A Progressive Refinement Approach for the Visualisation of Implicit Surfaces

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

Visualising implicit surfaces with the ray casting method is a slow procedure. The design cycle of a new implicit surface is, therefore, fraught with long latency times as a user must wait for the surface to be rendered before being able to decide what changes should be introduced in the next iteration.

In this paper, we present an attempt at reducing the design cycle of an implicit surface modeler by introducing a progressive refinement rendering approach to the visualisation of implicit surfaces. This progressive refinement renderer provides a quick previewing facility. It first displays a low quality estimate of what the final rendering is going to be and, as the computation progresses, increases the quality of this estimate at a steady rate.

The progressive refinement algorithm is based on the adaptive subdivision of the viewing frustrum into progressively smaller cells. An estimate for the average value and the variance of the implicit function inside each cell is obtained with an affine arithmetic range estimation technique. Overall, we show that our progressive refinement approach not only provides the user with visual feedback as the rendering advances but is also capable of completing the image faster than a conventional implicit surface rendering algorithm based on ray casting.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Interval arithmetic*; G.1.2 [**Numerical Analysis**]: Approximation—*Approximation of surfaces and contours*; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*Viewing algorithms*; 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 surface, progressive refinement, 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 [Desbrun and Gascuel 1995; Whitaker 1998; Foster and Fedkiw 2001]. An implicit surface is defined as the set of all points $\mathbf{x}$ that verify the condition $f(\mathbf{x}) = 0$ for some function $f(\mathbf{x})$ from $\mathbb{R}^3$ to $\mathbb{R}$. Modelling with implicit surfaces amounts to the construction of an appropriate function $f(\mathbf{x})$ that will generate the desired surface.

Rendering algorithms for implicit surfaces can be broadly divided into meshing algorithms and ray casting algorithms. Meshing algorithms convert an implicit

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.

surface to a polygonal mesh format, which can be subsequently rendered in real time with modern graphics processor boards [Lorensen and Cline 1987; Bloomenthal 1988; Velho 1996]. 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 and finding their intersection with the surface [Roth 1982; Hin et al. 1989].

Our ultimate goal is to use implicit surfaces as a tool to model and visualise realistic procedural planets over a very wide range of scales. The function $f(\mathbf{x})$ that generates the surface terrain for such a planet must have fractal scaling properties and exhibit a large amount of small scale detail. Examples of this type of terrain generating function can be found in Ebert et al. [2003]. In our planet modelling scenario, meshing algorithms are too cumbersome as they generate meshes with a very high polygon count in order to preserve all the visible surface detail. Furthermore, as the viewing distance changes, the amount of surface detail varies accordingly and the whole polygon mesh needs to be regenerated. For these reasons, we have preferred a ray casting approach because of its ability to render the surface directly without the need for an intermediate polygonal representation.

The visualisation of an implicit surface with ray casting is not without its problems, however. When the surface is complex, many iterations have to be performed along each ray in order to locate the intersection point with an acceptable accuracy [Mitchell 1990]. Imaging an implicit surface with ray casting can then become a slow procedure. This is further compounded by the fact that an anti-aliased image requires that many rays be shot for each pixel [Cook 1989].

We propose to alleviate the long rendering times associated with the modelling and subsequent ray casting of complex fractal surfaces by providing a quick previewer based on a progressive refinement rendering principle. The idea of progressive refinement for image rendering was first formalised in Computer Graphics by Bergman et al. [1986]. Progressive refinement rendering has received much attention in the fields of radiosity and global illumination [Cohen et al. 1988; Farrugia and Peroche 2004]. Progressive refinement approaches to volume rendering have also been developed [Laur and Hanrahan 1991; Lippert and Gross 1995]. Our previewer uses progressive rendering to visualise an increasingly better approximation to the final implicit surface. It allows the user to make quick editing decisions without having to wait for a full ray casting solution to be computed. Because the rendering is progressive, the previewer can be terminated as soon as the user is satisfied or not with the look of the surface. The previewer is also capable of completing the image in a shorter amount of time than it would take for a ray caster to render the same image by shooting a single ray through the centre of each pixel.

In Section 2 we describe our progressive refinement algorithm. We begin by introducing the reduced affine arithmetic framework for computing range estimates in Section 2.1. Section 2.2 describes how affine arithmetic is applied inside portions of the camera's viewing frustrum that we call *cells*. The processes of cell subdivision (Section 2.3) and rendering (Section 2.4) are then described. Section 3 presents results and Section 4 provides conclusions and some hints for future developments.

2. RENDERING IMPLICIT SURFACES WITH PROGRESSIVE REFINEMENT

The main stage of our method consists in the binary subdivision of the space, visible from the camera, into progressively smaller cells that are known to straddle the boundary of the surface. The subdivision mechanism stops as soon as the projected size of a cell on the screen becomes smaller than the size of a pixel. Information about the behaviour of the implicit function $f(\mathbf{x})$ inside a cell is returned by evaluating the function with affine arithmetic [Comba and Stolfi 1993]. Affine arithmetic is a framework for evaluating algebraic functions with arguments that are bounded but otherwise unknown. It is a generalisation of the older interval arithmetic framework [Moore 1966]. Affine arithmetic, when compared against interval arithmetic, is capable of returning much tighter estimates for the variation of a function, given input arguments that vary over the same given range. Affine arithmetic has been used with success in an increasing number of Computer Graphics problems, including the ray casting of implicit surfaces [Heidrich and Seidel 1998; Heidrich et al. 1998; de Figueiredo 1996; Junior et al. 1999; de Figueiredo et al. 2003]. We use a recently developed form of affine arithmetic that was termed *reduced affine arithmetic* [Gamito and Maddock 2005]. In the context of ray casting implicit surfaces made from procedural fractal functions, reduced affine arithmetic returns the same results as standard affine arithmetic while being faster to compute and requiring smaller data structures.

