COMPUTER GRAPHICS forum

DOI: 10.1111/j.1467-8659.2008.01189.x

COMPUTER GRAPHICS forum
Volume 28 (2009), number 1 pp. 26–40

Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic

A. Knoll[1,3], Y. Hijazi[2,3], A. Kensler[1], M. Schott[1], C. Hansen[1,3] and H. Hagen[2,3]

[1]SCI Institute, University of Utah

[2]University of Kaiserslautern

[3]International Research Training Group (IRTG 1131)

Abstract

Existing techniques for rendering arbitrary-form implicit surfaces are limited, either in performance, correctness or flexibility. Ray tracing algorithms employing interval arithmetic (IA) or affine arithmetic (AA) for root-funding are robust and general in the class of surfaces they support, but traditionally slow. Nonetheless, implemented efficiently using a stack-driven iterative algorithm and SIMD vector instructions, these methods can achieve interactive performance for common algebraic surfaces on the CPU. A similar algorithm can also be implemented stacklessly, allowing for efficient ray tracing on the GPU. This paper presents these algorithms, as well as an inclusion-preserving reduced affine arithmetic (RAA) for faster ray-surface intersection. Shader metaprogramming allows for immediate and automatic generation of symbolic expressions and their interval or affine extensions. Moreover, we are able to render even complex forms robustly, in real-time at high resolution.

Keywords: ray tracing, reduced affine arithmetic, shader metaprogramming

ACM CCS: I.3.1, I.3.5, I.3.7 [Computer Graphics]: *Graphics processors; curve, surface, solid and object representations; ray tracing*

1. Introduction

To render implicit surfaces, one is principally given two choices: sampling the implicit and extracting proxy geometry such as a mesh, volume or point cloud; or ray tracing the implicit directly. Though the former methods are often preferred due to the speed of rasterizing proxy geometries, extraction methods are view-independent and often scale poorly. Though computationally expensive, ray-tracing methods parallelize efficiently and trivially. Modern graphics hardware offers enormous parallel computational power, at the cost of poor efficiency under algorithms with branching and irregular memory access. GPU-based ray tracing [GPSS07, PBMH02] is increasingly common, but often algorithmically inefficient.

Ray-tracing methods for implicit surfaces have historically sacrificed either speed or correctness or flexibility. Piecewise algebraic implicits have been rendered in real-time on the GPU using Bezier decompositions [LB06], but approximating methods do not render arbitrary expressions directly, nor always robustly. Self-validated arithmetic methods, such as interval arithmetic (IA) or affine arithmetic (AA), are extremely general in that theoretically any composition of Lipschitz-boundable functions can be expressed as an inclusion extension and solved robustly. However, these approaches have historically been among the slowest.

This paper discusses optimization of interval and AA methods to allow for interactive ray tracing of arbitrary-form implicit surfaces on the CPU and GPU. Knoll *et al.* [KHW*07b] proposed an optimized coherent intersection algorithm using SSE vector instructions, achieving interactive ray tracing for most simple surfaces on a dual-core CPU. We first discuss this SIMD CPU algorithm (Section 4), and then extend it with two new contributions: an efficient implementation of a reduced affine arithmetic (RAA) that correctly

Submitted: March 2007
Revised: July 2007
Accepted: July 2007

preserves the inclusion property (Section 5.3); and a stackless interval bisection algorithm for ray-tracing implicits on the GPU (Section 5.5). Together, these enable real-time rendering of complex implicit functions. Shader metaprogramming allows users to design implicit forms and procedural geometry flexibly and dynamically, with full support for dynamic 4D surfaces. Ray tracing allows multi-bounce effects to be computed interactively without image-space approximations, enabling effects such as transparency and shadows which further assist visualization.

2. Related Work

2.1. Proxy geometry methods

Due to the popularity of GPU rasterization, the most common approach to rendering implicits has been extraction of a mesh or proxy geometry. Application of marching cubes [WMW86] or Bloomenthal polygonization [Blo94] can generate meshes interactively, but will entirely omit features smaller than the static cell width. More sophisticated methods deliver better results, at the cost of interactivity. Paiva *et al*. [PLLdF06] detail a robust algorithm based on dual marching cubes, using IA in conjunction with geometric oracles. Varadhan *et al*. [VKZM06] employ dual contouring and IA to decompose the implicit into patches, and compute a homeomorphic triangulation for each patch. These methods exploit inclusion arithmetic to generate desirable meshes that preserve topology within geometric constraints. However, they generally compute offline, and do not scale trivially. Moreover, each mesh is a view-independent reconstruction.

Non-polygonal proxy geometry is also practical. Dynamic particle sampling methods for implicits have been demonstrated by Witkin and Heckbert [WH94] and extended by Meyer *et al*. [MGW05]. Voxelization is also a valid approach to representing implicit forms as scalar fields [SK01]. Rendering of recursively voxelized object space with interval arithmetic was first proposed by [WQ80]. Direct volume rendering, or other GPU volume raycasting methods, can also be viable ways of visualizing isosurfaces [HSS*05].

2.2. Ray tracing implicit surfaces

Hanrahan [Han83] proposed a general but non-robust point-sampling algorithm using Descartes' rule of signs to isolate roots. van Wijk [vW85] implemented a recursive root bracketing algorithm using Sturm sequences, suitable for differentiable algebraics. Kalra and Barr [KB89] devised a method of rendering a subclass of algebraic surfaces with known Lipschitz bounds. Hart [Har96] proposed a robust method for ray tracing algebraics by defining signed distance functions from an arbitrary point to the surface. More recently, Loop and Blinn [LB06] implemented an extremely fast GPU ray caster approximating implicit forms with piecewise Bernstein polynomials. [RVdF06] proposed a hybrid GPU/CPU technique for casting rays through constructive solid geometry (CSG) trees of implicits. [dTLP07] demonstrated GPU ray casting of cubics and quartics using standard iterative numerical methods. [FP08] employ rule-of-signs interval methods in ray tracing generalized implicit (FRep) surfaces on the GPU.

2.2.1. *Ray tracing with interval and affine arithmetic*

Toth [Tot85] first applied IA to ray tracing parametric surfaces, in determining an initial convex bound before solving a nonlinear system. Mitchell [Mit90] ray traced implicits using recursive IA bisection to isolate monotonic ray intervals, in conjunction with standard bisection as a root refinement method. Heidrich and Seidel [HS98] employed AA in rendering parametric displacement surfaces. de Cusatis Junior *et al*. [dCJdFG99] used standard AA in conjunction with recursive bisection. Sanjuan-Estrada *et al*. [SECG03] compared performance of two hybrid interval methods with Interval Newton and Sturm solvers. Florez *et al*. [FSSV06] proposed a ray tracer that antialiases surfaces by adaptive sampling during interval subdivision. Gamito and Maddock [GM07] proposed RAA for ray casting specific implicit displacement surfaces formulated with blended noise functions, but their AA implementation fails to preserve inclusion in the general case. Knoll *et al*. [KHW*07b] implemented a generally interactive interval bisection algorithm for rendering arbitrary implicit forms on the CPU. Performance was achieved though SSE instruction-level optimization and coherent traversal methods; and exploiting the fact that numerically precise roots are not required for visual accuracy.

