Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic

Aaron Knoll∗
SCI Institute, University of Utah & IRTG

Younis Hijazi†
University of Kaiserslautern & IRTG

Charles Hansen‡
SCI Institute, University of Utah

Ingo Wald§
Intel Corporation & 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 (IA) both for robust root computation and guaranteed detection of topological features. In conjunction with ray tracing, this allows for rendering literally any programmable implicit function simply from its definition. 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.

Index Terms: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1 INTRODUCTION

In graphics, geometry is most often modeled explicitly as a piecewise-linear mesh. An alternative is a higher-order analytical representation in implicit or parametric form. This option presents advantages, such as compact storage and view-independent local smoothness. While implicits have not experienced as widespread adoption as parametric surfaces in 3D modeling, they are common in other fields, such as mathematics, physics and biology. Moreover, they serve as geometric primitives for isosurface visualization of point sets and volume data.

To render implicits in 3D, one is principally given a choice of extracting and rasterizing a mesh, or ray tracing the surface directly via root-solving. Mesh extraction methods that adaptively reconstruct geometric or topological features exist; however they remain limited in the features they can reproduce, and are not sufficiently fast for dynamic extraction alongside real-time rasterization. While ray tracing low-order implicits is often trivial, arbitrary implicits pose a difficult problem. In the past two decades, several techniques have been developed to ray trace general implicits robustly. Overall, these methods either are slow, restrict the class of functions they handle, or resort to piecewise approximations. Methods involving interval arithmetic (IA) are the most general in that they can accommodate any programmable function. As implemented, however, they are among the least efficient.

Recently, coherent traversal techniques, SIMD vector instructions and multicore CPUs have enabled interactive ray tracing. Applications have largely sought to compete with rasterization in rendering explicit geometries – principally offering scalability to large data, and more powerful, flexible and intuitive shading and lighting models. As geometries that cannot be trivially rasterized, arbitrary implicits make a particularly intriguing application for ray tracing. 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 common laptop hardware, with a system that accurately visualizes any implicit surface composable by interval algebra.

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. We require no special hardware, other than SIMD vector instructions prevalent on all modern CPUs. To render, we require only the implicit function itself, a desired graphing domain, and an appropriate precision criterion or tolerance. We demonstrate our method on various implicits, including difficult cases for extraction-based methods, such as functions with singularities and time-variant 4D hyper-surfaces.

2 RELATED WORK

2.1 Mesh Extraction

Naïve application of marching cubes [14, 28] on implicit functions can generate meshes interactively. However, topological features, particularly singularities, are easily lost. Paiva et al. [20] de-
tailed 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. [24], 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.

2.2 Ray Tracing Implicits


2.3 Ray Tracing Implicits with Interval Arithmetic

Mitchell [16] was the first to employ interval arithmetic for implicit ray tracing. He devised a hybrid algorithm that employed bisection to segment the ray into intervals on which the function is monotonic, followed by root refinement via a standard numerical root-finding method. Capriani et al. [2] combined interval bisection with various other iterative schemes, including the Interval Newton method. De Cusatis Junior et al. [3] used affine arithmetic, a higher-order interval algebra, to address the bound overestimation problem of pure interval arithmetic (see Section 3.2). Sanjuan-Estrada et al. [23] compared performance of two hybrid interval methods with implementations of the Interval Newton and a recursive point-sampling subdivision method in the POV-Ray framework. Florez et al. [5] proposed a ray tracer that antialiases surfaces by adaptive sampling during interval subdivision. Even when accounting generously for Moore’s Law, none of these methods would perform interactively on a modern PC if implemented naïvely.

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, \( CH(I) \).

3 BACKGROUND

3.1 Implicit Functions

An implicit surface \( S \) in 3D is defined as the set of solutions of an equation

\[
f(x,y,z) = 0
\]

where \( f : \Omega \subseteq \mathbb{R}^3 \rightarrow \mathbb{R} \). For our purposes, assume this function is defined by any analytical expression. In ray tracing, we seek the intersection of a ray

\[
P(t) = \hat{O} + t\hat{D}
\]

with this surface \( S \). By simple substitution of these position coordinates, we derive a unidimensional expression

\[
f_t(t) = f(O_x + tD_x, O_y + tD_y, O_z + tD_z)
\]

and solve where \( f_t(t) = 0 \) for the smallest \( t > 0 \).

