differences

To compute the normal at a hit position, a renderer requires the x, y, z partial derivatives of the implicit at that point. While analytical derivatives can be manually defined, they are not strictly necessary. If the user fails to define partials, we employ central differences by evaluating our function (using SSE, not SSE IA evaluation) six times to create a central differences stencil. The results look excellent in most cases, and have no appreciable impact on performance.

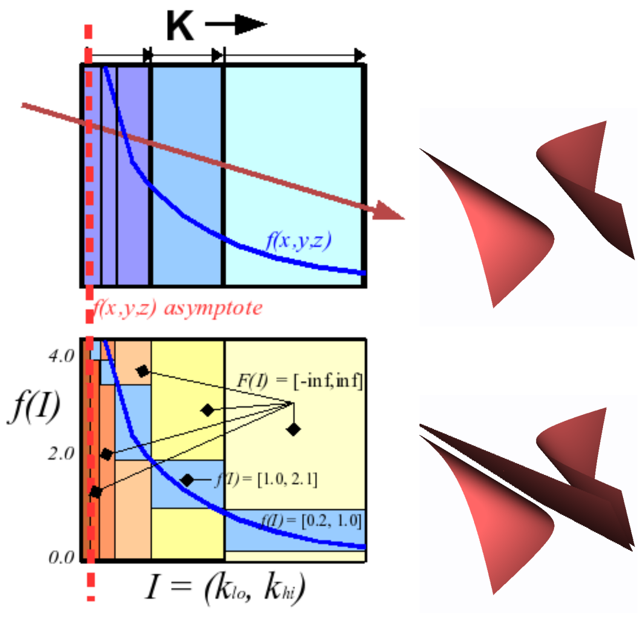


Figure 5: *Handling Asymptotes.* For functions with division, our IA implementation returns "infinite" intervals. We may detect this situation within the traverser before registering a hit, allowing us to visualize asymptotes if we choose. With division handled, we can robustly handle any function.

5.5.2 Shadows

In ray tracing, shadows are fairly trivial, requiring a shadow ray cast for every primary camera ray that hits a surface. This typically entails a 20% to 40% decrease in frame rate, depending on the coherent behavior of shadow rays. Fortunately, shadow rays require less accuracy than primary rays; in our application it suffices to cast shadow to a coarse termination depth, such as $d_{stop} = 8$, while employing a higher depth for primary rays. As shadows are primary useful as depth cues, this is generally acceptable. The performance penalty is reduced, and loss of shadow detail is seldom perceptible (Figures 1, 6, and 7).

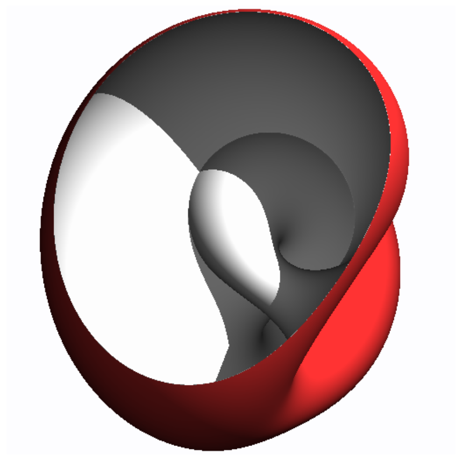


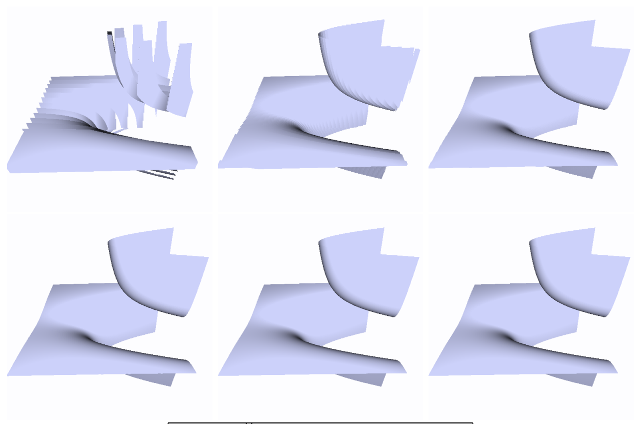
Figure 6: *Cut-away of a Klein bottle.* We can color-code front and opposing sides of an implicit by evaluating whether F is positive or negative at the hit position; this aids in navigating oddly-connected manifolds. In addition, dynamic shadows aid in depth perception.

