Graphics using Implicit Surfaces with Interval Arithmetic based Recursive Voxelization

Nilo Stolte

School of Computer Engineering
Nanyang Technological University
Singapore 639798

nilo.stolte@online.fr

ABSTRACT
Interval arithmetic is a powerful, convenient and efficient tool to solve graphics problems. However, the number of research papers concerning this subject has been considerably small even though the advantages of interval arithmetic overweigh their disadvantages. This work shows a collection of techniques successfully implemented using intervals that deal with graphics based on recursive voxelisation. Voxelization is seen here as a generic tool to harness intervals which can be extended to other applications.

Voxelization is the transformation of a continuous surface into voxels. Intervals allow the voxelization to be done recursively, since interval arithmetic guarantees that the zero of an implicit function, which describes the surface to be voxelized, cannot occur into a three-dimensional region. This allows the elimination of the whole region from subsequent analysis allowing an efficient time recursive subdivision.

Even though voxelization is used here for high quality interactive display of implicit surfaces, it could also be used in other rendering algorithms (such as ray tracing), in transformation of implicit surfaces to polygons (using marching cubes algorithm), or for other more generic tasks such as calculation the volume inside an implicit surface.

KEY WORDS
Voxel, Voxelization, 3D Visualization, Interval Arithmetic, Implicit Surfaces

1 Introduction

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 [11] or other techniques to correctly estimate the functions' zeros [13]. In addition, several known techniques [9] can be used to avoid this interval arithmetic drawback.

Interval arithmetic when used for voxelizing implicit surfaces, allows the use of logarithmic time recursive subdivision with efficient elimination of regions that cannot contain parts of the surface, that do not contain a zero for the function defining the surface [6, 11, 12]. Recursive subdivision already contributes for reducing the intervals drawback, since the size of the intervals shrink instead of growing, thus allowing classic interval arithmetic to be used to efficiently voxelize a variety of implicit surfaces. However, this might not always be the case, particularly when a surface is not defined in a closed form.

According to [9], given a region S and each interval obtained from applying interval analysis in the subdivided parts of S, they can be combined to produce one resulting interval for S (its lower bound being the minimal of the lower bounds and the upper bound being the maximal of the upper bounds of the intervals in the subdivided parts of S) that is in fact tighter than the interval obtained by applying interval analysis in S.

This means that tighter estimates can be obtained by simply subdividing one or two more levels to verify if a particular subdivided 3D region really contains a piece of the surface. But from the analysis of the real complexity of the recursive voxelization algorithm by measuring the times of the program reveals that applying this method for every subdivided 3D region increases the processing time of 4 and 16 times, respectively.

In our on-going research tighter interval arithmetic operations significantly increases the performance in some cases. We are currently examining more performing interval arithmetic operations by dealing with groups of operations instead of applying them individually as done in classical interval arithmetic. However, the analysis is ad-hoc and difficult in cases where the expressions contain terms involving more than one variable.

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"* [10, 1, 7, 2, 11] where generally only spheres and super-ellipsoids are used. In our experience, as shown in this article, interval arithmetic behaves very well with *"blobby models"* even if hundreds of primitives and quite complex blending functions with degrees up to 128 have been used.

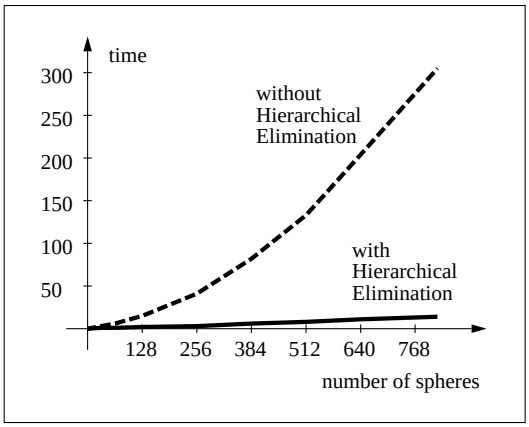

Figure 1. Voxelization times with and without Hierarchical Elimination

2 Efficient Voxelization of Blobby Models

To efficiently voxelize complex blobby models instead of evaluating all n primitives of a blobby model at every subdivision level, the algorithm here proposed evaluates n primitives at the first level trying to eliminate the primitives that will not contribute to the subsequent lower subdivision levels. The elimination process, here called hierarchical elimination, is active at every subdivision level, thus eliminating a primitive as early as possible. The goal is to minimize n_l (number of primitives at a given subdivision level l) in the last level of the subdivision, in which N_l (number of octants at a given subdivision level l) is maximal (N). In interval arithmetic a blobby model is represented by its *inclusion function* given in equation (1).