3. Background

3.1. Ray tracing implicit surfaces

A surface S in implicit form in 3D is the set of solutions of an equation

$$f(x, y, z) = 0 \tag{1}$$

where $f : \Omega \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$. In ray tracing, we seek the intersection of a ray

$$\vec{p}(t) = \vec{o} + t\vec{d} \tag{2}$$

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

$$f_t(t) = f(o_x + td_x, o_y + td_y, o_z + td_z) \tag{3}$$

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

In ray tracing, all geometric primitives are at some level defined implicitly, and the problem is essentially one of solving for roots. Simple implicits such as a plane or a sphere have closed-form solutions that can be solved trivially. More

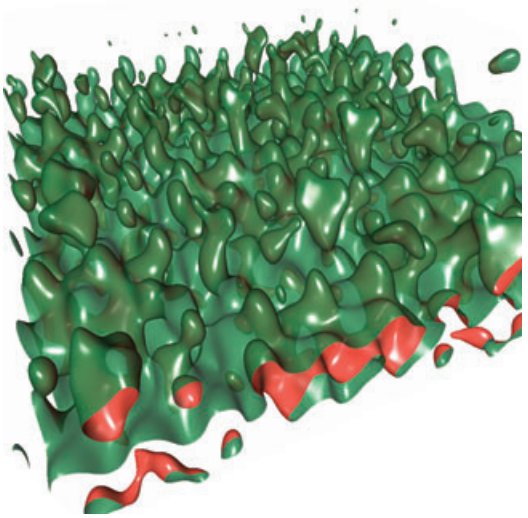

Figure 1: *An animated sinusoid-kernel surface. Ray-traced directly on fragment units, no new geometry is introduced into the rasterization pipeline. IA/AA methods ensure robust rendering of any inclusion-computable implicit.*

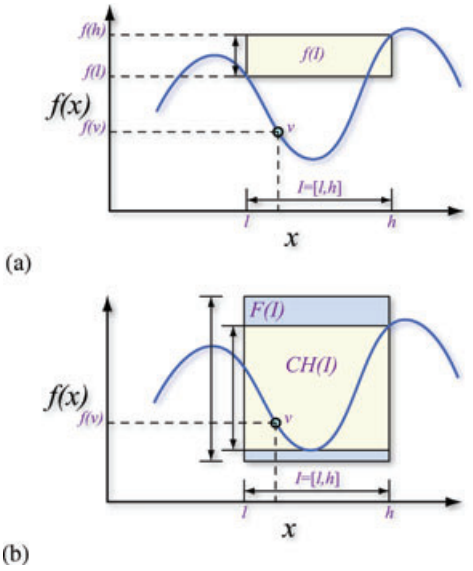

Figure 2: The inclusion property: *(a) when a function f is non-monotonic on an interval I, evaluating the lower and upper components of a domain interval is insufficient to determine a convex hull over the range. This is not the case with an inclusion extension F (b), which encloses all minima and maxima of the function within that interval. Ideally, F(I) closely envelopes the actual convex hull, CH(I), enclosing the upper and lower Lipschitz bounds of f.*

complicated surfaces without a closed-form solution require iterative numerical methods. However, easy methods such as Newton–Raphson, and even 'globally-convergent' methods such as *regula falsi*, only work on ray intervals where f is monotonic. As shown in Figure 2, 'point sampling' using the rule of signs (e.g. [Han83]) fails as a robust rejection test on non-monotonic intervals. While many methods exist for isolating monotonic regions or approximating the solution, inclusion methods using interval or AA are among the most robust and general. Historically, they have also been among the slowest, due to inefficient implementation and impractical numerical assumptions.

3.2. Interval arithmetic and inclusion

IA was introduced by Moore [Moo66] as an approach to bounding numerical rounding errors in floating point computation. The same way classical arithmetic operates on real numbers, IA defines a set of operations on intervals. We denote an interval as $\underline{\overline{x}} = [\underline{x}, \overline{x}]$, and the base arithmetic operations are as follows:

$$\underline{\overline{x}} + \underline{\overline{y}} = [\underline{x} + \underline{y}, \overline{x} + \overline{y}], \underline{\overline{x}} \times \underline{\overline{y}} = [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \quad (4)$$

$$\underline{\overline{x}} \times \underline{\overline{y}} = [\min(\underline{x}\,\underline{y}, \underline{x}\,\overline{y}, \overline{x}\,\underline{y}, \overline{x}\,\overline{y}), \max(\underline{x}\,\underline{y}, \underline{x}\,\overline{y}, \overline{x}\,\underline{y}, \overline{x}\,\overline{y})]. \quad (5)$$

Moore's fundamental theorem of IA [Moo66] states that for any function f defined by an arithmetical expression, the corresponding interval evaluation function F is an *inclusion function* of f:

$$F(\underline{\overline{x}}) \supseteq f(\underline{\overline{x}}) = \{f(x) | x \in \underline{\overline{x}}\} \quad (6)$$

where F is the interval *extension* of f.

The inclusion property provides a robust rejection test that will definitely state whether an interval $\underline{\overline{x}}$ *possibly* contains a zero or other value. Inclusion operations are powerful in that they are composable: if each component operator preserves the inclusion property, then arbitrary compositions of these operators will as well. As a result, in practice *any* computable function may be expressed as inclusion arithmetic [Mit90]. Some interval operations are ill-defined, yielding empty-set or infinite-width results. However, these are easily handled in a similar fashion as standard real-number arithmetic. A more difficult problem is converting existing efficient real-number implementations of transcendental functions to inclusion routines, as opposed to implementing an IA version from base operators. This requires ingenuity, but is usually possible and far faster than implementing an extension approximation from scratch.

The IA extension is often referred to as the *natural inclusion function*, but it is neither the only mechanism for defining an inclusion algebra, nor always the best. Particularly in the case of multiplication, it greatly overestimates the

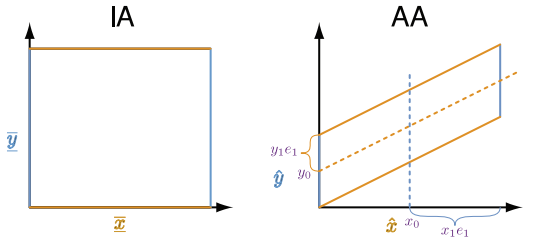

Figure 3: *Bounding forms resulting from the combination of two interval (left) and affine (right) quantities.*

actual bounds of the range. To overcome this, it is necessary to represent intervals with higher-order approximations.

3.3. Affine arithmetic

AA was developed by Comba and Stolfi [CS93] to address the bound overestimation problem of IA. Intuitively, if IA approximates the convex hull of f with a bounding box, AA employs a piecewise first-order bounding polygon, such as the parallelogram in Figure 3.

An affine quantity $\hat{x}$ takes the form:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i e_i \tag{7}$$

where the $x_i, \forall i \geq 1$ are the *partial deviations* of $\hat{x}$, and $e_i \in [-1,1]$ are the *error symbols*. An affine form is created from an interval as follows:

$$x_0 = (\overline{x} + \underline{x})/2,\ x_1 = (\overline{x} - \underline{x})/2,\ x_i = 0,\ i > 1 \tag{8}$$

and can equally be converted into an interval

$$\underline{\overline{x}} = [x_0 - \mathrm{rad}(\hat{x}), x_0 + \mathrm{rad}(\hat{x})] \tag{9}$$

where the *radius* of the affine form is given as:

$$\mathrm{rad}(\hat{x}) = \sum_{i=1}^{n} |x_i|. \tag{10}$$

Affine operations in AA, where $c \in \mathbb{R}$, are as follows:

$$\begin{aligned} c \times \hat{x} &= cx_0 + c\sum_{i=1}^{n} x_i e_i \\ c \pm \hat{x} &= (c \pm x_0) + \sum_{i=1}^{n} x_i e_i \\ \hat{x} \pm \hat{y} &= (x_0 \pm y_0) \pm \sum_{i=1}^{n} (x_i \pm y_i) e_i . \end{aligned} \tag{11}$$

However, *non-affine* operations in AA cause an additional error symbol e_z to be introduced. This is the case in multiplication between two affine forms,

$$\hat{x} \times \hat{y} = x_0 y_0 + \sum_{i=1}^{n} (x_i y_0 + y_i x_0) e_i + \mathrm{rad}(\hat{x})\mathrm{rad}(\hat{y}) e_z . \tag{12}$$

Other operations in AA, such as square root and transcendentals, approximate the range of the IA operation using a regression curve – a slope bounding a minimum and maximum estimate of the range. These operations are also non-affine, and require a new error symbol.

3.3.1. *Condensation and reduced affine arithmetic*

The chief improvement in AA comes from maintaining correlated error symbols as orthogonal entities. This effectively allows error among correlated symbols to diminish, as opposed to always increasing monotonically in IA. Unfortunately, as the number of non-affine operations increases, the number of non-correlated error symbols increases as well. Despite computing tighter bounds, standard AA ultimately is inefficient in both computational and memory demands. To remedy this, AA implementations employ *condensation*. If $\hat{x}$ has n symbols, then it can be condensed into an affine entity $\hat{y}$ with $m < n$ symbols as follows [CS93]:

$$\begin{aligned} y_i &= x_i \quad \forall i = 0, \ldots m-1 \\ y_m &= \sum_{i=m}^{n} |x_i| . \end{aligned} \tag{13}$$

While $\hat{y}$ indeed bounds $\hat{x}$, condensation destroys all correlations pertaining to e_m. As a result, after condensation involving a symbol e_m, only positive-definite affine operations involving that symbol may be applied in order to preserve inclusion. Gamito and Maddock [GM07] employ a three-term reduced AA that performs such condensation for every non-affine operation. Though symbol correlation is destroyed, they construct their specific extension evaluation to preserve inclusion. Nonetheless, condensation is ill-suited for arbitrary expressions, which may perform affine or non-affine operations in any order.

