Robust Voxelization of Surfaces

Nilo Stolte

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

Abstract

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 very 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 technical report presents several techniques to voxelize different kinds of surfaces guaranteeing robustness.

Keywords: Voxel,Voxelization Algorithms, 3D Visualization, Interval Arithmetic, Implicit Surfaces, Parametric Surfaces, Polygonal meshes, Parallel Processing

1 Introduction

In Discrete Geometry [5, 18, 20, 9] a 3D continuous volume is represented by a 3D grid of voxels. This representation is very convenient for a series 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 very 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 of lines and planes is relatively simple using scanline incremental techniques [13, 23]. Bezier splines surfaces' voxelization have been also accomplished using incremental techniques [14]. 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 very efficient because additions are generally much faster than the multiplication and division. In addition, it is a general belief that integer operations are faster than floating point operations. In principle this is true since floating point operations take 4 steps to be executed (allignment, execution, re-normalization 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 incremental techniques.

Therefore the only real advantage of using incremental techniques 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.

However, the problem of representing continuous models inside the machine still remains. Integer representation is still ideal because of its accuracy and simplicity. Consequently, discrete models seem ideal 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. This indicates that the continuous model is still important. Discrete models can materialize analytical surfaces much more accurately than using polygons. In fact, discrete models could be thought of as envelopes to the continuous surfaces, which could be refined at will. In this way, to generate this representation, the calculation accuracy should be preserved; otherwise, the representation would not be correct.

This technical report shows several ways to voxelize surfaces guaranteeing correctness, that is, that no part of the continuous surface will ever be missed by the discrete model. A basic tool introduced here is Interval Arithmetic. Interval Arithmetic has been simultaneously but independently introduced to Computer Graphics by Snyder and Duff [21, 6]. 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 tridimensional 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 tridimensional box can contain, or not, a part of the continuous model. This calculation is done much more efficiently than traditionally, where intersections calculation would be required. In addition, the result is guaranteed to be correct. However, interval arithmetic is very conservative, which means that large tridimensional 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 eight of the original volume, thus inducing a very fast convergence to the surface. 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 a uniform resolution. In addition, at any level the subdivision has stopped, it can be automatically continued from that point. This constitutes one of the great advantages of this method over previous methods.

Implicit surfaces are not only a very powerful modeling tool but can be seen as a general work tool in the discrete context if associated with interval arithmetics. Voxels can be easily located inside, outside or at the surface using interval arithmetics. In this way our voxelization method has yet another very important advantage: it is not only able to locate the voxels on the surface, but also to detect its interior voxels.

Frey and Borouchaki [8] 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 is not only slow (running time increases by a factor of 8, every time three-dimensional resolution is doubled), but also incorrect since it can miss voxels having no corner intersection yet containing the surface or part of it. This method is suitable neither for high resolution due to high running times, nor good quality voxelization since it does not always envelop the surface.

It is difficult for manifold implicit surfaces to be consistently voxelized without subdividing the space. We have presented three existing methods that can subdivide manifold implicit surfaces very elegantly by recursively subdividing the space [24]. We have generalized two of these subdivision methods to voxelize manifold implicit surfaces. We have shown that the voxelization method using interval arithmetics was the most efficient [24]. However, many open questions have remained including: how to apply this voxelization method to parametric surfaces and how to further accelerate the method. In this technical report we show some solutions to these problems.

A high-resolution voxel space is a very promising solution for displaying curved surfaces and other complex objects [22, 24, 15]. 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 report, have ray-casting quality. In addition, voxels are very widely used in accelerating ray-tracing and radiosity. In this domain, the need of voxelization algorithms that can guarantee an exact envelope of the surface is a *sine qua non* condition. Fine voxelizations in these cases, when associated with proper hierarchies [11, 26], mean faster rendering time since the fine subdivision takes part of the intersection calculation burden into a preprocessing stage.

2 Implicit Surfaces Voxelization Method

The voxelization is done by subdividing the space recursively in an octree fashion as in [12]. 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 [6, 21] have simultaneously yet independently introduced interval arithmetic to solve Computer Graphics problems. Duff concentrated in Ray Tracing algebraic implicit functions and Snyder, in more general problems such as silhouette edge detection, surface polygonization, minimum distance determination, etc.

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:

$$\begin{aligned}\mathbf{X} &= [\mathsf{x}, \mathsf{X}]\\ \mathbf{Y} &= [\mathsf{y}, \mathsf{Y}]\\ \mathbf{Z} &= [\mathsf{z}, \mathsf{Z}]\end{aligned}$$

These interval bounds in our case are the coordinates of the octant's 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 [21] 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 $F(x, y, z) = 0$. 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.¸

```
If zero is contained into the interval calculated by applying the
   inclusion function to the octant
   Then subdivide octant
   Else  reject octant
```

Figure 1: *Voxelization using the inclusion function*

The interval arithmetic operators are:

$$\mathbf{X} + \mathbf{Y} = [\mathsf{x} + \mathsf{y}, \mathsf{X} + \mathsf{Y}]$$

$$\begin{aligned}\mathbf{X} - \mathbf{Y} &= [\mathsf{x} - \mathsf{Y}, \mathsf{X} - \mathsf{y}] \\ \mathbf{X} \cdot \mathbf{Y} &= [\min(\mathsf{xy}, \mathsf{xY}, \mathsf{Xy}, \mathsf{XY}), \max(\mathsf{xy}, \mathsf{xY}, \mathsf{Xy}, \mathsf{XY})] \\ \mathbf{X} \,/\, \mathbf{Y} &= [\mathsf{x}/\mathsf{Y}, \mathsf{X}/\mathsf{y}] \text{ if } 0 \notin [\mathsf{y}, \mathsf{Y}]\end{aligned}$$

These operators are not enough for the functions used in practice. To include any algebraic expression we need:

$$\mathbf{X^n} = \begin{cases} [\mathsf{x}^n, \mathsf{X}^n] \text{ n odd or } \mathsf{x} >= 0 \\ [\mathsf{X}^n, \mathsf{x}^n] \text{ n even and } \mathsf{X} <= 0 \\ [0, \max(-\mathsf{x}, \mathsf{X})^n] \text{ n even and } 0 \in [\mathsf{x}, \mathsf{X}] \end{cases}$$

