Fast Display of Octree Representations of 3D Objects[1]

Jiangong Zhao[2] and Wayne A. Davis

Department of Computing Science
University of Alberta
Edmonton, Alberta T6G 2H1

ABSTRACT

A novel algorithm is proposed for displaying 3D voxel-based binary objects encoded as octrees. It traverses an octree recursively in a front-to-back (FTB) order determined by a given viewing direction and produces a depth image by scan-converting the visible portion of faces of black octants encountered during the traversal. The algorithm is efficient because many obscured nodes are never visited and many obscured faces of partially visible black octants are not processed during the scan-conversion phase. The key to the algorithm is a new technique termed blocking quadtrees, that enables the node visibility test to be performed 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 [3]. A 55-79% time reduction is achieved for a 3D 256^3 medical image that requires a reasonable amount of extra memory space for storing the blocking quadtrees (64 KB for a 256^3 image).

1. Introduction

Octrees have been used successfully to represent 3D voxel-based binary objects. One active research area has been to develop efficient algorithms to display octree encoded objects on a two-dimensional screen. Basically speaking, two approaches have been studied, the BTF and FTB approaches. Both methods are based on pre-sorting of the octree representation and a recursive octree traversal in a pre-determined order.

In the BTF approach proposed by Meagher [3], an octree to be rendered is recursively traversed according to a BTF order determined by a given viewing direction. If a black octant is encountered during the traversal, it is projected onto the display screen. If a gray octant is encountered, it is traversed further according to the same BTF order. White octants need not be processed. Obviously, when octree traversal is complete, the final screen image has the correct hidden surface relationship. Gargantini et al. [2] also used the same approach to display linear octrees.

Conversely, in a FTB approach, an octree is recursively traversed in a FTB order. A valid FTB order can be determined similarly to a BTF order. If a black octant is encountered during traversal, a check must be made to see whether the octant obscured by other black octants has been processed earlier. If it has been completely obscured, it is not rendered and discarded; otherwise, the non-obscured part of the octant is projected onto the screen. If a gray octant is encountered and has been completely obscured, the node and all its descendants are discarded and not processed any more; otherwise, it is further traversed in the same FTB order. Clearly, the final image obtained, when the traversal of the octree is complete, also has the correct hidden surface relation.

A FTB approach could be faster than its counterpart because many obscured octants are not visited nor projected. However, the overhead to determine the visibility of octants during octree traversal could well exceed the speed gain if the visibility test can't be done efficiently. Meagher [4] proposed to use a quadtree to represent the display screen and then perform polygon intersection tests to determine octant visibility by intersecting the projection of octants with quadrants of the quadtree. That algorithm can handle arbitrary viewing directions, but the speed gain over the simple BTF approach [3] cannot be justified because the polygon intersection tests are expensive and need to be performed extensively (no time complexity analysis and experimental statistics of computation time are given in [4]). Heal [5] adopted a similar approach, but restricted the viewing directions to be orthogonal to the octree space, thus making the projection of octants coincide with the quadrants of the quadtree and eliminating the need for expensive polygon intersection tests. However, the scope of Heal's algorithm is limited because of the limitations on the viewing directions.

In this paper, a new FTB algorithm is proposed to display octrees without any restrictions on the viewing direction. It traverses an octree recursively in a FTB order determined by a given viewing direction and produces a depth image by scan-converting the visible portion of faces of black octants encountered during the traversal. The algorithm is efficient because many

[1] This research was supported in part by NSERC Grant OGP 7634.

[2] Mr. Zhao is currently with Bell-Northern Research in Ottawa.

obscured nodes are never visited and many obscured faces of partially visible black octants are not processed during the scan-conversion phase at all. The key to the algorithm is a new concept called blocking quadtrees, that makes the octant visibility test very efficient and thus minimizes the overhead cost.

2. Background

An octree space is modeled as a cubic region consisting of unit cubes or voxels. 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. Octree representations have been extensively reviewed elsewhere [6] and will not be included.

3D Coordinate Transformation

Suppose that the octree space is described by a 3D right-handed XYZ coordinate system and that the direction from the origin to the viewer is given by the vector (x1, y1, z1) called the view plane normal (VPN). (x1, y1, z1) is viewer's position. To project a point onto the view plane, perpendicular to the VPN, or the screen, the following transformations are needed [2]: 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, now in the Y-Z plane.

