National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

University of Alberta

Fast Display of 3D Voxel-Based Objects via Octree Representations

by

Jiangong Zhao

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring 1991

National Library
of Canada

Canadian Theses Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale
du Canada

Service des thèses canadiennes

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-66709-5

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Jiangong Zhao
TITLE OF THESIS: Fast Display of 3D Voxel-Based Objects via Octree Representations

DEGREE: Master of Science
YEAR THIS DEGREE GRANTED: 1991

Permission is hereby granted to the UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) Zhao Jiangong

Permanent Address:
#1005, 11025-82 Ave,
Edmonton, Alberta,
Canada, T6G 0T1

Date: Jan 17, 1991

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Fast Display of 3D Voxel-Based Objects via Octree Representations** submitted by **Jian-gong Zhao** in partial fulfillment of the requirements for the degree of Master of Science.

Dr. W.A. Davis . . . Wayne A. Davis . . .
(Supervisor)

Dr. B. Joe . . . .

Dr. X. Li . . . Xiaobo Li . . .

Dr. Z. J. Koles . . . Z J Koles . . .

. . . .

. . . .

Date: Jan 16, 1991

To my wife Angela Q. Ruan and my parents.

Abstract

In this thesis, existing 3D display techniques are reviewed in depth and a new display technique is proposed and implemented. The new technique displays 3D voxel-based binary objects encoded as octrees. It traverses an octree recursively in front-to-back (FTB) order determined by a given viewing direction and produces a depth image by scan-converting the visible portion of the faces of black octants encountered during the traversal. The algorithm is very efficient because many obscured nodes are never visited and many obscured faces of partially visible black octants do not join the scan-conversion phase. The key to the algorithm is a new concept termed *blocking quadtrees*, which enables the node visibility test to be performed very efficiently and thus minimizes the overhead cost.

Complexity analysis and experimental results show that the new algorithm is much faster than the traditional back-to-front (BTF) approach (55-79% time reduction achieved for a 3D $256 \times 256 \times 256$ medical image) and requires a reasonable amount of extra memory space for storing the blocking quadtrees (64 KB for a $256 \times 256 \times 256$ image).

Acknowledgements

With sincere gratitude, I wish to thank my supervisor Dr. Wayne A. Davis for his invaluable guidance and encouragement throughout the research.

I am also grateful to the members of my examining committee, Dr. X. Li, Dr. B. Joe and Dr. Z. J. Koles, for their helpful comments.

I would like to thank the Department of Nuclear Medicine, Cross Cancer Institute, Edmonton, Alberta for providing image data for experimentation.

I would also like to thank Steven F. Sutphen for helping me prepare the photographs used in the thesis.

Finally, I am deeply indebted to my wife, Angela Q. Ruan. Without her love, understanding and encouragement, I would never have finished this thesis.

Contents

1 Introduction 1

2 Display of Non-Octree Encoded Objects 3

2.1 Surface Rendering for Objects Represented as 3D Arrays of Voxels . 5

2.1.1 Slice-by-Slice Boundary Extraction 5

2.1.2 Triangle Tiling 6

2.1.3 Marching Cubes 8

2.1.4 Surface Detection via Gradient Operators 10

2.1.5 Surface Detection via Graph-Theoretic Model 11

2.1.6 Surface Detection in Multidimensions 13

2.2 Volume Rendering for Objects Represented as 3D Arrays of Voxels . 15

2.2.1 Slice-by-Slice Back-to-Front Approach 15

2.2.2 Volume Rendering Based on Ray-Tracing 17

2.3 Volume Rendering via Line Segments 18

2.4 Gradient Shading 21

3 Display of Octree Encoded Objects 22

3.1 Octree Representation 23

3.2 Boundary Detection via Linear Octrees 24
    3.2.1 The First Algorithm 26
    3.2.2 The Improved Algorithm 27
3.3 Rendering of Octrees 29
    3.3.1 BTF Display of Octrees 30
    3.3.2 BTF Display of Linear Octrees 31
    3.3.3 FTB Display of Octrees with Arbitrary Viewing Directions . 32
    3.3.4 FTB Display of Octrees with Restricted Viewing Directions 34

4 Fast Display of Octree Encoded Objects **38**
4.1 3D Coordinate Transformation and FTB Visiting Order 39
4.2 Blocking Quadtrees 42
    4.2.1 Blocked Faces 42
    4.2.2 FTB Traversal with Blocking Quadtrees 44
    4.2.3 Blocking Bitmaps 49
4.3 Algorithm 52
    4.3.1 The Algorithm 52
    4.3.2 Further Explanations 54
4.4 Complexity Analysis 59
4.5 Experimental Results 62

5 Conclusion **67**

List of Figures

1 Triangle tiling. 7

2 Marching cube. 8

3 Slice-by-slice BTF read out sequence. 16

4 Selection of object and image space axes. 20

5 Octant labeling (octant [illegible] is hidden from view). 24

6 A simple object and its octree representation. 25

7 Octant projection and its overlay. 33

8 +X-face view. 35

9 Edge-XZ view. 36

10 Subdivision of an edge view. 36

11 Corner-7 view. 37

12 Grouping of triangles. 37

13 Mapping between VPNs and nearest octants. 41

14 Six faces of an octant. 42

15 An octant may have (a) 3, (b) 2, or (c) 1 potentially visible faces. . . 43

16 Six neighboring octants of an octant. 43

17 Octant labeling of octree space and quadrant labeling of blocking quadtree spaces. 46
18 Status of blocking quadtrees for displaying the object in Figure 6. . 47
19 Status (last stage) of blocking quadtrees for displaying the object in Figure 6 (assuming octant c is W). 48
20 Status (last stage) of blocking quadtrees for displaying the object in Figure 6 (assuming octant c is G). 48
21 A 4 by 4 quadtree space and its complete quadtree. 50
22 A complete quadtree and its blocking bitmap. 50
23 Cardinal numbers of a complete quadtree. 55
24 Definition of ind_tab (n = 8). 56
25 Definition of quad_tab. 57
26 Determination of axis and axis_nums. 60
27 Results for (a) H, and (b) Human skull. 64
28 Theoretical results for the human skull. 65
29 Depth shaded images. 66

Chapter 1

Introduction

Slow rendering speed is a major problem of existing software techniques for displaying 3D voxel-based objects on a 2D screen. Typically, minutes or tens of minutes are required to render a 3D scene consisting of 256 × 256 × 256 voxels via a software method [28, 16]. As pointed out by Goldwasser et. al. [17], this is not satisfactory for most applications. The problem becomes even more serious for applications involving interactive manipulation of objects via graphics input devices such as mice, trackballs or joysticks because such applications (for example, surgical planning [4, 7, 36]) require user feedback to be as fast as possible, preferably in real time.

In order to achieve a fast rendering speed, one approach is to design special computer architecture for displaying 3D voxel-based objects [27, 18, 20]. But this is more expensive and less flexible than software methods. In this thesis hardware methods are not discussed.

The objective of this thesis is to review existing software techniques for displaying 3D voxel-based objects and investigate possibilities of improving their rendering speed. Specifically, a new method is proposed for displaying 3D voxel-based objects via *octree* representations. Both theoretical and experimental results show that the new method is much faster than a conventional octree display algorithm and requires a reasonable amount of extra memory space.

The organization of this thesis is as follows. Chapter 2 reviews the major techniques for displaying non-octree encoded objects. Chapter 3 reviews the major techniques for displaying octree encoded objects. Chapter 4 presents the proposed method for displaying octree encoded objects and its associated theoretical and experimental results. Chapter 5 concludes the thesis.

Chapter 2

Display of Non-Octree Encoded Objects

In general, the following steps are necessary to display 3D voxel-based objects on a 2D screen: (1) Object identification, (2) Object representation, (3) Extraction of surfaces, and (4) Display of surfaces.

Automatic object identification is a complex 3D pattern recognition task because real data, for example, medical images, often have complex characteristics. The simplest and most commonly used method for object identification is by thresholding (or windowing) of the input 3D voxel values [17]. This method classifies voxels with values falling in some predetermined range(s) as objects and others as background. In certain situations, however, objects of interest cannot be separated only by their voxel values [33]. To solve this problem interactive segmentation methods based on positions can be used, which in general are time consuming since the segmentation is usually done in a slice-by-slice fashion [33].

After the objects of interest have been identified, a data structure is needed to represent them. In this thesis, the following 3D representations are used: *3D array of voxels*, *octrees*, and *line segments*.

Once the object representation is chosen, steps 3 and 4 can be realized by either *surface rendering* or *volume rendering*. In surface rendering, the surfaces of objects of interest in a 3D scene are first extracted, as in step 3, and then displayed via standard polygon rendering techniques for computer graphics [11], as in step 4. In volume rendering, steps 3 and 4 are combined and no surface is explicitly extracted. A 3D scene is projected onto a 2D screen as a whole. Care is taken to ensure that voxels obscured from the viewpoint are not projected onto the screen. Therefore, the visible part of object surfaces is automatically displayed.

Volume rendering is more suitable for interactive display of interior surfaces of objects than surface rendering. This is because all voxels in a 3D scene can be accessed at display time and the objects can be edited or modified, and then re-displayed without the additional time penalty to form new surfaces. Surface rendering, on the other hand, may require less memory space and have a faster rendering speed in non-interactive applications because the number of surface elements is in general much smaller than that of all voxels in a 3D scene.

In this chapter, the major rendering techniques based on 3D arrays of voxels and line segments are reviewed. Since octrees are of special interest in this thesis, the major rendering techniques based on octree representations will be reviewed in the next chapter.

2.1 Surface Rendering for Objects Represented as 3D Arrays of Voxels

The most important step in surface rendering is step 3 or surface extraction. Three major approaches have been published in the literature to extract surface information from objects represented as 3D arrays of voxels. The first approach [37] extracts boundaries of objects on a slice-by-slice basis by 2D edge following techniques and then displays the object surfaces as a wire-frame picture consisting of a stack of boundaries. The second approach [21, 13, 14, 9, 35, 5, 23] extracts object surfaces as triangular patches. And the third approach [22, 1, 34] operates on a 3D scene directly and detects object surfaces that consist of voxel faces. These methods will be discussed briefly in this section.

After surfaces are formed, they can be displayed as shaded images by either conventional techniques [11] or special methods that take advantage of a special surface format [6]. A detailed survey of this topic is given in [32] and will not be repeated in this thesis.

2.1.1 Slice-by-Slice Boundary Extraction

This approach [37] is based on the observation that the intersections of the surfaces of an object with the kth slice of a 3D scene are the borders of the object on the kth slice and hence the object surface can be represented as a stack of borders.

A thresholding process is executed on the input gray-scale data to produce a binary scene, assigning the value 1 to the object and 0 to the background. Then a stack of borders of the object is determined by applying a border-following algorithm to each slice of the binary scene. The stack of borders may be rotated in 3D space to bring the object to any desired orientation. To display the object on a 2D screen, a projection of the rotated stack onto the screen needs to be computed. This may be done with or without hidden-line elimination. The procedure yields a wire-frame display.

Representing surfaces as a set of borders cannot reflect all the shape subtleties. For instance, the surface details between a pair of boundary lines are totally unobtainable. Since a faithful rendition of surface details is very important in many applications [17], this approach seems only appropriate to obtain a quick preview of objects.

2.1.2 Triangle Tiling

The idea of these methods is to tile between a pair of adjacent borders on two adjacent slices with triangles to approximate the surfaces between the borders. By carrying out the tiling process in a slice-by-slice fashion, the whole surface of the object of interest can be represented by the triangles. As in the approach described in Section 2.1.1, a stack of borders needs to be produced from a binary scene in a slice-by-slice manner. A resampling step is usually performed on the borders to get fewer points and a smoother curve, and the resulting borders are referred to as contours.

A triangle is formed by using a pair of points on one contour and the third point on the other contour (Figure 1). The triangulation of adjacent contours is based on the assumption that matching of contours from slice to slice can keep track of the objects. Formally, the triangulation problem can be stated as: Given

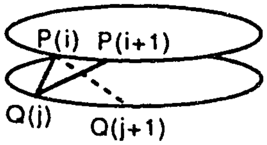


Figure 1: Triangle tiling.

a sequence of points P(i), i = 1, ..., m, representing the contour on the upper slice and another sequence Q(j), j = 1, ..., n, representing the contour on the lower slice (see Figure 1), produce a set of triangle patches forming an acceptable surface.

Various criteria have been proposed to characterize "acceptable surface". Basically, they can be classified as "optimization criteria" and "heuristic criteria". Methods based on optimization criteria [21, 13, 14] select an "optimal" triangle patch arrangement from many possible alternatives in a systematic way. Essentially, the problem is transformed into a graph-theoretic one: Establishing a directed graph corresponding to all possible patch arrangements, associating a weight to each edge of the graph, and finding a path in the graph that meets some "optimal" criterion. Complications usually arise in these methods when the upper and lower contours yield concave and convex segments which do not match properly. On the other hand, methods based on "heuristic criteria" [9, 35] use user interaction to guide the tiling of contours that do not match properly.

A recent triangle tiling technique [5] employs the concept of Delaunay triangulations. The 3D Delaunay triangulation of points lying on the contours of two slices is first computed by using only 2D operations. By pruning a series of such

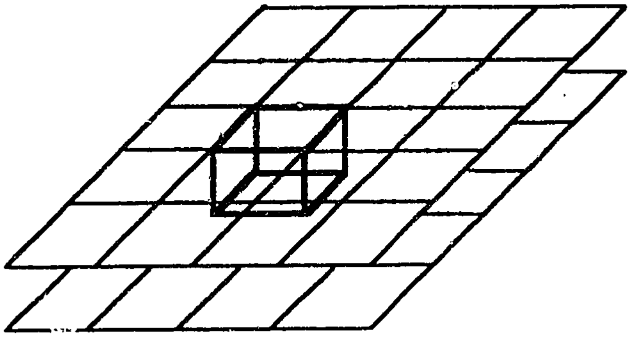


Figure 2: Marching cube.

Delaunay triangulations, the algorithm obtains a volume whose boundary is a polyhedron with triangle faces. Such a volume is an approximation of the original objects. This technique can handle the cases where the number of contours varies from one slice to the other.

