OCTREE REPRESENTATION AND DISPLAY OF THREE DIMENSIONAL OBJECTS

By

RAMESH L. PARMAR

Bachelor of Technology

in Chemical Engineering

Bharathidasan University

Tiruchirapalli, India

1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1989

OCTREE REPRESENTATION AND DISPLAY

OF THREE DIMENSIONAL OBJECTS

Thesis Approved:

J P Chandler
Thesis Advisor

Keith A. Teague

William David Miller

Norman N. Durham
Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to Dr. John P. Chandler for his advice and support throughout my graduate program. Without Dr. William D. Miller's intuitive ideas and suggestions, this thesis would not have been what it is now. My sincere thanks to him. I also thank Dr. Keith A. Teague for being on my graduate committee and giving me access to the hardware needed. I extend my regards to Dr. George E. Hedrick for his guidance as the Department Head.

My brothers, Goutham L. Parmar, and Gulraj L. Parmar encouraged me throughout and urged me to seek the best. I owe my respects to them, my sisters, my brothers-in-law, and sisters-in-law. My friend, Subha, and her mother helped me keep the end goal constantly in sight. I am indebted to them too.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The need for efficient 3D object representations is crucial in image processing, computer graphics, computer animation, computer-aided design and other related areas. Detailed surveys of several representation techniques are given in [1,22]. The representation mode is usually determined by the data acquisition technique or by the type of application. For instance, a surface representation is suitable for graphical display of opaque objects, whereas it is easier to perform operations such as matching and interference analysis with volumetric representations. A common problem with most representation techniques is that requirements for memory and processing time grow as exponential or quadratic functions of the input image size. This calls for a compact data structure that allows images to be compactly represented and facilitates time-efficient implementation of many graphical or image processing operations. The octree structures, a class of hierarchical data structure, is such a candidate.

If a given silhouette view or profile is swept along a line parallel to the viewing direction, it generates a cylinder for orthographic projection (cone for perspective projection). If three silhouette views in three perpendicular planes are given, this sweeping process generates three cylinders. The intersection of three cylinders constrains the object to lie in that volume. This is a good approximation to reconstruct an object from its silhouette views. As the number of silhouette views increases, the fit between the intersected volume and the object volume becomes better.

The following work is discussed in this thesis. Three octrees from the binary image arrays for the different orthographic face views are generated. A parallel algorithm is employed for this purpose. The intersected octree is obtained by merging the three octrees [6]. The complicated process of neighbor finding is the heart of the display process. A modified Cohen-Sutherland algorithm [9] is employed for clipping and hidden line removal. The line drawing of the object is based on the approach of [5]. The code is written in the "C" language. The octree generation is performed on an Intel iPSC/2 hypercube computer and the display is done on a SUN 3/60 workstation.

CHAPTER II

LITERATURE REVIEW

Hierarchical data structures such as the quadtree and octree have their roots in attempts to overcome problems that arise when the scene being modeled is more complex than the display grid (in size, precision, number of elements, etc.). The problems are solved with object-space hierarchies and image-space hierarchies [7].

Object Space Hierarchy

Two kinds of logistical problems present themselves in scene modeling. First, communication between the user software and the graphics package, i.e, the number of procedure calls (or commands transmitted on a graphics channel), can become a bottleneck for the system. The second problem is in determining what subset of the scene is actually visible. For example, in a 512*512*512 scene, only about 512*512 of it is actually visible at any given time. When the scene extends horizontally and vertically past the bounds of the viewing surface, the problem is further aggravated. The first problem has been addressed, in part, by observing that the universe can be

hierarchically organized into objects composed of subobjects, which are in turn composed of other objects, and so forth. This observation has been used as the basis for the organization of the user's interface to the data from the earliest graphics systems to the most recent graphics package designs.

Since the object space hierarchy must be kept to solve the communication problem, it is tempting to use this hierarchy to solve the visible-subset problem. One way to adapt the object hierarchy to the visible-subset problem is through the notion of bounding objects. When determining whether or not an object is visible, it is common to surround the object with a bounding box or even a sphere. If the bounding object is not visible, then clearly the object being bounded is also not visible. This technique produces a major computational savings, since it is usually much easier to test for visibility of the bounding object than the visibility of the bounded object. However, the approach cannot deal with the visible-subset problem when the number of objects is large. Researchers have noted that the objects being bounded need not be limited to the primitive objects of the scene; instead, bounding objects can also be placed around the complex objects formed by the different levels of the object hierarchy.

Image-Space Hierarchy

A natural alternative to processing graphics commands in the object-space hierarchy is to organize the data around an image-space hierarchy. One problem with traditional image-space representations (i.e 2D and 3D arrays) is that they require the user to fix the maximum resolution in advance. However, a hierarchical organization of the image space allows the resolution to vary with the complexity of the objects in various regions. Of course, there are many ways to partition the image space (when it is viewed as a continuous plane/space), but to interface easily with a Cartesian coordinate system and with the typical display device controller, a decomposition of the plane into square regions (and a space into cubical regions) is simplest.

While justifying the use of object-space hierarchies for image-space processing, it is often referred to as the property of area coherence, which means that objects tend to represent compact regions in the image space. Similarly, we might speak of object coherence as being a factor in image-space hierarchies, since regions that are close to each other tend to be parts of the same object. Thus, both types of hierarchies tend to approximate each other.

Quadtree

Quadtrees [7] are hierarchical data structures used for compact representation of 2D images. A quadtree is generated by dividing an image into quadrants and repeatedly subdividing the quadrants into subquadrants until each quadrant has uniform color (e.g "0" or "1" in a binary image). The root of a quadtree corresponds to the image it represents. A node in a quadtree either is a leaf node or has four child nodes. Each child node is associated with a quadrant of the block corresponding to its parent node (Figure 1). The advantages of the quadtree representation for images is that simple and well-developed tree traversal algorithms allow fast execution of certain operations such as superposition of two images, area and perimeter calculation, moment computation, and the generation of the octree representation of 3D objects. In addition, the 2D coordinate of each block is implicitly stored in and can be readily recovered from the quadtree representation [6].

Octrees

Octrees [7] are a 3D analog of quadtrees. While quadtrees encode the information in a 2D picture array of points, octrees encode the information in a 3D array of points. Starting with an upright cubical region of space that contains the object, one recursively decomposes the space into eight smaller cubes called octants which are

labeled 0 through 7 (Figure 2). If an octant is completely inside the object, the corresponding node in the octree is marked black; if completely outside the object, the node is marked white. If the octant is partially contained in the object, the octant is decomposed into eight sub-octants each of which is again tested to determine if it is completely inside or completely outside the object. The decomposition continues until all octants are either inside or outside the object or until a desired level of resolution is reached. Those octants at the finest level of resolution that are only partially contained in the object are approximated as occupied or unoccupied by some criterion such as viewpoint.

The starting cubic region is called "the universe cube". The recursive subdivision of the universe cube in the manner described above allows a tree description of the occupancy of the space (Figure 2). Each octant corresponds to a node in the octree and is assigned the label of the octant. Figure 3b shows the octree for the object in Figure 3a. The child nodes are arranged in increasing order of label values from left to right. A hatched ellipse represents a gray node, a dark circle represents a black node, and an empty circle represents a white node. In practice, of course, the white nodes need not be stored. The geometric information contained in octree data structures is implicit. Roughly, the location

of a subcube is derived by traversing the tree, and the size of the subcube is determined by the level of the tree at which it resides.

There are several advantages to this data structure. First, there is a single primitive shape, the cube. An arbitrary object can be represented to the precision of the smallest cube. Also, only a single set of manipulation and analysis algorithms is required for all objects. Operations such as hidden surface removal and interference detection show only linear growth because all objects are kept spatially pre-sorted at all time. By traversing the tree in the proper sequence, for example, regions of space will be visited in a uniform direction in space. Thus the hidden-surface algorithm requires no searching or sorting. The tree representing the object to be displayed is simply traversed in a specific order, depending on the view direction. However, this efficiency comes at the cost of the representation becoming very sensitive to object location and orientation. For instance, if the object moves, it occupies different cells of the cubic tessellation and as a result its octree may change drastically. Juyang Weng and Narendra Ahuja [15] deal with this problem with the object centered approach, where the placement of primitives is determined by the placement of the objects to be represented.

Display Technique

Doctor and Torborg [4] give a surface display algorithm that makes use of a quadtree to represent the image. Their algorithm includes a feature called "semitransparency", which provides the ability to view internal surfaces. Semitransparency is accomplished by averaging the color values of octree regions that project onto the same area in the image. The color values are multiplied by a weight factor on the basis of the thickness of the octree region, which represents the degree of opaqueness. An alternative method of displaying the object represented by an octree is described by Meagher [2]. His algorithm produces a surface display from an octree after hidden surface removal. However, surface displays depend upon light source positions. In addition, many output devices cannot draw shaded surfaces. Carlbom et al. [14] proposed a polytree structure in which a leaf node can be one of five types: full, empty, vertex, edge or surface. The class of objects represented by polytrees is restricted to polyhedrals. They also developed a scheme to generate the polytree of an object described by a set of polygons.

The thesis is organized as follows. Chapter III discusses the generation of facial octrees using a parallel algorithm. Chapter IV discusses the process of

merging of octrees and performance measures. Display of an octree represented object is discussed in chapter V. Chapter VI completes the body of the thesis by summarizing and concluding the results.

CHAPTER III

FACE OCTREE GENERATION

A "face view" is the view obtained when the line of sight is perpendicular to one of the faces of the universe cube and passes through the center of the cube. Thus a face X view is the orthographic projection of the object onto the YZ plane. A digitized silhouette image would be represented in the computer as a square array of pixels. Pixels having a value of 1 denote the region onto which the object projects. Pixels having a value of 0 represent the projection of free space.

The projection of the cube in Figure 1 along the X direction results in pairs of octants projecting onto the same region in the image. For example, octants 5 and 4 project onto the upper left quadrant, octants 7 and 6 project onto the upper right quadrant, and so on (Figure 4). This simple relationship between octants and their projections allows the construction of the octree directly from the pixels in a digitized silhouette image.

Given a square array of pixels representing a face X silhouette image, its contribution to the octree can be

obtained using the decomposition scheme shown in Figure 4. The quadrants of the silhouette image are processed as if a quadtree were being constructed. A quadrant is recursively decomposed until it is either all ones or all zeroes. But instead of adding to the tree only one node per quadrant during recursive decomposition, as is the case with quadtrees, two nodes are added. Thus, when a quadrant of the silhouette is further decomposed, each sub-quadrant could add up to four nodes to the octree instead of one. A similar procedure is used for the other two face views, the only difference being in the labeling scheme for the image quadrants (Figure 4b, 4c).

Parallel Algorithm

Generating octrees for different views is an inherently parallel process. However, generation of a single octree can also be parallelized in the sense that in Figure 5 the processing of subdomain NE is independent of processing of subdomain SW or any other subdomain. This concept can be applied recursively until the size of the object domain is 2*2. This exploitation of parallelism is achieved by dividing the object domain (i.e. image array) into square sub-blocks and allocating them to (2**n)*(2**n) different processors (For other decomposition techniques see [21]). Now each processor operates on its share of a subblock in parallel (Figure 6). Since the iPSC/2 hypercube computer does not share

memory among its processors, a master processor needs to collect the results from other processors through communication calls. As our primary algorithm describes an octree with a pointer data structure, it is imperative that this dynamic data structure be converted to a static data structure. Therefore, each processor must do this, contributing to overhead compared to a serial algorithm, before the result can be sent to the master processor.

Each processor is assigned a direction based on the sub-block it processes (Figure 6b). In the 2*2 mesh of processors, processor 0 is assigned North-West (NW), processor 1 is assigned North-East (NE), processor 2 is assigned South-West (SW), and processor 3 is assigned South-East (SE) direction. A similar direction assigning scheme can be extended to a larger mesh of processors. Since the communication calls may be initiated at different times for different processors, a check is to be made at the master processor during the collection of other nodes' contributions. For example, in Figure 6c, for face X octree generation, the result from node 3 (direction NE) will be stored in the sixth and seventh child pointers at the master processor, the result from node 7 (direction SE) will be stored in the second and third child pointers at the master processor and so on. Similar mapping is done for the other two face views. Reconversion of the static data structure to a dynamic data structure has to be done at the master processor

before the results can be stored for further processing. This process of sending the result is done recursively until all the processors have sent the result to the master processor.

CHAPTER IV

MERGING

As mentioned earlier, the object is constrained to lie in the intersection of three cylinders in the X, Y and Z directions. Instead of performing the intersection test explicitly, Ahuja and Veenstra [5] infer the octree nodes from silhouette images according to a predetermined table that pairs image region with their corresponding octree nodes. Chien and Aggarwal's [6] approach of carrying out intersection testing is quite intuitive and is followed instead. If at least one of the three octree nodes is a white node, then the corresponding node in the merged octree will also be a white node. If all three corresponding nodes in three octrees are black nodes, then the node in the resultant octree will also be a black node. If at least two are gray nodes, then the octree will also have a gray node in the corresponding location in the tree. This operation is done by procedures merge_3 and merge_2. The pseudocode can be presented as follows.

1. Start from the root node and traverse the three octrees in parallel.

2. If all three are gray nodes, then perform merge_3 on the eight combinations of their child nodes.
3. If two are gray nodes and one is a black node, then perform merge_2 on the combinations of the child nodes of two gray nodes as follows :

   o If both are gray nodes, then perform merge_2 on the eight combinations of their child nodes.

   o If one is a gray node and the other is a black node, then convert the subtree with the gray nodes as the root of an octree.

   o If both are black nodes, then the corresponding node in the octree is a black node.

   o If at least one is a white node, then the corresponding node in the octree is a white node

4. If one is a gray node and the other two are black nodes, then convert the subtree with the gray node as the root to an octree.

5. If all three are black nodes, then the corresponding node in the octree is a black node.

6. If at least one is a white node, then the corresponding node in the octree is a white node.

Converting a subtree with a gray node as the root of an octree is a simple copying process. Figure 7 depicts

the octrees for different face views of the object in Figure 3. At this stage, the information from three facial octrees is no longer needed, and, therefore, can be deleted to reuse the memory.

Performance Measure

Speedup of a mesh of processors can be defined as follows. It is the ratio of the time to execute on 1 processor to the time to execute on p processors. Ideally, it should be equal to p. But, due to the overhead (communication) and load imbalance, it might be less than the ideal value. Table 1 lists the number of octree nodes and speedup for the various test objects. The size of the universe cube except for the first two objects (128*128*128) was 64*64*64. As mentioned earlier, the process of octree generation involves sending each nodes' octree to the master processor. With this being the overhead, speedup as high as 10.6 was reported. The lower speedups can be attributed to the load imbalance problem. Some processors might be overloaded, where as others might be doing less of useful work. Another interesting observation can be drawn, though it is implementation dependent. The number of octree nodes in facial octrees (X, Y, or Z) for 1 processor is more for 16 processors.

This is due to the elimination of white nodes during the conversion of pointer structure to linear structure. This would result in the smaller octrees, and hence, can be merged faster.

CHAPTER V

DISPLAY OF OCTREE REPRESENTED OBJECTS

Once an octree has been constructed, it is natural to want to display it to monitor the correctness and accuracy of the representation. The two display techniques used most commonly are the perspective projection and the parallel projection. The perspective projection is formed with respect to a viewpoint and a viewplane. In this case, all points lying on a given line through the viewpoint project onto the same point on the viewplane. A parallel projection can be defined as a special case of the perspective projection such that the viewpoint is at infinity.

For scenes represented by octrees, the most common display technique is the parallel projection. The parallel projection of a raster octree is at its simplest when the viewplane is parallel to one of the faces of a node in the tree. Implicit in the task of displaying an octree is the solution of the hidden-surface task for the interaction among the objects represented by the octree. Not surprisingly, since the octree imposes a spatial ordering on objects, the hidden-surface task for scenes

represented by octrees can be solved more efficiently than the general hidden-surface task for arbitrary polygons. Note that any opaque object in the front four octants of an octree will occlude any opaque object in the back four octants. This property holds recursively within each of the suboctants.

In this work, I have decided to display the objects using straight lines based on [5]. The advantage of this line drawing process is that there is no restriction on the shape of object which can be drawn. The object is drawn using parallel projection with hidden lines removed. Any viewpoint can be specified and the algorithm will rotate the octree, if necessary, so that the view point is always in the positive octant (octant 7). Since this requires rotation by multiples of 90 degrees, it is performed by simply re-labeling the octants.

The octree representation is a volume description. To extract surface information, all the interfaces between black and white nodes should be labeled. This is accomplished by a multi-level boundary search scheme [6]. It is similar to the boundary size algorithm [18] with the dimensionality equal to 3, and is shown to be O(N), where N is the number of nodes in the octree.

The display algorithm consists of the following steps. The octree is traversed, visiting octants

recursively in increasing distance to the viewer, which is 7, 6, 5, 3, 4, 2, 1, and 0 (Figure 2). For each black node encountered, graphics information (level, length, etc.) is collected and stored in a "box node". When a box node is created it is added to the end of a linked list. Since the tree is traversed so that octants closer to the viewer are visited first, this linked list has the property that elements closer to the beginning of the list represent octants which are closer to the viewer. By traversing the tree in this manner, advantage of the spatial organization of the octree is taken, which simplifies the removal of hidden lines later on. During tree traversal, black nodes are made to point to their neighbors. This is discussed in detail later on. This allows the elimination of cracks, and is also useful in the final stage when the line segments are displayed.

