Graphical Models **63,** 387–412 (2001)
doi:10.1006/gmod.2001.0559

Novel Techniques for Robust Voxelization and Visualization of Implicit Surfaces

Nilo Stolte

School of Computer Engineering, Nanyang Technological University, Blk N4, Nanyang Avenue, Singapore 639798

and

Arie Kaufman

Center for Visual Computing/Computer Science Department, State University of New York at Stony Brook, Stony Brook, New York 11794-4400

Received March 22, 2001; revised September 28, 2001; accepted November 14, 2001

Voxelization is the transformation of geometric surfaces into voxels. Up to date this process has been done essentially using incremental algorithms. Incremental algorithms have the reputation of being efficient but they lack an important property: robustness. The voxelized representation should envelop its continuous model. However, without robust methods this cannot be guaranteed. This article describes novel techniques of robust voxelization and visualization of implicit surfaces. First of all our recursive subdivision voxelization algorithm is reviewed. This algorithm was initially inspired by Duff's image space subdivision method. Then, we explain the algorithm to voxelize implicit surfaces defined in spherical or cylindrical coordinates. Next, we show a new technique to produce infinite replications of implicit objects and their voxelization method. Afterward, we comment on the parallelization of our voxelization procedure. Finally we present our voxel visualization algorithm based on point display. Our voxelization algorithms can be used with any data structure, thanks to the fact that a voxel is only stored once the last subdivision level is reached. We emphasize the use of the octree, though, because it is a convenient way to store the discrete model hierarchically. In a hierarchy the discrete model refinement is simple and possible from any previous voxelized scene thanks to the fact that the voxelization algorithms are robust.

Key Words: voxel; voxelization; 3D visualization; interval arithmetic; implicit surfaces; parallel processing.

1524-0703/01 $35.00

1. INTRODUCTION

In discrete geometry [7, 20, 22] a 3D continuous volume is represented by a 3D grid of voxels. This representation is convenient for a number of applications such as mixing synthetic objects into medical imagery (MRI, CT, etc.). Integer arithmetic is often sufficient to treat most of the problems using this representation. This has several advantages: better precision, circuit simplicity, and speed. Nevertheless, many apparently simple problems in this domain can be difficult to solve and have been the subject of many important research works. One of these problems is the conversion of a synthetic object into the voxel format, often called voxelization.

Voxelization is widely used in accelerating ray-tracing and radiosity. In this domain, voxelization algorithms that can guarantee that the voxels completely envelop the surface i.e., that no part of the surface will ever be missed), such as the methods proposed in this paper, can be considerably beneficial.

However, voxelization is now often thought of as only a way to convert synthetic objects to be visualized using volume rendering techniques. The initial goal of volume rendering was to visualize data originally obtained in the voxel format that is difficult to convert to surfaces to be visualized by standard methods, such as in medical imagery. Another obvious application of volume rendering is to simulate and visualize phenomena or objects that are not solid such as gases, clouds, smoke, fog, etc. Both these applications have something in common: the absence of normal vectors. In the first case the normal vectors must be approximated in order to be able to render the objects in a more natural way. In the second case the normal vectors are not necessary because the phenomena being simulated are naturally not related to surfaces. Surfaces, except occasionally implicit surfaces and not considering scan conversion as a voxelization procedure, do not necessarily need to be voxelized in order to be visualized unless a specific engine that handles only voxels but no surfaces is used. In general, though, volume rendering systems can easily allow mixing volumes with surfaces. In the case of ray-casting, voxelization can be used to accelerate the intersection between rays and surfaces. In this case the original surface is still useful to calculate the normal vectors at the intersection points as normally done in ray-tracing. In volume rendering in general, though, the original surfaces are discarded once they were voxelized. In this particular case, then, lack of realism occurs because the surface description, the original source of the normal vectors, is missing. Thus, new methods as in [31], or other more modern related techniques such as filtration or distance field voxelisation techniques [24], try to compensate for the absence of normal vectors in order that the object can be properly rendered by certain volume rendering engines. This compensation technique should not be confused with the voxelization algorithm itself since it is specific to a particular rendering engine. Indeed, volume rendering does not emphasize robustness since this rendering technique and its specifically designed voxelization methods are intrinsically approximate. Also of mention, since in these cases the surface information is discarded once the voxelized scene is generated, there is no way to refine the voxelization from a given state. In principle, then, the whole scene must be voxelized again if one wishes a more refined view of a particular piece of an object because there is no way to identify to which surface a voxel belongs. Moreover, resolutions in volume rendering are generally fixed. One solution to this problem is maintaining one individual volume for each object in the scene. But, again, to refine a piece of an object one must voxelize the whole object again.

On the other hand, polygons are a popular way to approximate objects in current graphics engines. Although implicit surfaces are not naturally convertible to polygons, nonmanifold implicit surfaces polygonization methods exist [6]. A recent method proposed in [2] also allows converting manifold and nonmanifold implicit surfaces to polygons. It is a variation of the marching cubes algorithm [15] but using an octree to accelerate the conversion. However, even with this acceleration, the conversion to polygons still remains a time consuming task. Manifold and nonmanifold implicit surfaces, though, can be easily converted to voxels (i.e., using the algorithms presented in this article). However, it is difficult for manifold implicit surfaces to be consistently voxelized without subdividing the space. We presented three existing methods that can subdivide manifold implicit surfaces elegantly by recursively subdividing the space [27]. We used two of these subdivision methods to voxelize manifold implicit surfaces. We showed that the voxelization method using interval arithmetic is the most efficient [27].

The algorithms presented in this article are extentions to this interval arithmetic based voxelization method presented in [27] that was initially inspired of a method proposed by Duff [8]. Our method tries to emphasize robustness, discrete model refinement, and rendering algorithm independency, even though the surfaces normal vectors are stored with the voxelized scene to use our alternative rendering technique (see Section 6). However, the normal vector is considered part of the voxel content. Thus, it is transparent as far as the voxelization process is concerned.

A high-resolution voxel space is a promising solution for rendering curved surfaces and other complex objects [14, 25, 27]. This is particularly true in the case of manifold implicit surfaces that are not easily approximated by polygons but that can easily be voxelized. Voxels can be approximated by a point when they are sufficiently small and seen from a reasonable distance, thus being displayed at most by one pixel. The simplicity and the quality gained are the main advantages of this concept. Images rendered this way with standard hardwired Z-buffer, such as those shown in this article, have a ray-casting quality. This visualization algorithm is described in Section 6.

1.1. Incremental Voxelization Algorithms

Voxelization of lines and planes is relatively simple using scan-line incremental techniques [12]. Bezier splines surfaces' voxelization have been also accomplished using incremental techniques [13]. Unfortunately, incremental methods do not guarantee correctness. Since precision is one of the main motivations of discrete geometry, voxelization based on incremental techniques does not seem to be an adequate solution for the voxelization problem.

Incremental algorithms have the reputation of being efficient because additions are generally much faster than the multiplication and division. In addition, it is a general belief that integer operations are more efficient than floating point operations. In principle this is true since floating point operations take 4 steps to be executed (alignment, execution, renormalization, and rounding) while integer operations take only one step (execution). However, this scenario has been significantly changing in modern machines, since computer chip manufacturers have been supplying processors with special hardware acceleration for floating point arithmetic. Parallel techniques, such as Pipelines and redundant numeration systems (i.e., "carry-save"), and other hardware improvements are allowing floating-point operations to be closer to the integer operation performance as never before. This current reality

in the industry and its future trends force us to reconsider the axiom which has been the inspiration of integer based techniques.

Therefore, the only real advantage of using integer based methods would be of circuit simplicity, which might be important to special-purpose machines, but not for general-purpose machines. Special-purpose machines are not likely to be as popular as general-purpose machines. Hence, general-purpose machines are more likely to be supported by the industry in the near future. Thus, although incremental techniques are appealing for performance reasons they are not robust. Moreover, integer operations, the main inspiration for incremental methods and for a long time thought to be much more efficient than floating point operations, are really almost as efficient as floating point operations in the latest processors. This tendency has even worsened now with the arrival of graphics processing units (GPUs for short), new graphics chips embedding floating point processors.

However, the problem of representing continuous models inside the machine still remains. Discrete representation is still ideal because of its accuracy and simplicity. Consequently, discrete models seem convenient to describe continuous models in the machine. Nevertheless, all the well-founded axioms and theorems valid for real numbers are lost in the discrete model. Also, as previously stated, discrete models lack an important piece of information to be properly rendered: the normal vectors. This indicates that the continuous model is still important and that it should be maintained. Conversely, discrete models can materialize analytical surfaces elegantly. In fact, discrete models could be thought of as envelopes to the continuous surfaces, which could be refined at will using the continuous model. In this way, to refine from a previously voxelized model, the calculation accuracy should have been preserved; otherwise, the representation cannot be correctly refined. Thus, the need for robust voxelization methods becomes evident. As we shall see, robust methods also provide an efficient way to voxelize nonmanifold as well as manifold implicit surfaces.

