Line Drawings of Octree-Represented Objects

JACK VEENSTRA and NARENDRA AHUJA
University of Illinois

The octree structure represents the space occupied by an object as a juxtaposition of cubes, where the sizes and position coordinates of the cubes are integer powers of 2 and are defined by a recursive decomposition of three-dimensional space. This makes the octree structure highly sensitive to object location and orientation, and the three-dimensional shape of the represented object obscure. It is helpful to be able to see the actual object represented by an octree, especially for visual performance evaluation of octree algorithms. Presented in this paper is a display algorithm that helps visualize the three-dimensional space represented by the octree. Given an octree, the algorithm produces a line drawing of the objects represented by the octree, using parallel projection, from any specified viewpoint with hidden lines removed. The order in which the algorithm traverses the octree has the property that if node x occludes node y, then node x is visited before node y. The algorithm produces a set of long, straight visible edge segments corresponding to the visible surface of the polyhedral object represented by the octree. Examples of some line drawing produced by the algorithm are given. The complexity of the algorithm is also discussed.

Categories and Subject Descriptors: I.2.9 [**Artificial Intelligence**]: Robotics—*sensors*; I.2.10 [**Artificial Intelligence**]: Vision and Scene Understanding—*representations, data structures and transforms*; *shape*; I.4.9 [**Image Processing**]: Applications

General Terms: Algorithms

Additional Key Words and Phrases: Hidden line removal, line drawing, octree, three-dimensional representation

1. INTRODUCTION

The octree representation is a hierarchical description of the three-dimensional space occupied by objects. It decomposes the volume of the given objects using a cubical tessellation. Thus the volume is expressed as a union of a set of cubical blocks whose positions and sizes are restricted variables [1, 9, 14]. 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 0–7 (see Figure 1). If an octant is completely inside the object, the corresponding node in the octree is marked black; if it is completely outside the object, the node

This work was supported by the National Science Foundation under grant ECS 83-52408 and AT&T Information Systems.
Authors' current addresses: J. Veenstra, AT&T Information Systems, Naperville, IL 60566; N. Ahuja, Coordinated Science Laboratory, University of Illinois, 1101 W. Springfield Ave., Urbana, IL 61801.

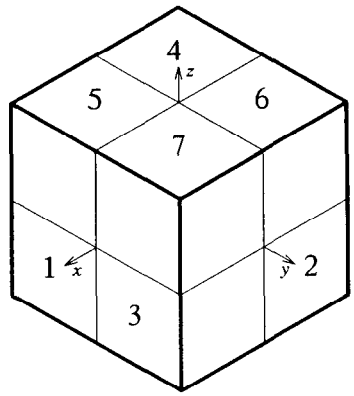

Fig. 1. A cube and its decomposition into octants.

is marked white. If the octant is partially contained in the object, the octant is decomposed into eight suboctants, each of which is again tested to determine whether 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 criteria. Since they represent space in terms of nonoverlapping volumetric primitives, octrees are well suited for performing geometric computations such as collision and intersection detection among three-dimensional objects.

We call the starting cubical region 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 (see Figure 2). Each octant corresponds to a node in the octree, and the node is assigned the label of the octant. Figure 2a shows a simple object. Figure 2b shows the same object enclosed in the universe cube, and Figure 2c shows the corresponding octree. The children nodes are arranged in increasing order of label values from left to right. The black nodes are shown as dark ovals, and the white and gray nodes are shown as empty ovals. In practice, of course, the white nodes need not be stored.

Because of restrictions on the positions and sizes of the cubes, even compact objects may require a large octree representation. Of course, the size of the octree varies with the position and orientation of the object; for some positions of the object the tree may be compact, whereas for other positions the tree may become very deep. Thus the restriction on the depth of the octree may have a small or large impact on the quality of the representation according to the position and orientation of the object.

A decomposition of an object shape into more naturally defined components than a fixed set of cubes would be desirable. For example, generalized cylinders [3] provide a good description of a variety of shapes. However, these representations are hard to derive from images. Since the purpose of octrees is to perform coarse occupancy analysis, octrees reduce the complexity of derivation such as characterizes the generalized cylinder representation for coarseness of the derived occupancy map. A by-product is that the representation makes the spatial

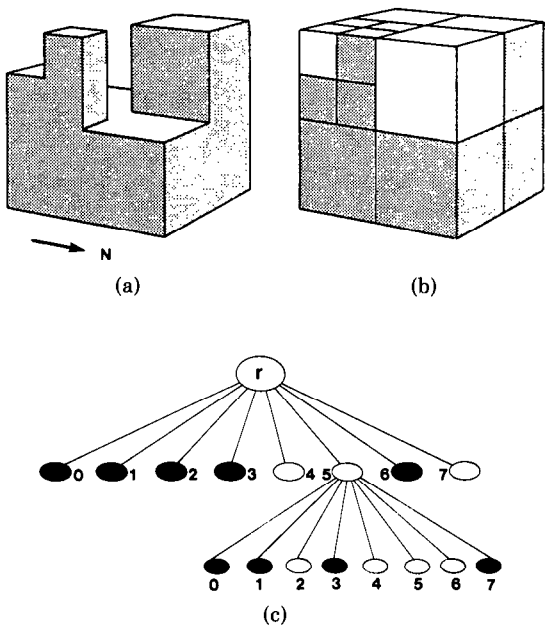

Fig. 2. (a) An object; (b) object enclosed in universe cube; (c) its octree representation (from [1]).

information obscure. Visualization of the three-dimensional shapes represented is difficult and requires analysis. The sensitivity of the octree to object location and orientation further compounds the problem.