The procedure for rendering implicit surfaces with progressive refinement can be broken down into the following steps:

(1) Build an initial cell coincident with the camera's viewing frustrum. The near and far clipping planes are determined so as to bound the implicit surface.

(2) Recursively subdivide this cell into smaller cells. Discard cells that do not intersect with the implicit surface. Stop subdivision if the size of the cell's projection on the image plane falls below the size of a pixel.

(3) Assign the shading value of a cell to all pixels that are contained inside its projection on the image plane. The shading value for a cell is taken from the evaluation of the shading model at the centre point of the cell.

The `Rayshade` public domain ray tracer by Craig Kolb and Rod Bogart implements a previewing tool that traces rays at the corners of rectangular regions in the image and subdivides these regions based on their estimated contrast [Kolb 1992]. Since the ray casting of implicit surfaces is one instance of the more general ray tracing problem, the same previewing mechanism could be used here. In a ray tracer, the process of ray-surface intersection is performed independently for each ray, with no information being shared between neighbouring rays. When a cell is subdivided, on the other hand, the information about the implicit surface contained in that cell is refined and passed down to the child cells. A previewer for implicit surfaces that uses cell subdivision will, therefore, converge more quickly than a generic previewer that uses subdivision in image space, as implemented in `Rayshade`.

Another previewing method for implicit surfaces with some similarities to ours was briefly described in de Figueiredo and Stolfi [1996]. They also propose a binary subdivision of space, using an octree, to track the boundary of the surface with in-

creasing accuracy. Surface visibility is determined by a painters algorithm whereby the octree is traversed in back to front order, relative to the camera, as the cells are rendered on the screen. Our method is more efficient and less memory intensive in that only the cells that are known to be be visible from the camera are considered. No time is wasted tracking and subdividing cells that lie on hidden portions of the surface. The following sections will explain how each of the steps in our rendering method work.

2.1 Reduced Affine Arithmetic

A variable a is represented with standard affine arithmetic (sAA) as a *central value* a_0 plus a series of *noise symbols* e_i of unknown value but bounded inside the interval $[-1, +1]$ and weighted by *noise coefficients* a_i:

$$\hat{a} = a_0 + a_1 e_1 + a_2 e_2 + \cdots + a_n e_n. \tag{1}$$

The notation $\hat{a}$ means that the exact value of a is unknown and that an affine arithmetic representation for that variable is being used. The noise symbols e_i in the representation $\hat{a}$ traduce the uncertainty associated with that variable. These noise symbols can be shared with other variables, possibly weighted by different noise coefficients [Comba and Stolfi 1993]. Through the sharing of noise symbols, sAA is able to maintain correlation information between related variables, leading to estimates that are tighter and more accurate than their equivalent interval arithmetic counterparts. The problem with sAA, however, is that the sequence of noise symbols for some variable a grows without bounds as more and more non-linear operations are performed on it. Because the representation in (1) is linear, the execution of an operation like, for example, a square root requires the addition of a new noise symbol e_{n+1} that accounts for the non-linear part of the operation. On long chains of computations, the performance of sAA degrades and its memory requirements increase as new noise symbols keep being added to the system.

In the reduced affine arithmetic framework (rAA), a variable a contains only a fixed small set of noise symbols that are related to the fundamental degrees of freedom of the problem under consideration. An extra noise symbol e_k is included in the representation of a to account for uncertainties in that variable that are not shared with any other variable. For example, in the context of a conventional ray casting system for implicit surfaces, a rAA variable would be represented as $\hat{a} = a_0 + a_t e_t + a_k e_k$, where t is the distance along a ray. Gamito and Maddock [2005] explain how non-linear operations on rAA variables are performed by updating the a_t coefficient with new uncertainties related to the distance along the ray and clumping all other uncertainties into the a_k coefficient.

In the rendering method that is being described in this paper, the degrees of freedom are the three parameters necessary to locate any point inside the viewing frustrum of the camera. These parameters are the horizontal distance u along the image plane, the vertical distance v along the same image plane and the distance t along the ray that passes through the point (u, v). A rAA variable $\hat{a}$ has, therefore, the following representation:

$$\hat{a} = a_0 + a_u e_u + a_v e_v + a_t e_t + a_k e_k. \tag{2}$$

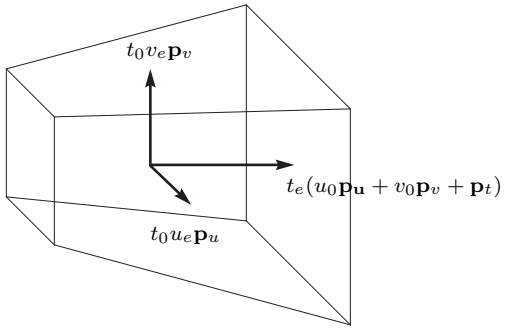

Fig. 1. The geometry of a cell. The vectors show the three medial axes of the cell.