After the linked list of box nodes has been created, each node is projected in perspective onto the image screen and the screen coordinates of the vertices of the projection are stored in the box node. Each box node represents a cube which projects as a hexagon. The numbering schemes for the corners and edges of a projected cube are given in Figure 8. The top corner or edge is numbered 0 and successive integers are assigned clockwise around the projection. Finally, hidden lines are removed by comparing each box node in the linked list against box nodes closer to the beginning of the list. Since box

nodes closer to the beginning of the list are closer to the viewer, any overlap represents part of a box node which should be hidden and is therefore removed. Clipping and hidden line removal are accomplished by a modified Cohen-Sutherland algorithm.

Neighbor Finding and Crack Elimination

A black/white (b/w) interface is called an i-th level interface if it is an interface between two adjacent i-th level cubes. A recursive procedure is used for this purpose. There are four interfaces between a pair of cubes. In Figure 9, the cube on the left is the cube under consideration. Four octants, 7, 6, 3, and 2 (not numbered) of the left cube are covered by the four octants, 5, 4, 1, and 0 (not numbered), of the cube on the right. Therefore, in 3D for each (i-1)th level gray node, there are 12 interfaces (4 in each direction) for 12 different combinations of child-node pairs that are adjacent to each other.

On each pair of child-nodes, the following steps are performed to detect the i-th level boundaries.

1. If both are gray nodes, then repeat the same procedure for 4 pairs of their child-nodes that are adjacent to each other.
2. If at least one is a black node, create a box node and then store this i-th level boundary information in the

black node.

3. If one is a gray node and other is a non-gray node, then traverse the subtree with the gray root until the non-gray nodes that are adjacent to the i-th level non-gray node are reached. Check the colors for each pair of adjacent nodes for the i-th level boundaries and create a box node and store the information in the node with lower level if the two colors are different.

Note that step 3 describes a case where a cube with a larger size is adjacent to a cube with a smaller size. In this case, the b/w interface is part of one face of the cube of larger size. If the boundary information is stored in the object node of the larger size, the location of the b/w interface on the face of the cube also needs to be stored. Hence, it is advantageous to store the boundary information in the cube with the smaller size.

A problem unique to line drawing from an octree representation is the elimination of cracks from the drawing. A crack is a line which should not be drawn because it corresponds to an edge between two adjacent octants whose surfaces are contiguous, and, were it to be drawn, would appear as a crack on an otherwise smooth surface. The pictorial representation of a simple case is given in Figure 10. The edges numbered 7 and 8 of the left cube and edges numbered 5 and 4 of the right cube

need to be deleted to make the surface smooth. Since a large octant may have many small neighbors along an edge, eliminating the cracks may fragment the edge into several pieces. For this reason edges are stored as linked lists of visible segments.

Since the child pointers of all black nodes are nil, they can be utilized to point to their neighbors. In the algorithm, pointer 0 is used for the right, pointer 2 is used for the front, and pointer 4 is used for the top neighbor. The odd numbered pointers are used for other three neighbors. Once all the neighbors have been found, a check is made to see if the neighbors share any common border(s). Based on the length of edges, splitting or deletion may occur. A simple case of splitting is illustrated in Figure 11. Part of an edge of a big cube is hidden by a complete edge of a small cube, and, therefore, should be removed. This, in effect, results in the splitting of the edge of the big cube. Figure 12 depicts all the possible cases where splitting and edge removal can occur.

Only black nodes are of importance for display. If all the six pointers of a black node are used to point to its neighbors, then the black node is hidden, and, therefore, will be skipped during the actual display of segments.

Hidden Line Elimination

After cracks are removed and the perspective projections are calculated, the hidden lines are removed [8,9] using an edge intersection technique. Each edge of a projected octant is tested for intersection with projections of all other octants which are closer to the viewer. Thus, the computational complexity to eliminate the hidden lines is proportional to the square of the number of box nodes.

To carry out intersection tests, a modified Cohen-Sutherland clipping algorithm is used. The equation of a straight line can be given as $Y = mX + b$. The points on the line can be considered to be the solution of the equivalent formula $Y-mX-b = 0$. Then points on one side of the line satisfy $Y-mX-b > 0$, and, the points on the other side of the line satisfy $Y-mX-b < 0$. Thus, the location of a point can be determined by evaluating the formula (or any equivalent) and checking the sign. The Cohen-Sutherland code (Figure 13a) is ORed based on this sign. The Cohen-Sutherland code for the point in Figure 13b is 000011 because the point is outside the edges with the code 000001 and 000010.

For each edge segment of the projected octant under consideration, the bit codes for its end points are calculated based on the projected octant which it is compared against. If the edge is completely outside the

projected octant, and therefore visible, the logical AND of the two bit codes will be non-zero. If the edge is completely inside the projected octant, and therefore hidden, then the bit codes will be zero. Otherwise, the edge partially overlaps the projected octant and therefore, should be split.

Implementation Details

Octrees were generated from a silhouette image of size 64*64 except for the two rectangles (128*128). Following is the data structure used for a typical node in the octree.

```
struct  octree {
        char    color;
        struct  octree  *child[8];
        struct  box     *boxptr;
};
```

The field "color" represents the color of the node. The eight pointers are the pointers to its children. Of course, for a non-gray node they are nil, but the pointers of black nodes are used in the neighbor finding process. Since the image array is recursively scanned until it is 2*2, compaction of a non-gray node may be necessary if all its children are of the same color. "boxptr", a pointer to box type data structure, is used to store the graphics information of the particular node, and has the following

data structure associated with it.

```
struct  box {
        int     origin[3], len, flag[6];
        double  corner[6][2], xleft, xright,
                yhigh, ylow, suth_const[6][2];
        struct  Edge    *edge[9];
        struct  box     *next;

};
```

"origin" is the coordinates of the corner farthest from the viewer; "len" is the length of the side of cube; "corner" is the six projected corners of a cube; "xleft" and "xright" gives the range of the X-coordinates of the extent; "ylow" and "yhigh" gives the range of the Y-coordinates of the extent; "flag" is used to store the condition to be inside the projected cube; array "suth_const" contains the slope and intercept of the six edges; "edge" is an array of pointer to Edge structure representing the nine potentially visible edge of the projected cube; and finally, "next" is a pointer to the next element in the linked list.

Edges of projected cubes are represented by the edge linked list. A typical element of the linked list is as following.

```
struct  Edge {
        int     min, max;
        double  xmin, ymin, xmax, ymax;
        struct  Edge    *next;
};
```

The elements "min" and "max" give the starting and ending position of a segment of the edge, and are found based on the size and origin of the cube pointing to this structure. "xmin", "ymin", "xmax", and "ymax" gives the screen coordinates corresponding to the min and max values.

Table 2 lists the number of box nodes, and edge nodes for the sample objects. As it can be seen, simple objects, such as object of Figure 1, can be graphically described by a small number of box and edge nodes. Figure 14 through Figure 17 are the line drawings of the sample objects. Though three views are sufficient to describe simple objects, more number of silhouette views are needed to get better approximation. The actual display was done by Sun CGI (Computer Graphics Interface) routines. Hardcopies of the drawings were taken on an NEC Ink Jet Printer.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

Conclusion

Since the octree is a very versatile data structure and allows for efficient manipulation of the representation, it can be used very efficiently in object recognition tasks. Any arbitrary shaped object, convex or concave with interior holes, can be represented to the precision of the smallest cell. Geometrical properties such as surface area, volume, center of mass and interference are easily calculated at different levels of precision. Because of the spatial sorting and the uniformity of representation (only three distinct node types are required), operations by octrees are efficient.

An octree saves memory over a space array because not every individual cell of the space array need be represented by the octree. Homogeneous "chunks" of contiguous cells may instead be represented as larger cells of the same density - full or void. The amount of memory needed to encode one size of cell is the same as the amount of memory needed to encode another size of cell, so the replacement of many cells by one cell leads

to memory savings.

Octree representation will not be a good choice for objects whose space array representations do not possess homogeneous regions. Objects that have alternating patches of void spaces and matter-filled spaces at or near the levels of detail that are of interest will not have efficient octree representations. Also, some accuracy is lost by approximating the shape of an object by cubes.

Future Work

Octree generation is a good problem for parallel machines. In pattern recognition and other related fields, more views will be required to describe an object. Instead of processing with three views, thirteen views (three face views, six edge views, and four corner views of the universe cube, since the other thirteen views will just be mirror images) could be used to get a better description of an object. As the number of views increases, processing time also increases. In real time applications, this may not be tolerable, so the obvious choice will be to use parallel algorithms.

In ray tracing, octrees can be used to speed up the determination of the objects that are intersected by rays emanating from the viewpoint. Raytracing is an approximate simulation of how the light that is propagated through a scene lands on the image plane. This simulation

is based on the classical optical notions of reflection (diffuse and specular) and refraction.

An important advantage of quadtrees and octrees is that it is easy to update them to reflect changes in the scene that they are representing. Thus it is natural that they would prove useful in the representation of scenes that change over time due to the motion of objects within the scene. Ahuja and Nash [16] represent motion by updating an octree structure as the object is moved. Alternatively, Samet and Tamminen [17] view a changing 3D scene as a 4D object and use a 4D bintree to represent the space-time object. Besides using octrees to represent motion, they also can be used to plan motion. Kambhamati and Davis [19] have developed a multiresolution path planning heuristic for 2D motion using quadtrees that could easily be extended to 3D motion using octrees. Fujimura and Samet [20] use a similar approach to do path planning in the presence of moving obstacles.

BIBLIOGRAPHY

1. A. A. G. Requicha, Representation for rigid solids: Theory, methods, and systems, Computing Surveys Vol. 12, No. 4, 1980, 437-464.

2. D. Meagher, Geometric Modeling Using Octree Encoding, Computer Graphics and Image Processing, Vol. 19, 1982, 129-147.

3. C. L. Jackins and S. L. Tanimoto, Octrees and their Use in Representing Three-Dimensional Objects, Computer Graphics and Image Processing, Vol. 14, 1980, 249-270.

4. L. J. Doctor and J. G. Torborg, Display Techniques for Octree- Encoded Objects, IEEE Computer Graphics and Applications, Vol. 1, No. 4, July 1981, 29-38.

5. N. Ahuja and J. Veenstra, Octree Generation and Display, Technical Report UILU-ENG-86-2215, Coordinated Science Laboratory, University of Illinois, Urbana, May 1986.

6. C. H. Chien and J. K. Aggarwal, Volume/Surface Octrees for the Representation of Three-Dimensional Objects, Computer Vision, Graphics, and Image Processing, 36, 1986, 100-113.

7. H. Samet and R. Webber, Hierarchical Data Structures and Algorithms for Computer Graphics, IEEE Computer Graphics and Applications, Vol. 8, May 1988, 48-68.

8. J. D. Foley and A. Van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley, Reading, Massachusetts, 1983.

9. Donald Hearn and Pauline Baker, Computer Graphics, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.

10. Ingrid Carlbom, Indranil Chakravarty and David Vanderschel, A Hierarchical Data Structure for Representing the Spatial Decomposition of 3D Objects, IEEE Computer Graphics and Applications, Vol. 5, No. 4, April 1985 24-31.

11. Mann-May Yau and Sargur N. Srihari, A Hierarchical Data Structure for Multidimensional Digital Images, Communications of the ACM, Vol. 26, No. 7, July 1983.

12. B. W. Kernighan and D. M. Ritchie - The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

13. K. Yamaguchi, T.L. Kunii, K. Fujimura, Octree-Related Data Structures and Algorithms, IEEE Computer Graphics and Applications, Vol. 4, No. 1, Jan 1984, 53-59.

14. Irene Gargantini, Linear Octrees for Fast Processing

of Three-Dimensional Objects, Computer Graphics and Image Processing, Vol. 20, 1982, 365-374.

15. Juyang Weng and Narendra Ahuja, Octrees of Objects in Arbitrary Motion: Representation and Efficiency, Computer Vision Graphics and Image Processing, Vol. 26, No. 2, Aug. 1987, 167-185.

16. Narendra Ahuja and C. Nash, Octree Representation of Moving Objects, Computer Vision Graphics and Image Processing, Vol. 26, No. 2, May 1984, 207-216.

17. Hanen Samet and Tamminen, Bintrees, CSG trees, and Time, Computer Graphics, Vol. 19, No. 3, July 1985, 121-130.

18. C. L. Jackins and S. L. Tanimoto, Quad-trees, oct-trees, and K-trees: A generalized approach to recursive decomposition of euclidean space, IEEE Transaction on Pattern Analysis and Machine Intelligence, PAMI, Vol. 7, No. 1, 1985, 94-98.

19. Kambhamati and Davis, Multiresolution path planning for mobile robots, IEEE Journal of Robotics and Automation, Vol. 2, No. 3, Sept. 1986, 135-145.

20. Fujimura and Hanen Samet, Path planning among moving obstacles using spatial indexing, Proceedings of the IEEE International Conference on Robotics and Automation, Philadelphia, April 1988.

21. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors, Volume I, Prentice Hall, Englewood Cliffs, New Jersey.

22. Sargur N. Srihari, Representation of three dimensional digital images, ACM Computing Surveys, Vol. 13, No. 4, Dec. 1981, 399-424.

APPENDIX A

FIGURES AND TABLES

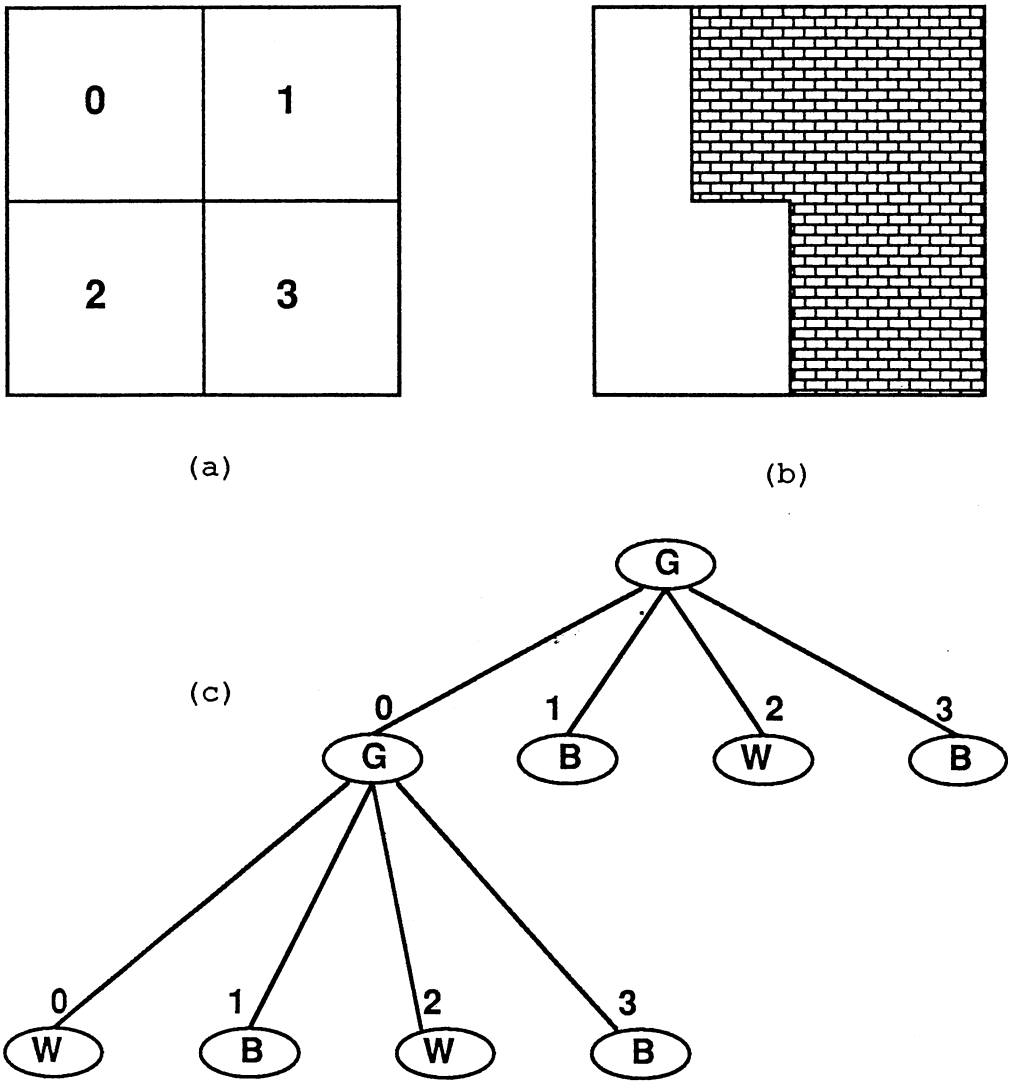
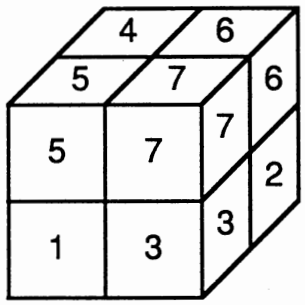
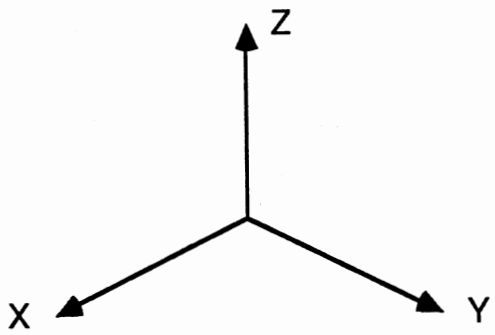


Figure 1. (a) shows the numbering of quadrants. (c) shows the quadtree of the object in (b)

