Ray Casting Implicit Procedural Noises with Reduced Affine Arithmetic

MANUEL N. GAMITO and STEVE C. MADDOCK
Department of Computer Science
The University of Sheffield

A method for ray casting implicit surfaces, defined with procedural noise models, is presented. The method is robust in that it is able to guarantee correct intersections at all image pixels and for all types of implicit surfaces. This robustness comes from the use of an affine arithmetic representation for the quantity that expresses the variation of the implicit function along a ray. Affine arithmetic provides a bounding interval estimate which is tighter than the interval estimates returned by conventional interval arithmetic. Our ray casting method is also efficient due to a proposed modification in the data structure used to hold affine arithmetic quantities. This modified data structure ultimately leads to a reduced affine arithmetic model. We show that such a reduced affine arithmetic model is able to retain all the tight estimation capabilities of standard affine arithmetic, in the context of ray casting implicit procedural noises, while being faster to compute and more efficient to store. We also show that, without this reduced model, affine arithmetic would not have any advantage over the more conventional interval arithmetic for ray casting the class of implicit procedural surfaces that we are interested in visualizing.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Computer Arithmetic*; G.1.5 [**Numerical Analysis**]: Roots of Nonlinear Equations—*Error analysis*; *Iterative methods*; G.4 [**Mathematical Software**]: Reliability and robustness; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Boundary representations*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Raytracing*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Affine arithmetic, implicit surfaces, procedural noise, ray casting

1. INTRODUCTION

Implicit surfaces play an important role in computer graphics. Surfaces exhibiting complex topologies, i.e. with many holes or disconnected pieces, can be easily modelled in implicit form. The same thing cannot be said for explicit surface representations, either with polygon meshes or bicubic patches. For this reason, implicit surface models have traditionally been used to represent complex objects like human characters [Wilhelms and Van Gelder 1997], fluids [Foster and Fedkiw 2001], amorphous substances [Desbrun and Gascuel 1995] or, perhaps more importantly, scientific and medical datasets [Whitaker 1998]. What is more, implicit surface representations handle animated surfaces, that undergo topological change through time, in a trivial way. Attempting to do the same with an explicit surface representation would require reparameterization of the surface every time a topological

Author's emails: M.Gamito@dcs.shef.ac.uk, S.Maddock@dcs.shef.ac.uk.
Author's address: Department of Computer Science, Regent Court 211 Portobello Street, Sheffield, S1 4DP, United Kingdom.

change occurred. It is, therefore, not surprising that a wealth of different rendering algorithms for implicit surfaces have been developed over the years.

Rendering algorithms for implicit surfaces can be broadly divided into two distinct categories:

—Meshing algorithms.
—Ray casting algorithms.

Meshing algorithms convert an implicit surface to a polygonal mesh format, which can be subsequently rendered in any way desired [Lorensen and Cline 1987; Bloomenthal 1988; Velho 1996]. The first meshing algorithm for computer graphics was proposed by Wright and Humbrecht [1979]. The advantage of a meshing algorithm is that an implicit surface is converted to a format which is ubiquitous in computer graphics: a mesh of triangles. Once the meshing step is performed, the implicit surface can be, for example, visualized in real time with modern GPU graphics boards. There are problems associated with mesh generation that make robust meshing algorithms for implicit surfaces difficult to implement. Firstly, the polygonal mesh should have the same topological structure of the implicit surface, with no pieces becoming connected after meshing or vice-versa [Stander and Hart 1997]. Secondly, the distance to the camera should be taken into account while generating the mesh. This insures the mesh will be optimal for a given viewing position, with no triangles that are either too large or too small. The mesh, however, will have to be regenerated if the camera changes position and popping effects might become visible during an animation.

Ray casting algorithms bypass mesh generation entirely and compute instead the projection of an implicit surface on the screen by casting rays from each pixel into three-dimensional space [Roth 1982; Hin et al. 1989]. The first application of a ray casting approach to implicit surfaces was presented by Blinn [1982]. The intersection point between a ray and the implicit surface is found, along with the normal at that point, and this is used to compute the luminous intensity at the pixel. In its simplest form, a ray casting algorithm marches along a ray, trying to find the first intersection with the surface. If the surface is very irregular, it may happen that some wrong intersections will be found or even that no intersections will be found for some rays that are actually intersecting the surface. This results in the familiar "surface acne" problem, which can potentially crop up in all rendering algorithms that rely on ray-surface intersection tests. Over the years, more robust ray casting algorithms have been developed to address this deficiency. Robust ray casting algorithms use concepts like Lifchitz bounds [Kalra and Barr 1989; Hart 1996] or interval arithmetic [Mitchell 1990] to obtain a confidence estimate that an intersection point will be found within some range of distances along the ray.

This work develops a ray casting algorithm for implicit surfaces generated from procedural noise functions. Our ultimate goal is to generate procedural planets, modelled as implicit surfaces. Ray casting was chosen over mesh generation because it is an elegant method that can handle easily the wide range of viewing distances that are required when visualizing a planet. Implicit surfaces based on procedural noise functions are typically very irregular and exhibit complex topologies. They require, therefore, an algorithm that is guaranteed to find all correct ray intersections. Failure to provide such a guarantee would amplify the surface

acne problem to a point that would make the final image nearly useless. Our algorithm evolves from the work of Mitchell [1990] where interval arithmetic was used to obtain estimates on the variation of the implicit surface's function along the ray. However, we replace interval arithmetic (IA) with affine arithmetic (AA) since the later is a more recent technology that is able to provide much tighter estimates for the aforementioned variation [Comba and Stolfi 1993].