In this paper we describe an algorithm that serves as an interface between the octree and the observer. It transforms the octree into a line drawing representation of the object, thus mapping all octree representations of the same shape (at different locations) onto a line drawing displayed by using parallel projection from any desired viewpoint and with hidden lines removed. The motivation for the algorithm came from the need for monitoring the performance of another algorithm we have developed that generates the octree of an object from its silhouette images [2, 16]. Other octree algorithms [4, 8, 15] should also benefit from the work reported in this paper.

Initially, we did the performance evaluation by printing each node in the octree with its associated label and verifying by hand that the octree was correct. As the octrees became larger, however, it became necessary to be able to view directly the object that the octree represented.

An alternative method of displaying the object represented by an octree is described by Meagher [11, 12]. His algorithm produces a surface display from octree after hidden surface removal and is designed to be implemented in VLSI processors for real-time applications. Meagher uses a quadtree to represent the display screen. Each octree node is projected onto the screen, and the quadtree nodes that are contained within that projection are assigned an intensity value, unless an intensity value has already been assigned. Thus, the first octree node to project onto the area covered by a quadtree node determines the intensity for that region of the screen. Since octree nodes closer to the viewer are processed first, hidden nodes are not displayed.

Doctor and Torborg [5] also use a surface display algorithm that makes use of a quadtree to represent the image. Their algorithm includes an added 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 weighting factor on the basis of the thickness of the octree region, which represents the degree of opaqueness.

Surface display algorithms, however, depend on light source positions. In addition, many output devices cannot draw shaded surfaces. A line drawing representation, on the other hand, captures the essential details of the object structure in the form of edges, since the objects are polyhedral. We therefore chose to display the objects represented by the octree as line drawings that can be easily drawn.

2. THE LINE DRAWING ALGORITHM

The algorithm consists of the following steps. First, the octree is traversed, using a recursive, front-to-back traversal method similar to the one described by Meagher [11, 12]. This traversal has the property that if node x occludes node y, then node x is visited first. For each black leaf node, graphics information is collected and stored in a "graphics node." (To avoid confusion with octree nodes, we call the data structure containing the graphics information a "graphics node.") When a graphics node is created, it is added to the end of a linked list of graphics nodes. As a consequence of the tree traversal method, this linked list also has the property that, if node x occludes node y, then the graphics node for x occurs earlier in the list than the graphics node for y. By traversing the tree in this manner, we take advantage of the spatial organization of the octree, which simplifies the removal of hidden lines later on. During tree traversal, black leaf nodes are "threaded" to point to their neighbors. This allows the elimination of "cracks," discussed below, and is useful in the final stage when the line segments are displayed.

After the linked list of graphics nodes has been created, each such node is projected onto the image screen and the screen coordinates of the vertices of the projection are stored in the graphics node. Each graphics node represents a cube that projects, in general, as a hexagon. The numbering schemes for the corners and edges of a projected cube are given in Figure 3. 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 graphics node in the linked list against nodes closer to the beginning of the list. Since a graphics node y may be occluded by another graphics node x that is closer to the beginning of the list, any overlap of node y with node x represents a hidden part of node y and is therefore removed.

The coordinate frame of reference for the octree is such that the coordinates of the viewer are always positive. The universe cube is rotated, if necessary, so that the viewpoint falls in the positive octant. Since this requires rotation by multiples of 90°, it is performed by simply relabeling the octants. We now present a more detailed description of the algorithm.

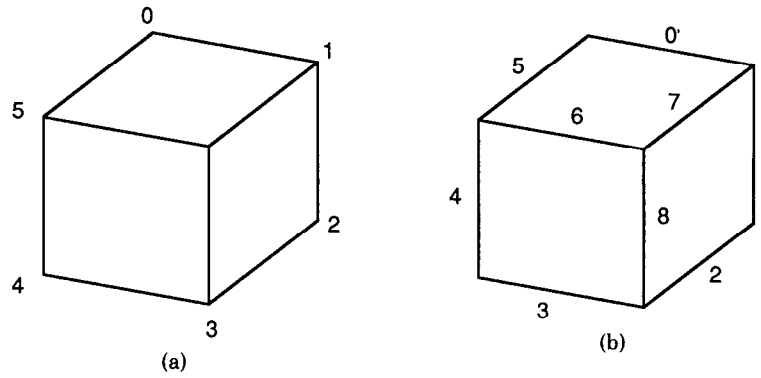

Fig. 3. The labeling scheme for the corners (a) and edges (b) of a projected octant.

2.1 Elimination of "Cracks"

A problem unique to line drawings generated from octrees is the elimination of "cracks" from the drawing. A "crack" is a line that 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. 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.

To eliminate cracks, all the neighbors must be found and tested to see whether they share a common border. In our algorithm we do this by traversing the tree and "threading" black leaf nodes to point to their neighbors. We use six of the eight unused child pointers of the black leaf node to point to neighbors in the six directions corresponding to the faces of a cube. Since the black nodes have no children, these pointers are known to be threads. The threaded octree turns out to be useful for other reasons as well. We use a seventh child pointer to point to the graphics node that is created at the time the black octant is first encountered.