1.2. Robust Implicit Surfaces Voxelization Using Recursive Subdivision

This article shows novel methods to voxelize surfaces guaranteeing robustness, that is, that no part of the continuous surface will ever be missed by the discrete model. Interval arithmetic [16, 17] is the basic tool used here for this purpose. Interval arithmetic was simultaneously but independently introduced to computer graphics by Snyder and Duff [8, 23]. Although many powerful properties have been presented, interval arithmetic still has not been considered in many computer graphics problems. Intervals are particularly useful in determining discrete models because any three-dimensional box in the space can be formally represented by 3 intervals, each one corresponding to one of the coordinates of the space. In addition, if the continuous model is defined by an implicit surface, interval arithmetic can be used to determine if the three-dimensional box can contain, or not, a part of the continuous model. This calculation is done much more efficiently than traditionally, where intersection calculations would be required. In addition, the result is guaranteed to be correct. However, interval arithmetic is conservative, which means that large three-dimensional boxes also induce large overestimations. Recursive space subdivision is an elegant way to overcome this problem, since at each step the volumes shrink to one-eighth of the original volume. Therefore, the voxelization is easily accomplished by applying this recursive subdivision until the desired grid resolution is reached. Nonetheless, this voxelization procedure does not require an uniform resolution. In addition, at any level where the subdivision has stopped, it can be automatically continued from that point at a later time.

There are other algorithms for voxelizing implicit functions by subdividing the space recursively. One example is the methods that use Lipschitz conditions [4, 11]. Taubin [30] also proposed a similar method, but restricted it to polynomial implicit surfaces.

To avoid the high memory consumption and to make discrete model refinement easier, we store the voxels into an octree [26, 27]. This allows us to achieve high resolutions with low memory consumption, since the majority of the scenes are almost empty. Octrees have also been used for storage purpose in [11], but as an auxiliary data structure for accelerating ray-tracing. Indeed, Lipschitz conditions were used for generating voxels and compared with the interval arithmetic approach in [27]. The result of this comparison is in favor of interval arithmetic as far as performance is concerned. This was the main reason interval arithmetic was adopted in our method. Interval arithmetic used in this way guarantees that we always envelop the surface provided that the proper rounding modes are observed [10]. But in practice, since interval arithmetic is conservative, one may relax switching rounding modes as this is generally a time consuming operation in present processors.

In contrast, a naive approach would simply evaluate the function on eight corners of every voxel. If there is no sign variation in the eight calculated values, the voxel is considered empty; otherwise it is considered full. This method can miss voxels having no corner intersection yet containing the surface or part of it. Also, its running time increases by a factor of eight, every time three-dimensional resolution is doubled in each axis. Unfortunately these two features are contradictory: the effect of missing parts of the surfaces is reduced in high resolution grids, but high resolution voxelizations using this method are computationally expensive.

The voxelization methods shown in this article display running times and voxel occupancy showing an increasing factor of 4 every time resolution is doubled in every coordinate axis (see Section 3.3). This provides a considerable advantage over naive algorithms. Quite high resolution voxelizations (in our case always including normal calculations in each voxel) can be obtained in reasonable time.

Thus, in our method, the surfaces are voxelized at a high discrete resolution, where each voxel has its color and a normal vector calculated through the gradient of the function in the middle of the voxel. The normal vector is sensitive to singular points but not the voxelization.

Once the surface is voxelized we need neither the surface equation nor the conversion of voxels into polygons to visualize it. We have used two different rendering algorithms: an efficient high resolution discrete ray-tracing [26] and a hardwired Z-buffer. The algorithm for visualizing the voxelized scene using a hardwired Z-buffer is given in Section 6.

2. OUR PREVIOUS IMPLICIT SURFACES VOXELIZATION METHOD

The voxelization is done by subdividing the space recursively in an octree fashion as in [11]. Each subdivided octant is represented in our case by three intervals, one for each variable (x, y, z), where the lower and higher bounds correspond to the octant bounding coordinates.

Duff and Snyder [8, 23] have simultaneously yet independently introduced interval arithmetic to solve computer graphics problems. Duff concentrated on ray-tracing algebraic implicit functions and Snyder on more general problems such as silhouette edge detection, surface polygonization, minimum distance determination, etc. The algorithm presented in

this section was initially inspired by the image space recursive subdivision method used by Duff [8, 27]. Our basic algorithm, already introduced in [27], uses the same strategy for subdividing the object space to produce the voxelization. For this reason we sometimes refer to it as *Duff's subdivision method* [27].

Interval arithmetic guarantees that the exact result of any arithmetic operation is between two values, called interval bounds. Any real number is represented by two interval bounds. For example, the coordinates, **X**, **Y**, and **Z** are represented in interval arithmetic as

$$\mathbf{X} = [\mathsf{x}, \mathsf{X}]$$
$$\mathbf{Y} = [\mathsf{y}, \mathsf{Y}]$$
$$\mathbf{Z} = [\mathsf{z}, \mathsf{Z}].$$

These interval bounds in our case are the coordinates of the octants' boundaries. Substituting in the implicit function equation each regular coordinate by the correspondent interval and each regular operation by the respective interval operation produces an interval version of the function, which Snyder [23] calls an *inclusion function*. We can verify if the surface does not pass through the octant by simply testing if the resulting interval does not *include* zero, that is, when the inclusion function resulting interval does not *include* a solution for the regular function $\mathcal{F}(x, y, z) = 0$. Then, if the resulting interval does not include zero, the function certainly does not have a zero in the octant; therefore, the surface does not pass through the octant.

The recursive subdivision method, which defines our voxelization, applies the algorithm in Fig. 1 in a recursive fashion.

Since we subdivide object space and not the viewing volume, the subdivision is independent of view point and does not need to be repeated each time the observer changes position or orientation. The method is also independent of the rendering algorithm.

The interval operators able to implement inclusion functions can be mainly found in [8, 23, 27]. Implicit surfaces specific operators like blending and others are not issued in this article but can be found in [4, 5, 9, 11, 19, 27, 32]. The use of most of these operators with our voxelization methods may be accomplished without any major difficulty.

Although interval arithmetic has a reputation of being too conservative when equations become more complex, its performance is better compared to alternative robust methods such as Lipschitz constants [27]. In addition, several known techniques [18] can be used to

If zero is contained into the interval calculated by applying the inclusion function to the octant

Then subdivide octant

Else reject octant

FIG. 1. Voxelization using the inclusion function.

avoid this interval arithmetic drawback. The simplest technique is to subdivide one or two more levels to see if a particular voxel really contains a piece of the surface. But applying this method for every voxel increases the voxelization time by 4 and 16 times, respectively. However, a smarter heuristic can be applied to know if the super sampling technique is necessary or not. See the discussion about this heuristic in Section 3.3.

Albeit using tighter interval arithmetic operations might increase the performance, our priority is not to research more performing interval arithmetic operations but to research ways to optimize the overall algorithm. One might also want to make use of CSG where simple equations are most likely to be employed since the main CSG goal is to represent complex shapes using simple primitives. Furthermore this is exactly the reasoning behind *blobby models* [4, 5, 11, 19, 27] where generally only spheres and super-ellipsoids are used. In our experience, interval arithmetic behaves very well with *blobby models* even if thousands of primitives and quite complex blending functions with degrees up to 128 have been used. However, the need for tighter operators in interval arithmetic is eminent and this has been a research area on its own.

The next sections present some new techniques in voxelizing implicitly defined objects using slightly modified versions of the same basic algorithm presented above.

3. NOVEL VOXELIZATION METHOD FOR IMPLICIT SURFACES IN SPHERICAL AND CYLINDRICAL COORDINATES

Because of axial symmetries, surfaces of CAD objects (and others) are sometimes defined in spherical or cylindrical coordinates. In this section, a method for converting such a representation to a rectangular-voxel representation of the surface is presented.

Given the spherical coordinates, r, θ, and ϕ (or the cylindrical coordinates, r and θ), the focus of attention here is the surfaces defined implicitly in the form $\mathcal{F}(r, \theta, \phi) = 0$ (see Figs. 9, 14, and 13) or $\mathcal{F}(r, \theta) = 0$ (or similar more complex formulations involving cartesian coordinates as well) in the cylindrical case. For simplicity, however, the examples shown in this text are in the form $\mathcal{F}(\theta, \phi) - r = 0$ or $\mathcal{F}(\theta) - r = 0$, in the cylindrical case.

One direct application of these techniques involves blending of spherical or cylindrical surfaces with rectangular implicit surfaces in the same voxel space, thus profiting from easy blending with other implicit forms and convenient modeling using spherical/cylindrical coordinates.

The method uses a spatial recursive subdivision and tests the potential spatial occupation of the surface in each subdivided rectangular region by using interval arithmetic as seen in Section 2.

