Scientitfic Computing and Imaging Institute, University of Utah. Technical Report No UUSCI-2007-014

Fast and Robust Ray Tracing of General Implicits on the GPU

Aaron Knoll*SCI Institute, University of Utah, IRTG
Younis Hijazi†University of Kaiserslautern, IRTG
Andrew Kensler‡SCI Institute, University of Utah
Mathias Schott§SCI Institute, University of Utah
Charles Hansen¶SCI Institute, University of Utah
Hans Hagen‖University of Kaiserslautern

Abstract

Existing methods for rendering arbitrary implicit functions are limited, either in performance, correctness or flexibility. Ray tracing methods in conjunction with an inclusion algebra such as interval arithmetic (IA) or affine arithmetic (AA) have historically proven robust and flexible, but slow. In this paper, we present a new stackless ray traversal algorithm optimized for modern graphics hardware, and a correct inclusion-preserving reduced affine arithmetic (RAA) suitable for fragment shader languages. Shader metaprogramming allows for immediate and automatic generation of functions and their interval or affine extensions, enhancing user interaction. Ray tracing lends itself to multi-bounce effects, such as shadows and depth peeling, which are useful modalities for visualizing complicated implicit functions. With this system, we are able to render even complex implicits correctly, in real-time at high resolution.

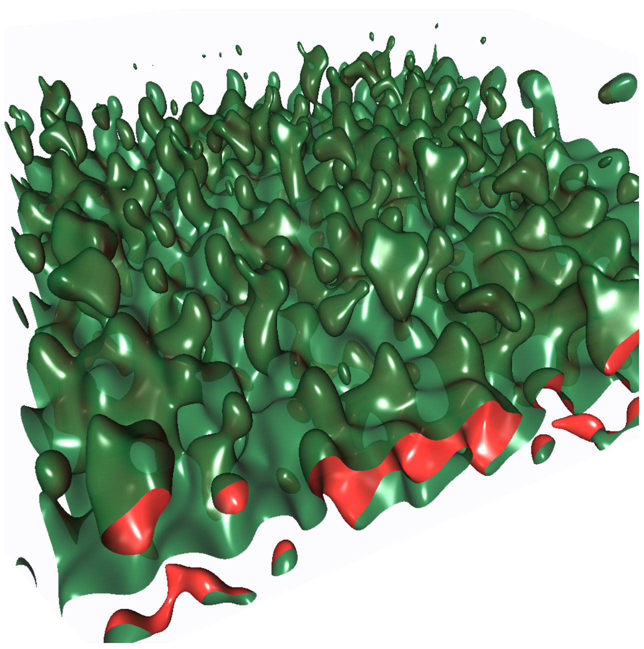
Figure 1: An animated sinusoid-kernel surface. Ray-traced directly on fragment units, no new geometry is introduced into the rasterization pipeline. IA/AA methods ensure robust rendering of any inclusion-computable implicit.

*e-mail: knolla@cs.utah.edu
†e-mail: hijazi@rhrk.uni-kl.de
‡e-mail: kensler@cs.utah.edu
§e-mail: mschott@cs.utah.edu
¶e-mail: hansen@cs.utah.edu
‖e-mail: hagen@informatik.uni-kl.de

1 Introduction

To render implicits, 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 yield isotropic geometry 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 [22] is increasingly common, but often algorithmically inefficient.

Ray tracing methods for implicits have historically sacrificed either speed, correctness or flexibility. Piecewise algebraic implicits have been rendered in real-time on the GPU using Bezier decompositions [14], but approximating methods do not render arbitrary functions directly, nor always robustly. Inclusion methods, such as interval arithmetic (IA) or affine arithmetic (AA), are considered the most general and robust, but traditionally the slowest. Recently, arbitrary implicits were rendered interactively on the CPU by optimizing IA ray tracing with SIMD vector instructions, and by making practical assumptions about the numerical precision needed for correct visualization [13]. Though that system is over two orders of magnitude faster than its predecessors, it is still only roughly interactive on current CPU hardware. A GPU implementation is desirable for its superior computational throughput, and use in conjunction with the conventional rasterization pipeline.