$$\mathsf{F} = \left(\sum_{i=1}^{n} b_i \cdot \mathcal{G}_i(|\mathcal{F}_i(\mathsf{X},\mathsf{Y},\mathsf{Z})|) \right) - C \qquad (1)$$

$\mathcal{G}_i$ is the inclusion function of the blending function $g_i(r_i)$, where r_i=$|f_i(x,y,z)|$ and f is the primitive ($\mathcal{F}_i(\mathsf{X},\mathsf{Y},\mathsf{Z})$ is its inclusion function while X, Y, Z are the intervals defining a cubic region in $\mathcal{R}^3$), the function that gives the shape of the blobby. This blending function is given by the following expression, knowing that R_i is a real constant and a_i is power of two:

$$g_i(r_i) = \begin{cases} \frac{1}{R_i^{2a_i}} \cdot (r_i - R_i^2)^{a_i} & \text{if} \quad r_i <= R_i \\ 0 & \text{if} \quad r_i > R_i \end{cases} \qquad (2)$$

A primitive can be eliminated when the upper bound of $\mathcal{G}_i$ is zero (since g_i is decreasingly monotonic) or simply if the lower bound of $\mathcal{F}_i(\mathsf{X},\mathsf{Y},\mathsf{Z})$ is greater or equal to R_i^2.

To implement the algorithm each blobby is represented by a position in an array with $n+1$ positions, each one containing a pointer to the next position (each individual blobby is identified by an index in this array which can then be used in the array or arrays containing the real blobby data), in which the first is dummy (pointing to the first active primitive - always the first blobby at the start), functioning as the head of the list containing the active primitives that are at a certain level. At the top subdivision level, this main list contains all the primitives. As the subdivision proceeds, primitives are eliminated by unchaining them from the main list. A stack contains the first dummy position of a list in each level. The unchained eliminated primitives from the main list are chained to the list headed by the stack position of the current subdivision level. These eliminated primitives still physically belong to the array of the main list but they are just not linked anymore to it. When the recursion goes up the stack, the eliminated primitives at each level are chained back to the main list. In this way only one array of primitives is used for all the lists and a significant amount of memory is saved.

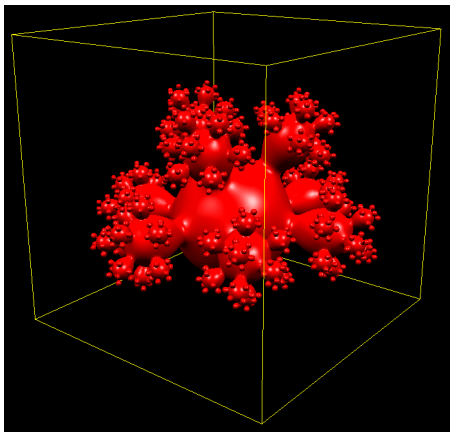

Figure 2. Voxelized Sphereflake with 820 blended spheres

The Sphereflake, a classical computer graphics test scene, was used to test the algorithm. The original Sphereflake generation program was modified to blend the spheres together using the local blending function given in equation (2). The a_i values were all equal to 64 (degree 128) except for the first level sphere where $a_i = 4$. The experiments were performed by measuring the voxelization time, with and without hierarchical elimination, in a voxel space resolution of 512^3, by gradually increasing the number of spheres until the maximum of 820 spheres was reached. With 820 spheres this scene took 305 seconds to voxelize without hierarchical elimination and only 15 seconds with hierarchical elimination. This corresponds to a speedup of more than 20. The results are summarized by the curves in Fig. 1 and the voxelized object shown in Fig. 2.

It is clear from these results that when the number of spheres increases the time grows exponentially for a voxelization without hierarchical elimination, whereas with hierarchical elimination the time is almost constant.

3 Voxelization of Implicit Surfaces in Spherical and Cylindrical Coordinates

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, which are converted to spherical intervals and then applied to the implicit spherical inclusion function for a containment test [12].