2.1.3 Marching Cubes

This method [23] can also be considered as a triangle tiling method in the sense that it forms surfaces between two adjacent slices by using triangle approximations. However, it differs from the previous triangle tiling methods in that no contours on the slices need to be formed before the tiling process. It operates on the slice data directly, and uses a logical cube created from eight voxels of two adjacent slices as the basic data structure (Figure 2), processing the data cube after cube (marching cubes).

The surface to be formed in this method corresponds to a user specified voxel density value. A cube's vertex is assigned "1" if the density value at that vertex exceeds (or equals) the surface value, and "0" otherwise. 1-vertices are considered to be "inside" or "on" the surface, while 0-vertices are "outside" the surface. It is observed that the surface intersects the edges of a cube if some vertices of the cube are outside the surface and others are inside or on the surface. By forming triangles using those intersection points, the surface within the cube can be approximated.

Creating triangles within a cube is easy. Since there are only $2^8 = 256$ different ways a surface can intersect a cube (eight vertices in a cube, each may have two states: 0 or 1), a look-up table can be pre-computed, which, indexed by the state of eight vertices (one byte, one bit for each vertex), contains the edges intersected for each possible vertex pattern. Furthermore it is observed that only 14 vertex patterns are distinct in terms of topology of the triangulated surface due to complementary and rotational symmetries of the cube, which simplifies the design of the look-up table. A surface-edge intersection point is found via linear interpolation of the densities of two vertices of an intersecting edge.

The major advantage of this method over the previous tiling techniques is that inter-slice connectivity coherence is considered. This enables more surface details between slices to be displayed. The disadvantage of the method is that it requires a large amount of memory space to store all triangle faces produced. These faces then need to be scan-converted into a 2D image buffer with hidden surface removal, which could consume a large amount of computation time because of the large number of triangle faces.

2.1.4 Surface Detection via Gradient Operators

Unlike the triangulation techniques discussed previously that treat voxels as points, the approaches discussed in this and the next two subsections treat voxels as 3D parallelepipeds with faces and edges of their own. By processing a 3D scene as a whole, object surfaces consisting of voxel faces can be detected.

The method [22] discussed in this subsection works on a 3D gray-scale scene. The basic idea of the method is to use a 3D surface detector which detects the 3D boundary of an object based on the changes of local gradient values and incorporate a backtracking control scheme to correct errors.

A 3D boundary is defined as an ordered set of boundary elements satisfying a certain property P. Three edges of a voxel along x, y and z directions define the boundary elements. P is defined based on three requirements: i) high contrast; ii) connectivity; and iii) agreement with a priori knowledge. A boundary element has four neighboring boundary elements, that is, an x-edge of a voxel v has the same edges of the four voxels that are face-adjacent to v in y and z directions as its neighbors. Similar definitions for y- and z-edges apply.

The algorithm works as follows. Initially three boundary elements (along x, y and z directions) are chosen interactively by the user as the seed elements, and put into a queue. The algorithm then proceeds by connecting the appropriate neighbors of the current element (the first element removed from the queue) in a breadth first fashion (progressing in all three directions simultaneously) to form the desired surface. Gradient operators are defined to evaluate the neighboring elements. Based on the values of the gradient operator, the most promising element of the neighbors is chosen according to an a priori criterion and put into the queue to be processed later. If no such neighbor exists, backtracking is applied until the process can continue. The algorithm either stops when the queue becomes empty or fails if there are no alternatives.

This algorithm is general in the sense that the other surface detection methods discussed in the next two subsections only work on binary scenes. However, since the algorithm needs to compute a large number of gradient values and perform backtracking, its execution speed can be very slow.

2.1.5 Surface Detection via Graph-Theoretic Model

The approach proposed by Artzy et al. [1] translates the problem of surface detection into a problem of traversal of a directed graph. The key to their approach is to characterize objects and surfaces in a unique way.

The algorithm operates on a binary scene V bound by X, Y, Z in the three directions, respectively. The scene is defined as a 3D array of voxels:

$$V = \{v \mid v = (x, y, z), 1 \leq x \leq X, 1 \leq y \leq Y, 1 \leq z \leq Z\}$$

The interior region of the scene V, denoted by V^*, is defined as

$$V^* = \{v \mid v \in V, x \neq 1, y \neq 1, z \neq 1, x \neq X, y \neq Y, z \neq Z\}$$

Let Q be the set of all those voxels of V^* which have density value 1. B is then called an object of Q if B is an edge-connected component of Q. If W is a face-connected component of $\overline{Q} = V - Q$, and B and W are adjacent (i.e., there is some voxel b in B and another w in W such that b and w are face-adjacent), then the boundary $P(B, W)$ between B and W is defined as a set of faces:

$$P(B, W) = \{(b, w) \mid b \in B, w \in W, b \text{ is face-adjacent to } w\}.$$

Note that a face is an ordered pair of voxels.

The surface detection problem can now be formulated as a graph-theoretic one. First, the boundary $P(Q,\overline{Q})$ between Q and $\overline{Q}$ is defined as:

$$P(Q,\overline{Q}) = \{(b,w) \mid b \in Q, w \in \overline{Q}, b \text{ is face-adjacent to } w\}.$$

Then a directed graph called the boundary digraph $G(Q)$ of Q is established as

$$G(Q) = (P(Q,\overline{Q}), E)$$

where

$$E = \{(f_1, f_2) \mid f_1, f_2 \in P(Q,\overline{Q}), f_2 \text{ is adjacent to } f_1\}$$

The nodes of $G(Q)$ correspond to faces separating voxels in Q from voxels not in Q, while the edges of $G(Q)$ are determined by a relatively complex "adjacent to" relation between two faces. Essentially, each face f in $P(Q,\overline{Q})$ has two faces f_1 and f_2 in $P(Q,\overline{Q})$ that are adjacent to f, and f itself is adjacent to two other faces in $P(Q,\overline{Q})$. That is, each node of $G(Q)$ has indegree 2 (two incoming edges) and outdegree 2 (two outgoing edges). This result of $G(Q)$, combined with the result that the subgraph of $G(Q)$ consisting of the nodes $P(B,W)$ and the associated edges is a strongly connected component of $G(Q)$, implies that, for each node of the subgraph $P(B,W)$, there is a binary spanning tree of the subgraph rooted at the node. Therefore, the boundary $P(B,W)$ can be detected by traversing the $G(Q)$ to find the binary spanning tree of the subgraph $P(B,W)$ starting with a face (b,w) such that $b \in B$ and $w \in W$. This seed face (b,w) can be specified in an interactive fashion, for example, by displaying a slice of the binary scene and indicating a point on the desired boundary in the slice using a graphic input device.

2.1.6 Surface Detection in Multidimensions

This approach [34] works on any multidimensional binary scene (including a 3D scene, of course) and detects the hypersurfaces of a multidimensional object. This ability is especially useful when displaying an object changing in time since a dynamic object is essentially four-dimensional. By stacking 3D arrays of voxels along the fourth dimension of time the boundary detection can be performed on the resulting four-dimensional object. Then the boundary detected is intersected by hyperplanes corresponding to time instances at which the display of the moving object is desired. This obviates the need for identifying the object of interest in each time frame, which could be difficult due to changes in shape and size of the object.

The characterizations of object adjacency, connectivity, and boundary in multidimensions are the basis of the approach. A d-dimensional space is partitioned by d sets of mutually orthogonal hyperplanes and the resulting volume elements are called hypervoxels. A hypervoxel h in a d-dimensional space is represented as a d-tuple of integers $(h_1, \ldots, h_d)$. Two hypervoxels h and h' are defined as n-adjacent $(0 \leq n \leq d)$ iff they differ in exactly n coordinates and the difference in each of these n coordinates is either 1 or -1, that is,

$0 \leq |h_i - h'_i| \leq 1, \; i = 1, \ldots, d$

and

$\sum_{i=1}^{d} |h_i - h'_i| = n$

Also, h and h' are defined as O(n)-adjacent iff they are k-adjacent for some $k \leq n$. That is, h and h' are different in at most n coordinates. For example, in the three-dimensional case (d=3), two voxels are 3-adjacent iff they are vertex adjacent;

O(3)-adjacent iff they are vertex-, edge-, or face-adjacent.

Connectivity characterization is based on the O(n)-adjacent relation. An O(n)-path from h to h' is a sequence of hypervoxels $h = h^0, \ldots, h^m = h'$ such that h^{p-1} is O(n)-adjacent to h^p, for $1 \leq p \leq m$. Suppose that the subset S consists of all 1-hypervoxels in a d-dimensional space, all O(n)-components of S then induce a partition on S. (An O(n)-component of S is a subset of S which is O(n)-connected, that is, there exists an O(n)-path between any pair of hypervoxels of the subset in S.)

The boundary of S is defined as the set of hypervoxel faces:

$B(S) = \{(h, \overline{h}) \mid h \in S, \overline{h} \in \overline{S}, h$ is O(1)-adjacent to $\overline{h}\}$.

To facilitate the boundary detection of the objects in S, the concept of (n, k)-boundaries is essential. The O(k)-border of S is the subset $B_k(S)$ of S defined as

$B_k(S) = \{h \mid h \in S, h$ is O(k)-adjacent to $\overline{h}$ for some $\overline{h} \in \overline{S}\}$.

Then an (n, k)-boundary of S is defined as B(C):

$B(C) = \{(h, \overline{h}) \mid h \in C, \overline{h} \in \overline{S}, h$ is O(1)-adjacent to $\overline{h}\}$,

where C is an O(n)-component of the O(k)-border $B_k(S)$ of S. Intuitively, every hypervoxel of C is O(k)-adjacent to some hypervoxel in $\overline{S}$, but only those hypervoxels in C which are O(1)-adjacent to some hypervoxels in $\overline{S}$ are used to form the (n, k)-boundary determined by C. For any choice of n and k, $1 \leq n, k \leq d$, the set of all (n, k)-boundaries of S constitutes a partition of B(S). By taking some care (see [34] for details), an (n, k)-boundary uniquely describes a particular boundary of interest. The (n, k)-boundary can also be completely specified by a "seed" hypervoxel in the appropriate O(k)-border. In fact, the algorithm in [1] detects the

(2,1)-boundaries of a three-dimensional binary object.

The boundary detection algorithm detects an (n, k)-boundary of the set S. n and k are chosen by the user. For any point h in $B_k(S)$ the algorithm proceeds by determining the contribution of h to the (n, k)-boundary and by entering in a queue a subset of the hypervoxels that are O(n)-adjacent to h. This subset comprises those points that qualify as members of $B_k(S)$, that have not been entered in the queue previously, and whose contribution to the (n, k)-boundary has not already been determined. The algorithm stops when the queue becomes empty.

2.2 Volume Rendering for Objects Represented as 3D Arrays of Voxels

Two different approaches fall in this category. The *object space approach* loops over voxels of a 3D array and projects them onto the display screen in an order that ensures correct hidden surface relationships. The *image space approach*, on the other hand, is based on the principle of ray-tracing and loops over pixels on the display screen.

2.2.1 Slice-by-Slice Back-to-Front Approach

This method [12] is an object space approach. It traverses the slices, rows, and columns of a 3D array of voxels in order of decreasing distance to the observer. It is based on the following observation: Assuming the origin is farthest from the observer, it is simply necessary to traverse voxels in order of increasing x, y and z to achieve a correct hidden surface removal. If the origin is not the farthest corner

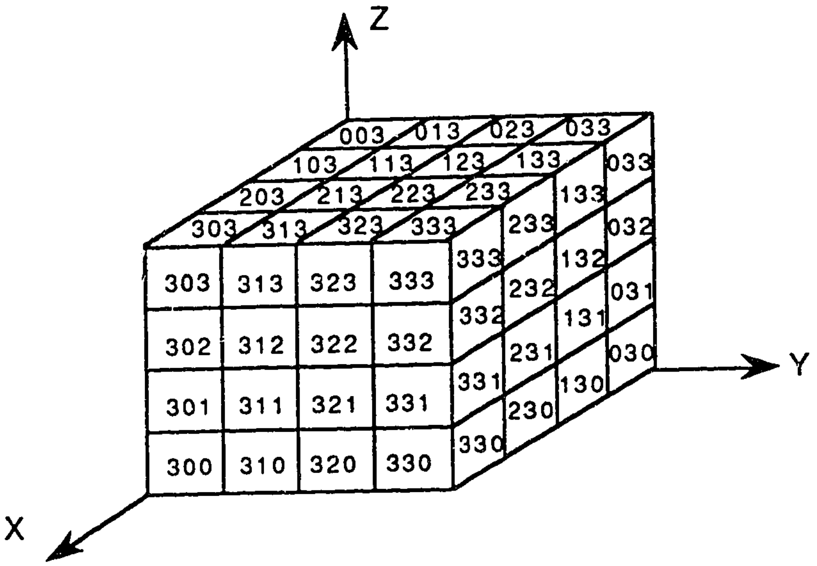


Figure 3: Slice-by-slice BTF read out sequence.

from the observer, then some of x, y, z should be increasing and some should be decreasing. However, the choice of which index changes fastest can be arbitrary. For example, for the orientation shown in Figure 3, one possible readout sequence is

000,100,200,300, 010,110,210,310, ... , 033,133,233,333

The coordinate transformation of a voxel is performed by multiplying the transformation matrix determined by a given viewing direction. The algorithm produces a depth shaded image and can handle gray-scale data.

One advantage of the approach is that data that are stored slice-by-slice can be read one slice (or portion of a slice) at a time from disk storage into main memory for processing. Once a slice is read in, the voxels in it are projected in the appropriated increasing or decreasing sequence of x and y values (assuming that slices are stacked in the z-direction). This ability makes it possible to implement the algorithm in a mini- or micro-computer that does not have a large main memory.

The main disadvantage of the method is its slow display speed. One important reason for this is that a lot of fruitless computations have to be done for those voxels that will be obscured later in the process.