To include gaussians, which are useful as blending functions in “blobby” models [16, 2, 12, 3] we have:

$$e^{-\mathbf{X}} = [e^{-\mathsf{X}}, e^{-\mathsf{x}}]$$

Surfaces rotations are linear transformations accomplished by multiplying a 3×3 matrix of constants by the corresponding intervals using matrix multiplication rules. It is important to note that the order of the interval bounds resulting from a multiplication between a constant and interval depends on the sign of the constant. That is, if a is a constant:

$$\mathbf{X} \cdot a = \begin{cases} [\mathsf{x} \cdot a, \mathsf{X} \cdot a] & a \geq 0 \\ [\mathsf{X} \cdot a, \mathsf{x} \cdot a] & a < 0 \end{cases}$$

Any other function can be similarly converted to interval arithmetic by breaking the function into their monotonic intervals [6].

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. Our method is also independent from the rendering algorithm.

In opposition to all previous methods, 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. This allows us to mix in the same discrete space objects modeled in other ways. To avoid the high memory consumption, we store the voxels into an octree [25, 28, 26, 27, 24]. This allows us to achieve high resolutions with low memory consumption, since the majority of the scenes are almost empty. Once the surface is voxelized we need neither the surface's equation nor the conversion the voxels into polygons to visualize it. We have used two different rendering algorithms: a fast high resolution discrete Ray Tracing [27, 26] and a hardwired Z-Buffer. In the Z-Buffer approach each voxel is displayed as a 3D point with the voxel normal and color. In this way the scene can be interactively visualized with high quality shading. Other data can be stored into the voxels as shadows information. Shadows can be precalculated casting discrete rays to the light sources. Radiosity data can also be precalculated and stored into the voxels. All these advantages have already been stressed in [33]. The difference in our approach is that we use high resolution to increase image quality and we are able to visualize the scene in near to real time. Notice that we do not need neither special architectures, nor parallel approach. We use conventional machines with hardwired Z-Buffer capable of showing 3D points with normals. Even though we still cannot allow near close-ups, this can be solved by using lazy evaluation and storing the extra space in virtual memory. Smooth continuity in the movement can be guaranteed using an efficient cache system. However, a powerful machine is advisable for voxelizing pieces of the surface in real time.

3 Implicit Sweeps

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 very 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. Most of the time approximations of these surfaces are used for visualization purposes only. Parametric surfaces are very 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: capability to know if a point in the space is in, out or on the surface; a notion of *"distance"* between a point and a surface by only evaluating the implicit function at the point; and blending different surfaces for easy connectivity. Although implicit surfaces are not naturally convertible to polygons, they can be easily converted to voxels. Many robust conversion methods [12, 32, 24] exist. Using Interval Arithmetic, for example, is an efficient and general way [24, 22] to accomplish a robust conversion. Unlike polygons, which do not properly represent the original surface, voxels obtained by these methods completely envelop the surface. In this manner, voxels represent curved surfaces in a much better way than polygons can. Thus, implicit surfaces are a very attractive way to model objects and voxels are very 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 [31]; thus, if any of this expressions is zero the whole function is also zero. An important 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 high level tool to solve this problem, called *implicit sweep*. The technique is based in a function mapping real values to integers, which are considered as a *replication factor*. 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 surfaces implicitly where previously they could only be defined parametrically.

3.1 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:

$$f(x_1, x_2, \cdots, x_n) \colon \mathcal{R}^n \to \mathcal{N}$$

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

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

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 function:

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

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 sweeps. 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.

3.2 Calculating the replication factor in Interval Arithmetic

Interval Arithmetic is a key tool for converting implicit surfaces into voxels as seen in [24, 22, 29].

However, even the simple replication function shown in equation 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 can be 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 [21]) of the torus given in equation 3, we first calculate two replication factors, one for each interval bound:

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

$$i_1 = \ (int)\ \frac{(z_1 \pm \frac{b}{2})}{b}$$

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 equation 3, will be enough to handle all the surface replications inside an interval $[z_0, z_1]$. Notice that the first part of equation 3 can be calculated outside the loop, since it does not depend on z coordinates.

Fig. 2 shows a voxelized model containing a replicated torus using this technique.

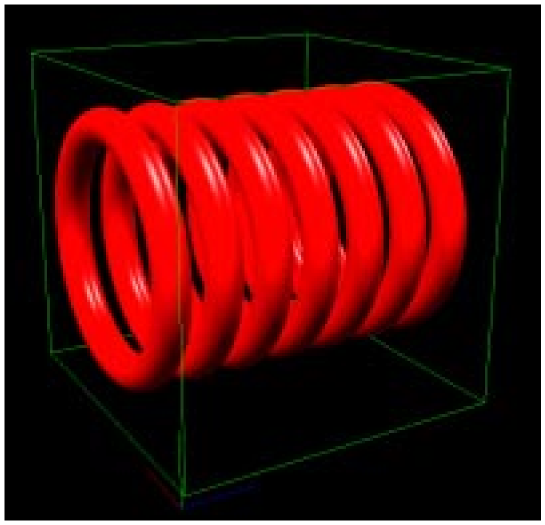


Figure 2: *Replication example of a torus*

4 Robust Voxelization of Parametric Surfaces

4.1 Spherical and Cylindrical Coordinates

Spherical coordinates are an extremely powerful tool for modeling objects which have a radial variation about an origin. It has many applications in CAD, art, entertainment, and related areas. Cylindrical coordinates, which can be seen as a special case of spherical coordinates, have special utility in industrial design, where many industrial products have cylindrical components.

This report describes a new robust technique for voxelizing spherical (or cylindrical) coordinates *implicit* functions, often defined as spherical (or cylindrical) coordinates *parametric* functions. This technique can specifically support CAD, ray-tracing, direct voxel visualization, or simulation. A very useful application of these techniques involves blending of spherical or cylindrical surfaces with rectangular implicit surfaces in the same voxel space, thus profiting from convenient modeling using spherical/cylindrical coordinates and easy blending with other implicit forms.