6 Results

6.1 General Performance

Figure 9 shows various implicit surfaces with their associated equations and performance. Performance is suprisingly good for this brute-force technique; on an Intel Core Duo 2.16 GHz test machine, our method achieves over 20 frames per second for simple objects such as the torus, sphere and conic sections. For more complex objects, such as the Mitchell or Steiner surfaces, performance can fall below interactive speeds on the evaluation hardware, but it still permits exploration around 1-5 fps even for the worst cases. Complicated expressions such as the SuSE logo also exhibit similar performance.

As previously mentioned, we are not restricted to any particular class of surfaces. Indeed, we are able to render any sort of implicitly defined surface, including asymptotic, non-differentiable, non-continuous, non-manifold, self-intersecting and linked implicits. These are shown in the bottom rows of Figure 9.



d_{stop}	**4**	**6**	**8**
FPS	32.0	25.7	20.3
d_{stop}	**10**	**12**	**16**
FPS	15.5	11.6	6.85

Table 1: *Quality at various d_{stop} terminarion depths*, shown in order from top left to bottom right above. As expected, performance is linear with depth, hence logarithmic with object complexity measured in number of interval subdivisions in the scene. At depth 8 the surface becomes recognizable, and depth 10 generally suffices to capture features.

6.2 Quality

We find that $d_{stop} = 8$ is sufficient for capturing general topology, and $d_{stop} = 12$ achieves excellent spatial sampling at close camera views. In practice, $d_{stop} = 10$ is a good balance of performance and feature reproduction (Table 1).

A close-up of the tear drop demonstrates how our algorithm can reproduce fine details that mesh-based approaches often omit (Figure 7). Meshing even using topologically intelligent methods (e.g. Paiva et al. [15]), frequently fails to capture such regions of a surface, leading to misclassification of details such as asymptotes or singularities. It is worth mentioning that such fine regions are best reproduced at shallower d_{stop}; at greater termination depths the surface region may become narrower than the spacing between the rays sampling it. To consistently reproduce such fine features, adaptive sampling or antialiasing would be desirable, but our naïve technique is sufficient to identify them.

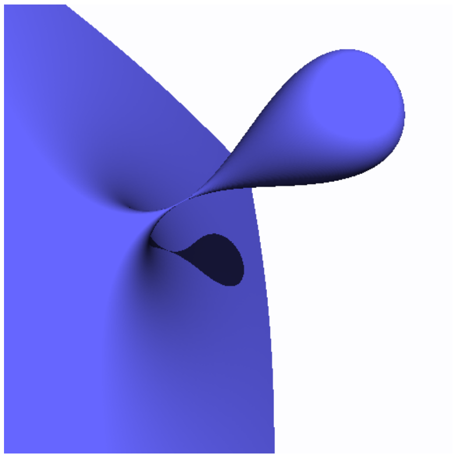


Figure 7: *The tear drop.* Though difficult even for intelligent mesh extraction methods, ray tracing successfully reproduces fine details and the correct connectivity of this surface. We used $d_{stop} = 10$.

6.3 Dynamic Scenes

Because we neither precompute an explicit representation of the object, nor a physical acceleration structure in memory, we have great flexibility in manipulating and rendering implicit functions over time. Moreover, we are not restricted to static 3D objects; implicits are theoretically capable of representing N-dimensional objects. In practice, we can render 4D implicits as 3D over time, using a $f(x,y,z,w)$ expression. An example of a two-sheeted hyperboloid morphing into a torus is shown in Figure 8.

