This is a repository copy of *Ray casting implicit fractal surfaces with reduced affine arithmetic* .

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/3762/

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Manuel N. Gamito · Steve C. Maddock

Ray Casting Implicit Fractal Surfaces with Reduced Affine Arithmetic

Abstract A method is presented for ray casting implicit surfaces defined by fractal combinations of procedural noise functions. The method is robust and uses affine arithmetic to bound the variation of the implicit function along a ray. The method is also efficient due to a modification in the affine arithmetic representation that introduces a condensation step at the end of every affine arithmetic operation. We show that our method is able to retain the tight estimation capabilities of affine arithmetic for ray casting implicit surfaces made from procedural noise functions while being faster to compute and more efficient to store.

Keywords Affine arithmetic · Implicit Surfaces · Procedural Noise Functions · Ray Casting

1 Introduction

This work develops an algorithm for ray casting implicit fractal surfaces generated from procedural noise functions. An implicit surface is defined as the set of all points for which the evaluation of some continuous function $f : \mathbb{R}^3 \to \mathbb{R}$ gives zero. If the function f is a fractal with dimension D_f, the implicit surface, being a zeroset of this function, is also fractal with dimension $D_f - 1$ [29]. A common procedural technique to obtain functions that are fractal over a finite range of scales is to accumulate several layers of noise. Each

Manuel N. Gamito
Department of Computer Science
The University of Sheffield
211 Portobello Street
Sheffield S1 4DP
E-mail: M.Gamito@dcs.shef.ac.uk

Steve C. Maddock
Department of Computer Science
The University of Sheffield
211 Portobello Street
Sheffield S1 4DP
E-mail: S.Maddock@dcs.shef.ac.uk

layer consists of a scaled and frequency shifted copy of some original band-limited procedural noise function n [25].

Implicit fractal surfaces are one example of *hypertextures* [24]. Hypertextures use functions to add volumetric detail to the surface of objects, thereby increasing their visual complexity. Hypertextured objects can either be visualised with a volume rendering approach or converted to an implicit surface representation [10].

Another application for implicit fractal surfaces is in *procedural planet modelling* [19]. One seeks to describe the terrain of an entire planet by perturbing the surface of a sphere with an appropriate fractal function. If the terrain is to be realistic, however, the implicit surface cannot be allowed to split into separate disconnected pieces. This possibility is currently avoided with the use of procedural noise functions in the form $n(\mathbf{x}/\|\mathbf{x}\|)$, effectively turning the implicit surface into a procedural displacement map over the sphere.

An implicit fractal surface is very irregular. The attempt to render such a surface by first converting it into a polygon mesh would require a very high polygon count if the surface was to be represented with any reasonable fidelity [3,14]. The best way to visualise implicit fractal surfaces is to directly render them with ray casting. The ray casting algorithm must be guaranteed to find all correct ray intersections. Failure to provide such a guarantee would produce the familiar "surface acne" problem, which can potentially crop up in all rendering algorithms that rely on ray-surface intersection tests. Our algorithm evolves from the work of Mitchell where interval arithmetic was used to obtain estimates on the variation of the implicit surface's function along the ray [16]. However, we replace interval arithmetic (IA) with affine arithmetic (AA) since the latter is able to provide much tighter estimates for the aforementioned variation [6].

Ray casting with affine arithmetic was developed by de Cusatis Jr. et al. [4]. When comparing AA against IA, de Cusatis Jr. et al. reported mixed results for several textbook mathematical surfaces like the Steiner surface or the double torus. Our work focuses, instead, on implicit surfaces generated from specific classes of procedural noise functions that find much employment in the field of computer graphics [5]. We have found that a direct implementation of AA, as pro-

posed by de Cusatis Jr. et al., is less efficient than the IA implementation of Mitchell for ray casting implicit surfaces generated from fractal sums of procedural noise functions. This has motivated our use of a reduced representation for AA, which is just as accurate as the original one while being more efficient [15]. It is the use of this reduced AA representation for ray casting implicit surfaces based on procedural noise functions that is the focus of this paper.

Section 2 presents previous work in this area. Section 3 gives a general formulation for procedural noise and applies this formulation to three commonly used noise functions. Understanding how noise functions are procedurally evaluated is essential to understanding why the reduced AA representation works. Section 4 presents affine arithmetic and explains how it is used to solve the ray-surface intersection problem of ray casting. The reduced AA framework is then presented and shown to be a simple modification of the standard AA framework. Section 5 shows results and presents a comparison between reduced AA, standard AA and IA. Section 6 presents conclusions, suggests possible enhancements, and shows other areas where our technique can be successfully applied.

2 Previous Work

Many methods have been presented to solve the intersection problem between a ray and an implicit surface. We concentrate here on methods that are robust. These methods can always find the correct intersection point and are limited only by the floating point precision of the machine. A survey of such methods is given by Hart [8].

Robust implicit surface intersection methods were initially developed for surfaces with a simple and well known shape. If the function $f(\mathbf{x})$ is a polynomial then the implicit surface is said to be *algebraic* and the intersection points can be obtained with polynomial root finders [7]. Surfaces generated by sweeping a sphere along a curve, called *generalised cylinders*, and surfaces that are subject to non-linear deformations have also been considered [30,1].

Implicit surfaces based on the blending of compactly supported radial basis functions are popular because of their ability to model objects with complex topology. Many authors who have worked with this type of surface have also developed ray intersection algorithms for them. Such authors include Blinn with his *blobby model*, Nishimura et al. with *metaballs* and Wyvill and Trotmann with *soft objects* [2,20,33]. Sherstiuk has developed a general intersection method for surfaces generated from sums of compactly supported basis functions [26]. His method approximates any basis function with piecewise Hermite polynomials, the roots of which can then be found with analytical formulas.

Two general approaches can be followed to find the intersection between a ray and an implicit surface when the function f that generates the surface has an arbitrary shape. One approach is based on *Lipchitz bounds* and the other is based on *interval arithmetic*. Lipchitz bounds impose a limit on the maximum rate of change that f can take inside some region of space. Kalra and Barr successfully rendered *LG-surfaces* by advancing rays inside an octree structure [12]. Inside each cell of the octree, a Lipchitz bound L is used for f and another Lipchitz bound G is used for the derivative of f along the ray direction. Hart also uses Lipchitz bounds in his *sphere tracing* method [9]. Unlike Kalra and Barr, it is not necessary to employ Lipchitz bounds for the derivatives of f. The method works by marching along a ray with steps that are guaranteed not to cause intersection with the surface. In both the LG-surface method and in sphere tracing it is necessary to specify *a priori* Lipchitz bounds related to the function f that one wishes to use. That can be difficult in a general case although Kalra and Barr and also Hart present bounds for some commonly used functions. If the Lipchitz bounds are not optimal, these methods will converge more slowly.

