A Fast Algorithm to Display Octrees

Sharat Chandran
Indian Institute of Technology
Bombay, India
sharat@cse.iitb.ernet.in

Ajay K. Gupta
Informix
Oakland, USA
ajayg@informix.com

Ashwini Patgawkar
National Center for Software Technology
Bombay, India
ashwini@ncst.ernet.in

Abstract

The octree is a common data structure used to model 3-dimensional data. The linear octree is a compact representation of an octree. In this paper we deal with the problem of displaying three-dimensional objects represented using linear octree, and offer a new, efficient solution.

Our algorithm rearranges voxels of a linear octree in order of increasing distance from the viewer, This list is rendered using a back to front painter's algorithm, or a front-to-back algorithm. We compare our visible surface determination algorithm with that of a variant based on prior work.

Keywords: Linear Octree, Front-to-back, Painter's Algorithm, Sorting, OpenGL

1 Introduction

There are a number of modeling techniques [9] that are used in computer graphics to model 3D objects and the choice of the modeling technique depends on the application. For example Constructive Solid Geometry (CSG) is used in CAD/CAM packages based on which objects are designed and manufactured.

Our interest is in the use of the octree in the form of three dimensional variable blocks or cubes of different sizes. This model is often used for spatial representation of geological data like geographical terrain, mineral deposits, etc., (where a polygonal model representation is cumbersome or impossible), and finds practical use in a Geographical Information System (GIS). The use of the linear octree has been championed in [2], [6], [14] for several reasons. Predominant is the compact representation, and the ability to perform operations such as neighbor finding.

Such a representation also requires efficient display techniques. Techniques to display octree data in general form are given in [7], [13], [4]. Techniques to display linear octree encoded data is given in [5], [10].

1.1 Our contributions

We develop a fast algorithm for displaying octrees. Our algorithm runs in $O(N \log N)$ time where N is the total number of entities to be displayed. We have implemented our algorithm on three different platforms. Further, we also implemented a variant algorithm based on prior work. The proposed algorithm runs faster (see Section 4 for details) than prior methods.

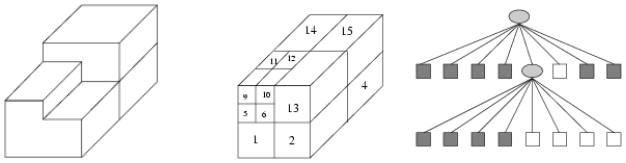
A highly desirable feature is the display of data even as the view plane normal [12] (VPN) is changed, perhaps interactively. Prior algorithms do not address this issue explicitly; the focus is on one specific view plane direction (recall that the BSP tree is most useful for polygonal data, and not for volumetric information). As the view plane normal changes, the underlying representation of the octree has to be changed. In contrast, our algorithm "sorts" the data on the fly based on the view plane normal to produce a correct ordering which can be rendered using the standard painters' algorithm, or the front-to-back technique.

1.2 Roadmap

The rest of this paper is organized as follows. In the next section, we formally define the problem. In Section 3, we motivate our algorithm using a weaker representation of spatial data. We also give our algorithm that uses a linear octree. In Section 4 we analyze the performance of our algorithm, and compare it to a variant. Final remarks are made in Section 5.

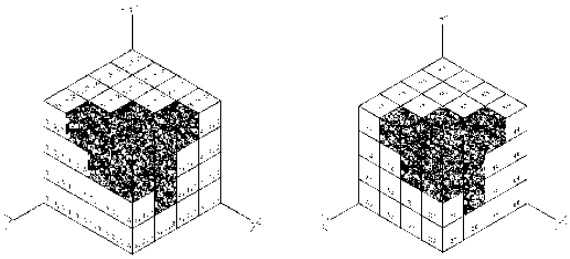
2 Definitions and Representation

An octree is a hierarchical tree structure used to represent the solid objects, such that each node corresponds to a (axis parallel) region of the three dimensional space. The octree takes advantage of spatial coherence to reduce storage requirements. If the object space corresponding to an octant is either entirely contained in the region or entirely disjoint from it, no further subdivision of the octant is made. This process is represented by a tree of degree 8 in which the root node represents the entire object, and the leaf nodes correspond to those cubes which no further subdivision is necessary. The number of subdivisions is said to be the *resolution* of the octree and the smallest size



(a) The object (b) A pointer based octree

Figure 1: Octree decomposition of simple staircase object.