One problem of the approach is that it treats a voxel as a point in space which projects onto a single pixel, rather than as a solid cube with visible faces to be painted. This could result in artifact holes at certain orientations because a solid voxel can in fact project onto several pixels at these orientations. Several approaches have been used to solve the problem [28, 12]. The simplest one is to restrict the scale-factor to be less than 1. If larger scale-factors are required, a projected image can be magnified in image space. An alternative is that for each voxel a region of the screen encompassing several pixels (for example, a rectangle corresponding to the bounding box of a projected voxel) can be painted. The third option is to consider each voxel as a cube with three visible faces, which can be painted with a standard polygon-filling algorithm [11]. The use of large pixels (magnified displays) in the first approach or large voxels in the second approach could result in a loss of real spatial information and the occurrence of aliasing errors. While the third approach gives the best image quality, it may require lots of computation time because of the large number of voxels to be processed.

2.2.2 Volume Rendering Based on Ray-Tracing

The principle of ray tracing is well-known. Briefly, from the eye of the observer a light ray is traced through the center of each image pixel until it encounters the surface of an object of interest, thus accomplishing visibility determination and perspective or parallel projection. The coordinate transformation is inherent in the orientation of the object with respect to the display screen. In the approach proposed in [30], the light rays are assumed to be parallel to each other and perpendicular to the screen; and further, first level ray-tracing is assumed: each light ray terminates when it encounters a voxel belonging to an object of interest. The algorithm establishes a line perpendicular to the screen through the center of each pixel. It then steps along this ray, testing the density values at multiple sample points to see if an object of interest has been encountered. If so, the stepping terminates, and a shading computation is invoked to assign an appropriate value to the image pixel.

The most important step of the algorithm is the criterion used to terminate the loop along a ray (e.g., when the ray is judged to have intersected the surface of an object of interest). This requires testing the voxel density at precise intervals along the ray, which in turn means that densities must be estimated (interpolated) between the collected points (voxels). First order (trilinear) interpolation, that is the weighted average of the 8 sample points (voxels) surrounding the point of interest is found to be most cost-effective.

The major advantage of the method is that more surface details can be retained and no aliasing "holes" can result. The major drawback is its slow execution speed.

2.3 Volume Rendering via Line Segments

To avoid useless computation for those voxels obscured later in the process, it is possible to take front-to-back (FTB) approaches for the slice-by-slice back-to-front scheme discussed in Section 2.2.1. However, a naive implementation of FTB has no advantage over the BTF approach since finding the pixel onto which a given voxel projects, and determining whether that pixel has already been lit, may be as expensive as simply painting over the previous value of the pixel. Some coherence within the data must be exploited to obtain a faster FTB approach.

The approach proposed in [29] is a slice-by-slice front-to-back method aimed to achieve a faster rendering speed than its counterpart. A dynamic data structure – the dynamic screen – is used to represent the unlit screen pixels. When each slice is accessed, only unlit pixels are processed and newly lit pixels are efficiently removed from the data structure and never considered again.

The object space is represented by a 3D array of binary voxels obtained by thresholding the corresponding 3D array of gray-scale voxels. Although in principle the approach can also deal with gray-scale data, it has an advantage in exploiting the data coherence of binary objects. Voxels are organized in a slice-by-slice and row-by-row (within a slice) fashion. Within each row runs of black voxels are represented as *line segments* in sorted order to facilitate the FTB traversal and the "merging" operation with the scanlines of the image.

The key observation to the approach is the special choice of object space and image space coordinate systems as well as object rotations. The object space coordinate system is chosen as: X_1, Y_1, Z_1 that are the three axes through the object center, fixed in image space with respect to the screen, and parallel to the axes of the image space X', Y', Z', respectively (Figure 4). In this way rows of voxels making up an object will be parallel to scanlines of the image screen after rotation about X_1 and/or Y_1 axes and projection (onto the X'Y' plane). Since a rotation γ about an axis perpendicular to the screen does not uncover any new

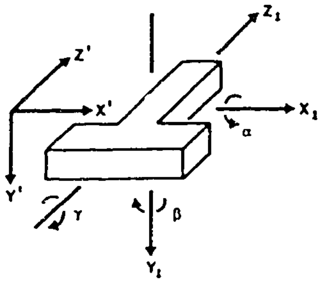


Figure 4: Selection of object and image space axes.

information of the objects on the screen, rotations α and β together basically provide any desired viewing direction. Therefore, by restricting rotations to α and β, line segments of an object can be simply "merged" with the line segments of the image to which they are parallel.

The dynamic screen data structure is a linked list representing the image on a scanline-by-scanline basis. The algorithm traverses the input data in a slice-by-slice, row-by-row, and line segment-by-line segment fashion, all in FTB order, and merges line segments of objects with scanlines of the screen via the dynamic screen data structure. Note that the dynamic screen data structure is used to check the line segments of the image that should be painted, and the associted depth images are stored in a separate two-dimensional array.

The run-time requirement of the algorithm is reported [29] to be much shorter (55slice-by-slice back-to-front method. As for the back-to-front approach, this approach also treats voxels as points in space, and therefore, the techniques discussed in Section 2.2.1 to avoid producing artifact holes can also be used here.

2.4 Gradient Shading

In general the output of an object space volume rendering method is a depth image (refer to methods discussed in Sections 2.2.1, 2.3 and later in Chapter 3 and 4). This makes it very difficult (if not impossible) to apply conventional polygon shading methods such as Gouraud or Phong shading methods [11] to generate final images with depth illusions. To overcome this difficulty, the gradient shading methods have been developed.

Gradient shading methods only use a depth image, that stores, for each pixel (x, y) of the display screen, the distance z(x, y) between the pixel and the visible point on the object surface that projects onto the pixel. The basic idea of gradient shading methods is that, from a depth image, the normal at any point of the object surface z = z(x, y) can be obtained from the gradient vector: $\nabla z = \left(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}, -1\right)$ by using difference estimations. The normal can then be used to calculate the shading value or intensity of a point on a surface via the basic shading formula $I = I_a + I_d + I_s$ [11]. A detailed review of major gradient shading methods is given in [32] and will not be repeated in this thesis.

Chapter 3

Display of Octree Encoded Objects

The *octree* representation of 3D objects is based on the principle of recursive subdivision. It has been studied for use in many applications. As surveyed in [8], many operations on objects benefit from the treelike structure of octrees and can be simply implemented as tree traversal procedures. Operations such as detecting intersection among objects, locating a point or a block in space, handling hidden-surface removal, connected component labeling, and neighbor finding can be greatly simplified by taking advantage of the hierarchical data structure, spatial addressability, and pre-sorted nature of octrees.

A number of algorithms for displaying 3D voxel-based objects via octree representations have appeared in the literature. Most of them work on binary objects [10, 25, 26, 38, 2, 3, 16, 8, 19], although some effort was also reported to display gray-scale images encoded by octrees [24]. Since octrees in general are not suitable to encode gray-scale images [16], only octrees of binary scenes are considered in this thesis. In this chapter, the octree representation is first reviewed, followed by the discussion of the existing techniques for displaying binary scenes via octree representations.

3.1 Octree Representation

An *octree space* is modeled as a cubic region consisting of $2^n \times 2^n \times 2^n$ unit cubes or voxels, where n and 2^n are the resolution and length of the octree space, respectively. Each voxel has a value 0 (white) or 1 (black), depending upon whether it is outside or inside the object that resides in the octree space. The octree representation of the object is obtained by recursively subdividing the cubic space into octants. A $2^d \times 2^d \times 2^d$, for $1 \leq d \leq n$, octant is divided into eight $2^{d-1} \times 2^{d-1} \times 2^{d-1}$ smaller octants if the voxels contained in the octant are not entirely 1's or 0's. The same process continues recursively until the octant (suboctant) contains voxels of a single value. Each octant generated in the process is assigned a label, from 0 to 7. Figure 5 shows the octant labeling used in this thesis, where the 0 octant is hidden from view.

The result of the recursive subdivision process is represented by a tree of degree 8 whose nodes are either leaves or have eight children. Thus, the tree is called an *octree*. Each node of the tree is given the same label as that assigned to its corresponding octant. The root node of the resulting tree represents the entire octree space. Leaf nodes represent octants containing voxels of the same value and are marked B (black) or W (white) accordingly. Non-leaf nodes are called

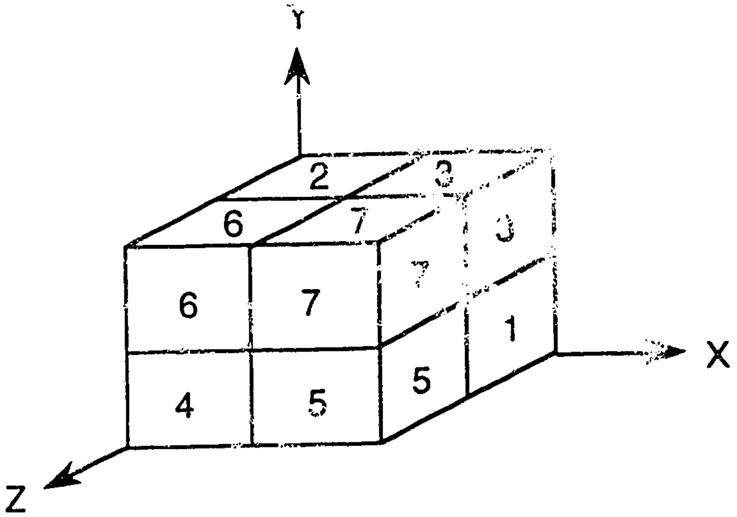


Figure 5: Octant labeling (octant 0 is hidden from view).

intermediate nodes and are marked G (gray). Figure 6 shows a simple object and its corresponding octree representation.

A number of variations of the basic octree definition have been proposed [8]. Among them, *linear octrees* [15] have received most attention because they generally take less memory space than regular octrees. A linear octree stores all B nodes of its corresponding octree in a one-dimensional array. Each node in the array has two fields: k (key) and r (resolution). k consists of n octal digits, which, from left to right, correspond to the labels along the path from the root to the node in the octree (0's are padded to the right if the path is shorter than n), and r is the resolution of the octant represented by the node. The nodes in the array are stored in ascending order sorted by their keys. For example, the linear octree representation of the object shown in Figure 6 is: (0,0) (1,0) (4,0).

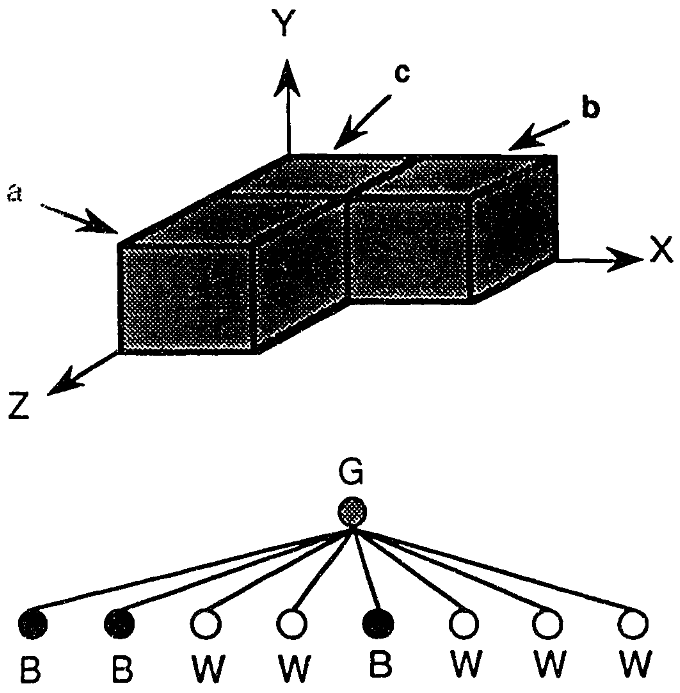


Figure 6: A simple object and its octree representation.

3.2 Boundary Detection via Linear Octrees

Unlike surface rendering approaches discussed in Chapter 2, which represent 3D boundaries by surface patches (triangles or voxel faces), the boundary detection algorithms [2, 3] discussed in this section detect 3D boundaries of a linear octree that consist of voxels or small octants. Such boundaries are also represented as linear octrees and thus can be displayed via the conventional linear octree display algorithm discussed later in Section 3.3.2.

3.2.1 The First Algorithm

Unlike previous surface detection algorithms discussed in Section 2.1, this approach [2] does not need a "seed" boundary element to start with. The algorithm works on binary objects represented as a linear octree. The basic idea of the approach is to eliminate the "internal boundaries" of the nodes of the linear octree repeatedly and obtain all boundary voxels simultaneously in the last step.

The approach is based on the following concepts. The object in a 3D scene is defined as the set of all black voxels and the background is defined as the set of all white voxels. The external border of the object, which is the output of the surface detection algorithm, is defined as a set of black voxels that are either on the boundary of the 3D scene or face-adjacent to at least one voxel of the background. Furthermore, the size of a node is p ($0 \leq p \leq n$) if the node corresponds to a $2^p \times 2^p \times 2^p$-cube, and the border of the node is the outmost layer of voxels of the cube. It is useful to consider the border of a node as consisting of six sub-borders associated with six principal directions (i.e., +X, -X, +Y, -Y, +Z, -Z). A sub-border of a node is an internal border if it is not an external border. Finally, the d-neighbor (d is +X, -X, +Y, -Y, +Z, or -Z) of a node A of a linear octree is defined as its neighboring node Y (Y may not be a node of the linear octree) such that Y lies to the d-direction of A and is of the same size as A. Node A is said to be blocked in direction d if node Y is black, otherwise A is said to be unblocked.

The basic process of the algorithm is to eliminate the internal borders starting from nodes with higher resolutions. Whenever all six borders of a node are deleted, the node itself is deleted too. This process can be applied recursively, all the interior sides and nodes disappear, and what remains is the external border.

The algorithm initially marks each black node of an input linear octree as unblocked in all six directions. Then starting from the nodes with the highest resolution it checks each node in this group to see if it is blocked in all six-directions and marks it. If the node is blocked in all six directions, it is deleted from the input list since it is an interior node and contributes nothing to the border. If the node is not blocked in all six directions, it is split into eight smaller nodes corresponding to the eight smaller octants, and the smaller nodes, except for those that are blocked in all six directions (a smaller node is blocked in three predetermined directions; for other three directions it is blocked if the parent node is), substitute for the parent node in the input list. The algorithm works on the input list from the highest resolution to 0 in this way, and, finally, the input list contains only the desired border voxels.