3.3.2. *Inclusion-preserving reduced affine arithmetic*

In our own search for a correlation-preserving RAA, we adopted a formulation equivalent to that proposed by Messine [Mes02]. In his AF1 formulation, condensation of an entity with $n+1$ total symbols,

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i e_i + x_{n+1} e_{n+1} \tag{14}$$

entails arithmetic operations as follows:

$$\begin{aligned}
c \pm \hat{x} &= (c \pm x_0) + \sum_{i=1}^{n} x_i e_i + |x_{n+1}| e_{n+1} \\
\hat{x} \pm \hat{y} &= (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i) e_i + (x_{n+1} + y_{n+1}) e_{n+1} \\
c \times \hat{x} &= (c x_0) + \sum_{i=1}^{n} c x_i e_i + |c x_{n+1}| e_{n+1} \\
\hat{x} \times \hat{y} &= (x_0 y_0) + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i) e_i + \\
&\quad (|x_0 y_{n+1}| + |y_0 x_{n+1}| + \mathrm{rad}(\hat{x})\mathrm{rad}(\hat{y})) e_{n+1}. \qquad (15)
\end{aligned}$$

Here, affine operations enforce positive-definite correlations between error symbols. While this does not compute as tight bounds as conventional AA, it is suitable for fixed-size vector implementation, and is in most cases a significant improvement over IA. We therefore use this as our formulation for RAA.

3.4. Ray tracing implicits with inclusion arithmetic

The inclusion property extends to multivariate implicits as well, making it suitable for a spatial rejection test in ray tracing. Moreover, by substituting the inclusion extension of the ray equation (Equation 2) into the implicit extension *F(x, y, z)*, we have a univariate extension $F_t(X, Y, Z)$. To check whether any given ray interval $\bar{t} = [\underline{t}, \overline{t}]$ *possibly* contains our surface, we simply check if $0 \in F_t(\bar{t})$. As a result, once the inclusion library is implemented, any function composed of its operators can be rendered robustly. To pick domain intervals on which to evaluate the extension, one has a wide choice of interval numerical methods. The simplest option is pure recursive bisection of intervals, examined in the order of the ray direction [dCJdFG99, GM07, KHW*07b, Mit90]. Alternatives involve quasi-Newton methods and variants of the Interval Newton algorithm [CHMS00, SECG03] that rely on the inclusion extension of the function gradient.

4. SIMD CPU Ray Tracing Algorithm

The SIMD CPU implementation was originally presented in [KHW*07b], and was motivated by the relatively high performance of coherent ray tracing algorithms involving grids [WIK*06] or octrees [KHW07a], compared to existing work in rendering algebraic surfaces. The approach of this system is to treat interval bisection as an iterative spatial traversal algorithm, exploiting ray coherence and SSE vector instructions to achieve speedup over conventional recursive single-ray algorithms. As shown in Section 6.1.3, brute-force bisection outperforms more sophisticated quasi-Newton methods, particularly for the purpose of rendering implicits which requires relatively low numerical precision. Ultimately, in Section 6.1.3, we find that strategies for maximizing SIMD coherence and performance differ on CPU and GPU platforms.

4.1. SIMD interval arithmetic

The largest cost in rendering general-form implicits is in evaluating the interval extension. The first optimization is therefore to write an IA library that exploits SIMD vector instructions. Although an interval is itself a 2-vector, it is most effective to operate on a vector of intervals. For example, given the four-float SSE type __m128:

Algorithm 1: SIMD Interval Arithmetic.

```
struct interval4{
  __m128 lo, hi;
};

interval4 isub4 (interval4 a, interval4 b){
  interval4 i;
  i.lo = _mm_sub_ps (a.lo, b.hi);
  i.hi = _mm_sub_ps (a.hi, b.lo);
  return i;
}
```

This computes an interval extension *F(X, Y, Z)* composed of these operators for four interval values, permitting simultaneous evaluation of the extension on four rays.

4.2. Coherent traversal

As interval evaluation is performed on four values at once in SIMD, the bisection algorithm must also operate on four rays simultaneously. As bisection of the ray distance parameter t is equivalent to subdivision of world space, our problem is essentially similar to acceleration structure traversal. Coherent SIMD methods [WBS02] perform a similar task by considering whether *any* ray in a packet intersects a node or descend a sub-tree; we instead query whether a given world-space region possibly contains a zero of the implicit surface for any ray.

Coherent traversal algorithms perform best when rays in the same packet exhibit similar behaviour; in our case, descending the same sides of the binary search tree whenever possible. As seen in Figure 4(a), direct bisection of the t distance parameter can cause ray behaviour to diverge, requiring more traversal steps. Our solution is to instead determine a major march direction **K** and bisect along that axis, resulting in improved coherent behaviour (Fig. 4(b)). Other strategies for ensuring coherence involve unitizing the rays, or normalizing directions with respect to a single reference ray in the packet. However, we have found that for explicit SIMD traversal, picking the dominant **K** axis works best.

SIMD traversal on the CPU is executed in a multi-threaded coherent ray tracer, such as Manta [BSP06]. The SSE

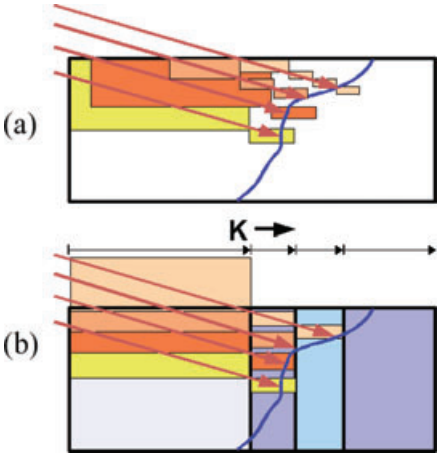

Figure 4: Spatial traversal with interval bisection. *The conventional single-ray method (a), as well as our GPU algorithm, bisects the ray distance parameter t until a surface is located to the satisfaction of a termination criterion. SIMD CPU traversal (b) picks the dominant ray direction* $\mathbf{K}$ *of a group of rays, and bisects that axis. This ensures more coherent and less divergent behaviour during traversal, and thus greater SIMD speedup.*

algorithm is iterative, not recursive, employing a precomputed array of t-space and world-space increments multiplied by $1/2^{d_{\max}}$, where $d_{\max}$ represents the maximum bisection depth. Redundant computation can be avoided by maintaining and incrementing the $\overline{x}, \overline{y}, \overline{z}$ intervals separately, rather than always computing the ray equation extension as a function of $\overline{t}$. A writable array of booleans keeps track of which side of the bisection tree is visited at each depth level. This approach relies on numerous registers and L1 cache as efficient substitutes for stack recursion, and is well-suited to CPU architectures. Pseudocode for this algorithm is given in [KHW*07b].

5. GPU Algorithm

The new contributions of this paper are a GPU implementation of the interval bisection algorithm (Section 5.5) and an implementation of RAA suitable for the GPU 5.3. Overall, shader languages such as Cg 2.0 allow for a more graceful implementation than the optimized SSE C++ counterpart on the CPU. Just-in-time shader compilation, in conjunction with metaprogramming, can easily and dynamically generate IA/AA extension routines from an input expression. Nonetheless, implementing a robust interval-bisection ray tracer on the GPU poses challenges. Principally, the CPU algorithm relies on an efficient iterative algorithm for bisection, employing a read/write array for the recursion stack. Storing such an array per-fragment occupies numerous infrequently-used registers, which slows processing on the GPU. Similar problems have clearly hampered performance of hierarchical acceleration structure traversal for mesh ray tracing [PGSS07]. Our most significant contribution is a traversal algorithm that overcomes this problem. By employing simple floating-point modulus arithmetic in conjunction with a DDA-like incremental algorithm operating on specially constructed intervals, we are able to perform traversal without any stack. Though this algorithm would be inefficient on a CPU, it is well-suited for the GPU architecture thanks to efficient floating-point division.