In this sense, ray tracing is a root-finding problem. For simple implicits such as a plane or sphere, \( f_t = 0 \) can be solved for \( t \) trivially. More complicated expressions, such as polynomials of degree 3 or higher, cannot be solved analytically.

Global iterative finding methods such as regula falsi can solve over an interval on which a root is known to exist, but fail otherwise. Recursive examination of sign changes, in conjunction with evaluation, work only when a function is monotonic over an interval. Such “point-sampling” methods (e.g. Kalra & Barr [10]) succeed when monotonicity assumptions can be made; otherwise they may fail to robustly determine zeros of the implicit, as illustrated in Figure 2(a). Fortunately, interval arithmetic provides us with a mechanism for testing whether or not a zero of a function exists over a sub-domain of the implicit.

3.2 Interval Arithmetic

Interval arithmetic was introduced by R. E. Moore [17] as an approach to bounding numerical 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 \( \oplus \in \{+, -, \times, /\} \), we define \( X \oplus Y = \{x \oplus y \mid x \in X \text{ and } y \in Y\} \). For example,

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

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

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

Moore’s fundamental theorem of interval arithmetic [17] states that for any function \( f : \Omega \subseteq \mathbb{R}^3 \rightarrow \mathbb{R} \) (where \( \Omega \) is an open subset of \( \mathbb{R}^3 \)) and a domain box \( B = X \times Y \times Z \subseteq \Omega \) the corresponding interval extension \( F : B \rightarrow F(B) \) is an inclusion function of \( f \), in that

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

Thus, by using interval arithmetic to evaluate \( F \), we have 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 identifying and skipping empty regions of space. Note, however, that although \( 0 \notin F(B) \) guarantees the absence of a root on an interval \( B \), that the converse does not necessarily hold: one can have \( 0 \in F(B) \) without \( B \) intersecting \( S \). When \( F(B) \) loosely bounds the convex hull, as in Figure 2(b), IA makes for a poor (though still reliable) rejection test. This overestimation problem is a well-known disadvantage, and is fatal to algorithms relying on iterative evaluation of non-diminishing intervals.
Fortunately, overestimation error is proportional to domain interval width; therefore IA guarantees convergence to the correct solution when interval domains diminish. This is the case in many algorithms involving implicit curve approximation [13], intersection of hierarchically subdivided spatial domains [4, 9], and ray tracing algorithms involving recursive interval bisection [16, 2]. Though the overestimation problem affects the efficiency of these algorithms, recursive IA methods robustly detect the zeros of an implicit, given an adequate termination criterion such as a sufficiently small precision $\varepsilon$ over the domain, or tolerance $\delta$ over the range.

As explained by Mitchell [15], any function can be expressed as an interval extension by considering its disjoint composition of piecewise-monotonic intervals. This includes non-algebraic piecewise- or periodic functions such as modulus, and transcendental such as exponential, logarithm and trigonometric functions [4]. While rigorous definition of the class of IA-expressible functions falls outside the scope of our work, intuitively one can derive an IA extension for any computable function. Once defined, IA operators are composable, allowing for trivial representation of arbitrary functions by their component real-operators. Ill-defined operations (e.g. division by zero, in Section 5.4), may require special-case handling, but are typically consistent with existing numerical solutions for real numbers.

### 3.3 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. [26, 21]. More conservative methods (e.g. [27]) exploit coherence on a smaller scale, specifically when encouraged by hardware. SIMD instruction sets such as SSE perform four floating point operations in parallel, encouraging operations on packets of four rays. While potential gains are more modest, rays with divergent behaviors may still benefit from instruction-level parallelism.