For an implicit surface, the vector $f(\mathbf{x})$ at some point $\mathbf{x}$ in space can be described in rAA format as a tuple of three cartesian coordinates, with the representation in (2), and where the noise symbols e_u, e_v and e_t are shared between the coordinates. Each coordinate has its own independentent noise symbol e_{k_i}, with $i = 1, 2, 3$. The rAA representation $\hat{\mathbf{x}}$ of the vector $\mathbf{x}$ describes not a point but a region of space spanned by the uncertainties associated with its three coordinates. Evaluation of the expression $\hat{y} = f(\hat{\mathbf{x}})$ leads to a range estimate $\hat{y}$ for the variance of $f(\hat{\mathbf{x}})$ inside the region spanned by $\hat{\mathbf{x}}$. Knowing $\hat{y}$, the average value $\bar{y}$ and the variance $\langle y \rangle$ for that range estimate can be computed as follows, based on the representation in (2) for an rAA variable:

$$\bar{y} \triangleq y_0, \tag{3a}$$

$$\langle y \rangle \triangleq |y_u| + |y_v| + |y_t| + |y_k|. \tag{3b}$$

The range estimate $\hat{y}$ is then known to lie inside the interval $[\,\bar{y} - \langle y \rangle,\, \bar{y} + \langle y \rangle\,]$. If this interval contains zero, the region spanned by $\hat{\mathbf{x}}$ may or may not intersect with the implicit function. This is because affine arithmetic (both in its standard and reduced forms) always computes conservative range estimates and it is possible that the exact range resulting from $f(\hat{\mathbf{x}})$ may be smaller than $\hat{y}$. What is certain is that if $[\,\bar{y} - \langle y \rangle,\, \bar{y} + \langle y \rangle\,]$ does not contain zero the region spanned by $\hat{\mathbf{x}}$ is either completely inside or completely outside the implicit surface and therefore does not intersect with it.

2.2 The Anatomy of a Cell

A cell is a portion of the camera's viewing frustrum that results from a recursive subdivision along the u, v and t parameters. Figure 1 depicts the geometry of a cell. It has the shape of a truncated pyramid of quadrangular cross-section, similar to the shape of the viewing frustrum itself. Four vectors, taken from the camera's viewing system, are used to define the spatial extent of a cell. These vectors are:

The vector $\mathbf{o}$. This is the location of the camera in the world coordinate system.

The vector $\mathbf{p}_u$. Represents the horizontal direction along the image plane. The length of this vector gives the width of a pixel in the image plane.

The vector $\mathbf{p}_v$. Represents the vertical direction along the image plane. The length of this vector gives the height of a pixel in the image plane. It is often called the *up vector*.

The vector $\mathbf{p}_t$. It is the vector from the camera's viewpoint and orthogonal to the image plane. The length of this vector gives the distance from the viewpoint to the image plane.

The vectors $\mathbf{p}_u$, $\mathbf{p}_v$ and $\mathbf{p}_t$ define a left-handed perspective viewing system. The position of any point $\mathbf{x}$ inside the cell is given by the following inverse perspective transformation:

$$\mathbf{x} = \mathbf{o} + (u\mathbf{p}_u + v\mathbf{p}_v + \mathbf{p}_t)t = \mathbf{o} + ut\mathbf{p}_u + vt\mathbf{p}_v + t\mathbf{p}_t. \tag{4}$$

The spatial extent of a cell is obtained from the above by having the u, v and t parameters vary over appropriate intervals $[\,u_a, u_b\,]$, $[\,v_a, v_b\,]$ and $[\,t_a, t_b\,]$. We must consider how to compute the rAA representation $\hat{\mathbf{x}}$ of this spatial extent. To do so a change of variables must first be performed. The rAA variable $\hat{u} = u_0 + u_e e_u$ will span the same interval $[\,u_a, u_b\,]$ as u does if we have:

$$u_0 = (u_b + u_a)/2, \tag{5a}$$

$$u_e = (u_b - u_a)/2. \tag{5b}$$

Similar results apply for the v and t parameters. Substituting $\hat{u}$, $\hat{v}$ and $\hat{t}$ in (4) for u, v and t, we obtain:

$$\begin{aligned}\mathbf{x} = \mathbf{o} &+ t_0u_0\mathbf{p}_u + t_0v_0\mathbf{p}_v + t_0\mathbf{p}_t + \\ &+ t_0u_ee_u\mathbf{p}_u + t_0v_ee_v\mathbf{p}_v + u_0t_ee_t\mathbf{p}_u + v_0t_ee_t\mathbf{p}_v + t_ee_t\mathbf{p}_t \\ &+ t_eu_ee_ue_t\mathbf{p}_u + t_ev_ee_ve_t\mathbf{p}_v.\end{aligned} \tag{6}$$