This section presents our method for voxelizing spherical/cylindrical implicit surfaces simultaneously offering robustness and low complexity. Robustness emerges with the use of interval arithmetic, while the low complexity results from recursive subdivision of the space associated with interval arithmetic (see Section 2).

The voxelization is done by subdividing the rectangular space in a recursive way in eight equal-sized cubes at each interaction. Each of these cubes represents three intervals in interval arithmetic (see Section 2), which are converted to spherical intervals (see Sections 3.1 and 3.2) and then applied to the implicit spherical inclusion function for a containment test.

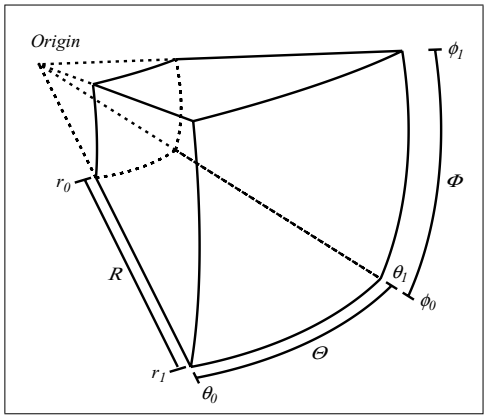

FIG. 2. Intervals $\mathbf{R}[r_0, r_1]$, $\mathbf{\Theta}[\theta_0, \theta_1]$, and $\mathbf{\Phi}[\phi_0, \phi_1]$.

3.1. Spherical Intervals

In the same way as with rectangular intervals, spherical intervals are interval versions of spherical coordinates. Thus, three intervals are defined, one for each spherical coordinate: r, θ, and ϕ. In spherical coordinates, r is the distance between a certain point and the surface origin. Also, by definition, θ is the angle between the projection of the radius on the XZ plane and the X axis, and ϕ is the angle between the radius and XZ plane.

The three spherical intervals are defined as

$$\mathbf{R} = [r_0, r_1]$$
$$\mathbf{\Theta} = [\theta_0, \theta_1]$$
$$\mathbf{\Phi} = [\phi_0, \phi_1].$$

The region defined by these intervals is not cubic, but has the form shown in Fig. 2. These spherical intervals can be inserted in the inclusion function of the spherically described implicit function. If the resulting interval does not include zero, the region defined by the spherical intervals does not contain any part of the surface.

3.2. Converting Rectangular to Spherical Intervals

The rectangular **X**, **Y**, and **Z** intervals have to be converted to spherical $\mathbf{\Theta}$, $\mathbf{\Phi}$, and **R** intervals. Although the conversion of rectangular to spherical coordinates is straightforward, the conversion of rectangular to spherical intervals is more involved. To guarantee robustness, the region in the space defined by the resulting spherical intervals (as shown in Fig. 2) must completely contain the region defined by the rectangular intervals. On the other hand, the resulting spherical intervals should be as tight as possible. The method presented here takes these requirements into consideration.

3.2.1. *Obtaining* $\mathbf{\Theta}$ *Bounds*

To find out the $\mathbf{\Theta}$ bounds we have defined 9 possible cases (see Fig. 3). These cases cover the whole angular domain ($[0, 2\pi]$). In each of these different cases there is a different

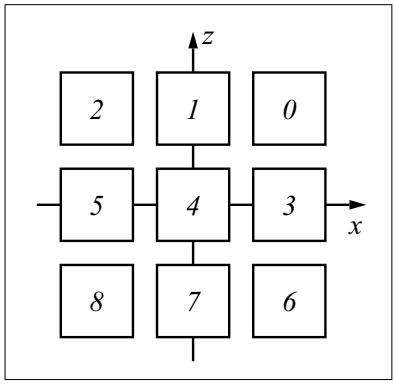

FIG. 3. Nine cases for determining $\boldsymbol{\Theta}$ bounds.

solution for finding $\boldsymbol{\Theta}$ bounds. Each square in Fig. 3 indicates a square defined by the intervals **X** and **Z**, that is, a projection on the XZ plane of the 3D rectangular region defined by **X**, **Y**, and **Z**. The numbers in the squares identify the different cases.

Due to space limitations, only one case, case 0, will be discussed. Nevertheless, case 4 is an exception and has no solution, since it would result in an $[0, 2\pi]$ interval, that is, the whole domain. This problem would not exist if the origin of the surface lies exactly at a vertex of a voxel. However, the origin of the surface cannot be imposed by the arrangement of the voxel grid. In this case, if the subdivision is not at the last level (the voxel level), our choice is to assume that the cube contains part of the surface, even if this is not true. Cases different than 4 will eventually show up for subdivided cubes inside this cube, so the calculation would be again possible. At the last level (the voxel level) if this case persists to show up it is ignored; that is, it is assumed that there is no surface in it. This does not constitute a problem because the only voxels of the surface that will be missing are those that intersect the Y axis. Hence, these voxels could then be calculated in another way.

Figure 4 shows how to obtain angles θ_0 and θ_1 ($\boldsymbol{\Theta}$ bounds) in case 0. In this case,

$$\theta_0 = \text{atan}(z_0/x_1) \quad (-\infty \text{ rounding})^{\dagger}$$
$$\theta_1 = \text{atan}(z_1/x_0) \quad (+\infty \text{ rounding})^{\dagger},$$

where the † denotes suggested rounding modes to guarantee numerical robustness.

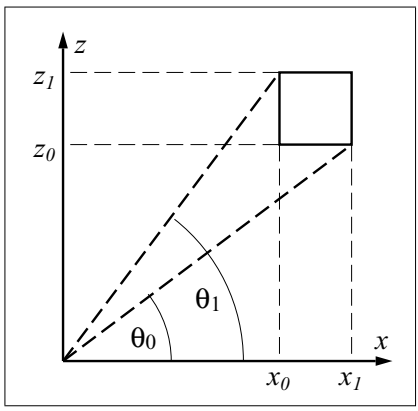

FIG. 4. Determining $\boldsymbol{\Theta}$ bounds for case 0.

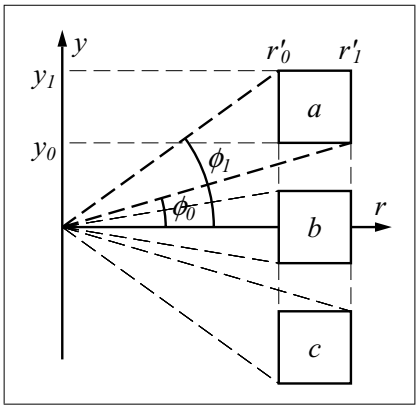

FIG. 5. Three cases for determining **Φ** bounds.

In the other cases, θ_0 and θ_1 are calculated in a similar fashion. Angles are always defined in such a way that cubes are totally enclosed by them as indicated in Fig. 4. This is done to define a spherical interval which contains the cubic one.

3.2.2. *Obtaining* **Φ** *Bounds*

These bounds are only applicable to spherical coordinates not cylindrical coordinates. To cover the whole spherical space, ϕ_0 and ϕ_1 need to be defined in only half of the angular domain, that is, $[-\frac{\pi}{2}, \frac{\pi}{2}]$, since θ_0 and θ_1 are already defined in the $[0, 2\pi]$ domain. Since case 4 is eliminated from the analysis, as discussed in the previous section, only three different cases are necessary to fully define **Φ** bounds.

The three cases are indicated in Fig. 5. Axis r in Fig. 5 is a rotating axis over XZ plane. Suppose that the cube being considered is case 0 (Figs. 3 and 4) and case a (Fig. 5), case 0a for short, ϕ_0 and ϕ_1 are calculated in the following way (r'_1 and r'_0 are not bounds for **R**),

$$
\begin{aligned}
r'_0 &= \sqrt{x_0^2 + z_0^2} \quad (-\infty \text{ rounding})^\dagger \\
r'_1 &= \sqrt{x_1^2 + z_1^2} \quad (+\infty \text{ rounding})^\dagger \\
\phi_0 &= \text{atan}(y_0/r'_1) \quad (-\infty \text{ rounding})^\dagger \\
\phi_1 &= \text{atan}(y_1/r'_0) \quad (+\infty \text{ rounding})^\dagger,
\end{aligned}
$$

where the † denotes suggested rounding modes to guarantee numerical robustness.

3.2.3. *Obtaining* **R** *Bounds*

Interval **R** lower and upper bounds correspond respectively to the minimal and maximal radius value in the 3D region defined by the three rectangular intervals. These maximal and minimal values are the distances between the surface origin and the points of the 3D region defined by the three rectangular intervals which are respectively the nearest and the farthest to this origin.

The process of obtaining these points is straightforward and only requires visualizing the cube in the three-dimensional space with relationship to the origin.