5.1. Application pipeline

As input, the user must simply specify a function in implicit form, a domain $\Omega \subset \mathbb{R}^3$, and a termination precision ε that effectively bounds relative error (see Section 6.1.2). Userspecified variables are stored on the CPU and passed dynamically to Cg as uniform parameters. Some runtime options, such as the implicit function, choice of inclusion algebra or shading modality, are compiled directly into the Cg shader through metaprogramming. In simple cases, the CPU merely searches for a stub substring within a base shader file, and replaces it with Cg code corresponding to the selected option. More advanced metaprogramming involves creating routines for function evaluation. Given an implicit function expression, we require two routines to be created within the shader: one evaluating the implicit f, and another evaluating the inclusion function, the interval or affine extension F. We use a simple recursive-descent parser to generate these routines in the output Cg shader. Alternately, we allow the user to directly provide inline Cg code. Because the shader compiler identifies common subexpressions, this is seldom necessary for improving performance. Our only examples employing inline code are special-case conditional evaluations in CSG objects (Figure 10).

Though our system is built on top of OpenGL, we use the fixed-function rasterization pipeline very little. Given a domain $\Omega \subset \mathbb{R}^3$ specified by the user, we simply rasterize that bounding box once per frame. We specify the world-space box vertex coordinates as texture coordinates as well. These are passed straight through a minimal vertex program, and the fragment program merely looks up the automatically interpolated world-space entry point of the ray and the bounding box. By subtracting that point from the origin, we generate a primary camera ray for each fragment.

5.2. Shader IA library

Implementing an IA library (Section 3.2) is straightforward in Cg. Most scalar operations employed by IA (such as min and max) are highly efficient on the GPU, and swizzling allows for effective horizontal vector implementation (Algorithm 2), unlike SSE SIMD on the CPU. Transcendental functions are

particularly efficient for both their floating-point and interval computations. Integer powers are yet more efficient, thanks to a bound-efficient IA rule for even powers, just-in-time compilation and Russian peasant multiplication [Mid65].

5.3. Shader RAA library

In implementing our RAA library on the GPU, we adopt a formulation similar to AF1 in Messine *et al*. [Mes02], with changes to the absolute value bracketing that are mathematically equivalent but slightly faster to compute. We implemented AF1 with $n = 1$ using a float3 to represent the reduced affine form. We also experimented with $n = 2$ (float4), and $n = 6$ (a double-float4 structure). For all the functions in our collection, the float3 version delivered the fastest results by far. We also found that the computational overhead of the bound-improved AF2 formulation [Mes02] was too high to be efficient. Examples of the float3 version of the forms in Equation 15 are given in Algorithm 3.

The float3 implementation of AF1 makes for a versatile and fast RAA. Particularly for functions with significant multiplication between non-correlated affine variables, such as the Mitchell or the Barth surfaces involving cross-multiplication of Chebyshev polynomials, significant speedup can be achieved over standard IA.

Algorithm 2: Excerpt of GPU Interval Arithmetic

```
typedef float2 interval;

interval iadd (interval a, interval b) {
  return interval ( add(a.x, b.x),
  add(a.y, b.y) );

}
interval imul (interval a, interval b) {
  float4 lh = a.xxyy * b.xyxy;
  return interval (min(lh.x, min(lh.y,
  min(lh.z, lh.w)))),
       max(lh.x, max4(lh.y, max(lh.z, lh.w)));
}
interval ircp(const float inf, interval i) {
  return ( (i.x <= 0 && i.y >= 0) ?
     interval(-inf, inf) : 1/i.yx );
}
```

Algorithm 3: Excerpt of GPU Reduced Affine Arithmetic

```
typedef float3 raf;

raf interval_to_raf (interval i){
  raf r;
  r.x = (i.y + i.x);
  r.y = (i.y - i.x);
  r.xy * = .5; r.z = 0;
  return r;
}
float raf_radius(raf a){
  return abs (a.y) + a.z;
}
interval raf_to_interval(raf a){
  const float rad = raf_radius(a);
  return interval(a.x - rad, a.x + rad);
}
raf raf_add (raf a, raf b){
  return (a + b);
}
raf raf_mu1{raf a, raf b){
  raf r;
  r.x = a.x * b.x;
  r.y = a.x*b.y + b.x*a.y;
  r.z = abs(a.x*b.z) + abs(b.x*a.z) +
        raf_radius(a)*raf_radius(b);
  return r;
}
```

5.4. Numerical considerations

A technical difficulty arises in the expression of infinite intervals, which may occur in division; and empty intervals that are necessary in omitting non-real results from a fractional power or logarithm. While these are natively expressed by nan on the CPU, GPU's are not always IEEE compliant. The G80 architecture correctly detects and propagates infinity and nan, but the values themselves (inf = 1/0 and nan = 0/0) must be generated on the CPU and passed into the fragment program and subsequent IA/AA calls.

Conventionally, IA and AA employ a rounding step after every operation, padding the result to the previous or next expressible floating point number. We deliberately omit rounding – in practice the typical precision ε is sufficiently large that rounding has negligible impact on the correct computation of the extension F. However, numerical issues can be problematic in certain affine operations: RAA implementations of square root, transcendentals and division itself all rely on accurate floating point division for computing the regression lines approximating affine forms. Though inclusion-preserving in theory, these methods are ill-suited for inaccurate GPU floating point arithmetic; and a robust strategy to overcome these issues has not yet been developed for RAA. We therefore resort to IA for functions that require regression-approximation AA operators.

5.5. Traversal

With the IA/RAA extension and a primary ray generated on the fragment unit, we can perform ray traversal of the domain $\Omega \subset \mathbb{R}^3$. Though not as trivial as standard numerical bisection for root finding, the ray traversal algorithm is nonetheless elegantly simple (Algorithm 4).

Initialization: We begin by computing the exit point p_{exit} of the generated ray and the bounding box Ω. We reparameterize the ray as $\vec{r}(t) := \vec{p}_{\text{enter}} + t(\vec{p}_{\text{exit}} - \vec{p}_{\text{enter}})$. The interval $\bar{t}$ along the ray intersecting Ω is now [0,1]. We now perform a first rejection test outside the main loop.

Rejection test: In the rejection test, we evaluate the IA/AA extensions of the ray equation to find X, Y and Z over $\underline{t}$, and use these (as well as scalars w, r_i for time and other animation variables) to evaluate the extension of our implicit function. The result gives us an interval or affine approximation of the range F. If $0 \in F$, then we must continue to bisect and search for roots. Otherwise, we may safely ignore this interval and proceed to the next, or terminate if it is the last.

Main loop: If the outer rejection test succeeds, we compute the effective bisection depth required for the userspecified ε. This is given by the integer ceiling:

$$d_{\max} := ceil\left(\log 2\left(\frac{(||\tilde{p}_{\text{exit}} - \tilde{p}_{\text{enter}}||}{\varepsilon}\right)\right). \qquad (16)$$

We initialize our depth $d = 0$, and distance increment, $t_{\text{incr}} = 0.5$. Now, recalling the bisection interval $\underline{t}$, we set $\bar{t} := \bar{t} + t_{\text{incr}}$. We then perform the rejection test on this new $\underline{t}$. If the test succeeds, we either hit the surface if we have reached $d = d_{\max}$, or recurse to the next level by setting $t_{\text{incr}} := t_{\text{incr}}/2$, and incrementing d.

If the rejection test fails, we proceed to the next interval segment at the current depth level by setting $\bar{t} := \underline{t}$. Within the main loop, we now perform another loop to back-recurse to the appropriate depth level.

Back-recursion loop: In back-recursion, we decrement the depth d (and update t_{incr}) until we find an unvisited segment of the bisection tree. This allows us to perform ray bisection iteratively, not recursively, and without employing registers to mimic a recursion stack. Specifically, we perform floating-point modulus ($\underline{t} \,\%\, 2t_{incr} = 0$) to verify whether the current distance has visited one or both bisected segments in question. Currently on the G80, the fastest method proves to be performing division and examining the remainder. Backrecursion proceeds iteratively until it finds an unvisited second branch of the bisection tree, or $d = -1$ in which case traversal has completed.