(a) Cartesian representation (b) Octree encoding

Figure 2: Multiple encoding for voxels in an object.

octants are termed *voxels*. Figure 1(a) is an example of a simple 3D object (a staircase), whose octree block decomposition is given in Figure 1(b), along with its tree representation. Note that this figure is shown only for simplicity. In a GIS system, input data is *not* available in polygonal form.

2.1 Linear Octrees

An effective way of storing octrees for 3D representation of objects is given in [6]. We demonstrate this representation by means of Figure 2 with $n = 2$ being the resolution, and $C = 11$ represent the total number of voxels present in the data. All voxels, except (2, 2, 2), are visible (shaded area) in this example.

In the representation technique for linear octree each octant at every subdivision level is labeled [0..7], depending on its position as given in Table 1, and shown in Figure 2(b). A convention is adopted on what is meant by forward and backward, and the various geographical directions (such as east). In the example of Figure 2, east is in the direction of the unit vector $\hat{i}$ along the x axis, and forward refers to the situation when we are positioned on the positive z axis and looking in the direction of $-\hat{k}$.

Each voxel belonging to the object is represented

Octant	Label
forward-north-east	0
forward-north-west	1
forward-south-east	2
forward-south-west	3
backward-north-east	4
backward-north-west	5
backward-south-east	6
backward-south-west	7

Table 1: Encoding of the eight subdivisions of an octant.

by an octal integer in a weighted system where the digit of weight 8^{n-1} identifies the largest octant according to the encoding given in the Table 1, the digit of weight $8^{(n-2)}$ identifies the second largest octant and so forth. The numeric value of a cubic voxel Q is described by the expression

$$Q = q_{n-1}8^{n-1} + q_{n-2}8^{n-2} + ... + q_0 8^0$$

where q_l, $l = 0, 1, 2, ..., n-1$ is one of the octal digits { 0, 1, 2, 3, 4, 5, 6, 7}. For instance, the "invisible" voxel (2, 2, 2) of Figure 2 belongs to the "forward-north-east" octant in the first subdivision,. Thus $q_{n-1} = q_1 = 0$. It belongs to the "backward-south-west" octant in the second subdivision. Thus $q_{n-2} = q_0 = 7$. The encoding for the invisible voxel is, therefore, 07. Similarly, we have, for the region shaded in Figure 2, the representation

{10, 01, 00, 03, 02, 05, 04, 07, 06, 24, 40}

It is possible to condense the data further. If there are eight voxels belonging to the same octant, they can be grouped together by replacing the digit relative to the common octant by a marker. If we denote marker by X then X should be greater that 7. It will help to keep the data structure remain sorted. So the final representation (linear octree encoding) of the 3D object of Figure 2 is

{0X, 10, 24, 40}

The representation structure here is a simple one dimensional array whose length is determined at run time consisting of sorted mixed-octal[1] codes. Every element in the digit of the octal code (left to right) gives the path of leaf node from the root. There are many algorithms based on the linear octrees for encoding[6], decoding, neighborhood finding, etc.

2.2 Problem Definition

The input to our algorithm is a linear octree based representation of the data (for example the list {0X,

[1]It is referred as mixed-octal because a special character, here X, is mixed with octal numbers

10, 24, 40} in the case of Figure 2), and a view plane normal represented by a vector (not necessarily a unit vector).

Our algorithm produces as output a list of (equal sized) voxels which may be correctly rendered by a front-to-back algorithm, or the painter's algorithm. (In order to verify the correctness of our algorithm along with variants, we implemented the actual display as seen in Figure 6.)

3 The Algorithm

Many approaches ([5], [1], [4]) can be taken for visualization of octree encoded objects. Unlike these older algorithms, the main emphasis of our algorithm is a scenario where we would like to vary the view plane normal continuously, and see the resulting picture. For any arbitrary view, prior algorithms *require* reconstructing the octree for the given viewing coordinate system. This makes these algorithms compute intensive for an arbitrary chosen view.