Only black octants have associated graphics nodes, and a black octant that is surrounded on all six sides by other black octants is skipped since it will not show in the display.

After cracks are removed, the projections of the black octants are calculated and stored in their respective graphics nodes. Each graphics node also contains a pointer to the octant it represents so that the neighbors of a graphics node can be found quickly. This facilitates the plotting of long, straight lines as a unit instead of a sequence of short, contiguous line segments.

2.2 Elimination of Hidden Lines

After cracks are removed and the projections are calculated, the hidden lines are removed by using a straightforward edge intersection technique [6, 13]. Each edge of a projected octant is tested for intersection with projections of all other octants whose graphics nodes occur earlier in the list. Thus the computation time

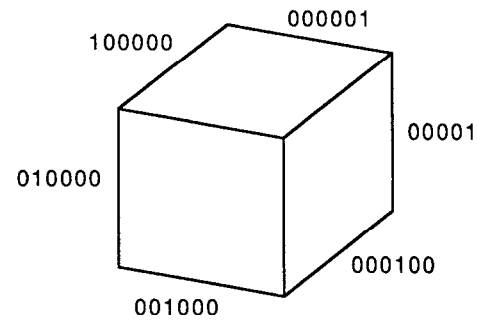

Fig. 4. The bit codes for the six half-planes defined by the edges of a projected octant.

to eliminate hidden lines is proportional to the square of the number of graphics nodes. In practice, however, we have observed that the computation time is more closely related to the complexity of the image (see Section 3.2).

A modified Cohen–Sutherland clipping algorithm is used to carry out the intersection tests (see Figure 4). The line that contains an edge of a projected octant defines two half-planes, one of which contains the projection and one that does not. The six edges define six half-planes that do not contain the projected octant. Each of these half-planes is assigned a different bit in a 6-bit code. The code for a point is the logical OR of the bit codes of the half-planes that contain that point.

Given an edge segment, we calculate the bit codes for its two endpoints. If the edge is completely outside the projection (and therefore visible), the logical AND of the two bit codes will be nonzero. If the edge is completely within the projection (and therefore hidden), then both bit codes will be zero. Otherwise, the edge partially overlaps the projection. The overlapping segment of the edge corresponds to its hidden part and must be removed.

If an edge segment partially overlaps the projected octant, then the intersection with the projection must be calculated. This can be done by taking the intersection of the edge segment with the appropriate edge of the projection that corresponds to a nonzero bit in the bit code for an endpoint of the edge segment.

2.3 Elimination of "Dots"

After hidden lines are removed, the data are ready for display. At this point the graphics nodes contain pointers to all the visible edge segments. On certain output devices (notably moving-pen plotters) an aesthetic problem will arise if the edges are output in the obvious sequence of their order of occurrence in the linked list of graphics nodes. Because the octree representation decomposes the object into various sized cubes, a long, smooth line on the object may be broken up and represented as several pieces—each piece in a different octree node. Therefore, instead of plotting one long line segment, several short line segments are plotted. On a moving-pen plotter, a small dot is visible at the start and end of every line segment. Not only is this displeasing to the eye, but, in addition, when a long line is plotted as several dozen short segments, the tip of the pen takes a beating. These problems do not arise on a graphics terminal or on a laser printer, but plotting short segments has other undesirable traits that apply to all output devices. On all output devices, plotting a long list of short line

segments will, in general, be less efficient (slower) than plotting a single long line segment. Furthermore, since each segment is processed separately, the plotted segments may have slightly different slopes and may not line up exactly, thereby giving a jagged, broken appearance to what should be a smooth straight line.

The solution to these problems is to logically connect contiguous line segments before plotting. The threaded octree is useful for this purpose since it allows us to check the neighboring octant and connect adjacent edge segments. The result is a line drawing without any jagged edges.

2.4 Representation of Graphics Information

The above algorithm was implemented in the C programming language [10] under the UNIX[1] operating system. Details of our implementation are given below.

To facilitate integer instead of floating-point representations, the side length of an octant at the lowest level in the octree (i.e., the smallest possible octant) is defined to be 1. The center of the octree is defined to be the origin of the octree coordinate system.

The graphics node used to represent an octant and its projection onto the screen coordinate system is defined by the following C structure:

```
typedef struct box {
  OCTREE *oct;
  int origin[3];
  int len;
  float corners[6][2];
  float xhigh, yhigh, xlow, ylow;
  float xleft, yleft, xright, yright;
  EDGE *edges[9];
  struct box *next;
} BOX;
```

Each BOX structure describes a cube. The first element, *oct*, points to the black octree node; *origin* contains the coordinates of the corner farthest from the viewer (i.e., the hidden corner); *len* is the side length of the cube; *corners* is an array of the screen coordinates of the vertices of the hexagonal projection of the cube; *xhigh*, *yhigh*, *xlow*, *ylow*, *xleft*, *yleft*, *xright*, *yright* are the screen coordinates of the highest, lowest, leftmost, and rightmost vertices, respectively, in *corners*; *edges* is an array of pointers to EDGE structures representing the nine potentially visible edges of the projected cube; finally, *next* is a pointer to the next element in the linked list.