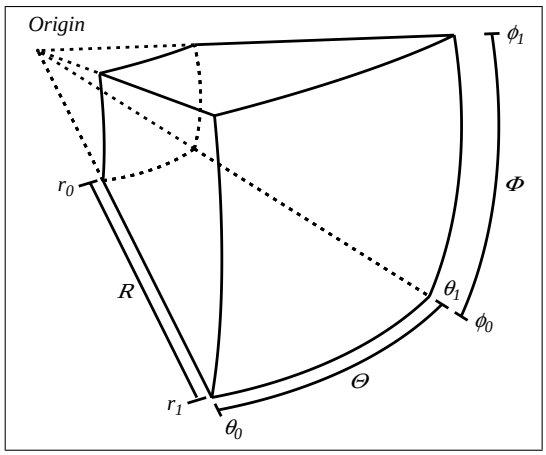

Figure 3. *Spheric Interval:* $\mathbf{R}[r_0, r_1]$,$\mathbf{\Theta}[\theta_0, \theta_1]$, $\mathbf{\Phi}[\phi_0, \phi_1]$

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 follows:

$$\begin{aligned} \mathbf{R} &= [r_0, r_1] \\ \mathbf{\Theta} &= [\theta_0, \theta_1] \\ \mathbf{\Phi} &= [\phi_0, \phi_1] \end{aligned}$$

The region defined by these intervals is not cubic, but has the form shown in Fig. 3. 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. Therefore, the rectangular **X**, **Y** and **Z** intervals have to be converted to spherical **Θ**, **Φ** and **R** intervals.

To find out the **Θ** bounds we have defined 9 possible cases (see Fig. 4). 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. 4 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.

Figure 5 shows how to obtain angles θ_0 and θ_1 (**Θ** bounds) in case 0. In this case:

$$\begin{aligned} \theta_0 &= atan(z_0/x_1) \quad (-\infty\ rounding)^\dagger \\ \theta_1 &= atan(z_1/x_0) \quad (+\infty\ rounding)^\dagger \end{aligned}$$

†suggested rounding modes to guarantee numerical robustness.

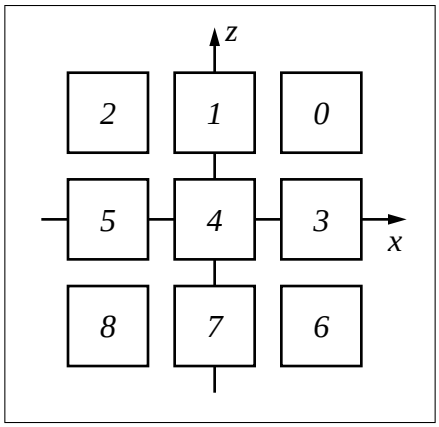

Figure 4. *Nine cases for determining* **Θ** *bounds*

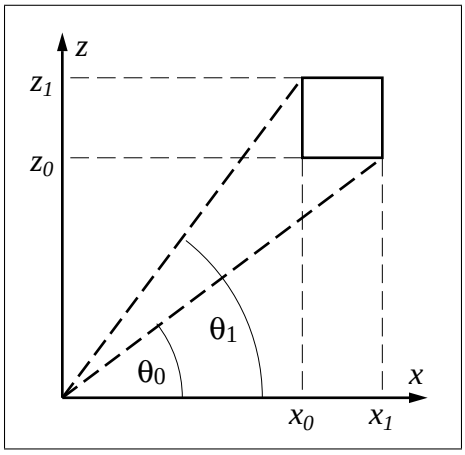

Figure 5. *Determining* **Θ** *bounds for case 0*

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. 5. This is done to define a spherical interval which contains the cubic one.

The **Φ** 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 [0, 2π] domain. Case 4 is eliminated from the analysis, since this case has no solution because it covers the whole domain. This case is dealt by always accepting and subdividing it until other cases are found or the last level is found, when it is then rejected. This is the reason why only three different cases are necessary to fully define **Φ** bounds.