Instead of recomputing the octree data based on the view plane normal, we develop a method to achieve front-to-back display of linear octree.

3.1 Using sorting for hidden surface removal

The octree encoding provides position of the octant in 3D world precisely and stored in a certain order. Voxels can be sorted according to their distance from a view point. Traversing this sorted list of octants and projecting their surfaces gives us a front-to-back approach that can be rendered using standard algorithms in graphics.

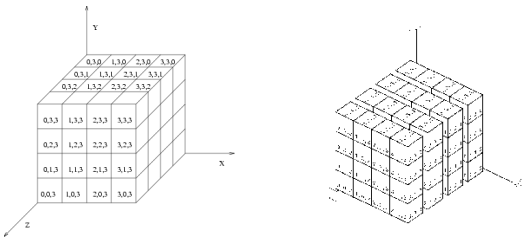
To motivate the algorithm we use three representations of an octree:

(a) Three tuple indices (i, j, k), which are positional indices of the voxels as in Figure 2(a). This scheme provides no data compression, and would not be a suitable representation in real life scenarios.

(b) Linear octree-like encoding where all the octants are of equal size. This scheme provides data compression, but since it does not use mixed-octal, one could do better.

(c) The linear octree encoding as discussed in Section 2.1.

The algorithm is easy to understand if we use the octree representation in three tuple form. We use the idea for the linear octree encoding representation.



(a) Object space is divided into voxels.

(b) Slices of voxels parallel to the X-Y plane.

Figure 3: Using the three tuple representation for 3D data.

3.2 Voxel with three tuples

The idea of the front-to-back approach for octree encoded objects can be explained easily if (equal sized) voxels are used as the basis for representation. A 3-dimensional array of flags is maintained for all voxels in the object space; a flag is set for voxels which are part of the object.

Figure 3(a) shows the subdivision of the entire space with resolution 2. In this case the total number of voxels is 64. In the three dimensional array of flags, array subscripts varies from 0 to 3 in all dimensions.

We refer to the the three planes X-Y, X-Z, and Y-Z as *principal*. By *slices* we mean the set of planes (parallel to a principal plane) that partition voxels. Figure 3(b) shows an example, where each *slice* corresponds to sixteen voxels.

Our algorithm relies first on the selection of a principal plane, and then within slices, rows and columns. The selection of the plane is dependent on the view direction. Intuitively, the first plane is selected in such a way that all the voxels on one slice in the plane are nearer to the observer compared to the voxels on other slices in the same plane. Within each slice, voxels are visited either in row-wise or column-wise in an order (increasing or decreasing) that is also decided based on the view direction.

3.2.1 A specific view direction

Let us consider an object viewed from a point which is located at (x', y', z') such that x', y' and z' are positive integers and $z' > y' > x'$. If the view direction is along the vector joining the view point and origin, i.e., the view plane normal is given by vector (x', y', z'), then the algorithm for traversing voxels in front-to-back order is given in Figure 4.

Procedure **Front_To_Back_Traversal**
  for (each slice S on X-Y plane, z varies from 3 to 0)
    for (each row R on slice S, y varies from 3 to 0)
      for (on row R, x varies from 3 to 0)
        if voxel(x, y, z) is part of the object
          add voxel(x, y, z) to list to be rendered

Figure 4: Algorithm for front-to-back traversal of voxels.

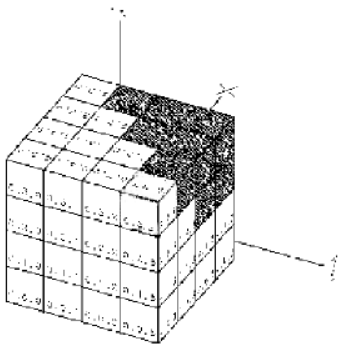


Figure 5: Using the algorithm to render from the viewpoint $(-2.0, 1.5, 1.0)$.

3.2.2 An example

Consider the front-to-back traversal for the object, given in Figure 2(a), from a view point $(1.0, 1.1, 1.2)$.