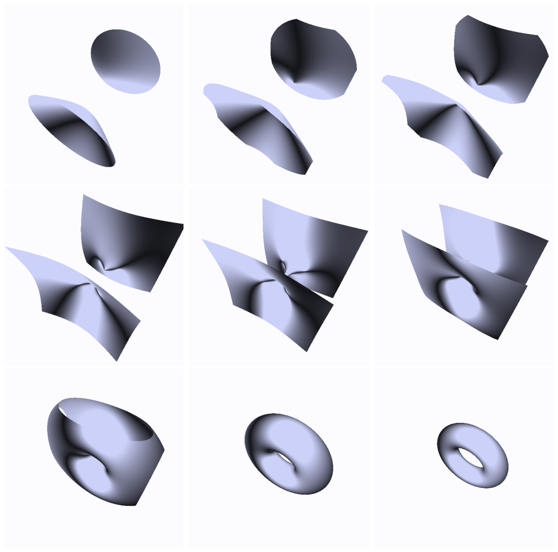


Figure 8: *Animated 4D implicits.* As our algorithm does not compute or store any acceleration structure, we can make arbitrary changes to the implicit function on the fly. In this example, we interactively morph a hyperboloid into a torus.

One of the simplest 4D implicit objects is the hyper-sphere, $f(x,y,z,w) = x^2 + y^2 + z^2 + w^2 - r^2$, whose resulting scene would be an animated sphere which suddenly appears, smoothly increases, then decreases in size, and finally disappears. Despite being easily obtained with our method, these dynamic scenes would be much harder to achieve using topologically-guided meshing techniques.

6.4 Comparison to Existing Techniques

It is difficult to assess the performance of prior works in implicit ray tracing. Most related work is dated, making performance comparisons difficult to interpret. Fortunately, nearly all papers in this area evaluate performance with a sphere, though at varying quality and image resolution. Perhaps the first rigorous benchmark, with source code available for the IA and affine arithmetic libraries, was the work of De Cusatis Junior et al. [3], which reported rendering the sphere at around 1.3 fps at 64x64 on a Pentium 166. Even accounting for Moore's Law, we achieve between two and three orders of magnitude better performance. Similarly, the Sanjuan-Estrada [18] implementation on top of POV-Ray is likely two orders of magnitude slower than our method. The most recent paper by Florez et al. [4] rendered a sphere in 40 seconds at 300x300 resolution on a P4 2.4 GHz, albeit with adaptive anti-aliasing; our method delivers well over two orders of magnitude improvement in frame rate (Figure 9).

7 Conclusion

We have detailed a fast coherent ray tracing technique for rendering arbitrary implicit functions. By combining coherent acceleration structure algorithms with a simple SSE interval arithmetic library, we are able to render implicits on the CPU, at reasonable quality, and over two orders of magnitude faster than previously published results.

The original goal of this work was to provide a general, accurate and reasonably interactive method of viewing implicits. A secondary motivation was to exploit two strengths of CPU ray tracing, SSE instructions and programming flexibility, that allowed implementation of a general-purpose SSE IA library. Overall, our results demonstrate success on both fronts. The ability to dynamically render arbitrary implicits has broad implications. One reason why implicit surfaces, compared to parametrics, have been largely unpopular in graphics has been their comparative difficulty in rendering. With this obstacle reduced, the modeling community may renew interest in experimenting with implicit forms. In ray tracing environments, implicits could be used to model physical deformations, perhaps replacing subdivision surfaces in certain instances. Finally, in visualization, higher-order implicit isosurfaces can more easily be rendered using this technique on a variety of data, from point-based to volumetric. Logarithmic complexity allows ray tracing to decompose complex scenes easily; with the appropriate acceleration structure, scenes containing many piecewise implicits may require only modestly more computation than scenes rendering one implicit function.

Possibilities abound for future work. Performance could potentially be further improved by using larger packets and more aggressive coherence-exploitation techniques, which have proven so successful for polygonal scenes. Adaptive antialiasing (e.g. Florez et al. [4]) might be desirable for better image quality and reproduction of fine features. An adaptive termination criterion in the spirit of [8] or [2] instead of a fixed (high) subdivision depth could also further boost performance. Moreover, our algorithm is non-recursive and elegantly simple. Given shader programs implementing both an interval arithmetic library and the desired implicits, a GPU implementation of our traversal would be feasible and likely fast. The Cell Broadband Architecture seems particularly attractive for this algorithm, as its threaded SIMD hardware model is similar to multicore CPU's and the memory footprint of our technique is negligible.

Of related interest would be applying these SIMD IA techniques to rendering arbitrary parametric surfaces, which pose many of the same challenges to ray tracers as implicits. Integrating implicits into full featured ray tracing software might be desirable, given relatively coherent applications and support for SIMD packets. The ability to evaluate the implicit directly at shading time enables a myriad of mathematical visualization techniques that could be explored further: shading topological features, rendering higher-dimensional surfaces, constructive solid geometry methods, and volume rendering of 3-manifolds and their features. Having an interactive system for rendering complicated implicits enables future exploration of these functions.