The three cases are indicated in Fig. 6. Axis *r* in Fig. 6 is a rotating axis over XZ plane. Suppose that the cube being considered is case 0 (Figures 4 and 5) and case *a* (Fig. 6), 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**):

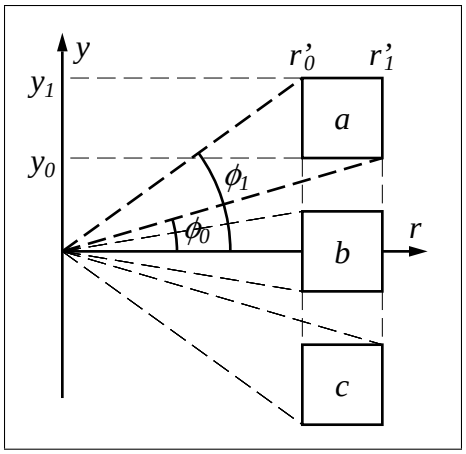

Figure 6. *Three cases for determining* $\mathbf{\Phi}$ *bounds*

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

†suggested rounding modes to guarantee numerical robustness.

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.

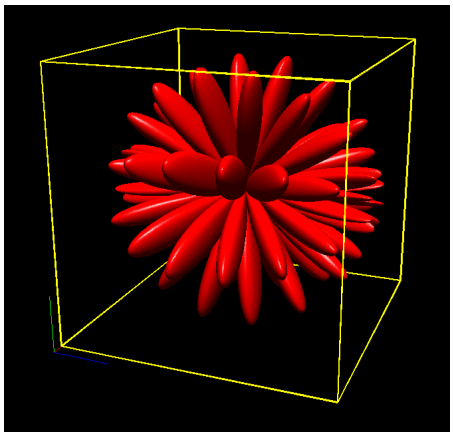

Figure 7. $sin(4\theta)\cdot sin(8\phi)-R=0$

Table 1 shows some voxelization times (in seconds) and voxel occupancy (in millions of voxels) for the surface given by the equation $sin(n\cdot\theta)\cdot sin(m\cdot\phi)-r=0$ for different values of n and m and different 3D resolutions. One of the voxelized objects can be seen in Figure 7. All voxelizations were generated on a laptop with a Pentium III 1.8 GHz processor. Voxel occupation (indicated in "*Occ.*" columns) in Table 1 is given in millions of occupied voxels.

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=3, m=4*	
Res.	*Time*	*Occ.*	*Time*	*Occ.*	*Time*	*Occ.*
1024^3	171"†	23.8	123"†	18.	49"	7.4
512^3	37"	5.87	28"	4.45	13"	1.85
256^3	8"	1.42	7"	1.08	3"	0.46

†swapping occured.

4 Voxelization of Implicitly defined Cyclical Parametric Surfaces

Representing cyclical parametric surfaces implicitly is not a trivial task. Cyclical parametric surfaces generally have angular parameters used to generate translations dependent on these angles. The problem is that it is impossible to determine an exact angle only based on the rectangular coordinates in the implicit representation without any extra hint to indicate in which cycle a certain part of the surface is located. For this reason, cyclical parametric surfaces are generally reduced to only one cycle when represented in the implicit form.

Infinite implicit replication [12] is able to restore the missing information in a quite simple way and at a quite low cost. The case of an infinite helicoidal torus (a mechanical spring) is going to be used onwards to illustrate and to explain the technique. The parametric equation of a mechanical spring can be defined as shown in equation (3).

$$\left\{ \begin{array}{rcl} x & = & (R + a\cos\phi)\cos\theta \\ y & = & (R + a\cos\phi)\sin\theta \\ z & = & a\sin\phi + \frac{b\theta}{4\pi} \end{array} \right. \quad (3)$$

This is the parametric equation of the torus with an extra factor $\frac{b\theta}{4\pi}$ added to the z coordinate. It is possible to represent this surface implicitly by subtracting the same factor from the z coordinate as shown in the implicit equation (4), given in cylindrical coordinates.

$$(r - R)^2 + (z - \frac{b\theta}{4\pi})^2 - a^2 = 0 \quad (4)$$

However, the obtained surface has only one cycle of the infinite spring obtained with the parametric equation (3). To reestablish the infinite other cycles, the single cycle defined implicitly can be infinitely replicated using infinite implicit replication. But joining the different replications of this object implies interpenetration of the cyclical regions which is not previewed in the infinite implicit 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.