Flags of the following voxels are set in the input to the algorithm.
$(1,3,3)$, $(2,2,2)$, $(2,2,3)$, $(2,3,2)$,
$(2,3,3)$, $(3,1,2)$, $(3,2,2)$, $(3,2,3)$,
$(3,3,1)$, $(3,3,2)$, $(3,3,3)$

For the given view point slices are made along the X-Y plane as per the algorithm in Figure 4. The nearest slice to the observer is all the voxels which are on the plane $z = 3$. The voxels belonging to this slice are $(1,3,3), (2,3,3), (3,3,3), (2,2,3), (3,2,3)$. The second slice is for all the voxels on the plane $z = 2$ and the voxels are $(2,3,2), (3,3,2), (2,2,2), (3,2,2), (3,1,2)$. And the farthest slice is on the plane $z = 1$ and the voxels on this slice are $(3,3,1)$.

Within each slice we traverse (in this case) the voxels in row major order, i.e., all the voxels for $y = 3$ are visited first and then the voxels for $y = 2$ and so on. Thus, the list of voxels in sorted order to be rendered is $(3,3,3), (2,3,3), (1,3,3), (3,2,3), (2,2,3), (3,3,2), (2,3,2), (3,2,2), (2,2,2), (3,1,2), (3,3,1)$.

From Figure 2(a) it is clear that the above order of voxels is in non-decreasing order of distance from the viewer.

3.2.3 An alternate view direction

For the object in Figure 2(a) and the view point $(-x', y', z')$ where $x' > y' > z'$, we get a picture similar to Figure 5.

As mentioned earlier, the selection of the plane for slices, row and column axis and order within each slice, depends on the view. For this example, the traversal of voxels should be done in the following order

- Outer loop on slices (Y-Z plane, increasing x from 0 to 3).
- Within each slice, an outer loop on the rows (Y-direction, decreasing order).
- Innermost loop in the Z-direction (decreasing order).

3.2.4 The main idea

We observe that for a given view plane normal vector (i, j, k), the index along the x-axis should be increasing if i is $-ve$ and should be decreasing if i is $+ve$. The order of the axis in the three loops will depend on the absolute value of i, j, and k of the view plane normal vector. The smallest value amongst i, j, and k will decide if the innermost loop is along x, y, or z axis.

This idea is unfortunately difficult to implement if the input data is *not* given in the (i, j, k) three tuple form. This is the case for our linear octree.

3.3 A new encoding mechanism

In Section 3.2 we have used an explicit representation of octants of the same size. Here we will use an encoding mechanism of the voxels which will be similar to that given in Section 2.1. Using this encoding mechanism all the voxels of Figure 3(a) will be coded as given in Figure 2(b). We store codes of only those voxels which belong to the objects. These encoded voxels, which consist of octal digits, are kept in sorted order (as required by many GIS packages [11], [3]). This encoding mechanism is very similar to the one used in linear octree, except that here we divide all the octants to the smallest level, even though, the volume covered by an octant is homogeneous. In other words, we do not use the mixed-octal values.

3.3.1 An example

Consider the data in Figure 2(b). If the viewpoint is given as, say, $(1.0, 1.1, 1.2)$, the order of traversal of voxels we desire for the nearest slice is

$00, 01, 10, 11, 02, 03, 12, 13, 20, 21, 30, 31, 22, 23, 32, 33$ because slices are made along the X-Y plane. The original octree encoding representation is $00, 01, 02, 03, 04, 05, 06, 07, 10, 11, 12 \cdots$.

We achieve the right order by processing the input sequence such that we produce a new code for each voxel with the following important property: The new code when traversed in sorted order produces the voxels in front-to-back order.

For the example, we want the mapping

00 to 00	01 to 01
10 to 02	11 to 03
02 to 04	03 to 05
12 to 06	13 to 07

and so on.

For the example, we recompute the new value as follows. If a voxel is encoded as Q and $q_l, l = 0, 1, 2, \cdots$ are the octal digits in Q, then $Q = q_{n-1}8^{n-1} + q_{n-2}8^{n-2} + \cdots + q_0$.

If each octal number q_l is represented in binary digits L, M, and R such that $q_l = L_l M_l R_l$ then Q is represented in binary form as $Q = L_{n-1}M_{n-1}R_{n-1}L_{n-1}M_{n-1}R_{n-1}\cdots L_0 M_0 R_0$.