Coherent ray tracing performs best when rays in a packet behave similarly. Ideally, neighboring rays march in lockstep, requiring the fewest total traversal steps to examine a region of space. In the Wald et al. [26] coherent grid traversal (CGT) algorithm, coherent traversal of rectilinear space is accomplished by choosing a major march axis $K$ corresponding to the dominant ray direction, and examining slices of the other dimensions along fixed $K$ intervals. A hierarchical octree extension of CGT was proposed by Knoll et al. [11], and is the major algorithmic inspiration for this work.

### 4 Coherent Ray Tracing of Implicits with IA

Our algorithm simplifies the interval bisection method first proposed by Mitchell [16], and employs a variant of coherent octree traversal [11] as opposed to direct bisection of $t$ intervals along the ray. Together, these decisions allow us to perform bisection in a non-recursive manner, evaluate intervals quickly using SIMD vector instructions, and avoid unnecessary per-step interval multiplication. The simplicity and efficiency of this algorithm allow it to interactively visualize most implicit functions.

The conventional Mitchell algorithm [16] employs interval bisection to reject empty (rootless) intervals. For each nonempty interval, it then computes the gradient interval, and determines whether $0 \notin F(T)$, i.e. if the function is monotonic over an interval $T$. When this occurs, Mitchell resorts to a robust numerical “refinement” method, such as non-IA bisection or regula falsi. Interval Newton methods (e.g. [2, 23]) also compute $F(T)$ per iteration. Gradient interval computation proves expensive. Although previous works suggest these techniques offer improved convergence and efficiency compared to pure bisection, that position has been weakly scrutinized. In the context of coherent traversal, we find that interval bisection yields unequivocally better performance, and achieves equivalent visual results efficiently at coarser sampling rates.

To leverage SIMD vector operations, we perform interval bisection on four rays at a time. Rather than bisecting $t$ along the ray direction as in Figure 3(a), we bisect space along a major directional axis $K$, similar to the coherent octree volume traversal proposed in [11], and illustrated in Figure 3(b). Particularly when the space between rays exceeds the domain sampling width $\varepsilon$, this ensures more regular sampling of the function across neighboring rays, and preserves the spatial lockstep of coherent traversal (see Section 6.5).

The process of evaluating intervals is then simple. Given an interval box $B = X \times Y \times Z$, our function $f$ and its corresponding IA evaluation $F$, we evaluate whether $0 \in F(B)$ for any ray in the packet. If so, we bisect that interval along the major march axis, or register a hit if a maximum depth threshold is reached. Rather than evaluating the IA extension of the implicit $F(T)$ projected along the ray, as preferred by previous works, our $K$-marching technique (b) marches rays along a common axis in lockstep. Evaluating along 3D interval boxes $B$ requires slightly less computation per iteration than evaluating the projected function $f(t)$. More importantly, traversing along a common spatial axis induces more coherent behavior between rays in a packet.

![Figure 3: Interval bisection methods.](image)

**Figure 3**: Interval bisection methods. The conventional method (a) recursively bisects each ray along its parameter $t$ until a surface is located to the satisfaction of a termination criterion. Our $K$-marching technique (b) marches rays along a common axis in lockstep. Evaluating along 3D interval boxes $B$ requires slightly less computation per iteration than evaluating the projected function $f(t)$. More importantly, traversing along a common spatial axis induces more coherent behavior between rays in a packet.

### 5 Implementation

Our application takes as inputs a domain $\Omega \subseteq \mathbb{R}^3$, and an implicit function expression. For simplicity, we chose to hard-code most functions as IA expressions; however the function can also be received from the user as a string and then parsed and compiled into IA code in a dynamic library on-the-fly.