Ray casting with affine arithmetic has already been presented in the literature [Junior et al. 1999]. When comparing AA against IA, Junior et al. [1999] reported mixed results for implicit surfaces generated from several textbook mathematical functions that do not find much application in computer graphics. Our work focuses, instead, on implicit surfaces generated from Perlin-like procedural noises whose usefulness to computer graphics is without question [Ebert et al. 2003]. We have found that a direct implementation of AA, as proposed by Junior et al. [1999], was less efficient than the IA implementation of Mitchell [1990] for ray casting implicit procedural noises. This motivated our work in obtaining a reduced representation for AA, which is as accurate as the original one, while being more efficient. It is this reduced AA representation for ray casting implicit surfaces that is described in the remainder of this paper.

In Section 2, a more precise formulation of the ray casting problem is presented. The approach to solving this problem with interval arithmetic is also given. Section 3 gives a brief discussion of affine arithmetic models in general. Affine arithmetic is then applied to the ray casting problem in Section 4, after our reduced AA representation has been derived. Section 5 shows results and presents a comparison between reduced AA, standard AA, and IA for ray casting surfaces defined from procedural noise functions. Section 6 presents the conclusions that we have reached with this work.

2. RAY CASTING IMPLICIT SURFACES

Consider a function $f(\mathbf{x})$ from $\mathbb{R}^3$ to $\mathbb{R}$, which is continuous everywhere. This function, when applied to any point in three-dimensional space, returns a scalar value. The exact interpretation to give to this scalar is irrelevant. What is important to us here is the sign of $f(\mathbf{x})$. We say that, if $f(\mathbf{x}) > 0$, we are on the outside of some volume. Conversely, when $f(\mathbf{x}) < 0$, we are on the inside. The points where $f(\mathbf{x}) = 0$ are then on the interface between the inside and the outside of the volume[1]. Formally, the set S of all such points defines a surface (more precisely, a two-dimensional manifold) in $\mathbb{R}^3$ given by:

$$S = \left\{ \mathbf{x} \in \mathbb{R}^3 \,:\, f(\mathbf{x}) = 0 \right\}. \tag{1}$$

Because we can only know if a point $\mathbf{x}$ belongs to the surface after $f(\mathbf{x})$ has been evaluated, we say that S is an *implicit surface*. By changing the function $f(\mathbf{x})$, we change the shape of the surface. For example, if we want to obtain a spherical surface of unit radius, centered at the origin, we can use $f(\mathbf{x}) = \|\mathbf{x}\|^2 - 1$.

[1]The choice of positive $f(\mathbf{x})$ for the outside and negative $f(\mathbf{x})$ for the inside is entirely arbitrary. The opposite would have been equally valid.

Enter ray casting. We seek to obtain a visualization of S as seen through a virtual camera placed somewhere in $\mathbb{R}^3$. To do so, we shoot a ray from every pixel in the camera's screen space into three-dimensional space and try to determine if and where this ray will intersect with S. A ray $\mathbf{r}(t)$ will be represented in explicit form as a semi-infinite line, parameterized by the distance t:

$$\mathbf{r}(t) = \mathbf{o} + t\,\mathbf{d}, \quad \text{with } t \geq 0. \tag{2}$$

In (2), the position vector $\mathbf{o}$ is the starting point of the ray and is also the location of the camera in $\mathbb{R}^3$. Vector $\mathbf{d}$ is the ray direction. It is usual, though not necessary, for $\mathbf{d}$ to be normalized ($\|\mathbf{d}\| = 1$). If $\mathbf{d}$ is indeed normalized then t gives the correct euclidean distance for any point along the ray. To find if the ray intersects the surface, plug (2) into (1) and try to find a solution for t:

$$f(\mathbf{o} + t\,\mathbf{d}) = 0. \tag{3}$$

If we define the auxiliary function $g : \mathbb{R} \mapsto \mathbb{R}$, such that $g(t) = f(\mathbf{o} + t\,\mathbf{d})$, it becomes clear that we are dealing with a non-linear root-finding problem in one dimension:

$$g(t) = 0. \tag{4}$$

Only real roots of (4) have meaning in the context of ray casting. If there is no such root, the ray does not intersect the surface. If there are multiple real roots of (4), the ray intersects the surface multiple times along its way. In this later case, we are interested in the solution with the smallest root t, since this will correspond to the intersection point that is visible from the camera. Some roots can also have a multiplicity of two, for rays that are only tangent to the surface at some point. Such roots can be ignored if we arbitrate that a ray tangent to the surface does not create an intersection point.

Traditional numerical methods for root finding are not of much help because, if (4) has multiple roots, these methods provide very limited control (if any) over which root will be returned [Corliss 1977]. The root or roots must first be bracketed inside some suitable interval $[t_a, t_b]$ before root finding takes place. With traditional methods, the only way one can be certain that the smallest root will be returned is to make sure that $[t_a, t_b]$ contains this and only this root. Such root bracketing is very difficult to achieve in a general way for any given $g(t)$ function.

Enter interval arithmetic. An IA representation $\hat{t} = [t_a, t_b]$ expresses our limited knowledge about the value of the quantity t. All we are able to say with this representation is that $t_a \leq t \leq t_b$. It is possible to derive an algebra for IA, similar to the algebra of real numbers, by replacing all real arithmetic operations with equivalent operations working on interval quantities [Moore 1966]. For example, if $\hat{u} = [u_a, u_b]$ and $\hat{v} = [v_a, v_b]$ are two intervals, their sum $\hat{w} = \hat{u} + \hat{v}$ is given by $\hat{w} = [u_a + v_a, u_b + v_b]$. What is more, the transcendental functions in $\mathbb{R}$ can also be generalized to receive intervals as arguments and return intervals. It is perfectly reasonable, therefore, to write something like $g(\hat{t}\,) = \sin \hat{t}$.