We assign new code Q' by the equation $Q' = L_{n-1}L_{n-2}\cdots L_0 M_{n-1}M_{n-2}\cdots M_0 R_{n-1}R_{n-2}\cdots R_0$.

The new octal code Q' produced using the above encoding mechanism, when sorted, will produce the voxels in the order which is the same as voxels when traversed using the following three loops:

- Outer loop on slices (Z-direction, decreasing z from 3 to 0).
- Within each slice, an outer loop on the rows (Y-direction, decreasing order).
- Innermost loop in the X-direction (decreasing order).

This procedure is illustrated with the index values (x, y, z), octal encoding (Q) and the new code (Q') for the example in Table 2:

In Table 2 voxels are arranged in the sorted order as they appear in the octree encoding. If we sort the voxels based on Q' then we get the voxels $(3,3,3), (2,3,3), (1,3,3), (3,2,3), (2,2,3), (3,3,2),$ $(2,3,2), (3,2,2), (2,2,2), (3,1,2), (3,3,1)$.

It may be noted that this order of the voxels is the same, as that in Section 3.2.

3.3.2 A second example

For the second case when the view plane normal (i, j, k) is such that $\mid i \mid > \mid j \mid > \mid k \mid$ and i is negative, the new code Q' is computed as $Q' =$

(x, y, z)	Q	Q' $(1.0, 1.1, 1.2)$	Q' $(-2.0, 1.5, 1.0)$
3 3 3	00	00	60
2 3 3	01	01	40
3 2 3	02	04	64
2 2 3	03	05	44
3 3 2	04	20	61
2 3 2	05	21	41
3 2 2	06	24	65
2 2 2	07	25	45
1 3 3	10	02	20
3 1 2	24	30	71
3 3 1	40	40	62

Table 2: Encoding voxels for view plane normals $(1.0, 1.1, 1.2)$ and $(-2.0, 1.5, 1.0)$.

$\overline{R^{n-1}R^{n-2}\cdots R^0}M^{n-1}M^{n-2}\cdots M^0 L^{n-1}L^{n-2}\cdots L^0$ and is also shown in Table 2.

If we sort the voxels according to Q' we get the correct order $(1,3,3), (2,3,3), (2,3,2), (2,2,3), (2,2,2), (3,3,3)$ $(3,3,2), (3,3,1), (3,2,3), (3,2,2), (3,1,2)$.

3.3.3 The general case

For all view plane normals suitable equations for Q' has been derived formally [8]. We present them in the appendix. From a computer implementation, all these forms involve simple bit manipulations.

3.4 Front-to-back traversal of linear octree

The encoding mechanism we have used in Section 3.3 is similar to the one used in linear octree. In a linear octree whenever an octant contains a homogeneous part of the object, it is not further divided into sub-octants. While encoding such an octant a special character X is used in the code. For example the object in Figure 2 is encoded as $(0X, 10, 24, 40)$.

In our algorithm, an octant with X in its code is handled by breaking the octant into (smaller sized) voxels and a new code is generated for all the voxels of the octant. In other words, the algorithm degenerates to the algorithm mentioned in Section 3.3.

Breaking of the large octants is *necessary*. Given a large octant, we can always find smaller octants in the object such that the smaller octants need to be visited after some part of the larger octant is displayed and before the rest of the larger octant is displayed. This becomes clear from Table 2; octant 0X is broken into voxels $01, 02, 03, 04, 05, 06,$ and 07; parts of this octant (e.g., the voxel $(3,3,3)$ (or 00 as per Q) come

after the octant $(2,2,2)$ whereas other parts (e.g., the voxel $(2,3,3)$ (or 01 as per Q) comes before the octant $(2,2,2)$ in the front-to-back traversal.

In practice, we have observed that breaking large octants is not a major overhead because the rendering process is simplified as all the voxels are of equal size.

As an example for the linear encoding $(0X, 10, 24, 40)$, the new codes generated given the view direction $(-2.0, 1.5, 1.0)$ is $(60, 40, 64, 44, 61, 41, 65, 45, 20, 62, 71)$. This list is sorted and passed to the front-to-back rendering phase.