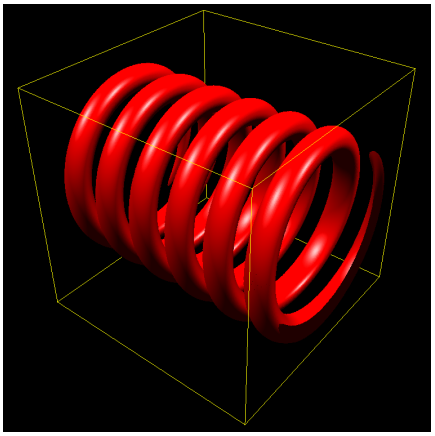

Figure 8. *Voxelized implicitly represented mechanical spring using replication*

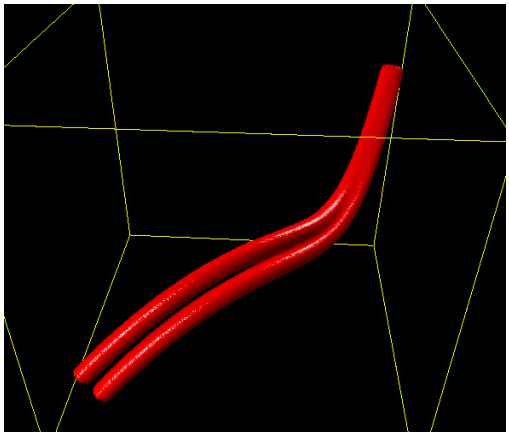

Figure 9. *Voxelized blending between two generalized cylinders*

In the case of the mechanical spring above, each cycle varies between $-\frac{b}{4}$ and $\frac{b}{4}$ in relationship to the middle of the replicated region (the "origin" of the cycle). The equation of the replicated object with $a = 0.1$, $b = 0.7$ and $R = 0.7$ is given in (5) and the voxelized object at a resolution of 512^3 using recursive voxelization and interval arithmetic is shown in Fig. 8.

$$\begin{aligned} \mathcal{A}(r,\theta,z) &= (r-R)^2+\left[z-b\cdot\left(i_{\mathcal{A}}+\tfrac{\theta}{4\pi}\right)\right]^2-a^2 \\ \mathcal{B}(r,\theta,z) &= (r-R)^2+\left[z-b\cdot\left(i_{\mathcal{B}}+\tfrac{\theta}{4\pi}-\tfrac{1}{2}\right)\right]^2-a^2 \\ \mathcal{F}(r,\theta,z) &= min(\mathcal{A}(r,\theta,z),\mathcal{B}(r,\theta,z))=0 \qquad (5) \\ & \theta\in[-\pi,\pi] \end{aligned}$$

Where,

$$i_{\mathcal{A}} = (\,int\,)\,\frac{(z\pm\frac{b}{2})}{b} \qquad i_{\mathcal{B}} = (\,int\,)\,\frac{((z+\frac{b}{2})\pm\frac{b}{2})}{b}$$

This example clearly shows that infinite implicit replication can also be used to represent other cyclical parametric surfaces implicitly by proceeding as shown above.

5 Generalized Cylinders Voxelization

Generalized cylinders [3, 4, 5] are defined by arbitrary two-dimensional contours and skeletons as opposed to the circle sweep in which the contour is a circle. In these both cases the normal vector of the plane containing the contour is aligned with the tangent vector of the skeleton curve at the point where the plane cuts the curve. In other words, the plane containing the contour is always perpendicular to the curve defining the sweep skeleton. A 2D contour in 3D is here defined by using the intersection between a surface and the plane perpendicular to the skeleton. If the surface is implicit, the resulting generalized cylinder is implicit.

The voxelization of implicit generalized cylinders defined in this way is an extension of an existing subdivision [6] and voxelization method [11, 12] for implicit surfaces. The extension proposed here is to include in the subdivision process an extra dimension corresponding to the parametrical variable t of a three-dimensional Bézier curve describing the skeleton of the generalized cylinder. The subdivision is a recursive procedure that breaks a hypercube into 16 equal parts at each interaction (as opposed to 8 equal parts of a cube in the three-dimensional case). Each of these 16 resulting hypercubes are then tested to verify if the swept surface is contained in the hypercube. If a hypercube is known to not contain the surface it is abandoned, otherwise it is further subdivided in other 16 hypercubes of one-sixteenth of the size of the original hypercube. This procedure continues until a certain desired number of levels is reached.