Spherical parametric functions, and every parametric function defined from $\mathcal{R}^2 \rightarrow \mathcal{R}^3$, are suitable for generating polygon models, but not for conversion to voxels. Parametric functions, based on polynomials, can be converted to voxels using incremental techniques (*e.g.* Bezier surfaces [14]) - but, due to the machine finite precision, incremental techniques are not robust. In addition, they can be of very little help in transcendental functions, as for example, in most spherical parametric surfaces. Although parametric surfaces are sometimes first converted to polygons and then converted to voxels [26], this solution is inappropriate, since the polygonal model is not an exact representation of the surface. Polygonal models are very convenient for flat surfaces but not for curved surfaces.

This section presents a novel method for voxelizing spherical/cylindrical implicit surfaces, often defined by parametric equations, simultaneously offering robustness and low complexity. Robustness evolves from the use of interval arithmetic, while the low complexity results from recursive subdivision of the space associated with interval arithmetic (see Section 2).

Converting spherical parametric functions to implicit functions would be very helpful for voxelization, since the algorithms cited above could be used for a high quality voxelization. However, the conversion can be very cumbersome, involving several algebraic manipulations. Rational parametric surfaces can be automatically implicitized [7, 10], and while several advances in recent years have significantly reduced conversion times, it still remains a time consuming process. Spherical parametric surfaces, which often involve transcendental functions, are particularly difficult to cope with.

The method proposed in this section spares the use of this automatic conversion by accepting the implicit function directly in a spherical/cylindrical coordinates format (see Section 4.1.1). The voxelization is done by subdividing the space in a recursive way, producing eight equal sized cubes at each interaction. Each of these cubes represents three intervals in interval arithmetic, which are converted to spherical intervals and then applied to the implicit spherical inclusion function for a containment test. This algorithm is shown to display running times asymptotically approaching an increasing factor of 4 every time resolution is doubled in every coordinate axis. This provides a great advantage over algorithms cited previously, which exhibit running times increasing by a factor of 8, every time resolution is doubled in every coordinate axis.

This method has additional advantages normally found only in implicit representation: a simpler and stand-alone equation, with no intermediate variables; only one equation instead of three, and the capability of being blended with other implicit surfaces.

4.1.1 Spherical/Cylindrical Parametric to Implicit

The problem of voxelizing spherical/cylindrical parametric functions can be bypassed by first transforming them to implicit functions and then voxelizing the corresponding implicit function. This requires transforming a function $f : \mathcal{R}^2 \rightarrow \mathcal{R}^3$ to a function $g : \mathcal{R}^3 \rightarrow \mathcal{R}$. This conversion is not always easy and several solutions might be possible, if one exists. Our goal is to simplify this task and to suggest that defining the function directly into the implicit form might be much simpler. Unfortunately, a general method to convert all functions does not exist and a case-by-case analysis is sometimes necessary. However, in many cases, when spherical/cylindrical coordinates are used, the conversion is easy and often produces a simplification of the function expression. Here we give some examples:

A torus is a surface that can be parametrically defined as follows:

$$\begin{aligned} x &= (a + b \cdot cos\ \phi) \cdot cos\ \theta \\ y &= b \cdot sin\ \phi \\ z &= (a + b \cdot cos\ \phi) \cdot sin\ \theta \end{aligned}$$

One simple way to express it in the implicit form is using cylindrical coordinates, remembering the circle implicit equation:

$$(r - a)^2 + y^2 - b^2 = 0 \qquad (4)$$

Equation 4 defines a circle with radius b translated to the point $(a,0)$. Since the horizontal axis is r, a rotating axis about y, it describes the surface produced by the rotation of the translated circle, generating a torus. Equation 4 is much simpler than its parametric counterpart and it is very easily obtained.

Consider the following function in the parametric form using spherical coordinates, which defines the radius by a function R=R(θ,ϕ) as follows (see Fig. 11-a, 11-b and 11-e):

$$\begin{aligned} R &= sin(n \cdot \theta) \cdot sin(m \cdot \phi) && (5) \\ x &= R \cdot cos\ \phi \cdot cos\ \theta && (6) \\ y &= R \cdot sin\ \phi && (7) \\ z &= R \cdot cos\ \phi \cdot sin\ \theta && (8) \end{aligned}$$

In this case the transformation is trivial, since only Equation 5 is necessary. As we can see Equations 6 to 8 serve only to convert from spherical to rectangular coordinates, which is no longer necessary when working directly in spherical coordinates. Then:

$$R = R(\theta, \phi) \rightarrow R(\theta, \phi) - R = 0$$

Which in our example gives the following implicit function:

$$sin(n \cdot \theta) \cdot sin(m \cdot \phi) - R = 0$$

The obtained implicit functions are not only very simple to define but also have interesting properties that the parametric form did not have - *i.e.*, this new form can be blended with other surfaces in either spherical, cylindrical or rectangular coordinates. The reason is very simple: implicit functions always return a scalar real value giving an idea of distance, regardless of the kind of coordinates it uses.

4.1.2 Converting to Spherical Intervals

The intervals generated by the recursive subdivision described in Section 2 are strictly rectangular. This is consistent with the voxelization process since voxels are essentially rectangular intervals. However, the equations used in this section are in spherical coordinates. Instead of converting the implicit function to rectangular coordinates, which can be a very cumbersome process, the rectangular intervals are converted to spherical intervals. Afterwards, they are applied to the inclusion function to see if the 3D interval contains a part of the surface, as explained in the previous section. This approach greatly simplifies the process and allows, at the same time, having rectangular coordinates and spherical coordinates implicit functions in the same space. Since the functions always return a scalar interval, independent of the kind of coordinates system, implicit surfaces can be blended together through a blending function.