The one problem with IA is its over-conservativeness. The radius $|\hat{t}\,| = (t_b - t_a)/2$ of an interval $\hat{t} = [t_a, t_b]$ can be used as a measure of this conservativeness in IA

computations. Consider the subtraction $\hat{t} - \hat{t}$ of an IA quantity $\hat{t}$ with itself. Naturally, the result should be zero. Strict application of the IA rules for subtraction lead, however, to the result $[t_a - t_b, t_b - t_a]$. The correct result of zero is indeed contained in the interval but its radius $|\hat{t} - \hat{t}| = t_b - t_a$ is much larger than it needs to be. This is because IA was not able to recognize that the two operands to the subtraction are correlated (in fact, they are the same in this example) and had to apply conservative rules to ensure the resulting interval would enclose every possible answer.

Returning to the ray casting problem expressed by (4), it is now possible to obtain an interval estimate $g(\hat{t}\,)$ for the variation of the implicit function when the distance t along the ray ranges over some interval $\hat{t} = [t_a, t_b]$. The IA quantity $g(\hat{t}\,)$ is always an estimate, to some degree, of the true variation along t. Interval arithmetic guarantees that an estimate $g(\hat{t}\,)$ will contain the set $\bar{g}(\hat{t}\,) = \{g(t) : t \in \hat{t}\}$ of all values returned by $g(t)$ for the range $[t_a, t_b]$. Trying to know how tightly the estimate $g(\hat{t}\,)$ encloses $\bar{g}(\hat{t}\,)$ takes us back to the over-conservativeness problem of IA. What always happens is that, as the number of IA operations required to compute $g(\hat{t}\,)$ increases, this later estimate becomes progressively conservative, to the point of ceasing to give any more useful information about $\bar{g}(\hat{t}\,)$. For a sufficiently complex implicit surface we will have, in the limit, $|g(\hat{t}\,)| \to +\infty$ and $g(\hat{t}\,)$ will converge to the whole real line.

How can we know if a root of (4) is contained in some interval $\hat{t} = [t_a, t_b]$? We evaluate $g(\hat{t}\,)$ with IA and check to see if this estimate contains the value zero. Keeping in mind the conservative properties of interval arithmetic, two outcomes can result from this test:

The estimate $g(\hat{t}\,)$ contains zero. The fact that $0 \in g(\hat{t}\,)$ does not necessarily imply that $0 \in \bar{g}(\hat{t}\,)$. Nothing can be concluded in this case. The best thing to do is to split the interval $\hat{t}$ into two smaller intervals and try again.

The estimate $g(\hat{t}\,)$ does not contain zero. Since $g(\hat{t}\,) \supseteq \bar{g}(\hat{t}\,)$, it is true that $0 \notin g(\hat{t}\,) \Rightarrow 0 \notin \bar{g}(\hat{t}\,)$. The interval $\hat{t}$ can be discarded because no root of (4) can possibly exist inside it.

A simple recursive subdivision procedure can be implemented, using the rationale just explained. It starts with an initial range of distances $[t_a, t_b]$, known to bound all possible roots for ray intersection. This initial range is obtained by intersecting the ray with a bounding object that is guaranteed to enclose the implicit surface. A stack data structure is used to store intervals waiting to be tested for the existence of roots. The procedure, which will be called `RayIntersect`, is listed in Figure 1.

The recursion stops either when a small enough interval bounding a root has been found or when the stack becomes empty. The later scenario occurs in situations where a ray does not intersect the surface. The order with which the two subintervals are pushed onto the stack, in the case where both may contain a root, is not arbitrary. By pushing $\hat{t}_r$ first and then $\hat{t}_l$ the nearest intersection is guaranteed to be found.

If properly implemented, a ray casting algorithm with interval arithmetic is already robust in the sense that all correct ray intersections with the implicit surface will be found. The problem with such an algorithm, however, is that it is not efficient. A large number of iterations is required before the position of the first root

```
push initial [t_a, t_b] onto stack;
while stack not empty
  pop t̂ = [t_a, t_b] from the stack;
  if |t̂| < ϵ
    return t_a;
  let t_i = (t_a + t_b)/2;
  let t̂_l = [t_a, t_i] and t̂_r = [t_i, t_b];
  if g(t̂_r) ∋ 0
    push t̂_r onto stack;
  if g(t̂_l) ∋ 0
    push t̂_l onto stack;
```

Fig. 1. The `RayIntersect` algorithm.

along each ray can be tracked down with sufficient accuracy. This is ultimately due to the aforementioned over-conservativeness problem of IA.

3. AFFINE ARITHMETIC

To overcome the deficiencies of interval arithmetic, Comba and Stolfi [1993] proposed a different representation for quantities involving uncertainty, which they called *affine arithmetic*. The representation of some quantity with affine arithmetic (AA) tries to model the uncertainties about that quantity so that it is always bounded inside a known interval. The advantage over the simpler interval arithmetic framework is that AA tries to keep correlations between quantities, calculated along some arbitrarily long chain of computations. AA keeps correlations between similar quantities through the use of *error symbols*. A quantity $\hat{t}$ in AA is represented as a central value t_0 plus a sequence of error symbols e_i, each with its associated error coefficient:

$$\hat{t} = t_0 + t_1 e_1 + t_2 e_2 + \cdots + t_n e_n. \tag{5}$$

The error symbols lie in the interval $[-1, +1]$ but are otherwise unknown and the coefficients t_i express the contribution of each symbol to the AA quantity. Error symbols can be shared among several AA quantities and that is how correlation information can be kept among related quantities. The sequence of error symbols in (5) is typically sparse, with many of the coefficients t_i being zero. This is because, if the uncertainty associated with some error symbol e_i of an AA quantity is not shared with any other AA quantities, the later must all have their t_i coefficients null. When implementing AA, expensive book-keeping routines are often required to manage the large but sparse sequences of error coefficients.