3.2.2 The Improved Algorithm

This approach [3] improves the first algorithm discussed in Section 3.2.1 in terms of execution time and storage space. The key features of this approach are the node expansion scheme and the employment of a recursive process.

In the first algorithm a node with resolution $r\,(1 \leq r)$ would be expanded into eight smaller nodes with resolution $r-1$ if the node was found to be unblocked in at least one direction, and the smaller nodes that are unblocked in at least one direction are then added to the node list joining the subsequent process for the nodes with resolution $r-1$. However, in general, the node resolutions of an input linear octree are: $g(r), g(r-1), \ldots, g(0)$ $(g(i) > g(i-1), i = 1, \ldots, r)$, with $g(r)$ being the largest resolution and $g(0)$ being the smallest one. This implies that two adjacent resolution numbers may not necessarily have a difference of 1. Bearing this in mind a new expansion scheme was proposed: When a node with resolution $g(i)$ needs to be expanded, the node can be directly mapped to its border nodes with resolution $g(i-1)$. For example, if $g(i) - g(i-1) = 1$, the border nodes are just the eight child octants resulting from the subdivision of the node. In general, $g(i) - g(i-1) = m$, so the node is subdivided m times recursively, with the border nodes being the border octants thus obtained. This direct mapping can be done efficiently by using look-up tables. Note that in the first algorithm a node with resolution $g(i)$ is also expanded ultimately into its $g(i-1)$ border nodes, but most of the expansion and searching process from resolution $g(i) - 1$ to $g(i-1) + 1$ are fruitless.

In the first algorithm a linked list data structure for nodes has to be used to facilitate the node deletion and insertion, and this is costly. In the improved algorithm the nodes of an input linear octree are stored in $r+1$ one-dimensional arrays: $L(r), L(r-1), \ldots, L(0)$, with $L(i)$ containing the nodes of resolution $g(i)$ ($i = 0, \ldots, r$) in sorted order. The algorithm starts processing each node in $L(r)$, then $L(r-1)$, ..., until $L(0)$. For each node in $L(r)$ the algorithm invokes the main procedure with parameter r to check whether the node is blocked in all six directions (searching its black neighbors in $L(r)$ and marking it). If the node is blocked, the process for the node is finished. Otherwise the node is mapped to its $g(r-1)$ border nodes. The algorithm then goes to process each of these unblocked border nodes by invoking the main procedure with parameter $r-1$ recursively until a border node with resolution $g(0)$ is found and output as the border node of the object. Note that no new nodes resulting from the expansion are needed in the searching since their effects have been incorporated in the expansion process. Also the output is one by one in a depth first fashion. That is, all $g(0)$-border-nodes (if any) from the first $g(r)$-node of the input linear octree are first produced one by one, then the ones from the second $g(r)$-node, ..., and finally the ones from the last $g(0)$-node. The border thus obtained is a set of $g(0)$-nodes and may not be voxels. Note that the border nodes obtained by this approach are not in a sorted order as in the first algorithm.

3.3 Rendering of Octrees

Methods discussed in this section can be considered as volume rendering approaches because they can access any part of a 3D scene represented by an octree (or linear octree). Essentially, all these methods are based on the following observation: Since the octree representation maintains all elements of an object in a spatially pre-sorted order, one only needs to follow a specific order to visit the octree (or linear octree) nodes and project B octants onto the screen with the correct hidden surface results.

There are two types of visiting orders: back-to-front (BTF) [10, 25, 16] and front-to-back (FTB) [26, 19, 8, 38]. In the BTF traversal order, octants farther away from the viewer are projected before those that are closer. Here, an octant visited later overwrites the painted region of any octant visited earlier. For example, in Figure 5, a valid BTF order is 02416537 since octant 0 (hidden) can not obscure the remaining seven octants and octants 2 can not obscure the remaining six octants, and so on. Note that more than one visiting order may exist. For instance, the visiting orders 02146357, 04126357, and 04123567 are all eligible in this example. In the FTB visiting order, octree nodes closer to the viewer are projected before those which are farther away. Once a region of the screen is painted, any other nodes projected onto it are ignored.

3.3.1 BTF Display of Octrees

This approach [25] traverses an octree to be rendered in a BTF order determined by an arbitrary viewing direction. If a B octant is encountered during the traversal, it is projected onto the display screen. If a G octant is encountered, it is further traversed according to the same BTF order. W octants need not be processed.

The algorithm is very simple, but the projection of all B octants could be time-consuming. This is because to obtain a high quality image, visible faces of all B octants need to be scan-converted to generate the depth image of visible surfaces, which can then be used for gradient shading [32]. In general the scan-conversion [11] process takes most of the execution time of the algorithm (see Sections 4.4 and 4.5 for time complexity analysis and some experimental results of the BTF algorithm).

To improve its efficiency, Doctor et al. [10] simplified the scan-conversion process by restricting viewing directions along the six principal directions (+X, -X, +Y, -Y, +Z, -Z) of the octree space. The application scope of that algorithm, however, is limited because of the limitation on viewing directions.

3.3.2 BTF Display of Linear Octrees

Linear octrees can also be displayed by simulating the BTF traversal of octrees [16]. To do so, simply invoke the procedure BTF_LO(LO, 1), where LO is an input linear octree, stored as a one dimensional array, and procedure BTF_LO is defined as follows.

```
BTF_LO(SUBLO, L)
/* SUBLO is a sub-set of LO. L is the level of recursion. */
{
    If (SUBLO contains a single node, NODE) {
        Project the visible faces of NODE onto the screen.
    } else {
        1. Scan SUBLO and register addresses of 1st nodes of
           each non-empty octant at level L (at most 8 such
           octants) by looking at the Lth digits (from left)
           of the keys of the nodes in SUBLO.
        2. According the BTF order determined by the viewing
           direction, process each of the non-empty octants
           by putting its nodes into SUBLO' and invoke
           BTF_LO(SUBLO', L+1).
    }
}
```

Note that in this algorithm a G octant containing a single B sub-octant is not traversed any further and the B sub-octant is projected onto the screen immediately.

3.3.3 FTB Display of Octrees with Arbitrary Viewing Directions

A FTB approach could be faster than a BTF one because many obscured octants are not visited nor projected. The overhead, however, to determine the visibility of octants during octree traversal could well exceed the speed gain if the visibility test isn't efficient. Several FTB methods for octree representations exist [26, 19, 8, 38]. Among them, the algorithm proposed by Meagher [26] is the only one that allows arbitrary viewing directions. The basic idea of that algorithm is discussed in this subsection.

In Meagher's algorithm, a quadtree [31] is used to represent the display screen. Initially, the quadtree contains only one root node, colored W. The input octree is then traversed in a FTB order determined by a given viewing direction. To determine the visibility of a G or B octant visited during the octree traversal,

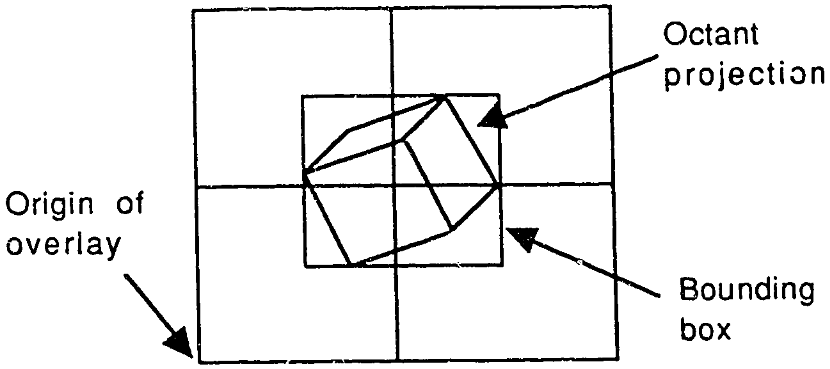


Figure 7: Octant projection and its overlay.

a concept called *overlay* was introduced. As shown in Figure 7, in general, the projection of an octant consists of three quadrilaterals. These polygons can be enclosed by a bounding box, that is the smallest rectangle with edges oriented parallel to the coordinate system of the quadtree. The overlay of the projection is defined as the four possibly intersecting quadrants at the lowest level (of the quadtree) such that the largest dimension of the bounding box is the same size or smaller than the edge size of a quadrant.

Since the overlay of the projection of an octant encloses or covers the projection completely, the visibility test of the octant can be performed in the following manner: If all four intersecting quadrants of the overlay are colored B, the octant is hidden and discarded with all its descendants (if any). If not, additional work is required. If the octant is G, its eight children are processed in like manner. Otherwise (the octant is B), the children of the overlay quadrants are examined. Intersection tests between the projection and a child quadrant must be performed to see whether the projection (1) covers, (2) partially intersects, or (3) does not intersect the quadrant. If (1) holds, the child quadrant is colored B; If (3) holds, the child quadrant is colored W; If (2) holds, the child quadrant is colored G, and its eight children are examined in a similar manner.

Intersection tests between the projection of an octant and a quadrant are performed as a two step process. First, the quadrant is compared to the bounding box of the projection. If it does not intersect the bounding box, it will not intersect the projection itself. Otherwise, the second step is needed. Observe that the six outer edges of the projection form a convex polygon. Each line (on which an outer edge is a segment) divides the plane of the screen into two half planes. The half plane containing the projection is the positive side of the line. The other half-plane is the negative side. If the quadrant is entirely on the negative side of any of the six lines, from the definition of a convex polygon there can be no intersection. If the quadrant is intersecting with some of the six lines, it also intersects the projection. The quadrant is enclosed by the projection if and only if it is entirely on the positive sides of all six lines.

No time complexity analysis, comparison results with other octree display algorithms, and experimental statistics of computation time of the algorithm are given in [26]. As can be seen from the discussion above, the computation for intersection tests is expensive and needs to be performed extensively. Therefore, it can not be justified that the algorithm is faster than the simple BTF algorithm discussed in Section 3.3.1.

6, 7	2, 3
4, 5	0, 1

Figure 8: +X-face view.

3.3.4 FTB Display of Octrees with Restricted Viewing Directions

Approaches discussed in this subsection [19, 8, 38] display an octree (or linear octree) in a FTB fashion, but avoid the expensive polygon intersection tests in determining the octant visibility. The basic idea of these approaches is to restrict the viewing directions in exchange for the efficiency of octant visibility tests.

In [19] the viewing directions are restricted to be along coordinate axes of the octree space. This makes the projection of octants coincide with quadrants of the quadtree. For example, the mapping between octants and quadrants of the +X-face view of Figure 5 is shown in Figure 8. Obviously, the visibility of an octant can be determined by simply locating its corresponding quadrant in the quadtree. No geometrical intersection is involved; only simple quadtree traversal is needed.

The approach used in [8] is a little complicated. There, only an edge-view is allowed, that is, the viewing direction is parallel to the plane of two axes of the octree space and bisects the angle formed by these two axes. For example, Figure 9 shows the edge-XZ view of the octants of Figure 5. As shown in the figure, the

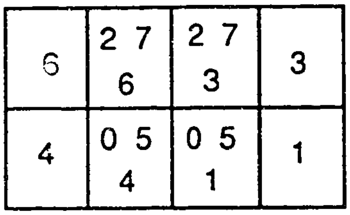


Figure 9: Edge-XZ view.

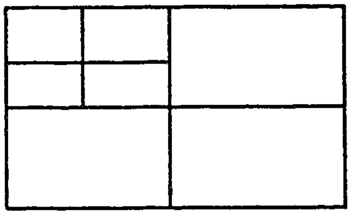


Figure 10: Subdivision of an edge view.

edge view of a cube is a rectangular region of aspect ratio $\sqrt{2}$. By subdividing an edge view recursively as shown in Figure 10 and representing it as a quadtree, the octant visibility test can also be performed without polygon intersections.

Finally, the octant visibility can also be determined without polygon intersection test for a corner view [38]. A corner view is a view where the viewing direction is along the line joining a corner and the center of an octant. A corner view can be referred to by the corner that defines it. As an example, the corner-7 view of a cube is shown in Figure 11. The corner view of a cube is a regular hexagon. Observe that the projections of the eight octants partition the hexagon into 24

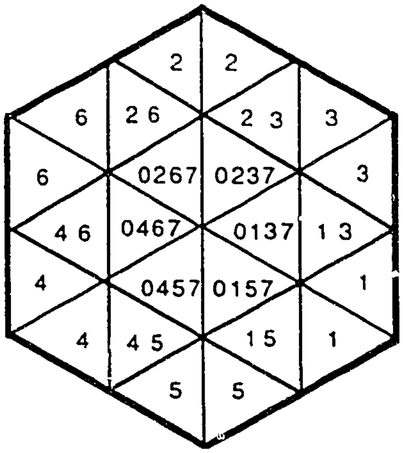


Figure 11: Corner-7 view.

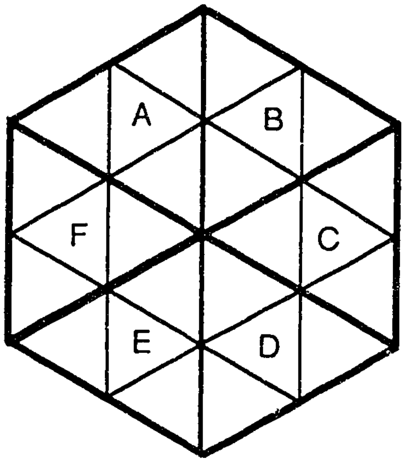


Figure 12: Grouping of triangles.

triangles and that each octant projects to six such triangles. The approach is to group the 24 triangles into six larger triangles (see Figure 12) and represent each by a triangular quadtree.