Algorithm 4: Traversal Algorithm with RAA

```
float traverse (float3 penter, float3 pexit,
  float w,
   float max_depth, float eps, float nan,
   float inf){
  const float3 org = penter;
  const float3 dir = pexit-penter;
  interval t(0,1);
  raf F, it, ix, iy, iz;
  //rejection test
  ix = raf_add(org.x, raf_mu(it, dir.x));
  iy = raf_add (org.y, raf_mu(it, dir.y));
  iz = raf_add (org.z, raf_mu(it, dir.z));
  F = evaluate_raf (ix, iy, iz, w, nan, inf);
  if (raf_contains(F, 0)){
    int d = 0;
    float tincr = .5;
    const int dlast = log2
    (length(dir)/epsilon);
    //main loop
    for(;;){
      t.y = t.x + tincr;
      (compute ix, iy, iz, F again for
      rejection test)
      if (raf_contains(F, 0)){
       if (d = = dlast){return t.x; /*hit*/}
       else{tincr * = .5; d++; continue;}
      }
      t.x = t.y;
      //back-recursion
      float fp = frac(.5*t.x/tincr);
      if (fp < 1e-8){
       for(int j = 0; j< = dlast; j++){
        tincr * = 2;
        d--;
        fp = frac(.5*t.x/tincr);

        if (d = = −1 || fp > 1e-8) break;
       }
       if (d = = −1) break;
      }
    }
  }
  return −1; //no hit
}
```

5.6. Traversal metaprogramming

The traversal algorithm largely remains static, but some functions and visualization modalities require special handling. To render functions containing division operations, we must check whether intervals are infinitely wide before successfully hitting, as detailed in Knoll *et al.* [KHW*07b]. Multiple isovalues and transparency require modifications to the rejection test and hit registration, respectively, as discussed in Section 6.2. More generally, modifications to the traversal algorithm are simple to implement via 'inline' implicit files (Section 5.1). We allow the user to directly program behaviour of the rejection test, hit registration and shading. This is particularly useful in rendering special-case CSG objects (Figure 10).

5.7. Shading

Phong shading requires a surface normal, specifically the gradient of the implicit at the found intersection position. We find central differencing to be more than adequate, as it requires no effort on the part of the user in specifying analytical derivatives, nor special metaprogramming in computing separable partials via automatic differentiation. By default, we use a stencil width proportional to the traversal precision ε; variable width is often also desirable [KHW*07b].

6. Results

All benchmarks are measured in frames per second at 1024 × 1024 frame buffer resolution. CPU GPU

6.1. Performance

Table 1 shows base frame rates of a variety of surfaces using single ray casting and basic Phong shading. Performance on the NVIDIA 8800 GTX is up to 22× faster than the SIMD

Table 1: *Performance in fps for various surfaces at 1024 × 1024 resolution, with corresponding renderings indicated by the figure numbers in parentheses.*

ε	CPU 2^{-11}	GPU 2^{-11}	2^{-11}	Converged ε
Arithmetic	IA	IA	RAA	IA or RAA
Sphere	15	75	147	165 /RAA /2^{-10}
Steiner (6)	7.5	34	40	38 /RAA /2^{-12}
Mitchell (5)	5.2	16	58	60 /RAA /2^{-10}
Teardrop (7a)	5.5	102	115	121 /0AA /2^{-10}
4-bretzel (7c)	13	78	48	90 /IA /2^{-10}
Klein b. (7b)	11	30	110	101 /RAA /2^{-12}
Tangle (7d)	3.2	15	68	71 /RAA /2^{-10}
Decocube (9)	5.5	28	27	28 /IA /2^{-11}
Barth sex. (8l)	7.4	31	76	88 /RAA /2^{-10}
Barth dec. (8r)	0.92	4.9	15.6	15.6 /RAA /2^{-11}
Superquadric	18	119	8.3	108 /IA /2^{-12}
icos.csg (10l)	1.8	13.3	–	13.3 /IA /2^{-11}
sesc.csg (10r)	1.6	8.9	–	7.2 /IA /2^{-13}
sin.blob (1)	0.71	6.0	–	6.0 /IA /2^{-12}
Cloth (11l)	2.2	38	–	44 /IA /2^{-9}
Water (11r)	2.2	37	–	44 /IA /2^{-9}

The CPU SIMD algorithm is benchmarked on a four-core 2.33 GHz Intel Xeon desktop, using only IA. The GPU algorithm runs on an NVIDIA 8800GTX; results are shown with both IA and RAA. Results in these first three columns are evaluated with common $\varepsilon = 2^{-11}$; the last column labelled 'converged ε' shows performance at the highest ε yielding a correctly converged visual result, using either IA or RAA on the GPU.

SSE method on a 4-core Xeon 2.33 GHz CPU workstation. Frame rate is determined both by the bound tightness of the chosen inclusion extension, and the computational cost of evaluating it. In practice, the order of the implicit form has little impact on performance. Forms of these implicits can be found in the Appendix.

6.1.1. *IA versus RAA*

For typical functions with fairly low-order coefficients and moderate cross-multiplication of terms, RAA is generally 1.5–2× faster than IA. For functions with high bound overestimation, such as those involving multiplication of large polynomial terms (e.g. the Barth surfaces) or Horner forms, RAA is frequently three to four times faster. Conversely, thanks to an efficient inclusion rule for integer powers, IA remains far more efficient for superquadrics, as evident in Table 1. As explained in Section 5.4, IA is currently required for extensions of division, transcendentals and fractional powers.

6.1.2. *Error and quality*

As seen in Equation 16, a global user-specified ray-length precision ε is used to determine a per-ray maximum bisection depth $d_{\max}$. If a candidate ray interval $\bar{t}$ contains a zero, then the actual error is

$$\varepsilon_{\text{actual}} \leq ||\bar{t}|| = 2^{-d_{\max}} \leq \varepsilon. \tag{17}$$

This effectively specifies an upper bound on the absolute error in ray space t; by scaling by the magnitude of the ray segment over Ω, $\|p_{\text{exit}} - p_{\text{enter}}\|$, we normalize to bound relative error in world space. Our application also allows the user to specify a tolerance δ, which halts bisection only when the width of the interval $\|F\| < \delta$. This would seem a more adaptive way of guaranteeing convergence, as bisection proceeds until the interval width is sufficiently small to better guarantee existence (or non-existence) of a root. However, range interval width varies widely by function, and is more difficult for the user to gauge than the domain-space ε.

Choice of appropriate ε depends greatly on the implicit in question. For most of our examples, $\varepsilon = 2^{-11}$ yields a topologically correct rendering, and thus is suitable as a default. Figure 5 shows the impact of precision ε, controlling relative error, on the Mitchell and Barth decic surfaces, both examples with particularly high bound overestimation and sensitivity to low precision. RAA generally converges far more quickly than IA, given lesser-bound overestimation at low ε. In addition, refining ε has lesser impact on frame rate once RAA has effectively converged. Finally, we note that increasing ε generates progressively tighter convex hulls around the ideal surface at $\varepsilon = 0$.

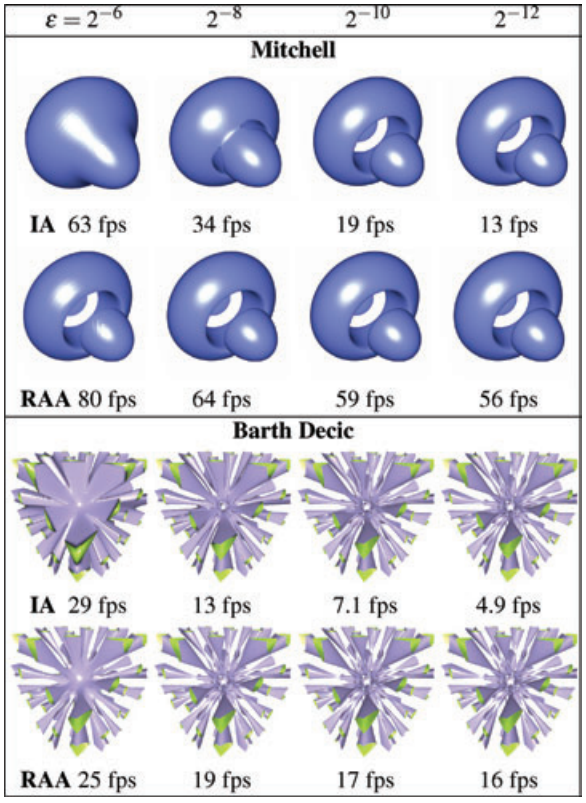

Figure 5: *Rendering the Mitchell (top) and Barth Decic (bottom) functions, with IA and RAA at various ε.*