The computation of affine operations on AA quantities does not result in the creation of any new error symbols. For two AA quantities $\hat{u}$, $\hat{v}$ and a scalar α, the affine operations are:

$$\begin{aligned} \alpha\hat{u} &= (\alpha u_0) + (\alpha u_1)e_1 + \cdots + (\alpha u_n)e_n, \\ \hat{u} \pm \alpha &= (u_0 \pm \alpha) + u_1 e_1 + \cdots + u_n e_n, \\ \hat{u} \pm \hat{v} &= (u_0 \pm v_0) + (u_1 \pm v_1)e_1 + \cdots + (u_n \pm v_n)e_n. \end{aligned} \tag{6}$$

Application of the above rules to an operation like $\hat{t} - \hat{t}$ produces a result of exactly zero, something that, as we have seen, IA could not do.

For non-affine operations, like multiplication or square root, things are a bit more complex. Because these operations cannot be expressed in affine form, a new error symbol must be introduced to express the non-linearity of the operator. For example, if $\hat{w} = \hat{u} \cdot \hat{v}$, the result $\hat{w} = w_0 + w_1 e_1 + \cdots + w_n e_n + w_k e_k$ has the new error symbol e_k appended to it. The coefficients for the result are:

$$\begin{aligned} w_0 &= u_0 v_0, \\ w_i &= u_0 v_i + v_0 u_i, \quad \text{for } i = 1, \ldots, n, \\ w_k &= \sum_{i=1}^{n} |u_i| \cdot \sum_{i=1}^{n} |v_i|. \end{aligned} \tag{7}$$

As a sequence of AA operations progresses, quantities have an increasingly larger number of error symbols, slowing down subsequent AA computations and increasing the memory requirements. Still, AA overcomes the over-conservativeness problem of IA and is able to produce much tighter bounds after long chains of operations. This is quite useful for ray casting where the number of operations required to compute $g(\hat{t})$ is usually large. Because of its ability to maintain correlations among uncertain quantities, AA has been shown to be potentially useful in several computer graphics applications like the discretization of parametric curves [de Figueiredo et al. 2003] or the ray tracing of displacement maps [Heidrich and Seidel 1998].

4. RAY CASTING IMPLICIT PROCEDURAL NOISES

We propose to model implicit surfaces based on an initial spherical surface, which is subsequently modulated by one or more procedural noises. The general form for our implicit surface generation function is (recall (1)):

$$f(\mathbf{x}) = \|\mathbf{x}\|^2 - 1 + \sum_{i=1}^{N} a_i n(b_i \mathbf{x}). \tag{8}$$

We could equally well have used a geometric shape other than a sphere as the seed to our implicit surface. However, we intend to model whole planets in a procedural way in the future and so the choice of a sphere is appropriate. In the above equation, N layers of a procedural noise $n(\mathbf{x})$ are added to the implicit function, making the surface of the sphere increasingly more corrugated as N increases. The parameters a_i and b_i are used to scale the contribution of each layer of noise to the final surface.

There are several forms of procedural noises that have been developed in the literature for computer graphics. In the most general way, a procedural noise function can be written as:

$$n(\mathbf{x}) = \sum_{i \in \mathcal{N}(\mathbf{x})} h_i(\mathbf{x} - \mathbf{x}_i), \tag{9}$$

where the $h_i(\mathbf{x})$ are kernel functions with a finite support in $\mathbb{R}^3$. Each kernel function takes the influence of some node $\mathbf{x}_i$ and spreads it throughout its support. The influence that each node $\mathbf{x}_i$ has on the final noise is expressed either as a random value attributed to it, a random gradient vector, or both. Because the support of the kernel functions is finite, only a small set $\mathcal{N}(\mathbf{x})$ of them will contribute to any point $\mathbf{x}$ in three-dimensional space. If this was not so the evaluation of a procedural noise would be computationally intractable.

Perlin procedural noises use nodes located at the vertices of an integer sized lattice. The kernels are made of separable Hermite interpolation polynomials with support on the interval $[-1, +1]$ along each coordinate axis. This means that only the eight nodes on the vertices of the lattice cell where the point $\mathbf{x}$ is located need to be taken into account when computing (9). The kernels can either be cubic or quintic interpolation polynomials [Perlin 1985; 2002]. Sparse convolution noises use K nodes semi-randomly placed inside each lattice cell. For each point $\mathbf{x}$ in space, $27K$ kernels need to be evaluated: K for the cell where $\mathbf{x}$ is contained plus $26 \times K$ for the twenty six surrounding cells. Sparse convolution noises are, therefore, more expensive to compute than Perlin noises but they have much better spectral characteristics [Lewis 1989]. Each kernel in a sparse convolution noise is a function of the distance $\|\mathbf{x} - \mathbf{x}_i\|$ to its node and can be arbitrarily specified by the user, subject only to the constraint of having support on the range of distances given by the interval $[0, 1]$. Voronoi procedural noises are similar to sparse convolution noises but use instead linear combinations of minimum distance functions as their kernels [Worley 1996].