Chapter 4

Fast Display of Octree Encoded Objects

In this chapter a new approach is proposed for displaying octree encoded objects. It is a front-to-back approach aimed at achieving a fast rendering speed with no restriction on the viewing directions. The key to the approach is a new concept called *blocking quadtrees*, which enables the octant visibility test to be performed very efficiently. Section 4.1 describes how to determine the viewing transformation matrix and FTB visiting order. Section 4.2 introduces the blocking quadtrees. Section 4.3 presents the display algorithm. Section 4.4 analyzes the time and space complexity of the algorithm. Section 4.5 gives the experimental results of implementing the algorithm.

4.1 3D Coordinate Transformation and FTB Visiting Order

3D Coordinate Transformation.

Suppose that the octree space is described by a 3D right-handed XYZ coordinate system (Figure 5) and that the direction from the origin to the viewer is given by the vector (x1, y1, z1) called ***view plane normal*** (VPN). To project a point of the octree space onto the view plane (which is perpendicular to the VPN) or the screen, the following transformations are needed [16]: First, the X- and Z-axes are rotated about the Y-axis so that the positive Z-axis coincides with the projection (x1, 0, z1) of the VPN on the X-Z plane, thus making the Y-axis, the Z-axis, and the VPN coplanar. Then the Y- and Z-axes are rotated around the X-axis so that the positive Z-axis coincides with the VPN (which is now in the Y-Z plane).

So far the rotations of the axes are around the origin. But the origin is conventionally located in one corner of the octree space (Figure 5), and it is desirable to rotate the object around the center of the octree space (which is at (2^{n-1}, 2^{n-1}, 2^{n-1})) so that it stays in the octree space. Therefore, the necessary transformations are as follows: the axes are first translated so that the origin is in the center, then the rotations are performed, and finally the axes are translated back. The resulting X-, Y- and Z-axes are then called X'-, Y'- and Z'-axes respectively, which form the so called *image space*. The viewing matrix, V-MATRIX, that performs the above transformations is defined as:

V-MATRIX = $T_1 R_Y R_X T_4$ =

$$\begin{pmatrix} sin\alpha & -(cos\alpha)(sin\beta) & (cos\alpha)(cos\beta) & 0 \\ 0 & (cos\beta) & sin\beta & 0 \\ -cos\alpha & -(sin\alpha)(sin\beta) & (sin\alpha)(cos\beta) & 0 \\ 2^{n-1}f & 2^{n-1}g & 2^{n-1}h & 1 \end{pmatrix}$$

Where

$f = 1 - sin\alpha + cos\alpha,$

$g = 1 - cos\beta + (cos\alpha + sin\alpha)(sin\beta),$

$h = 1 - sin\beta - (cos\alpha + sin\alpha)(cos\beta),$

$sin\alpha = z1/D1, cos\alpha = x1/D1, (0 \leq \alpha \leq 2\pi),$

$D1 = (x1^2 + z1^2)^{1/2},$

$sin\beta = y1/D, cos\beta = D1/D, (-\pi/2 \leq \beta \leq \pi/2).$

$D = (x1^2 + y1^2 + z1^2)^{1/2}.$

Note that the projection described above is parallel orthographic [11].

Determination of FTB orders.

A FTB order can be easily determined by determining the octant nearest to the viewer, which in turn can be determined by simply looking at the coordinates of the VPN. Figure 13 gives the mapping from VPN's to the corresponding nearest octants. The nearest octant should be visited first, followed by the three octants, in any order, whose labels differ from that of the nearest octant by 1 binary digit, and then the three octants, in any order, whose labels differ from that of the nearest octant by 2 binary digits, and finally the octant whose labels differ from that of the nearest octant by 3 binary digits. For example, if a given VPN is (1, 1, 1), then a valid FTB order is 7 (111_2), 6 (110_2), 5 (101_2), 3 (011_2), 4 (100_2), 1 (001_2), 2 (010_2), 0 (000_2). Note that a VPN cannot be given as (0, 0, 0).

x1	y1	z1	octant
<0	<0	<0	0
<0	<0	= 0	0
<0	<0	>0	4
<0	= 0	<0	0
<0	= 0	= 0	0
<0	= 0	>0	4
<0	>0	<0	2
<0	>0	= 0	2
<0	>0	>0	6

x1	y1	z1	octant
= 0	<0	<0	0
= 0	<0	= 0	0
= 0	<0	>0	4
= 0	= 0	<0	0
= 0	= 0	= 0	-
= 0	= 0	>0	4
= 0	>0	<0	2
= 0	>0	= 0	2
= 0	>0	>0	6

x1	y1	z1	octant
>0	<0	<0	1
>0	<0	= 0	1
>0	<0	>0	5
>0	= 0	<0	1
>0	= 0	= 0	1
>0	= 0	>0	5
>0	>0	<0	3
>0	>0	= 0	3
>0	>0	>0	7

Figure 13: Mapping between VPNs and nearest octants.

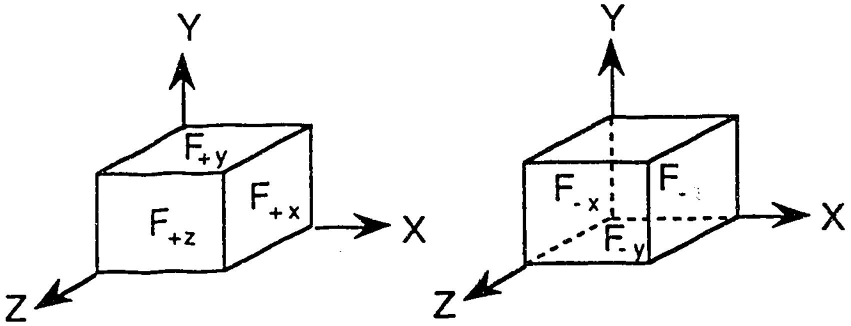


Figure 14: Six faces of an octant.

4.2 Blocking Quadtrees

4.2.1 Blocked Faces

An octant has six faces as shown in Figure 14, where the outward face normals of faces F_{+x}, F_{-x}, F_{+y}, F_{-y}, F_{+z} and F_{-z} are along the +X-, -X-, +Y-, -Y-, +Z- and -Z-axes, respectively. For an arbitrary viewing direction, at most three of the six faces of an octant are visible (Figure 15). These faces of an octant visible from a given viewing direction are called ***potentially visible faces*** of the octant, and the rest ***potentially invisible faces*** of the octant. Obviously, a potentially invisible face of an octant in an octree is always invisible from the viewing direction, while a potentially visible face may be completely visible (if no other B octants obscure the face), partially visible (if other B octants obscure the face partially), or completely invisible (if other B octants obscure the face completely) from the viewing direction.

For the convenience of discussion, it is assumed hereafter that a given viewing direction makes three faces of an octant potentially visible. This assumption is general since, as will be seen later, the derived algorithm can be easily extended

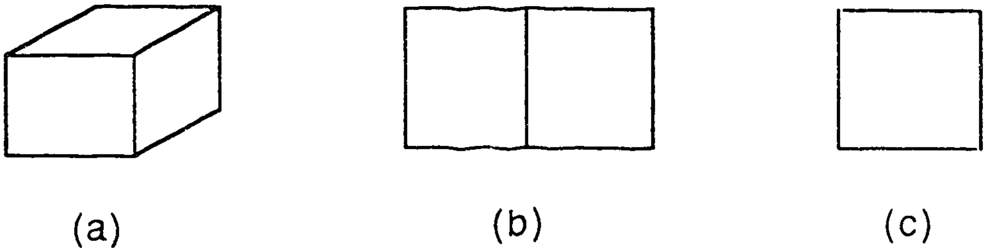


Figure 15: An octant may have (a) 3, (b) 2, or (c) 1 potentially visible faces.

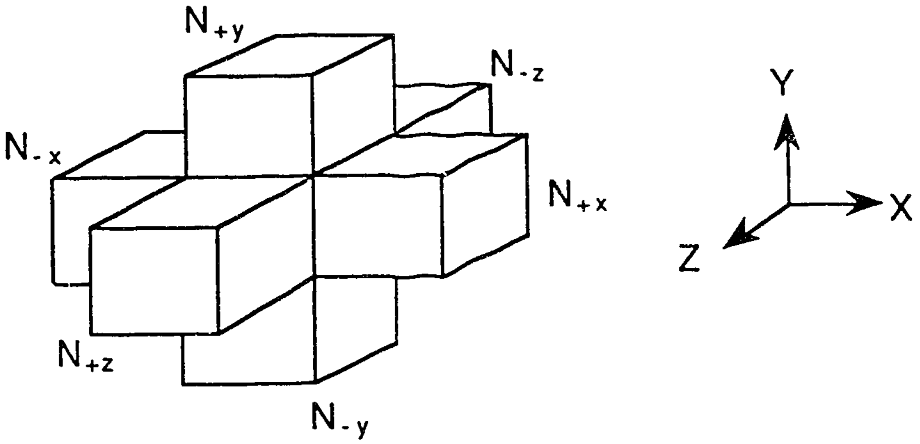


Figure 16: Six neighboring octants of an octant.

to handle arbitrary viewing directions.

An octant has at most six neighboring octants of the same size that share a face with it (Figure 16). Note the correspondence between labels of an octant's six faces and its neighboring octants (Figure 14 and Figure 16). These neighboring octants sharing the potentially visible faces of an octant are called *front neighboring octants* of the octant, and the rest *back neighboring octants* of the octant. For example, in Figure 16 , the front neighboring octants are N_{+x}, N_{+y} and N_{+z}, and the back neighboring octants are N_{-x}, N_{-y} and N_{-z}.

A *blocked face* of an octant is a potentially visible face of the octant whose corresponding neighboring octant is a blocking octant. A *blocking octant* is defined as (1) a B octant, or (2) a blocked W or G octant. A *blocked octant* is an octant with three blocked faces. On the other hand, those potentially visible faces of an octant that are not blocked faces are called *un-blocked faces* of the octant. For example, in Figure 6, octant **c** has two blocked faces, F_{+x} and F_{+z}, and face F_{+y} of octant **c** and face F_{+z} of octant **a** are un-blocked faces. It should be noted that a face without a neighboring octant is an un-blocked face.

Several simple observations can be made from the above definitions. First, a blocked face of an octant is invisible from the given viewing direction. Second, a blocked octant is invisible from the given viewing direction. Third, the front and back neighboring octants of an octant are visited *before* and *after* the octant itself, respectively, if the corresponding octree is traversed in the FTB order determined by the viewing direction (refer to Section 4.1). These observations ensure the correctness of the fundamental schema of the proposed algorithm described in the next subsection.

4.2.2 FTB Traversal with Blocking Quadtrees

The fundamental schema of the proposed algorithm for displaying an octree is as follows: For a given viewing direction, an octree to be rendered is traversed in the FTB order determined by the viewing direction. During the traversal an octant visited is processed as follows: If it is a blocked octant, set the faces shared by the octant and its back neighboring octants as blocked faces and discard the octant with all its descendants (if it is a G octant). Otherwise (the octant is not a blocked octant): If it is a B octant, paint the visible part of all un-blocked faces of the octant and set the faces shared by the octant and its back neighboring octants as blocked faces. If it is a W octant, set the faces shared by the octant and its back neighboring octants as un-blocked faces. If it is a G octant, traverse its eight sub-octants according to the FTB order and process them similarly.

The above schema is efficient because of three reasons. First, many blocked octants are simply discarded with their descendants (if any). Second, only un-blocked faces of B octants join the painting phase. Third, the overhead to keep track of the necessary information about blocked and un-blocked faces during the octree traversal can be minimized by a technique called blocking quadtrees.

The *blocking quadtrees* consist of three quadtrees with the same resolution as that of the octree to be rendered. They are called *quadtree-X*, *quadtree-Y* and *quadtree-Z*, representing faces orthogonal to X-, Y-, and Z-axes, respectively. The following modified schema makes use of the technique of the blocking quadtrees.

Initially, each of the three quadtrees has only one node, a W root node, representing the corresponding un-blocked face of the root octant. During octree traversal, the potentially visible faces of an octant encountered are mapped onto their corresponding nodes of the quadtrees. That is, the face perpendicular to the X-axis is mapped onto the corresponding node of quadtree-X, and so on. If all three quadtree nodes are B, the octant is blocked and is discarded with all its descendants (if any). Otherwise: If the octant is a B octant, paint the visible part of its potentially visible faces whose corresponding quadtree nodes are not B and then change these quadtree nodes to B. If it is a W octant, change all corresponding quadtree nodes to W. Lastly, if it is a G octant, perform the following three steps

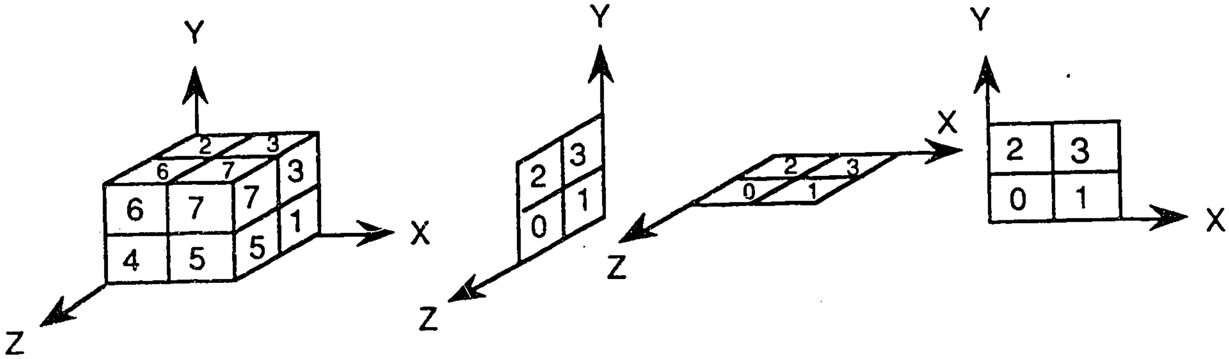


Figure 17: Octant labeling of octree space and quadrant labeling of blocking quadtree spaces.

in sequence: (1) For each corresponding B (or W) quadtree node, change it to G and generate four B (or W) son nodes for it, (2) process eight sub-octants of the octant recursively according to the FTB order, and (3) change each corresponding quadtree node to B if the quadtree node has four B son nodes.