#### 5.1 SSE Interval Arithmetic

The foundation of our implicit ray tracing system is our own SSE IA library, which allows us to quickly evaluate intervals in SIMD. Implementation is straightforward; interval multiplication is partic-
Algorithm 1 SIMD Interval Arithmetic

```c
struct interval4 { 
    simd lo, hi; 
};
interval4 add4(interval4 a, interval4 b) { 
    return interval4(add(a.lo, b.lo), add(a.hi, b.hi)); 
} 
interval4 mul4(interval4 a, interval4 b) { 
    simd lo = mul4(a.lo, b.lo); 
    simd hi = mul4(a.hi, b.hi); 
    return interval4(min4(lo, hi), max4(lo, hi)); 
}
```

5.2 Ray Packet Structure

We chose conservative 2x2 packets for our implementation. Above all, we wish to evaluate baseline performance with SIMD ray tracing using 4-wide SSE vectors; thus behavior of our system should be consistent on wider SIMD hardware, such as a GPU or FGPA. Though larger packets coupled with multi-level algorithms could be significantly faster (e.g. [21]), 2x2 packet traversal is better-suited for general-purpose ray tracing, and easily allows our implicits to be integrated into a ray tracer as geometric intersection primitives. Though larger packets coupled with multi-level algorithms could be significantly faster (e.g. [21]), 2x2 packet traversal is better-suited for general-purpose ray tracing, and easily allows our implicits to be integrated into a ray tracer as geometric intersection primitives.

5.3 Traversal

Once the user has supplied a function, a domain box \( \Omega \subset \mathbb{R}^3 \), and a maximum depth \( d_{\text{stop}} \), we are ready to perform traversal. As in coherent grid traversal [26], we first find \( K \), the dominant axis of the first ray in the packet, and denote the remaining two axes \( U \) and \( V \). We then perform a standard ray bounding-box test on our domain. We store the actual \( t_{\text{enter}} \) and \( t_{\text{exit}} \) parameters as well as the intersections with the \( K \) entry and exit planes, \( t_{\text{Kenter}} \) and \( t_{\text{Kexit}} \). Now, we consider the total increment along \( K \), \( t_{\text{Kexit}} - t_{\text{Kenter}} \), and compute the total \( U \) and \( V \) increments over the entire domain. As our implementation is iterative, not recursive, we store an array containing a traversal “stack” for each depth \( \{0..d_{\text{stop}} - 1\} \), containing the \( t \), \( K \), \( U \) and \( V \) increments bisected at each level.

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

At each march iteration, we evaluate the IA function expression on this domain interval \( B = X \times Y \times Z \). If \( 0 \in F(X,Y,Z) \), we “re-curse” by incrementing \( d \) and using the bisected increments one level deeper. We register a hit on the surface when \( d = d_{\text{stop}} - 1 \) (or another hit criterion is met, such as \(|F(B)| < \delta \)), as in Section 5.5. Finally, we mask rays that successfully hit or terminate traversal when all rays hit. Traversal is illustrated in Figure 3(b), and pseudocode is given in Appendix A.

5.4 Division

IA division requires a slight modification to the above algorithm. 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 \( A/B = [-\infty, \infty] \). 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 be rendered. To avoid rendering asymptotes, we simply neglect to register a hit when \( F_B - F_I = \infty \). This principle is illustrated in Figure 4. With division correctly handled, our traverser will work for literally any function composed of IA operators.

5.5 Precision Criterion

In our implementation, \( d_{\text{stop}} \) determines the default precision for rendering the implicit. Roughly, this corresponds to a domain precision of \( 2^{-d_{\text{stop}}} \), though indeed this varies by ray. However, for a more view-independent domain-space metric, the user may optionally specify an \( \varepsilon \), such that \(|B| < \varepsilon \) serves as hit criterion, where \( B \) is an interval box \( X \times Y \times Z \). In this case, the stopping depth is determined adaptively per-packet as

\[
d_{\text{stop}} = \log_2(\Lambda_{\text{packet}}/\varepsilon)
\]  

(6)

where for world-space ray entry and exits \( P_t \) with the domain box \( \Omega \), and their corresponding \( K \)-coordinates \( K_r \),

\[
\Lambda_{\text{packet}} = \max_{t \in \text{packet}} \left( \frac{|P_{\text{exit}} - P_{\text{enter}}|^2}{|K_{\text{exit}} - K_{\text{enter}}|^2} \right)
\]  

(7)