We use two bounding spheres, which are guaranteed to completely surround the implicit surface given in (8), both from the inside and the outside. These spheres are used to obtain the initial interval estimate $\hat{t} = [t_a, t_b]$ for the distance along a ray, the purpose of which has been explained in Section 2. Knowing that all procedural noises return values in the range $[-1, +1]$, the radii for the two spheres are given by[2]:

$$\begin{aligned} r_{\max} &= 1 + \sum_{i=1}^{N} |a_i|, \\ r_{\min} &= 1 - \sum_{i=1}^{N} |a_i|. \end{aligned} \tag{10}$$

One of the following three situations will arise while performing the initial intersection tests between a ray and the two spheres:

— The ray does not intersect the outer sphere. In this case, the ray is simply discarded and its originating pixel receives no light contribution.

[2]If the a_i are large enough, the inner radius $r_{\min}$ may become negative. In this case we simply ignore the inner bounding sphere.

— The ray intersects the outer sphere going in and then again going out. This will happen for grazing rays. Grazing rays are more expensive to compute than other rays because they travel a greater distance inside the volume bounded by the spheres.

— The ray intersects the outer sphere and then the inner sphere. This is the most typical situation. There is another pair of intersections on the opposite side of both spheres but this is never considered since an intersection will always be found inside the first pair.

Once the initial interval estimate for intersection is known, the `RayIntersect` algorithm can start either using IA or AA operations.

4.1 Reduced Affine Arithmetic

The arguments presented in Section 3 have motivated our development of a reduced form of affine arithmetic for ray casting implicit surfaces based on procedural noises. In a procedural noise of the form expressed by equation (9), each node $\mathbf{x}_i$ has a contribution to $n(\mathbf{x})$ that is statistically independent from the contributions of all other nodes. Correlations between the different i terms in the summation of (9) are not expected to exist. The only correlation that will be maintained across all AA quantities is related to the uncertainty along the length of the ray.

A reduced AA representation $\hat{t}$ maintains only three parameters from the original AA representation: the central value t_0, the coefficient t_1, expressing uncertainty along the ray and a final coefficient t_2, expressing uncertainties involved in the computation of $\hat{t}$ alone. The coefficient t_1 is the only correlation that exists between $\hat{t}$ and other AA quantities. The expression for $\hat{t}$ is:

$$\hat{t} = t_0 + t_1 e_1 + t_2 e_2. \tag{11}$$

The intervals $\hat{t} = [t_a, t_b]$ bounding the position of roots along a ray and used in the `RayIntersect` algorithm are written with:

$$\hat{t} = \frac{t_b + t_a}{2} + \frac{t_b - t_a}{2} e_1. \tag{12}$$

The error symbol e_2 is not yet necessary for these intervals and, therefore, $t_2 = 0$. All subsequent quantities, computed during the evaluation of $g(\hat{t}\,)$, will have a correlation with $\hat{t}$ expressed through the coefficients of their e_1 error symbols.

When performing AA operations, the contribution from any new error symbol is accumulated, in modulus form, into the coefficient for e_2 instead of being appended to the representation. Consider again the example of multiplication between AA quantities. If $\hat{w} = \hat{u} \cdot \hat{v}$, the three parameters of $\hat{w}$ will be:

$$\begin{aligned} w_0 &= u_0 v_0, \\ w_1 &= u_0 v_1 + v_0 u_1, \\ w_2 &= |u_0 v_2 + v_0 u_2| + (|u_1| + |u_2|) \cdot (|v_1| + |v_2|)\,. \end{aligned} \tag{13}$$

This is a reduction of the general rule for AA multiplication, as given by (7), hence the name of reduced affine arithmetic. In (13), the terms w_0 and w_1 are

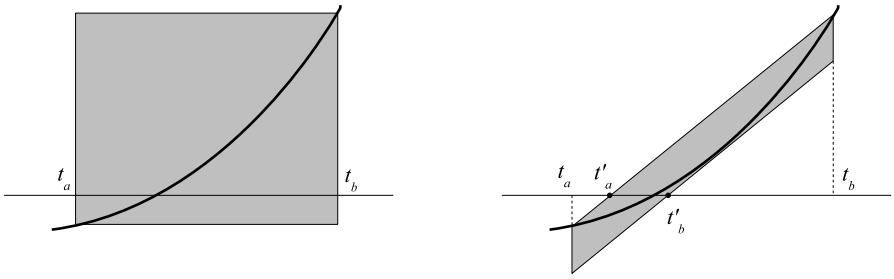

Fig. 2. Comparison between the information conveyed by IA (left) and reduced AA (right) for the behaviour of the implicit function $g(\hat{t})$ inside an interval $\hat{t} = [t_a, t_b]$.

the same as w_0 and w_1 from (7). The term w_2 sums the modulii of the terms w_2 and w_k from (7) when $n = 2$. The rules for all other AA operations in reduced form can be obtained in a similar way. With the representation (11), the time required to compute an AA operation remains constant throughout the sequence of computations. The storage size of a reduced AA quantity also remains constant. The costly book-keeping procedures, necessary to handle the sparse sequences of error symbols in standard AA, are no longer required for reduced AA.

4.2 Interval Optimization with Affine Arithmetic

The idea of optimizing the size of the interval bounding the first root of (4) was first presented in Junior et al. [1999]. We present it here again in the framework of reduced affine arithmetic.