Figure 9 shows the voxelization of a generalized cylinder whose contour is two circles which gradually blend together. Artifacts in the image resulted from wrong normal calculation due to the under sampling of the parametrical variable t.

6 Visualization Method

The visualization method used to generate images for this article is based on high-resolution voxel spaces (512^3 resolution). Voxels are stored in an octree, thus allowing quite large discrete spaces without consuming a significant amount of memory. Normal vectors are calculated during the voxelization by evaluating the 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, textures or other information could also be stored as well. 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 an efficient BSP scheme to order the octants to display the voxels from back to front in the painter's algorithm fashion.

One of the goals of this experiment was to verify the gain obtained with accelerated graphics cards in relation-

ship to a software solution using advanced features of recent processors such as SIMD instructions. The conclusion of this analysis was that the software solution described here was slightly more efficient than using graphics cards that calculate illumination and hidden surface elimination in hardware. The SIMD instructions were responsible for a 40% optimization, while backface culling reduced the display time by roughly 50%.

The BSP ordering algorithm is based on the location of the viewer, as generally done in BSP traversals, in relationship to the middle splitting axis-aligned planes that divide an octant in eight equal sized cubes in the octree. The 3 Boolean values saying if the observer coordinates are above or on the respective planes are then grouped in three successive bits from right to left and stored in the variable *zyx*. The efficiency of the ordering algorithm comes from its extreme simplicity and reduced number of instructions. It uses an array (*order*) with 8 positions containing the index giving the real order of the octants in each level of the octree. Every time the algorithm goes down one level, it calculates the content of the array based on the value obtained in *zyx* variable. This is done by the following statements:

```
order[ zyx xor 7 ] = 0;
order[ zyx xor 6 ] = 1;
order[ zyx xor 5 ] = 2;
order[ zyx xor 4 ] = 3;
order[ zyx xor 3 ] = 4;
order[ zyx xor 2 ] = 5;
order[ zyx xor 1 ] = 6;
order[ zyx ]       = 7;
```

The variable *zyx* in the way it is calculated can be interpreted as the index of the octant where the viewer is looking from. The **xor** (exclusive or) allows obtaining the back to front order as required.

7 Conclusion

We have shown in this article several techniques involving graphics using implicit surfaces with interval arithmetic based recursive voxelization. The graphics are generated using a unique algorithm based on voxels and point rendering, where the voxels were previously obtained from implicit surfaces voxelization. For portability purposes OpenGL is only used to open the window and to transform window to viewport coordinates, otherwise everything is calculated in software. This display procedure is more efficient than using accelerated graphics cards.

These techniques show the feasibility of the use of interval arithmetic for efficient voxelization of implicit surfaces.

Voxelization is a quite powerful and simple tool that may be used for a variety of applications besides the obvious ones already stressed in volume graphics, conversion of implicit surfaces into polygons [8], ray tracing and radiosity. The recursive voxelization is a divide and conquer technique that can also be used for volume calculation, intersections calculations between implicit surfaces, collision detection, implicit surfaces bounding box calculation, implicit surfaces tiling or scan-conversion, etc.

References

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

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

[3] Willem F. Bronsvoort and Klok F. Ray tracing generalized cylinders. *ACM Transaction on Graphics*, 4(4), 1985, 291–303.

[4] Willem F. and Bronsvoort. A surface-scanning algorithm for displaying generalized cylinders. *The Visual Computer*, 8, 1992, 162–170.

[5] Erik de Voogt, Aaadjan van der Helm, and Willem F. and Bronsvoort. Ray tracing deformed generalized cylinders. *The Visual Computer*, 16, 2000, 197–207.

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

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

[8] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4), 1987, 163–169.

[9] S. P. Mudur and P. A. Koparkar. Interval Methods for Processing Geometric Objects . *IEEE - CGA*, February 1984, pages 7–17.

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

[11] Nilo Stolte and René Caubet. Comparison between different Rasterization Methods for Implicit Surfaces. In Rae Earnshaw, John A. Vince and How Jones (Ed.), *Visualization and Modeling*, 10 (Academic Press, 1997), 191–201.

[12] Nilo Stolte and Arie Kaufman. Novel Techniques for Robust Voxelization and Visualization of Implicit Surfaces. *Graphical Models*, 63(6), 2001, 387–412.

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