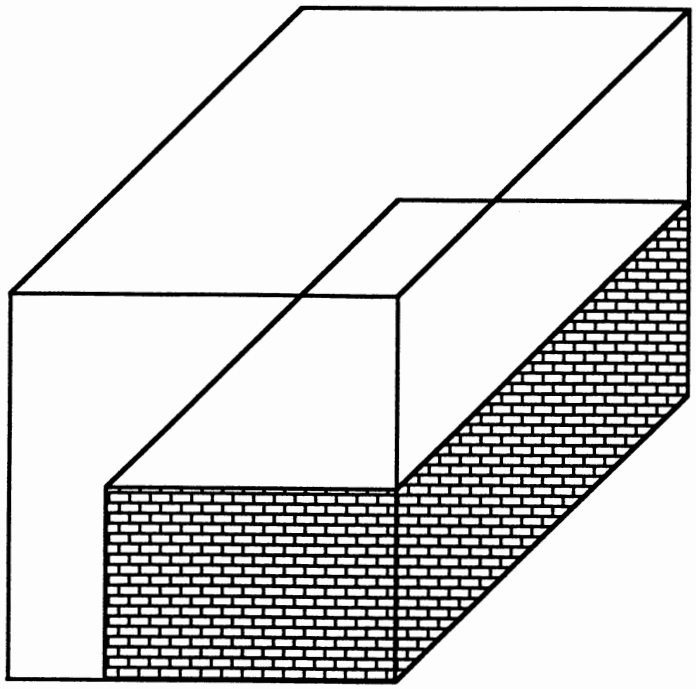


(a)

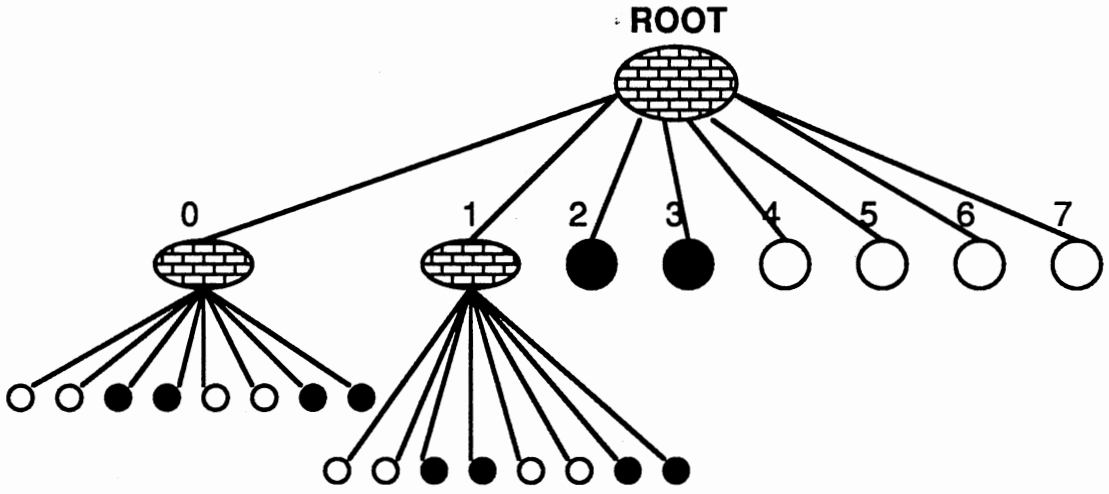


(b)

Figure 2. (a) shows the numbering of octants. The direction assigned in this work is shown in (b)



(a)



(b)

Figure 3. (b) shows the octree of the object in (a)

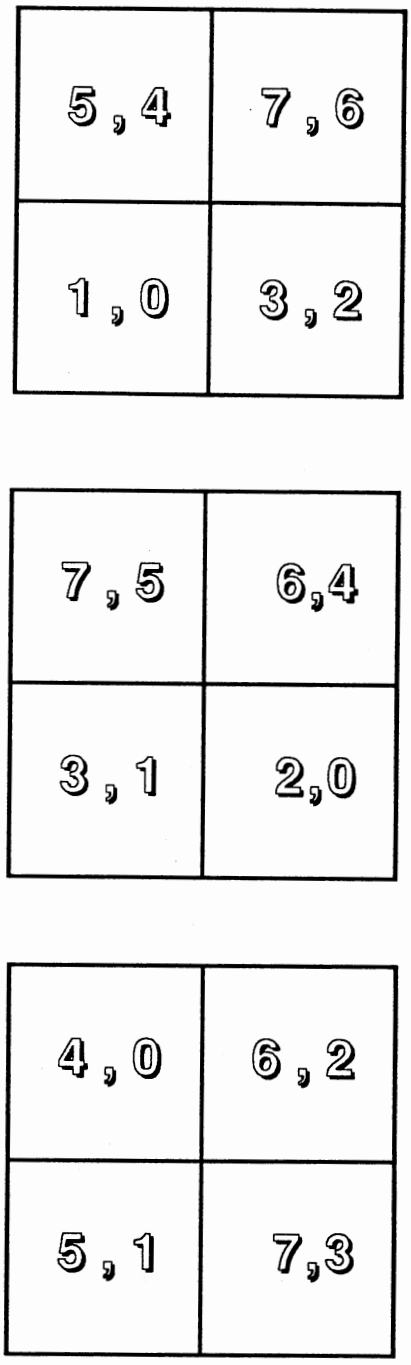


Figure 4. The numbering of quadrants for the (a) face X view, (b) face Y view, (c) face Z view. Each quadrant is assigned two numbers based on the facial view

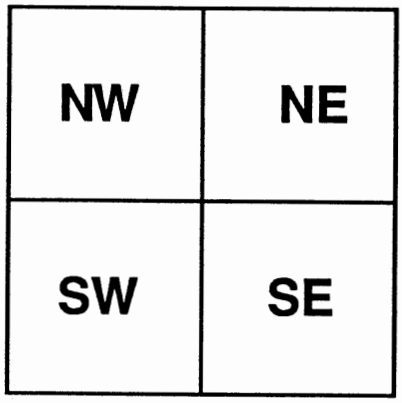


Figure 5. Direction assigning in a 2*2 mesh of processors

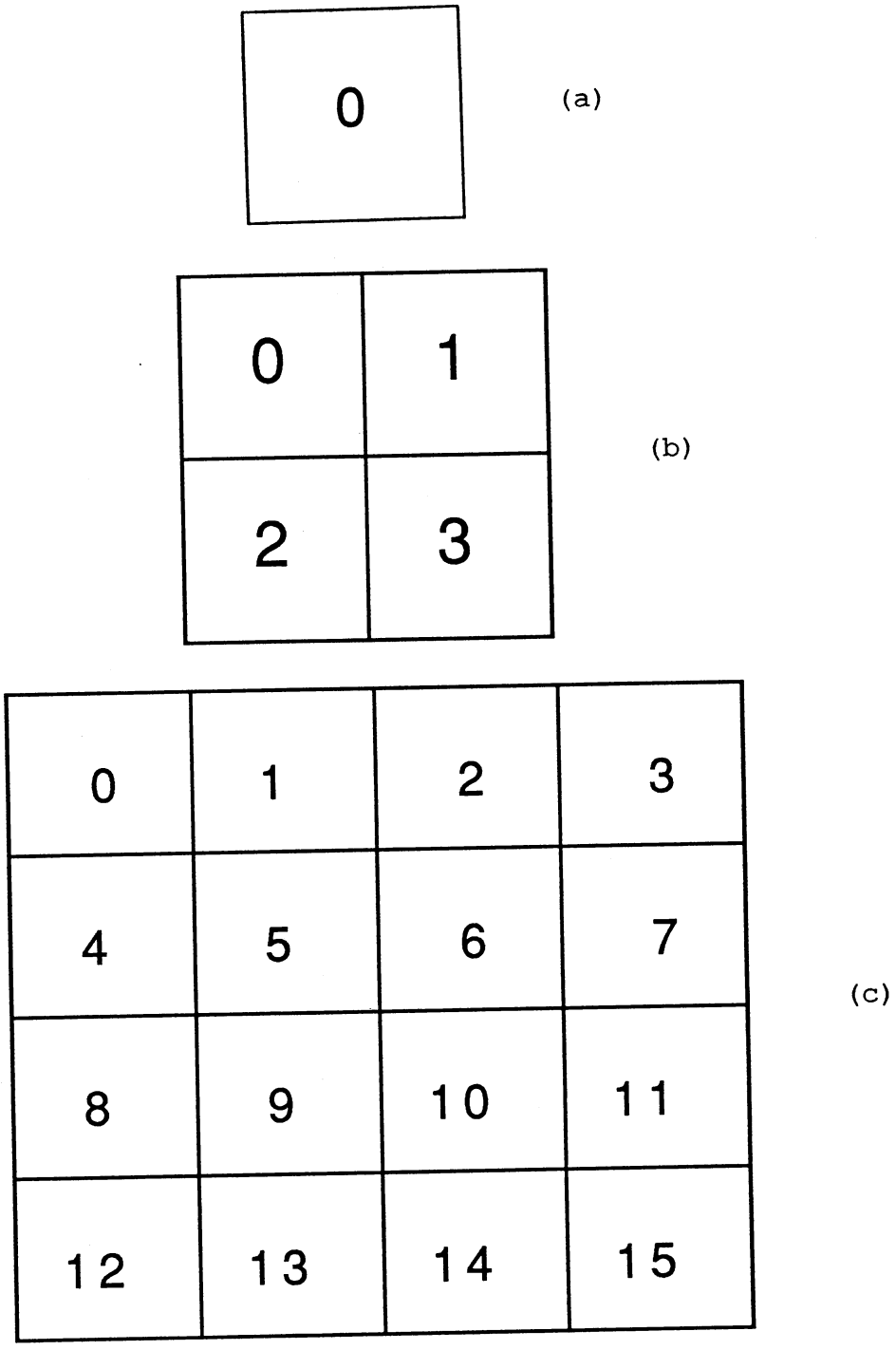


Figure 6. Problem domain decomposition for (a) 1 processor, (b) 4 processors, and, (c) 16 processors. The numbers correspond to processor numbers

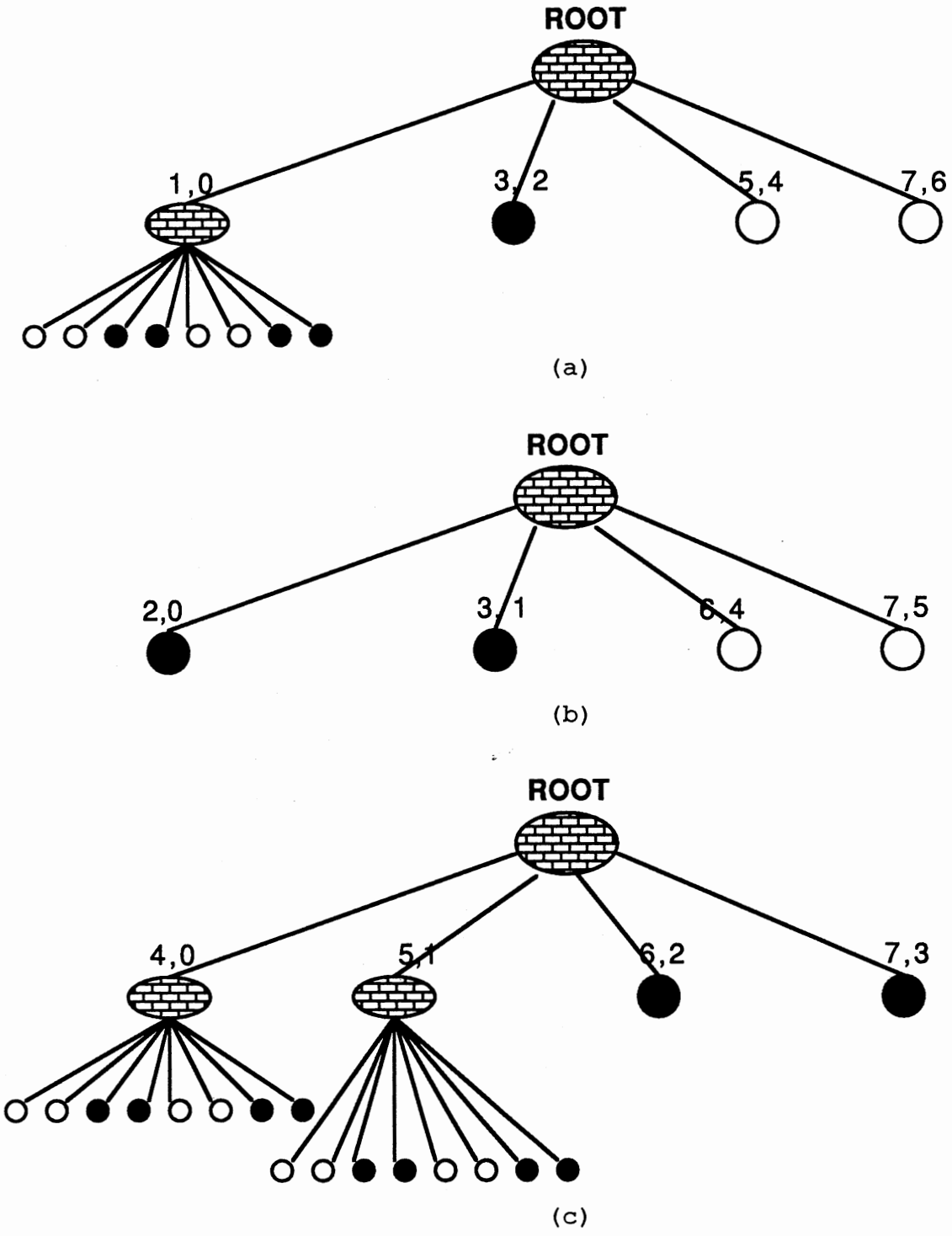


Figure 7. Octree corresponding to (a) face X view, (b) face Y view, and (c) face Z view of the object in Figure 3(b)

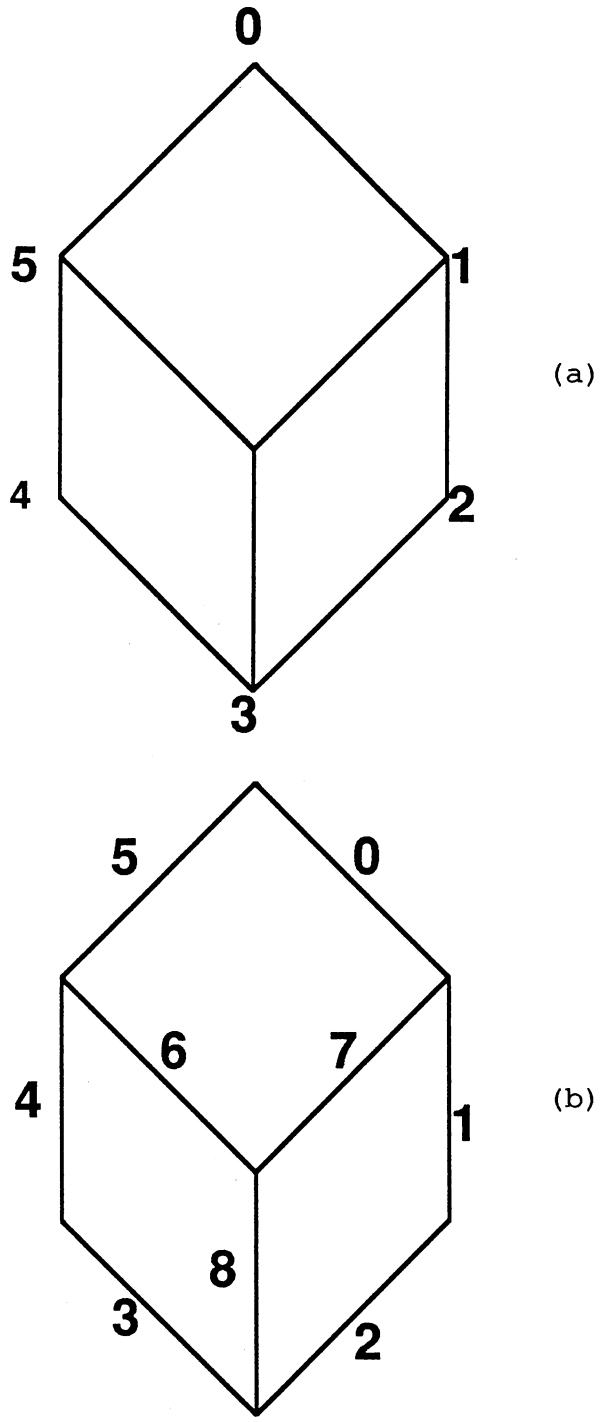


Figure 8. (a) indicates the numbering of corners, and (b) numbering of edges of the projected octant

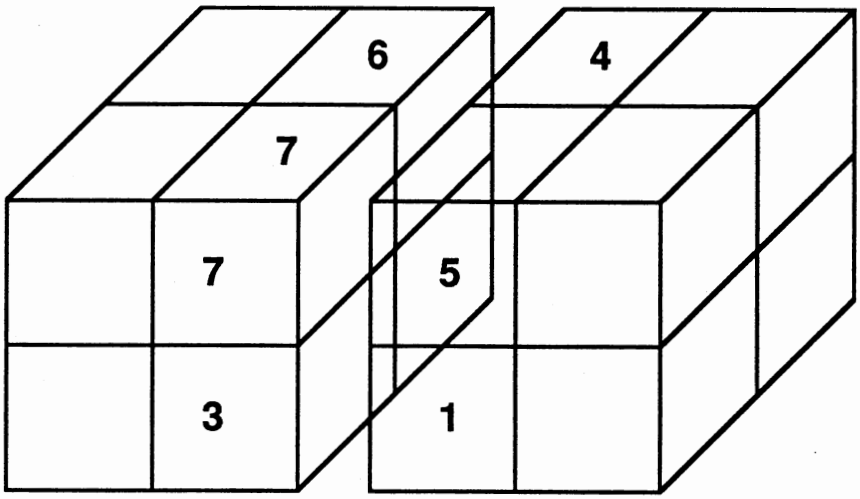


Figure 9. Four interface between a pair of cubes. The four octants 7, 6, 3, and 2 (not numbered) of the left cube form interfaces with four octants 5, 4, 1, and 0 (not numbered) of the right cube