Figure 18 shows the status of the blocking quadtrees (in quadrant format) at four consecutive stages of the octree traversal of the object shown in Figure 6. Note that the correspondence between octant labeling and quadrant labeling is determined by Figure 17. Figure 19 shows the last stage of the blocking quadtrees by assuming that octant **c** is a W octant. Notice that two B quadrants are changed to W quadrants because the W octant **c** updates the quadtrees. Figure 20 shows how the blocking quadtrees are refined before a G octant is traversed (assuming octant **c** is a G octant).

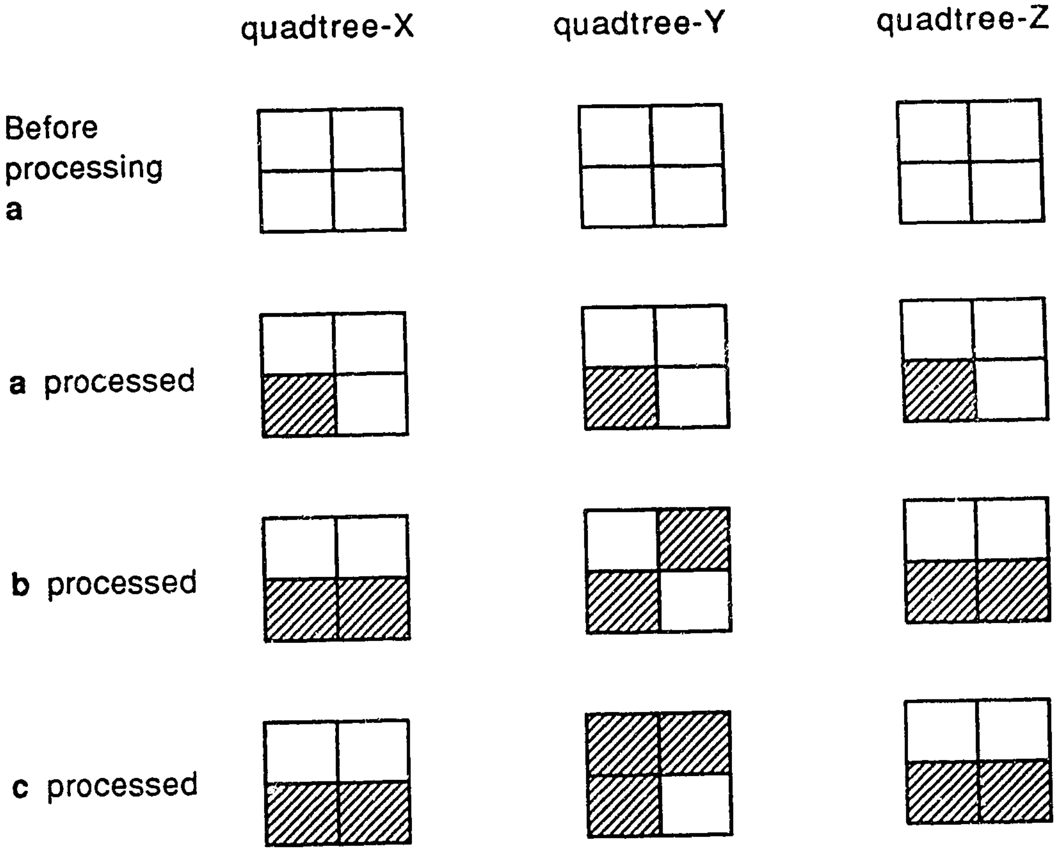


Figure 18: Status of blocking quadtrees for displaying the object in Figure 6.

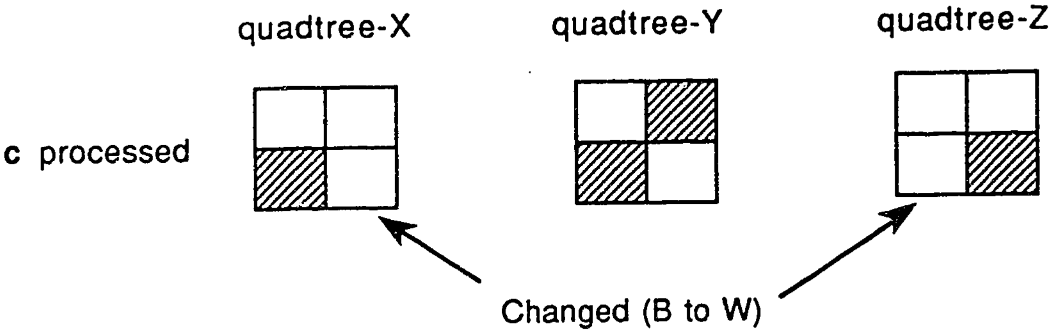


Figure 19: Status (last stage) of blocking quadtrees for displaying the object in Figure 6 (assuming octant c is W).

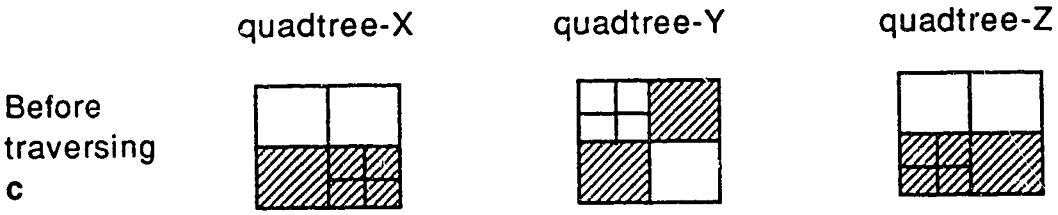


Figure 20: Status (last stage) of blocking quadtrees for displaying the object in Figure 6 (assuming octant c is G).

4.2.3 Blocking Bitmaps

Three formats could be used to represent a quadtree of the blocking quadtrees. They are: a regular quadtree with pointers, a linear quadtree, and a proposed format called *blocking bitmap.* As discussed later, the proposed format is best suited for fast display with a reasonable memory requirement.

A *blocking bitmap* is a one dimensional array of all nodes of its corresponding complete quadtree. A *complete quadtree* is a specialized quadtree which represents the recursive subdivision of the quadtree space down to the pixel level. In other words, a complete quadtree represents all bitmaps with different resolutions of the quadtree space. For example, Figure 21 shows the complete quadtree of a $2^2 \times 2^2$ quadtree space, where the root node represents the quadrant corresponding to the quadtree space, and its four child nodes represent the four quadrants obtained by subdividing the quadtree space, and so on. The nodes of a complete quadtree are stored in a top-to-bottom and left-to-right fashion in its corresponding blocking bitmap, and each node of the blocking bitmap takes 2 bits of memory space to store one of the three possible colors (W, B, G) of the corresponding complete quadtree node. For example, the correspondence between a complete quadtree and its blocking bitmap is shown in Figure 22.

The major advantage of representing the blocking quadtrees as blocking bitmaps is that mapping a potentially visible face of an octant to its corresponding blocking quadtree node becomes basically a simple array indexing operation. This ensures that the most extensive part of the overhead computation incurred by the FTB approach is performed in an extremely efficient manner. The mapping mechanism will be described in detail in Section 4.3 where the display algorithm is presented.

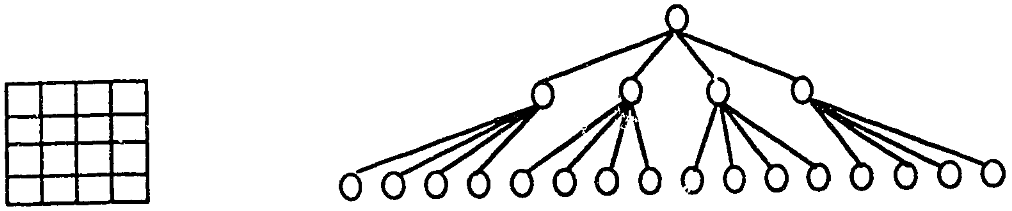
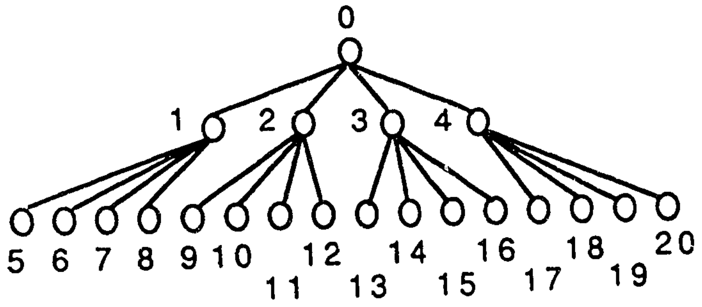


Figure 21: A 4 by 4 quadtree space and its complete quadtree.



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20			

Figure 22: A complete quadtree and its blocking bitmap.

The other advantage of using blocking bitmaps is that updating the blocking quadtrees can be performed very efficiently because no memory allocation is needed and only different values are written into already existing nodes.

As for the memory space requirement, since each node takes only two bits and no pointers are stored, the overhead is acceptable (refer to Section 4.4 for the space complexity analysis).

Alternatively, regular quadtrees with pointers or linear quadtrees could be used to represent blocking quadtrees. However, the following disadvantages make these approaches inappropriate.

For regular quadtrees with pointers, space must be allocated and deallocated dynamically when the blocking quadtrees are updated, which adds on extra computational burden. More seriously, if the quadtrees become sufficiently complex, the memory space needed to store them will be greater than that of blocking bitmaps because of the space taken by the pointers. The memory space for the worst case of a regular quadtree with pointers is far greater than that of the proposed format and has yet to be guaranteed since in general it can't be determined in advance how complex the quadtrees will become during the course of the program execution.

For linear quadtrees, the space requirement might be smaller than the proposed format since they only store the B nodes of the corresponding regular quadtrees. However, there is no guarantee since each node in a linear quadtree is represented by its linear code, which takes 3n bits (where n is the resolution of the octree space), and the linked list structure must be used to maintain the linear quadtree (since it is to be updated dynamically), which needs extra space for storing pointers. More seriously, the computation for checking against and updating blocking quadtrees is in general much more expensive than that of the proposed format. This is because: (1) a binary search must be performed to find a desired quadrant in a linear quadtree for mapping of an octant face, whose speed depends on the number of nodes in the linear quadtree and is slower than that of a simple constant-time array indexing operation as for the proposed format in most situations, and (2) updating a linear quadtree involves allocating and deallocating memory space and linked list pointer re-arrangement, which consumes more computation time than simply writing integer values to array elements.

4.3 Algorithm

The proposed FTB octree display algorithm traverses an octree in a FTB order determined by a given viewing direction and produces a depth image of visible surfaces of the object represented by the octree. The depth image is stored in a two-dimensional *z-buffer* array that can be used to generate a shaded image with any of the existing gradient shading methods [32] in a post-processing step. In the following, the algorithm is first presented, then some of the details are further explained. To avoid confusion, octree nodes will be called octants and blocking quadtrees nodes will be called nodes.

4.3.1 The Algorithm

Algorithm: FTB Display of an Octree.

1. If the root octant of the octree is W, then terminate.

2. Initialize the z-buffer to "un-painted".

3. If the root octant of the octree is B, then scan-convert its visible faces to the z-buffer and terminate.
4. Initialize the root nodes of the blocking quadtrees to W.
5. Invoke Procedure "Process a G Octant" to process the root octant, and then terminate.

Procedure: Process a G Octant.

1. Locate nodes of the blocking quadtrees that correspond to the potentially visible faces of the octant.
2. If at least one node is not B, do steps 3-5.
3. Refine the blocking quadtrees.
4. Invoke Procedure "Process a G Octant" or "Process a W or B Octant" to process each of the eight suboctants of the octant according to the FTB order.
5. Compact the blocking quadtrees.

Procedure: Process a W or B octant.

1. Locate nodes of the blocking quadtrees that correspond to the potentially visible faces of the octant.
2. If at least one node is not B, do steps 3-4.
3. If the octant is a B octant, then invoke Procedure "Scan-Convert a Face" for each un-blocked face of the octant.

4. Update the blocking quadtrees.

Procedure: Scan-Convert a Face.

1. Find the plane equation of the face in image space.
2. For each scanline that intersects the face, do steps 3-4.
3. Find intersections of the scanline with the left and right edges of the face.
4. Fill in all "un-painted" pixels of the z-buffer between the pair of intersections with the corresponding z values of the face obtained through the plane equation of the face.

4.3.2 Further Explanations

Indexing of Blocking Bitmaps

A more detailed explanation is needed for procedures "Process a G Octant" and "Process a W or B Octant" to understand the operations on the blocking quadtrees. Remember that the blocking quadtrees are represented as blocking bitmaps.

Two parameters, *depth* and *card_num*, are needed to invoke procedure "Process a G Octant". *depth* is the depth of the octant of the octree, where the root octant of the octree has depth 0, and its child octants have depth 1, and so on. *card_num* is a one-dimensional array of three elements. *card_num*$[i]$ (i is X, Y or Z) stores the cardinal number of the potentially visible i-face of the octant. An *i-face* of an octant is perpendicular to the i-axis of the octree space and the *cardinal number* of a face is defined as the cardinal number of its corresponding

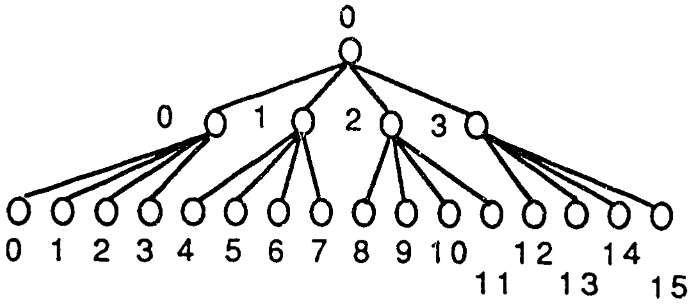


Figure 23: Cardinal numbers of a complete quadtree.

node in a blocking quadtree as shown in Figure 23. Note the difference between Figure 23 and Figure 22. The numbers in Figure 22 are called *index numbers* of the nodes. It should be noted that the cardinal numbers of three visible faces of the root octant are 0.