Alternately, the user may specify a range tolerance \( \delta \), in which case our algorithm registers a hit when \(|F(B)| < \delta \). Empirically, the performance differences between these metrics proved minor, and at low precision the \( d_{\text{stop}} \) method yields more continuous results for neighboring rays. Thus, we use \( d_{\text{stop}} \) as the default metric for evaluating performance at varying sampling quality.
5.6 Shadows
In ray tracing, hard shadows are fairly trivial, requiring a shadow ray cast for every primary camera ray that hits a surface. This typically entails a 20% to 50% decrease in frame rate, depending on the coherent behavior of shadow rays. Fortunately, useful shadow rays require less accuracy than primary rays: it frequently suffices to cast shadows to a coarser termination depth, such as \( d_{\text{stop}} = 2 \), while employing a higher depth for primary rays. As shadows are primarily useful as depth cues, this is generally acceptable. The performance penalty is reduced, and loss of shadow detail is seldom perceptible (Figures 1 and 5).

5.7 Gradient Computation
For Lambertian shading, we require the surface normal at the ray hit position, given by the \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \) partial derivatives at that point. While analytical gradients 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. We allow the user to specify stencil width; this is frequently beneficial for surface regions with near-zero gradient magnitude (Figure 6).

6 Results
6.1 General Performance
All benchmarks were performed on an Intel Core Duo 2.16 GHz laptop with a 512\(^2\) frame buffer. Figure 9 shows various implicit surfaces with their associated equations and performance. Our system achieves well over 20 frames per second for simple objects such as the torus, sphere and conic sections. For more complex objects, performance can fall below interactive speeds on our hardware, but generally exploration at 1-5 fps is possible even for the worst cases. Complicated implicit surfaces such as the Barth-sixtic exhibit similar performance. Most importantly, we are not restricted to any particular class of surfaces. Non-differentiable, non-continuous, non-manifold, self-intersecting and linked implicit surfaces are all robustly rendered.

<table>
<thead>
<tr>
<th>( d_{\text{stop}} )</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.4 fps</td>
<td>13.6 fps</td>
<td>7.1 fps</td>
<td>4.2 fps</td>
<td></td>
</tr>
<tr>
<td>20.9 fps</td>
<td>9.1 fps</td>
<td>4.3 fps</td>
<td>2.4 fps</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Gradient normal computation, on the Heart function \( f(x,y,z) = (2x^2 + y^2 + z^2 - 1)^3 - (x^2 + y^2 - z^2)^2 \). Left: using analytical partial derivatives as gradient, we see shading artifacts where the gradient magnitude approaches zero. Center: with a central differences stencil of width \( \delta = 0.001 \), the results are visually indistinguishable. Right: smoother normals with \( \delta = 0.01 \). All images render at 6.7 fps.

Figure 7: Quality at various \( d_{\text{stop}} \) bisection depths. Performance is inversely proportional to depth. Top: the 1st-order Lagrangian trilinear interpolant patch, a cubic implicit, yields tight intervals and converges quickly to the correct contour. Bottom: the Mitchell function causes relatively high IA bound overestimation, and requires greater depth for correct visualization. Even here, a coarse precision criterion \( \epsilon < 10^{-3} \) is sufficient to capture the correct topology.

6.2 Precision and Quality
We use a common bisection depth \( d_{\text{stop}} \) for benchmarking, which corresponds to a domain precision of \( 2^{-d_{\text{stop}}} \) along the \( \epsilon \) axis of a given packet. The minimum depth required for accurate visualization depends largely on the bound overestimation of the composed IA rules for that function (Figure 7). As seen in Figure 9, \( d_{\text{stop}} = 10 \) is in practice a good balance of performance and feature reproduction for the vast majority of functions we test. This finding is surprising: a domain precision of \( \epsilon = 2^{-10} \approx 10^{-3} \) suffices to accurately visualize most implicits.

Figure 8: Reproduction of fine features. Though robust for each individual ray, ray tracing (as opposed to beam tracing) may fail to capture infinitely thin features. Coarser-contour visualization at lower precision actually aids in understanding these functions. Left: \( d_{\text{stop}} = 10 \) at 11 fps. Right: \( d_{\text{stop}} = 14 \) at 6.5 fps.