The major contributions of this paper are a new iterative spatial traversal algorithm for implicit intersection; and an efficient implementation of a correct reduced affine arithmetic (RAA) suitable for shader languages. Together, these allow real-time rendering of complex implicit functions. Shader metaprogramming allows users to design implicits and procedural hypertextures flexibly, with immediate results and full support for dynamic 4D surfaces. The ray tracing algorithm enables multi-bounce effects to be computed interactively without image-space approximations, enabling effects such as translucent depth peeling 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 [31] or Bloomenthal polygonization [1] 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. [19] detail a robust algorithm based on dual marching cubes, using interval arithmetic in conjunction with geometric oracles. Varadhan et al. [28] 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. Splatting uses view-dependent point sampling of an implicit reconstruction of point cloud data [24]. Dynamic particle sampling methods for implicits have been demonstrated by Witkin & Heckbert [29] and extended by Meyer et al. [16]. Slice-based GPU volume rendering, often in conjunction with ray casting, is a practical method of visualizing implicits [8].

2.2 Ray Tracing Implicits

Hanrahan [9] proposed a general but non-robust point-sampling algorithm using Descartes' rule of signs to isolate roots. Van Wijk [27] implemented a recursive root bracketing algorithm using Sturm sequences, suitable for differentiable algebraics. Kalra & Barr [12] devised a method of rendering a subclass of algebraic surfaces with known Lipschitz bounds. Hart [10] proposed a robust method for ray tracing algebraics by defining signed distance functions from an arbitrary point to the surface. More recently, Loop & Blinn [14] implemented an extremely fast GPU ray caster approximating implicits with piecewise Bernstein polynomials. Romeiro et al. [23] proposed a hybrid GPU/CPU technique for casting rays through constructive solid geometry (CSG) trees of implicits. De Toledo et al. [5] demonstrated interactive ray casting of cubics and quartics using standard iterative numerical methods on the GPU.

Ray Tracing with Interval and Affine Arithmetic: Toth [26] first applied interval arithmetic to ray tracing parametric surfaces, in determining an initial convex bound before solving a nonlinear system. Mitchell [17] ray traced implicits using recursive IA bisection to isolate monotonic ray intervals, in conjunction with standard bisection as a root refinement method. De Cusatis Junior et al. [4] used standard affine arithmetic in conjunction with recursive bisection. Sanjuan-Estrada et al. [25] compared performance of two hybrid interval methods with Interval Newton and Sturm solvers. Florez et al. [6] proposed a ray tracer that antialiases surfaces by adaptive sampling during interval subdivision. Gamito and Maddock [7] proposed reduced affine arithmetic 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. [13] implemented a generally interactive interval bisection algorithm for arbitrary implicits 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 Implicits

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

$$f(x,y,z) = 0 \qquad (1)$$

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

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

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

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

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. General implicits 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 Fig. 2, "point sampling" using the rule of signs (e.g. [9]) 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 affine arithmetic are among the most robust and general. Historically, they have also been among the slowest, due to inefficient implementation and impractical numerical assumptions.

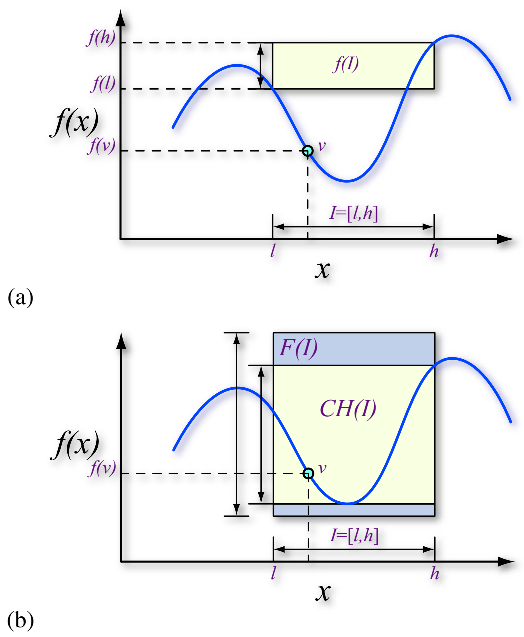

Figure 2: *The inclusion property.* (a) When a function f is non-monotonic on an interval I, evaluating the lower and upper components of a domain interval is insufficient to determine a convex hull over the range. This is not the case with an inclusion extension F (b), which, when evaluated, will enclose all minima and maxima of the function within that interval. Ideally, $F(I)$ is equal or close to the bounds of the convex hull, $CH(I)$.