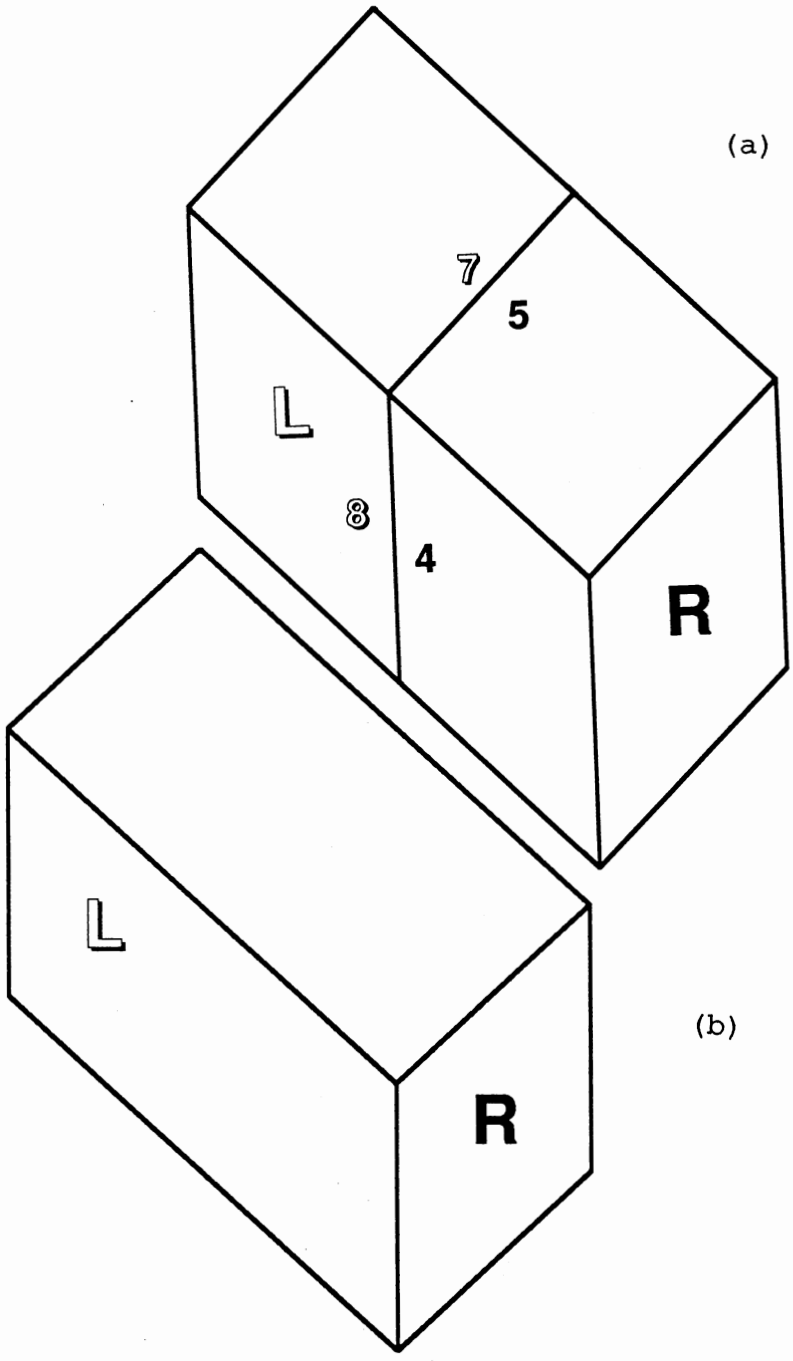


Figure 10. Cubes share a pair of edges in common in (a). These would appear as crack on a smooth surface. (b) shows the removal of common edges yield smooth surface

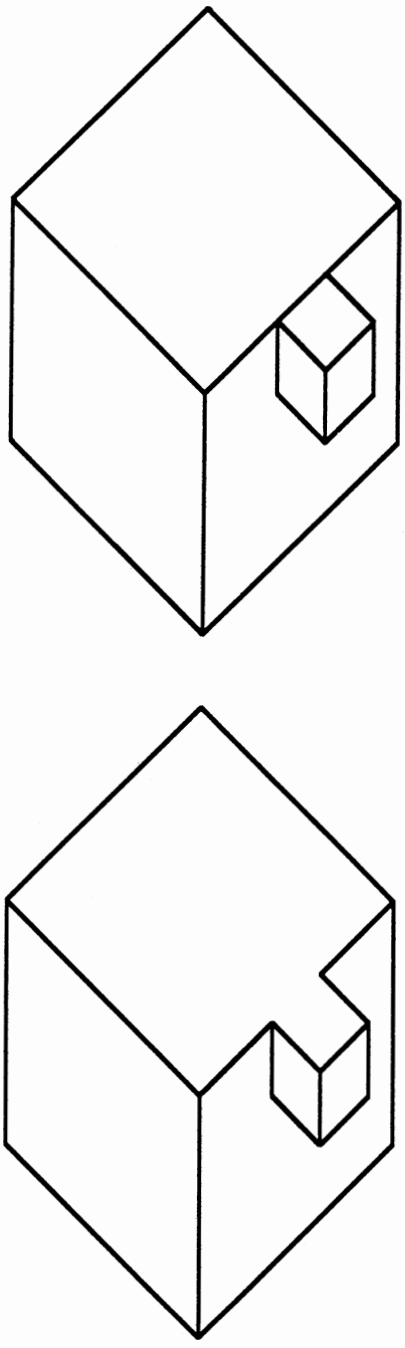


Figure 11. A part of the edge of big cube in common with small cube (a). The edge of big cube splits after removal of common part of the edge (b)

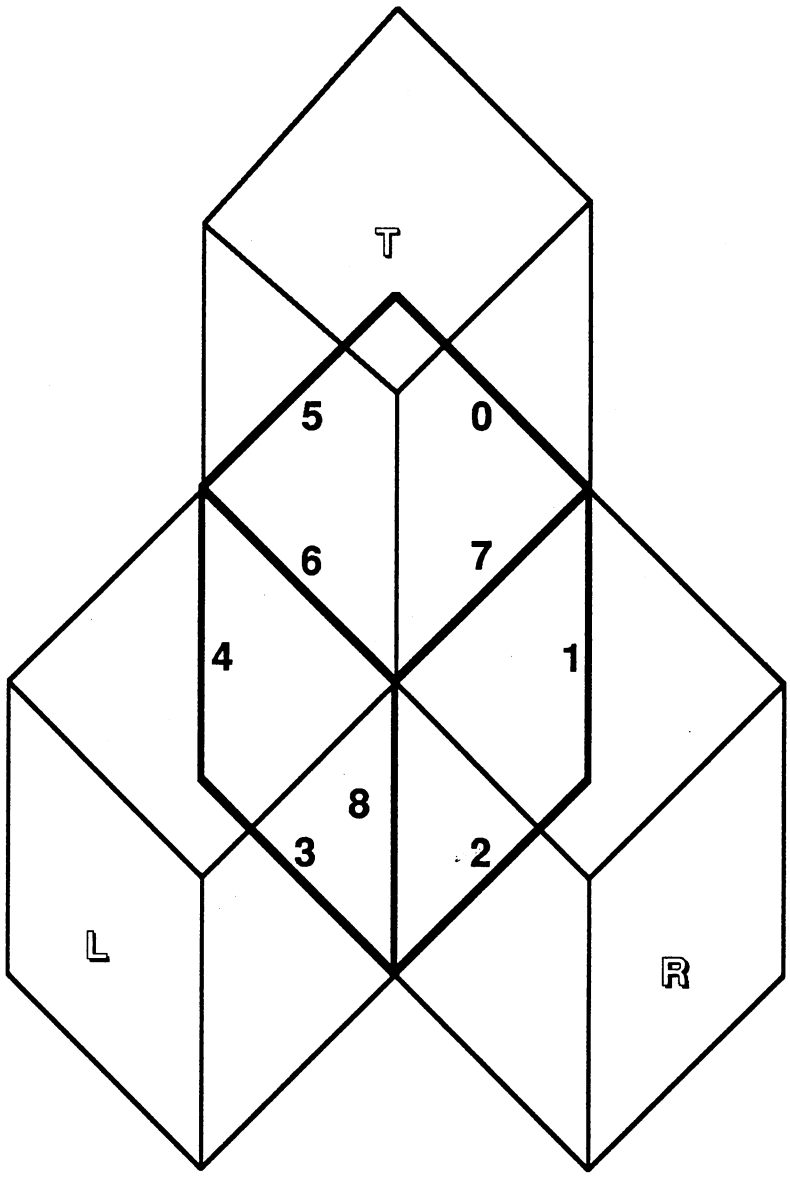


Figure 12. A cube in center (drawn with dark edges) may be sorrounded by at most two adjacent cubes (labelled L, labelled R, and, labelled T) to be partially visible

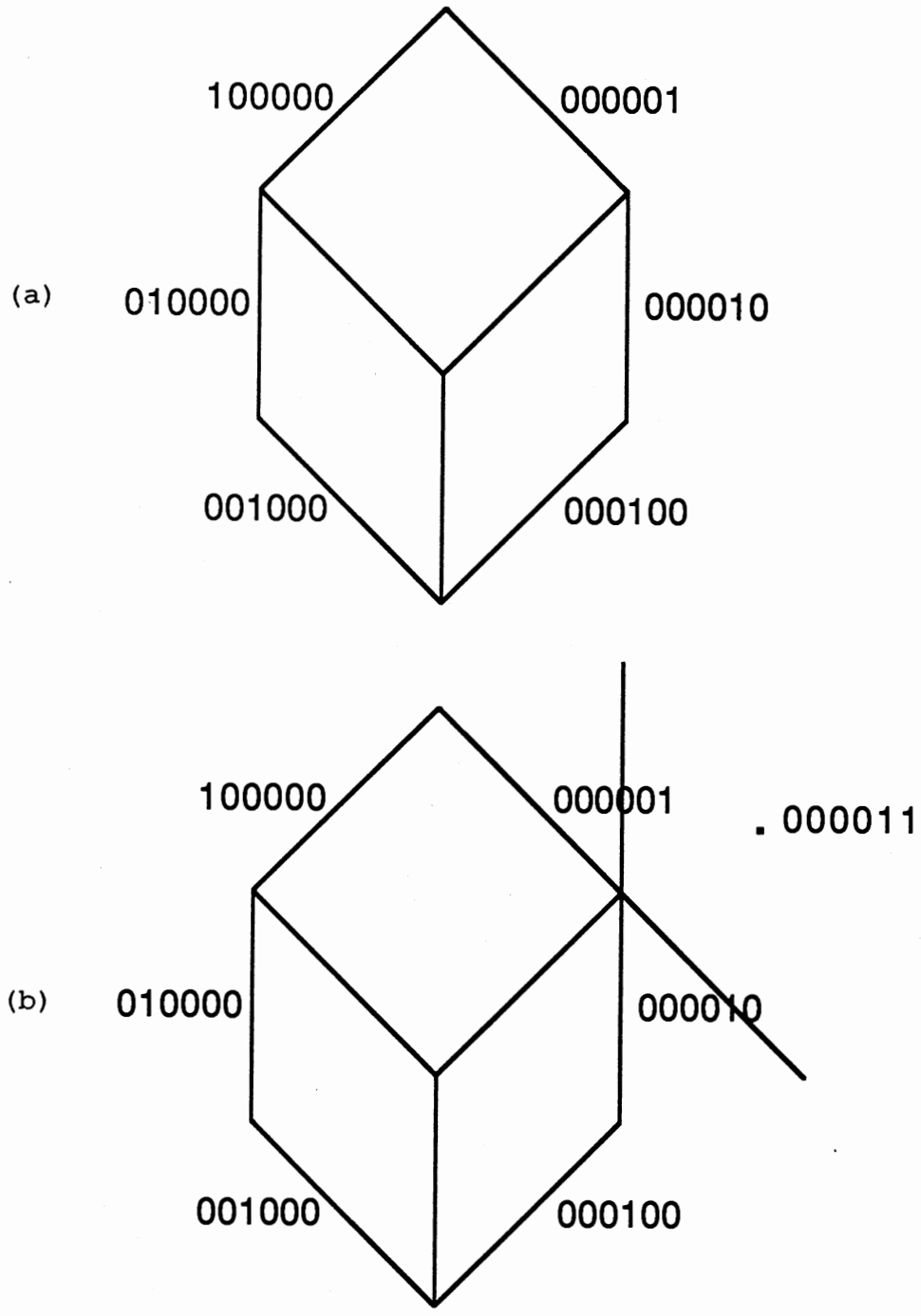


Figure 13. (a) Assignment of Cohen-Sutherland six bit codes to six edges of the projected octant. (b) Calculation of six bit code for the point

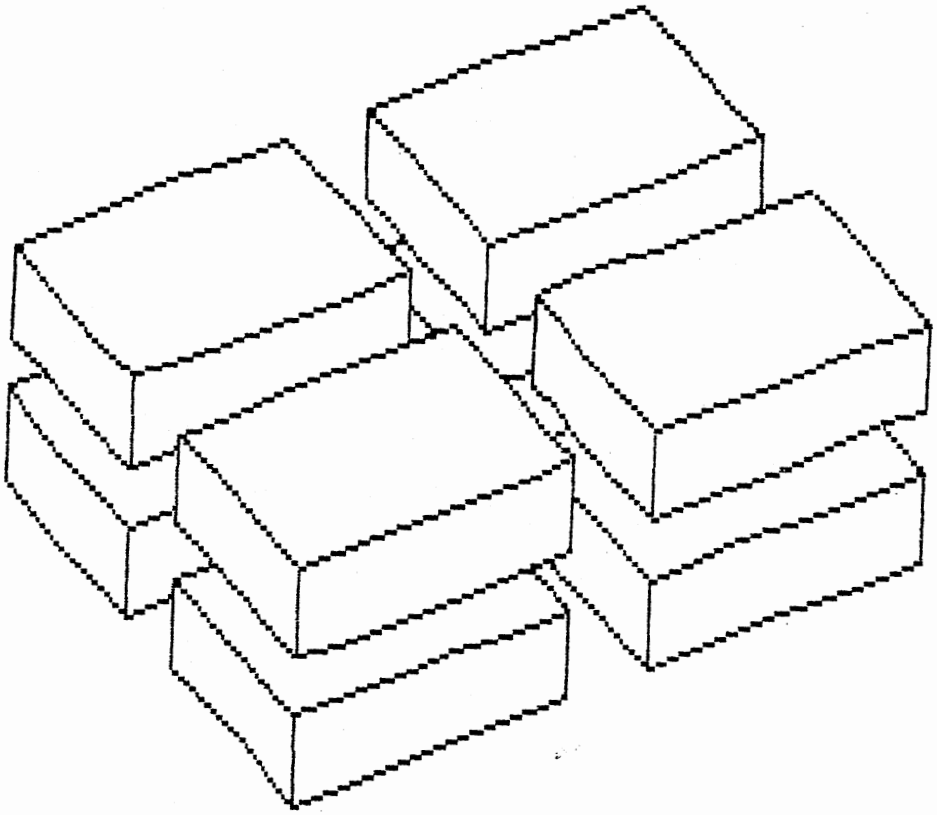


Figure 14. Line drawing of a solid rectangle sliced in three planes

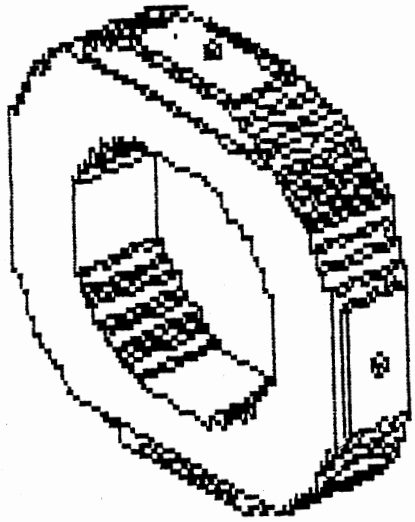


Figure 15. Line drawing of octree represented doughnut

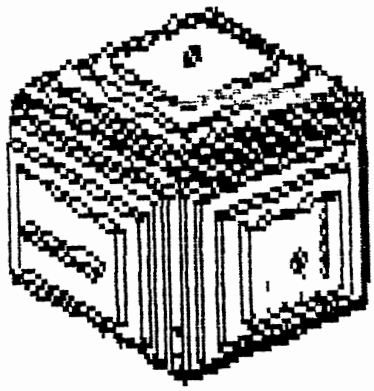


Figure 16. Line drawing of octree represented sphere

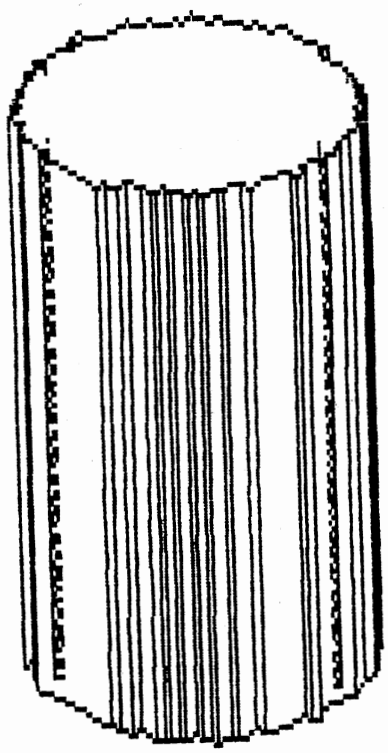


Figure 17. Line drawing of octree represented cylinder

Object	No. of box nodes	No. of edge nodes	Execution time (s)
Rectangle of Figure 3 128*96*64	7	18	15.6
Rectangle sliced in three planes	351	216	24.6
Doughnut min. radius=12 maj. radius=5	614	401	31.2
Sphere radius=4	109	134	13.1
Cylinder radius=10 height=16	558	475	36.1

Table 2. Display Process Statistics on SUN 3/60

Object	1 Processor				16 Processors				No. of Merged Octree Nodes
	No. of Octree Nodes			Speedup	No. of Octree Nodes			Speedup	
	Face X	Face Y	Face Z		Face X	Face Y	Face Z		
Rectangle of Figure 3 128*96*64	17	7	29	1	13	5	25	6.0	13
Rectangle sliced in three planes	733	925	1229	1	681	873	1161	10.6	689
Doughnut min. radius=12 maj. radius=5	7068	3046	3046	1	7009	3001	3001	9.7	1307
Sphere radius=4	1826	1826	1826	1	1785	1785	1785	3.6	227
Cylinder radius=10 height=16	3888	3888	4441	1	3833	3833	4377	8.8	1528

Table 1. Octree Generation Statistics on Hypercube

APPENDIX B

OVERVIEW OF MACHINES USED

INTEL iPSC/2 HYPERCUBE

Intel Personal Super Computer, iPSC/2, is a parallel computer system that is scalable in size. Each node (node and processor are used interchangeably) in the system is a self-contained computer with substantial processing and memory capabilities. The iPSC/2 can be viewed as an ensemble of processing nodes and each is connected to its neighbours via high speed communication channels. Each node has D neighbours, where D is the dimension of the system, yielding the hypercube architecture. For example, this school's iPSC/2 has 2**5 nodes. Therefore, each node has five neighbours. Since each node is independent, a high speed network is used for optimized message passing between nodes.