6.3 Feature Reproduction
The tear drop implicit (Figure 8) demonstrates how our algorithm can reproduce fine details that extraction-based approaches often omit. View-independent mesh extraction methods, e.g. [20], frequently fail to capture such regions of a surface, leading to misclassification of details such as asymptotes, singularities or infinitely thin connected surfaces. However, when thin regions or singularities lie between two rays and the interval bounds are sufficiently small, both discrete rays will (correctly) miss the surface, even though that surface would be encountered by an interval beam. To accurately reproduce such sub-pixel features would be expensive, requiring both supersampling and beam tracing of ray intervals, as detailed by Gavriliu [7]. Rendering at lower precision can actually aid in visualizing these features, as the IA inclusion property guarantees that our rendered surface will always form a convex contour of the actual zero-set (Figure 7). In this way, the user can iteratively modify \( d_{\text{stop}} \) until the true surface topology is understood.
Figure 9: Selected implicit functions, covering a wide range of different shapes and topologies. All examples are rendered at $d_{\text{stop}} = 10$ 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. Barth-sextic was rendered using $\tau = \frac{1 + \sqrt{5}}{2}$. 

<table>
<thead>
<tr>
<th>Torus</th>
<th>Sphere</th>
<th>Parabolic Cylinder</th>
<th>Hyperboloid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(r^2 - \sqrt{x^2 + y^2})^2 + z^2 - r^2_a$</td>
<td>$x^2 + y^2 + z^2 - r^2$</td>
<td>$x^2 - y - r^2$</td>
<td>$-\frac{x^2 - y - z}{x} - 1$</td>
</tr>
<tr>
<td>22.9 fps</td>
<td>20.5 fps</td>
<td>34.1 fps</td>
<td>19.2 fps</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Steiner 1</th>
<th>Steiner 2</th>
<th>Mitchell</th>
<th>Tangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>$xy + xz + xz + xz$</td>
<td>$(x^2 + y^2 + z^2)^3 + xz$</td>
<td>$4(x^4 + (x^2 + z^2)^2) + 17zx(x^2 + z^2) - 20(x^2 + y^2 + z^2) + 17$</td>
<td>$x^4 - 5x^2 - y^2 - 5y^2 - 5z^2 - 5z^2 + 11.8$</td>
</tr>
<tr>
<td>6.1 fps</td>
<td>17.2 fps</td>
<td>5.9 fps</td>
<td>3.8 fps</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Blobby</th>
<th>Absolute value</th>
<th>Inverse function</th>
<th>Teardrop</th>
<th>Intersecting planes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum_{i=1}^{N} \frac{x_i^2}{y_i^2} - 1$</td>
<td>$</td>
<td>x</td>
<td>+</td>
<td>y</td>
</tr>
<tr>
<td>4.7 fps</td>
<td>30.8 fps</td>
<td>20.8 fps</td>
<td>15.3 fps</td>
<td>33.0 fps</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Klein Bottle</th>
<th>Linked tori</th>
<th>Barth-sextic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x^2 + y^2 + z^2 + 2y - 1)((x^2 + y^2 + z^2 - 2y - 1)^2 - 8z^2)$</td>
<td>$g(x, y, z, c) = (x^2 + y^2 + z^2 + 1)^3 - 53(x^2 + y^2)$</td>
<td>$4(\tau^2 x^2 - y^2)(\tau^2 y^2 - z^2)(\tau^2 z^2 - x^2)$</td>
</tr>
<tr>
<td>$+16xz(x^2 + y^2 + z^2 - 2y - 1)$</td>
<td>$g(x, y, z, c) = (x^2 + y^2 + z^2 + c)^2 - 53(x^2 + y^2)$</td>
<td>$-(1 + 2\tau)(x^2 + y^2 + z^2 - 1)^2$</td>
</tr>
<tr>
<td>6.0 fps</td>
<td>4.0 fps</td>
<td>4.9 fps</td>
</tr>
</tbody>
</table>
6.4 Dynamic Scenes
Because we neither precompute an explicit representation of the object, nor a physical acceleration structure in memory, we have great flexibility in rendering dynamically changing N-dimensional implicits. For example, 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 10. Though dynamic implicits would be difficult to achieve with mesh extraction techniques, they are trivial in our ray tracing system.