Worley and Hart introduced several optimisations in the sphere tracing method for the case of implicit surfaces generated from hypertextures [32]. The improved sphere tracing method takes into account the fact that hypertextured objects are often generated from the sums of many procedural functions. Other optimisations include a spatial coherence technique to reduce the number of function evaluations and image coherence and overshooting techniques to increase the stepping size along the rays.

Mitchell computes ray-surface intersections with interval arithmetic [16]. Interval arithmetic (IA) is a framework that replaces arithmetic operators and function evaluations on real numbers with equivalent operators and functions that are evaluated on intervals [17]. With IA it is possible to obtain interval bounds for the variation of f along some arbitrary span along a ray. The method by Mitchell performs a recursive binary subdivision along the length of a ray, computing interval bounds for the function and its derivative inside each ray span. Newton's method is used to find the root once the interval bounds indicate the function has become monotonic inside some ray span. The method by Mitchell was later extended to use affine arithmetic (AA), instead of IA [4]. Ray casting with AA produces interval bounds that are much tighter than those obtained with IA, therefore increasing the efficiency of the intersection algorithm. One advantage of interval methods over Lipchitz methods is that interval bounds are computed automatically and on the fly. It is not necessary to supply some initial parameter, in the form of a conservative estimate for the Lipchitz bound, that will ultimately determine the efficiency of the algorithm.

3 Procedural Evaluation of Noise Functions

Procedural noise functions generate random fluctuations that possess a band-limited spectrum. These functions implement what is called *procedural noise* because it can be embodied as a procedure in a computer program. Procedural noise is commonly used as a building block to construct complex and natural looking textures, terrain elevation data and dy-

namic phenomena such as fire, water or clouds [5]. The key to the success of procedural noise functions is that they can be evaluated independently at any desired point in space.

The value of a procedural noise function n at some point $\mathbf{x}$ in $\mathbb{R}^3$ depends on the position of $\mathbf{x}$ relative to a discrete but infinite set $S = \{\mathbf{x}_i \in \mathbb{R}^3 : i = 0, 1, 2, \ldots\}$ of node points $\mathbf{x}_i$ that are distributed throughout space. Because S has an infinite number of node points, the evaluation of $n(\mathbf{x})$ is feasible when $n(\mathbf{x})$ is made to depend only on a small subset $S(\mathbf{x})$ of S. At each location $\mathbf{x}$, the subset $S(\mathbf{x})$ is the finite set of node points in S that surround $\mathbf{x}$ according to some specified criterion.

For our purposes, we can define the value of a procedural noise function n at $\mathbf{x}$ as a sum of kernel functions ϕ_k that depend on the displacement vectors between $\mathbf{x}$ and the node points in $S(\mathbf{x})$:

$$n(\mathbf{x}) = \sum_{k=0}^{L} \phi_k(\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_N), \qquad (1)$$

where $\mathbf{d}_j = \mathbf{x} - \mathbf{x}_j$ and $\mathbf{x}_j$, with $j = 0, 1, \ldots, N$, belongs to $S(\mathbf{x})$. The characteristics of each particular noise function come from the choice of several factors, namely:

– The shape of the kernels.
– The number L of kernels used.
– The criterion used to define $S(\mathbf{x})$.
– The distribution of the $\mathbf{x}_i$ in space to form S.

The random fluctuations exhibited by procedural noise result from the introduction of stochastic components into some of the previous factors. In some cases of procedural noise functions, the distribution of node points through space follows a desired probability density. Random variables are also often included in the definition of the kernel functions.

3.1 Perlin Gradient Noise Functions

Perlin's gradient noise function was the first procedural noise function to be proposed in the literature [22]. In this noise function, the node points $\mathbf{x}_i$ coincide with the vertices of a regular lattice placed at integer coordinate positions: $S = \{(u, v, w) : u, v, w \in \mathbb{Z}\}$. For each location $\mathbf{x}$, the set $S(\mathbf{x})$ is made of the eight node points at the vertices of the lattice cell in which $\mathbf{x}$ resides. There are eight kernels and each one depends on a single node point from $S(\mathbf{x})$. A kernel ϕ that depends on the displacement $\mathbf{d}_j = (x_j, y_j, z_j)$, relative to node point $\mathbf{x}_j$, is written as:

$$\phi(\mathbf{d}_j) = (\xi_1 x_j + \xi_2 y_j + \xi_3 z_j)\, h(x_j) h(y_j) h(z_j). \qquad (2)$$

The function h is a cubic hermite polynomial and the vector (ξ_1, ξ_2, ξ_3) is randomly distributed over the surface of a sphere with unit radius. There are several variations of Perlin's gradient noise function, which include Perlin's value noise function and value-gradient noise function, but these will not be described here [21]. They fit easily into formulation (1) with kernels that are similar to (2). Recently,

Perlin improved his gradient noise function by using quintic hermite polynomials for h and having (ξ_1, ξ_2, ξ_3) be a random vector that can only take values from a discrete set of vectors [23].

3.2 Sparse Convolution Functions

Sparse convolution noise functions were first proposed by Lewis [13]. As with Perlin's noise functions, a regular lattice placed at integer positions is also used. Inside each cell in this lattice, K node points are uniformly distributed. This simple scheme attempts to approximate a Poisson distribution of node points. The value of n at each location $\mathbf{x}$ depends on the node points of the cell that contains $\mathbf{x}$ plus the node points in the twenty six surrounding cells. The set $S(\mathbf{x})$, therefore, always contains $27K$ node points. There is an equal number of kernels, one for each node point. A kernel ϕ depends only on the distance $\|\mathbf{d}_j\|$ to its corresponding node point:

$$\phi(\mathbf{d}_j) = \xi h(\|\mathbf{d}_j\|). \qquad (3)$$

The scalar ξ is a gaussian random variable and the function h can take any shape as long as it is compactly supported on the interval $[0, 1]$. This last requirement is necessary to guarantee that only the $27K$ node points in the cell that contains $\mathbf{x}$ and in its immediate neighbours can possibly influence the value of $n(\mathbf{x})$.