3.2 Interval Arithmetic and Inclusion

Interval arithmetic (IA) was introduced by Moore [18] as an approach to bounding numerical rounding errors in floating point computation. The same way classical arithmetic operates on real numbers, interval arithmetic 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:

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

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

Moore's fundamental theorem of interval arithmetic [18] states that for any function f defined by an arithmetical expression, the corresponding interval evaluation function F is an *inclusion function* of f:

$$F(\underline{\overline{x}}) \supseteq f(\underline{\overline{x}}) \ = \ \{f(x) \mid x \in \underline{\overline{x}}\} \qquad (6)$$

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

3.3 Affine Arithmetic

Affine arithmetic (AA) was developed by Comba & Stolfi [3] 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 a parallelogram (Fig. 3).

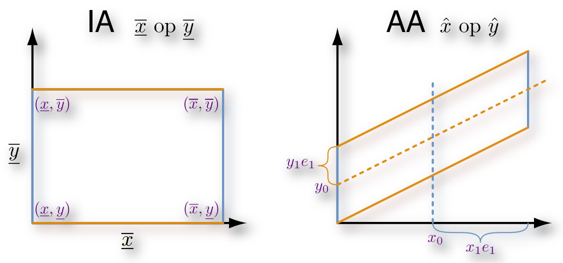
IA $\overline{\underline{x}}$ op $\overline{\underline{y}}$
$(\underline{x},\overline{y})$
$(\overline{x},\overline{y})$
$\overline{\underline{y}}$
$(\underline{x},\underline{y})$
$(\overline{x},\underline{y})$
$\overline{\underline{x}}$
AA $\hat{x}$ op $\hat{y}$
y_1e_1
y_0
x_0
x_1e_1

Figure 3: Bounding forms of interval and affine arithmetic operations.

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:

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

and can equally be converted into an interval

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

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

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

Base affine operations in AA are as follows:

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

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.

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.

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 $\overline{\underline{t}} = [\underline{t}, \overline{t}]$ *possibly* contains our surface, we simply check if $0 \in F_t(\overline{\underline{t}})$. As a result, once the inclusion library is implemented, any function composed of its operators can be rendered robustly. To select domain intervals on which to evaluate the extension, one has a wide choice of interval numerical methods [17, 2, 25]. Empirical results [13, 7, 4] suggest that simple bisection works best, particularly at coarser precision ε. In practice, evaluating a gradient extension is expensive, and higher-order convergent methods resort to bisection on non-monotonic regions. Moreover, high numerical precision is seldom required for accurate visualization [13].

4 Ray Tracing Implicits with IA and AA on the GPU

In many ways, modern shader languages such as Cg or GLSL allow for a far more graceful implementation than the optimized SSE C++ counterpart on the CPU. Thanks to this language flexibility, it is possible to design a full ray tracer within a fragment program. On-the-fly 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 [21]. 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 prohibitively expensive on a CPU, it is well-suited for the GPU architecture due to efficient division operations.

In implementing affine arithmetic to mitigate IA bound overestimation, it was immediately clear that a full array-based implementation of conventional AA would be impractical on the GPU. Though efficient, the reduced affine arithmetic method proposed by Gamito & Maddock [7] only preserves inclusion under specific circumstances. Fortunately, with modifications ensuring that the last error term is positive-definite, a formulation similar to that of Messine et al. [15] implements a correct inclusion for all compositions of AA operations. In adopting such an arithmetic, we implement a robust reduced AA suitable for ray tracing on the GPU. Particularly for complex implicits requiring cross-multiplication between interval entities, this yields more correct results at lower required precision than standard IA, and superior frame rates for most functions.