Each node in the iPSC/2 consists of the Intel 80386 processor, a Weitek 1167 numeric coprocessor, local memory ranging from 1 to 4 megabytes, power of 4 MIPS, a Direct-Connect Module for high speed message passing, 2.8 MB/sec, and carries its own multi-tasking operating system, called NX/2 (Node eXecutive). System Resource Manager, an independent computer by itself, serves as the iPSC/2 connection to the outside world. It consists of an Intel 80386 processor, an 80387 numeric coprocessor, and memory of

8 megabytes. Instead of NX/2, it carries Unix System V.3.2 operating system. Other than serving as gateway to other computers and workstations, the SRM acts as the administrative console for the system and host for various development tools. Only one user at a time can use any subset of nodes, but several users can use different subsets of nodes. Each node in the system is assigned a logical number ranging between 0 and (p-1) where p is the number of nodes in the subset. Node 0 handles all the communication between the system and the SRM. Messages larger than 256 KB cannot be sent to the host where as the message length limit between nodes is the available physical memory.

SUN-3/60 WORKSTATION

Sun-3/60 workstation has an MC68020 processor and an MC68881 floating-point coprocessor, both running at 20 MHz and power of 3 MIPS. It supports Unix 4.3 BSD (Berkeley Software Division) operating system.

APPENDIX C

SOURCE CODE

/* The program host.c acts as controller for the whole process. It sends input, by csend call, to the Hypercube nodes. It receives the message from the node 0 when the whole process gets over.

The node.c program on the other hand, does the actual work. It receives the message from the host.c program to start execution. node.c does the following operations: based on the number of processors, it divides the image array, and generate octrees. The pointer octree is converted by conv_array() routine into linear octree. collect_x(), collect_y(), and collect_z() routines are used to send the linear octree to node 0 in specific order. At node 0, the linear octrees are reconverted to pointer octree by build_tree(). After this phase, merging of three octrees is performed. The logic of the program allows reduction in the number of white nodes.

Based on the view point, view_rotate() rotates the octree such that the view point is in the octant 7. This is done by just relabelling the nodes. The find_neighbor() routine calls itself recursively until all the nodes have been visited and their neighbours have been established. Once it is known that two suboctants are neighbours, two_comp() routine is called to do graphics processing. It eliminates hidden surfaces, and cracks. The project() routine calculates the screen coordinates of the box nodes, and associated edge nodes, for parallel projection. The hidden_clip() routine examines each edge node and decides