Sparse convolution noise functions are more expensive to evaluate than gradient noise functions. A total of $27K$ kernels need to be evaluated against only eight for Perlin's gradient noise function. Sparse convolution functions, however, have the advantage of providing exact control over the frequency spectrum of the resulting noise. It is possible to show that the spectrum of n for sparse convolution noise functions is proportional to the spectrum of the function h that is chosen for (3) [13].

3.3 Cellular Texture Functions

Cellular texture functions, proposed by Worley, rely on a Voronoi decomposition of space based on the location of the $\mathbf{x}_i$ node points [31]. As with sparse convolution functions, an approximation to a Poisson distribution of node points is generated inside an integer lattice, although the technique used to achieve this effect is slightly different from the one employed by Lewis. The kernels for cellular texture noise functions consist of the ordered set of increasing distances between any location $\mathbf{x}$ and the node points. For some location $\mathbf{x}$, let $D(\mathbf{x}) = \{\|\mathbf{d}_i\| : i = 0, 1, 2 \ldots\}$ be the set of distances from $\mathbf{x}$ to all node points in S. Let also $p : \mathbb{N}_0 \to \mathbb{N}_0$ be a permutation of the indices in $D(\mathbf{x})$ so that the new set $\overline{D}(\mathbf{x}) = \{\|\mathbf{d}_j\| : j = p(i), i = 0, 1, 2 \ldots\}$ is ordered by increasing distances. The j-th kernel function is then taken as the j-th element in the ordered set $\overline{D}(\mathbf{x})$:

$$\phi_j = a_j \|\mathbf{d}_j\|, \qquad (4)$$

where a_j is some chosen scaling constant. Worley points out that only the kernel functions ϕ_0 to ϕ_3 are useful for texture synthesis. The ϕ_j with $j > 3$ resemble ϕ_3 and do not add significant new details. With the first four kernel functions, several interesting combinations are possible by choosing the appropriate values of the a_j constants. Kernel functions can also be turned off by setting $a_j = 0$.

The technique Worley employs to distribute the node points throughout the integer lattice guarantees that for every location $\mathbf{x}$ the four nearest node points will always be found inside the lattice cell that contains $\mathbf{x}$ or inside one of its neighbours. Similarly to sparse convolution noise functions, the set $S(\mathbf{x})$, used to compute ϕ_0 to ϕ_3, is then made of all the node points that are contained inside the $3 \times 3 \times 3$ cube of lattice cells that is centred on the cell containing $\mathbf{x}$.

4 Robust Ray Casting of Implicit Surfaces

Ray casting an implicit surface consists of determining the intersection point between the surface and any ray, parameterised as $\mathbf{r}(t) = \mathbf{o} + t\mathbf{l}$ with $t \geqslant 0$. Because the implicit surface is the zeroset of some function f, ray casting amounts to finding the first root of the non-linear equation:

$$f(\mathbf{r}(t)) = 0. \tag{5}$$

Let the parameter along the ray vary inside some interval $T = [t_{min}, t_{max}]$. For simplicity of notation, let us also introduce the auxiliary function $g = f \circ \mathbf{r}$. We wish to find an interval estimate $G(T)$ of the corresponding variation in $g(t)$ as t takes values from T. The function G is called an *interval extension* of g. The interval extension function provides information about the presence of roots of (5) inside some interval. If $0 \notin G(T)$ then no root can be present in T. If, on the other hand, $0 \in G(T)$, a root may or may not exist in T. This is because current techniques for computing G can only provide a conservative interval estimate that contains the true variation of g. The fact that $0 \in G(T)$ does not necessarily mean that $g(t) = 0$ for some $t \in T$. The best strategy in this case is to split T into smaller intervals and test the interval extension function on each of them.

Figure 1 lists a robust algorithm, called `RayIntersect`, that is used to find the first intersection point between a ray and the implicit surface. The algorithm relies on the subdivision of an initial interval T_0 and the information returned by the interval extension function G. A stack is used to store the subdivided intervals that are waiting to be tested for the existence of roots. The algorithm terminates either when a small enough interval bounding a root has been found or when the stack becomes empty. The latter scenario occurs in situations where a ray does not intersect the surface. Each interval taken from the stack is subdivided if there is the possibility that it may contain a root. The order with which the two subintervals are then pushed onto the stack is not arbitrary. By pushing T_r first and then T_l, the nearest intersection is guaranteed to be found.

push T_0 onto stack;
while stack not empty
    pop $T = [t_{min}, t_{max}]$ from the stack;
    if $0 \in G(T)$
        if $t_{max} - t_{min} < \varepsilon$
            return t_{min};
        let $t_i = (t_{max} + t_{min})/2$;
        let $T_l = [t_{min}, t_i]$;
        let $T_r = [t_i, t_{max}]$;
        push T_r onto stack;
        push T_l onto stack;

Fig. 1 The `RayIntersect` algorithm.

The `RayIntersect` algorithm is a simplified version of the interval algorithm by Mitchell [16]. In Mitchell's original algorithm, an interval extension G' of the derivative of g along the ray was also computed. When $0 \in G(T)$ and $0 \notin G'(T)$ were both verified, the function was known to have an isolated root inside the interval T and Newton's method could then be used to provide quadratic convergence. In the case of fractal combinations of procedural noise functions, however, g varies erratically and only for very small intervals do the conditions for monoticity exist that enable us to isolate a single root. We have found that the use of G' does not provide any speedup while ray casting fractal implicit surfaces and, in fact, slows down the algorithm since two interval extensions have to be computed instead of one. de Cusatis Jr. et al. reached the same conclusion for the type of implicit surfaces that they were interested in rendering [4].

4.1 Standard Affine Arithmetic

Affine arithmetic is a technique proposed by de Figueiredo and Stolfi [6]. This technique can provide accurate estimates for the interval extension function $G(T)$ featured in the `RayIntersect` algorithm. Affine arithmetic represents an improvement over the previous interval arithmetic technique [17]. 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 t_i:

$$\hat{t} = t_0 + t_1 e_1 + t_2 e_2 + \cdots + t_n e_n. \tag{6}$$

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 computation of affine operations on AA quantities does not result in the creation of any new error symbols[1]. For two AA quantities $\hat{u}$, $\hat{v}$ and a scalar α, the affine operations are:

$$\alpha\hat{u} = (\alpha u_0) + (\alpha u_1)e_1 + \cdots + (\alpha u_n)e_n,$$
$$\hat{u} \pm \alpha = (u_0 \pm \alpha) + u_1 e_1 + \cdots + u_n e_n, \tag{7}$$
$$\hat{u} \pm \hat{v} = (u_0 \pm v_0) + (u_1 \pm v_1)e_1 + \cdots + (u_n \pm v_n)e_n.$$