The first line of (6) contains only constant terms. The second line contains linear terms of the noise symbols e_u, e_v and e_t. The third line contains two non-linear terms e_ue_t and e_ve_t, which are a consequence of the non-linearity of the perspective transformation. Since a rAA representation cannot accommodate such non-linear terms they are replaced by the independent noise terms e_{k_1}, e_{k_2} and e_{k_3} for each of the three cartesian coordinates of $\hat{\mathbf{x}}$. The rAA vector $\hat{\mathbf{x}}$ is finally given by:

$$\begin{aligned}\hat{\mathbf{x}} = \mathbf{o} &+ t_0(u_0\mathbf{p}_u + v_0\mathbf{p}_v + \mathbf{p}_t) + \\ &+ t_0u_e\mathbf{p}_ue_u + t_0v_e\mathbf{p}_ve_v + t_e(u_0\mathbf{p_u} + v_0\mathbf{p}_v + \mathbf{p}_t)e_t + \begin{bmatrix} x_{k_1}e_{k_1} \\ x_{k_2}e_{k_2} \\ x_{k_3}e_{k_3}\end{bmatrix},\end{aligned} \tag{7}$$

with

$$x_{k_i} = |t_eu_ep_{u_i}| + |t_ev_ep_{v_i}|, \quad i = 1, 2, 3. \tag{8}$$

A consequence of the non-linearity of the perspective projection and its subsequent approximation with rAA is that the region spanned by $\hat{\mathbf{x}}$ is going to be larger than the spatial extent of the cell. Figure 2 shows the geometry of a cell and the region spanned by its rAA representation in profile. Because the rAA representation has been linearised, its spatial extent is a prism rather than a truncated pyramid. This has further consequences in that the evaluation of $f(\hat{\mathbf{x}})$ and

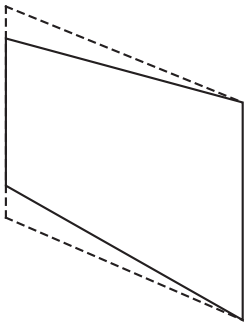

Fig. 2. The outline of a cell (solid line) and the outline of its rAA representation (dashed line) shown in profile. The rAA representation is a prism that forms a tight enclosure of the cell.

$g(\hat{\mathbf{x}}, \hat{\mathbf{n}}, \hat{\mathbf{v}})$ is going to include information from the regions of the prism outside the cell and will, therefore, lead to range estimates that are larger than necessary. The linearisation error is more pronounced for cells that exist early in the subdivision process. As subdivision continues and the cells become progressively smaller, their geometry becomes more like that of a prism and the discrepancy with the geometry of $\hat{\mathbf{x}}$ decreases[1].

The subdivision of a cell proceeds by first choosing one of the three perspective projection parameters u, v or t and splitting the cell in half along that parameter. This scheme leads to a k-d tree of cells where the sequence of dimensional splits is only determined at run time. The choice of which parameter to split along is based on the average width, height and depth of the cell:

$$\bar{w}_u = 2\, t_0 u_e \|\mathbf{p}_u\|, \tag{9a}$$

$$\bar{w}_v = 2\, t_0 v_e \|\mathbf{p}_v\|, \tag{9b}$$

$$\bar{w}_t = 2\, t_e \|u_0 \mathbf{p_u} + v_0 \mathbf{p}_v + \mathbf{p}_t\|. \tag{9c}$$

If, say, $\bar{w}_u$ is the largest of these three measures, the cell is split along the u parameter. The two child cells will have their u parameters ranging inside the intervals $[\, u_a, u_0 \,]$ and $[\, u_0, u_b \,]$, where $[\, u_a, u_b \,]$ was the interval spanned by u in the mother cell. In practice, the factors of 2 in (9) can be ignored without changing the outcome of the subdivision. This subdivision strategy ensures that, after a few iterations, all the cells will have an evenly distributed shape, even when the initial cell is very long and thin.

2.3 The Process of Cell Subdivision

Cell subdivision is implemented in an iterative manner rather than using a recursive procedure. A priority queue is used to store the cells waiting to be subdivided. The cells are kept sorted on this priority queue in descending order relative to the variance $\langle y \rangle = \langle f(\hat{\mathbf{x}}) \rangle$. The cell at the top of the queue has the largest amount of variance. The subdivision of this cell leads to an optimal refinement step towards an increased image quality. The algorithm starts by placing the initial cell, which

[1] This can be demonstrated by the fact that the terms $t_e u_e$ and $t_e v_e$ in (6) decrease more rapidly than any of the linear terms u_e, v_e and t_e of the same equation as the latter converge to zero.

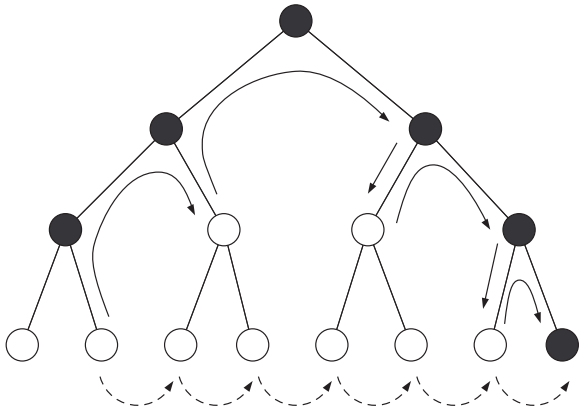

Fig. 3. Scanning along the depth subdivision tree. Cells represented by black nodes may intersect with the surface. Cells represented by white nodes do not. The solid arrows show progression by depth-first order. The dotted arrows show progression by breadth-first order.

corresponds to the complete viewing frustrum, on the priority queue. At the start of every new iteration, a cell is removed from the top of the queue. If the extent of the cell's projection on the image plane is larger than the extent of a pixel, the cell is subdivided and its two children are examined. In the opposite case, the cell is considered a leaf cell and is discarded after being rendered. The two conditions that indicate whether a cell should be subdivided are:

$$u_b - u_a > 1, \tag{10a}$$

$$v_b - v_a > 1. \tag{10b}$$

The values on the right hand sides of (10) are a consequence of the definition of $\mathbf{p}_u$ and $\mathbf{p}_v$ in Section 2.2, which cause all pixels to have a unit width and height.