6.1.3. *Algorithm coherence and performance*

Table 2 shows the relative performance of various algorithms on the Mitchell and Barth Decic functions shown in Figure 5, at $\varepsilon = 2^{-11}$ and $\varepsilon = 2^{-22}$. Our suggested implementations (also used in Table 1) are shown in boldface. The efficiency of both CPU (Section 4) and GPU (Section 5) algorithms depends on exploitation of SIMD coherence. The CPU SSE algorithm benefits from explicit spatial coherence, as shown in Figure 4(b). With the *t*-marching method in SSE (Figure 4(a)), rays in the same packet can fall out of lockstep, destroying coherence. Conversely, the GPU algorithm requires more general instruction-level coherence, with a minimum of used registers. A modification of the GPU algorithm to march along the major **K**-axis yielded noticeable performance decrease. We also note that both the SSE CPU and GPU implementations of the Mitchell [Mit90] algorithm (employing IA followed by standard numerical root refinement) perform far worse than naïve bisection, particularly at higher ε. This can be attributed to the high cost of evaluating the gradient interval, and both worse instruction-level coherence on the GPU and spatial coherence in the SSE CPU algorithm. Though difficult to fairly evaluate on the GPU, our experimentation with SSE versions of other quasi-Newton methods such as Interval Newton method and [CHMS00] empirically suggested far worse results. However, these algorithms could prove desirable if efficiently mapped to a SIMD architecture.

Table 2: *Performance of various algorithms on the Mitchell and Barth decic functions, using interval arithmetic only*

Function	Mitchell		Barth Decic	
ε	2^{-11}	2^{-22}	2^{-11}	2^{-22}
CPU SSE				
t-bisection	5.0	1.0	0.90	0.061
K-bisection	**5.1**	**1.2**	**0.92**	**0.18**
Mitchell	0.54	0.22	0.19	0.036
GPU (IA)				
t-bisection	**16**	**6.2**	**4.9**	**1.4**
K-bisection	11	5.6	4.4	1.1
Mitchell	3.9	1.0	1.1	0.29

Values in bold are results from our preferred algorithmic variant on the CPU and GPU, respectively.

Figure 6: Fine feature visualization in the Steiner surface. *Left to right: shading with depth peeling and gradient magnitude colouration; close-up on a singularity with IA at $\varepsilon = 2^{-18}$; and with RAA at the same depth.*

6.1.4. *Feature reproduction and robustness*

As it entails more floating-point computation than IA, RAA has worse numerical conditioning, particularly with smaller ε. Figure 6 illustrates the challenge in robustly ray tracing the Steiner surface with IA and AA. Both inclusion methods identify the infinitely thin surface regions at the axes, but a small $\varepsilon < 2^{-18}$ is required for correct close-up visualization of these features. AA yields a tighter contour of the true zero-set than IA, but with some speckling. Nonetheless, both IA and RAA yield more robust results than non-inclusion ray tracing methods [LB06] on the Steiner surface, or than inclusion-based extraction [PLLdF06] on the teardrop (Figure 7(a)).

6.2. Shading modalities

As our algorithm relies purely on ray tracing, we can easily support per-pixel lighting models and multi-bounce effects, many of which would be difficult with rasterization

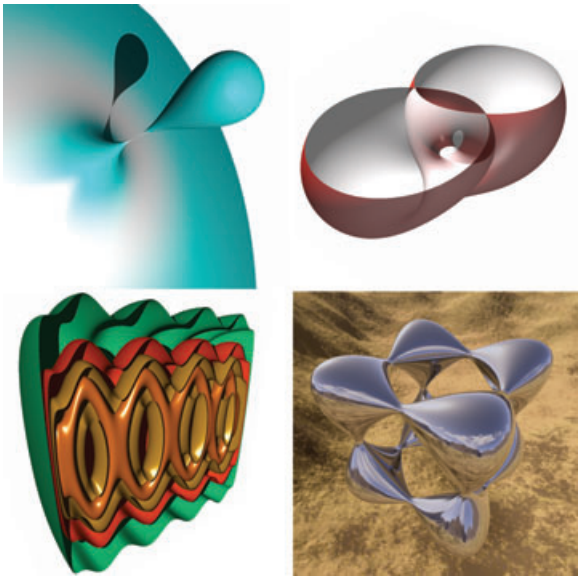

Figure 7: *Shading Effects. Top left to bottom right: (a) shadows on the teardop (40 fps); (b) transparency on the klein bottle (41 fps); (c) shadows and multiple isovalues of the 4-Bretzel (18 fps); and (d) the tangle with up to six re-flection rays (44 fps).*

(Figure 7). We briefly describe the implementation of these modalities, and their impact on performance.

Shadows: Non-recursive secondary rays such as shadows are straightforward to implement. Within the main fragment program, after a successfully hit traversal, we check whether $\vec{N} \cdot \vec{L} > 0$, and if so, perform traversal with a shadow ray. To ensure we do not hit the same surface, we cast the shadow from the light to the hit position, and use their difference to reparameterize the ray so that $\bar{t} = [0, 1]$, as for primary rays. Shadows often entail around 20–50% performance penalty. One can equally use a coarser precision for casting shadow rays than primary rays. RAA is sufficiently accurate for secondary rays even at $\varepsilon > 0.01$, which can decrease the performance overhead to 10–30%.

Transparency: Transparency is also useful in visualizing surfaces, particularly functions with odd connectivity or disjoint features. With ray tracing, it is simple to implement front-to-back, order-independent transparency, in which rays are only counted as transparent if a surface behind them exists. Our implementation lets the user specify the blending opacity, and casts up to four transparent rays. This costs around $3\times$ as much as one primary ray per pixel.

Multiple isosurfaces: One may equally use multiple isovalues to render the surface. This is significantly less expensive than evaluating the CSG object of multiple surfaces, as the implicit extension need only be evaluated once for the surface. The rejection test then requires that *all* of those isovalues miss. At hit registration, we simply determine which of those isovalues hit, and flag the shader accordingly to use different surface colours. With no other effects, multiple isovalues typically entail a cost of anywhere from 10% to 40%.

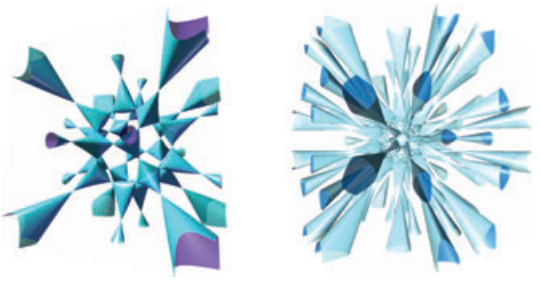

Figure 8: *The Barth sextic and decic surfaces.*

Reflections: Reflections are a good example of how built-in features of rasterization hardware can be seamlessly combined with the implicit ray tracing system. Looking up a single reflected value from a cubic environment map invokes no performance penalty. Tracing multiple reflection rays in an iterative loop is not significantly more expensive (20–30%), and yields clearly superior results (Figure 7(d)).

6.3. Applications

Mathematical visualization: The immediate application of this system is a graphing tool for mathematically interesting surface forms in 3D and 4D. Ray tracing ensures view-dependent visualization of infinitely thin features, as in the teardrop and Steiner surfaces. It is similarly useful in rendering singularities – Figure 8 shows the Barth sextic and decic surfaces, which contain the maximum number of ordinary double points for functions of their respective degrees in $\mathbb{R}^3$.

Interpolation, morphing and blending: Implicit forms inherently support blending operations between multiple basis functions. Such forms need only be expressed as an arbitrary 4D implicit $f\ (x,y,z,w)$, where w varies over time. As ray tracing is performed purely on-the-fly with no precomputation, we have great flexibility in dynamically rendering these functions. The blending function itself can operate on multiple kernels, and be of arbitrary form. Figure 9 shows morphing between a decocube and a sphere by interpolating a sigmoid convolution of those kernels.