For non-affine operations, like multiplication or square root, a new error symbol must be introduced to express the non-linearity of the operator. The result of some non-affine operator is a new AA quantity $\hat{w} = w_0 + w_1 e_1 + \cdots + w_n e_n + w_k e_k$, where the extra error symbol e_k has been added to the representation. For example, if $\hat{w} = \hat{u}\hat{v}$, the coefficients of $\hat{w}$ are given by:

$$w_0 = u_0 v_0,$$
$$w_i = u_0 v_i + v_0 u_i, \quad \text{for } i = 1, \ldots, n,$$
$$w_k = \sum_{i=1}^{n} |u_i| \cdot \sum_{i=1}^{n} |v_i|. \tag{8}$$

The coefficient w_k in (8), associated with the newly inserted error symbol e_k, is positive and represents the magnitude of the error introduced by the linearisation of $\hat{u}\hat{v}$ into an affine form. The same property of w_k holds in the case of all the other non-affine operations.

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. 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 latter must all have a null coefficient for e_i. When implementing AA, cumbersome book-keeping routines are required to manage the large but sparse sequences of error coefficients[2]. This inefficiency associated with AA representations has been acknowledged by Stolfi and de Figueiredo [27]. They recommend that a procedure called *condensation* be periodically applied on an AA quantity when its sequence of error symbols grows too large. An AA quantity $\hat{u}$, with m error symbols, can be condensed to form another quantity $\hat{v}$, with $n < m$ error symbols, according to:

$$v_i = u_i, \quad \text{for } i = 0, \ldots, n-1,$$
$$v_n = \sum_{i=n}^{m} |u_i|. \tag{9}$$

Condensation brings some large AA quantity $\hat{u}$ down to a more manageable size but it also destroys the correlation information that was kept in the error symbols e_i, with

---

[1] Assuming that rounding errors are ignored. Otherwise, affine operations, just like their non-affine counterparts, require the insertion of a new error symbol that accounts for the rounding error of the operation.

[2] A simpler alternative would be to store in full all the coefficients of an AA quantity. This would be wasteful of memory, considering that many of those coefficients are zero. It would also be wasteful of CPU cycles. As example (8) shows, the computation of AA operations involves a loop over all the e_i error symbols. If the sequence of coefficients is sparse, many of the loop iterations become unnecessary.

$n < i \leqslant m$. This is not a problem if the aforementioned error symbols were unique to $\hat{u}$. The accuracy of subsequent computations is affected, however, if the e_i were being shared with other AA quantities that are involved in those computations.

4.2 Reduced Affine Arithmetic

The problem of having to deal with ever increasing sets of error symbols in standard AA has motivated our use of a reduced AA form for ray casting implicit surfaces made from procedural noise functions. Reduced affine arithmetic was proposed by Messine as the first of several possible extensions to affine arithmetic [15]. In his work, this first extension is called Affine Form 1 (AF1).

As equation (1) shows, procedural noise functions are built from sums of independent kernel functions. No correlations exist between the sequence of computations that are performed for any two kernel functions ϕ_i and ϕ_j during the computation of n. Correlations during the evaluation of a procedural noise function have a very localised nature and are isolated inside the sequence of computations for each individual kernel. The only global correlation that is expected to exist throughout the computation of n is related to the uncertainty with the position of the root t along a ray. This happens when n is embedded in the equation $g(t) = 0$ that must be solved by the ray caster.

Our reduced AA representation $\hat{t}$ is equivalent to an AF1 representation which considers only two error symbols: the symbol e_1, expressing the uncertainty along the ray, and the symbol e_2, which is always non-negative and expresses uncertainties involved in the computation of $\hat{t}$ alone. The error symbol e_1 is the only symbol that is shared between $\hat{t}$ and other AA quantities. The expression for $\hat{t}$ is:

$$\hat{t} = t_0 + t_1 e_1 + t_2 e_2. \tag{10}$$

The starting point for the computation of the interval extension $G(T)$, as part of the `RayIntersect` algorithm, is the conversion of the interval $T = [t_{min}, t_{max}]$ into the reduced AA form $\hat{t}$:

$$t_0 = (t_{max} + t_{min})/2,$$
$$t_1 = (t_{max} - t_{min})/2, \tag{11}$$
$$t_2 = 0.$$

Reduced AA operations are always followed by a condensation step to remove any extra error symbols that would have been introduced otherwise. Reduced AA can, therefore, be seen as a modification of affine arithmetic that employs an aggressive form of condensation. We present, as an example, the case of the multiplication between two reduced AA quantities $\hat{u}$ and $\hat{v}$. Originally, the result of this multiplication would give:

$$\hat{u}\hat{v} = u_0 v_0 + (u_0 v_1 + v_0 u_1)e_1 +$$

$$+ u_0 v_2 e_{2v} + v_0 u_2 e_{2u} + (|u_1| + |u_2|) \cdot (|v_1| + v_2)e_k, \tag{12}$$

where we have written the second error symbols of $\hat{u}$ and $\hat{v}$ as e_{2u} and e_{2v}, respectively, to make it clear that they are not correlated. The last three terms of (12) are condensed into a new symbol e_2 that is unique to the $\hat{u}\hat{v}$ quantity, giving:

$$\hat{u}\hat{v} = u_0 v_0 + (u_0 v_1 + v_0 u_1) e_1 +$$

$$+ (|u_0| v_2 + |v_0| u_2 + (|u_1| + u_2) \cdot (|v_1| + v_2)) e_2. \quad (13)$$

In practice, all operations in reduced AA are modified so that a condensation step is automatically built into them. The affine operators (7) now become:

$$\begin{aligned}
\alpha \hat{u} &= (\alpha u_0) + (\alpha u_1) e_1 + (|\alpha| u_2) e_2, \\
\hat{u} \pm \alpha &= (u_0 \pm \alpha) + u_1 e_1 + u_2 e_2, \\
\hat{u} \pm \hat{v} &= (u_0 \pm v_0) + (u_1 \pm v_1) e_1 + (u_2 + v_2) e_2.
\end{aligned} \quad (14)$$

The last stage in the computation of an interval extension for ray casting is the conversion of the reduced AA form $\hat{g} = g(\hat{t})$ into the interval $G(T) = [g_{min}, g_{max}]$, which allows the test $0 \in G(T)$ to be performed trivially:

$$\begin{aligned}
g_{min} &= g_0 - |g_1| - g_2, \\
g_{max} &= g_0 + |g_1| + g_2, \\
0 &\in G(T) \Leftrightarrow g_{min} \leqslant 0 \leqslant g_{max}.
\end{aligned} \quad (15)$$