whether it needs to be displayed or not. The end points are set to equal if the edge is not to be displayed. boxgen() routine calculates the slope and intercept of six edges of projected octant. Co_suth() calculates the Cohen_Sutherland code for any point. shorten() routine resets the coordinates of an edge. split() on the otherhand, splits and edge segment into two, and calls shorten() to shorten the parts. The display() routine just traverses each box node and collects the coordinates of the edge segment, which in turn are displayed by CGI routine, polyline().

```
::::::::::::::
defs.h
::::::::::::::
#include   "stdio.h"
#define  FALSE   0
#define  TRUE    1
#define  NIL     0
#define  DIM   128

typedef struct box {
   int      origin[3], len, flag[6];
   double   corner[6][2], xhigh, yhigh, xlow, ylow,
            xleft,yleft,xright,yright,suth_const[6][2];
   struct   Edge *edge[9];
   struct   box  *next;
} box_node;

typedef box_node *BOX;

typedef struct octree {
   char           color;
   struct octree  *child[8];
   struct box     *boxptr;
} node;

typedef node *OCTREE;

typedef struct Edge {
   int     min, max;
   double  xmin, ymin, xmax, ymax;
   struct  Edge *next;
} edge_node;

typedef edge_node *EDGE;

struct    vp {
     unsigned  z : 1;
     unsigned  y : 1;
     unsigned  x : 1;
};

union v_point {
     struct vp      coord;
     unsigned  short       reg;
};

::::::::::::::
host.c
::::::::::::::
#include "defs.h"
#include "ctype.h"
#include "stdio.h"
#define HOST_PID 100 /* process id of the host process */
#define APPL_PID   0 /* process id */
```

```
#define INIT_TYPE  0 /*type of INITial mesg. into cube*/
#define END_TYPE   1
#define ALL_NODES -1 /* symbol for all nodes in cube */
#define ALL_PIDS  -1 /* symbol for all processes */
#define DEB        1 /* debug switch */

struct array {
    int size, x[DIM][DIM], y[DIM][DIM], z[DIM][DIM];
};

struct   array image;
int      n_oct=0, n_edge=0;
extern BOX   head_graph;
extern n_box;

union v_point   view_pt;

main(argc,argv)
int  argc;
char *argv[];
{

   static   int   org[]={0, 0, 0};
   int      dim = DIM, row = 0, col = 0, i, j;
   int      start_row, end_row, start_col, end_col;
   double   view[3];
   FILE     *in, *fopen();
   char     cubes[15], buffer[20];

   if(argc < 5) {
     printf("Usage: host n[nodes]m4 x_pt y_pt z_pt\n");
     exit(1);
   }

   sscanf(argv[2],"%lf",&view[0]);
   sscanf(argv[3],"%lf",&view[1]);
   sscanf(argv[4],"%lf",&view[2]);

   view_pt.coord.x = (view[0] >= 0.0) ? 1 : 0;
   view_pt.coord.y = (view[1] >= 0.0) ? 1 : 0;
   view_pt.coord.z = (view[2] >= 0.0) ? 1 : 0;

   initialize(image_x);
   initialize(image_y);
   initialize(image_z);
   sphere();

   getcube("octree",argv[1],0);
   setpid(HOST_PID);

   /* load the node program */
   load ("node", ALL_NODES, APPL_PID);
```

```
   for(i=0; argv[1][i] != 'm'; i++)
      cubes[i] = argv[1][i];
   cubes[i] = '\0';

   image.size = atoi(cubes);

   /* send the size and number of nodes specification
   to all nodes */
   csend(INIT_TYPE, &image, sizeof(struct array),
         ALL_NODES, APPL_PID);

   /* receive a message indicating the process of
   octree generation is over */

   crecv(END_TYPE,buffer,20);
   printf("buffer=%s\n",buffer);

   killcube(ALL_NODES, ALL_PIDS);
   relcube("octree");
   count_nodes(root);
   find_neighbor(root,0,org,DIM);
   project(head_graph,view);
   hidden_clip(root);
   display(head_graph);

}

init(X)
int  X[DIM][DIM];
{

   int   l, m;

   for(l=0; l < DIM; l++)
      for(m=0; m < DIM; m++)
         X[l][m]=0;
}

/* This routine generates binary image arrays for doughnut */
donut()
{

   int    rad1=25, rad2=15, i, j, ht=DIM/2,h=DIM/2, k=DIM/2;
   int   rad1sq, rad2sq, idiff, jdiff, loc,
         locleft, locright, minrad, majrad,
         minrsq, toprow, botrow, leftcol, jdleft,
         rightcol, idleft, idright, jdright;

   rad1sq=rad1*rad1;
   rad2sq=rad2*rad2;

   for(i=0; i < DIM; i++)
      for(j=0; j < DIM; j++) {
         idiff=i-h;jdiff=j-k;
```

```
            loc=idiff*idiff+jdiff*jdiff;
            if((loc >= rad2sq) && (loc <= rad1sq))
                image.x[i][j]=1;

        }

    minrad=(rad1-rad2)/2;    minrsq=minrad*minrad;
    toprow=ht+minrad;        botrow=ht-minrad;
    majrad=(rad1+rad2)/2;
    leftcol=h-majrad;        rightcol=h+majrad;

    for(i=0; i < DIM; i++)
       for(j=0; j < DIM; j++){
         idleft=i-leftcol;     jdright=jdleft=j-ht;
         locleft=idleft*idleft+jdleft*jdleft;
         idright=i-rightcol;
         locright=idright*idright+jdright*jdright;

         if(((i>=leftcol) && (i<=rightcol) && (j>=botrow)
              && (j<=toprow)) || (locleft<=minrsq) ||
                (locright<=minrsq)){
              image.y[i][j]=image.z[j][i]=1;
         }
       }

}

/*This routine generates binary image array for sphere */
sphere()
{

    int   rad=32, i, j, h=DIM/2, k=DIM/2, idiff, jdiff, radsq;

    for(radsq=rad*rad,i=0; i < DIM; i++)
      for(j=0; j < DIM; j++){
       idiff=i-h;    jdiff=j-k;
       if((idiff*idiff+jdiff*jdiff) <= radsq) {
          image.x[i][j]=image.y[i][j]=image.z[i][j]=1;
       }
      }


}

/* This routine generates binary image arrays for cylinder*/
cylinder()
{

    int   rad=4, i, j, ht=12, h=DIM/2, k=DIM/2, idiff, jdiff;
    int   toprow, botrow, leftcol, rightcol, radsq;

    toprow=DIM-h-ht/2; botrow=DIM-h+ht/2;
    leftcol=k-rad; rightcol=k+rad;
```

```
    for(radsq=rad*rad,i=0; i < DIM; i++)
      for(j=0; j < DIM; j++){
       idiff=i-h;    jdiff=j-k;
       if((idiff*idiff+jdiff*jdiff) <= radsq)
          image.z[i][j]=1;
      }

    for(i=0; i < DIM; i++)
       for(j=0; j < DIM; j++)
         if((j>=leftcol) && (j<=rightcol) &&
           (i>=toprow) && (i<=botrow)){
           image.x[i][j]=image.y[i][j]=1;
       }


}

::::::::::::::
node.c
::::::::::::::
#include "defs.h"
#include "stdio.h"
#include "math.h"
#include "ctype.h"

#define HOST_PID 100
#define APPL_PID   0
#define INIT_TYPE  0
#define END_TYPE   1
#define ALL_NODES -1      /* symbol for all nodes in cube */
#define ALL_PIDS  -1      /* symbol for all processes */
#define MAX     2800
#define PROC      16      /* number of processors */

struct array {
    int nodes, x[DIM][DIM], y[DIM][DIM], z[DIM][DIM];
};

struct dimension {
    int row, col;
};

struct X {
    short  child[8];
    char color;
};

struct   X  array_rep[MAX], array0[MAX];
struct   array image;
struct   dimension matrix[PROC];

int     nodeNW,nodeNE,nodeSW,nodeSE,array_rep[MAX];
int     my_pid,my_node,NODE,dim,range,index,elements;
```

```
main() {

     OCTREE    face_x(),face_y(),face_z(),merge_3();
               collect_x(),collect_y(),collect_z();
               root, root_x, root_y, root_z;
     long      i, j, end, start, work[16], clock[16];
     char      buffer[20];

     my_pid = mypid();   /* get process id */
     my_node = mynode(); /* get node number */

     for(  ;  ;  ) {
       crecv(INIT_TYPE, &image, sizeof(image));
       if(my_node < image.nodes) {

      /* size of matrix for 1 node */
         dim = DIM/sqrt((double)image.nodes);
         range = sqrt((double)image.nodes);

      /* assign row and column number */
         for(NODE=0,i=0; i < range; i++)
            for(j=0; j < range; j++, NODE++){
               matrix[NODE].row = DIM/range * i;
               matrix[NODE].col = DIM/range * j;
            }
         start = mclock();
         root_x = face_x(dim,matrix[my_node].row,
                         matrix[my_node].col);
         root_y = face_y(dim,matrix[my_node].row,
                         matrix[my_node].col);
         root_z = face_z(dim,matrix[my_node].row,
                         matrix[my_node].col);

         if(image.nodes > 1){
           for(elements=index=i=0; i < MAX;
                  array_rep[i].color='\0',i++)
              for(j=0; j < 8; j++) /* initialize */
               array_rep[i].child[j] = -1;

            if(root_x-> color != 'G') {
               array_rep[0].color= root_x->color;
               elements = 1;
            }
            else {
               elements= -1;
               conv_array(root_x);
            }

           root_x = collect_x(dim=DIM, 0, 0);
           for(elements=index=i=0; i < MAX;
                  array_rep[i].color='\0',i++)
              for(j=0; j < 8; j++)
               array_rep[i].child[j] = -1;
```

```
            if(root_y-> color != 'G') {
               array_rep[0].color= root_y->color;
               elements = 1;
            }
            else {
               elements= -1;
               conv_array(root_y);
            }

          root_y = collect_y(dim=DIM, 0, 0);
          for(elements=index=i=0; i < MAX;
                 array_rep[i].color='\0',i++)
             for(j=0; j < 8; j++)
              array_rep[i].child[j] = -1;

            if(root_z-> color != 'G') {
               array_rep[0].color= root_z->color;
               elements = 1;
            }
            else {
               elements= -1;
               conv_array(root_z);
            }

          root_z = collect_z(dim=DIM, 0, 0);
        }

        for(i=0; i < PROC; i++)
           clock[i]=0;
        clock[mynode()] = mclock()-start;
        gisum(clock,PROC,work);
        strcpy(buffer,"PROCESS OVER\n");
        if(mynode() == 0){
          for(i=end=0; i < PROC; i++)
             end += clock[i];

          root = merge_3(root_x, root_y, root_z);
          csend(END_TYPE,buffer,20,myhost(),HOST_PID);
        }
        rel_mem(root_x);
        rel_mem(root_y);
        rel_mem(root_z);
      } /* if */
    }  /*for( ; ; ) */
}      /* end main node prog */

/* This procedure makes all the child pointers nil if all
the children are of the same color except gray. It colors
root with the color of its children */

OCTREE    compact(root)
OCTREE     root;
{
```

```
    int     i, flag;
    char    colour;

    /* flag=TRUE means the children are of SAME color */

    for(i=0; i < 8; i++)
       if((root->child[i]) && (root->child[i]->color != 'W'))
         break;

    if(i == 8) {
       root -> color = 'W';
       for(i=0; i < 8; i++)
         if(root->child[i]) {
            free(root->child[i]);
            root->child[i]=NIL;
         }
       return(root);
    }

    colour = root -> child[i] -> color;

    for(flag=TRUE, i=0; i < 8; i++)
       if((root->child[i] == 0) ||
          (root->child[i]->color == 'G') ||
          (root->child[i]->color != colour)) {
         flag = FALSE;
         break;
       }

    if(flag == TRUE) {
      for(i=0; i < 8; i++)
         if(root->child[i]) {
            free(root->child[i]);
            root->child[i]=NIL;
         }

      root -> color = colour;
    }
    return(root);
}

/* This routine is called recursively until the size of
the matrix is 2*2. Since in the face_x view the octants
5 and 4; 7 and 6; 3 and 2; 1 and 0 project to the same
area, recursive calls are made from only four locations.
Other nodes are just copied. */

OCTREE face_x(dim, row, col)
int         dim, row, col;
{

    OCTREE   root, copy(), getnode(), compact();
    char     type;
```

```
   root = getnode(type='G');
   dim = dim/2;

   if(dim > 1) {
      root -> child[5] = face_x(dim, row, col);
      root -> child[4] = copy(root->child[5]);
      root -> child[7] = face_x(dim, row, col+dim);
      root -> child[6] = copy(root->child[7]);
      root -> child[1] = face_x(dim, row+dim, col);
      root -> child[0] = copy(root->child[1]);
      root -> child[3] = face_x(dim, row+dim, col+dim);
      root -> child[2] = copy(root->child[3]);
   }
   else {
      if(image.x[row][col]) {
        root -> child[5] = getnode(type='B');
        root -> child[4] = getnode(type='B');
      }

      if(image.x[row][col+1]) {
        root -> child[7] = getnode(type='B');
        root -> child[6] = getnode(type='B');
      }

      if(image.x[row+1][col]) {
        root -> child[1] = getnode(type='B');
        root -> child[0] = getnode(type='B');
      }

      if(image.x[row+1][col+1]) {
        root -> child[3] = getnode(type='B');
        root -> child[2] = getnode(type='B');
      }
   }   /* ELSE */

   return(compact(root));

}

/* This procedure generates the octree corresponding to
face Y and returns the pointer to the root of the tree */

OCTREE face_y(dim, row, col)
int         dim, row, col;
{

   OCTREE   root, copy(), getnode(), compact();
   char     type;

   root = getnode(type='G');
   dim = dim/2;

   if(dim > 1) {
      root -> child[7] = face_y(dim, row, col);
```

```
      root -> child[5] = copy(root->child[7]);
      root -> child[6] = face_y(dim, row, col+dim);
      root -> child[4] = copy(root->child[6]);
      root -> child[3] = face_y(dim, row+dim, col);
      root -> child[1] = copy(root->child[3]);
      root -> child[2] = face_y(dim, row+dim, col+dim);
      root -> child[0] = copy(root->child[2]);
   }
   else {
      if(image.y[row][col]) {
        root -> child[7] = getnode(type='B');
        root -> child[5] = getnode(type='B');
      }

      if(image.y[row][col+1]) {
        root -> child[6] = getnode(type='B');
        root -> child[4] = getnode(type='B');
      }

      if(image.y[row+1][col]) {
        root -> child[3] = getnode(type='B');
        root -> child[1] = getnode(type='B');
      }

      if(image.y[row+1][col+1]) {
        root -> child[2] = getnode(type='B');
        root -> child[0] = getnode(type='B');
      }
   }   /* ELSE */
   return(compact(root));

}

/* This routine generates the octree for the face Z view
and returns a pointer to the root of the tree */

OCTREE face_z(dim, row, col)
int           dim, row, col;
{
   OCTREE   root, copy(), getnode(), compact();
   char     type;
   root = getnode(type='G');
   dim = dim/2;

   if(dim > 1) {
      root -> child[4] = face_z(dim, row, col);
      root -> child[0] = copy(root->child[4]);
      root -> child[6] = face_z(dim, row, col+dim);
      root -> child[2] = copy(root->child[6]);
      root -> child[5] = face_z(dim, row+dim, col);
      root -> child[1] = copy(root->child[5]);
      root -> child[7] = face_z(dim, row+dim, col+dim);
      root -> child[3] = copy(root->child[7]);
   }
```

```
    else {
       if(image.z[row][col]) {
         root -> child[4] = getnode(type='B');
         root -> child[0] = getnode(type='B');
       }

       if(image.z[row][col+1]) {
         root -> child[6] = getnode(type='B');
         root -> child[2] = getnode(type='B');
       }

       if(image.z[row+1][col]) {
         root -> child[5] = getnode(type='B');
         root -> child[1] = getnode(type='B');
       }

       if(image.z[row+1][col+1]) {
         root -> child[7] = getnode(type='B');
         root -> child[3] = getnode(type='B');
       }
    }   /* ELSE */

    return(compact(root));

}

/* This routine collects the contributions of octrees
from other nodes and the result is stored in NODE 0. It
is again a recursive routine and called until DIM is
not greater than the dimension of submatrix
(the size of matrix of a node) */

OCTREE collect_x(dim, row, col)
int   dim, row, col;
{

    int    i;
    OCTREE root,copy(),getnode(),build_tree(),compact();

    root = getnode('G');
    dim = dim/2;

    /* if dim is > the size of matrix of a node */
    if(dim > DIM/range) {
       root -> child[5] = collect_x(dim, row, col);
       root -> child[4] = copy(root->child[5]);
       root -> child[7] = collect_x(dim, row, col+dim);
       root -> child[6] = copy(root->child[7]);
       root -> child[1] = collect_x(dim, row+dim, col);
       root -> child[0] = copy(root->child[1]);
       root -> child[3] = collect_x(dim, row+dim, col+dim);
       root -> child[2] = copy(root->child[3]);
    }
    else {
```

```
    /* see if the node is in the North West direction of
    the 2 * 2 mesh of processors */

      nodeNW = find_node(row, col);
      if(mynode() == nodeNW)
        csend(nodeNW,array_rep,sizeof(array_rep),0,my_pid);
      if(mynode() == 0){
        crecv(nodeNW, array0, sizeof(array0));
        root->child[5] = build_tree(array0,index=0);
        root->child[4] = copy(root->child[5]);
      }

    /* see if the node is in the North East direction of
    the 2 * 2 mesh of processors */

      nodeNE = find_node(row, col+dim);
      if(mynode() == nodeNE)
        csend(nodeNE,array_rep,sizeof(array_rep),0,my_pid);
      if(mynode() == 0){
        crecv(nodeNE,array0,sizeof(array0));
        root->child[7] = build_tree(array0,index=0);
        root->child[6] = copy(root->child[7]);
      }

    /* see if the node is in the South West direction of
    the 2 * 2 mesh of processors */

      nodeSW = find_node(row+dim, col);
      if(mynode() == nodeSW)
        csend(nodeSW,array_rep,sizeof(array_rep),0,my_pid);
      if(mynode() == 0){
        crecv(nodeSW,array0,sizeof(array0));
        root->child[1] = build_tree(array0,index=0);
        root->child[0] = copy(root->child[1]);
      }

    /* see if the node is in the South East direction of
    the 2 * 2 mesh of processors */

      nodeSE = find_node(row+dim, col+dim);
      if(mynode() == nodeSE)
        csend(nodeSE,array_rep,sizeof(array_rep),0,my_pid);
      if(mynode() == 0){
        crecv(nodeSE, array0, sizeof(array0));
        root->child[3] = build_tree(array0,index=0);
        root->child[2] = copy(root->child[3]);
      }

    }/* else */

    return(compact(root));

}
```

```
/* This is the corresponding routine to collect the
contributions from other nodes for the face Y octree. */

OCTREE collect_y(dim, row, col)
int   dim, row, col;
{

   int    i;
   OCTREE root,copy(),getnode(),build_tree(),compact();

   root = getnode('G');
   dim = dim/2;

   /* if dim is > the size of matrix of a node */
   if(dim > DIM/range) {
      root -> child[7] = collect_y(dim, row, col);
      root -> child[5] = copy(root->child[7]);
      root -> child[6] = collect_y(dim, row, col+dim);
      root -> child[4] = copy(root->child[6]);
      root -> child[3] = collect_y(dim, row+dim, col);
      root -> child[1] = copy(root->child[3]);
      root -> child[2] = collect_y(dim, row+dim, col+dim);
      root -> child[0] = copy(root->child[2]);
   }
   else {
     nodeNW = find_node(row, col);
     if(mynode() == nodeNW)
       csend(nodeNW,array_rep,sizeof(array_rep),0,my_pid);
     if(mynode() == 0){
       crecv(nodeNW, array0, sizeof(array0));
       root->child[7] = build_tree(array0,index=0);
       root->child[5] = copy(root->child[7]);
     }
     nodeNE = find_node(row, col+dim);
     if(mynode() == nodeNE)
       csend(nodeNE,array_rep,sizeof(array_rep),0,my_pid);
     if(mynode() == 0){
       crecv(nodeNE, array0, sizeof(array0));
       root->child[6] = build_tree(array0,index=0);
       root->child[4] = copy(root->child[6]);
     }

     nodeSW = find_node(row+dim, col);
     if(mynode() == nodeSW)
       csend(nodeSW,array_rep,sizeof(array_rep),0,my_pid);
     if(mynode() == 0){
       crecv(nodeSW, array0, sizeof(array0));
       root->child[3] = build_tree(array0,index=0);
       root->child[1] = copy(root->child[3]);
     }

     nodeSE = find_node(row+dim, col+dim);
     if(mynode() == nodeSE)
       csend(nodeSE,array_rep,sizeof(array_rep),0,my_pid);
```

```
      if(mynode() == 0){
        crecv(nodeSE, array0, sizeof(array0));
        root->child[2] = build_tree(array0,index=0);
        root->child[0] = copy(root->child[2]);
      }

   }/* else */

   return(compact(root));

}

/* This is the corresonding routine for collecting
contributions from other nodes for the face Z octree */

OCTREE collect_z(dim, row, col)
int   dim, row, col;
{

   int    i;
   OCTREE root,copy(),getnode(),build_tree(),compact();

   root = getnode('G');
   dim = dim/2;

   /* if dim is > the size of matrix of a node */
   if(dim > DIM/range) {
      root -> child[4] = collect_z(dim, row, col);