When using IA to bracket an intersection point along the ray, nothing is known about the behaviour of the implicit surface inside the interval where the point lies. The only possible strategy, while trying to converge to the intersection point, is to split the bounding interval in half and test each of the halves in turn. With AA, however, it is possible to reduce the size of the interval before performing a split by taking advantage of the extra information provided by the AA representation. Figure 2 compares the type of information conveyed by an IA representation of the implicit function $g(\hat{t}\,)$ inside some interval $\hat{t} = [t_a, t_b]$ along the ray and the information conveyed by a reduced AA representation over the same interval.

We seek the point where the graph of the function crosses the horizontal axis. The IA representation is graphically equivalent to a bounding box that spans the minimum and maximum values of the function inside $\hat{t}$. The reduced AA representation is equivalent to a parallelipiped, which bounds the function much more tightly than the IA box. The bounding interval can be reduced to $\hat{t}' = [t'_a, t'_b]$ before performing any splitting. It is clear from the drawing that significant convergence towards the root is achieved with just a single evaluation of the function in AA form and before any further subdivision takes place.

The optimized interval $\hat{t}'$ is obtained from the reduced AA representations for the interval $\hat{t} = t_0 + t_1 e_1$ and the implicit function $g(\hat{t}\,) = g_0 + g_1 e_1 + g_2 e_2$ in the following way:

$$
\begin{aligned}
t'_a &= \max\left(t_0 - \frac{g_0}{g_1}t_1 - \left|\frac{g_2}{g_1}\right| t_1, t_a\right), \\
t'_b &= \min\left(t_0 - \frac{g_0}{g_1}t_1 + \left|\frac{g_2}{g_1}\right| t_1, t_b\right).
\end{aligned}
\tag{14}
$$

The `RayIntersect` algorithm is modified to push the optimized intervals $\hat{t}'_l$ and $\hat{t}'_r$ onto the stack when $\hat{t}_l$ or $\hat{t}_r$ contain the value zero[3]. This interval optimization strategy reduces significantly the number of iterations necessary to find the first root of (4) by providing quadratic convergence instead of the linear convergence of a strict recursive subdivision method.

5. RESULTS

Figure 3 shows two images of an implicit sphere, the surface of which has been modified by the addition of a Perlin procedural gradient noise with quintic interpolants [Perlin 2002]. The image on the top of Figure 3 has only one layer of noise, while the image on the bottom has three.

Correct implementation of both interval and affine arithmetic requires control over the rounding direction of the processor to ensure that computed estimates are always conservative. Changing the rounding direction on the fly is an expensive operation for most modern processors. We have found, however, that using $\epsilon = 10^{-8}$ in `RayIntersect` did not require any rounding control because, for this epsilon, the required accuracy is still much lower than the maximum accuracy of double precision floating point numbers.

Tables I and II show some statistics that enable a comparison between all the interval estimation techniques that have been presented. The rendering time was obtained for a 800×600 resolution image. The average number of function evaluations per ray gives an indication of how often the algorithms needed to compute bounds for the implicit function. The theoretical optimum value for this statistic would be 1.0, for a perfect algorithm that could be capable of finding the intersection point by evaluating the implicit function only once. It turns out, surprisingly, that, if the implicit function is polynomial, there does exist a bounding algorithm capable of attaining the optimum value; more on this in the next section. When computing the average number of function evaluations per ray, only rays that intersected the outer bounding sphere and needed further testing were accounted for. The rays that miss the bounding sphere entirely are trivially discarded.

As expected, the standard IA surface intersection algorithm needs a large number of function evaluations due to the over-conservativeness of the IA representation. Standard IA, however, compensates for this lack of accuracy by being quite fast, which makes it competitive with some of the more advanced algorithms. Straightforward replacement of the IA operations with AA equivalents leads to a more inefficient algorithm, due to the need to compute sequences of error symbol coeffi-

[3]When $g_1 \to 0$ the parallelogram on the right of Figure 2 tends toward the rectangle on the left. In the limit, no optimization is possible and the original intervals t_l or t_r must be pushed onto the stack.

Table I. Statistics for an implicit sphere with one layer of noise.

	Avg. Evals. p/ray	*Time*
Standard IA	57.18	3m27.8s
Standard AA	29.21	15m29.4s
Reduced AA	28.86	2m39.8s
Reduced AA + Int. Opt.	11.92	1m06.4s

Table II. Statistics for an implicit sphere with three layers of noise.

	Avg. Evals. p/ray	*Time*
Standard IA	82.04	20m20.3s
Standard AA	40.29	1h29m49.8s
Reduced AA	38.24	14m25.6s
Reduced AA + Int. Opt.	22.25	8m31.8s

cients that grow progressively larger. Nevertheless, standard AA is able to reduce the average number of function evaluations, which shows that AA does have the potential to optimize ray-surface intersection algorithms, if only it can be implemented in a more efficient manner.

The better performance of IA over standard AA for procedural noise models has already been implicitly acknowledged in Heidrich et al. [1998]. In that work, IA was used for computing the interval estimates of a Perlin noise. These interval estimates were then converted into AA form for use in the rest of the application. The authors do not state a reason for preferring IA over AA when computing a Perlin noise but it is symptomatic that such a decision was taken in a paper whose purpose was to propose AA as a better alternative to IA.

Efficiency with AA is obtained in the reduced AA representation, where a maximum of two error symbols per quantity are used. It is elucidating to see that the number of function evaluations with reduced AA is essentially the same as with standard AA, meaning that no accuracy was lost by constraining AA quantities to two error symbols[4]. Reduced AA proves to be a more efficient representation than standard AA by allowing significantly shorter rendering times and making it slightly better than IA. The final optimization comes from reducing the size of the intervals, as explained in Section 4.2. Now, both rendering statistics are much lower than with any of the other bounding techniques.