3.5 Rendering recoded voxels

VPN	Visible face
$x > 0$	eastern
$x < 0$	western
$y > 0$	southern
$y < 0$	northern
$z > 0$	front
$z < 0$	back

Table 3: Visibility test of faces depending on VPN (x, y, z).

The recoded voxels are converted into polygons representing the faces. Depending upon the VPN, the sides to be drawn are decided as in Table 3. At most three sides of the voxel are drawn for any VPN.

4 Performance Analysis

We have analyzed our algorithm both in a theoretical sense, and by explicitly coding our algorithm. The results are summarized in this section.

4.1 Big-Oh complexity

It is clear that we process and sort all the nodes in the linear octree after generating the new code. Generating the new code involves bit manipulation, and the length of each code is $O(n)$ bits; here n is the level of subdivision in the linear octree.

If the number of nodes in the linear octree is N after converting all the octants to (equal size) voxels (this step is necessary), then the complexity of the algorithm is $O(N \log N)$ when we use a standard sorting algorithm such as heap sort.

If we use the radix sort algorithm, then, using 8 buckets, we can sort the octants in $O(Nn)$ time. Since $N = \Theta(8^n)$, $n = O(\log_8 N)$ the complexity is $O(N \log_8 N)$.

The space requirement for the algorithm is linear. An array is maintained for keeping the octal codes of all voxels of the octree. The same array is used to store the recoded value of the voxels. Given the view plane normal, indices for any voxel can be computed using bit manipulations.

4.2 Empirical results: Comparison with OpenGL

We compared our algorithm to a straightforward algorithm that does not use our recoding technique. The idea [2] here is to send the faces of the list of unsorted voxels to OpenGL's z-buffer rendering engine which performs hidden surface removal. To make the OpenGL algorithm run *faster*, we **enabled** the back-face culling test so that unnecessary polygons do not get considered.

Our algorithm was ported on three different implementations of OpenGL: The SGI Indy machine, a SUN Solaris Ultrasparc machine, and a Pentium III Windows NT machine. A comparative study of our algorithm and the back face culling method of OpenGL was done with various parameters on data consisting of about 4000 voxels to 73,000 voxels. Table 4 summarize the results for one specific case, with three different view directions. Figure 7 and Figure 8 shows the situation on different platforms. The objects on which the algorithm was performed was a cube, and the object shown in Figure 6(d).

We conclude that our algorithm renders objects faster than the back face culling mechanism of OpenGL (despite possible hardware optimizations done on the SGI platform).

5 Final Remarks

Visible surface computation of a part of a scene is an important step because all other steps (such as clipping, transformations, perspective normalization) in the rendering pipeline take time independent of other parts of the scene. In this paper, we have given a new algorithm to render a linear octree that allows the dynamic change of the view plane normal. We have compared our algorithm to a variant loosely based on prior work. Our visible surface algorithm runs in $O(N \log N)$ time.

Just as the selection of representation of 3D data is open, so is the selection of the display method given any one representation. The ten different hidden surface algorithms [15], and the subsequent variants are well known. We have chosen to use the linear octree representation rampant in 3D GIS, and the front-to-back traversal display paradigm. A related question is therefore "What if you use display method X in rep-

[2] This algorithm is a variant of the algorithm in [5] which converts the input list of voxels to the three tuple representation (i,j,k) and then performs hidden surface removal.

resentation Y?" It would be beyond the scope of this paper to address this issue in its full generality.

References

[1] Louis J. Doctor and John G. Torborg. Display techniques for octree-encoded objects. *IEEE Computer Graphics and Applications*, 1(3):29–38, July 1981.

[2] S.P. Dunstan and A.J.B.Mill. Spatial indexing of geological models using linear octree. *Computer and Geossciences*, 15(8):1291–1301, 1989.

[3] F.Major, J. Malenfant, and N.F. Stewart. Distance between objects represented by octree defined in different coordinate systems. *Computer and Graphics*, 13(4):497–503, July/Sept 1989.

[4] Gideon Frieder, Dan Gordon, and R. Anthony Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Application*, 5(1):52–60, January 1985.

[5] I. Garganitini, G. Schrack, and H.H. Atkinson. Adaptive display of linear octrees. *Computer and Graphics*, 13(3):337–343, July/Sept 1989.