The sequence of events after a cell has been subdivided depends on which of the parameters u, v or t was used to perform the subdivision. If the subdivision occurred along t, there will be two child cells with one in front of the other and totally occluding it. The front cell is first checked for the condition $0 \in f(\hat{\mathbf{x}})$. If the condition holds, the cell is pushed into the priority queue and the back cell is ignored. If the condition does not hold, the back cell is also checked for the same condition. The difference now is that, if $0 \notin f(\hat{\mathbf{x}})$ for the back cell, a new cell must be searched by marching along the t direction. The first cell scanned, at the same subdivision level of the front and back cells, for which $0 \in f(\hat{\mathbf{x}})$ holds is the one that is pushed into the priority queue. On the other hand, if the subdivision occurred along the u or v directions, there will be two child cells that sit side by side relative to the camera without occluding each other. Both cells are processed in the same way. If, for any of the two cells, $0 \in f(\hat{\mathbf{x}})$ holds, that cell is placed on the priority queue, otherwise a farther cell must be searched by marching forward in depth.

The process of marching forward from a cell along the depth direction t tries to find a new cell, at the same level of subdivision of the original cell, that has a possibility of intersecting the implicit surface by verifying the condition $0 \in f(\hat{\mathbf{x}})$. The process is invoked when the starting cell has been determined not to verify

the same condition. The reason for having this scanning in depth is because cells that do not intersect with the surface must be discarded. Only cells that verify $0 \in f(\hat{\mathbf{x}})$ are allowed into the priority queue for further processing. Figure 3 shows an example of this marching process. The scanning is performed by following a depth-first ordering relative to the tree that results from subdividing in t. The scanning sequence skips over the children of cells for which $0 \notin f(\hat{\mathbf{x}})$. The possibility of scanning in breadth-first order, by marching along all the cells at the same level of subdivision, is not recommended because in deeply subdivided trees a very high number of cells would have to be tested.

2.4 Rendering a Cell

The shading value of a cell is obtained from evaluating the shading function at the centre of the cell. The central point $\mathbf{x}_0$ for the cell is determined from (7) to be:

$$\mathbf{x}_0 = \mathbf{o} + t_0(u_0\mathbf{p}_u + v_0\mathbf{p}_v + \mathbf{p}_t). \tag{11}$$

During rendering, the shading value of a cell is assigned to all the pixels that are contained within its image plane projection. The centre of a pixel (i, j) occupies the coordinates $\mathbf{c}_{ij} = (i + 1/2, j + 1/2)$ on the image plane. All the pixels that verify $\mathbf{c}_{ij} \in [u_a, u_b] \times [v_a, v_b]$ for the cell being rendered will be assigned its shading value. Any previous shading values stored in these pixels will be overwritten. This process happens after cell subdivision and before the newly subdivided cells are placed on the priority queue. The subdivided cells will overwrite the shading value of their mother cell on the image buffer. The same process also takes place for leaf cells before they are discarded. In this way, the image buffer always contains the best possible representation of the image at the start of every new iteration.

2.5 Some Remarks on Implementation

The best implementation strategy for our rendering method is to have an application that runs two threads concurrently: a subdivision thread and a rendering thread. The rendering thread is responsible for periodically updating the graphical output of the application with the latest results from the subdivision thread. The subdivision thread requires read-write access to the image buffer, while the rendering thread requires read-only access. A timer is used to keep a constant frame refresh rate. Except for the periodical invocation of the timer handler routine, the rendering thread remains in a sleep state so that the subdivision thread can use all the CPU resources.

It is possible that, on machines with a small amount of main memory, excessive paging may occur due to the need to store a large number of cells in the priority queue. We have implemented our application on a Pentium 4 1.8GHz with 1Gb of memory. All the results shown in the next section were tested on this computer and it was found that the use of swap memory was never necessary. In any case, it is advisable that the data structure used to hold a cell be as light as possible.

3. RESULTS

Figure 4 shows several stages in the progressive refinement rendering of a 800×600 image of an implicit sphere modulated with a Perlin procedural noise function [Perlin 2002]. The large scale features of the surface become settled quite early and

the latter stages of the progression are mostly concerned with resolving small scale details. Figure 5 shows the plot of the variance $\langle y \rangle$ for the cell at the top of the priority queue versus the number of iterations, concerning the progression of Figure 4. The spikes in this plot occur when the depth marching procedure, explained in Section 2.3, is invoked. When a cell is replaced by a more distant cell, the farthest cell will occupy a larger volume as a consequence of the perspective projection transformation. The disparity in the size of the two cells becomes more pronounced the farther away they are from each other. Since the new cell is bigger, it will contain a larger variance of the implicit function. It will go straight to the top of the priority queue and cause a spike in the plot. This new cell will then be subdivided during the next few iterations and the log curve will resume its original behaviour.

Table I. Rendering statistics for an implicit sphere with several layers of Perlin noise.