Three local variables, *ind_num*, *ind_num_son* and *card_num_son* are needed in procedure "Process a G Octant". They are explained as follows.

ind_num is a one-dimensional array of three elements. *ind_num*[*i*] (*i* is X, Y or Z) stores the index number of the potentially visible *i*-face of the octant and is determined by

$$ind\_num[i] = ind\_tab[depth] + card\_num[i] \tag{4.1}$$

where *ind_tab* is a pre-computed table with *ind_tab*[*depth*] giving the index number of the left-most node at depth *depth* in a complete quadtree. For example, nodes 0, 1 and 5 in Figure 22 are the left-most nodes at depth 0, 1 and 2, respectively. Figure 24 defines *ind_tab* for n = 8. Once *ind_num*[*i*] is found, the actual location

depth	0	1	2	3	4	5	6	7	8
	0	1	5	21	85	341	1365	5461	21845

Figure 24: Definition of ind_tab (n = 8).

of the node in quadtree-i is determined by

$$(ind\_num[i]\ div\ 4, ind\_num[i]\ mod\ 4) \tag{4.2}$$

where $ind\_num[i]\ div\ 4$ determines the array index of the byte that contains the node and $ind\_num[i]\ mod\ 4$ determines the location of the node inside the byte. Formula (4.2) completes step 1 of the procedure.

ind_num_son is a two-dimensional array of 12 elements. $ind\_num\_son[i][j]$ (i is X, Y or Z, and j is 0, 1, 2 or 3) stores the index number of the jth child node of the potentially visible i-face of the octant. $ind\_num\_son[i][j]$ is determined by

$$ind\_num\_son[i][j] = ind\_tab[depth + 1] + 4 \times card\_num[i] + j \tag{4.3}$$

$ind\_num\_son[i][j]$ is needed in step 3 of the procedure. If node $ind\_num[i]$ is W (or B), then all its 4 child nodes are set to W (or B) and the node is set to G. Otherwise ($ind\_num[i]$ is G), nothing needs to be done.

card_num_son is a one-dimensional array of three elements. $card\_num\_son[i]$ (i is X, Y or Z) stores the cardinal number of the potentially visible i-face of a child octant of the octant. *card_num_son* is used as a parameter (*card_num*) to invoke procedure "Process a G Octant" or "Process a W or B Octant" for processing a sub-octant of the octant in step 4 of the procedure. The cardinal number of the

oct \ i	X	Y	Z
0	1	2	0
1	1	3	1
2	3	2	2
3	3	3	3
4	0	0	0
5	0	1	1
6	2	0	2
7	2	1	3

Figure 25: Definition of quad_tab.

potentially visible i-face of the sub-octant, *oct* (*oct* is 0, 1, 2, 3, 4, 5, 6 or 7), of the octant is determined by

$$card\_num\_son[i] = 4 \times card\_num[i] + quad\_tab[oct][i] \qquad (4.4)$$

where *quad_tab* is defined in Figure 25.

Step 5 of procedure "Process a G Octant" needs *ind_num* and *ind_num_son*. If the four child nodes of node *ind_num*[i] are all B, then the node is set to B. This compaction is necessary because otherwise blocked faces of octants to be processed later may be mistakenly considered as un-blocked faces. Although this has no effect on the correctness of the final image (refer to step 5 in procedure "Scan-Convert a Face", where a check is made before painting a pixel to ensure that only "un-painted" pixels are painted), it may cause some redundant computation.

Similar to the procedure "Process a G Octant", procedure "Process a W or B Octant" also needs two parameters, *depth* and *card_num*. It needs one local variable *ind_num*. Step 4 of the procedure sets node *ind_num*[*i*] (*i* is X, Y or Z) to the color of the octant to update the quadtree-*i*.

Finally, it should be noted that formulas (4.1)-(4.4) involve only array indexing and integer addition, shift and modular operations. Therefore, the operation on the blocking quadtrees can be performed very efficiently.

Scan-Conversion of a Face.

The following techniques are used to make procedure "Scan-Convert a Face" as efficient as possible. First, since all octant faces perpendicular to a coordinate axis of the octree space have the same face normal vector in the image space, only three face normals (in the image space) are calculated at the beginning of the algorithm for faces perpendicular to the X-, Y- and Z-axes, respectively. Therefore, the normal vector of a face to be scan-converted need not to be recomputed and the plane equation of the face can be easily found by using the normal vector and the image space coordinates of a vertex of the face. Second, instead of using the standard edge-table technique [11], the simple method shown in the algorithm is used to scan-convert an octant face because it is a convex polygon in the image space. This is faster because keeping track of *left* and *right* edges is more efficient than updating the edge-table. Third, intersections of scanlines and face edges are calculated incrementally involving only additions by employing edge coherence [11]. And lastly, z values of the face are also calculated incrementally involving only additions by employing scanline coherence [11].

Arbitrary Viewing Directions.

So far the algorithm has been described based on the assumption that a given viewing direction makes three faces of an octant potentially visible. To handle an arbitrary viewing direction, two global variables, *axis* and *axis_nums*, are needed. *axis* is a one-dimensional array of three elements. *axis_nums* is an integer. They are initialized at the beginning of the algorithm according to the given viewing direction as shown in Figure 26, where *viewing direction* is defined as follows:

- XYZ: three faces of an octant are potentially visible.
- XY: an X-face and a Y-face of an octant are potentially visible.
- XZ: an X-face and a Z-face of an octant are potentially visible.
- YZ: a Y-face and a Z-face of an octant are potentially visible.
- X: an X-face of an octant is potentially visible.
- Y: a Y-face of an octant is potentially visible.
- Z: a Z-face of an octant is potentially visible.

Obviously, only the corresponding *axis_nums* blocking bitmaps need to be operated. The correct formulas can be obtained by substituting $axis[ii]$ for all i's in formulas (4.1)-(4.4), where $0 \leq ii \leq axis\_nums - 1$.

4.4 Complexity Analysis

Space complexity.

The space taken by the blocking quadtrees is:

$(1 + 4 + 4^2 + ... + 4^n) \times 3 \times 2/8 =$

viewing direction	axis[0]	axis[1]	axis[2]	axis_nums
XYZ	X	Y	Z	3
XY	X	Y	-	2
XZ	X	Z	-	2
YZ	Y	Z	-	2
X	X	-	-	1
Y	Y	-	-	1
Z	Z	-	-	1

Figure 26: Determination of axis and axis_nums.

$(4^{n+1} - 1)/4 \doteq$

4^n (bytes)

where n is the resolution of the octree space.

Time complexity.

For an octree to be rendered and a given viewing direction, assume that

N_W: number of W octants of the octree.

N_B: number of B octants of the octree.

N_G: number of G octants of the octree.

N_{b-W}: number of blocked W octants of the octree.

N_{nb-W}: number of non-blocked W octants of the octree.

N_{b-B}: number of blocked B octants of the octree.

N_{nb-B}: number of non-blocked B octants of the octree.

N_{b-G}: number of blocked G octants of the octree.

N_{nb-G}: number of non-blocked G octants of the octree.

N_{vb-W}: number of W octants of the octree that are visited and found blocked.

N_{vb-B}: number of B octants of the octree that are visited and found blocked.

N_{vb-G}: number of G octants of the octree that are visited and found blocked.

Then the time spent to render the octree is proportional to (note that formulas (4.1)-(4.4) involve only constant-time calculations):

$$N_{nb-W} + N_{vb-W} + N_{nb-G} + N_{vb-G} + N_{nb-B} + N_{vb-B} + T(nb - B)$$

where $N_{nb-W} + N_{vb-W}$ is the time spent to process W octants, $N_{nb-G} + N_{vb-G}$ is the time spent to process G octants, $N_{nb-B} + N_{vb-B}$ is the time spent to process B octants before rendering them, and $T(nb - B)$ is the time spent to render all non-blocked B octants.

The non-blocked B octants are rendered by scan-converting their un-blocked faces into the z-buffer. Assume that

$N_{nb-face}$: number of un-blocked faces of all non-blocked B octants.

$N_{scanline}$: number of accumulated intersecting scan lines.

N_{pixel}: number of accumulated pixels to be checked and painted in the z-buffer.

(**Explanation of $N_{scanline}$ and N_{pixel}.** Suppose that there are totally two faces to be rendered, and that face a has 5 intersecting scan lines and face b has 7, then $N_{scanline}$ in this case is 12. Note that face a and b may have some common scan lines. Similarly, if 30 pixels are to be checked and painted for face a and 40 for face b, then N_{pixel} is 70).

Then $T(nb - B)$ is proportional to:

$$N_{nb-face} + N_{scanline} + N_{pixel}$$

where $N_{nb-face}$ is the time spent for obtaining plane equations of faces, $N_{scanline}$ is the time spent for calculating intersections, and N_{pixel} is the time spent for calculating depth values.

Therefore, the worst case time complexity of the algorithm is:

$O(T_{WG} + T_B)$

where T_{WG} (or $N_{nb-W} + N_{vb-W} + N_{nb-G} + N_{vb-G}$) is the time to process W and G octants, and T_B (or $N_{nb-B} + N_{vb-B} + N_{nb-face} + N_{scanline} + N_{pixel}$) is the time to process B octants.

Time complexity of a BTF algorithm.

For comparison purposes, the time complexity of the traditional BTF algorithm [25] is:

$O(N_G + N_B + M_{face} + M_{scanline} + M_{pixel})$

where M_{face} is the number of all potentially visible faces of all B octants, $M_{scanline}$ is the number of accumulated intersecting scan lines, and M_{pixel} is the number of accumulated pixels to be painted in the z-buffer. Similarly, it can also be written as:

$O(T'_G + T'_B)$

where T'_G (or N_G) is the time to process all G octants, and T'_B (or $N_B + M_{face} + M_{scanline} + M_{pixel}$) is the time to process all B octants.

4.5 Experimental Results

The proposed algorithm has been implemented using C on a Sun-3/60 and a Sun-SparC. For comparison purposes, the traditional BTF algorithm [25] was also implemented. Two example objects have been used to test the two algorithms. The first object is artificially created (typed in) and represented as an octree with a resolution of n = 7. The octree consists of 32 W octants, 32 B octants and 9 G octants. The second object is part of a human skull obtained from 24 (256 × 256) CT slices after preprocessing (linear interpolation and thresholding). It is represented as an octree with a resolution of n = 8 that consists of 348,996 W octants, 285,954 B octants and 90,707 G octants. Figure 27(a) shows the statistics of running the two programs using the artificial object on a Sun-3/60 and Figure 29(a) shows the corresponding depth shaded image produced. Figure 27(b) shows the statistics of running the two programs using the human skull on a Sun SparC and Figure 29(c)-(f) show the corresponding depth shaded images produced. Note that the statistics are obtained by the UNIX utility program *gprof*.

From Figure 27 it is clear that the major reason for the significant speed-up of the new FTB algorithm over the conventional BTF one is the drastic reduction of the time to process B octants. This is achieved because many blocked B octants and B octant faces are simply discarded in the FTB algorithm. In other words, many floating point operations needed to process B octants (3D coordinate transformation and polygon scan-conversion) are eliminated. This is effective since only simple integer operations are involved to process W and G octants in the FTB algorithm. Therefore, the overall result is that the overhead of the FTB algorithm is much smaller than its speed gain.

To verify the speed-up from the time complexity, Figure 28 is produced for the object of human skull. For the face view, the result in Figure 28 is very consistent with that in Figure 27(b). For other cases, since more than one quadtrees have to be processed, the constant factors are greater than 1, which explains the differences between Figure 27(b) and Figure 28.

	VPN	No. of B octant-faces scan-converted	No. of B octants visited	No. of W octants visited	No. of G octants visited	Time to process B octants (sec)	Total time (sec) (excluding I/O)	Time reduction of FTB over BTF (%)
FTB	(1 -1 1)	48	32	32	9	4.87	5.15	52.6
BTF	(1 -1 1)	96	32	-	9	10.63	10.87	-

(a)

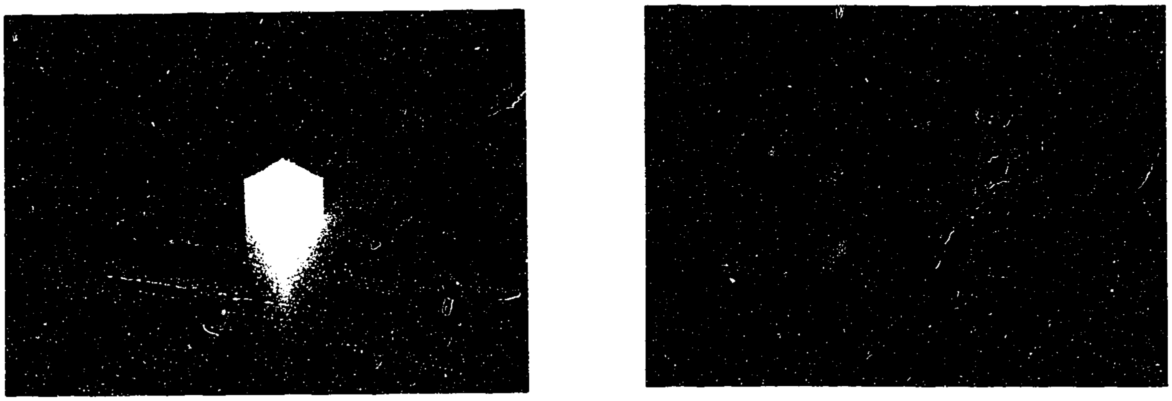
	VPN	No. of B octant-faces scan-converted	No. of B octants visited	No. of W octants visited	No. of G octants visited	Time to process B octants (sec)	Total time (sec) (excluding I/O)	Time reduction of FTB over BTF (%)
FTB	(1 -2 1)	204728	248186	319197	84986	76.89	112.83	55.9
BTF	(1 -2 1)	857862	285954	-	90707	247.86	255.60	-
FTB	(-1 -2 1)	202692	244345	314453	83395	75.20	111.48	56.5
BTF	(-1 -2 1)	857862	285954	-	90707	248.44	256.16	-
FTB	(1 -1 0)	118269	153722	215954	60765	37.18	57.24	63.2
BTF	(1 -1 0)	571908	285954	-	90707	147.94	155.57	-
FTB	(0 -1 0)	32943	80222	121396	35639	10.72	19.42	79.2
BTF	(0 -1 0)	285954	285954	-	90707	85.33	93.30	-

(b)

Figure 27: Results for (a) H, and (b) Human skull.