Although simple, the conversion should be subdivided case-by-case to simplify calculations. Rectangular **X**, **Y** and **Z** intervals should be converted to spherical **R** (**R**=[r_0, r_1]), Φ (Φ=[ϕ_0,ϕ_1]) and Θ (Θ=[θ_0,θ_1]) intervals. Interval **R** bounds correspond respectively to the minimal and maximal radius value in the 3D region defined by the three rectangular intervals. The radius value is the distance between a certain point and the surface origin, as in the very definition of spherical coordinates. The 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 other two intervals (Θ and Φ) correspond to the angles θ and ϕ in spherical coordinates. By definition, θ is the angle between the radius and the X axis, and ϕ is the angle between the radius and XZ plane. However, Θ and Φ each refers to two radii, one for each angle bound, which are not the bounds of **R**. Perception of these angles bounds on the 3D cubic region (defined by **X**, **Y** and **Z**) is very subtle and complex. To be able to calculate them, we subdivide the problem into several different cases.

Obtaining Θ bounds

To find out the Θ bounds we have defined 9 possible cases (see Fig. 3). These cases cover the whole angular domain ([0, 2π]). In each of these different cases there is a different solution for finding Θ 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 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, 2π] interval, that is, the whole domain.

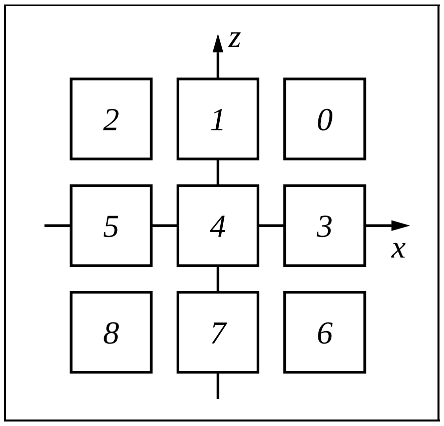


Figure 3: *Nine cases for determining* Θ *bounds*

Because of this, every cube in this case is trivially accepted, except at the last level (the voxel level) where the voxel might be rejected, that is, it is assumed that the surface does not pass through it.

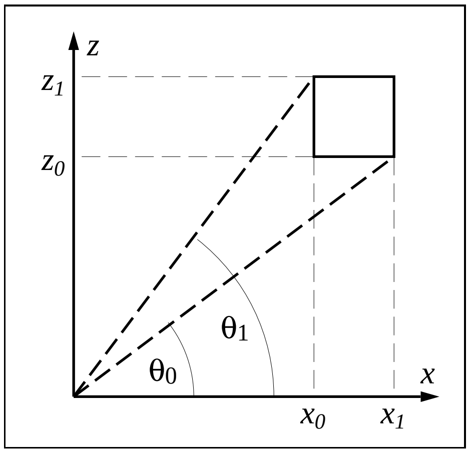


Figure 4: *Determining* Θ *bounds for case 0*

Figure 4 shows how to obtain angles θ_0 and θ_1 (Θ bounds) in case 0. In this case:

$$\begin{aligned}\theta_0 &= atan(z_0/x_1)\\ \theta_1 &= atan(z_1/x_0)\end{aligned}$$

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.

Obtaining Φ bounds

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 [0, 2π] 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.

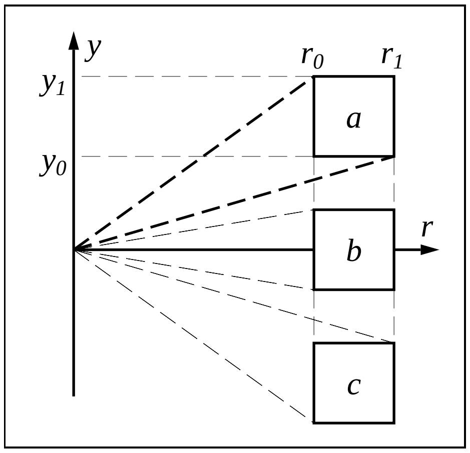


Figure 5: *Three cases for determining Φ bounds*

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 (Fig. 3 and 4) and case a (Fig. 5), ϕ_0 and ϕ_1 are calculated the in following way:

$$\begin{aligned} r_0 &= \sqrt{{x_0}^2 + {z_0}^2} \\ r_1 &= \sqrt{{x_1}^2 + {z_1}^2} \\ \phi_0 &= atan(y_0/r_1) \\ \phi_1 &= atan(y_1/r_0) \end{aligned}$$

4.1.3 Results

	n=9 m=18		*n=9, m=10*		*n=5, m=6*		*n=3, m=4*	
Res.	*Time*	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*
1024^3	653”	149.363	258”	113.738	307”	70.472	208”	48.621
512^3	160”	39.285	120”	30.777	74”	20.363	50”	14.980
256^3	35”	12.261	28”	10.308	18”	7.886	12”	6.597
128^3	6”	5.738	5”	5.324	3”	4.816	2”	4.511

Figure 6: Rasterization times for $sin(n{\cdot}\theta){\cdot}sin(m{\cdot}\phi){-}R{=}0$

The complexity of the algorithm is linear (O(N)) for the voxels which must be changed, since the octree is descended only once, and at each last node there is a constant amount of computation. This is true because all eight elements in the same cell are accessed linearly. Fig. 6 shows some voxelization times 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. Complexity increases as the resolution grows, but running time asymptotically approaches an increasing factor of 4 every time resolution doubles in every coordinate axis. At higher resolutions this tendency is even more clear (for n=3 and m=4, at 2048^3 resolution, estimated time is 13'52", that is 3'28"$\times$4, and real time was 13'53"). On the other hand, voxelization methods sampling all voxels [8] have a labored time increase factor of 8; each time resolution is doubled in each coordinate axis. For a 512^3 resolution our proposed algorithm can be roughly evaluated as being 512 times faster. This dramatic speed-up comes from the recursive space partition and from the fact that large empty regions can be easily eliminated using interval arithmetic. The voxelization in a given resolution can be seen as the set of voxels where empty regions could not be eliminated using interval arithmetic.

The image in Fig. 11-c 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 report. All other parts were voxelized using constraints implementing CSG operations of cylinders and planes. The voxelization time for this model at 512^3 resolution was 12 seconds.

The image in Fig. 11-d illustrates another example of practical application of our voxelization method using cylindrical coordinates. The contour of the non-cylindrical parts is given by the implicit function $r-(1.2+0.05\cdot sin(3\ \text{hspace1pt}\cdot\theta))=0$, that is, basically the same equation used for the gear. The voxelization time for this model at 512^3 resolution was 14 seconds.