Edges of projected cubes are represented by linked lists and are defined in C as

```
typedef struct edge {
  int min, max;
  float xmin, ymin, xmax, ymax;
  struct edge *next;
} EDGE;
```

[1] UNIX is a trademark of AT&T Bell Laboratories.

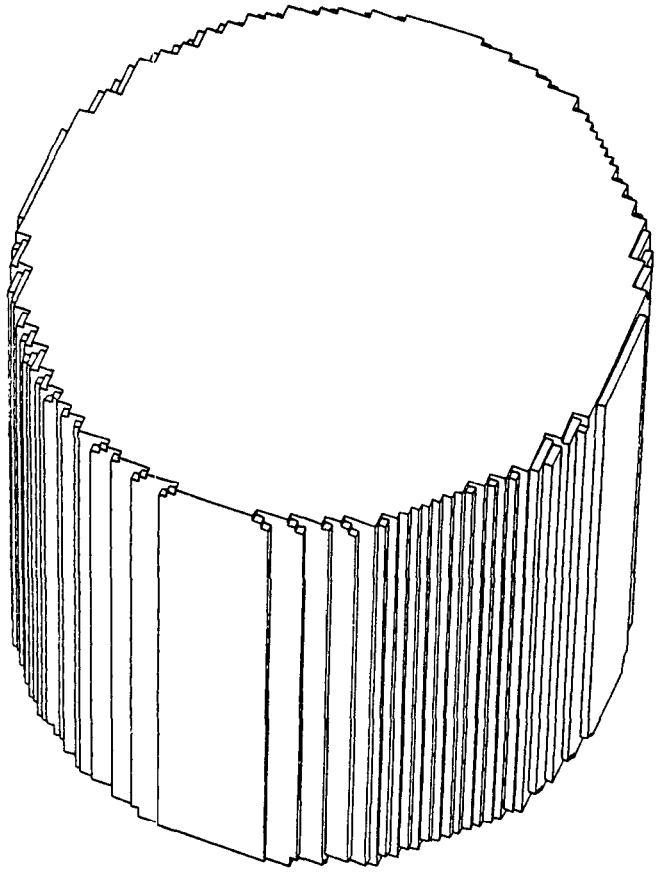

Fig. 5. The line drawing for the octree representation of a cylinder.

The first two elements of the EDGE structure, *min* and *max*, store the beginning and ending positions of a segment of the edge. The values of *min* and *max* represent the distance from the beginning of the edge in terms of the side length of the smallest octree node. The next four elements, *xmin*, *ymin*, *xmax*, *ymax*, are the screen coordinates of the points represented by *min* and *max*. Finally, *next* points to the next edge segment.

3. PERFORMANCE OF THE LINE DRAWING ALGORITHM

In this section we present examples of line drawings produced by our algorithm for a number of octrees and some comments on the computational complexity of the algorithm.

To evaluate the performance of our display algorithm, we supplied to it the octrees produced by our octree generation algorithm [2, 16]. The octrees were generated for a known set of objects. It was easy to check the correctness of the line drawing algorithm since the octree generation algorithm followed by the line drawing generation algorithm should provide a display of the original object as represented by the octree.

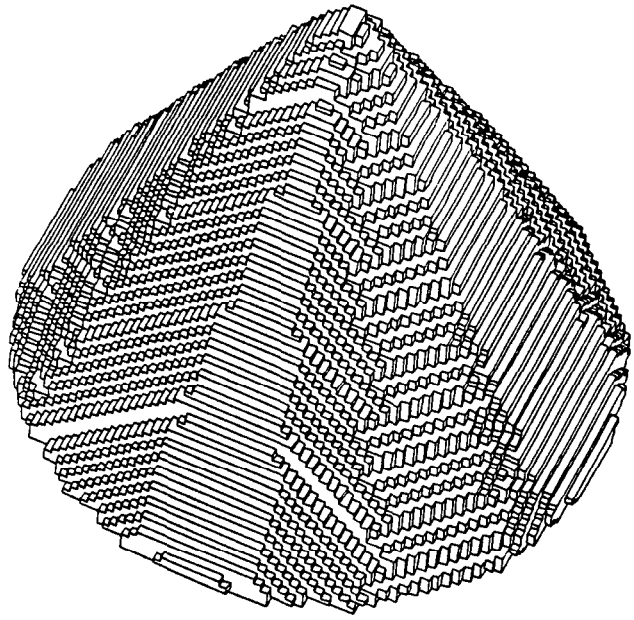

Fig. 6. The line drawing for the octree representation of a cone.

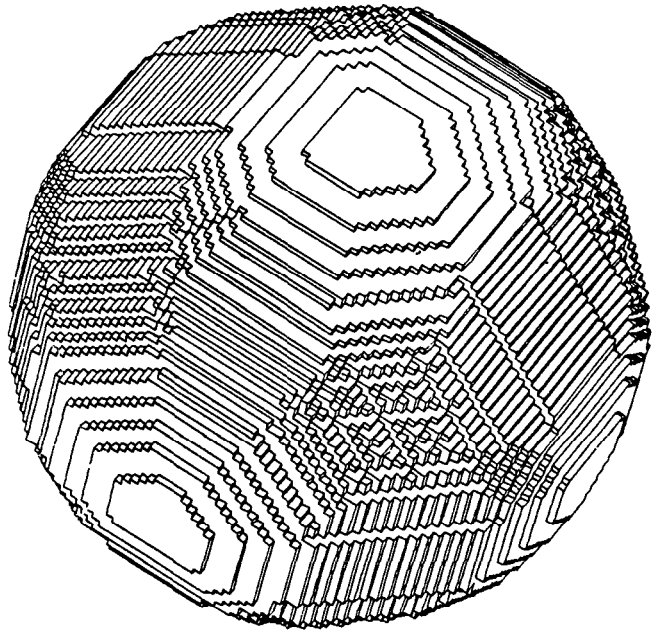

Fig. 7. The line drawing for the octree representation of a sphere.