So far the rotations are around the origin. However, the origin is conventionally located in one corner of the octree space, and it is desirable to rotate the object around the center of the octree space so that it stays in the octree space. The necessary transformations are easily obtained and will not be presented. Note that the projection used is parallel orthographic [1].

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. 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 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).

3. Blocking Quadtrees

An octant has six faces, 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. 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 could be completely visible (if no other B octants obscure the face), partially visible (other B octants obscure the face partially), or completely invisible (if other B octants obscure the face completely) from the viewing direction.

For convenience, 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 to handle arbitrary viewing directions.

An octant has at most six neighboring octants of the same size that share a face with it. These neighboring octants sharing potentially visible faces of an octant are called front neighboring octants of the octant, and the rest back neighboring octants of the octant.

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 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. It should be noted that a face without a neighboring octant is always 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 Sec. 2).

3.1 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 and all descendants (if it is a G octant). Otherwise, the octant is not a blocked octant and: 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.

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 the 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 unblocked 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 these quadtree nodes are all B, the octant is blocked and is discarded with all its descendants, if any. Otherwise: if it 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 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 its eight sub-octants of the octant recursively according to the FTB order, and 3) change each corresponding quadtree node to B if the quadtree has four B son nodes.

3.2 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 the proposed format called blocking bitmaps. 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 the 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. 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.

The major advantage of representing 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 will be performed in an extremely efficient manner. The mapping mechanism will be described in detail in Sec. 4, where the display algorithm is presented.

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 Sec. 5 for the space complexity analysis).

Alternatively, regular quadtrees with pointers or linear quadtrees [6] 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 extra computation 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 quadtree 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, that 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), that 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. 2) Updating a linear quadtree involves allocating and deallocating memory space and linked list pointer re-arrangement, consumimg more computation time than simply writing integer values to array elements.

4. 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 [7] in a post-processing step. In the following, the algorithm is first presented, then some of the details are explained further. To avoid confusion, octree nodes will be called octants and blocking quadtrees nodes will be called nodes.

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.

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 = 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 node in a blocking quadtree. 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 = X, Y or Z) stores the index number of the potentially visible i-face of the octant and is determined by:

$$ind_num[i] = ind_tab[depth] + card_num[i], \quad (1)$$

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. Once *ind_num[i]* is found, the actual location of the node in quadtree-i is determined by

$$(ind_num[i]\ div\ 4,\ ind_num[i]\ mod\ 4), \quad (2)$$

where *ind_num[i] div 4* determines the array index of the byte that contains the node and *ind_num[i] mod 4* the location of the node inside the byte. Formula (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 j-th child node of the potentially visible i-face of the octant. Where:

$$ind_num_son[i][j] = ind_tab[depth+1] + 4 \times card_num[i] + j. \quad (3)$$

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

$$card_num_son[i] = 4 \times card_num[i] + quad_tab[oct][i], \quad (4)$$

where *quad_tab* is defined in Table I.

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 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 = X, Y or Z) to the color of the octant to update blocking-quadtree- i. Finally, it should be noted that formulas (1) - (4) involve only array indexing and integer addition, shift and modular operations. Therefore, the operation on the blocking quadtrees can be performed 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 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 [1], 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, intersection of scanlines and face edges are calculated incrementally involving only additions by

employing edge coherence [1]. Lastly, z values of the face are also calculated incrementally involving only additions by employing scanline coherence [1].

Arbitrary Viewing Direction

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, 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 used. The correct formulas can be obtained by substituting *axis[ii]* for all i's in formulas (1)-(4), where $0 \leq ii \leq axis_nums$.