An analysis of the localised nature of the correlations during AA computations, as part of the evaluation of a procedural noise function, requires that the kernels for each individual noise function be examined in turn. In the case of Perlin's gradient noise function, the kernel (2) has three AA multiplications, each of which would introduce new error symbols in a standard AA representation. However, once these three multiplications are performed, the evaluation of $\phi(\mathbf{d}_j)$ is complete and the new error symbols can be safely condensed. At the same time, the AA evaluation of the cubic hermite polynomial h is performed through a direct process of Chebyshev affine approximation rather than applying all the usual algebraic operations [27,11]. This means that no internal correlations have to be considered during the evaluation of h because the reduced AA result is computed in one single step. In the end, it is possible to say that the evaluation of $\phi(\mathbf{d}_j)$ with reduced AA does not lose any correlation information and has the same accuracy as standard AA.

In the case of Lewis's sparse convolution noise function, the kernel (3) depends only on the distance $\|\mathbf{d}_j\|$ to some node point $\mathbf{x}_j$. This distance computation features four non-affine operations, namely three squares and one square root operation. The distance $\|\mathbf{d}_j\|$, however, is involved in the computation of ϕ_j alone and does not influence the other ϕ_i kernels (with $i \neq j$) that are required for the evaluation of n. The condensation of the new error symbols from the evaluation of $\|\mathbf{d}_j\|$ does not, therefore, lead to any loss of accuracy. The evaluation of the function h is performed directly by Chebyshev approximation and, again, we can say that a reduced AA computation of ϕ_j is as accurate as a standard AA computation.
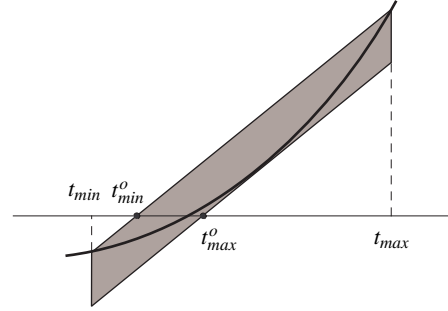

Fig. 2 The information conveyed by reduced affine arithmetic for the behaviour of the function g inside an interval $T = [t_{min}, t_{max}]$.

In the case of Worley's cellular texture function, the kernel is evaluated by iteratively applying a binary minimum operator $\min(l_i, l_j)$ on all the pairs of distances $l_k = \|\mathbf{d}_k\|$ from $\mathbf{x}$ to the node points that belong to the set $S(\mathbf{x})$. The minimum operator is evaluated with affine arithmetic according to the expression:

$$\min(l_i, l_j) = \frac{l_i + l_j}{2} - \frac{|l_i - l_j|}{2}. \quad (16)$$

For example, if $l_i > l_j$ we have $\min(l_i, l_j) = (l_i + l_j)/2 - (l_i - l_j)/2 = l_j$. This exact cancellation effect can only be achieved if all correlations between $l_i = \|\mathbf{d}_i\|$ and $l_j = \|\mathbf{d}_j\|$ are maintained. However, the distance computations with reduced AA involve the condensation of error symbols, as we have seen in the case of Lewis's sparse convolution noise function, and an exact cancellation cannot be obtained. For this reason, the application of reduced AA to cellular texture functions incurs a loss of accuracy. In Section 5, the loss of accuracy of reduced AA will be compared against its increased performance relative to standard AA for cellular texture functions.

4.3 Interval Optimisation

The idea of optimising the size of the interval bounding the first root of (5) was initially presented by de Cusatis Jr. et al. [4]. We present it here again in the framework of reduced affine arithmetic. When implementing the `RayIntersect` algorithm with affine arithmetic, it is possible to reduce the size of the interval T being tested at the start of each iteration, and prior to its subdivision, by taking advantage of the extra information provided by reduced AA.

Figure 2 shows an example of the information conveyed by a reduced AA representation of the function g, evaluated inside some interval $T = [t_{min}, t_{max}]$ along the ray, for a situation where g increases smoothly. The purpose of the ray casting algorithm is to find the point where the graph of g crosses the horizontal axis. The reduced AA representation $g(\hat{t})$, where $\hat{t}$ encodes T in reduced AA form according to

(11), is geometrically equivalent to a parallelogram that encloses the graph of g for the interval T. The bounding interval can be optimised by reducing it to $T^o = [t^o_{min}, t^o_{max}]$ prior to subdivision. It is clear from the drawing that significant convergence towards the root is achieved with just a single evaluation of $g(\hat{t})$ in reduced AA form.

The optimised interval T^o is obtained from the reduced AA representations $\hat{t} = t_0 + t_1 e_1$ for the interval and $g(\hat{t}) = g_0 + g_1 e_1 + g_2 e_2$ for the function g in the following way:

$$
\begin{aligned}
t^o_{min} &= \max\left(t_0 - \frac{g_0}{g_1}t_1 - \frac{g_2}{|g_1|}t_1, t_{min}\right), \\
t^o_{max} &= \min\left(t_0 - \frac{g_0}{g_1}t_1 + \frac{g_2}{|g_1|}t_1, t_{max}\right).
\end{aligned}
\tag{17}
$$

A derivation of these equations is given in Appendix A.

To summarise, we show here the steps necessary to compute the interval extension $G(T)$ in the `RayIntersect` algorithm of Figure 1, after the interval T has been removed from the stack:

1. Compute the reduced AA variable $\hat{t}$ from $T = [t_{min}, t_{max}]$, written as (10) and with coefficients given by (11).
2. Compute the reduced AA estimate $\hat{g} = g(\hat{t})$, using reduced affine arithmetic operators.
3. Compute the interval extension $G(T)$ from $\hat{g}$, using (15).

If, after step 3, it is found than $0 \in G(T)$, the optimised interval $T^o = [t^o_{min}, t^o_{max}]$ is obtained from the initial interval T, its reduced AA representation $\hat{t}$, and the estimate $\hat{g}$, using (17). It is T^o, rather than T, which is then subdivided and its subintervals pushed back onto the stack for further processing during subsequent iterations of the algorithm[3].