REFERENCES

[1] James Blinn. A Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[2] Holger Dammertz and Alexander Keller. Improving Ray Tracing Precision by World Space Intersection Computation. In *Proceedings of IEEE Symposium of Interactive Ray Tracing*, 2006.

[3] A. de Cusatis Junior, L. de Figueiredo, and M. Gattas. Interval methods for raycasting implicit surfaces with affine arithmetic. In *Proceedings of XII SIBGRPHI*, pages 1–7, 1999.

[4] J. Florez, M. Sbert, M.A. Sainz, and J. Vehi. Improving the interval ray tracing of implicit surfaces. In *Lecture Notes in Computer Science*, volume 4035, pages 655–664, 2006.

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

[6] Y. O. Hijazi and T. M. Breuel. Sweeping Arrangements using Interval Arithmetic. Technical report, IRTG, University of Kaiserslautern, 2006. (to be published).

[7] D. Kalra and A. H. Barr. Guaranteed ray intersections with implicit surfaces. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 297–306, New York, NY, USA, 1989. ACM Press.

[8] Aaron Knoll, Charles Hansen, and Ingo Wald. Coherent Multiresolution Isosurface Ray Tracing. Technical Report UUSCI-2007-001, SCI Institute, University of Utah, 2007. (submitted for publication).

[9] Charles Loop and Jim Blinn. Real-time GPU rendering of piecewise algebraic surfaces. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 664–670, New York, NY, USA, 2006. ACM Press.

[10] H. Lopes, J. Oliveira, and L. de Figueiredo. Robust adaptive approximation of implicit curves. In *SIBGRAPI 2001*, 2001.

[11] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 21(4):163–169, 1987.

[12] Gerd Marmitt, Heiko Friedrich, Andreas Kleer, Ingo Wald, and Philipp Slusallek. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, pages 429–435, 2004.

[13] Don Mitchell. Robust ray intersection with interval arithmetic. In *Proceedings on Graphics Interface 1990*, pages 68–74, 1990.

[14] R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[15] Afonso Paiva, Hlio Lopes, Thomas Lewiner, and Luiz Henrique de Figueiredo. Robust adaptive meshes for implicit surfaces. In *19th Brazilian Symposium on Computer Graphics and Image Processing*, pages 205–212, 2006.

[16] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics*, 24(3):1176–1185, 2005. (Proceedings of ACM SIGGRAPH).

[17] Fabiano Romeiro, Luiz Velho, and Luiz Henrique de Figueiredo. Hardware-assisted Rendering of CSG Models. In *SIBGRAPI*, pages 139–146, 2006.

[18] J. F. Sanjuan-Estrada, L. G. Casado, and I. Garcia. Reliable algorithms for ray intersection in computer graphics based on interval arithmetic. In *XVI Brazilian Symposium on Computer Graphics and Image Processing, 2003. SIBGRAPI 2003.*, pages 35–42, 2003.

[19] John Schreiner, Carlos Scheidegger, and Claudio Silva. High-Quality Extraction of Isosurfaces from Regular and Irregular Grids. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1205–1212, 2006. Proceedings of IEEE Visualization 2006.

[20] Nilo Stolte and Rene Caubet. Fast High Definition Discrete Ray Tracing Implicit Surfaces. In *5th DGCI - Discrete Geometry for Computer Imagery, pages 61-70, Clermont-Ferrand, Universite d'Auvergne.*, 1995.

[21] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, pages 485–493, 2006. (Proceedings of ACM SIGGRAPH).

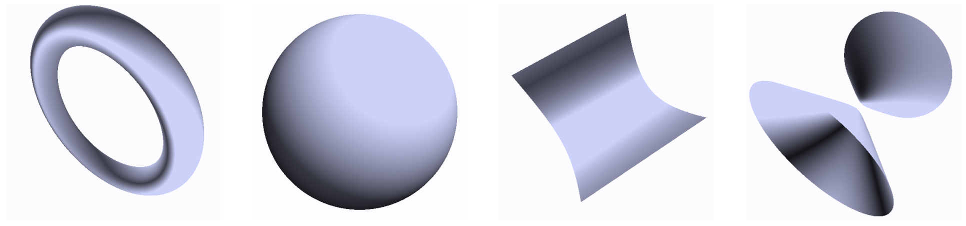
[22] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[23] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2:227–234, 1986.