Constructive solid geometry: Multiple-implicit CSG objects can accomplish similar effects to product surfaces and sigmoid blending, but with C^0 trimming. Unions and intersections of functions can be expressed natively using min and max operators, which are well defined for both interval and affine forms. However, this inevitably requires evaluation of all sides of a compound expression. A more

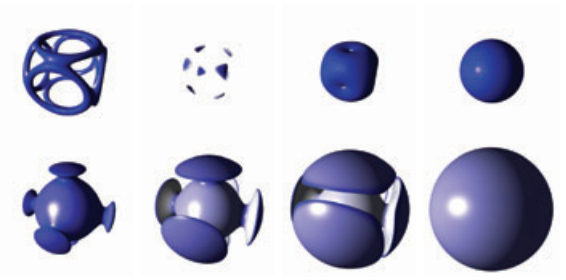

Figure 9: *Blending between a sigmoid convolution of a decocube and a sphere, with interpolation and extrapolation phases, running at 33–50 fps.*

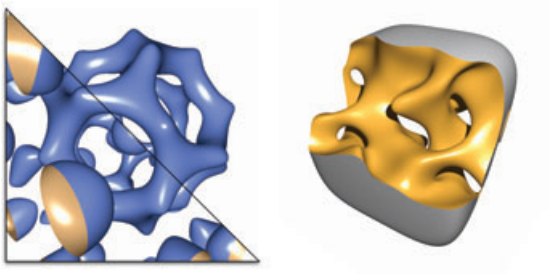

Figure 10: *CSG using inequalities on 3-manifold solids.*

efficient approach employs 3-manifold level-sets, or inequality operations on CSG solids, as conditions over an implicit or set or implicits. Given an implicit $f(\omega)$ and a condition $g(\omega)$, inclusion arithmetic allows us to verify $g_+ = \{\underline{g}(\omega) \geq 0\}$ or $\underline{g} = \{g(\omega) \leq 0\}$, given the interval form of the inclusion extension G over an interval domain $\omega \subseteq \Omega$. Then, one can render $f \cap g_+$ or $f \cap g_-$ for arbitrary level sets of g; and during traversal identify which surface is intersected. In the case of union, only the first condition need be evaluated if it contains a zero. Solid conditions are evaluated independently as boolean expressions; by determining which level sets are intersected inside the traversal, we can shade components differently as desired (Figure 10).

Procedural geometry: Implicits have historically been non-intuitive and unpopular for modelling large-scale objects. However, the ability to render dynamic surfaces and natural phenomena using combinations of known closed-form expressions could prove useful in modelling small-scale and dynamic features. As expressions, for example, define closed-form solutions of simple wave equations for modelling water and cloth (Figure 11). Previous applications of implicit hypertextures focused on blended procedural noise functions [GM07, PH89]. Recently, implicits based predominately on generalized sinusoid product forms similar to that in Figure 1 have been used within some modelling communities [k3d]. Arbitrary implicits are intriguing in their flexibility, and ray tracing promises the ability to dynamically render

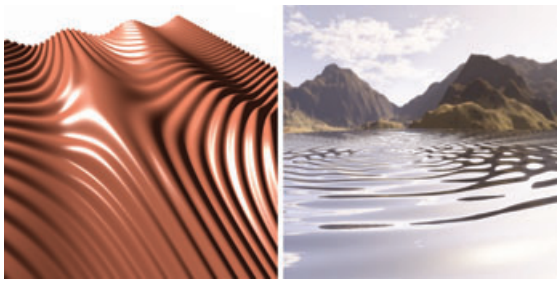

Figure 11: *Sinusoid procedural geometry for dynamic simulation of cloth and water. With IA, these surfaces render at 38 and 37 fps, respectively.*

entire new classes of procedural geometries, independently from any polygonal geometry budget.

7. Conclusion

We have demonstrated a fast, robust and general algorithm for rendering implicits of arbitrary form, using interval and AA. On both the CPU and GPU, the key to performance lies in optimization of the interval bisection algorithm. Coherent traversal using SIMD vector instructions; and a stackless fractional modulus traversal algorithm, aid the efficiency of CPU and GPU algorithms, respectively. We also demonstrate a correct inclusion-preserving RAA, which exhibits improved bound estimation and performance compared to IA.

IA/AA methods require more computation than approaches involving point sampling [Han83], though those methods are not generally robust. Inclusion methods may be slower than approximating methods (e.g. [LB06]), but more accurately render the original form. Finally, methods for rendering implicits formulated as distance functions [Har96] may be competitive as well. A comprehensive comparison of optimized implementations of these methods would be useful. Also, while robust per-ray, our system ignores aliasing issues on boundaries and sub-pixel features. To robustly reconstruct the surface between pixels would require supersampling and ideally beam tracing.

Many extensions to this implementation would be useful. Further development of approximating regression operations for RAA could allow for correct and fast affine extensions of transcendental functions and their compositions. Optimized implementation of the coherent **K**-marching method could perform better in a data-parallel SIMD setting such as CUDA; or on future wider SIMD vector hardware such as Intel Larrabee. More generally, the application front-end could be extended to support point, mesh or volume data, which could then be reconstructed by arbitrary implicit filters. Scalable rendering of complex objects featuring multiple piecewise implicits with CSG operators is also important future

work. Finally, though applied here to general implicits, inclusion methods could potentially be employed in rendering arbitrary parametric or free-form surfaces.

Acknowledgements

This work was supported by the German Research Foundation, the Deutsche Forschungsgemeinschaft (DFG) through the University of Kaiserslautern International Research Training Group (IRTG) 1131; as well as the US Department of Energy through CSAFE grant W-7405-ENG-48, the National Science Foundation under CISE grants CRI-0513212, CCF-0541113, and SEII-0513212 and NSF-CNS 0551724. It was also supported by the US Department of Energy SciDAC VACET, Contract No. DE-FC02-06ER25781 (SciDAC VACET). Additional thanks to Warren Hunt, Bill Mark, Ingo Wald, Steven Parker, Alex Reshetov and Jim Hurley for their insights, and to Intel Corporation and NVIDIA for support and equipment.

Appendix A: Reference Implicits

Table A1: *Formulas of test surfaces used in Table 1.*

Sphere	$x^2 + y^2 + z^2 - r^2$
Steiner	$x^2y^2 + y^2z^2 + x^2z^2 + xyz$
Mitchell	$4(x^4 + (y^2 + z^2)^2) + 17(x^2(y^2 + z^2) - 20(x^2 + y^2 + z^2)) + 17$
Teardrop	$\frac{x^5+x^4}{2} - y^2 - z^2$
4-bretzel 1	$\frac{1}{10}(x^2(1.21 - x^2)^2(3.8 - x^2)3 - 10y^2)^2 + 60z^2 - 2$
Klein bottle	$(x^2 + y^2 + z^2 + 2y - 1)((x^2 + y^2 + z^2 - 2y - 1)^2 - 8z^2) + 16xz(x^2 + y^2 + z^2 - 2y - 1)$
Tangle	$x^4 - rx^2 + y^4 - 5y^2 + z^4 - 5z^2 + 11.8$
Decocube	$((x^2 + y^2 - 0.82)^2 + (z^2 - 1)^2)((y^2 + z^2 - 0.82)^2 + (x^2 - 1)^2)((z^2 + x^2 - 0.82)^2 + (y^2 - 1)^2) - 0.02$
Barth sextic	$4(\tau^2 \times 2 - y^2)(\tau^2y^2 - z^2)(\tau^2z^2 - x^2) - (1 + 2\tau)(x^2 + y^2 + z^2 - 1)^2$ where τ is the golden ratio, $\frac{1+\sqrt{5}}{2}$
Barth decic	$8(x^2 - \tau^4y^2)(y^2 - \tau^4z^2)(z^2 - \tau^4 \times x^2)(x^4 + y^4 + z^4 - 2x^2 \times 2y^2 - 2x^2 \times 2z^2 - 2y^2z^2) + (3 + 5\tau)(x^2 + y^2 + z^2 - w^2)^2$ $(x^2 + y^2 + z^2 - (2 - \tau)w^2)^2w^2$, $\tau = \frac{1+\sqrt{5}}{2}$
Superquadric	$x^{500} + \frac{1}{2}\lvert y\rvert^{35} + \frac{1}{2}z^4 - 1$
icos.csg	$ic(x,y,z) = 2 - (\cos(x + \tau y) + \cos(x - \tau y) + \cos(y + \tau z) + \cos(y - \tau z) + \cos(z - \tau x) + \cos(z + \tau x))$, $\tau = \frac{1+\sqrt{5}}{2}$ CSG condition (on inclusion intervals): $(0 \in ic)$ and $sphere_{\text{inner}} < 0$ and $sphere_{\text{outer}} > 0$
sesc.csg	CSG of superellipsoid (se) and sinusoid convolution (sc) $sc(x, y, z) = x^6 + \frac{1}{2}(y^4 + z^4)^4 - 20$ $sc(x,y,z) = xy + \cos(z) + 1.741\sin(2x)\sin(z)\cos(y) + \sin(2y)\sin(x)\cos(z)$ $+ \sin(2z)\sin(y)\cos(x) - \cos(2x)\cos(2y)$ $+ \cos(2y)\cos(2z) + \cos(2z)\cos(2x) + 0.05$ CSG condition (on inclusion intervals): $((sc > 0)$ and $(0 \in se))$ or $((se < 0)$ and $(0 \in sc))$
sin.blob	$1 + r_1(y + w) + \cos(r_2z) + 4(\sin(4r_0r_2x)\sin(r_0z)\cos(r_1y)$ $+ \sin(2r_0r_1y)\sin(r_2x)\cos(r_2z) + \sin(2r_2z)\sin(r_1y)\cos(r_2x))$ $- (\cos(2r_2x)\cos(2r_0y) + \cos(2r_0r_1y)\cos(2r_2z) + \cos(2r_2z)\cos(2r_2x))$ where $r_0 = 0.104$, $r_1 = 0.402$, $r_2 = -0.347$
Cloth	$y - 0.5\sin(x + 3w) - 0.1(1 + 0.1\sin(xz))\cos(z + 3w)$ where $w = [0,2\pi]$ is a time-dependent variable
Water	$\sin(\sqrt{((x + r_1)^2 + z^2)} - w)/(10 + \sqrt{((x + r_1)^2 + z^2)}) + \sin(\sqrt{((x - r_1)^2 + z^2)} - w + r_0)/(10 + \sqrt{((x - r_1)^2 + z^2)}) +$ $\sin(2\sqrt{((z - r_1)^2 + x^2)} - w - r_0)/(10 + \sqrt{((z - r_1)^2 + x^2)}) - \frac{y}{2}$ where $r_0 = 2.736$, $r_1 = 15$, $r_2 = -0.830746$ and w is time-dependent.