4.1 Application Pipeline

As input, the user must simply specify an implicit function, a domain $\Omega \subset \mathbb{R}^3$, and a termination precision ε. User-specified 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. The most complicated metaprogramming involves creating routines for function evaluation. Given an implicit function, we generally require two routines to be created within the shader: one evaluating the implicit f, and another evaluating an 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 provide "inline" Cg code, which can be useful in optimizing performance of implicits with repeated identical blocks of terms, and expressing special-case CSG models.

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.

4.2 Interval Arithmetic Library

Implementing an IA library is straightforward in Cg. Most operations employed in interval arithmetic (such as min and max) are highly efficient on the GPU, and swizzling allows for graceful SIMD computation (Algorithm 1). Transcendental functions are particularly efficient for both their floating-point and interval computations. Moderate integer powers are yet more efficient, thanks to unrolling multiplication chains via metaprogramming and the Russian peasants algorithm; and a bound-efficient IA rule for even powers.

Algorithm 1 Interval Arithmetic examples.

```
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) {
  const bool ic0 = (i.x <= 0 && i.y >= 0);
  return ( (i.x <= 0 && i.y >= 0) ?
           interval(-inf, inf) : 1/i.yx );
}
```

4.3 Reduced Affine Arithmetic Library

In implementing our RAA library on the GPU, we adopt a formulation similar to AF1 in Messine et al. [15], with changes to the absolute value bracketing that are mathematically equivalent but slightly faster to compute. In AF1, for some constant n a reduced affine form is given as:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i e_i + x_{n+1} e_{n+1} \tag{13}$$

Our arithmetic operations are then as follows:

$$\begin{aligned}
\mathbf{c} \pm \hat{x} &= (\mathbf{c} \pm x_0) + \sum_{i=1}^{n} x_i \varepsilon_i + |x_{n+1}|\varepsilon_{n+1} \\
\hat{x} \pm \hat{y} &= (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i)\varepsilon_i + (x_{n+1} + y_{n+1})\varepsilon_{n+1} \\
\mathbf{c} \times \hat{x} &= (\mathbf{c}x_0) + \sum_{i=1}^{n} \mathbf{c}x_i \varepsilon_i + |cx_{n+1}|\varepsilon_{n+1} \\
\hat{x} \times \hat{y} &= (x_0 y_0) + \sum_{i=1}^{n} (x_o y_i + y_0 x_i)\varepsilon_i + \\
&\qquad (|x_0 y_{n+1}| + |y_0 x_{n+1}| + \mathrm{rad}(\hat{x})\mathrm{rad}(\hat{y}))\varepsilon_{n+1}
\end{aligned} \tag{14}$$

We implemented this formulation 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 [15] was too high to be efficient. Examples of the float3 version are given in Algorithm 2.

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

Algorithm 2 Reduced Affine Arithmetic examples.

```
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_add(raf a, raf b){
  return a + b;
}
raf_mul{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;
}
```

4.4 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 NVIDIA 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 negligable 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 interval arithmetic for functions that require regression-approximation AA operators.