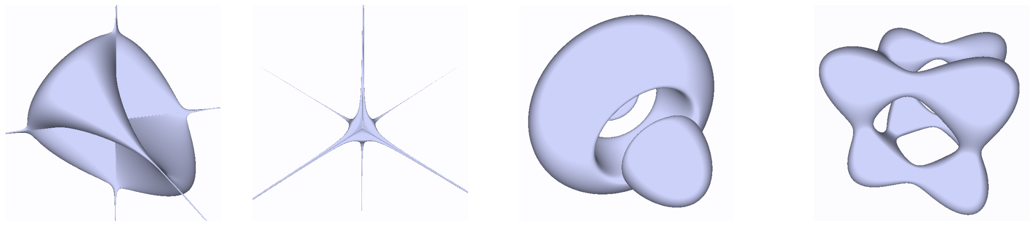
A TRAVERSAL PSEUDOCODE

Algorithm 3 Ray-Implicit Traversal. We assume basic familiarity with binary operations on SIMD instructions, and particularly boolean operations such as comparisons, and the concept of masking.

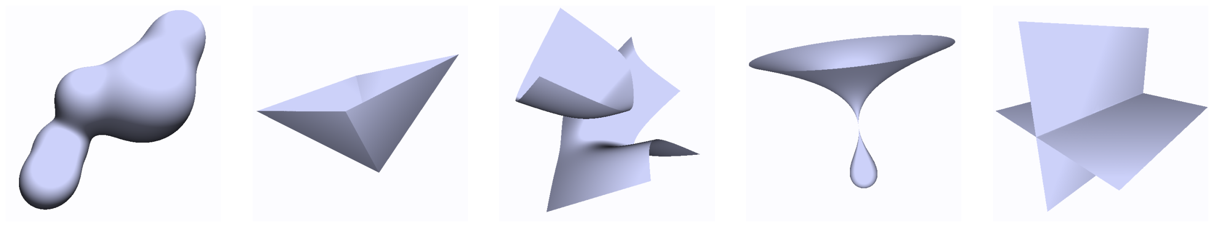
```
template<int K, int U, int V, int DK>
void traverse(RayPacket r, Box domain,
              Implicit implicit, int d_stop) {
  (get t_enter, t_exit, t_kenter, t_kexit)
  simd validmask = intersectBB(r, domain);
  //validmask indicates rays that are active
  float full_tk = tk_exit - tk_enter;
  float full_u = mul4(r.dir[U], full_tk);
  float full_v = mul4(r.dir[V], full_tk);
  struct Stack {
    simd t_incr;
    simd u_incr, v_incr;
    float k_incr;
    char side;
  };
  Stack stk[maxDepth];
  for(int d=0;d<maxDepth;d++){
    float width = 1.f / (float)(1<<d);
    stk[d].t_incr = mul4(full_tk, width);
    stk[d].u_incr = mul4(full_u, width);
    stk[d].v_incr = mul4(full_v, width);
    stk[d].side = -1;
  }
  int depth = 0;
  float curr_k = DK==+1 ? domain.min[K]:domain.max[k];
  simd curr_t, curr_u, curr_v;
  curr_t = t_kenter;
  curr_u = add4(r.org[U],mul4(r.dir[U],curr_t));
  curr_v = add4(r.org[V],mul4(r.dir[V],curr_t));
  simd next_t, next_u, next_v;

  for(;;) {
    stk[depth].side++;
    next_k = DK==+1 ?
             curr_k + stk[depth].k_increment :
             curr_k - stk[depth].k_increment;
    next_u = add4(curr_u, stk[depth].u_increment);
    next_v = add4(curr_v, stk[depth].v_increment);
    next_t = add4(curr_t, stk[depth].t_increment);
    hitmask = and4(validmask, cmp_ge4(next_t, tenter));
    if (any4(simd_hitmask)) {
      interval4 ibox;
      (fill ibox with curr and next k,u,v)
      interval4 F = implicit.evalute_interval4(ibox);
      if (any4(F.contains(0))) {
        if (!all4(cmp_ge4(sub4(F.hi,F.lo),INFINITY))){
          if (depth == maxDepth-1){
            //hit
            hit(r, curr_t);
            (compute normal);
            if (all4(r.hitmask))
              return;
          } else {
            //recurse
            depth++;
            continue;
          }
        }
      }
    }
    validmask = and4(validmask, cmp_le4(next_t, texit));
    if (none4(validmask))
      return;
    curr_k = next_k;
    curr_t = next_t;
    curr_u = next_u;
    curr_v = next_v;
    if (stk[depth].side & 1)
    {
      do{
        if (depth-- == -1)
          return; }
      while(stk[depth] & 1);
      continue;
    }
  }
}
```