6. CONCLUSIONS AND FUTURE WORK

Ray casting implicit surfaces with affine arithmetic becomes efficient only after a reduced representation for uncertain quantities has been introduced. A reduced affine arithmetic quantity uses a maximum of two error symbols. It has been shown that without this reduced representation affine arithmetic would not be able to compete against a simpler interval arithmetic representation. These results were obtained while ray casting implicit surfaces generated from procedural noise functions that are widely used in computer graphics. Procedural noise models are based on the summation of several statistically independent terms. Only the correlation related to the uncertainty in the position of the root along the ray needs

[4]In fact, Tables I and II show a little gain in accuracy with reduced AA. This comes from working with the assumption that uncertainty about the position along the ray is always stored in the t_1 parameter of the reduced AA representation.

to be kept. It is for this reason that reduced affine arithmetic can achieve the same results as standard affine arithmetic while being more efficient.

An interval bounding technique that could potentially be more efficient than both interval and affine arithmetic represents uncertain quantities with truncated Taylor polynomials followed by an interval that expresses the uncertainty in the representation [Berz and Hoffstätter 1998]. An attractive feature of this technique is the inherent scalability that comes from choosing the maximum order of the Taylor polynomial. With an order zero polynomial, only the uncertainty interval remains which is equivalent to an IA algorithm. An order one Taylor polynomial gives a constant plus an interval, which is somewhat similar to AA with only one error symbol. Still higher Taylor polynomials give progressively better estimates, at the cost of having to compute increasingly higher derivatives of the implicit function. If the implicit function is known to be a polynomial of order n then a Taylor polynomial, also of order n, will provide the solution with a single evaluation of the function and with no need for the additional uncertainty interval (except where the interval might be necessary to account for the small precision errors of the FPU hardware). One then only needs to compute the smallest root of an order n polynomial, for which several stable numerical methods exist, to get the distance along the ray to the intersection point.

ACKNOWLEDGMENT

The authors would like to express their gratitude to Luiz Henrique de Figueiredo for graciously making available the source code used in Junior et al. [1999].

REFERENCES

Berz, M. and Hoffstätter, G. 1998. Computation and application of Taylor polynomials with interval remainder bounds. *Reliable Computing 4,* 1, 83–97.

Blinn, J. F. 1982. A generalization of algebraic surface drawing. *ACM Transactions on Graphics 1,* 3 (July), 235–256.

Bloomenthal, J. 1988. Polygonisation of implicit surfaces. *Computer Aided Geometric Design 5*, 341–355.

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

Corliss, G. F. 1977. Which root does the bisection algorithm find? *SIAM Reviews 19,* 2, 325–327.

de Figueiredo, L. H., Stolfi, J., and Velho, L. 2003. Approximating parametric curves with strip trees using affine arithmetic. *Computer Graphics Forum 22,* 2, 171–171.

Desbrun, M. and Gascuel, M. 1995. Animating soft substances with implicit surfaces. In *SIGGRAPH 95 Conference Proceedings*, R. Cook, Ed. Annual Conference Series. ACM SIGGRAPH, Addison Wesley, 287–290. held in Los Angeles, California, 06-11 August 1995.

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. 2003. *Texturing & Modeling: A Procedural Approach*, Third ed. Morgan Kaufmann Publishers Inc.

Foster, N. and Fedkiw, R. 2001. Practical animation of liquids. In *SIGGRAPH 2001 Conference Proceedings, August 12–17, 2001, Los Angeles, CA*, ACM, Ed. ACM Press, New York, NY 10036, USA, 23–30.

Hart, J. C. 1996. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12,* 9, 527–545. ISSN 0178-2789.

Heidrich, W. and Seidel, H.-P. 1998. Ray-tracing procedural displacement shaders. In *Proceedings of the 24th Conference on Graphics Interface (GI-98)*, W. Davis, K. Booth, and A. Fournier, Eds. Morgan Kaufmann Publishers, San Francisco, 8–16.

Heidrich, W., Slusallik, P., and Seidel, H. 1998. Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics 17,* 3 (July), 158–176.

Hin, A. J. S., Boender, E., and Post, F. H. 1989. Visualization of 3D scalar fields using ray casting. In *Eurographics Workshop on Visualization in Scientific Computing*.

Junior, A., de Figueiredo, L., and Gattas, M. 1999. Interval methods for raycasting implicit surfaces with affine arithmetic. In *Proc. XII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI '99)*. 1–7.

Kalra, D. and Barr, A. H. 1989. Guaranteed ray intersections with implicit surfaces. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, J. Lane, Ed. Vol. 23. 297–306.

Lewis, J.-P. 1989. Algorithms for solid noise synthesis. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, J. Lane, Ed. Vol. 23. 263–270.

Lorensen, W. E. and Cline, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, M. C. Stone, Ed. Vol. 21. 163–169.

Mitchell, D. P. 1990. Robust ray intersection with interval arithmetic. In *Proceedings of Graphics Interface '90*. 68–74.

Moore, R. 1966. *Interval Arithmetic*. Prentice-Hall, Englewood Cliffs (NJ), USA.

Perlin, K. 1985. An image synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, B. A. Barsky, Ed. Vol. 19. 287–296.

Perlin, K. 2002. Improving noise. *ACM Transactions on Graphics 21,* 3 (July), 681–682.