All voxelizations were generated on a Challenger SGI workstation using a single 200MHz R10000 processor. Memory occupation in Fig. 6 is given in MBytes. All images were generated on the same machine using our interactive voxel visualization software. This software uses an octree to store the voxels and GL primitives to display each voxel as a 3D point.

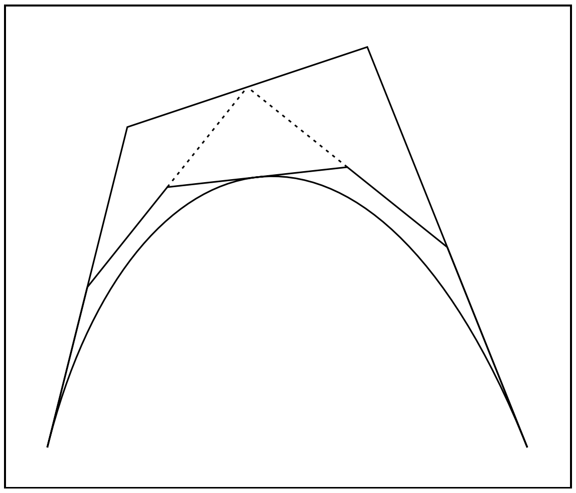


Figure 7: *De Casteljeau Subdivision*

4.2. Bezier Patches

Bezier patches voxelization have been accomplished using incremental techniques [14]. As pointed out before, incremental methods do not guarantee correctness. Unfortunately, the convertion of Bezier patches to the implicit form is very involved. In addition, Bezier patches are not defined in the whole parametric domain as in the spheric or cylindrical coordinates parametric surfaces seen before (section 4.1). Generally they are considered only in the interval [0 1] for both parametric variables, while the rest of the parametric domain is just ignored. This fact imposes very strict conditions for transforming it to the implicit form, leaving it even more difficult if ever possible. However, if it is possible, it would be a very powerful modeling tool, since this kind of surface would inherit all implicit surfaces properties. All those difficulties just mentioned have been forbidding this conversion.

Another way to treat this problem is generalizing De Casteljeau [4] algorithm to 3D. This algorithm implements a fast subdivision of the Convex Hull of Bezier patches. Bezier patches are always guaranteed to be contained inside the Convex Hull. Fig. 7 shows the De Casteljeau for a Bezier curve. The outside Convex Hull is subdivided in two smaller ones, one on the right and another on the left. The common point between both is a point over the surface. All the new points are obtained by calculating the middle points of the line segments indicated in Fig. 7.

A simple way to understand our 3D subdivision is assuming that the 16 Bezier patch control points are in a 4×4 matrix. The subdivision method applies De Casteljeau algorithm for each line of this matrix producing a 7×4 matrix. Then, De Casteljeau algorithm is applied to each column of the later matrix, producing a 7×7 matrix containing the the four sub-patches. The same algorithm is then applied recursively to each one of the four sub-patches. The process is illustrated in Fig. 8. Notice that the points a, b, c and d in the figure never change. The same happens to all the other points that touch the surface, namely the points in the following final 7×7 matrix positions (3,0), (0,3), (3,3), (6,3) and (3,6). These points are very critical for guaranteeing the robustness of the approach and must be exact. This is an important restriction. The best way to solve this problem is using multiple precision calculation. This is not difficult to implement since the calculations involved in the subdivisions are only additions and shifts.

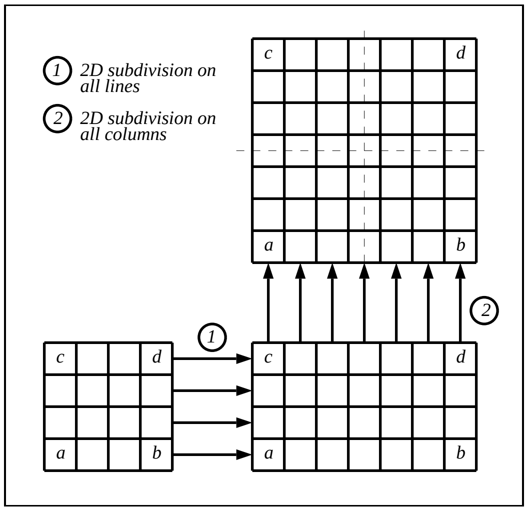

1 2D subdivision on all lines
2 2D subdivision on all columns
c
d
a
b
1
2

Figure 8: *3D subdivision using 2D De Casteljeau algorithm (Fig. 7)*

The voxelization algorithm would apply this subdivision algorithm until a certain condition is met. This condition should be when the patch is so flat that further subdivisions would not contribute to eliminate none of the existing voxels for that patch. One solution would be when points a, b, c and d are in the same voxel. This was tried but the solution proved to be too conservative, requiring too many subdivisions. Therefore an efficient stop condition is still required for an efficient robust voxelization using this technique.

5 Efficient Parallel Voxelization using Recursive Subdivision

The transformation of geometric surfaces into voxels is a very 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: 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 [1]. This algorithm exhibits fairly good results with up to 4 processors. For more than 4 processors the results are not as satisfying. As in [1], we use a shared-memory machine and get approximately the same behavior, but with better results.

Unfortunately, the greatest limitation of the algorithm in [1] is the assumption that the containment of a surface into an octant can be known at any moment. For certain subdivision algorithms [12, 6, 32, 24] 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 very 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, the voxel can be stored into the octree. This implies that the octree storage must be very fast. Thanks to the efficient octree traversal algorithm presented in this report, the time of storing voxels in the octree is negligible in relationship to the rest of the task.

The goal of 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. These tasks are very time-consuming yet parallelization is desirable. 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; kill the slave processes when the work is done, and displays the voxelized scene. This approach has very promising results, as shown in section 5.3.

5.1 Efficient 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. 9. 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 non-null pointer points to another array of eight pointers, further subdividing the region. This process continues until the leaf node is found, where each non-null 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. 9). We use this bit to filter the coordinates bits and to control the algorithm as in the octree ray traversal algorithm in [26].