5 Results

A hypertextured implicit surface generated by the following function was used to test our reduced affine arithmetic ray casting method:

$$
f(\mathbf{x}) = \|\mathbf{x}\| - 1 + 0.6 \sum_{k=0}^{3} 2^{-0.8k} n(2^{k+2}\mathbf{x}).
\tag{18}
$$

The term $\|\mathbf{x}\| - 1$ is responsible for giving an overall spherical shape to the surface. The remaining summation on the right of (18) employs a procedural noise function n and represents the hypertexture, being responsible for the generation of all the surface detail. This summation produces a fractal surface with a dimension of 2.2, according to Saupe[4] [25].

---

[3] When $g_1 \to 0$, the enclosing parallelogram in Figure 2 tends toward an axis aligned rectangle. In the limit, no optimisation is possible and the original interval T must be subdivided.

[4] To be more precise, a fractal surface with dimension 2.2 would result if the summation had an infinite number of terms, with $k \in \mathbb{Z}$. As it stands, the function (18) produces a fractal surface only over a limited range of scales.

Table 1 Statistics for Perlin's gradient noise function.

	Avg. Evals. p/ray	Time
IA	78.40	$8m27.5s$
Standard AA	34.60	$36m22.9s$
Reduced AA	34.60	$5m57.1s$
Reduced AA + Int. Opt.	20.81	$3m43.2s$

Table 2 Statistics for Lewis's sparse convolution noise function.

	Avg. Evals. p/ray	Time
IA	48.44	$26m31.0s$
Standard AA	23.25	$32m24.7s$
Reduced AA	23.25	$10m47.2s$
Reduced AA + Int. Opt.	13.46	$6m29.3s$

Table 3 Statistics for Worley's cellular texture function.

	Avg. Evals. p/ray	Time
IA	45.70	$25m54.7s$
Standard AA	32.56	$2h56m15.0s$
Reduced AA	43.78	$29m39.9s$
Reduced AA + Int. Opt.	21.29	$14m25.9s$

Figure 3 shows, from left to right, a rendering of the implicit surface when n is Perlin's improved gradient noise function, Lewis's sparse convolution function with $K = 2$ (refer to Section 3.2) and Worley's cellular texture function with $a_0 = 1$ and $a_1 = a_2 = a_3 = 0$ (refer to Section 3.3). We have implemented a reduced affine arithmetic model of cellular texture functions that can use linear combinations of the ϕ_0 and ϕ_1 kernels only. We have found that the ϕ_2 and ϕ_3 kernels are too complex to implement when using AA. This is due to the difficulty in determining the third and fourth smallest distances in the set $S(\mathbf{x})$ when all the $\|\mathbf{d}_j\|$ have an arbitrary degree of uncertainty. For this reason, our current implementation of Worley's cellular texture functions must enforce the restriction $a_2 = a_3 = 0$.

Tables 1, 2 and 3 show some statistics that enable a comparison between all the interval estimation techniques for the procedural noise functions under consideration. We have compared the performance of interval arithmetic (IA), standard AA, reduced AA and reduced AA with interval optimisation. The rendering time was obtained for a 800×600 resolution image on a dual Athlon 2.1GHz processor. The average number of function evaluations per ray tells how often an interval extension $G(T)$ had to be computed as part of the `RayIntersect` algorithm. This statistic is a measure of the accuracy of each particular interval estimation technique. A more accurate technique causes the ray casting algorithm to converge to the intersection point with fewer iterations and fewer interval extension computations.

As expected, the IA intersection algorithm needs a large number of function evaluations due to the excessive conservativeness of IA estimates. Interval arithmetic, however, compensates for this lack of accuracy by being quite fast, which makes it competitive with some of the more advanced
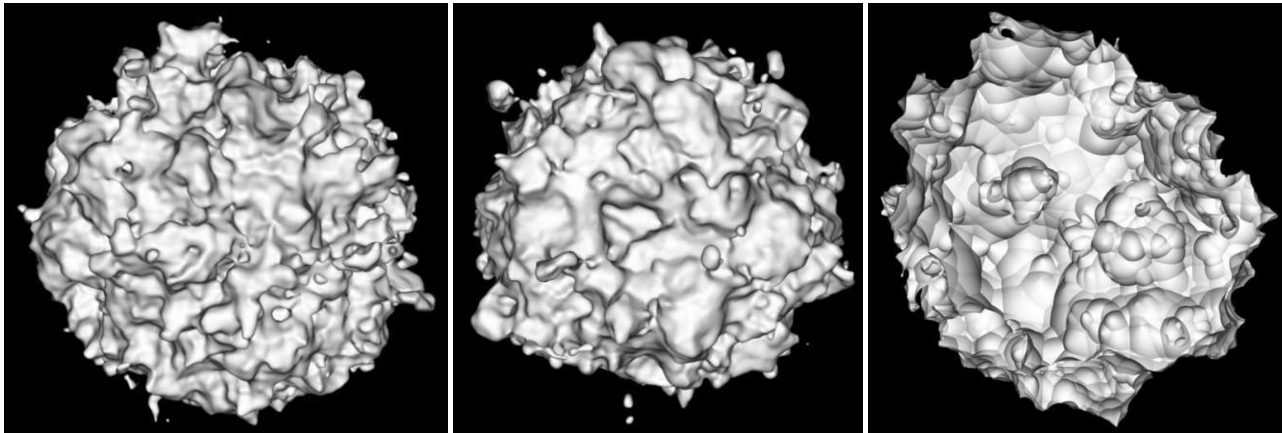
Fig. 3 An implicit surface representing a sphere that has been hypertextured with three layers of (from left to right) Perlin's gradient noise function, Lewis's sparse convolution noise function and Worley's cellular texture function.

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 coefficients 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 optimise ray-surface intersection algorithms, if only it can be implemented in a more efficient manner.

The better performance of IA over standard AA for the evaluation of procedural noise functions was acknowledged implicitly by Heidrich et al. [11]. In their work, IA was used for computing the interval estimates of a Perlin noise function. 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 function 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. As we had predicted in Section 4.2, no accuracy is lost by the use of reduced AA for the gradient noise function and the sparse convolution function. There is a loss of accuracy in the case of the cellular texture function, which is compensated by its increased computation speed so that, overall, reduced AA performs much better than standard AA for all three procedural noise functions. The final improvement comes from optimising the size of the intervals, as explained in Section 4.3. Reduced AA combined with interval optimisation gives the lowest rendering statistics of all interval estimation techniques.