3.1 Example Line Drawings

The line drawing algorithm was executed on a VAX and the output sent to a QMS laser printer. Figures 5–8 show the line drawings generated for the octrees of some simple geometric objects. Figure 9 is a line drawing generated from an

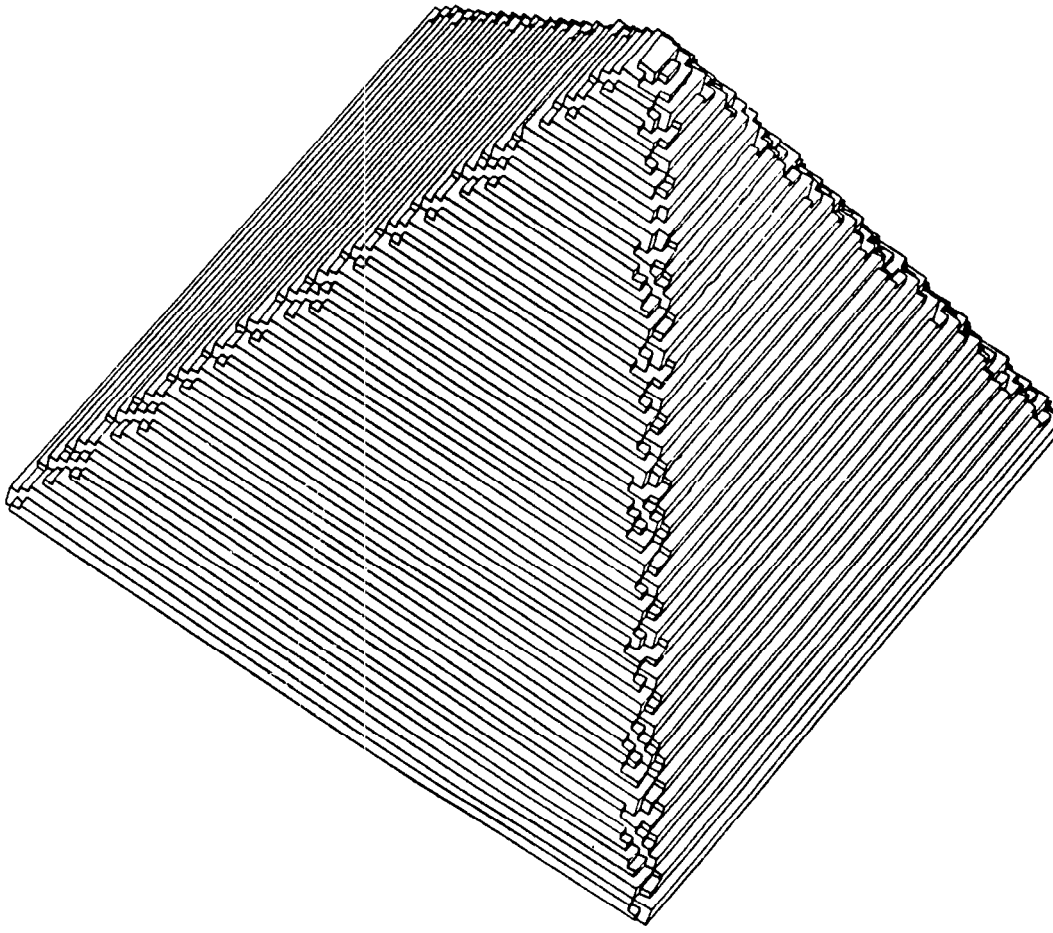

Fig. 8. The line drawing for the octree representation of a pyramid.

octree that was obtained from gray-level silhouette images of a coffee cup taken from three orthogonal viewing directions. Figure 10 shows the drawing for a more complex object, which exhibits self-occlusion. Figure 11 shows an object whose silhouette is a diamond when viewed from any face of the universe cube that contains it. The blocky structure of the surfaces actually represented by the constructed octree can be seen in all figures. For example, there are small steps leading down from the top face of the cylinder in Figure 5. Some blockiness will always be present in the surfaces represented by octrees because of their finite resolution. Our algorithm results in additional blockiness for the following reasons. First, the intersections performed over successive cylinders give only a polyhedral approximation to a smooth surface. Second, the silhouettes of the objects used in our experiments were stored as digital regions, with jagged boundaries, resulting in a staircase approximation of continuous boundaries, as can be seen in Figure 5.