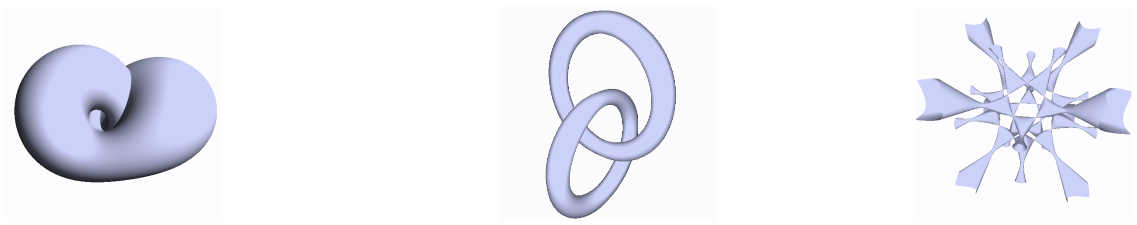
Torus	Sphere	Parabolic Cylinder	Hyperboloid
$(r_i - \sqrt{x^2+y^2})^2 + z^2 - r_o^2$	$x^2+y^2+z^2-r^2$	x^2-y-r^2	$-\frac{x^2}{a^2} - \frac{y^2}{a^2} + \frac{z^2}{c^2} - 1$
22.9 fps	20.5 fps	34.1 fps	19.2 fps



Steiner 1	Steiner 2	Mitchell	Tangle
$xy+yz+xz+xyz$	$(x^2y^2+y^2z^2+x^2z^2)^2+xyz$	$4(x^4+(y^2+z^2)^2)+17x^2(y^2+z^2)$ $-20(x^2+y^2+z^2)+17$	$x^4-5x^2+y^4-5y^2+z^4-5z^2+11.8$
6.1 fps	17.2 fps	5.9 fps	3.8 fps



Blobby	Absolute value	Inverse function	Teardrop	Intersecting planes
$\sum_{i=1}^{N} \frac{r_i^2}{\lVert x-p_i\rVert^2} - 1$	$\lvert x\rvert + \lvert y\rvert - z$	$\frac{1}{x-y^2} - z$	$0.5x^5+0.5yx^4-y^2-z^2$	xy
4.7 fps	30.8 fps	20.8 fps	15.3 fps	33.0 fps



Klein Bottle	Linked tori	SuSE95
$(x^2+y^2+z^2+2y-1)((x^2+y^2+z^2-2y-1)^2-8z^2)$ $+16xz(x^2+y^2+z^2-2y-1)$	$g(10x,10y-2,10z,13)g(10z,10y+2,10x,13)+1000$ $g(x,y,z,c)=(x^2+y^2+z^2+c)^2-53(x^2+y^2)$	$4(\tau^2x^2-y^2)(\tau^2y^2-z^2)(\tau^2z^2-x^2)$ $-(1+2\tau)(x^2+y^2+z^2-1)^2$
6.0 fps	4.0 fps	4.9 fps

Figure 9: *Selected implicit functions*, covering a wide range of different shapes and topologies. All examples are rendered at $d_{stop} = 10$ at 512^2 frame buffer resolution, on an Intel Core Duo 2.16 GHz. Performance is largely dependent on the number of operations required to evaluate the implicit, the entailed cost of computing the associated IA expressions, and the spatial complexity (effectively, implicit surface area) of the scene. SuSE95 was rendered using $\tau = \frac{1+\sqrt{5}}{2}$.