The most trivial calculation is obtaining the lower bounds in cases $1b$, $3b$, $5b$, and $7b$. They are respectively $|z_0|$, $|x_0|$, $|x_1|$, and $|z_1|$. The lower bounds for cases $1a$, $3a$, $5a$, and $7a$ are respectively $\sqrt{z_0^2 + y_0^2}$, $\sqrt{x_0^2 + y_0^2}$, $\sqrt{x_1^2 + y_0^2}$, and $\sqrt{z_1^2 + y_0^2}$. Similarly, the lower bounds for cases $1c$, $3c$, $5c$, and $7c$ are respectively $\sqrt{z_0^2 + y_1^2}$, $\sqrt{x_0^2 + y_1^2}$, $\sqrt{x_1^2 + y_1^2}$, and $\sqrt{z_1^2 + y_1^2}$.

The lower bounds for cases $0b$, $2b$, $6b$, and $8b$ are equally simple. They are respectively $\sqrt{x_0^2 + z_0^2}$, $\sqrt{x_1^2 + z_0^2}$, $\sqrt{x_0^2 + z_1^2}$, and $\sqrt{x_1^2 + z_1^2}$.

For all the other bounds in these and other cases, the maximal and minimal points are always vertices of the rectangular region defined by the rectangular intervals. In these cases the lower bound is the minimal distance between the origin and one of the vertices; the upper bound is the maximal distance.

The calculation of **R** bounds generally requires a square root per bound. However, the signs of the results of the square roots are ambiguous in certain cases. In spherical coordinates a negative r may have meaning. To simplify the calculation and to save computation time one can sometimes work with the square of r instead. This solution was adopted here when spherical coordinates were used. To use r^2, the function $\sin(n \cdot \theta) \cdot \sin(m \cdot \phi) - r = 0$, for example, can then be evaluated as $[\sin(n \cdot \theta) \cdot \sin(m \cdot \phi)]^2 - r^2 = 0$.

When r^2 cannot be used (when only r shows up in the equation and it cannot be isolated) then $-r$ should also be considered if the sign of the radius is important. In this case, two equations are considered instead of only one. An interval can be rejected only if it is rejected in both equations, that is, if the resulting interval of both inclusion functions does not include zero.

3.3. Results

Table 1 shows some voxelization times (in seconds) and voxel occupancy (in millions of voxels) for the function $\sin(n \cdot \theta) \cdot \sin(m \cdot \phi) - r = 0$ for different values of n and m and different 3D resolutions (see Figs. 9 and 14). All voxelizations were generated on a laptop with a Pentium III 1-GHz processor. Times marked with the letter a indicate that swapping occurred. Voxel occupation (indicated in *Occ.* columns) in Table 1 is given in millions of occupied voxels. After the voxelization, all surfaces were examined in the same laptop using a GeForce2 GO GPU and our interactive voxel visualization algorithm described in Section 6. Normal vector calculation and normalization are included in the voxelization times.

The overall complexity increases as the resolution grows, but running time and number of occupied voxels approach an increasing factor of 4 every time resolution doubles in every

TABLE 1
Voxelization Times for $\sin(n \cdot \theta) \cdot \sin(m \cdot \phi) - R = 0$

	$n = 9, m = 18$		$n = 9, m = 10$		$n = 5, m = 6$		$n = 3, m = 4$	
Res.	Time	Occ.	Time	Occ.	Time	Occ.	Time	Occ.
1024^3	$171''^a$	23.8	$123''^a$	18.	$77''^a$	10.9	$49''$	7.4
512^3	$37''$	5.87	$28''$	4.45	$18''$	2.72	$13''$	1.85
256^3	$8''$	1.42	$7''$	1.08	$4''$	0.67	$3''$	0.46

[a] Swapping occurred.

coordinate axis. For $n = 3$ and $m = 4$, this quadratic behavior is even more pronounced: the number of occupied voxels is exactly 4 times the value of the previous row. The times also display the same proportion. By contrast, a method that examines all voxels would have the time increased by a factor of 2^3. For a 512^3 resolution our proposed algorithm can be roughly evaluated as being 512 times faster.

These results show a clear pattern of behavior. Voxel occupation tends to be multiplied by 4 each time the resolution is multiplied by 2 in each axis. This tendency is stronger the smaller the voxel is. This behavior must come from the fact that smooth surfaces have a tendency to behave as planes when sampled in narrowing volumes, thus occupying at maximum only half of the voxels in an octant. This explanation is confirmed in theory since in the limit when the size of the voxel tends to zero, the voxel tends to a point and the surface tends to the tangent plane at that point. This means that this knowledge can be used to establish an heuristic to control how well interval arithmetic is performing. If the last octant contains more than 4 occupied voxels, this must be seen as suspicious and further subdivision should be accomplished to eliminate the spurious voxels. If the extra subdivision did not confirm the presence of spurious voxels, then they must be assumed to be occupied. Another way of testing is to calculate the implicit function value in each voxel vertex and verify if there is a change in sign. If there is a change the voxel cannot be considered spurious; however, nothing can be said if there are no sign variations. A single occupied voxel or less than 4 occupied voxels in an octant are more likely to be correct but still have a strong possibility to be spurious. Further study with statistical analysis is desirable to draw further conclusions in these cases.

3.4. Cylindrical Coordinates

The image in Fig. 6 shows a gear generated using cylindrical coordinates. Gear teeth were generated by the implicit function $r - (0.8 + 0.05 \cdot \sin(32 \cdot \theta)) = 0$ and voxelized by the method described in this article. All other parts were voxelized using constraints implementing CSG operations of cylinders and planes. The voxelization time for this model at a 512^3 resolution was 4 seconds with a Pentium III 1-GHz processor and the voxelized scene contained 1, 014, 512 occupied voxels.

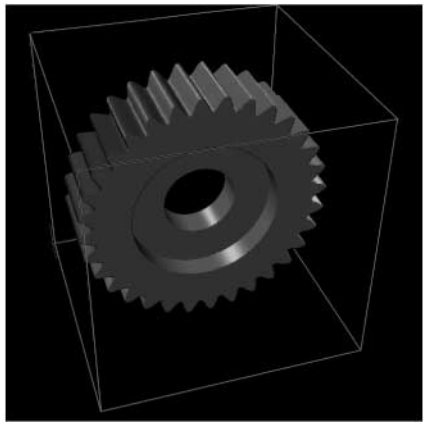

FIG. 6. Gear.

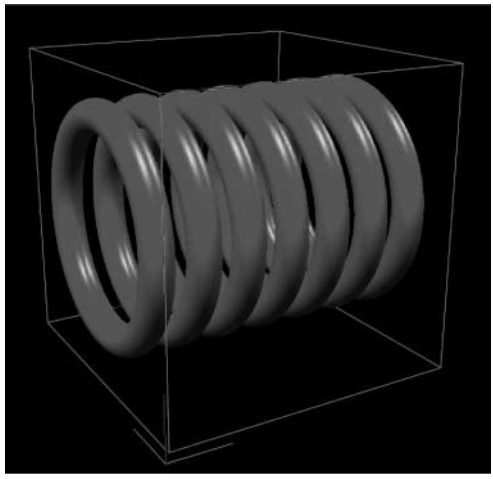

FIG. 7. Example of a torus replication.

Other examples of implicit surfaces in cylindrical coordinates are the replicated tori and the mechanical spring from Section 4 (see Figs. 7 and 8). The tori from Fig. 7 are generated using Eq. (3). They were voxelized using a combination of the technique shown in this section and that of Section 4. The original torus is the one in the middle of Fig. 7 and it is given by Eq. (2).

3.5. Limitations

It is commonplace to express, in spherical coordinates, surfaces on which points with $r < 0$ are actually significant. To be more precise, one has a mapping from $\mathcal{R}^3 \to \mathcal{R}^3$ given by $(r, \theta, \phi) \to (r \cdot \cos\theta \cdot \cos\phi, r \cdot \sin\phi, r \cdot \sin\theta \cdot \cos\phi)$; this mapping can be applied to an arbitrary subset of the domain $\mathcal{R}^3$ to define a subset of the codomain $\mathcal{R}^3$. Unfortunately, the mapping is not invertible, so we cannot simply map points in the codomain $\mathcal{R}^3$ to the corresponding points in the domain $\mathcal{R}^3$. Thus without ad hoc methods, our algorithm is limited to computing isosurfaces that lie within the $(-\infty, \infty) \times [0, 2\pi] \times [-\frac{\pi}{2}, \frac{\pi}{2}]$ interval in the domain space. In certain cases, it is possible to express r in terms of θ and ϕ, and hence get an expression for r^2; in these cases, besides the considerable economy in calculation times, only one version of the equation can be used instead of the two suggested above.

4. NEW TECHNIQUE: INFINITE IMPLICIT REPLICATION

Defining complex scenes with simple expressions is an ultimate goal in modeling and computer graphics. Fractals have fascinated everyone by their beauty and simplicity. Unfortunately, they are normally inefficient to be generated, since they require a large number of recursive iterations to be evaluated. Efficient evaluation is also a high priority in modeling and computer graphics, since the model rendering times are almost always directly connected with the evaluation efficiency.