Algorithm 3 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_mul(it, dir.x));
  iy = raf_add(org.y, raf_mul(it, dir.y));
  iz = raf_add(org.z, raf_mul(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<24; 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
}
```

4.5 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.

Initialization: We begin by computing the exit point p_{exit} of the generated ray and the bounding box Ω. We reparameterize the ray as $\vec{r}(t) := \vec{p}_{enter} + t(\vec{p}_{exit} - \vec{p}_{enter})$. The interval $\underline{\bar{t}}$ 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 $\underline{\bar{t}}$, 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 user-specified precision ε. This is given by

$$d_{max} := \log_2\left(\frac{||\vec{p}_{exit} - \vec{p}_{enter}||}{\varepsilon}\right) \tag{15}$$

We initialize our depth $d = 0$, and distance increment, $t_{incr} = 0.5$. Now, recalling the bisection interval $\underline{\bar{t}}$, we set $\bar{t} := \underline{t} + t_{incr}$. We then perform the rejection test on this new $\underline{\bar{t}}$. If the test succeeds, we either hit the surface if we have reached $d = d_{max}$, or recurse to the next level by setting $t_{incr} := t_{incr}/2$, 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 basically decrement the depth (and update t_{incr}) as long as we have visited both "sides" of the bisection tree at the current depth. Conventionally, this algorithm is performed by caching an array in place of a recursion stack. As this is ineffective on the GPU, we note that we can perform a similar query by a floating-point modulus: checking if $(\underline{t} \,\%\, 2t_{incr} = 0)$. Currently on the G80, the fastest method proves to be performing division and examining the remainder. Back-recursion proceeds iteratively until either one side of the bisection has not yet been visited, or $d = -1$.

4.6 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. [13]. Multiple isovalues and transparency require modifications to the rejection test and hit registration, respectively, as discussed in Section 5.2. More generally, modifications to the traversal algorithm are simple to implement via "inline" implicit files (Section 4.1). We allow the user to directly program behavior of the rejection test, hit registration and shading. This is particularly useful in rendering special-case constructive solid geometry objects (Fig. 9).

4.7 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 [13].

5 Results

All benchmarks are measured in frames per second on an NVIDIA 8800 GTX, at 1024x1024 frame buffer resolution.

		CPU	GPU		
		$\varepsilon = 2^{-11}$			correct
function (fig)	**degree**	IA		RAA	IA/RAA
sphere	2	15	75	147	165
steiner (5)	4	7.5	34	40	38
mitchell (4)	8	5.2	16	58	60
teardrop (6a)	4	5.5	102	115	121
4-bretzel (6c)	12	13	78	48	90
klein b. (6b)	6	11	30	110	101
tangle (6d)	4	3.2	15	68	71
decocube (8)	4	5.5	28	27	28
barth sex. (7l)	6	7.4	31	76	88
barth dec. (7r)	10	0.9	4.9	15.6	15.6
superquadric	200	18	119	8.3	108
icos.csg (9l)	na	-	13.3	-	13.3
sq.csg (9r)	na	-	8.9	-	7.2
sin.blob (1)	na	-	6.0	-	6.0
cloth (10l)	na	-	38	-	44
water (10r)	na	-	37	-	44

Table 1: Single-ray casting performance in fps. We indicate the figure illustrating each function where available. We compare the SSE IA implementation of Knoll et al. [13] on four 2.33 GHz cores; and our IA and RAA implementations on the G80 GPU, using a common $\varepsilon = 2^{-11}$. The last column shows frame rate at the lowest ε yielding visually correct results, using either IA or RAA.

5.1 Performance