      root -> child[0] = copy(root->child[4]);
      root -> child[6] = collect_z(dim, row, col+dim);
      root -> child[2] = copy(root->child[6]);
      root -> child[5] = collect_z(dim, row+dim, col);
      root -> child[1] = copy(root->child[5]);
      root -> child[7] = collect_z(dim, row+dim, col+dim);
      root -> child[3] = copy(root->child[7]);
   }
   else {
     nodeNW = find_node(row, col);
     if(mynode() == nodeNW)
       csend(nodeNW,array_rep,sizeof(array_rep),0,my_pid);
     if(mynode() == 0){
       crecv(nodeNW, array0, sizeof(array0));
       root->child[4] = build_tree(array0,index=0);
       root->child[0] = copy(root->child[4]);
     }
     nodeNE = find_node(row, col+dim);
     if(mynode() == nodeNE)
       csend(nodeNE,array_rep,sizeof(array_rep),0,my_pid);
     if(mynode() == 0){
       crecv(nodeNE, array0, sizeof(array0));
       root->child[6] = build_tree(array0,index=0);
       root->child[2] = copy(root->child[6]);
     }
```

```
      nodeSW = find_node(row+dim, col);
      if(mynode() == nodeSW)
        csend(nodeSW,array_rep,sizeof(array_rep),0,my_pid);
      if(mynode() == 0){
        crecv(nodeSW, array0, sizeof(array0));
        root->child[5] = build_tree(array0,index=0);
        root->child[1] = copy(root->child[5]);
      }

      nodeSE = find_node(row+dim, col+dim);
      if(mynode() == nodeSE)
        csend(nodeSE,array_rep,sizeof(array_rep),0,my_pid);
      if(mynode() == 0){
        crecv(nodeSE, array0, sizeof(array0));
        root->child[7] = build_tree(array0,index=0);
        root->child[3] = copy(root->child[7]);
      }

   }/* else */

   return(compact(root));

}

/*  Based on the starting row and column indices, this
routine returns the node number */

find_node(row, col)
int     row, col;
{
   int   i;

   for(i=0; i < image.nodes; i++)
      if((matrix[i].row == row) && (matrix[i].col == col))
        return(i);

}


/* The purpose of this routine is to release storage in
postorder fashion */

rel_mem(tree)
OCTREE   tree;
{
   short   i;

   if(tree > NIL) {
     for(i=0; i < 8; i++)
        rel_mem(tree->child[i]);
     free(tree);
     tree = 0;
   }
}
```

```
/* Routine to convert the tree representation to the
equivalent array representation. Works on the idea that
if the root is stored at index I and its children will be
stored from index I*8+1 to I*8+8 */

int   conv_array(root)
OCTREE      root;
{

   short   j, retval= -1;

   if(root > NIL) {
     retval = ++elements;
     array_rep[retval].color=root->color;
     for(j=0; j < 8; j++)
       if((root->child[j]) && (root->child[j]->color != 'W'))
           array_rep[retval].child[j]=
                                conv_array(root->child[j]);
   } /* if */
   else
     return(0);
 }

/* Once the linear octree has been collected on node 0,it
is reconverted to octree representation by this recursive
routine */

OCTREE   build_tree(Array,index)
struct   X *Array,
int      index;
{
   OCTREE   root=0, getnode();
   short    i;

   if((Array[index].color == '\0') ||
                           (Array[index].color == 'W'))
     return(0);

   root=getnode(Array[index].color);
   if(root->color == 'G')
     for(i=0; i < 8; i++)
       if(Array[index].child[i] > 0)
           root->child[i]=
            build_tree(Array,Array[index].child[i]);

   return(root);

}

/* Merge_3 routine merges 3 octrees and produces one
octree. If all the tree nodes are gray, then this
routine is called recursively. If at least one of
the nodes is white, the resultant node in the octree
is white. If two nodes are black and one is gray, the
```

```
tree with gray node will be made root to an subtree. If
all the three nodes are black, the resultant node is
black. If two nodes are gray and one is black, the
routine merge_2 is called to perform merging on two
subtrees with the gray nodes as the root */

OCTREE merge_3(root_x, root_y, root_z)
OCTREE         root_x, root_y, root_z;
{
   OCTREE   root, copy(), getnode(), merge_2();
   short    i;
   char     type;

   if((!root_x) || (!root_y) || (!root_z) ||
      (root_x->color == 'W') || (root_y -> color == 'W')
    ||(root_z->color == 'W'))
      return(0);

   root = getnode(type='G');

   /* if all the THREE nodes are GRAY */
   if((root_x->color == 'G') && (root_y->color == 'G') &&
      (root_z -> color == 'G'))
     for(i=0; i < 8; ++)
       root->child[i]=merge_3(root_x->child[i],
            root_y->child[i],root_z->child[i]);
    else   /* if TWO GRAY and ONE BLACK node */
     if((root_x->color=='G') && (root_y->color=='G') &&
                    (root_z -> color == 'B'))

       for(i=0; i < 8; i++)
         root->child[i]=
            merge_2(root_x->child[i],root_y->child[i]);
   else
     if((root_x->color=='G') && (root_y->color=='B') &&
                    (root_z -> color == 'G'))
       for(i=0; i < 8; i++)
         root->child[i]=
            merge_2(root_x->child[i],root_z->child[i]);
   else
     if((root_x->color=='B') && (root_y->color=='G') &&
                    (root_z -> color == 'G'))
       for(i=0; i < 8; i++)
         root->child[i]=
            merge_2(root_y->child[i],root_z->child[i]);
   else   /* if TWO BLACK and ONE GRAY node */
     if((root_x->color=='G') && (root_y->color=='B') &&
                    (root_z -> color == 'B'))
       for(i=0; i < 8; i++)
         root->child[i]=copy(root_x->child[i]);
   else
     if((root_x->color=='B') && (root_y->color=='G') &&
                    (root_z -> color == 'B'))
       for(i=0; i < 8; i++)
```

```
            root->child[i]=copy(root_y->child[i]);
      else
        if((root_x->color=='B') && (root_y->color=='B') &&
                          (root_z -> color == 'G'))
          for(i=0; i < 8; i++)
            root->child[i]=copy(root_z->child[i]);
      else
        if((root_x->color=='B') && (root_y->color=='B') &&
                          (root_z -> color == 'B'))
            root -> color = 'B';

      return(root);
}

/* This routine merges two octrees into one. If at least
one of them is a white node, the resultant node is a
white node. If both are black nodes,the resultant node
in the octree is a black node. If one is black and other
is gray, then the subtree with grey node is made root to
a subtree as the result. If both are grey nodes, then this
routine is called recursively. */

OCTREE merge_2(tree_1,tree_2)
OCTREE         tree_1,tree_2;
{
      OCTREE   copy(), tree, getnode();
      short    i;
      char    type;


      if((!tree_1) || (!tree_2) || (tree_1->color == 'W') ||
         (tree_2->color == 'W'))
         return(0);

      tree = getnode(type='G');

      if((tree_1->color == 'B') && (tree_2->color == 'B'))
        tree->color = 'B';
      else
        if((tree_1->color == 'G') && (tree_2->color == 'G'))
          for(i=0; i < 8; i++)
             tree->child[i] =
                 merge_2(tree_1->child[i],tree_2->child[i]);
      else
        if((tree_1->color == 'G') && (tree_2->color == 'B'))
          for(i=0; i < 8; i++)
             tree->child[i]=copy(tree_1->child[i]);
      else
        if((tree_1->color == 'B') && (tree_2->color == 'G'))
          for(i=0; i < 8; i++)
             tree->child[i]=copy(tree_2->child[i]);

      return(tree);
}
```

```
/* Routine to allocate and initialize memory equivalent to the
size of a node */
OCTREE getnode(type)
char          type;
{
   int    I;
   OCTREE onenode;

   onenode = (OCTREE) malloc(sizeof(node));
   onenode -> color = type;   onenode->boxptr=0;
   for(I=0; I < 8; I++)
      onenode -> child[I] = NIL;
   return(onenode);

}

/*This routine makes a copy of the tree pointed by source and
returns the root of the copy */

OCTREE copy(source)
OCTREE source;
{

   int    I;
   char   type;
   OCTREE root, getnode();

   if((source <= 0) ||
      (source->color == 'W'))
     return(0);

   root=getnode(type=source->color);
   if(source->color == 'G')
     for(I=0; I < 8; I++)
        root->child[I]=copy(source->child[I]);

   return(root);

}
::::::::::::::
view.c
::::::::::::::
#include "defs.h"

extern   union v_point   view_pt;

static   short rot_mat[7][8]={7,3,5,1,6,2,4,0,
                              6,7,4,5,2,3,0,1,
                              5,4,7,6,1,0,3,2,
                              2,3,6,7,0,1,4,5,
                              3,2,1,0,7,6,5,4,
                              1,3,0,2,5,7,4,6,
                              2,0,3,1,6,4,7,5 };
```

```
/*Routine rotates the octree such that the view point is in
octant 7. Rotation is done by just relabeling the tree. */

view_rotate(root)
OCTREE      root;
{
   extern   OCTREE   getnode();
   OCTREE   temp;
   int      I, J, K;

   temp = getnode(root->color);
   for(K=0; K < 8; K++)
       temp->child[K] = root->child[K];

   for(I=0; I < 8; I++)
    root->child[I]=temp->child[rot_mat[view_pt.reg][I]];

   for(J = 0; J < 8; J++)
      if(root->child[J]) && (root->child[J]->color == 'G'))
        view_rotate(root->child[J]);

   return(root);
}

::::::::::::::
neibor.c
::::::::::::::
#include "defs.h"

         /* arranged in increasing distance to the viewer */
int      seq[]={7, 6, 5, 3, 4, 2, 1, 0};

/* direction assignment to all possible pair of suboctants.
For example 7,6,2 indicates 7 is in front (2) of 6 */

static   int   table[12][3]={7,6,2, 7,5,0, 7,3,4, 6,4,0,
                             6,2,4, 5,4,2, 5,1,4, 3,2,2,
                             3,1,0, 4,0,4, 2,0,0, 1,0,2};

/* Any subcube could be covered by atmost three subcubes (in
three directions) to be partially visible. This array
indicates the various combinations of pairs of suboctants in
the three directions. The first index is for direction and
index 0 is for cube covered in left
index 1 is for cube covered in front
index 2 is for cube covered on top */

static   int   comp_ind[3][4][2]={5,7, 4,6, 1,3, 0,2,
                                  6,7, 4,5, 2,3, 0,1,
                                  3,7, 2,6, 1,5, 0,4};
           /*  X,Y,Z  */
static   int   sig[8][3] = {0,0,0, 1,0,0, 0,1,0, 1,1,0,
                            0,0,1, 1,0,1, 0,1,1, 1,1,1};
```

```
int   n_box;
BOX   head_graph=0, previous=0;

white_node(ptr)
OCTREE      ptr;
{
    if(ptr==NIL)
     return(TRUE);
   else
     if(ptr->color == 'W')
       return(TRUE);
   else
     return(FALSE);
}

find_neighbor(root, level, origin, length)
OCTREE        root;
int           level, *origin, length;
{
   int   i, j, org_out[3], first_org[3], second_org[3];

   if((root > NIL) && (root -> color == 'G')) {
     for(i=0; i < 8; i++) {
       org_fix(origin, seq[i], length, org_out);
       find_neighbor(root->child[seq[i]], level+1,
                     org_out, length/2);
       for(j=0; j < 12; j++) {
        if(table[j][0] == seq[i]) {
          org_fix(origin,table[j][0],length,first_org);
          org_fix(origin,table[j][1],length,second_org);
          two_comp(root->child[table[j][0]],
          root->child[table[j][1]],level+1,level+1,
          first_org,second_org,length/2,length/2,
          table[j][2]);
        } /* IF */
       }    /* FOR J = 0; J < 12; J++ */
      if((i==7)&&(root->color == 'B')){
        org_fix(origin, 0, length, first_org);
        two_comp(root->child[0],NIL,level+1,0,first_org,
        NIL,length/2,length/2,0);
       }
    } /* FOR I = 0; I < 8; I++ */
  } /* IF */
}

not_gray(ptr)
OCTREE   ptr;
{
  if((ptr<=NIL) || (ptr->color=='B') || (ptr->color=='W'))
     return(TRUE);
  else
     return(FALSE);
}
```

```
two_comp(first, second, first_level, second_level,
    first_org, second_org, first_len, second_len, ird)
OCTREE  first, second;
int     first_level,second_level,*first_org,*second_org,
        first_len,second_len,ird;
{

   int   i, org_1[3], org_2[3];
   BOX   neighb, center;

   if((not_gray(first)) && (not_gray(second))){
     if((first) && (first -> color == 'B')){
       if(first_len <= second_len)
         first -> child[ird+1] = second;

       if((white_node(first->child[0])) ||
         (white_node(first->child[2])) ||
         (white_node(first->child[4]))) {
         if(first->boxptr == NIL)
           new_graph(first, first_len, first_org);

      /* one face covered on the right */
         if((!white_node(first->child[0])) &&
           (white_node(first->child[2])) &&
             (white_node(first->child[4])))
           one_faceR(first);

      /* one face covered on the front */
         if((white_node(first->child[0])) &&
           (!white_node(first->child[2])) &&
             (white_node(first->child[4])))
           one_faceF(first);

      /* one face covered on the top */
         if((white_node(first->child[0])) &&
           (white_node(first->child[2])) &&
             (!white_node(first->child[4])))
           one_faceT(first);


         if((!white_node(first->child[0]))&&
           (!white_node(first->child[2])) &&
             (white_node(first->child[4])))
           two_faceRF(first);

         if((!white_node(first->child[0]))&&
           (white_node(first->child[2])) &&
             (!white_node(first->child[4])))
           two_faceRT(first);

         if((white_node(first->child[0]))&&
           (!white_node(first->child[2])) &&
             (!white_node(first->child[4])))
           two_faceFT(first);
```

```
        }   /* if child[0] || child[2] || child[4] == NIL */


        if((second) && (second -> color == 'B') &&
            (first_len < second_len)) {
          if(first->boxptr == NIL)
            new_graph(first, first_len, first_org);

          if(second->boxptr == NIL)
            new_graph(second, second_len, second_org);

          neighb = second -> boxptr;
          center = first -> boxptr;

          if(ird+1 == 1)
            far_faceL(neighb, center);

          if(ird+1 == 3)
            far_faceBk(neighb, center);

          if(ird+1 == 5)
            far_faceBt(neighb, center);

        } /* covered on the far_face */

      }/* if (first->color == 'B') */

      if((second) && (second-> color == 'B') &&
        (second_len <= first_len))
         second->child[ird] = first;

    }
/*if((first->color!='G')&&(second->color!='G'))*/
/* first == 'G' AND second == 'G' */
    if((!not_gray(first)) && (!not_gray(second))){
        for(i=0; i < 4; i++){
          org_fix(first_org,comp_ind[ird/2][i][0],
                  first_len,org_1);
          org_fix(second_org,comp_ind[ird/2][i][1],
                  second_len, org_2);
          two_comp(first->child[comp_ind[ird/2][i][0]],
             second->child[comp_ind[ird/2][i][1]],
             first_level+1, second_level+1, org_1, org_2,
             first_len/2, second_len/2, ird);
        } /* for */
    }

  /* first == 'G' AND second != 'G' */
    if((!not_gray(first)) && (not_gray(second)))
      for(i=0; i < 4; i++){
         org_fix(first_org,comp_ind[ird/2][i][0],
                 first_len, org_1);
```

```
            two_comp(first->child[comp_ind[ird/2][i][0]],
               second,first_level+1,second_level,org_1,
               second_org,first_len/2,second_len,ird);

         } /* for */

      /* first != 'G' AND second == 'G'*/
      if((not_gray(first)) && (!not_gray(second)))
        for(i=0; i < 4; i++){
            org_fix(second_org,comp_ind[ird/2][i][1],
                    second_len, org_2);
            two_comp(first,
               second->child[comp_ind[ird/2][i][1]],
               first_level,second_level+1,first_org,org_2,
               first_len, second_len/2, ird);
        }

}

org_fix(origin, sequence, length, org_out)
int    *origin, sequence, length, *org_out;
{
   int   i;

   for(i=0; i < 3; i++)
      org_out[i]=origin[i]+sig[sequence][i]*length/2;

}

new_graph(root, len, root_org)
OCTREE    root;
int       len, *root_org;
{

   short   i, j;
   BOX   boxnode;

   boxnode = (BOX) malloc(sizeof(box_node));

   if(previous > NIL){
      previous -> next = boxnode;
      previous = boxnode;
   }
   else{
      head_graph = previous = boxnode;
   }
   root -> boxptr = boxnode;
   boxnode -> origin[0] = root_org[0];
   boxnode -> origin[1] = root_org[1];
   boxnode -> origin[2] = root_org[2];
   boxnode -> len = len;    boxnode -> next = NIL;
   for(i=0; i < 9; i++){
      boxnode->edge[i] = (EDGE) malloc(sizeof(edge_node));
      boxnode->edge[i]->next = NIL; }
```

```
  boxnode->edge[0]->min=boxnode->edge[3]->min=
   boxnode->edge[6]->min=root_org[1];

   boxnode->edge[1]->min=boxnode->edge[4]->min=
   boxnode->edge[8]->min=root_org[2];

   boxnode->edge[2]->min=boxnode->edge[5]->min=
   boxnode->edge[7]->min=root_org[0];

   for(i=0; i < 9; i++)
      boxnode->edge[i]->max=boxnode->edge[i]->min+len;

   n_box++;

}

EDGE   new_edge()
{

   EDGE   edgenode;

   edgenode = (EDGE) malloc(sizeof(edge_node));
   return(edgenode);

}

/*This routines processes the cube covered in Right */
one_faceR(first)
OCTREE first;
{

   BOX   neighb, center;
   short   log2, log0;

   neighb = first->child[0]->boxptr;
   center = first->boxptr;

   if(neighb){
   /* 1 if top surfaces match*/
     log2 = (neighb->origin[2]+neighb->len ==
             center->origin[2]+center->len) ? 1 : 0;
   /* 1 if front surfaces match*/
     log0 = (neighb->origin[0]+neighb->len ==
             center->origin[0]+center->len) ? 1 : 0;
     if(log2)    /* aligned along the X direction */
       remspl(neighb, 5, center, 7);

     if(log0)    /* aligned along the Z direction */
       remspl(neighb, 4, center, 8);
   }
   edge_zap(center->edge[1]);
   edge_zap(center->edge[2]);
}
```

```
/*This routine processes the cube covered in Front */
one_faceF(first)
OCTREE first;
{

   BOX     neighb, center;
   short   log2, log1;

   neighb = first->child[2]->boxptr;
   center = first->boxptr;

   if(neighb){

     /* 1 if top surfaces match*/
     log2 = (neighb->origin[2]+neighb->len ==
             center->origin[2]+center->len) ? 1 : 0;

     /* 1 if right surfaces match*/
     log1 = (neighb->origin[1]+neighb->len ==
             center->origin[1]+center->len) ? 1 : 0;

     if(log2)    /* aligned along the Y direction */
       remspl(neighb, 0, center, 6);

     if(log1)    /* aligned along the Z direction */
       remspl(neighb, 1, center, 8);

   } /* if */
   edge_zap(center->edge[3]);
   edge_zap(center->edge[4]);

}

/*This routine processes the cube covered on Top */
one_faceT(first)
OCTREE    first;
{

   BOX     neighb, center;
   short   log1, log0;

   neighb = first->child[4]->boxptr;
   center = first->boxptr;

   if(neighb) {

     /* 1 if right surfaces match*/
     log1 = (neighb->origin[1]+neighb->len ==
             center->origin[1]+center->len) ? 1 : 0;

     /* 1 if front surfaces match*/
     log0 = (neighb->origin[0]+neighb->len ==
             center->origin[0]+center->len) ? 1 : 0;
```

```
      if(log0)    /* aligned along the Y direction */
        remspl(neighb, 3, center, 6len);

      if(log1)    /* aligned along the X direction */
        remspl(neighb, 2, center, 7);

     } /* if */
     edge_zap(center->edge[0]);
     edge_zap(center->edge[5]);
}

/* code for handling neighbors farther from the viewer */

/* This routine processes the cube covered on left */
far_faceL(neighb,center)
BOX     neighb,center;
{

   short   log0, log2;

   /* 1 if top surfaces match*/
   log2 = (neighb->origin[2]+neighb->len ==
           center->origin[2]+center->len) ? 1 : 0;

   /* 1 if front surfaces match*/
   log0 = (neighb->origin[0]+neighb->len ==
           center->origin[0]+center->len) ? 1 : 0;

   if(log2)    /* aligned along the X direction */
     remspl(neighb, 7, center, 5len);

   if(log0)    /* aligned along the Z direction */
     remspl(neighb, 8, center, 4);

}

/* This routine processes the cube covered in back */
far_faceBk(neighb,center)
BOX        neighb,center;
{
   short   log1, log2;

   /* 1 if top surfaces match*/