Defining surfaces procedurally or with functions allows compact representation of the model while giving a precise definition of the continuous surface of the modeled objects. On the other hand, these abstract objects are difficult to be materialized and visualized.

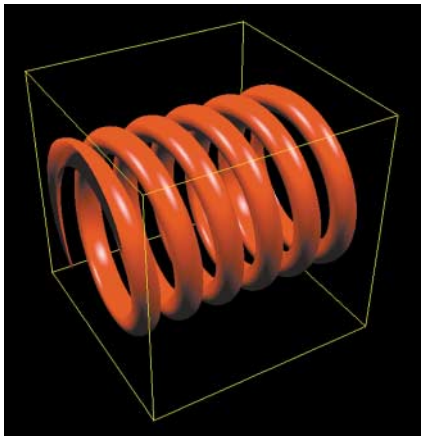

FIG. 8. Replication of two subsets of pieces of a mechanical spring.

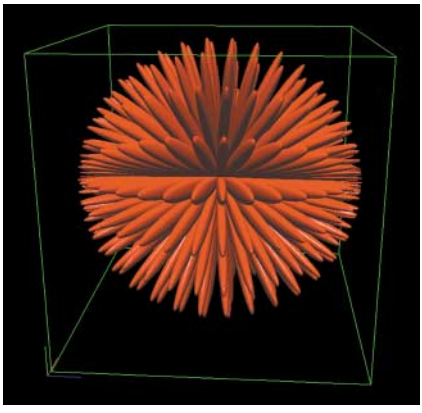

FIG. 9. $\sin(9\theta) \cdot \sin(18\phi) - R = 0$.

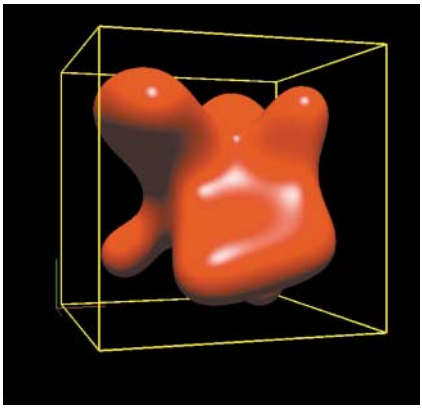

FIG. 12. Ten spheres blended.

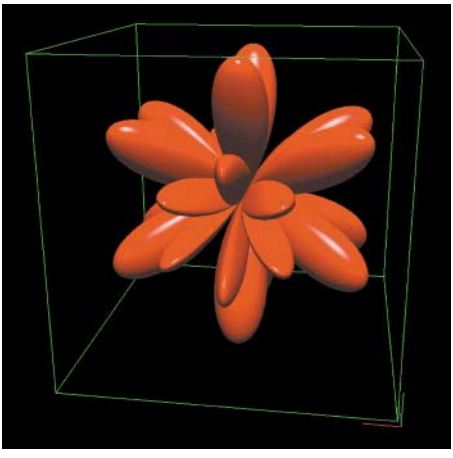

FIG. 14. $\sin(3\theta) \cdot \sin(4\phi) - R = 0$.

Most of the time approximations of these surfaces are used for visualization purposes only. Parametric surfaces are popular in this sense since they can be easily approximated by polygons, and polygons are a popular way to approximate objects in current graphics engines. However, parametric surfaces lack many interesting properties normally found only in implicit surfaces, namely: the capability to know if a point in the space is in, out, or on the surface; the notion of *distance* between a point and a surface by only evaluating the implicit function at the point; and the blending of different surfaces for easy connectivity. Although implicit surfaces are not naturally convertible to polygons, they can be easily converted to voxels using the algorithm presented in Section 2. Thus, implicit surfaces are an attractive way to model objects and voxels are appealing to materialize these models as well as to visualize them. During their conversion to voxels a serious concern is the evaluation time for the surface equation. This function is evaluated for every voxel containing the surface, and for all parent cubes in the recursive subdivision of the space. Particularly when several copies of the same object, or slightly modified versions of the same object, are desired, all their equations are normally evaluated at every voxel or parent cube. A classic way to proceed is to multiply all the surfaces equations among them [29]; thus if any one of these expressions is zero the whole function is also zero. A major problem in this approach is that the exact expressions which are zero in a certain region are unknown, which potentially forces the evaluation of each expression. Moreover infinite replications become impractical. Thus, an automatic way to detect which surface corresponds to each region is highly desirable.

We propose in this section a new high level tool to solve this problem, called *infinite implicit replication*. The technique is based on a function mapping real values to integers, which are considered as a *replication factor*. This method is unique since, to our knowledge, it has not been shown in any known theorem, axiom, or publication. The simplest form of replication is attaching the replication factor to translations. In this way the same surface can be infinitely repeated. Even in this simplest form the technique already has a number of applications. One of these applications is representing some surfaces implicitly where previously they could only be defined parametrically.

4.1. Constructing and Applying the Replication Factor

The replication factor is an integer which is a function of one or more real coordinates, that is, it is a function

$$f(x_1, x_2, \ldots, x_n) : \mathcal{R}^n \to \mathcal{N}.$$

For simplicity, let the replication factor i be a function of the z coordinate, $f(z) : \mathcal{R} \to \mathcal{N}$, that is,

$$i = f(z).$$

Now, let S be a surface to be replicated infinitely along the Z axis separated by a distance $b \in \mathcal{R}$. Then a replication factor can be given by

$$i = (\text{int}) \frac{\left(z \pm \frac{b}{2}\right)}{b}. \tag{1}$$

When z is positive $\frac{b}{2}$ is added to z to shift the whole surface to the integer boundary; otherwise only half of S is replicated. When z is negative $\frac{b}{2}$ must be subtracted to obtain the same effect.

Let S be a torus defined implicitly in cylindrical coordinates by the equation

$$(r - R)^2 + z^2 - a^2 = 0, \tag{2}$$

where a is the small radius and R is the large radius. Applying the replication factor to S would give

$$(r - R)^2 + (z - (i \cdot b))^2 - a^2 = 0. \tag{3}$$

In this example we demonstrate the simplicity and the power of implicit replication. The torus is automatically replicated (translated) by steps given by b up to the infinity, without any extra cost but the computation of i and $z - (i \cdot b)$. These computations are quite negligible in comparison with the obtained effect. Much more complex scenes can be derived by applying i in more involved replication functions. This can be seen as a kind of fractal without the normal intrinsic cost in fractal generation.

4.2. Calculating the Replication Factor in Interval Arithmetic

Interval arithmetic is a key tool for converting implicit surfaces into voxels as seen in Section 2 and in [25, 27]. However, even the simple replication function shown in Eq. (1) has a condition (expressed by $\pm$) which is difficult to control in interval arithmetic. Potentially, the interval can include more than one instance of the object. The simplest way to solve this problem is evaluating all the instances in each interval I. If one of these evaluations produces an interval with different signs in its bounds, I is accepted for further subdivision. Otherwise, I is rejected, since no part of any surface passes through the region delimited by I. Notice that further the subdivision advances less surfaces are potentially evaluated. Most of the evaluations will include only one surface, thus saving an important amount of computing time.

Given an interval $[z_0, z_1]$ to be evaluated in the inclusion function (the function in interval arithmetic [23]) of the torus given in Eq. (3), we first calculate two replication factors, one for each interval bound:

$$i_0 = (\text{int})\frac{\left(z_0 \pm \frac{b}{2}\right)}{b}$$

$$i_1 = (\text{int})\frac{\left(z_1 \pm \frac{b}{2}\right)}{b}.$$

The number of surfaces to be potentially evaluated in this interval is $i_1 - i_0 + 1$. Therefore, a simple loop starting in i_0, ending in i_1 with a unit increment and containing the evaluation of the inclusion function of Eq. (3), will be enough to handle all the surface replications inside an interval $[z_0, z_1]$. Notice that the first part of Eq. (3) can be calculated outside the loop, since it does not depend on z coordinates.

Figure 7 shows a voxelized model containing the replicated torus given by Eq. (3) with $a = 0.1$, $b = 0.3$, and $R = 0.7$. The voxelization with a Pentium III 1-GHz processor at a resolution of 512^3 was accomplished in 4 seconds and the object occupied 1,875,731 voxels.

4.3. Parametrical to Implicit Using Replication

The simple technique presented above can be further developed to be an alternative to represent some parametrical surfaces in the implicit form. Recent work [1] has focussed on how to model a solid object usually represented by parametric methods, i.e., through a B-rep approach. Since parametrically defined objects cannot adequately represent the solid properties associated with them, authors suggest using implicit sweeps to replace the parametric representation by an implicit one. The method given in [1] uses Jacobian rank deficiency conditions to accomplish the sweep of an implicit surface along a curve defined parametrically. This method has a dual advantage of being very elegant and general. Any implicit objects can then be swept through any parametric curve which is a very powerful tool in modelling.