Table 1 shows base frame rates of a variety of implicits using single ray-casting and basic Phong shading. Performance on the NVIDIA 8800 GTX is up to 22× faster than the SSE method of Knoll et al. [13] on the 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 has little impact on performance. Equations for most functions can be found in [13] and [30].

IA vs RAA: For typical functions with fairly low-order coefficients and moderate cross-multiplication of terms, reduced affine arithmetic is generally $1.5-2\times$ faster than interval arithmetic. For functions with high bound overestimation, such as those involving multiplication of large polynomial terms (the Barth implicits) or Horner-like forms, RAA is frequently 3 to 4 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 4.4, IA is currently required for extensions of division, transcendentals, and fractional powers.

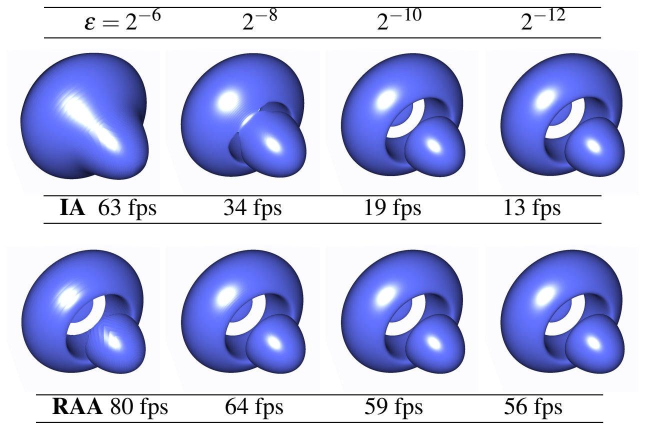

Figure 4: *IA (top) and RAA (bottom) at various ε*

Precision and Quality: Concerning visual quality and robustness, our findings for IA are generally in line with those of the CPU implementation of Knoll et al. [13]. For the analytic functions it supports, and particularly pathological cases for IA, RAA usually converges far more quickly to the correct solution, given lesser bound overestimation at low precision ε. In addition, refining ε has little impact on frame rate once RAA has effectively converged (Fig. 4).

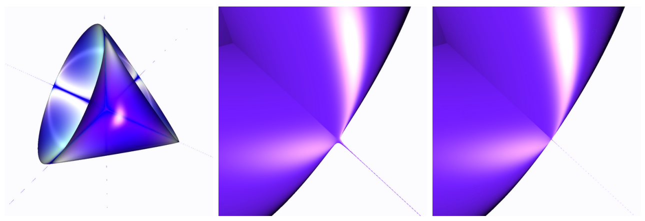

Figure 5: *Fine feature visualization in the Steiner surface.* Left to right: shading with depth peeling and gradient magnitude coloration; close-up on a singularity with IA at $\varepsilon = 2^{-18}$; and with RAA at the same depth.

Correctness and Robustness: As it entails more floating-point computation than IA, RAA has worse numerical conditioning. This is particularly noticeable with more precise ε. Fig. 5 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 fairly precise $\varepsilon < 2^{-18}$ is required for correct close-up visualization of these features. Affine arithmetic yields a tighter contour of the true zero-set than IA, but with some speckling artifacts. Nonetheless, both IA and RAA yield more robust results than non-inclusion ray tracing methods [14], or inclusion-based extraction [19] on the teardop (Fig. 6a).

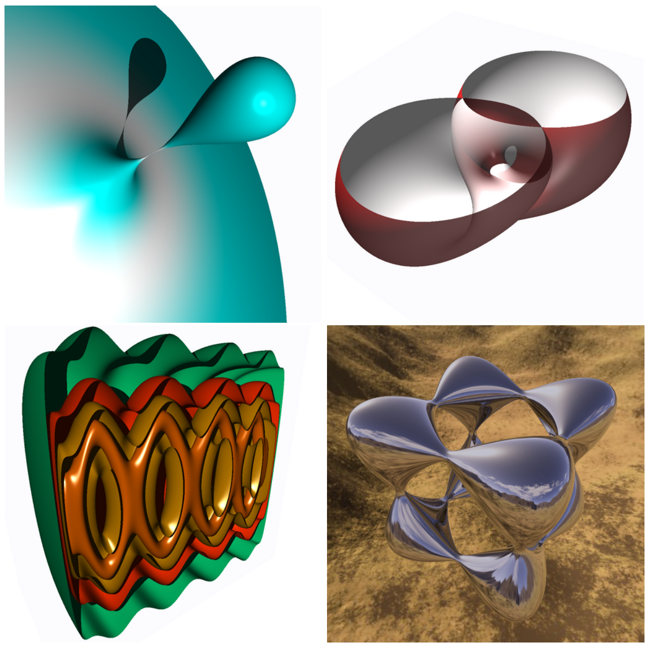

Figure 6: *Shading Effects.* Top left to bottom right: (a) shadows on the teardop (40 fps); (b) transparency on the klein bottle (41 fps); (c) shadows and multiple isovalues of the 4-Bretzel (18 fps); and (d) the tangle with up to six reflection rays (44 fps).

5.2 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 (Fig. 6). 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 $\underline{t} = [0,1]$, 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. With RAA, contour overestimation is seldom a problem even at $\varepsilon > .01$; this can decrease the performance overhead to $10-30\%$.

Transparency: Transparency is also useful in visualizing implicits, 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× 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 *any* of those isovalues hit. At hit registration, we simply determine which of those isovalues hit, and flag the shader accordingly to use different surface colors. With no other effects, multiple isovalues typically entail a cost of anywhere from $10-40\%$.

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 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 (Fig. 6d).

5.3 Applications

Mathematical Visualization: The immediate application of this system is a graphing tool for mathematically interesting implicits 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 – Fig. 7 shows the Barth sextic and decic surfaces, which contain the maximum number of ordinary double points for functions of their respective degrees in $\mathbb{R}^3$.