Figure 4 shows a procedurally defined planet. The implicit surface uses a combination of the procedural noise functions studied in this paper. The faceted aspect of the terrain, in particular, is a consequence of the Voronoi regions created by the cellular texture function. In this example, no attempt was made to avoid the occurrence of disconnected pieces of terrain that arise naturally from the implicit representation, giving the planet a somewhat surrealistic look. This image took 6 hours and 37 minutes to render with reduced AA and

Fig. 4 A procedural planet represented as an implicit surface with a radius equal to the radius of the Earth and seen from an altitude of 100 metres. The implicit surface uses a mixture of all three procedural noise functions that are studied in this paper.

interval optimisation. Given the results in Tables 1 to 3, no attempt was made to render this image with any of the other interval estimation techniques. This is an example of a complex surface, with detail that is visible over a wide range of distances, that would have been impracticable to render with standard AA and whose rendering becomes feasible with reduced AA.

6 Conclusions and Future Developments

Ray casting implicit fractal surfaces with affine arithmetic becomes efficient only with the introduction of a reduced representation for uncertain quantities. The representation of an uncertain quantity with reduced affine arithmetic 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. Such procedural noise functions are based on the summation of several statistically independent terms. By maintaining only the correlation related to the uncertainty in the position of the root along the ray, reduced affine arithmetic can achieve the same results as standard affine arithmetic while being more efficient.

As predicted in Section 4.2 and subsequently confirmed in Section 5, the application of reduced AA to cellular texture functions incurs a loss of accuracy. This is ultimately due to the destruction of important correlation information through the condensation of error symbols. The loss of accuracy, however, is compensated by the greatly increased efficiency that comes from dealing with only two error symbols for each AA quantity, with a rendering time that drops from almost 3 hours with standard AA to only 29 minutes with reduced AA. It is possible, however, that for some other procedural noise functions, with kernels that we have not tested, the loss of accuracy may be more significant. In such a case, standard AA can be used for the calculation of the kernel $\phi(\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n)$ and a switch to reduced AA can be done for the remainder of the calculations in (1).

Standard and reduced AA quantities can easily be interchanged. Any reduced AA quantity is also a valid standard AA quantity that happens to have only two error symbols. The second error symbol e_2 has to be given a new and unique index number, after conversion to standard AA, to express the fact that it is not shared with any other standard AA quantities. A standard AA quantity can be transformed to a reduced AA quantity through condensation. For ray casting purposes, it is required that both AA representations agree that the common error symbol e_1 is used to express uncertainty relative to the position of the root along the ray. This is so that the interval optimisation procedure of Section 4.3 can be properly implemented. In a situation like that of Figure 4, where several procedural noise functions are used, the majority of the noise functions would be entirely computed with reduced AA while only the more problematic ones would use standard AA internally to compute their kernel functions.

Interpolating implicit surfaces have gained much popularity in recent years because of their ability to interpolate any set of constraint points [28, 18]. Interpolating implicit surfaces are now the method of choice to generate a continuous and smooth surface approximation from a set of scattered data points. From a structural point of view, interpolating implicit surfaces are entirely similar to sparse convolution noise functions, with equation (1) being used to sum the contribution of several independent radial basis functions (RBFs). Each RBF $\phi(\mathbf{d}_i)$ depends only on the distance $\|\mathbf{d}_i\|$ to constraint point $\mathbf{x}_i$. A RBF is written as $\phi(\mathbf{d}_i) = a_i h(\|\mathbf{d}_i\|)$, where h is some continuous and differentiable function. The difference between this RBF and (3) is only in the meaning of the scaling constant a_i or ξ, respectively. The a_i are pre-computed so as to cause the surface to interpolate through the required constraint points whereas ξ is the outcome of a random variable.

Given the structural similarity between sparse convolution noise functions and interpolating implicit surfaces made from sums of RBFs we can say that our reduced affine arithmetic method can also be used successfully to render the latter with ray casting. Current methods for rendering interpolating implicit surfaces resort to sphere tracing, where a Lipchitz bound must be supplied by the user before rendering takes place [9]. The optimal Lipchitz bound for a continuous and differentiable function f in three dimensions, such as the one generated through (1) for sums of RBFs, is the maximum magnitude $\|\nabla f\|$ of the gradient vector. For an arbitrary set of constraint points $\mathbf{x}_i$, this maximum gradient magnitude can only be found through a costly global optimisation procedure. Morse et al. avoid this procedure by evaluating $\|\nabla f\|$ at a large set of random points and using the maximum value thus obtained as their Lipchitz bound [18]. Clearly, this does not lead to a robust rendering algorithm as it is not possible to guarantee that all correct ray-surface intersections will be found. Problematic areas will be those where the value of f changes more rapidly than predicted by the Lipchitz bound that Morse et al. use. By ray casting interpolating implicit surfaces with reduced affine arithmetic, one has an automatic, robust, efficient, and verifiable method of computing all ray intersections without the burden of having to estimate the Lipchitz bound as a pre-computation step.

A Derivation of the Interval Optimisation Equation

When computing $\hat{g} = g(\hat{t})$, we have that $\hat{g} = g_0 + g_1 e_1 + g_2 e_2$ and $\hat{t} = t_0 + t_1 e_1$, where the error symbols e_1 and e_2 are unknown but vary in the interval $[-1, +1]$. Replacing the expression for $\hat{t}$ in the expression for $\hat{g}$, by way of the e_1 error symbol, we have:

$$\hat{g} = \frac{g_1}{t_1}(\hat{t} - t_0) + g_0 + g_2 e_2. \tag{A.1}$$

Equation (A.1) is the equation for a line in the $\hat{g}$-$\hat{t}$ plane with a slope of g_1/t_1. By letting e_2 vary in $[-1, +1]$ and keeping $t_{min} \leqslant \hat{t} \leqslant t_{max}$ we sweep the parallelogram that is shown in Figure 2. The upper and lower edges of this parallelogram are obtained from (A.1) when $e_2 = \pm 1$. The intersections of these two edges with the horizontal axis give the new and optimised limits for the interval. Setting (A.1) equal to zero, with $e_2 = \pm 1$, and rearranging, we have:

$$\hat{t} = t_0 - \frac{g_0}{g_1} t_1 \pm \frac{g_2}{g_1} t_1. \tag{A.2}$$

Independently of the sign of g_1 (g_2 is always positive and t_1 is also always positive because of they it is constructed in equation (11)), the

left and right solutions to (A.2) along the horizontal axis are, respectively:

$$t^o_{min} = t_0 - \frac{g_0}{g_1}t_1 - \frac{g_2}{|g_1|}t_1,$$
$$t^o_{max} = t_0 - \frac{g_0}{g_1}t_1 + \frac{g_2}{|g_1|}t_1. \tag{A.3}$$

We only use the results from (A.3) if they lead to a tighter interval than the original $[t_{min}, t_{max}]$, hence the min and max functions in (17).