Infinite implicit replication can help to represent certain parametrically defined objects into implicit form at a very low cost. For example, an infinite helicoidal torus (a mechanical spring) can be obtained by expressing one cycle of such an object implicitly and infinitely replicating this cycle. However, joining the different replications of this object implies interpenetration of the cyclical regions which is not previewed in our infinite replication technique (it assumes that the replication region is unique, not shared with any other region). The solution is to divide the object into two subsets and replicate each subset independently. The two implicit functions, one for each subset, in cylindrical coordinates can be combined using Ricci's union operator [21] as shown in Eq. (4). The voxelized object with $a = 0.1$, $b = 0.7$, and $R = 0.7$, is shown in Fig 8. The voxelization at a resolution of 512^3 was accomplished in less than 6 seconds on a Pentium III 1-GHz processor and the object occupied 1,564,250 voxels,

$$\begin{aligned}
\mathcal{A}(r,\theta,z) &= (r-R)^2 + \left[z - b\cdot\left(i + \frac{\theta}{4\pi}\right)\right]^2 - a^2 \\
\mathcal{B}(r,\theta,z) &= (r-R)^2 + \left[z - b\cdot\left(i + \frac{\theta}{4\pi} - \frac{1}{2}\right)\right]^2 - a^2 \\
\mathcal{F}(r,\theta,z) &= \min(\mathcal{A}(r,\theta,z), \mathcal{B}(r,\theta,z)) = 0 \\
&\quad \theta \in [-\pi, \pi].
\end{aligned} \tag{4}$$

In order to obtain an implicit surface using Cartesian coordinates (x, y, z) we are obliged to have a mapping $\mathcal{M}(x, y, z): \mathcal{R}^3 \to (r, \theta, z) \in \mathcal{R}^3$, such that the surface is defined by the points satisfying $\mathcal{F}(r, \theta, z) = 0$. Therefore, the domain of the angle θ in Eq. (4) is restricted to $[-\pi, \pi]$ since θ must be calculated from the Cartesian coordinates (x, y, z) using, for example, the function *atan2* (see Section 3.5). Here this limitation is corrected using the infinite implicit replication. The result is that the mapping $\mathcal{M}$ becomes bijective for all points of $\mathcal{F}(r, \theta, z) = 0$.

Our goal is just to show the power of infinite implicit replication and not to try to demonstrate its generality to solve other problems of this or any other kind. On the other hand, one is tempted to think that this technique could be used to generate sweeps along arbitrary paths as those in [1]. This technique as defined here can only generate discrete instances of objects as shown in this article, but when the object is very thin and the replication distance very small it can generate objects resembling continuous sweeps. However, this resemblance is just in appearance since the object created in this way is not a continuous sweep. Conversely, some sweeps in [1] are used to represent parametrical surfaces in the implicit form. One

example is the mechanical spring reproduced above using infinite implicit replication that can also be generated using sweeps as seen in [1]. The advantage of using our replication technique in this realm is clearly the efficiency as well as the simplicity of the approach in relationship to sweeps.

5. NOVEL PARALLEL VOXELIZATION METHOD USING RECURSIVE SUBDIVISION

The transformation of geometric surfaces into voxels is an important research topic for volume visualization. It allows mixing geometric with volumetric data into the same volume. Implicit surfaces, in particular, are usually transformed into voxels before being transformed to polygons. Voxels are a natural way to represent implicit surfaces in the same way polygons are a natural way to represent parametric surfaces. For parametric surfaces the parametric space can be subdivided recursively to produce polygons, while for implicit surfaces the three-dimensional space can be subdivided to produce voxels. Space recursive subdivision is an elegant way to produce efficient and robust voxelization. Other important advantages we can cite are its simplicity to deal with manifold objects, no need for clipping, low algorithm complexity, and facility to classify regions inside and outside the surface. Octrees are natural data structures to store volumes where the interior is homogeneous or nonexistent, and to avoid representing the voxels outside a surface.

Although octrees are not natural candidates for parallelization, good algorithms exist addressing this subject. One example that particularly fits our problem of surface voxelization is [3]. This algorithm exhibits fairly good results with up to 4 processors. For more than 4 processors the results are not as satisfying. As in [3], we use a shared-memory machine and get approximately the same behavior, but with better results.

Unfortunately, the greatest limitation of the algorithm in [3] is the assumption that the containment of a surface into an octant can be known at any moment. For certain subdivision algorithms [8, 11, 27, 30] this assumption is not correct. With these subdivisions we can determine only if the surface is *not* contained in an octant. When the subdivision reaches the leaf level, there is no guarantee that the voxel really contains a part of the surface. Nevertheless, the probability of the voxel belonging to the surface grows quickly at each further subdivision, and at the last level we assume that this probability is high. The voxelization obtained is guaranteed to always envelop the surface. No voxels of the surface will ever be missed.

These subdivision algorithms require a totally different approach for parallelization. First, the octree must be separated from the subdivision. The subdivision must continue until the last level, and only then can the voxel be stored in the octree. This implies that the octree storage must be efficient. Thanks to our octree traversal algorithm presented in Section 5.1, the time of storing voxels in the octree is negligible in relationship to the rest of the tasks.

The goal of our parallelization is the test determining if the octant does not belong to the surface. In our case we also include the calculation of the normal vector (only on the last level) for every voxel for visualization purposes. We assign these tasks to several slave processes that run in parallel. The master process creates the slave processes when the voxelization is required, controls the work balance, kills the slave processes when the work is done, and displays the voxelized scene. This approach has promising results, as shown in Section 5.3.

5.1. Serial Octree Traversal

Our octree is a classical pointer octree, where the root node is defined by a pointer called "octree," as shown in Fig. 10. This pointer points to an array of pointers with eight elements, each one representing one-eighth of the original volume. A null pointer means that the region is empty, while a nonnull pointer points to another array of eight pointers, further subdividing the region. This process continues until the leaf node is found, where each nonnull pointer points to a voxel.

The efficiency of our octree lies in its simplicity. We keep one integer variable *mask1* with a set bit exactly at the bit position n, where n is the current octree level, which is the total number of octree levels in the beginning (see Fig. 10). We use this bit to filter the coordinate's bits and to control the algorithm as in the octree ray traversal algorithm in [26].

```
char *octree;        /*  pointer to the first free octree byte                  */
char *free_space;    /*  pointer to the first free byte in a block              */
int free_bytes;      /*  number of remaining free bytes in a block              */
int X_ant, Y_ant, Z_ant, mask1, mask2;
init_octree( ) {
/*   Initialize mask1 and mask2 as follows (each square is a bit)               */
/*    n = number of octree levels and nb+1 = number of variable bits            */
```

	nb		n-3	n-2	n-1	n	n+1	n+2		5	4	3	2	1	0
mask1←	0	...	0	0	0	1	0	0	...	0	0	0	0	0	0
mask2←	1	...	1	1	1	0	0	0	...	0	0	0	0	0	0

```
octree←free_space←alloc_block( ); /* allocates one block */
free_bytes←Size_of_Block−Bytes_in_Cell;
free_space←free_space+Bytes_in_Cell;
push(octree);
X_ant←0; Y_ant←0; Z_ant←0; /*variables to find common parent */
}
store_in_octree(X,Y,Z_input)
int X,Y,Z;
any input;
   { char **pcel;
     /* Ascend octree to find a common parent */
     while ( ( (X and mask2) ≠ (X_ant and mask2) ) or
             ( (Y and mask2) ≠ (Y_ant and mask2) ) or
             ( (Z and mask2) ≠ (Z_ant and mask2) ) )
      { pop;
        mask1←mask1<<1; mask2←mask1<<1;
      }
     pop(pcel);
     while (TRUE) /* Descends octree until the voxel*/
      { push(pcel);
        if (Z and mask) pcel←pcel+4;
        if (Y and mask) pcel←pcel+2;
        if (X and mask) pcel←pcel+1;
        if ((mask and 1) = 0)
         { mask1←mask1>>1; mask2←mask1>>1;
           if (*pcel = 0) /* if node does not exist, creates it */
            { if (free_bytes<Bytes_in_Cell)
                { free_space←alloc_block( ); /* allocates one block */
                  free_bytes←Size_of_Block;
                }
              *pcel←free_space; /*creates and descends */
              pcel←free_space;
              free_space←free_space+Bytes_in_Cell;
              free_bytes=free_bytes-Bytes_in_Cell;
            }
           else pcel←*pcel; /* Otherwise descends only*/
         }
        else break; /* Leaf reached. Exit loop */
      }
     X_ant←X; Y_ant←Y; Z_ant←Z;
     *pcel←input;
   }
```

FIG. 10. Octree traversal algorithm.

The algorithm in Fig. 10 is given in a "C-like" pseudo-code. For the sake of clarity the type castings are omitted; each attribution command is given by a ←; the logical commands are written with its names (**and**, **or**) instead of symbolically, and the recursive stack operations are denoted by **push** (to put an element into the stack) and **pop** (to remove an element from the stack—a **pop** without argument only affects the stack pointer).