References

[Blo94] Bloomenthal J.: An implicit surface polygonizer. In *Graphics Gems IV*, 1994, New York, Academic Press, pp. 324–349.

[BSP06] Bigler J., Stephens A., Parker S.: Design for parallel interactive ray tracing systems. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (2006), pp. 187–195.

[CHMS00] Capriani O., Hvidegaard L., Mortensen M., Schneider T.: Robust and efficient ray intersection of implicit surfaces. *Reliable Computing 6* (2000), 9–21.

[CS93] Comba J. L. D., Stolfi J.: Affine arithmetic and its applications to computer graphics. In *Proc. VI Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'93)* (1993), pp. 9–18.

[dCJdFG99] de Cusatis Junior A., de Figueiredo L., Gattas M.: Interval methods for raycasting implicit surfaces with affine arithmetic. In *Proceedings of XII SIBGRPHI* (1999), pp. 1–7.

[dTLP07] DE TOLEDO R., LEVY B., PAUL J.-C.: Iterative methods for visualization of implicit surfaces ongpu. In *ISVC, International Symposium on Visual Computing* (Lake Tahoe, Nevada/California, November 2007), Lecture Notes in Computer Science, SBC – Sociedade Brasileira de Computacao, Springer. pp. 598–609.

[FP08] FRYAZINOV O., PASKO A.: Interactive ray shading of FRep objects. In *WSCG 2008 Communications Papers Proceedings* (2008), pp. 145–152.

[FSSV06] FLOREZ J., SBERT M., SAINZ M., VEHI J.: Improving the interval ray tracing of implicit surfaces. In *Lecture Notes in Computer Science* (2006), Vol. *4035*, pp. 655–664.

[GM07] GAMITO M. N., MADDOCK S. C.: Ray casting implicit fractal surfaces with reduced affine arithmetic. *The Visual Computer 23*, 3 (2007), 155–165.

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007* (September 2007), pp. 113–118.

[Han83] HANRAHAN P.: Ray tracing algebraic surfaces. In *SIGGRAPH '83: Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1983), ACM Press, pp. 83–90.

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

[HS98] HEIDRICH W., SEIDEL H.-P.: Ray-tracing procedural displacement shaders.In *Proceedings of Graphics Interface* (1998), pp. 8–16.

[HSS*05] HADWIGER M., SIGG* C., SCHARSACH H., BUHLER K., GROSS* M.: Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum 24*, 3 (2005), 303–312.

[k3d] K3dsurf: The K3DSurf project. Available at http://k3dsurf.sourceforge.net/.

[KB89] KALRA D., BARR A. H.: Guaranteed ray intersections with implicit surfaces. In *SIGGRAPH '89: Proceedings of the 16th Annual Conference on Computergraphics and Interactive Techniques* (New York, NY, USA, 1989), ACM Press, pp. 297–306.

[KHW07a] KNOLL A., HANSEN C., WALD I.: *Coherent Multiresolution Isosurface Ray Tracing*. Tech. Rep. UUSCI-2007–001, SCI Institute, University of Utah, 2007. (To appear in *The Visual Computer*).

[KHW*07b] KNOLL A., HIJAZI Y., WALD I., HANSEN C., HAGEN H.: Interactive ray tracing of arbitrary implicits with simd interval arithmetic. In *Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 11–18.

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

[Mes02] MESSINE F.: Extentions of affine arithmetic: Application to unconstrained global optimization. *Journal of Universal Computer Science 8*, 11 (2002), 992–1015.

[MGW05] MEYER M. D., GEORGEL P., WHITAKER R. T.: Robust particle systems for curvature dependent sampling of implicit surfaces. In *SMI '05: Proceedings of the International Conference on Shape Modeling and Applications 2005 (SMI' 05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 124–133.

[Mid65] MIDONICK H. O.: *The Treasury of Mathematics*. Philosophical Library, 1965.

[Mit90] MITCHELL D.: Robust ray intersection with interval arithmetic. In *Proceedings on Graphics Interface 1990* (1990), pp. 68–74.

[Moo66] MOORE R. E.: *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 21*, 3 (2002), 703–712.

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum 26*, 3 (2007). (Proceedings of Eurographics).

[PH89] PERLIN K., HOFFERT E. M.: Hypertexture. In *SIGGRAPH '89: Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1989), ACM Press, pp. 253–262.

[PLLdF06] PAIVA A., LOPES H., LEWINER T., DE FIGUEIREDO L. H.: Robust adaptive meshes for implicit surfaces. In *19th Brazilian Symposium on Computer Graphics and Image Processing* (2006), pp. 205–212.

[RVdF06] ROMEIRO F., VELHO L., DE FIGUEIREDO L. H.: Hardware-assisted rendering of CSG models. In *SIBGRAPI* (2006), pp. 139–146.

[SECG03] SANJUAN-ESTRADA J. F., CASADO L. G., GARCIA I.: 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*. (2003), pp. 35–42.

[SK01] STOLTE N., KAUFMAN A.: Voxelization of implicit surfaces using interval arithmetics. *Graphical Models 63*, 6 (2001), 387–412.

[Tot85] TOTH D. L.: On ray tracing parametric surfaces. In *SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1985), ACM Press, pp. 171–179.

[VKZM06] VARADHAN G., KRISHNAN S., ZHANG L., MANOCHA D.: Reliable Implicit surface polygonization using visibility mapping. In *SGP '06: Proceedings of the Fourth Eurographics Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, Switzerland, 2006), Eurographics Association, pp. 211–221.

[vW85] VAN WIJK J.: Ray tracing objects defined by sweeping a sphere. *Computers & Graphics 9* (1985), 283–290.

[WBS02] WALD I., BENTHIN C., SLUSALLEK P.: *OpenRT – A Flexible and Scalable Rendering Engine for Interactive 3D Graphics*. Tech. rep., Saarland University, 2002. Available at http://graphics.cs.unisb.de/Publications.

[WH94] WITKIN A. P., HECKBERT P. S.: Using particles to sample and control implicit surfaces. In *SIGGRAPH '94: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1994), ACM Press, pp. 269–277.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493. (Proceedings of ACM SIGGRAPH).

[WMW86] WYVILL G., MCPHEETERS C., WYVILL B.: Data structure for soft objects. *The Visual Computer 2* (1986), 227–234.

[WQ80] WOODWARK J., QUINLAN K.: The derivation of graphics from volume models by recursive subdivision of object space. In *Proceedings of Computer Graphics'80 Conference, Brighton, UK* (1980), pp. 335–343.