5. Complexity Analysis

Space Complexity

The space taken by the blocking quadtrees is:
(1+ 4 + 42 + ... + 4n) × 3 × 2/8 =
(4n+1 - 1)/4 = 4n (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:

NW: number of W octants of the octree.
NB: number of B octants of the octree.
NG: number of G octants of the octree.
Nb-W: number of blocked W octants of the octree.
Nnb-W: number of un-blocked W octants of the octree.
Nb-B: number of blocked B octants of the octree.
Nnb-B: number of un-blocked B octants of the octree.
Nb-G: number of blocked G octants of the octree.
Nnb-G: number of un-blocked G octants of the octree.
Nvb-W: number of W octants of the octree that are visited and found blocked.
Nvb-B: number of B octants of the octree that are visited and found blocked.
Nvb-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 (1)-(4) involve only constant-time calculations):

$$Nnb\text{-}W + Nvb\text{-}W + Nnb\text{-}G + Nvb\text{-}G + Nnb\text{-}B + Nvb\text{-}B + T(nb\text{-}B),$$

where *Nnb-W + Nvb-W* is the time spent to process W octants, *Nnb-G + Nvb-G* is the time spent to process G octants, *Nnb-B + Nvb-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 to the z-buffer. Assume that:

Nnb-face: number of un-blocked faces of all un-blocked B octants.
Nscanline: number of accumulated intersecting scan lines.
Npixel: number of accumulated pixels to be checked and painted in the z-buffer.

Explanation of *Nscanline* and *Npixel.* Supposing there are two faces to be rendered, and that face a has 5 intersecting scan lines and face b has 7, then *Nscanline* in this case is 12. Note that face a and face 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 *Npixel* is 70.

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

$$Nnb\text{-}face + Nscanline + Npixel,$$

where *Nnb-face* is the time spent for obtaining plane equations of faces, *Nscanline* is the time spent for calculating intersections, and *Npixel* is the time spent for calculating depth values.

Therefore, the worst time complexity for the algorithm is:

$$O(TWG + TB),$$

where *TWG (or Nnb-W + Nvb-W + Nnb-G + Nvb-G)* is the time to process W and G octants, and *TB (or Nnb-B + Nvb-B + Nnb-face + Nscanline + Npixel)* is the time to process B octants.

5.1 Time complexity of BTF algorithm

For comparison purposes, the time complexity of the traditional BTF algorithm [3] is:

$$O(NG + NB + Mface + Mscanline + Mpixel),$$

where *Mface* is the number of all potentially visible faces of all B octants, *Mscanline* is number of accumulated intersecting scan lines, and *Mpixel* is the number of accumulated pixels to be painted in z-buffer. Similarly, it can also be written as:

$$O(T'G + T'B),$$

where *T'G (or NG)* is the time to process all G octants, and *T'B (or NB = Mface + Mscanline + Mpixel)* is the time to process all B octants.

6. 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 [3] was also implemented. Two example objects have been used to test 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 x 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. Table II(a) shows the statistics of running the two programs using the artificial object on a Sun-3/60 and Plate 1(a) shows the corresponding depth shaded image produced. Table II(b) shows the statistics of running the two programs using the human skull on a Sun-SparC and Plates 1(b)-(e)

show the corresponding depth shaded images produced. Note that the statistics are obtained by the UNIX utility program gprof.

From Table II 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 are eliminated (3D coordinate transformation and polygon scan-conversion). 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, Table III is produced for the object of human skull. For the face view, the result in Table III is very consistent with that in Table II(b). For other cases, since more than one quadtree has to be processed, the constant factors are greater than 1, which explains the differences between Table II(b) and Table III.

7. Conclusion

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 [3] from both theoretical and experimental viewpoints. The new algorithm achieves a 55-79% time reduction over the conventional algorithm in rendering a 256 x 256 x 256 medical image (human skull) acquired by a CT (Computed Tomography) scanner. The timing result is also verified by 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 8 MB of main memory can be used for the user's program, of which only 64 KB 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 [2] and the results presented in this paper. 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, comparisons 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.

Acknowledgements

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

References

1. J. D. Foley & A. VanDam, Fundamentals of Interactive Computer Graphics, Addison-Wesley, Reading, Mass., 1982.

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

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

4. D. J. Meagher, "Efficient Synthetic Image Generation of Arbitrary 3-D Objects", *Proc of the IEEE Conference on PRIP*, pp. 473-478, Hawaii, Jun 1982.

5. B. W. Heal, "Hidden Octree Node Removal as a Pipeline Process", *Computer Graphics Forum*, Vol. 8, pp. 199-206, 1989.

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

7. Y. W. Tam & W. A. Davis, "Display of 3D Medical Images", *Proc. Graphics Interface '88*, Edmonton, pp. 78-86, Jun 1988.