Once initialized (calling **init_octree**( )) the octree is dynamically created by calling **store_in_octree**( ) for each new produced voxel. This function receives 4 parameters—the three voxel coordinates (X, Y, and Z) and a pointer to the voxel content (*input*). In our case, it is the pointer to the surface normal in the voxel.

A remarkable feature of this algorithm is that it does not require descending all octree levels from the root. It starts from the *cell* where the last voxel was stored. In most cases the current voxel will lie in the same *cell* or in a nearby relative *cell*. If it does not lie in the same *cell*, the algorithm ascends some levels until the common parent is found. This happens in the first part of the algorithm. To find the common parent we use the variable *mask2* as shown into the algorithm. This part is extremely efficient because of its simplicity and since the variables used are always in the cache memory.

The next part of the algorithm descends the octree from the common parent *cell*, creating new *cells* when it does not yet exist (when *pcel = 0). The code is quite straightforward, thus no further details are given here. See [26] for a deeper view of this part. Also see [28] which uses similar techniques, but for a proprietary linear octree.

5.2. Parallel Recursive Subdivision

Our parallel implementation is a simple master-slave configuration. This configuration was implemented into a shared memory SGI Challenge multi-processor system. The master creates the slaves and controls their activities. The master maintains an internal work stack where all octants that are going to be subdivided are stored. Initially, only the first eight octants are stored into this stack. The master creates the slaves and enters into a loop until the work is completed. In this loop, the master scans for all nonidle slave queues in search of their results to store them in the stack or, at the leaf level, in the octree. Initially all slaves are idle; thus only the eight original octants remain in the stack. After that, it distributes the octants from the stack to the idle slaves, if there are any.

Each slave which receives one octant starts to subdivide it and test if the surface is contained in each sub-octant. This test is the most time-consuming task, thus the focus of our parallelization algorithm. If the test is true for a given sub-octant, it is stored in the slave queue. This queue has only eight positions and can be accessed by two different indices: one for the master and one for the slave. When the master scans a slave queue it uses its own index. When this index is smaller than the slave index, it is incremented and the octant from its correspondent position in the slave queue is transferred to the appropriated data structure. If the slave working octree level is a leaf level, the octants are voxels and are not written into the working stack but directly into the octree. In this way the quantity of information passing through the work stack is reduced, thus slightly contributing to a better performance. Once the slave is finished and its entire queue has been transferred away, it becomes idle waiting for a new octant from the master. This process is described by the pseudo-code in Fig. 11.

```
slave( )
 { while (TRUE);
    { wait for a master job;
      get octant(X,Y,Z,my→level);
      index ← my→index ← -1;
      for (each of eight sub-octants)
       { determine XS,YS and ZS for sub-octant;
         if (octant(XS,YS,ZS,my→level) may contain the surface)
          { index ← index+1;
            if (my→level is leaf);
             { put voxel(XS,YS,ZS,normal) in my→queue[index]
             }
            else put octant(XS,YS,ZS) in my→queue[index];
            my→index=index;
          }
       }
    }
 }
master( )
 { init_octree( );
   initialize data structures;
   push first eight octants into work stack;
   make copies of slave( ) to all processors and execute them;
   while (there is still work)
    { for (each non-idle slave)
       { i ← slave→master_index;
         level ← slave→level;
         if (level is leaf);
          { while (i < slave→index)
             { i ← i+1;
               get voxel(X,Y,Z,normal) from slave→queue[i];
               store_in_octree(X,Y,Z,normal);
             }
          }
         else
          { while (i < slave→index)
             { i ← i+1;
               get X,Y,Z from slave→queue[i];
               push octant (X,Y,Z,level+1) in the work stack;
             }
          }
         slave→master_index ← i;
       }
      for (each idle slave)
       { pop octant (X,Y,Z,level) from work stack;
         if pop was successful give octant to slave;
       }
    }
   kill all slaves;
 }
```

FIG. 11. Parallel recursive subdivision algorithm.

5.3. Results

Table 2 summarizes our results for a resolution of 512^3. Scene 1 is shown in Fig. 12; Scene 2, in Fig. 13; and Scene 3, in Fig. 14). We show the times as a function of the number of slaves, and the yield in relationship to the time spent for just one slave. The yield is calculated by dividing the estimated time (t_1/n, where t_1 is the time for just one slave and n is the number of slaves) by the real time (t_n), that is,

$$\text{yield} = \frac{t_1}{n \cdot t_n}.$$

The yield calculated this way gives a clear idea of the slaves' activity. The algorithm was conceived to have a high parallel performance because the work tends to be evenly distributed among the slaves. However, the results showed an unexpected outstanding performance in Scene 3 for 2 to 4 slaves and for 7 slaves. This result seems to be linked to

TABLE 2
Performance Results for a Voxelization Resolution of 512^3

	Scene 1		Scene 2		Scene 3	
Slaves	Time	Yield	Time	Yield	Time	Yield
1	88	—	111	—	71	—
2	45	97%	58	95%	35	100%
3	34	88%	38	97%	23	100%
4	26	84%	31	89%	17	100%
5	21	83%	28	79%	16	88%
6	20	73%	24	77%	14	83%
7	14	89%	21	75%	11	92%

the exceptional behavior this scene had in Section 3.3. It suggests that the load balance performance is probably near to optimal. The inferior performance in other scenes and even in this scene for 5 or 6 slaves might not come from the load balancing scheme.

These results were obtained in a Challenger SGI workstation using multiple 200-MHz R10000 processors. Scene 3 at a resolution of 512^3 using 7 slaves took 11 seconds to be voxelized. The same scene in a Pentium III 1-GHz laptop was voxelized in 13 seconds as seen in Section 3.3.

The results in Table 2 are interesting and show that the algorithm deserves further consideration.

6. VISUALIZATION METHOD

The visualization method used to generate images for this article is based on high-resolution voxel spaces (512^3). Voxels are stored in an octree, thus, allowing quite huge discrete spaces without a high memory consumption. Normal vectors are calculated during the voxelization (times in Table 1 and Table 2 include this calculation) by evaluating the

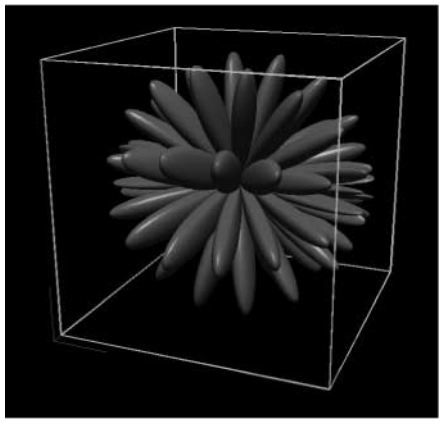

FIG. 13. $\sin(4\theta) \cdot \sin(8\phi) - R = 0$.

```
cell = root = octree root cell address;
i=0;
push(cell);
push(i);
do {
   /* ascend the octree until i<8 */
   while ((i>7) and (cell≠root)) {
      pop(i);                    /* Ascend one*/
      pop(cell);                 /* octree level.*/
      X=X >> 1;  Y=Y >> 1;  Z=Z >> 1;
   }
   while  (i≤7) { /* Descend or move right */
    aux=cell[i];
    X=(X << 1) or (i and 1);          /* Calculate */
    Y=(Y << 1) or ((i >> 1) and 1);   /* the voxel */
    Z=(Z << 1) or ((i >>2) and 1);    /* coord.    */
    if (aux=Leaf Node) {              /* Leaf?     */
       Display voxel (X,Y,Z) as a point with the
       normal vector pointed by "aux";
       i=i+1;                         /* Go right  */
       X=X >> 1;  Y=Y >> 1;  Z=Z >> 1;
    }
    else {                            /* Not Leaf! */
       if (aux≠0) {                   /* Empty?    */
          push(cell);                 /* Descend 1*/
          push(i+1);                  /* level     */
          cell=aux;
          i=0;
       }
       else {                         /* Empty !   */
          i=i+1;                      /* Go right  */
          X=X >> 1;  Y=Y >> 1;  Z=Z >> 1;
       }
    }
   }
} while (cell≠root)
```

FIG. 15. Visualization algorithm.

gradient in the middle of the voxel and then normalizing it. A voxel, located at the leaf octree level, is just a pointer to a structure containing the three normal vector components, color, and other information. Higher octree level nodes contain only octree children pointers, when they exist, or zero otherwise. All the voxels are considered as points and rendered using SGI's GL or OpenGL.

This visualization method can eventually allow close-ups of the surface using levels of details. Levels of details are quite natural to hierarchical voxel models, the models used in this article, because the transition between the original and refined model is indistinguishable.

The algorithm describing the visualization technique is given in Fig. 15. This algorithm was first presented in [25]. Point display techniques have become quite popular in the later years, but none of them was used to render implicit surfaces or other voxelized scenes as in our case.