Layers	*Iterations*	*Time*	*Raycasting*
1	350759	27.8s	1m10.4s
2	349465	1m16.8s	4m16.7s
3	359659	3m01.5s	8m51.7s

Figure 6 shows the same surface of Figure 4 with two and three layers, respectively, of a Perlin noise function. Table I shows the total number of iterations and the computation time for these three figures. The table also shows the computation time for ray casting the same Figures by shooting a single ray through the centre of each pixel. The number of iterations required to complete the progressive rendering algorithm is largely independent of the complexity of the surface. It depends only on the image resolution and on the percentage of the image that is covered by the projected surface. As estimated by the results in Table I, previewing by progressive refinement is on average 65% faster than previewing by ray casting without anti-aliasing. It should be added that these numbers do not entirely reflect the reality of the situation because, as demonstrated in the example of Figure 4, progressive refinement previewing already gives an accurate rendering of the surface at early stages of refinement. From a perceptual point of view, therefore, the difference between the two previewing techniques is greater than what is shown in Table I.

Figure 7 shows, on the left, an implicit sphere modulated with a sparse convolution noise [Lewis 1989]. On the right of this figure, a variance map is displayed, which was obtained by rendering the $\langle y \rangle$ variances instead of the cell shading values. The lighter areas of this map indicate the areas of the image that were given higher priority by the progressive renderer. For a smooth surface, such as the one shown in this figure, the variance is distributed evenly through the image. The more distant parts of the surface have a slightly higher variance because of the larger size of the cells that are far from the camera. Figure 8 shows an implicit sphere modulated with a cellular noise [Worley 1996]. The variance map for Figure 8 shows bright bands over the surface discontinuities, indicating that cells containing such discontinuities were refined earlier than other cells.

4. CONCLUSIONS AND FUTURE WORK

The rendering method, here presented, offers the possibility of visualising implicit surfaces with progressive refinement. The main features of a surface become visible early in the rendering process, which makes this method ideal as a previewing tool during the editing stages of an implicit surface modeler. In comparison, a meshing method would generate expensive high resolution preview meshes for the more complex surfaces while a ray caster would be slower and without the progressive refinement feature. Our rendering method, however, does not implement anti-aliasing and cannot compete with an anti-aliased ray caster as a production tool. Production quality renderings of some of the surfaces shown in this paper are typically done overnight, a fact which further justifies the need for a previewing tool.

It would have been straightforward to incorporate anti-aliasing into our rendering method by allowing cells to be subdivided down to sub-pixel size and then applying a low-pass filter to reconstruct the pixel samples. There is, however, one issue that prevents the use of our method as a production tool and which makes this implementation effort not worth the while. As explained in Section 2.1, the computation of range estimates with affine arithmetic is always conservative. This conservativeness implies that some cells a small distance away from the surface may be incorrectly flagged as intersecting with it. As a consequence, some portions of the surface may appear dilated after rendering. The surface offset error is in the same order as the size of a pixel. This artifact can be tolerated during previewing but is not acceptable for production quality renderings.

The capabilities of our progressive refinement rendering method can be expanded by including self-shadowing together with reflection and refraction phenomena. Pyramidal rays can be shot from any cell in the direction of the light sources or along the reflected and refracted directions, in a manner similar to that described in Genetti et al. [1998]. These pyramidal rays can then be tested for intersection using reduced affine arithmetic. They can also be subdivided with the same technique used here to subdivide the camera's viewing frustrum to obtain such effects as soft shadows and glossy reflections or refractions.

REFERENCES

Bergman, L., Fuchs, H., Grant, E., and Spach, S. 1986. Image rendering by adaptive refinement. D. C. Evans and R. J. Athay, Eds. *Computer Graphics (SIGGRAPH '86 Proceedings) 20,* 4 (Aug.), 29–37.

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

Cohen, M. F., Chen, S. E., Wallace, J. R., and Greenberg, D. P. 1988. A progressive refinement approach to fast radiosity image generation. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, J. Dill, Ed. Vol. 22. 75–84.

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.

Cook, R. L. 1989. Stochastic sampling and distributed ray tracing. In *An Introduction to Ray Tracing*, A. S. Glassner, Ed. Academic Press, Chapter 5, 161–199.

de Figueiredo, L. H. 1996. Surface intersection using affine arithmetic. In *Graphics Interface '96*. 168–175.

de Figueiredo, L. H. and Stolfi, J. 1996. Adaptive enumeration of implicit surfaces with affine arithmetic. *Computer Graphics Forum 15,* 5, 287–296. ISSN 0167-7055.

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.

Farrugia, J. P. and Peroche, B. 2004. A progressive rendering algorithm using an adaptive perceptually based image metric. *Computer Graphics Forum (Eurographics 2004 Proceedings) 23,* 3 (Sept.).

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.

Gamito, M. N. and Maddock, S. C. 2005. Ray casting implicit procedural noises with reduced affine arithmetic. Memorandum CS – 05 – 04, Dept. of Comp. Science, The University of Sheffield. April. submitted for publication.

Genetti, J., Gordon, D., and Williams, G. 1998. Adaptive supersampling in object space using pyramidal rays. *Computer Graphics Forum 16,* 1, 29–54. ISSN 1067-7055.

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.

Kolb, C. E. 1992. Rayshade user's guide and reference manual. Draft 0.4.

Laur, D. and Hanrahan, P. 1991. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, T. W. Sederberg, Ed. Vol. 25. 285–288.

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

Lippert, L. and Gross, M. H. 1995. Fast wavelet based volume rendering by accumulation of transparent texture maps. *Computer Graphics Forum 14,* 3 (Aug.), 431–444. ISSN 1067-7055.