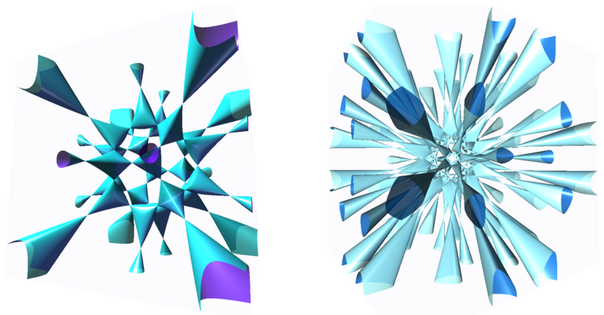
Figure 7: The Barth sextic and decic surfaces.

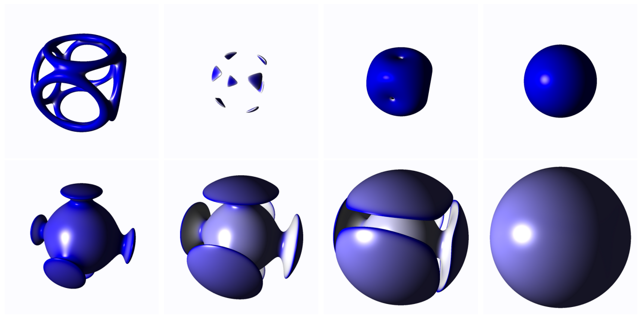
Figure 8: 4D sigmoid blending of the decocube and a sphere, with interpolation and extrapolation phases, running at $33-50$ fps.

Interpolation, Morphing and Blending: Implicits 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. Useful morphing methods include product implicits, linear interpolation between surfaces; and gaussian or sigmoid blending, shown in Fig. 8 between the decocube and the sphere.

Constructive Solid Geometry: Multiple-implicit CSG objects can accomplish similar effects to product surfaces and sigmoid blending, but with C^0 trimming. In particular, CSG intersection allows us to specify 3-manifold level sets as arbitrary conditions over an implicit or set of implicits. Given an implicit $f(\omega)$ and a condition $g(\omega)$, inclusion arithmetic allows us to verify $g_+ = \{\underline{g}(\omega) > 0\}$ or $g_- = \{\overline{g}(\omega) < 0\}$, 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. By determining which level sets are intersected inside the traversal, we can shade components differently as desired (Fig. 9).

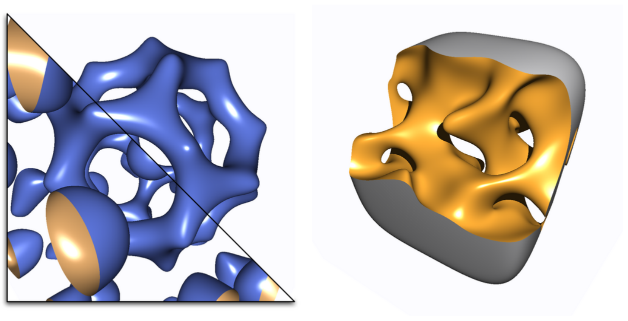
Figure 9: CSG surfaces using level-set conditions.

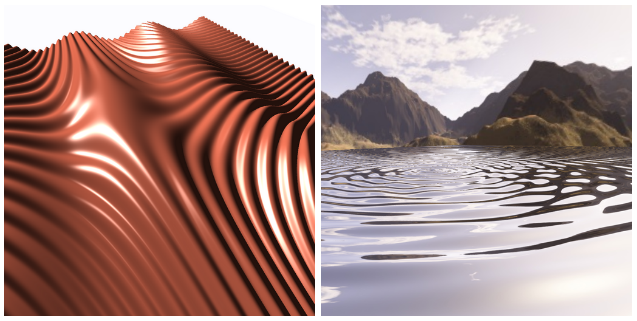
Figure 10: Sinusoid procedural geometry for dynamic simulation of cloth and water. With IA, these surfaces render at 38 and 37 fps respectively.

Procedural Geometry: Implicits have historically been nonintuitive and unpopular for modeling large-scale objects. However, the ability to render dynamic surfaces and natural phenomena using combinations of known closed-form expressions could prove useful in modeling small-scale and dynamic features. Sinc expressions, for example, define closed-form solutions of simple wave equations for modeling water and cloth (Fig. 10). Previous applications of implicit hypertextures focused on blended procedural noise functions [20, 7]. Recently, implicits based predominately on generalized sinusoid product forms similar to that in Fig. 1 have been used within some modeling communities [11]. Arbitrary implicits are intriguing in their flexibility, and ray tracing promises the ability to dynamically render entire new classes of procedural geometries, independently from any polygonal geometry budget.

6 Conclusion

We have demonstrated a fast, robust and general algorithm for rendering implicits on the GPU. Performance was achieved by devising a stackless-recursion ray traversal algorithm; and a shader-language implementation of a generally correct reduced affine arithmetic, which improves performance for complex functions with high bound overestimation. We have shown the flexibility and potential of this approach for mathematical function visualization and rendering of procedural geometry.

Some drawbacks should be noted. While general, correct and fast, IA/AA methods still require copious computation compared to other approaches involving basis approximations, distance functions, or point sampling. A comprehensive comparison using 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, one would require beam tracing and likely supersampling.

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. Of more general importantance would be support on the application front-end for point, mesh or volume data, which could then be filtered and reconstructed by arbitrary implicits. This could be accomplished either by extending the rasterization system and restricting the application to ray casting; or by attempting a full ray-tracing system, with hierarchical acceleration structures, for the fragment shader. Though applied here to general implicits, inclusion methods could potentially be employed in rendering arbitrary parametric or free-form surfaces.