The variables *cell* and *root* in the algorithm have initially the address of the root of the octree. Variable *i* is an index varying from 0 to 7 used to access the current octree element into an eight-elements cell. These eight elements identify eight equal-sided neighbor cubes, defining a recursive subdivision of a single cube. Each of these elements contains a pointer to a new cell, when this cell contains any part of the surface, or a null pointer otherwise. The recursion is controlled by a stack denoted by the instructions push (to introduce a value in the stack) and a pop operator (to extract a value from the stack). The variable *i* is assigned

a zero value denoting a left to right tree traversal. Both, *cell* and i are pushed in the stack to start the recursive traversal. The recursion is implemented by the do-while loop as shown in the algorithm. The first part inside the loop ascends the tree if i reaches an index greater than 7. Since i is zero in the beginning of the algorithm, the control passes immediately to the second part which descends the tree. This part is a while loop which takes place while $i \leq 7$, indicating that this part also advances to all the elements of the current cell from left to right. The voxel coordinates X, Y, and Z are built, bit by bit, from the i values. Notice that the previous coordinate's bits are saved by shifting them to the left at each new interaction.

If the current cell is a *leaf* node, then X, Y, and Z contain the complete coordinates of the voxel to be displayed and the current element (*cell[i]*) contains a pointer to the normal vector of the voxel. These pieces of information are sent to the graphics card using GL point primitives to display the point with the normal vector. In practice, these pieces of information are first stored in a list and when the list is full all the points are displayed at once to increase efficiency. These details are omitted in the algorithm. Notice that after displaying the voxel, i is incremented to advance to the next element to the right of the current element. Also notice that the coordinate variables must be shifted one bit to the right.

If the cell does not correspond to a *leaf* node, and if the current element (*cell[i]*) is zero, the element does not exist; therefore the algorithm advances to the next element (by incrementing i) and shifts the coordinates one bit to the right. However, if the current element is not zero, the address of *cell* and the next element index ($i + 1$) are saved in the stack, and the algorithm descends the tree by attributing to *cell* the address contained in the current element (*cell[i]*) and making i equal to zero (to restart from the extreme left side again in the new cell).

Once i reaches the value 8, which happens when all the elements of a cell are visited, the control is passed again to the main loop that continues if cell $\neq$ root. This time $i > 7$, and the first *while* loop takes the control. This loop extracts from the stack: (1) the indexes i of the current elements and (2) the cell addresses corresponding to all those cells that were already completely visited. At each interaction this loop also shifts the coordinates one bit to the right. Notice that the loop either stops when a cell not yet completely visited is found (denoted by i values less than or equal to 7) or when the root cell is found. If the root cell is found and i is greater than 7, all cells in the tree have been visited and the algorithm finishes.

At the current time, this method allows interactive visualization for easy surface inspection. The images produced in this article are snapshots from the visualization method viewing window. Image quality is comparable to that of ray-casting.

7. CONCLUSION

We have shown in this article our techniques for robust voxelization of implicit surfaces. The use of spatial recursive subdivision and interval arithmetic is the key for most of the methods presented. The basic voxelization method shown in Section 2 guarantees that no part of the surface is ever missed, defining a new concept in this domain. This concept allows us to voxelize an object starting from a previous voxelization of the same object, instead of revoxelizing the object again. This creates a new paradigm to be possibly used

in interactive walk-throughs using voxels, where objects can be voxelized on the fly when needed. The efficiency of the approach is quite promising as shown in this article.

We have shown original algorithms for parallel implementation of these voxelization algorithms, and robust voxelization of implicit surfaces expressed in spherical and cylindrical coordinates. In addition we show a new concept called *infinite implicit replication*, where an implicit surface can be replicated without extra evaluation cost, also showing how to voxelize them robustly using interval arithmetic.

Finally, we have also presented an innovative technique for rendering the voxelized scene using the hardwired Z-buffer by displaying each voxel as a point. The images in this article, generated using this technique, prove that the method can produce images with quality near to ray-casting, but at a fraction of ray-casting rendering times.

REFERENCES

1. K. Abdel-Malek, J. Yang, and D. Blackmore, On swept volume formulations: Implicit surfaces, *Comput. Aided Design* **33**(1), 2001, 113–121.
2. R. J. Balsys and K. G. Suffern, Visualisation of implicit surfaces, *Comput. Graphics* **25**, 2001, 89–107.
3. M. A. Bauer, S. T. Feeney, and I. Gargantini, Parallel 3D filling with octrees, *J. Parallel Distrib. Comput.* **22**, 1994, 121–128.
4. H. B. Bidasaria, Defining and rendering of textured objects through the use of exponential functions, *Graphical Models Image Process*. **54**(2), 1992, 97–102.
5. J. Blinn, A generalization of algebraic surface drawing, *ACM Trans. Graphics* **1**(3), 1982, 235–256.
6. J. Bloomenthal and K. Ferguson, Polygonization of non-manifold implicit surfaces, *Comput. Graphics* **29**, 1995, 309–316.
7. J. M. Chassery and A. Montanvert, *Géométrie Discrète*, Editions Hermès, France, 1991.
8. T. Duff, Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry, *Comput. Graphics* **26**(2), 1992, 131–138.
9. G. C. Wyvill, B. McPheeters, and B. Wyvill, Data structure for soft objects, *Visual Comput*. **2**, 1986, 227–234.
10. D. Goldberg, What every computer scientist should know about floating-point arithmetic, *ACM Comput. Surveys* **23**(1), 1991, 5–48.
11. D. Kalra and A. Barr, Guaranteed ray intersections wih implicit surfaces, *Comput. Graphics* **23**(3), 1989, 297–306.
12. A. Kaufman, An algorithm for 3D scan-conversion of polygons, in *Eurographics' 87, August 1987*, pp. 197–208. North-Holland, Amsterdam, 1987.
13. A. Kaufman, Efficient algorithms or 3D scan-conversion of parametric curves, surfaces, and volumes, *Comput. Graphics* **21**(4), 1987, 171–179.
14. A. Kaufman, D. Cohen, and R. Yagel, Volume graphics, *IEEE Comput.* **26**(7), 1993, 51–64.
15. W. E. Loresen and H. E. Cline, Marching cubes: A high resolution 3D surface construction algorithm, *Comput. Graphics* **21**(4), 1987, 163–169.
16. R. E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1966.
17. R. E. Moore, *Methods and Application of Interval Analysis*, Soc. for Industr. & Appl. Math. Philadelphia, 1979.
18. S. P. Mudur and P. A. Koparkar, Interval methods for processing geometric objects, in *IEEE-CGA, February 1984*, pp. 7–17.
19. S. Muraki, Volumetric shape description of range data using "Blobby Model," *Comput. Graphics* **25**(4), 1991, 227–235.
20. J. P. Reveillès, *Géométrie Discrète, Calcul en nombres entiers et Algorithmique*, Ph.D. thesis, Université Louis Pasteur de Strasbourg. 1991.
21. A. Ricci, A constructive geometry for computer graphics, *Comput. J.* **16**(2), 1973, 157–160.

22. A. Rosenfeld and R. A. Melter, "Digital Geometry," *Math. Intelligencer* **11**(3), 1989, 69–72.
23. J. M. Snyder, Interval analysis for computer graphics, *Comput. Graphics* **26**(2), 1992, 121–130.
24. M. Sramek and A. Kaufman, Alias-free voxelization of geometric objects, *Trans. Visualization Comput. Graphics* **3**(5), 2000, 236–252.
25. N. Stolte, *High Resolution Discrete Spaces: A New Approach for Modeling and Realistic Rendering (Espaces Discrets de Haute Résolutions: Une Nouvelle Approche pour la Modelisation et le Rendu d'Images Réalistes)*, Ph.D. thesis, Université Paul Sabatier, Toulouse, France, April 1996.
26. N. Stolte and R. Caubet, Discrete ray-tracing of huge voxel spaces, *Comput. Graphics Forum* **14**(3), 1995, 383–394.
27. N. Stolte and R. Caubet, Comparison between different rasterization methods for implicit surfaces, in *Visualization and Modeling* (R. Earnshaw, J. A. Vince, and H. Jones, Eds.), Chap. 10, pp. 191–201, Academic Press, San Diego, 1997.
28. K. Sung, A DDA traversal algorithm for ray tracing, in *Eurographics'91, Amsterdam, June 1991*, pp. 73–85, North-Holland, Amsterdam, 1991.
29. G. Taubin, Distance approximation for rasterizing implicit curves, *ACM Trans. Graphics* **13**(1), 1994, 3–42.
30. G. Taubin, Rasterizing algebraic curves and surfaces, in *IEEE-CGA, March 1994*, pp. 14–23.
31. W. S. Wang and A. Kaufman, Volume-sampled 3D modeling, in *IEEE-CGA, September 1994*, pp. 26–32.
32. B. Wyvill, E. Galin, and A. Guy, Extending the CSG tree: Warping, blending and boolean operations in an implicit surface modeling system, *Comput. Graphics Forum* **18**(2), 1999, 149–158.