Roth, S. D. 1982. Ray casting for modeling solids. *Computer Graphics and Image Processing 18,* 2 (Feb.), 109–144.

Stander, B. T. and Hart, J. C. 1997. Guaranteeing the topology of an implicit surface polygonization for interactive modeling. In *SIGGRAPH 97 Conference Proceedings*, T. Whitted, Ed. Annual Conference Series. ACM SIGGRAPH, Addison Wesley, 279–286.

Velho, L. 1996. Simple and efficient polygonization of implicit surfaces. *Journal of Graphics Tools 1,* 2. ISSN 1086-7651.

Whitaker, R. T. 1998. A level-set approach to 3D reconstruction from range data. *Int. J. Computer Vision 29*, 203–231.

Wilhelms, J. and Van Gelder, A. 1997. Anatomically based modeling. In *SIGGRAPH 97 Conference Proceedings*, T. Whitted, Ed. Annual Conference Series. ACM SIGGRAPH, Addison Wesley, 173–180.

Worley, S. P. 1996. A cellular texture basis function. In *SIGGRAPH 96 Conference Proceedings*, H. Rushmeier, Ed. Annual Conference Series. ACM SIGGRAPH, Addison Wesley, 291–294. held in New Orleans, Louisiana, 04-09 August 1996.

Wright, T. and Humbrecht, J. 1979. ISOSRF — an algorithm for plotting iso-valued surfaces of a function of three variables. *Computer Graphics (SIGGRAPH '79 Proceedings) 13,* 3 (Aug.), 182–189.

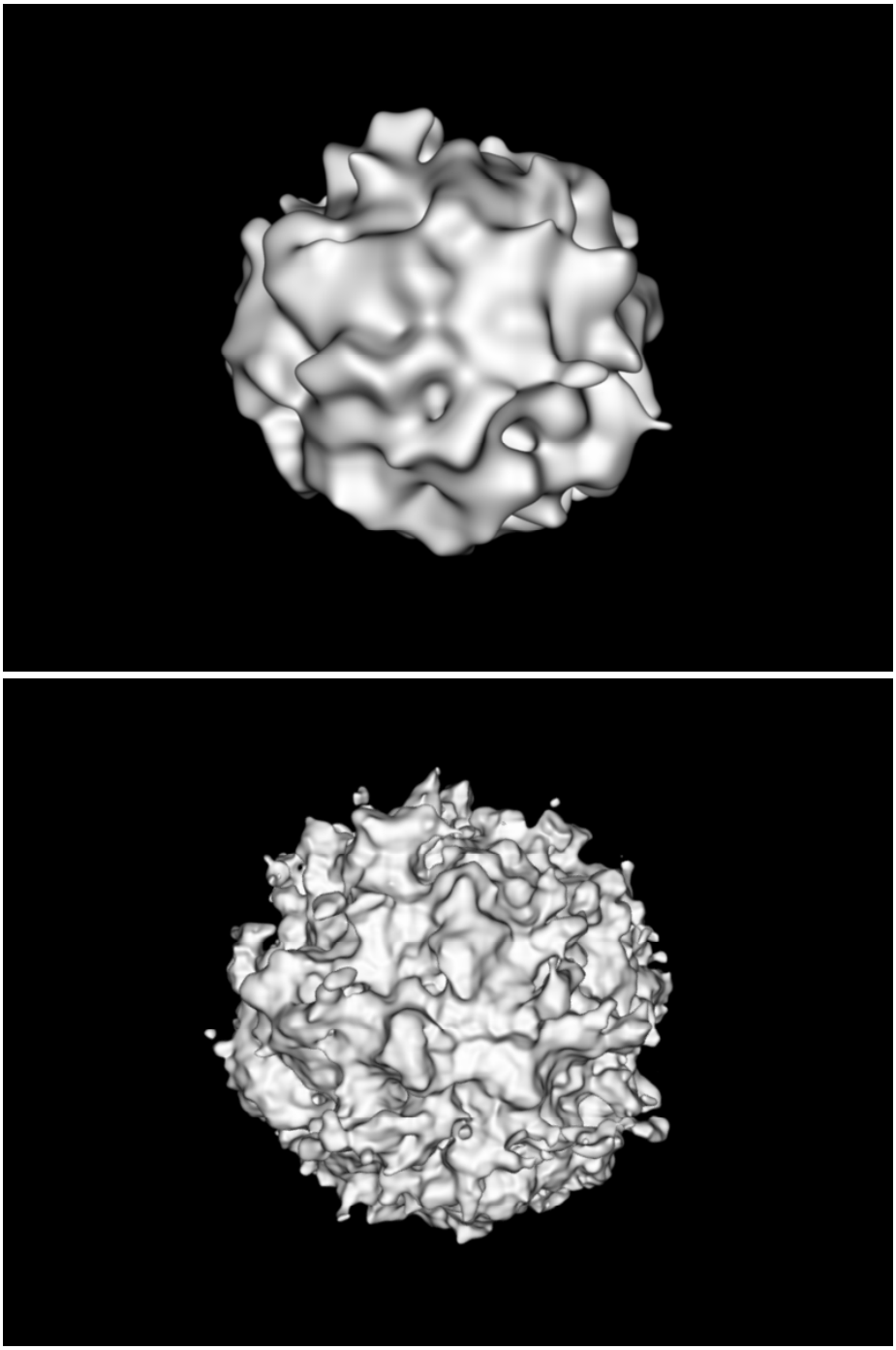

Fig. 3. An implicit sphere modulated with one layer (above) and three layers (below) of a Perlin procedural gradient noise.