The algorithm in Fig. 9 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 $\leftarrow$; the logical commands are written with its names (**and** and **or**) instead of symbolically, and the recursive stack operations are denoted by **push** (to put and element into the stack) and **pop** (to remove an element from the stack - a **pop** without argument only affects the stack pointer).

Once initialized, 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

```
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 reachead. Exit loop */
       }
     X_ant←X; Y_ant←Y; Z_ant←Z;
     *pcel←input;
   }
```

Figure 9: Octree traversal algorithm

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 fast 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 **pcell=0*). The code is quite straightforward, thus no further details are given here. See [26] for a deeper view in this part. Also see [30] which uses similar techniques, but for a proprietary linear octree.

5.2 Efficient 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 non-idle slaves queues in search of their results to store them into the stack or, at the leaf level, into 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.

```
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 $X_S$,$Y_S$ and $Z_S$ for sub-octant;
            if (octant($X_S$,$Y_S$,$Z_S$,my→level) may contain the surface)
              { index ← index+1;
                if (my→level is leaf);
                  { put voxel($X_S$,$Y_S$,$Z_S$,normal) in my→queue[index]
                  }
                else put octant($X_S$,$Y_S$,$Z_S$) 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 succesful give octant to slave;
          }
      }
    kill all slaves;
  }
```

Figure 10: Parallel recursive subdivision algorithm

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 into 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. 10.

5.3 Results

	Scene 1		*Scene 2*		*Scene 3*	
slaves	*time (sec.)*	*yield*	*time (sec.)*	*yield*	*time (sec.)*	*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%

Table 1: Performance results for a voxelization resolution of 512^3

Table 1 summarizes our results for a resolution of 512^3. Scene 1 is shown in Fig. 11-f; Scene 2, in Fig. 11-e; and Scene 3, in Fig. 11-a. 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 ($\frac{t_1}{n}$, 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:

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

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 behavior is probably linked to the fact that this scene contributes to a better cache coherence. Task distribution is centralized in the work stack, but neighbor regions have a tendency to be evenly distributed among the processors. When they are done, all information to be copied to the octree has the tendency to be in approximately the same area, thus increasing the cache coherence in certain cases. The surprising drop of performance for levels 5 and 6 could also be related to these problems. Thus, Scene 3 can be considered a good test scene for further improvements of the algorithm. The improvements can be obtained by forcing the same conditions for other scenes through the implementation of a smarter task distribution algorithm which takes these facts into account.

The results in Table 1 are very interesting and show that the algorithm deserves further consideration. This is very good news since voxelization times are strongly dependent on the area of the surface to be voxelized. Therefore, efficient parallel algorithms might be helpful in this field. Our results show that the technique presented here is very promising.

6 Robust Voxelization of polygonal meshes

Polygonal meshes could be initially thought of as CSG intersections of planes. In this case, the object can be represented implicitly by their plane equations using Ricci's set theoretic operators

[19, 17]. This would be a very convenient way to represent a polygon mesh, since it inherits all implicit surfaces desirable properties, and would allow the robust voxelization of polygon meshes by applying the algorithm for implicit surfaces explained in section 2. However, this is true only for meshes defining convex objects. In addition, the process of determining if an object is convex is complex (one method is verifying if every object vertex substituted in every plane equation is less than or equal to zero).

Star-shape objects (convex objects are a particular case of star-shape objects) defined by polygons can be represented implicitly, having one central point, referred as *"object center"*, from where all distances would have to be calculated. For any given point in the space a straight line can be defined between that point and the object center. Then, the intersection point between this line and the correspondent polygon should be calculated. The value of the implicit function in a point would be the distance between that point and the calculated intersection point. This process is simple but time consuming. In addition, it is restricted to star-shape objects and not for general polygon meshes. However, with this approach the algorithm seen in section 2 can be applied to produce robust voxelization of this kind of objects.

In these two approaches the polygonal mesh is treated as an implicit function and voxelized as an implicit surface. An advantage of using these approaches is that all implicit surfaces properties, such as blending, are maintained. However they lack generality. Not every polygonal meshes are representable.

Another method is considering, as in the first approach, the polygons plane equations. The recursive subdivision method of section 2 can be applied by considering these plane equations as implicit surfaces. At each subdivision level, the octants are tested against the the interval arithmetic version of the plane equations. If the octant belongs to a given plane, then it is tested against the prism formed by the polygon largest projection plane formed by two coordinate axis. This prism is perpendicular to the projection plane, that is, parallel to the third coordinate axis. The prism is produced in a preprocessing stage by calculating the 2D normal vectors for each polygon edges at the polygon plane. For each polygon edge defined by two points on the polygon plane, $P_0(x_0, y_0)$ and $P_1(x_1, y_1)$, these normals are calculated as follows:

$$\begin{aligned}\vec{N} &= x \cdot \vec{\imath} + y \cdot \vec{\jmath} \\ x &= -(y_1 - y_0) \\ y &= (x_1 - x_0)\end{aligned}$$

Once these normals are obtained as indicated, the plane equations are calculated. The CSG intersection of these planes defines the prism. To test if the octant is inside the prism it is enough to test if the resulting interval lower bound is always negative, after applying the octants intervals in each interval plane equation of the prism. This assumes the polygons are convex. If the octant is inside the prism, the corresponding polygon is included in the list of polygons to be analyzed in the following subdivision. The process follows in this way until the octant is the size of a voxel in the desired volume where the mesh is being voxelized. If there is only one polygon in the list, this is the polygon assumed to be contained by the voxel, otherwise the voxel might contain a vertex, defined by at least three polygons, or an edge defined by two polygons.

The resulting voxelization is guaranteed to be robust, because of the recursive subdivision and interval arithmetic. In addition, it allows any kind of polygon meshes, including open meshes. However, the implicit surfaces features are lost, because of the use of the prism to filter voxels which are outside the polygon. Several potential neighborhoods are not considered and, thus, compromising desirable features such as blending. In order to obtain implicit surfaces properties, the prisms would have to be parallel to the normals of the polygons (thus, respecting Vonoroi boundaries) and the regions outside the prisms should be classified in relationship to an edge or vertex. Nevertheless, the process starts to become very complex for non-convex polytopes, since an octant could be classified inside the domain of several polygons, edges or vertex.