Figure 10: 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 at 9-20 fps.

6.5 Algorithm Performance Analysis
Perhaps our most striking finding is that practical IA-based implicit rendering is not inherently slow, even though previous techniques yielded generally poor performance. Implementations such as Mitchell [16] and Capriani et al. [2] sought to render implicits at up to machine precision (up to \( \varepsilon = 10^{-30} \)) with superlinearly convergent numerical methods. Despite its slower theoretical convergence, we find that pure interval bisection is more efficient than these methods, particularly at lower precision which is more than adequate for correct visualization (see Section 6.2). To verify this, we implement an SSE variation of the Mitchell [16] algorithm, which performs interval bisection until all rays in the packet have \( 0 \notin F(B) \), followed by non-interval bisection for root refinement. Implemented in SSE, this method proves far slower than pure bisection, even with small \( \varepsilon \). In addition, we compare our K-marching algorithm with a standard \( \varepsilon \)-bisection. For large, partial \( \varepsilon \), standard \( \varepsilon \)-marching only performs 5% – 20% slower, depending on scene and computational demand of implicit evaluation. However, at smaller \( \varepsilon \), where the actual domain intervals of neighboring rays diverge spatially (Figure 3(a)), coherence suffers and K-marching is significantly more efficient, potentially by an order of magnitude. These findings are summarized in Table 1, and overall encourage implementation of our K-marching method.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>K-bisection</th>
<th>( \varepsilon )-bisection</th>
<th>Mitchell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain ( \varepsilon )</td>
<td>1e-3</td>
<td>2e-7</td>
<td>1e-3</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>FPS</td>
<td>10.6</td>
<td>2.8</td>
</tr>
<tr>
<td>trilerp</td>
<td>5.9</td>
<td>1.3</td>
<td>5.7</td>
</tr>
</tbody>
</table>

Table 1: Algorithm performance comparison between our K-bisection method, an SSE 2x2 packet implementation of the Mitchell [16] algorithm, and a pure \( \varepsilon \)-marching interval bisection. For the K-bisection method, these \( \varepsilon \) correspond to \( d_{\text{imp}} = 10 \) and \( d_{\text{imp}} = 22 \). Refer to Figure 7 for images of the trilinear interpolant (trilerp) and Mitchell functions.

6.6 Comparison to Existing Techniques
It is difficult to assess the performance of comparable works in implicit IA ray tracing. Fortunately, many papers evaluate performance with a sphere. [3] reported around 1.3 fps at 64x64 on a Pentium 166. Accounting generously for Moore’s Law (doubling performance every 18 months), we still achieve between two and three orders of magnitude better performance. Similarly, the hybrid and Interval Newton methods benchmarked in [23] perform at two to three orders of magnitude slower than our method. Florez et al. [5] rendered a sphere in 40 seconds at 300x300 resolution on a P4 2.4 GHz, albeit with adaptive antialiasing; again our method delivers over two orders of magnitude better frame rate (Figure 9).

7 CONCLUSION
We have detailed a coherent ray tracing technique for rendering arbitrary implicit functions. By combining a coherent traversal algorithm with an efficient SSE interval arithmetic library, we are able to visualize implicits robustly, accurately, and interactively at rates over two orders of magnitude faster than previous implementations.

Possibilities for extending our system abound. Performance could be further improved by using larger packets and multilevel coherent ray tracing techniques. Adaptive methods (e.g. [5]) might be desirable for better image quality at lower cost, particularly in conjunction with beam tracing (e.g. [7]), which could robustly anisotropically sample thin features and singularities. Performance with computationally difficult implicits, and particularly those with high bound overestimation, would improve with a higher-order inclusion rule set such as affine arithmetic [3] or midpoint-Taylor arithmetic [7]. Though it would entail some sacrifice in generality and portability, a similar interval bisection algorithm would be simple to implement, and likely fast, as a fragment program on the GPU.