Table I. Determination of *quad_tab*.

	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

Table II a) Timing Results for H.

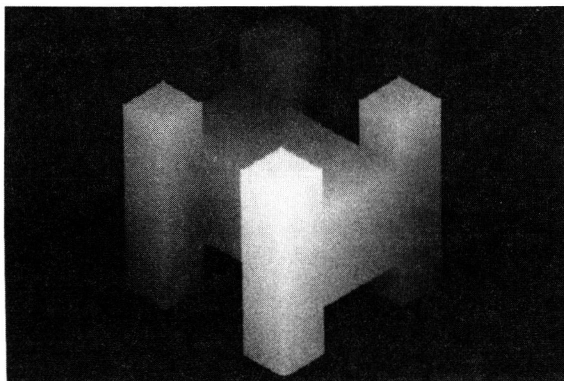
	VPN	# of B octant-faces scan-converted	# of B octants visited	# of W octants visited	# of G octants visited	Time to process B octants (sec)	Total time (sec) (excluding I/O)	(%) reduction FTB over BTF
FTB	1 -1 1	48	32	32	9	4.87	5.15	52.6
BTF	1 -1 1	96	32	-	9	10.63	10.87	-

Table II b) Timing Results for Skull.

	VPN	# of B octant-faces scan-converted	# of B octants visited	# of W octants visited	# of G octants visited	Time to process B octants (sec)	Total time (sec) (excluding I/O)	(%) reduction 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	857967	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	85.83	93.30	-

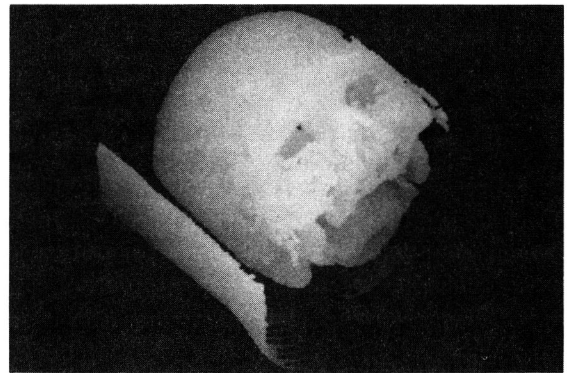
Table III Theoretical Results for Skull

	VPN	# of scanlines accumulated	# of pixels accumulated	Total # of pixels scanlines, octant faces & octants processeed	% 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	-

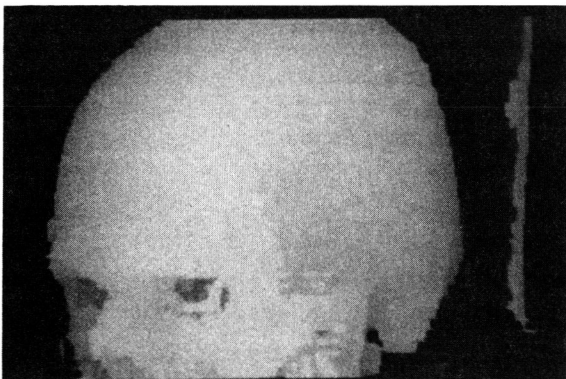


a) VPN = 1, -1, 1

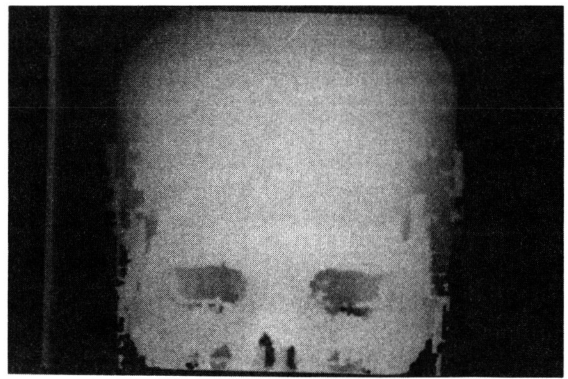
b) VPN - 1, -2 1



c) VPN = -1, -2, 1



d) VPN = 1, -1, 0



e) VPN = 0, -1 0

Plate 1. Depth Shaded Images.