The algorithm described before was implemented using bitmaps where each bit corresponds to one polygon of the model. Initially, all the bitmap bits are set to one in order to consider all the polygons. As the subdivision advances, the polygons that are discarded for a given octant are reset in the bitmap for the following subdivision. If there are no marked polygons for a certain octant, the octant is rejected.

7 Conclusion

We have proposed in this technical report several techniques for robust voxelization of surfaces. The use of spatial recursive subdivision and interval arithmetic are the key for most of the methods presented. The space is subdivided recursively and each octant is considered as three intervals which are then tested against the interval arithmetic implicit function describing the surface. If the octant does not contain the surface it is rejected, otherwise it is further subdivided until the final resolution is reached. This method guarantees that no part of the surface is ever missed, defining a new concept in this domain. This concept allows to voxelize an object starting from a previous voxelization of the same object, instead of revoxelizing the object again. This creates a new paradigm for 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 report.

We have also shown original algorithms for parallel implementation of these voxelization algorithms, and robust voxelization of: polygon meshes, Bezier Surfaces using De Casteljeau subdivision and implicit surfaces expressed in spherical and cylindrical coordinates. In addition we show a new concept called "implicit sweep", where an implicit surface can be replicated without extra evaluation cost, also showing how to voxelize them robustly using interval arithmetic.

References

[1] M. A. Bauer, S. T. Feeney, and I. Gargantini. Parallel 3D Filling with Octrees. *Journal of Parallel and Distributed Computing*, 22:121–128, 1994.

[2] H. B. Bidasaria. Defining and Rendering of Textured Objects through The Use of Exponential Functions. *Graphical Models and Image Processing*, 54(2):97–102, March 1992.

[3] James Blinn. A Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[4] P. de Casteljau. courbes et surfaces a poles. *Andres Citroen Automobiles*, 1959.

[5] J. M. Chassery and A. Montanvert. *Géométrie Discrète* . Editions Hermès, France, 1991.

[6] Tom Duff. Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry. *Computer Graphics*, 26(2):131–138, July 1992.

[7] George Fix, Chih-Ping Hsu, and Tie Luo. Implicitization of Rational Parametric Surfaces. *Journal of Symbolic Computation*, 21(3):329–336, March 1996.

[8] Pascal J. Frey and Houman Borouchaki. Finite Element Meshes by Means of Voxels. In *6th DGCI'96 - Discrete Geometry for Computer Imagery*, pages 165–172, Lyon, France, November 1996. Springer.

[9] D. H. Greene and F. F. Yao. Finite Resolution Computatinal Geometry. In *27th IEEE Symp. on Found. Comp. Sci.*, pages 143–152, Toronto.

[10] Christoph M. Hoffman. Implicit Curves and Surfaces in CAGD. *IEEE - CGA*, 13(4):79–88, 1993.

[11] David Jevans and Brian Wyvill. Adaptative Voxel Subdivision for Ray Tracing. In *Proceedings of Graphics Interface '89*, pages 164–172, Toronto, Ontario, June 1989. Canadian Information Processing Society.

[12] Devendra Kalra and Alan Barr. Guaranteed Ray Intersections with Implicit Surfaces. *Computer Graphics*, 23(3):297–306, July 1989.

[13] A. Kaufman. An Algorithm for 3D Scan-Conversion of Polygons. In *Eurographics'87*, pages 197–208, Amsterdam, August 1987. North Holand.

[14] Arie Kaufman. Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes. *Computer Graphics*, 21(4):171–179, July 1987.

[15] Arie Kaufman, Daniel Cohen, and Rony Yagel. Volume Graphics. *IEEE Computer*, 26(7):51–64, July 1993.

[16] Shigeru Muraki. Volumetric Shape Description of Range Data using "Blobby Model". *Computer Graphics*, 25(4):227–235, July 1991.

[17] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function Representation in Geometric Modeling: concepts, Implementation and Applications. *The Visual Computer*, 25(4):227–235, July 1996.

[18] J. P. Reveillès. *Géométrie Discrète, Calcul en nombres entiers et Algorithmique*. PhD thesis, Université Louis Pasteur de Strasbourg, 1991.

[19] A. Ricci. A constructive Geometry for Computer Graphics. *The Computer Journal*, 16(2):157–160, May 1973.

[20] A. Rosenfeld and R. A. Melter. "Digital Geometry". *The Mathematical Intelligencer*, 11(3):69–72, 1989.

[21] John M. Snyder. Interval Analysis For Computer Graphics. *Computer Graphics*, 26(2):121–130, July 1992.

[22] Nilo Stolte. *Espaces Discrets de Haute Résolutions: Une Nouvelle Approche pour la Modelisation et le Rendu d'Images Réalistes*. PhD thesis, Université Paul Sabatier - Toulouse - France, April 1996.

[23] Nilo Stolte and René Caubet. A fast scan-line method to convert convex polygons into voxels. In *Compugraphics'93*, pages 164–170, Alvor, December 1993. Harold P. Santo.

[24] Nilo Stolte and René Caubet. Comparison between different Rasterization Methods for Implicit Surfaces. In *Visualization and Modeling*, pages 434–447, Leeds, December 1995. University of Leeds.

[25] Nilo Stolte and René Caubet. Discrete Ray-Tracing High Resolution 3D Grids. In *The Winter School of Computer Graphics and Visualization 95*, pages 300–312, Plzen, February 1995. Vaclav Skala.

[26] Nilo Stolte and René Caubet. Discrete Ray-Tracing of Huge Voxel Spaces. In *Eurographics 95*, pages 383–394, Maastricht, August 1995. Blackwell.

[27] Nilo Stolte and René Caubet. Fast High Definition Ray Tracing Implicit Surfaces. In *5th DGCI - Discrete Geometry for Computer Imagery*, pages 61–70, Clermont-Ferrand, September 1995. *Université d'Auvergne*.