While powerful, our method has some limitations. It is not an interval beam tracer; aliasing may occur when rendering functions with sub-pixel features at small tolerance. Though interactive for most implicits we tested, it is still computationally demanding and may not be as fast as special-case intersections, particularly for lower order implicits. More generally, implicits have not experienced widespread adoption in graphics compared to explicit modeling methods for smooth surfaces such as subdivision surfaces, though this has perhaps been partly due to their difficult rendering.

An immediate application for this work is a general-purpose 3D graphing application, for use in conjunction with a mathematical software package. CPU ray tracing is particularly attractive for this task as it requires no specialized graphics hardware. Ultimately, the ability to efficiently render general implicits could have interesting implications in graphics. Point-set rendering methods such as MPU [19] relying on rational implicits could easily be ray-traced using this technique. Procedural noise implicits could be employed for surfaces, as in [6]. In visualization, isosurfaces of higher-order finite elements [18] could be more efficiently rendered. Also of interest would be using a similar IA technique to ray-trace arbitrary parametric surfaces, as suggested by Mitchell [15].

8 ACKNOWLEDGMENTS
This work has been supported by the US Department of Energy Center for the Simulation of Accidental Fires and Explosions (CSAFE) grant W-7405-ENG-48, the National Science Foundation, CISE grants CRI-0513212, CCF-0541113 and SEI-0513212; and the Director, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under Contract DE-FE02-06ER25781, the SciDAC Visualization and Analytics Center for Enabling Technologies (VACET). Further support has been provided by the German Science Foundation (DFG IRTG 1131) as part of the International National Research Training Group; and through a visiting professorship sponsored by Intel Corp. Special thanks to John C. Hart, Burkhard Lehner, Matthias Gross, Andrew Kensler, Peter Djeu and Warren Hunt for their support and insights.

REFERENCES
Algorithm 2 Ray-Implicit Traversal.

templates<K, int U, int V, int D>
void traverse(RayPacket r, Box domain, Implicit implicit, int d_stop) {
    simd validmask = intersectBB(r, domain);
    //validmask indicates rays that are active
    float full_tk = t_kexit - t_kenter;
    float full_u = mul4(r.dir[U], full_tk);
    float full_v = mul4(r.dir[V], full_tk);
    struct Stack {
        simd u_incr, v_incr;
        float k_incr;
        char side;
    }
    stack stk[maxDepth];
    int depth = 0;
    while(stk[depth] & 1);
    do{
        if (any4(validmask)) {
            hitmask = and4(validmask, cmp_ge4(next_t, tenter));
            next_t = add4(curr_t, stk[depth].t_incr);
            next_v = add4(curr_v, stk[depth].v_incr);
            next_u = add4(curr_u, stk[depth].u_incr);
            stk[depth].side++;}
        else {
            return;
        }
        if (--depth == -1)
            break;
        depth++;
    }
    if (any4(F.contains(0))) {
        interval4 F = implicit.evaluate_interval4(ibox);
        (fill ibox with curr and next k,u,v)
        interval4 ibox;
        simd u_incr, v_incr;
        simd t_incr;
        float k_incr;
        simd curr_k = stk[0].k_incr;
        k_incr = curr_k - stk[depth].k_incr;
        return;}
    //recurse depth++;
    continue;
    }
    }
    if (!all4(cmp_ge4(sub4(F.hi,F.lo),INFINITY)){
        if (any4(F.contains(0))) {
            interval4 F = implicit.evaluate_interval4(ibox);
            (compute normal)
            return;
        }
    //recurse depth++;
    continue;
    }
    }
    validmask = and4(validmask, cmp_lef(next_r, teals));
    if (none4(validmask))
        return;
    curr_k = next_k;
    curr_r = next_r;
    curr_u = next_u;
    curr_v = next_v;
    if (inside(depth),side & 1)
        do{
            if (--depth == -1)
                return;
            while(stk[depth] & 1);
            continue;
        }
    }
Algorithm 2 Ray-Implicit Traversal.