   log2 = (neighb->origin[2]+neighb->len ==
           center->origin[2]+center->len) ? 1 : 0;

   /* 1 if right surfaces match*/
   log1 = (neighb->origin[1]+neighb->len ==
           center->origin[1]+center->len) ? 1 : 0;

   if(log2)    /* aligned along the Y direction */
     remspl(neighb, 6, center, 0);
```

```
   if(log1)    /* aligned along the Z direction */
     remspl(neighb, 8, center, 1);

}

/* This routine processes the cube covered in bottom */
far_faceBt(neighb,center)
BOX        neighb,center;
{
   short   log0, log1;

   /* 1 if front surfaces match*/
   log0 = (neighb->origin[0]+neighb->len ==
           center->origin[0]+center->len) ? 1 : 0;

   /* 1 if right surfaces match*/
   log1 = (neighb->origin[1]+neighb->len ==
           center->origin[1]+center->len) ? 1 : 0;

   if(log0)    /* aligned along the Y direction */
     remspl(neighb, 6, center, 3);

   if(log1)    /* aligned along the X direction */
     remspl(neighb, 7, center, 2);

}

/*This routine processes the linklist of edges pointed by
neighbcurr and centercurr */

remspl(neighb, neighbnum, center, centernum)
BOX    neighb, center;
int    neighbnum, centernum;
{

   EDGE neighbcurr=neighb->edge[neighbnum],
        centercurr=center->edge[centernum];

   for(   ; neighbcurr > 0; neighbcurr=neighbcurr->next)
      for(centercurr=center->edge[centernum];centercurr>0;
              centercurr=centercurr->next)
         rem_do(neighbcurr,centercurr);

}

/*This routine compares the lenght of two edge segments and
shortens (elongates) one or other */

rem_do(neighb, center)
EDGE   neighb, center;
{
   short  code1, code2;
   int    temp;
   EDGE   newedge;
```

```
if((neighb->max<=center->min) ||
  (neighb->min>=center->max) ||
  (neighb->min==neighb->max) ||
  (center->min == center->max))
  return(0);

if(neighb->max > center->max)
   code1 = 0;
else
  if(neighb->max == center->max)
     code1 = 1;
else
  code1 = 2;

if(neighb->min > center->min)
   code2 = 0;
else
  if(neighb->min == center->min)
     code2 = 1;
else
  code2 = 2;

if((code1 == 1) && (code2 == 1)) { /* delete both */
  edge_remove(center);
  edge_remove(neighb);
}
else        /* delete one and shorten other */
  if((code1 == 0) && (code2 == 1)) {
     neighb->min = center->max;
     edge_remove(center);
  }

else
  if((code1 == 1) && (code2 == 2)) {
     neighb->max = center->min;
     edge_remove(center);
  }
else
  if((code1 == 2) && (code2 == 1)) {
    center->min = neighb->max;
    edge_remove(neighb);
  }
else
  if((code1 == 1) && (code2 == 0)) {
    center->max = neighb->min;
    edge_remove(neighb);
  }
else
  /*splitting an edge */
  if((code1 == 0) && (code2 == 2)) {
    newedge = new_edge();
    newedge->next = neighb->next;
    neighb->next = newedge;
    newedge->max = center->min;
```

```
        newedge->min = neighb->min;
        neighb->min = center->max;
        edge_remove(center);
      }
    else
      if((code1 == 2) && (code2 == 0)) {
        newedge = new_edge();
        newedge->next = center->next;
        center->next = newedge;
        newedge->max = neighb->min;
        newedge->min = center->min;
        center->min = neighb->max;
        edge_remove(neighb);
      }
    else
      /*overlap but no deletion */
      if((code1 == 0) && (code2 == 0)) {
        temp=neighb->min;
        neighb->min=center->max;
        center->max=temp;
      }
    else
      if((code1 == 2) && (code2 == 2)) {
        temp=center->min;
        center->min=neighb->max;
        neighb->max=temp;
      }

}

/*This routine removes an edge pointed by edge_ptr. This is
done by setting the maximum and minimum equal. */

edge_remove(edge_ptr)
EDGE        edge_ptr;
{
   edge_ptr->max =  edge_ptr->min;
}

/*This routine removes the linklist of edge pointed by
edge_ptr. This is done by setting the maximum and minium
equal. */

edge_zap(edge_ptr)
EDGE     edge_ptr;
{
   EDGE   temp;

   for(temp=edge_ptr; temp > 0; temp=temp->next)
       temp->max = temp->min;
}

/* This routine processes the cube covered in Right and Front
directions */
```

```
two_faceRF(first)
OCTREE     first;
{
   BOX   neighb1, neighb2, center;

   neighb1 = first->child[0]->boxptr;
   neighb2 = first->child[2]->boxptr;
   center = first->boxptr;
   /* see if one pair of top surfaces matches */
   if((neighb1) &&
      (neighb1->origin[2]+neighb1->len ==
       center->origin[2]+center->len))
      remspl(neighb1,5,center,7);

   /* see if the other pair of top surfaces matches */

   if((neighb2) &&
     (neighb2->origin[2]+neighb2->len ==
      center->origin[2]+center->len))
      remspl(neighb2,0,center,6);

   edge_zap(center->edge[1]);
   edge_zap(center->edge[2]);
   edge_zap(center->edge[3]);
   edge_zap(center->edge[4]);
   edge_zap(center->edge[8]);
}

/*This routine processes the cube covered in Right and Top
direcions */

two_faceRT(first)
OCTREE     first;
{

   BOX   neighb1, neighb2, center;

   neighb1 = first->child[0]->boxptr;
   neighb2 = first->child[4]->boxptr;
   center = first->boxptr;

   /* see if one pair of front surfaces matches */

   if((neighb1) &&
     (neighb1->origin[0]+neighb1->len ==
      center->origin[0]+center->len))
      remspl(neighb1,4,center,8);

   /* see if the other pair of front surfaces matches */

   if((neighb2) &&
      (neighb2->origin[0]+neighb2->len ==
       center->origin[0]+center->len))
       remspl(neighb2,3,center,6);
```

```
    edge_zap(center->edge[0]);
    edge_zap(center->edge[1]);
    edge_zap(center->edge[2]);
    edge_zap(center->edge[5]);
    edge_zap(center->edge[7]);
}

/*This routine processes the cube covered in Front and Top
directions */

two_faceFT(first)
OCTREE     first;
{
    BOX   neighb1, neighb2, center;

    neighb1 = first->child[2]->boxptr;
    neighb2 = first->child[4]->boxptr;
    center = first->boxptr;

    /* see if one pair of right surfaces matches */
    if((neighb1) &&
       (neighb1->origin[1]+neighb1->len ==
        center->origin[1]+center->len))
        remspl(neighb1,1,center,8);

    /* see if the other pair of right surfaces matches */

    if((neighb2) &&
       (neighb2->origin[1]+neighb2->len ==
        center->origin[1]+center->len))
        remspl(neighb2,2,center,7);

    edge_zap(center->edge[0]);
    edge_zap(center->edge[3]);
    edge_zap(center->edge[4]);
    edge_zap(center->edge[5]);
    edge_zap(center->edge[6]);
}
::::::::::::::
proj.c
::::::::::::::
#include  "defs.h"
#include  "math.h"
#define    scale    100
#define    Xoff   18000
#define    Yoff   10000

/* this part performs the projections --our viewpoint is
in the (+++) octant, and the view up direction is parallel
to the positive z axis as it is before transformation */

vecsum(v1,v2,v3)
double v1[3],v2[3],v3[3];
```

```
{
   int   i;

   for(i=0; i < 3; i++)
      v3[i]=v1[i]+v2[i];
}

vecdif(v1,v2,v3)
double v1[3],v2[3],v3[3];
{
   int   i;

   for(i=0; i < 3; i++)
      v3[i]=v1[i]-v2[i];
}

cross_prod(v1,v2,v3)
double v1[3],v2[3],v3[3];
{
   v3[0] = v1[1]*v2[2] - v1[2]*v2[1];
   v3[1] = v1[2]*v2[0] - v1[0]*v2[2];
   v3[2] = v1[0]*v2[1] - v1[1]*v2[0];
}

double vecmag(v1)
double v1[3];
{
   return(sqrt(v1[0]*v1[0]+v1[1]*v1[1]+v1[2]*v1[2]));
}

double dotprod(v1,v2)
double v1[3],v2[3];
{
   return(v1[0]*v2[0]+v1[1]*v2[1]+v1[2]*v2[2]);
}

/* ptr points to the beginning of the list of box nodes */

double   infty=1.0e30;

/* This routine projects the visible edge segments on the
screen and stores the screen coordinates in them. */

project(ptr, view)
BOX     ptr;
double   *view;
{
   static   double z_ax[3]={0.,0.,1.};
   double   rz[3],rx[3],ry[3],temp[3],temp2[3];
   EDGE     edgept;
   double   viewlen, xlen;
   short    i, j;              /* for edges */
   static   short   copy[9]={1,2,0,1,2,0,1,0,2},
```

```
/* 1,2,1,0 means Y & Z components and ONLY Y changes
the origin is in the center of the cube and comparison
is done with this as the reference point */

         load[9][4]={0,2,0,1,  0,1,0,1,  1,2,1,0,
                     0,2,1,0,  0,1,1,0,  1,2,0,1,
                     0,2,1,1,  1,2,1,1,  0,1,1,1},

         /* coordinates for vertices */
         disp[6][3]={0,0,1, 0,1,1, 0,1,0,
                     1,1,0, 1,0,0, 1,0,1};

viewlen = vecmag(view);
cross_prod(view,z_ax,rx);
xlen = vecmag(rx);
for(i=0; i < 3; i++){
/*transformation to screen coordinate system */
   rz[i] = view[i]/viewlen;
   rx[i] = rx[i]/xlen;
}
cross_prod(rz,rx,ry);
while(ptr > NIL){
  for(i=0; i < 9; i++){
    edgept = ptr->edge[i];
    while(edgept > NIL){
      if(edgept->min != edgept->max){
        temp[copy[i]] = edgept->min;
        for(j=0; j < 2; j++){
          if(load[i][j+2])
            temp[load[i][j]]=
                ptr->origin[load[i][j]]+ptr->len;
          else
            temp[load[i][j]]=
                ptr->origin[load[i][j]];
        }
        vecdif(temp,view,temp2);
        edgept->xmin= -dotprod(temp2,rx)*scale+Xoff;
        edgept->ymin= Yoff-dotprod(temp2,ry)*scale;

        temp[copy[i]]=edgept->max;
        vecdif(temp,view,temp2);
        edgept->xmax= -dotprod(temp2,rx)*scale+Xoff;
        edgept->ymax= Yoff-dotprod(temp2,ry)*scale;
      } /* if */

     edgept = edgept->next;
   }   /* while */
} /* for */

for(i=0; i < 6; i++){
  for(j=0; j < 3; j++)
```

```
        if(disp[i][j])
          temp[j]=ptr->origin[j]+ptr->len;
        else
          temp[j]=ptr->origin[j];

      vecdif(temp,view,temp2);

      ptr->corner[i][0]= -dotprod(temp2,rx)*scale+Xoff;
      ptr->corner[i][1]= Yoff-dotprod(temp2,ry)*scale;
   }

   box_values(ptr);
   ptr=ptr->next;
  }   /* outter while */

}

box_values(ptr)
BOX     ptr;
{

   short   i;
   double  xsmall=infty, ysmall=infty,
           xlarge= -infty, ylarge= -infty;

   for(i=0; i < 6; i++){
      if(ptr->corner[i][1] > ylarge) {
        ptr->yhigh = ylarge = ptr->corner[i][1];
        ptr->xhigh = ptr->corner[i][0];
      }

      if(ptr->corner[i][1] < ysmall) {
        ptr-> ylow = ysmall = ptr->corner[i][1];
        ptr->xlow = ptr->corner[i][0];
      }

      if(ptr->corner[i][0] > xlarge) {
        ptr->xright = xlarge = ptr->corner[i][0];
        ptr->yright = ptr->corner[i][1];
      }

      if(ptr->corner[i][0] < xsmall) {
        ptr->xleft = xsmall = ptr->corner[i][0];
        ptr->yleft = ptr->corner[i][1];
      }

   }

}


::::::::::::::
clip.c
::::::::::::::
```

```
#include   "defs.h"
#define      Max(i,j)     (i > j) ? i : j
#define      Min(i,j)     (i < j) ? i : j

/* index 0 - slope; 1 - b */
/* flag = 1 : above the slanted line
         -1 : below the slanted line
          2 : right of the vertical line
         -2 : left of the vertical line */

extern int   seq[8];
extern BOX   previous, head_graph;

BOX    cur_box, prevpt;
EDGE   edgept;
short  bitpat0=0, bitpat1=0;
static short bitmask[6]={0x0001, 0x0002, 0x0004, 0x0008,
                         0x0010, 0x0020};

/*This routine traverses the linklist of boxnodes. Each
boxnode is compared against the boxnodes closer to the viewer.
The linklist of edges pointed by this boxnode is then
shortened or removed based on Cohen-Sutherland code */

hidden_clip(root)
OCTREE      root;
{
   int   j;

   previous = head_graph;
   boxgen(root);
   for(cur_box=head_graph->next; cur_box > NIL;
       cur_box=cur_box->next)
      for(prevpt=head_graph; prevpt > 0;
          prevpt=prevpt->next)
        if((prevpt->origin[0] >= cur_box->origin[0]) &&
             (prevpt->origin[1] >= cur_box->origin[1]) &&
             (prevpt->origin[2] >= cur_box->origin[2]) &&
           (prevpt != cur_box)) {
          for(j=0; j < 9; j++)
          /* 9 linklist of edges pointed by this boxnode */
             for(edgept=cur_box->edge[j]; edgept > 0;
                 edgept=edgept->next){
               /* If it is a valid edge */
                 if(edgept->min != edgept->max){
                    bitpat0=bitpat1=0;
                    obscure(edgept);
                 }
          }    /* for */
        } /* if */
 }
```

```
boxgen(ptr)
OCTREE   ptr;
{

   double slope;
   int    j, i;
   BOX    temp;

   if(ptr->color == 'G')
     for(j=0; j < 8; j++)
        if(ptr->child[seq[j]])
          boxgen(ptr->child[seq[j]]);
   else
     if((ptr->color == 'B') && (ptr->boxptr > NIL)){
       temp = ptr->boxptr;
       for(i=0; i < 6; i++)
         switch (i) {

      case 0 :
      case 5 : temp->flag[i] = -1;
               if(temp->corner[(i+1)%6][0] ==
                     temp->corner[i][0])
                   slope=0.;
               else
                   slope=(temp->corner[(i+1)%6][1]-
                          temp->corner[i][1])/
                         (temp->corner[(i+1)%6][0]-
                          temp->corner[i][0]);

               temp->suth_const[i][0]=slope;
               temp->suth_const[i][1]=
               -slope*temp->corner[i][0]+temp->corner[i][1];
                break;

      case 2 :
      case 3 : temp->flag[i] = 1;
               if(temp->corner[i+1][0] == temp->corner[i][0])
                   slope=0.;
               else
                 slope=(temp->corner[(i+1)][1]-
                        temp->corner[i][1])/
                       (temp->corner[(i+1)][0]-
                        temp->corner[i][0]);
                 temp->suth_const[i][0]=slope;
                 temp->suth_const[i][1]= -slope*
                   temp->corner[i][0]+temp->corner[i][1];
           break;


      case 1 :
      case 4 : if(temp->corner[i+1][0] == temp->corner[i][0]){
                 if(i==1)
                   temp->flag[1] = -2;
                 else
```

```
                    temp->flag[4]= 2;
                  temp->suth_const[i][1]=temp->corner[i][0];
                }
                else{     /* for perspective projection */
                    slope=(temp->corner[(i+1)][1]-
                           temp->corner[i][1])/
                          (temp->corner[(i+1)][0]-
                           temp->corner[i][0]);
                    temp->suth_const[i][0]=slope;
                    temp->suth_const[i][1]=
                      -slope*temp->corner[i][0]+
                    temp->corner[i][1];
                    if(temp->corner[i+1][0]>
                         temp->corner[i][0])
                       temp->flag[i] = -1;
                    else
                      temp->flag[i]=1;
                }
                break;
        } /* switch */
      }
    else
      return(0);

}

obscure(ptr)
EDGE     ptr;
{

   extern   int   edge_remove();

   if(ptr->min == ptr->max)
     return(0);

   if(((Max(ptr->xmax,ptr->xmin)) <= prevpt->xleft) ||
     ((Min(ptr->xmax,ptr->xmin)) >= prevpt->xright) ||
     ((Max(ptr->ymax,ptr->ymin)) <= prevpt->ylow) ||
     ((Min(ptr->ymax,ptr->ymin)) >= prevpt->yhigh))
      return(0);

   Co_Suth(ptr->xmin,ptr->ymin,&bitpat0);
   Co_Suth(ptr->xmax,ptr->ymax,&bitpat1);

   if((bitpat0 & bitpat1) != 0){
      return(0);
   }
   if((bitpat0 == 0) && (bitpat1 == 0)) {
      edge_remove(ptr);
      return(1);
   }
   if((bitpat0 != 0) && (bitpat1 != 0)) {
      split(ptr);
      return(1);
```

```
    }
    shorten(ptr);
}

/* Calculate the Cohen-Sutherland code and store it in bitpat
*/

Co_Suth(X,Y,bitpat)
double  X,Y;
short  *bitpat;
{
    int   I;

    for(I=0; I < 6; I++)
       switch (prevpt->flag[I]) {

         case  1 : if(prevpt->suth_const[I][0]*X-Y+
                          prevpt->suth_const[I][1] > 0)
                     (*bitpat) = (*bitpat) | bitmask[I];
                   break;

         case -1 : if(prevpt->suth_const[I][0]*X-Y+
                          prevpt->suth_const[I][1] < 0)
                     (*bitpat) = (*bitpat) | bitmask[I];
                   break;

         case  2 : if(X-prevpt->suth_const[I][1] < 0)
                     (*bitpat) = (*bitpat) | bitmask[I];
                   break;

         case -2 : if(X-prevpt->suth_const[I][1] > 0)
                     (*bitpat) = (*bitpat) | bitmask[I];
                   break;
       } /* switch */
}

shorten(ptr)
EDGE   ptr;
{

    double  xcent=0., ycent=0., fabs();
    double  xsmall=ptr->xmin, xbig=ptr->xmax;
    double  ysmall=ptr->ymin, ybig=ptr->ymax;
    short   bitsmall=bitpat0, bitbig=bitpat1, bitcent=0;

    if((fabs(xbig-xsmall) <= 1.0) && (fabs(ybig-ysmall)
                                      <=1.0)){
       ptr->max=ptr->min;
       return(0);
    }

    while(Max(fabs(xbig-xsmall),fabs(ybig-ysmall))>1.0){
       xcent = (xbig+xsmall)/2.0;
       ycent = (ybig+ysmall)/2.0;
```

```
        bitcent=0;
        Co_Suth(xcent,ycent,&bitcent);
        if(((bitcent == 0) && (bitpat0 == 0)) ||
           ((bitcent != 0) && (bitpat1 == 0))){
           xsmall = xcent;      ysmall = ycent;
           bitsmall = bitcent;
        }
        else {
          xbig = xcent;   ybig = ycent;
          bitbig = bitcent;
        }

    } /* while */

    if(bitpat0 == 0) {
      ptr->xmin=xcent; ptr->ymin=ycent;
    }
    else{
      ptr->xmax=xcent; ptr->ymax=ycent;
    }

}

split(ptr)
EDGE   ptr;
{
    double  xcent=0., ycent=0.;
    double  xsmall=ptr->xmin, xbig=ptr->xmax;
    double  ysmall=ptr->ymin, ybig=ptr->ymax;
    short   bitsmall=bitpat0, bitbig=bitpat1, bitcent=0;
    EDGE    newedge;
    extern  EDGE   new_edge();

    do{
      xcent = (xbig+xsmall)/2.0;
      ycent = (ybig+ysmall)/2.0;

      bitcent=0;
      Co_Suth(xcent,ycent,&bitcent);
      if(bitcent != 0) {
        if(bitcent & bitsmall){
          xsmall = xcent;      ysmall = ycent;
        }
        else {
          xbig = xcent;        ybig = ycent;
        }
      } /* if */

    } while ((bitcent != 0)&&(Max(fabs(xbig-xsmall),
              fabs(ybig-ysmall)) > 1.0));
    /* now (xcent,ycent) is inside the box */

    newedge=new_edge();
    newedge->next=ptr->next; ptr->next=newedge;
```

```
    newedge->xmax=ptr->xmax; newedge->ymax=ptr->ymax;
    newedge->xmin=xcent; newedge->ymin=ycent;
    newedge->min=ptr->min; newedge->max=ptr->max;
    ptr->xmax=xcent; ptr->ymax=ycent;

    bitpat1 = bitcent;
    shorten(ptr);
    bitpat0 = bitcent;
    bitpat1 = bitbig;
    shorten(newedge);

}
::::::::::::::
pict.c
::::::::::::::
#include   <stdio.h>
#include   <cgidefs.h>

main(argc,argv)
int   argc;
char *argv[];
{

   Ccoor     box[2];
   Ccoorlist boxlist;
   Cint      name;
   Cvwsurf   device;

   double    xmin, xmax, ymin, ymax;
   FILE      *input;

   boxlist.n = 2; /*number of points to be connected */
   boxlist.ptlist = box;
   NORMAL_VWSURF(device, CGPIXWINDD);
   if((argc < 2) || ((input=fopen(argv[1],"r")) <= 0)){
     printf("ERROR IN THE COMMAND LINE OR OPENING FILE \n");
     exit(1);
   }

   open_cgi();
   open_vws(&name, &device);

   while(fscanf(input,"%lf %lf %lf %lf\n",
         &xmin,&ymin,&xmax,&ymax) > 0) {
       box[0].x = xmin;   box[0].y = ymin;
       box[1].x = xmax;   box[1].y = ymax;
       polyline(&boxlist); /* draw the line */
   }

   getchar();  /* wait for user's response */
   close_vws(name);
   close_cgi();

}
```

VITA

Ramesh L. Parmar

Candidate for the Degree of

Master of Science

Thesis: OCTREE REPRESENTATION AND DISPLAY OF THREE DIMENSIONAL OBJECTS

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Madras, India, May 4, 1966, son of Mr. and Mrs. Lakshmichand R. Parmar.

Education: Graduated from A. G. Jain Higher Secondary School, Madras, in June 1983; received Bachelor of Technology Degree in Chemical Engineering from Regional Engineering College, Tiruchirapalli, May 1987; completed requirements for the Master of Science degree at Oklahoma State University in December, 1989.

Professional Experience: In-plant Trainee, Kothari Chemicals Pvt. Ltd., May 1985 to July 1985; Research Assistant, School of Business Administration, Summer 1988; Teaching Assistant, Department of Computing and Information Sciences, Oklahoma State University, January 1988 to May 1989.