[28] Nilo Stolte and René Caubet. Lancer de Rayons Discret pour des Grilles de Hautes Résolutions. In *Montpellier'95 - L'interface des Mondes Réels et Virtuels*, pages 335–344, Montpellier, June 1995. EC2 & Cie.

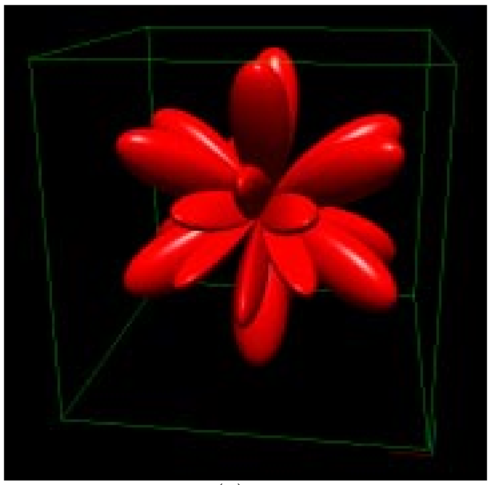
[29] Nilo Stolte and Arie Kaufman. Robust Voxelisation for Implicit Surfaces in Spherical Coordinates. Submitted to *IEEE - CGA*, 1997.

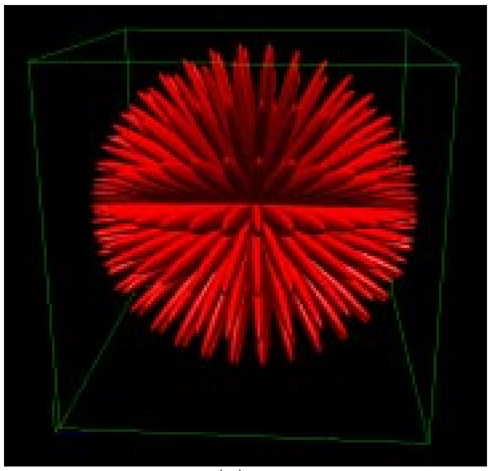
[30] Kelvin Sung. A DDA Traversal Algorithm for Ray Tracing. In *Eurographics'91*, pages 73–85, Amsterdam, June 1991. North Holand.

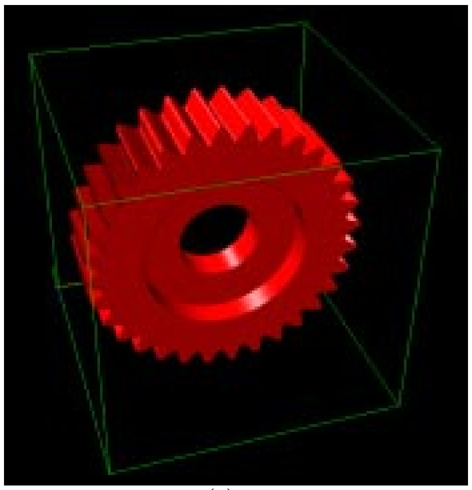
[31] Gabriel Taubin. Distance Aproximation for Rasterizing Implicit Curves. *ACM Transactions on Graphics*, 13(1):3–42, January 1994.

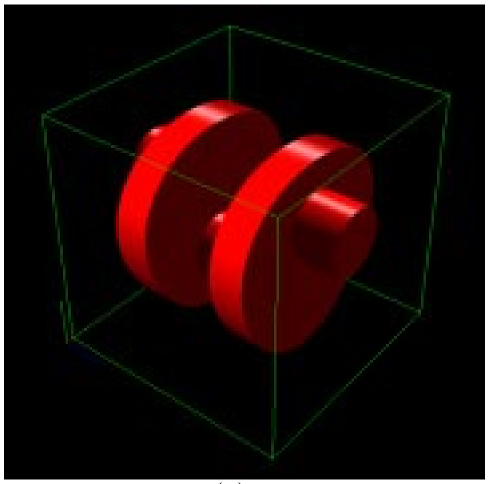
[32] Gabriel Taubin. Rasterizing Algebraic Curves and Surfaces. *IEEE - CGA*, pages 14–23, March 1994.

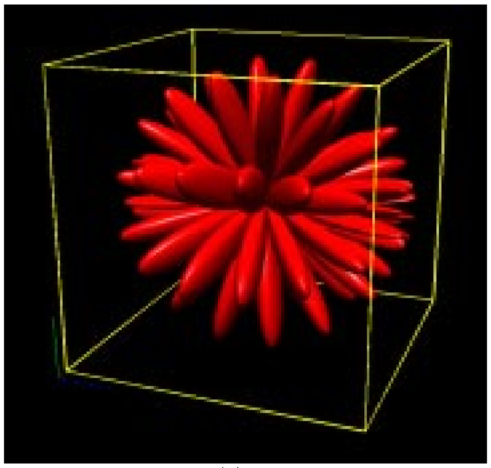
[33] Roni Yagel, Daniel Cohen, and Arie Kaufman. Discrete Ray Tracing. *IEEE - CGA*, 12(5):19–28, 1992.


(a)


(b)


(c)


(d)


(e)

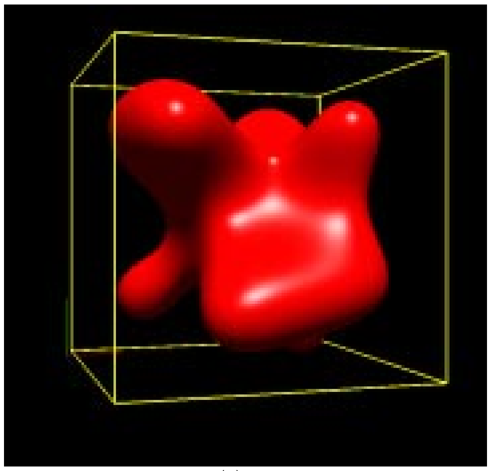

(f)

Figure 11: (a) $sin(3\theta)\cdot sin(4\phi)-R=0$ and (b) $sin(9\theta)\cdot sin(18\phi)-R=0$ (c) gear (d) a part of a crank shaft (e) $sin(4\theta)\cdot sin(8\phi)-R=0$ and (f) Ten spheres blended