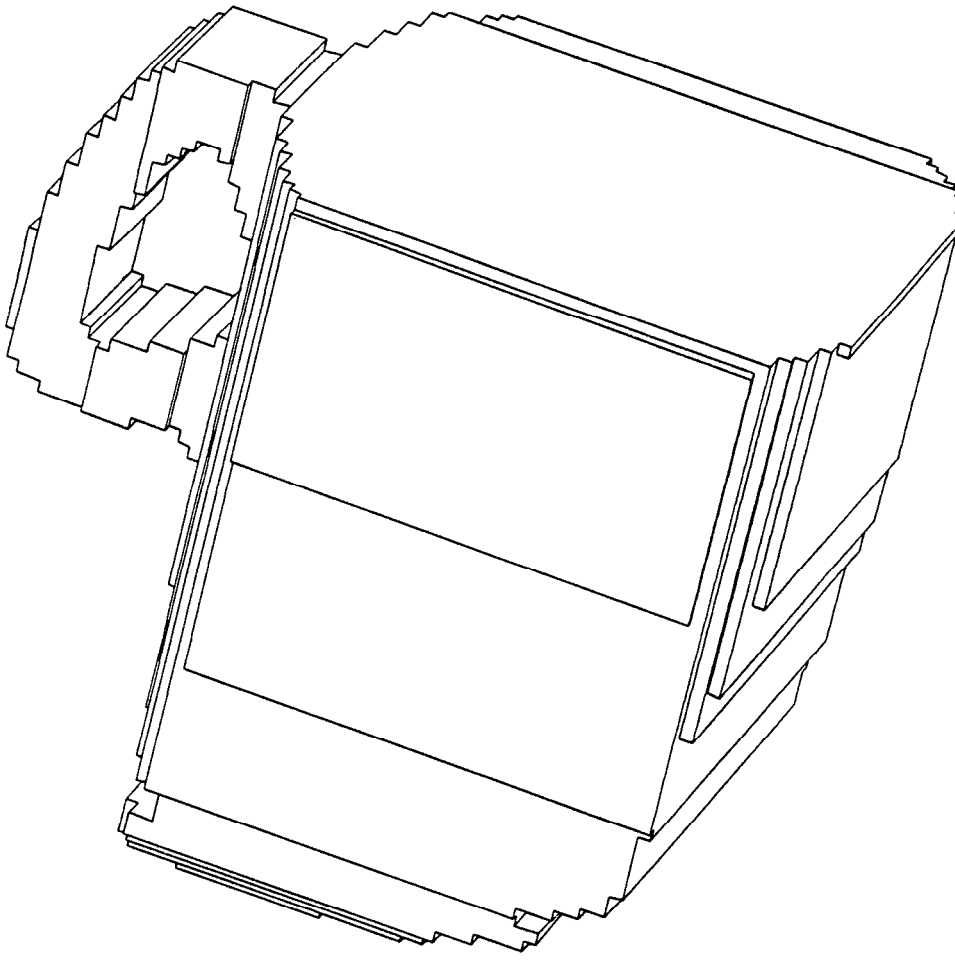

Fig. 9. A line drawing of a coffee cup.

3.2 Complexity

Figure 12 shows the execution time of the line drawing algorithm as a function of the number of nodes in the octree. The graph shows that there is no simple relationship between the two. The octree is traversed once to construct the linked list of graphics nodes, but each graphics node is revisited for every graphics node that occurs later in the linked list, thus resulting in a number of visits that are proportional to the square of the number of graphics nodes. However, the processing time required at each graphics node varies greatly. Extra processing is required at a node if its screen projection overlaps the screen projections of other nodes. Processing time is further increased in the cases in which edge intersections between two overlapping projections must be computed. Since the computation of edge intersections dominates the processing time at a node, the overall processing time is largely determined by the number of edge intersections

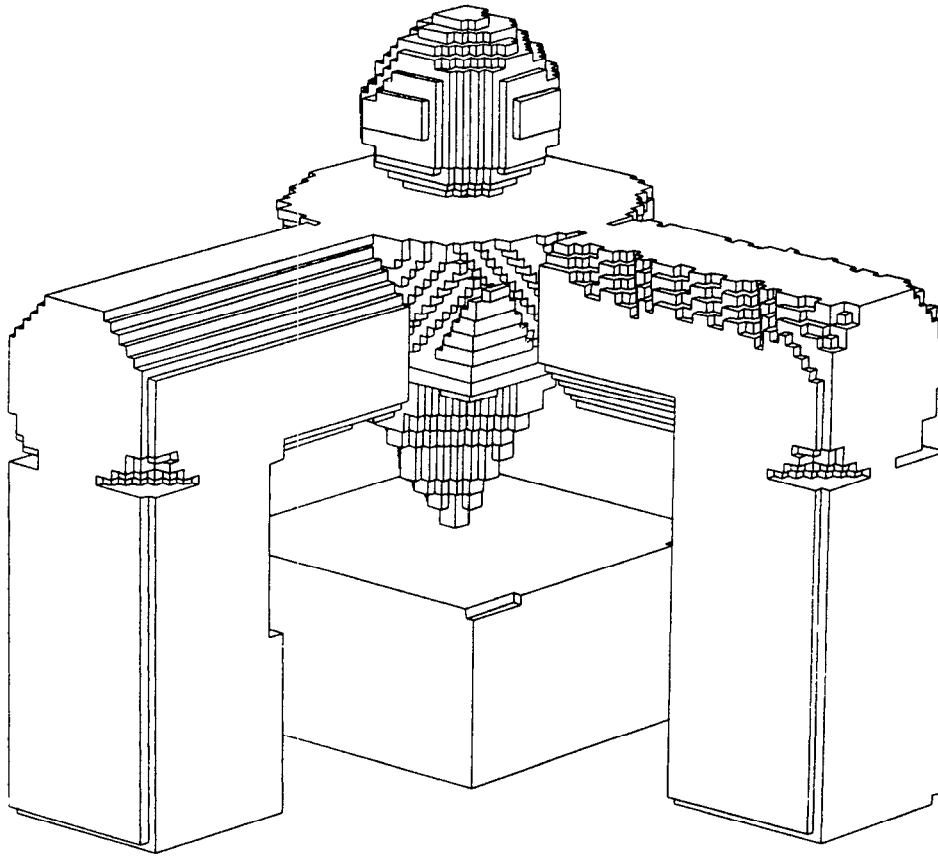

Fig. 10. A complex object that exhibits self-occlusion.