References

1. Barr, A.H.: Ray tracing deformed surfaces. In: D.C. Evans, R.J. Athay (eds.) Computer Graphics (SIGGRAPH '86 Proceedings), pp. 287–296. ACM Press (1986)
2. Blinn, J.F.: A generalization of algebraic surface drawing. ACM Transactions on Graphics **1**(3), 235–256 (1982)
3. Bloomenthal, J.: Polygonisation of implicit surfaces. Computer Aided Geometric Design **5**(4), 341–355 (1988)
4. de Cusatis Jr., A., de Figueiredo, L.H., Gattas, M.: Interval methods for raycasting implicit surfaces with affine arithmetic. In: Proc. XII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI '99), pp. 65–71 (1999)
5. Ebert, D.S., Musgrave, F.K., Peachey, D.R., Perlin, K., Worley, S.P.: Texturing & Modeling: A Procedural Approach, 3rd edn. Morgan Kaufmann Publishers Inc. (2003)
6. de Figueiredo, L.H., Stolfi, J.: Affine arithmetic: Concepts and applications. Numerical Algorithms **37**(1–4), 147–158 (2004)
7. Hanrahan, P.: Ray tracing algebraic surfaces. In: P.P. Tanner (ed.) Computer Graphics (SIGGRAPH '83 Proceedings), pp. 83–90. ACM Press (1983)
8. Hart, J.: Ray tracing implicit surfaces. In: Modeling, Visualizing and Animating Implicit Surfaces, pp. 13.1–13.15 (1993). SIGGRAPH '93 Course Notes 25
9. Hart, J.C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. The Visual Computer **12**(9), 527–545 (1996)
10. Hart, J.C.: Implicit representation of rough surfaces. Computer Graphics Forum **16**(2), 91–99 (1997). ISSN 0167-7055
11. Heidrich, W., Slusallek, P., Seidel, H.: Sampling procedural shaders using affine arithmetic. ACM Transactions on Graphics **17**(3), 158–176 (1998)
12. Kalra, D., Barr, A.H.: Guaranteed ray intersections with implicit surfaces. In: J. Lane (ed.) Computer Graphics (SIGGRAPH '89 Proceedings), vol. 23, pp. 297–306. ACM Press (1989)
13. Lewis, J.P.: Algorithms for solid noise synthesis. In: J. Lane (ed.) Computer Graphics (SIGGRAPH '89 Proceedings), vol. 23, pp. 263–270. ACM Press (1989)
14. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3D surface construction algorithm. In: M.C. Stone (ed.) Computer Graphics (SIGGRAPH '87 Proceedings), vol. 21, pp. 163–169. ACM Press (1987)
15. Messine, F.: Extentions to affine arithmetic: Application to unconstrained global optimization. Journal of Universal Computer Science **8**(11), 992–1015 (2002)
16. Mitchell, D.P.: Robust ray intersection with interval arithmetic. In: Proceedings of Graphics Interface '90, pp. 68–74. Canadian Information Processing Society (1990)
17. Moore, R.: Interval Arithmetic. Prentice-Hall (1966)
18. Morse, B.S., Yoo, T.S., Chen, D.T., Rheingans, P., Subramanian, K.R.: Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions. In: B. Werner (ed.) Proceedings of the International Conference on Shape Modeling and Applications (SMI-01), pp. 89–98. IEEE Computer Society (2001)
19. Musgrave, F.K.: Mojoworld: Building procedural planets. In: D.S. Ebert, F.K. Musgrave (eds.) Texturing & Modeling: A Procedural Approach, 3rd edn., chap. 20, pp. 565–615. Morgan Kauffman Publishers Inc. (2003)
20. Nishimura, H., Hirai, M., Kawai, T., Kawata, T., Shirakawa, I., Omura, K.: Object modeling by distribution function and a method of image generation. Trans. IECE Japan, Part D **J68-D**(4), 718–725 (1985)
21. Peachey, D.R.: Building procedural textures. In: D.S. Ebert, F.K. Musgrave (eds.) Texturing & Modeling: A Procedural Approach, 3rd edn., chap. 2, pp. 7–94. Morgan Kauffman Publishers Inc. (2003)
22. Perlin, K.: An image synthesizer. In: B.A. Barsky (ed.) Computer Graphics (SIGGRAPH '85 Proceedings), pp. 287–296. ACM Press (1985)
23. Perlin, K.: Improving noise. ACM Transactions on Graphics (SIGGRAPH '02 Proceedings) **21**(3), 681–682 (2002)
24. Perlin, K., Hoffert, E.M.: Hypertexture. In: J. Lane (ed.) Computer Graphics (SIGGRAPH '89 Proceedings), vol. 23, pp. 253–262. ACM Press (1989)
25. Saupe, D.: Point evaluation of multi-variable random fractals. In: H. Jüergens, D. Saupe (eds.) Visualisierung in Mathematik und Naturissenschaften - Bremer Computergraphik Tage, pp. 114–126. Springer-Verlag (1989)
26. Sherstyuk, A.: Fast ray tracing of implicit surfaces. Computer Graphics Forum **18**(2), 139–147 (1999)
27. Stolfi, J., de Figueiredo, L.H.: Self-validated numerical methods and applications (1997). Course notes for the 21st Brazilian Mathematics Colloquium
28. Turk, G., O'Brien, J.F.: Modelling with implicit surfaces that interpolate. ACM Transactions on Graphics **21**(4), 855–873 (2002)
29. Voss, R.F.: Fractals in nature: From characterization to simulation. In: H.O. Peitgen, D. Saupe (eds.) The Science of Fractal Images, chap. 1, pp. 21–70. Springer-Verlag (1988)
30. van Wijk, J.J.: Ray tracing objects defined by sweeping a sphere. Computers & Graphics **9**(3), 283–290 (1985)
31. Worley, S.P.: A cellular texture basis function. In: H. Rushmeier (ed.) Computer Graphics (SIGGRAPH '96 Proceedings), vol. 30, pp. 291–294. ACM Press (1996)
32. Worley, S.P., Hart, J.C.: Hyper-rendering of hyper-textured surfaces. In: Proc. of Implicit Surfaces '96, pp. 99–104 (1996)
33. Wyvill, G., Trotman, A.: Ray-tracing soft objects. In: Computer Graphics International'90, pp. 469–475 (1990)