[6] Irene Gargantini. Linear octrees for fast processing of three-dimensional objects. *Computer graphics and Image processing*, 20:365–374, 1982.

[7] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[8] Ajay Kumar Gupta. Masters Thesis. Master's thesis, Indian Institute of Technology, December 1995.

[9] Jonathan I. Raper. *Three dimensional applications in Geographic Information System*. Burgess Science Press, 1989.

[10] Djaffer Ibaroudene and Raj Acharya. Parallel display of objects represented by linear octrees. *IEEE Transactions on Parallel and Distributed Systems*, 6(1), January 1995.

[11] Toliyasu L. Kunil, Toshiaki Satoh, and Kazunari Yamaguchi. Generation of topological boundary representations from octree encoding. *IEEE Computer Graphics and Application*, 5(3):29–38, March 1985.

[12] Jackie Neider, Tom Davis, and Mason Woo. *The Official Guide to Learning OpenGL*. Addison-Wesley Publishing Company, 1994.

Data voxels	VPN=(1, 0, 0)		VPN=(1, 1, 0)		VPN=(1, 1, 1)	
	bfc μ sec	f2b μ sec	bfc μ sec	f2b μ sec	bfc μ sec	f2b μ sec
4k	90	40	90	50	100	60
24k	480	190	490	270	500	340
32k	650	260	660	360	710	460
50k	980	390	1000	530	1070	690
64k	1310	510	1310	710	1360	960
73k	1370	580	1490	800	1580	1020

Table 4: Comparison of time required to render the voxels using OpenGL's back face culling (bfc) and our front to back recoding (f2b) on a Pentium III with Windows NT.

[13] Ananth Potty. Efficient ray casting. Master's thesis, Indian Institute of Technology, December 1997.

[14] Hanan Samet. Neighbor finding in images represented by octrees. *Computer Vision, Graphics and Image Processing*, 46:367–386, 1989.

[15] I. Sutherland, R. Sproull, and R. Schumacker. A characterization of ten hidden surface algorithms. *Computing Surveys*, 6:1–55, 1974.

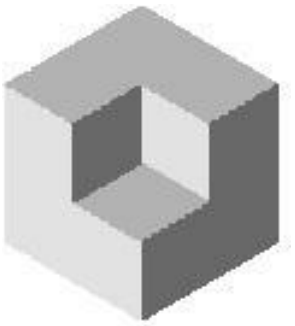
Appendix A

If a voxel is encoded as Q and $q_l, l = 0, 1, 2, \cdots$ are the octal digits in Q, then $Q = q_{n-1}8^{n-1} + q_{n-2}8^{n-2} + \cdots + q_0$. If each octal number q_l is represented in binary digits L, M, and R such that $q_l = L_l M_l R_l$ then Q is represented, in binary form, as: $Q = L_{n-1}M_{n-1}R_{n-1}L_{n-2}M_{n-2}R_{n-2}\cdots L_0 M_0 R_0$. Given the VPN (i, j, k) we first compute three quantities A, B, and C, whose values depends on the signs of i, j and k using Table 6.

The value Q' depends on relative values of $\mid i \mid, \mid j \mid$ and $\mid k \mid$. In binary notation Q' is as in Table 5.

Relative values	Q'
$\mid i \mid < \mid j \mid < \mid k \mid$	Q' = C B A
$\mid j \mid < \mid i \mid < \mid k \mid$	Q' = B A C
$\mid k \mid < \mid i \mid < \mid j \mid$	Q' = A C B
$\mid i \mid < \mid k \mid < \mid j \mid$	Q' = C A B
$\mid j \mid < \mid k \mid < \mid i \mid$	Q' = B C A
$\mid k \mid < \mid j \mid < \mid i \mid$	Q' = A B C

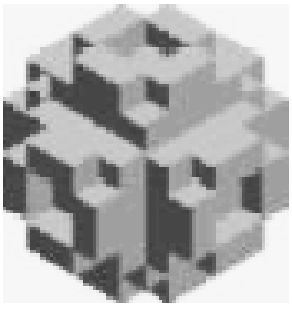
Table 5: Bit manipulation for sorting.