REFERENCES

[1] Jules Bloomenthal. An implicit surface polygonizer. pages 324–349, 1994.

[2] Ole Capriani, Lars Hvidegaard, Mikkel Mortensen, and Thomas Schneider. Robust and efficient ray intersection of implicit surfaces. *Reliable Computing*, 6:9–21, 2000.

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

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

[5] Rodrigo de Toledo, Bruno Levy, and Jean-Claude Paul. Iterative methods for visualization of implicit surfaces on gpu. In *ISVC, International Symposium on Visual Computing*, Lecture Notes in Computer Science, Lake Tahoe, Nevada/California, November 2007. SBC - Sociedade Brasileira de Computacao, Springer.

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

[7] Manuel N. Gamito and Steve C. Maddock. Ray casting implicit fractal surfaces with reduced affine arithmetic. *Vis. Comput.*, 23(3):155–165, 2007.

[8] Markus Hadwiger, Christian Sigg*, Henning Scharsach, Khatja Buhler, and Markus Gross*. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005.

[9] Pat Hanrahan. Ray tracing algebraic surfaces. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 83–90, New York, NY, USA, 1983. ACM Press.

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

[11] k3dsurf. The K3DSurf Project. http://k3dsurf.sourceforge.net/.

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

[13] Aaron Knoll, Younis Hijazi, Ingo Wald, Charles Hansen, and Hans Hagen. Interactive ray tracing of arbitrary implicits with simd interval arithmetic. In *Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing*, pages 11–18, 2007.

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

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

[16] Miriah D. Meyer, Pierre Georgel, and Ross T. Whitaker. 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)*, pages 124–133, Washington, DC, USA, 2005. IEEE Computer Society.

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

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

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

[20] K. Perlin and E. M. Hoffert. Hypertexture. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262, New York, NY, USA, 1989. ACM Press.

[21] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum*, 26(3), September 2007. (Proceedings of Eurographics), to appear.

[22] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 21(3):703–712, 2002.

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

[24] Szymon Rusinkiewicz and Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proc. of ACM SIGGRAPH*, pages 343–352, 2000.

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

[26] Daniel L. Toth. On ray tracing parametric surfaces. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 171–179, New York, NY, USA, 1985. ACM Press.

[27] J.J. van Wijk. Ray tracing objects defined by sweeping a sphere. *Computers & Graphics*, 9:283–290, 1985.

[28] Gokul Varadhan, Shankar Krishnan, Liangjun Zhang, and Dinesh Manocha. Reliable implicit surface polygonization using visibility mapping. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing*, pages 211–221, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[29] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 269–277, New York, NY, USA, 1994. ACM Press.

[30] Wolfram Research. Mathworld. http://mathworld.wolfram.com.

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