in the image rather than by the number of graphics nodes visited. Therefore, the execution time depends more on the complexity of the image (i.e., how many hidden lines must be removed) than on the number of octree nodes.

The memory requirements for the graphics information (in addition to the memory used by the octree) consist of at most one graphics node for each black octree node. Each graphics node contains pointers to linked lists (representing visible edges), which may contain zero or more elements (depending on the degree of fragmentation of an edge). If the image has a large number of partially occluded octants, then some graphics nodes may require long linked lists to represent the visible edge fragments. In addition, if the object is oriented poorly with respect to the decomposition space of the universe cube, the octree itself may be very large. Thus the space complexity is dependent on both the complexity of the image and the efficiency of the octree decomposition.

Gargantini [7] has proposed a data structure for storing octrees that does not use pointers but instead encodes the octree as a linear list of the black octants, where each black octant is uniquely determined by its "path" from the root. This

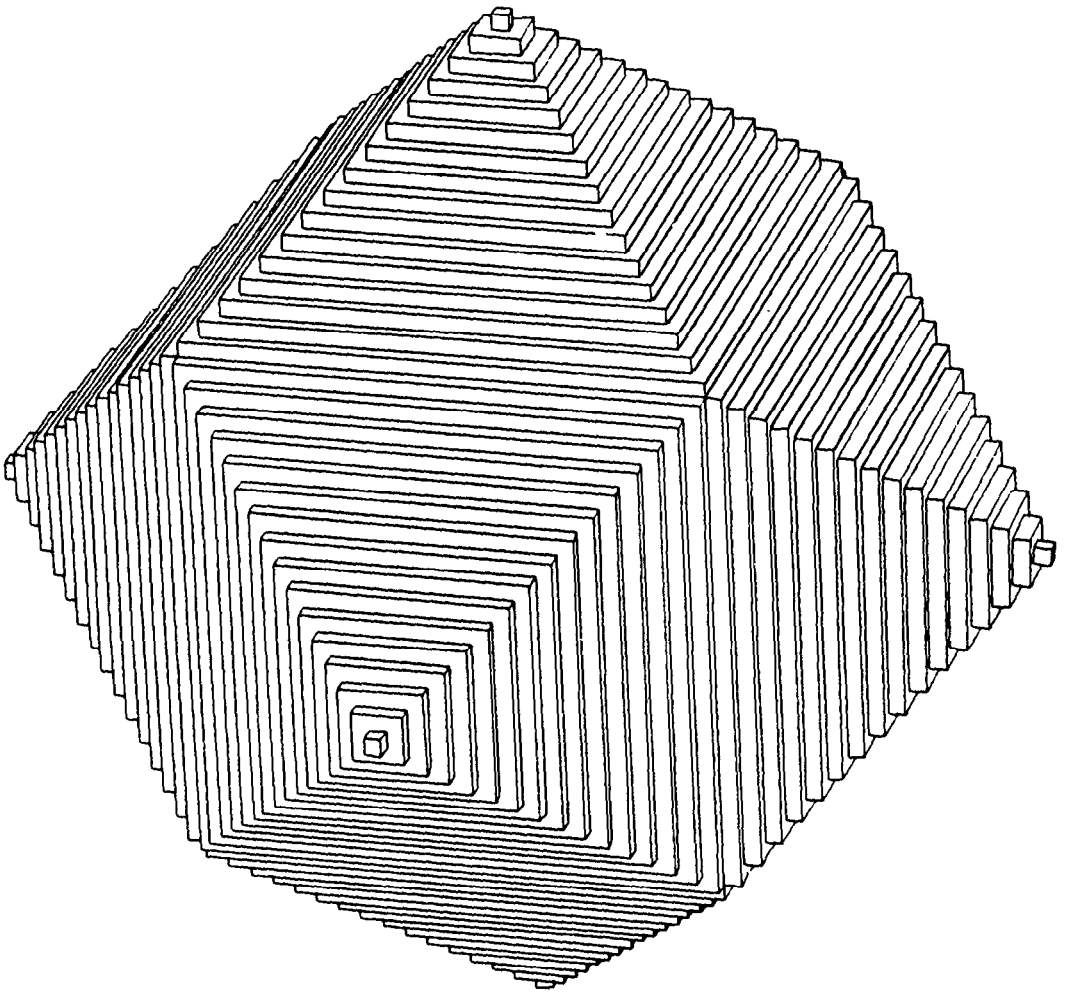

Fig. 11. A line drawing of a diamond-shaped object.

method requires less memory for storing a given octree than does a data structure that uses child pointers. Our line drawing algorithm, however, frequently needs to examine neighboring octants of a given node, which is efficiently accomplished by following pointers in a threaded octree.