(a) VPN=(1,-0.9,0.5)

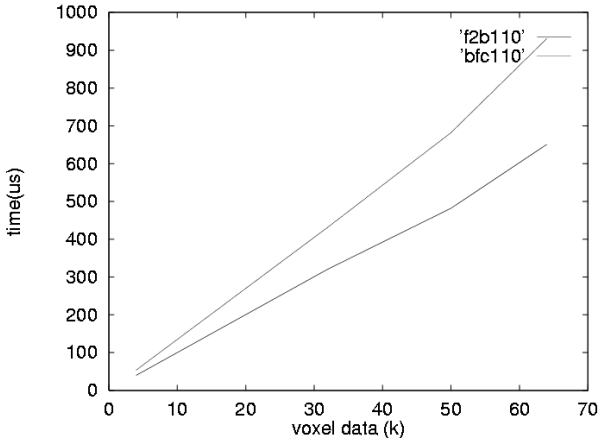
(b) VPN=(1,1.1,1.2)

(c) VPN=(1,1,1)



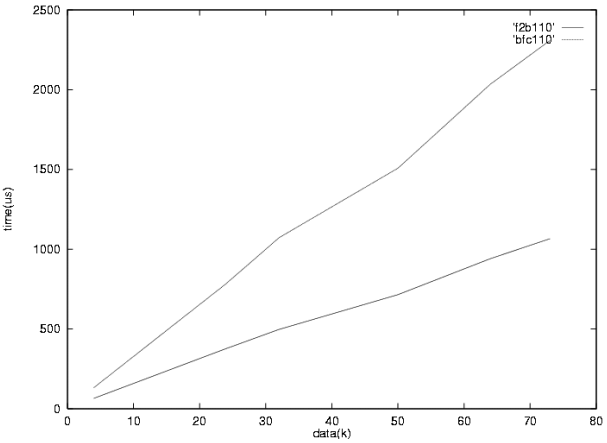
(d) VPN=(2,1.0,0.9)

Figure 6: Sample rendered shapes.



(a) VPN = (1, 1, 0)

Figure 7: Comparison of time required to render the voxels using OpenGL's back face culling(bfc) and our front to back recoding(f2b) on an SGI Indy machine.



(a) VPN = (1, 1, 0)

Figure 8: Comparison of time required to render the voxels using OpenGL's back face culling(bfc) and our front to back recoding(f2b) on a Sun Ultrasparc machine.

(i,j,k)	A	B	C
$(+,+,+)$	$\overline{L_{n-1}L_{n-2}\cdots L_0}$	$M_{n-1}M_{n-2}\cdots M_0$	$R_{n-1}R_{n-2}\cdots R_0$
$(+,+,-)$	$L_{n-1}L_{n-2}\cdots L_0$	$M_{n-1}M_{n-2}\cdots M_0$	$R_{n-1}R_{n-2}\cdots R_0$
$(+,-,+)$	$\overline{L_{n-1}L_{n-2}\cdots L_0}$	$\overline{M_{n-1}M_{n-2}\cdots M_0}$	$R_{n-1}R_{n-2}\cdots R_0$
$(+,-,-)$	$L_{n-1}L_{n-2}\cdots L_0$	$M_{n-1}M_{n-2}\cdots M_0$	$\overline{R_{n-1}R_{n-2}\cdots R_0}$
$(-,+,+)$	$\overline{L_{n-1}L_{n-2}\cdots L_0}$	$M_{n-1}M_{n-2}\cdots M_0$	$\overline{R_{n-1}R_{n-2}\cdots R_0}$
$(-,+,-)$	$L_{n-1}L_{n-2}\cdots L_0$	$\overline{M_{n-1}M_{n-2}\cdots M_0}$	$\overline{R_{n-1}R_{n-2}\cdots R_0}$
$(-,-,+)$	$\overline{L_{n-1}L_{n-2}\cdots L_0}$	$\overline{M_{n-1}M_{n-2}\cdots M_0}$	$R_{n-1}R_{n-2}\cdots R_0$
$(-,-,-)$	$\overline{L_{n-1}L_{n-2}\cdots L_0}$	$\overline{M_{n-1}M_{n-2}\cdots M_0}$	$\overline{R_{n-1}R_{n-2}\cdots R_0}$

Table 6: Bit manipulation for sorting.