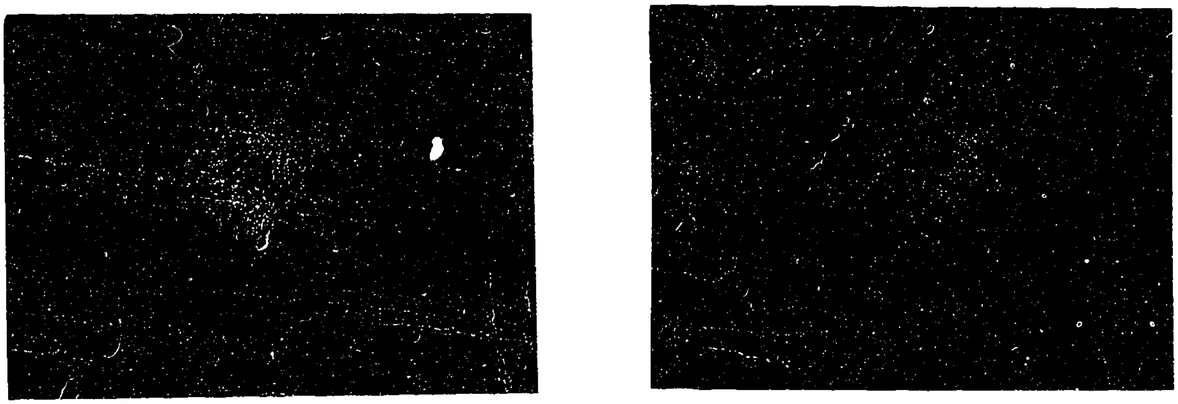
	VPN	No. of scanlines accumulated	No. of pixels accumulated	Total No. of pixels, scanlines, B octant faces, and octants processed	No. reduction of FTB over BTF (%)
FTB	(1 -2 1)	547118	1188518	2592733	70.5
BTF	(1 -2 1)	2352103	5213049	8799675	-
FTB	(-1 -2 1)	540965	1174078	2559928	70.9
BTF	(-1 -2 1)	2352225	5213246	8799994	-
FTB	(1 -1 0)	241407	516725	1306842	73.2
BTF	(1 -1 0)	1195784	2735609	4879962	-
FTB	(0 -1 0)	36456	82038	388694	79.1
BTF	(0 -1 0)	344781	850416	1857812	-

Figure 28: Theoretical results for the human skull.



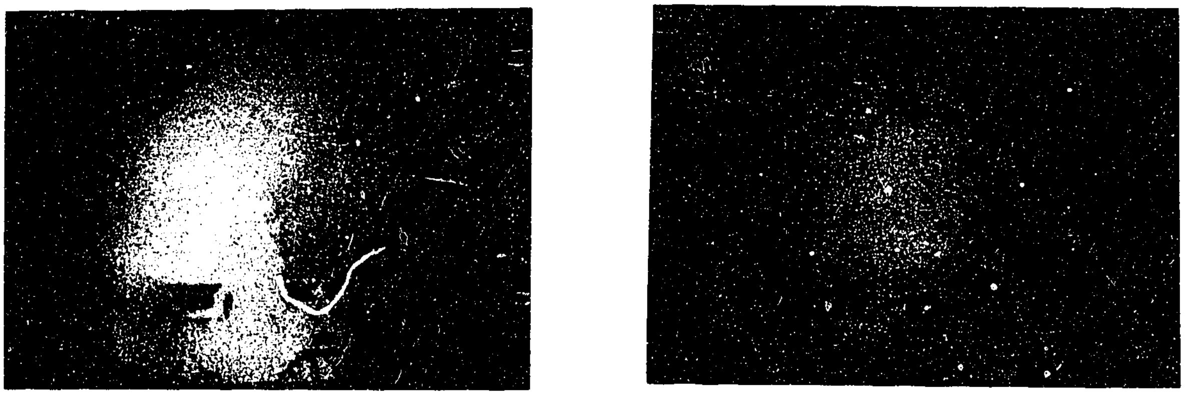
(a) VPN = (1, -1, 1)

(b) VPN = (1, -3, 1)



(c) VPN = (1, -2, 1)

(d) VPN = (-1, -2, 1)



(e) VPN = (1, -1, 0)

(f) VPN = (0, -1, 0)

Figure 29: Depth shaded images.

Chapter 5

Conclusion

Existing software techniques for displaying 3D voxel-based objects are reviewed in this thesis. To improve rendering speed, a new algorithm is proposed, analyzed and implemented for displaying 3D voxel-based binary objects encoded via octrees. The performance of the algorithm is compared with a conventional octree display algorithm [25] from both theoretical and experimental viewpoints. The new algorithm achieves a 55-79% time reduction over the conventional algorithm in rendering a 256 × 256 × 256 medical image (human skull) acquired by a CT (Computed Tomography) scanner. The timing result is also verified by the theoretical analysis.

The proposed octree display algorithm requires extra memory space to store the blocking quadtrees in order to perform the octant visibility test. The space overhead is acceptable for two reasons. First, in general, the storage required by an octree is far greater than the corresponding blocking quadtrees. For example, the octree for the human skull requires about 24 MB, while the blocking quadtrees require only 64 KB or less than 0.3% of the octree storage. Of course, when implemented on a Sun-SparC, only a portion of the octree can reside in the main memory for processing. Swapping between memory and disk has to be performed. Second, for a Sun-SparC up to 8MB of main memory can be used for the user's program, of which only 64KB or 0.8% stores the blocking quadtrees.

Future research could proceed in the following directions. First, the algorithm for displaying linear octrees based on the concept of blocking quadtrees can be designed and implemented. It would be interesting to compare the timing results of that algorithm with the conventional linear octree display algorithm [16] and the results presented in this thesis. Second, interactive display techniques for octree encoded objects have not been studied in the past. It would be very interesting to investigate the impact of the concept of blocking quadtrees on the design of an efficient interactive display system via octree representations. Third, comparison of the proposed algorithm with other volume rendering methods would be a very interesting task from both theoretical and experimental viewpoints. Finally, it is possible to investigate the design of a special architecture based on the concept of blocking quadtrees to achieve an even faster rendering speed.

Bibliography

[1] E. Artzy, G. Frieder, and G.T. Herman, "The Theory, Design, Implementation and Evaluation of a Three-Dimensional Surface Detection Algorithm", *Computer Graphics and Image Processing*, Vol. 15, pp. 1-24, 1981.

[2] H. H. Atkinson, I. Gargantini, and M. V. S. Ramanath, "Determination of the 3D Border by Repeated Elimination of Internal Surfaces", *Computing*, Vol. 32, pp. 279-295, 1984.

[3] H. H. Atkinson, I. Gargantini, and M. V. S. Ramanath, "Improvements to a Recent 3D-Border Algorithm", *Pattern Recognition*, Vol. 18, pp. 215-226, 1985.

[4] L. J. Brewster, S. S. Trivedi, H. K. Tuy, and J. K. Udupa, "Interactive Surgical Planning", *IEEE CG&A*, Vol. 4, pp. 31-40, March 1984.

[5] J. D. Boissonnat, "Shape Reconstruction from Planar Cross Sections", *Computer Vision, Graphics, and Image Processing*, Vol. 44, pp. 1-29, 1988.

[6] L. S. Chen, G. T. Herman, R. A. Reynolds, and J. K. Udupa, "Surface Shading in the Cuberille Environment", *IEEE CG&A*, Vol. 5, 33-43, October 1985.

[7] L. S. Chen, G. T. Herman, H. M. Hung, H. Levkowitz, S. S. Trivedi, and J. K. Udupa, "Interactive Manipulation of Three-Dimensional Data via a Two-Dimensional Device", *Optical Engineering*, Vol. 24, pp. 893-900, 1985.

[8] H. H. Chen and T. S. Huang, "A Survey of Construction and Manipulation of Octrees", *Computer Vision, Graphics and Image Processing*, Vol. 43, pp. 409-431, 1988.

[9] H. N. Christiansen and T. W. Sederberg, "Conversion of Complex Contour Line Definitions into Polygonal Element Mosaics", *Computer Graphics (Siggraph)*, Vol. 12, pp. 187-192, 1978.

[10] L. J. Doctor and J. G. Torborg, "Display Techniques for Octree-Encoded Objects", *IEEE CG&A*, Vol. 1, pp. 29-38, January 1981.

[11] J. D. Foley and A. VanDam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982.

[12] G. Frieder, D. Gordon, R. A. Reynolds, "Back-to-Front Display of Voxel-Based Objects", *IEEE CG&A*, Vol. 5, pp. 52-60, January 1985.

[13] H. Fuchs, Z. Kedem, and S. P. Uselton, "Optimal Surface Reconstruction from Planar Contours", *Commun. ACM*, Vol. 20, No. 10, pp. 693-702, 1977.

[14] S. Ganapathy and T. G. Dennehy, "A New Triangulation Method for Planar Contours", *Computer Graphics (Siggraph)*, Vol. 16, pp. 69-75, 1982.

[15] I. Gargantini, "Linear Octtrees for Fast Processing of Three-Dimensional Objects", *Computer Graphics and Image Processing*, Vol. 20, pp. 365-374, 1982.

[16] I. Gargantini, T. R. Walsh and O. L. Wu, "Viewing Transformation of Voxel-Based Objects via Linear Octrees", *IEEE CG&A*, Vol. 6, pp. 12-21, October 1986.

[17] S. M. Goldwasser, R. A. Reynolds, D. A. Talton, and E. S. Walsh, "Techniques for the Rapid Display and Manipulation of 3-D Biomedical Data", submitted to: *Computerized Radiology*, 1987.

[18] S. M. Goldwasser and R. A. Reynolds, "Real-Time Display and Manipulation of 3-D Medical Objects: The Voxel Processor Architecture", *Computer Vision, Graphics, and Image Processing*, Vol. 39, pp. 1-27, 1987.

[19] B. W. Heal, "Hidden Octree Node Removal as a Pipeline Process", *Computer Graphics Forum*, Vol. 8, No. 3, pp. 199-206, 1989.

[20] A. Kaufman and R. Bakalash, "Memory and Processing Architecture for 3D Voxel-Based Imagery", *IEEE CG&A*, Vol. 8, pp. 10-23, November 1988.

[21] E. Keppel, "Approximating Complex Surfaces by Triangulation of Contour Lines", *IBM J. Res. Develop.*, Vol. 19, pp. 2-11, 1975.

[22] H. K. Liu, "Two- and Three-Dimensional Boundary Detection", *Computer Graphics and Image Processing*, Vol. 6, pp. 123-134, 1977.

[23] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", *Computer Graphics (Siggraph)*, Vol. 21, pp. 163-169, 1987.

[24] X. Mao, T. L. Kunii, I. Fujishiro, and T. Noma, "Hierarchical Representations of 2D/3D Gray-Scale Images and Their 2D/3D Two-Way Conversion", *IEEE CG&A*, Vol. 7, pp. 37-44, December 1987.

[25] D. J. Meagher, "Geometric Modeling Using Octree Encoding", *Computer Graphics and Image Processing*, Vol. 19, pp. 129-147, 1982.

[26] D. J. Meagher, "Efficient Synthetic Image Generation of Arbitrary 3-D Objects", *Proceedings of the IEEE Computer Society Conference on Pattern Recognition and Image Processing*, pp. 473-478, Hawaii, USA, June 1982.

[27] R. A. Reynolds, "Some Architectures for Real-Time Display of Three-Dimensional Objects: A Comparative Survey", *Tutorial* delivered at: 17th Hawaii Int. Conf. on System Sciences, Honolulu, Jan. 1984.

[28] R. A. Reynolds, *Fast Methods for 3D Display of Medical Objects*, Ph.D Dissertation, Dept. of Computer and Information Science, University of Pennsylvania, May 1985.

[29] R. A. Reynolds, D. Gordon, and L. S. Chen, "A Dynamic Screen Technique for Shaded Graphics Display of Slice-Represented Objects", *Computer Vision, Graphics, and Image Processing*, Vol. 38, pp. 275-298, 1987.

[30] R. A. Reynolds, P. Decuypere, S. M. Goldwasser, D. A. Talton, and E. S. Walsh, "Realistic Presentation of Three-Dimensional Medical Datasets", *Graphics Interface '88*, pp. 71-77, Edmonton, Alberta, Canada, June 1988.

[31] H. Samet, "The Quadtree and Related Hierarchical Data Structures", *ACM Computing Surveys*, Vol. 16, pp. 187-260, 1984.

[32] Y. W. Tam, *Three Dimensional Display of Medical Images*, M.Sc thesis, Dept. of Computing Science, University of Alberta, 1988.

[33] J. K. Udupa, "Interactive Segmentation and Boundary Surface Formation for 3-D Digital Images", *Computer Graphics and Image Processing*, Vol. 18, pp. 213-235, 1982.

[34] J. K. Udupa, S. N. Srihari, and G. T. Herman, "Boundary Detection in Multidimensions", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-4, No. 1, pp. 41-50, 1982.

[35] J. K. Udupa, "Display of 3D Information in Discrete 3D Scenes Produced by Computerized Tomography", *Proceedings of the IEEE*, Vol. 71, No. 3, pp. 420-431, March 1983.

[36] J. K. Udupa and D. Odhner, "Display of Medical Objects and Their Interactive Manipulation", *Graphics Interface '89*, pp. 40-46, London, Ontario, Canada, June 1989.

[37] M. W. Vannier, J. L Marsh, and J.O. Warren, "Three Dimensional Computer Graphics for Craniofacial Surgical Planning and Evaluation", *Computer Graphics (Siggraph)*, Vol. 17, No. 3, pp. 263-273, July 1983.

[38] K. Yamaguchi, T. L. Kunii, and K. Fujimura, "Octree-Related Data Structures and Algorithms", *IEEE CG&A*, Vol. 4, pp. 53-59, January 1984.