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. 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.

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.

Worley, S. P. 1996. A cellular texture basis function. In *Computer Graphics SIGGRAPH 96 Conference Proceedings.* Vol. 30. ACM SIGGRAPH, Addison Wesley, 291–294. held in New Orleans, Louisiana, 04-09 August 1996.

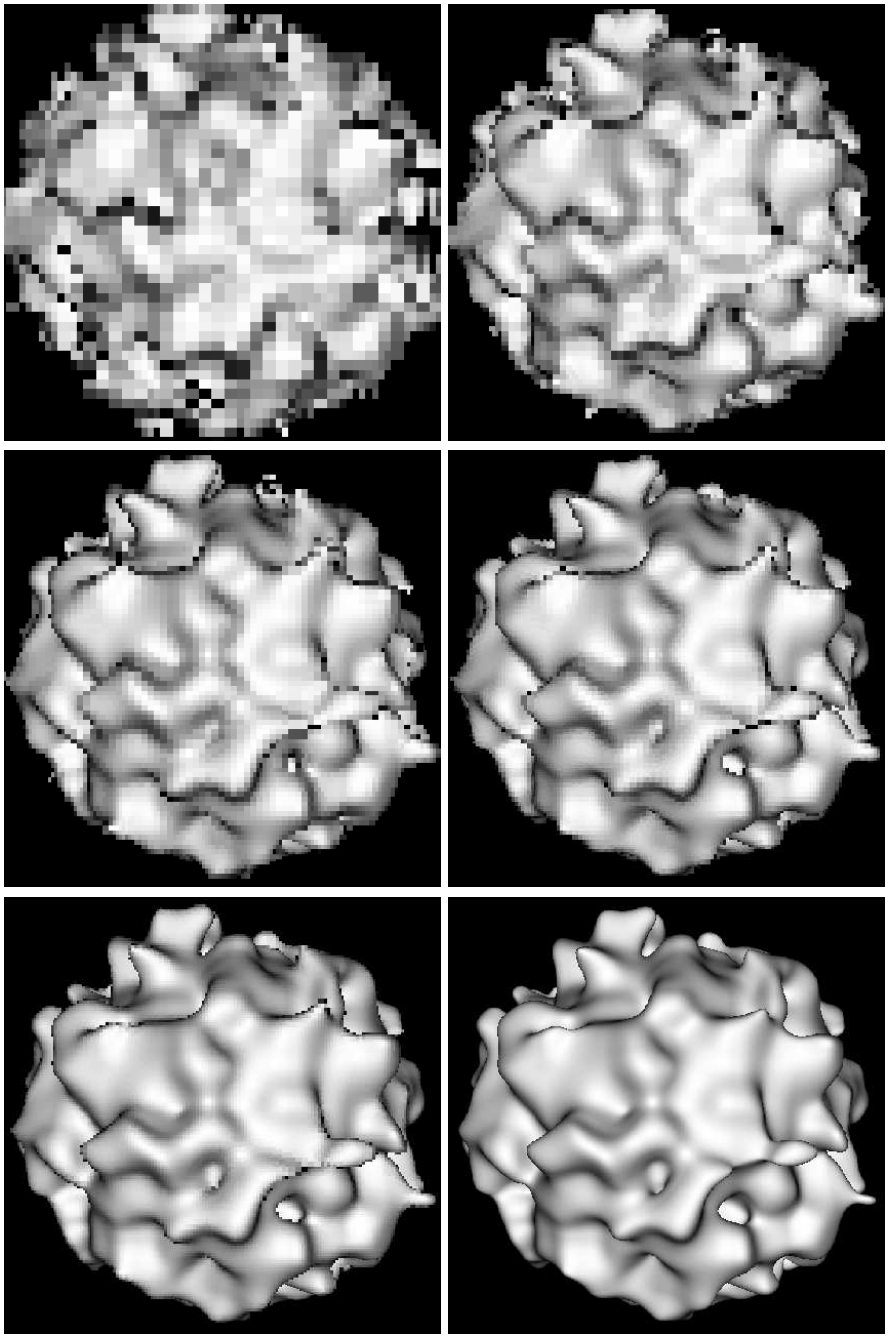

Fig. 4. From left to right, top to bottom, snapshots taken during the progressive refinement rendering of a procedural noise function. The snapshots were taken after 5000, 10000, 15000, 28000, 66000, and 350759 iterations, respectively. The wall clock times at each snapshot are 1.02s, 1.98s, 2.65s, 4.18s, 7.47s and 27.80s, respectively.

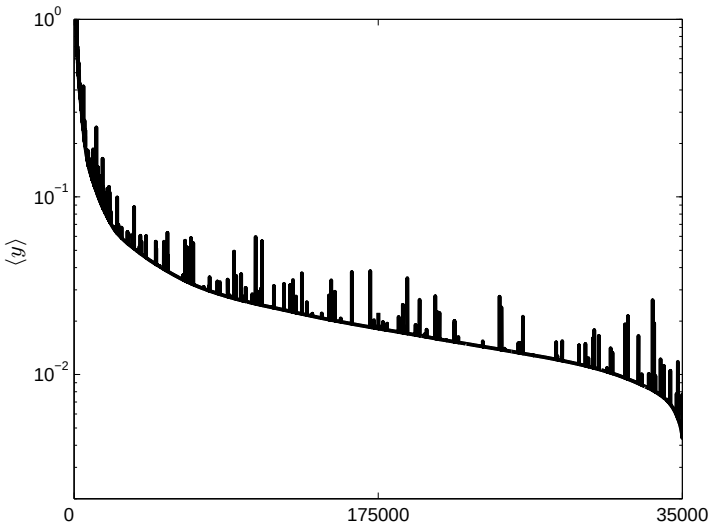

Fig. 5. The decrease in the maximum variance $\langle y \rangle = \langle f(\hat{\mathbf{x}}) \rangle$ for all cells as the algorithm progresses. The vertical scale shows $\langle y \rangle$ for the cell at the top of the priority queue. The horizontal scale is the number of iterations.