4. SUMMARY

We have presented a display algorithm to produce a line drawing of an object from its octree. The object is drawn using parallel projection with cracks and hidden lines removed for any specified viewpoint. Line drawings are a useful tool in visualizing objects represented by octrees that are hard to interpret as geometric objects by direct inspection.

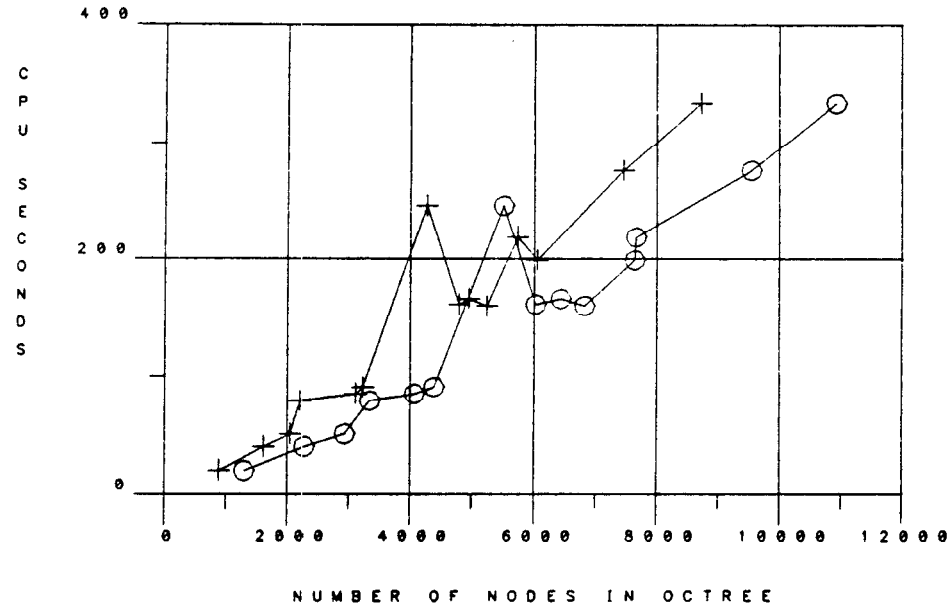

Fig. 12. Graph of line drawing generation time as a function of the number of nodes in the octree: "o" represents total nodes; "+" represents black (leaf) nodes.

ACKNOWLEDGMENTS

Thanks are due to anonymous reviewers who made very helpful comments on the manuscript.

REFERENCES

1. AHUJA, N., AND NASH, C. Octree representations of moving objects. *Comput. Vision Graph. Image Process. 26* (1984), 207–216.
2. AHUJA, N., AND VEENSTRA, J. Octree generation and display. Tech. Rep. UILU-ENG-86-2215. Coordinated Science Laboratory, Univ. of Illinois, Urbana, May 1986.
3. BROOKS, R. Symbolic reasoning among models and 2-D images. *Artif. Intell. 17* (1981), 285–348.
4. CHIEN, C. H., AND AGGARWAL, J. K. Volume/surface octrees for the representation of 3-D objects. *Comput. Vision Graph. Image Process. 36* (1986), 100–113.
5. DOCTOR, L. J., AND TORBORG, J. G. Display techniques for octree-encoded objects. *IEEE Comput. Graph. Appl. 1*, 3 (1981), 29–38.
6. FOLEY, J., AND VAN DAM, A. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, Mass., 1983.
7. GARGANTINI, I. Linear octrees for fast processing of three-dimensional objects. *Comput. Graph. Image Process. 20*, 4 (Dec. 1982), 365.
8. HONG, T. H., AND SHNEIER, M. Describing a robot's workspace using a sequence of views from a moving camera. *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-7* (Nov. 1985), 721–726.
9. JACKINS, C. L., AND TANIMOTO, S. L. Oct-trees and their use in representing three-dimensional objects. *Comput. Graph. Image Process. 14*, 3 (Nov. 1980), 249–270.
10. KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N. J., 1978.
11. MEAGHER, D. Efficient synthetic image generation of arbitrary 3-D objects. In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing* (Las Vegas, Nev., June 14–17, 1982), p. 473.

12. MEAGHER, D. Geometric modeling using octree encoding. *Comput. Graph. Image Process. 19*, 2 (1982), 129.
13. NEWMAN, W., AND SPROULL, R. *Principles of Interactive Computer Graphics.* McGraw-Hill, New York, 1979.
14. OSSE, W., AND AHUJA, N. Efficient octree representation of moving objects. In *Proceedings of the IAPR 7th International Conference on Pattern Recognition* (Montreal, Quebec, July 30–Aug. 2, 1984), pp. 821–823.
15. SHNEIER, M., KENT, E., AND MANSBACH, P. Representing workspace and model knowledge for a robot with mobile sensors. In *Proceedings of the IAPR 7th International Conference on Pattern Recognition* (Montreal, Quebec, July 30–Aug. 2, 1984), pp. 199–202.
16. VEENSTRA, J., AND AHUJA, N. Octree generation of an object from silhouette views. In *Proceedings of the 1985 IEEE International Conference on Robotics and Automation* (St. Louis, Mo., Mar. 25–28, 1985), pp. 843–848.

Received June 1986; revised May 1987; final revision accepted August 1987