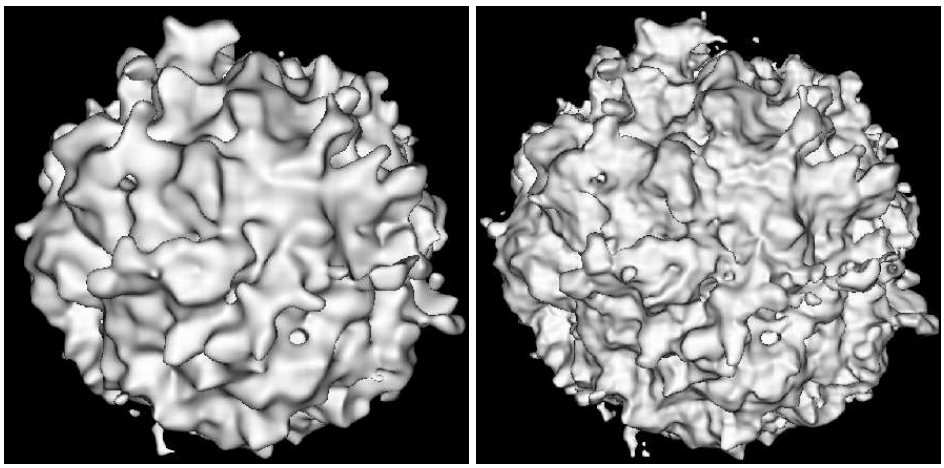

Fig. 6. An implicit surface with two layers (left) and three layers (right) of a Perlin noise function.

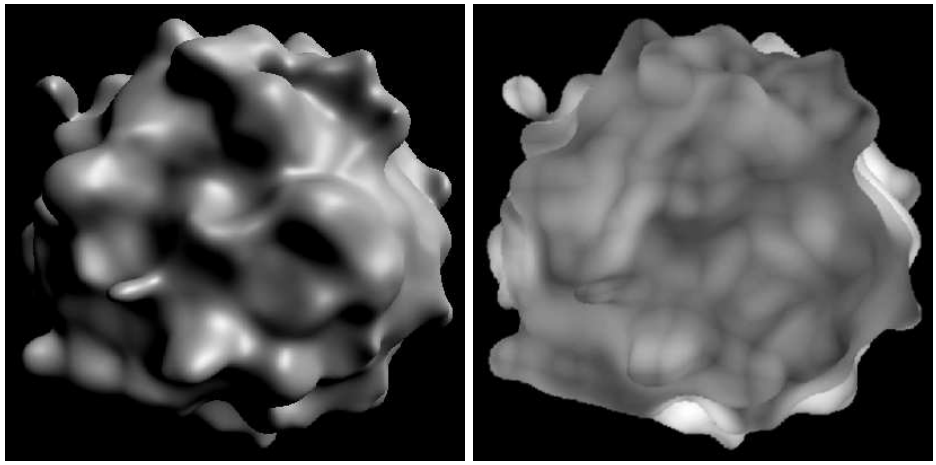

Fig. 7. An implicit sparse convolution procedural surface (left) and its variance map (right).

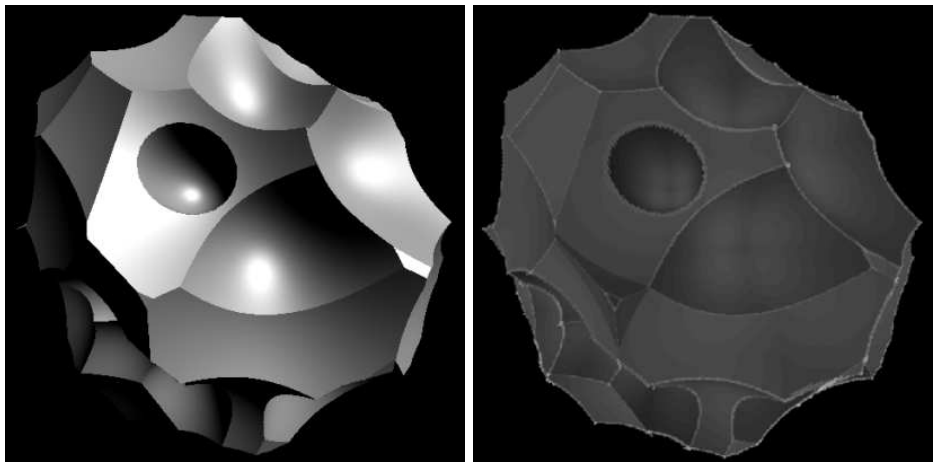

Fig. 8. An implicit cellular noise procedural surface (left) and its variance map (right).