May 1986 UILU-ENG-86-2215

COORDINATED SCIENCE LABORATORY
College of Engineering

OCTREE GENERATION AND DISPLAY

Narendra Ahuja
Jack Veenstra

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-86-2215	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS (City, State and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State and ZIP Code) 1800 G Street Washington, D.C. 20550
8a. NAME OF FUNDING/SPONSORING ORGANIZATION National Science Foundation 8b. OFFICE SYMBOL (If applicable) N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ECS 83-52408
8c. ADDRESS (City, State and ZIP Code) 1800 G Street Washington, D.C. 20550	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: N/A PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) "Octree Generation and Display"	
12. PERSONAL AUTHOR(S) Narendra Ahuja and Jack Veenstra	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) May 1986 15. PAGE COUNT 73
16. SUPPLEMENTARY NOTATION N/A	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) three-dimensional object representation, octree, orthographic projection, silhouette, octree generation algorithm

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report addresses the following problem: given a sequence of silhouette views of an object, construct the octree representing the object which gave rise to those views. A given silhouette constrains the object to lie in a cone (for perspective projection) or a cylinder (for orthographic projection) whose cross section is defined by the shape of the silhouette. In this report, we will consider orthographic projection of an object into a plane perpendicular to a viewing direction. We will call "extended silhouette" the solid region of space defined by sweeping the silhouette along a line parallel to the viewing direction used in obtaining the silhouette. The object is constrained to lie in the intersection of all extended silhouettes. As the number of silhouettes processed increases, the fit of the volume of intersection of the cylinders to the object volume becomes tighter. In our algorithm, we do not perform the intersection explicitly, but infer the octree nodes from silhouette images according to a predetermined table that pairs image regions with their corresponding octree nodes.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ OTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code) 22c. OFFICE SYMBOL NONE

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

TABLE OF CONTENTS

1. INTRODUCTION

Three-dimensional object representation is of crucial importance to robot vision. Part of the task lies in the generation and maintenance of a spatial occupancy map of the environment. The occupancy map describes the space occupied by objects. Some of the uses of such a representation include robot navigation and manipulation of objects on an assembly line. This report is concerned with the construction of one such representation, namely, the octree representation, of an object from its silhouette images.

An octree [1,6,11] is a tree data structure. 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 (see Figure 1). 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 criteria.

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 is assigned the label of the octant. Figure 2(b) shows the octree for the object in Figure 2(a). The children nodes are arranged in increasing order of label values from left to right. The black nodes are shown as

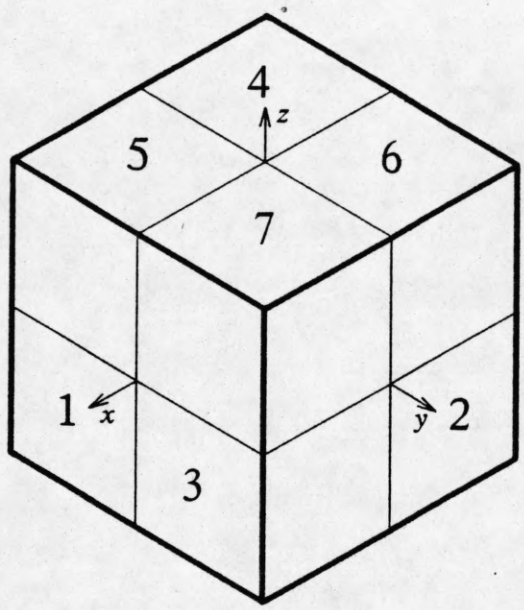


Figure 1

A cube and its decomposition into octants.

---

dark circles and the white and gray nodes are shown as empty circles. In practice, of course, the white nodes need not be stored.

This report addresses the following problem: given a sequence of silhouette views of an object, construct the octree representing the object which gave rise to those views. A given silhouette constrains the object to lie in a cone (for perspective projection) or a cylinder (for orthographic projection) whose cross section is defined by the shape of the silhouette. In this report, we will consider orthographic projection of an object onto a plane perpendicular to a viewing direction. We will call "extended silhouette" the solid region of space defined by sweeping the silhouette along a line parallel to the viewing direction used in obtaining the silhouette. The object is constrained to lie in the intersection of all extended silhouettes. As the number of silhouettes processed increases, the fit of the volume of intersection of the cylinders to the object volume becomes tighter. In our algorithm, we do not perform the intersection explicitly, but infer the octree nodes from silhouette images according to a predetermined table

that pairs image regions with their corresponding octree nodes.

An alternate approach, due to Shneier et al. [5, 12], to the problem of constructing the octree from silhouettes explicitly tests for the intersection between an octree node and the extended silhouette by projecting the nodes of the tree onto the silhouette image. The relative performance of algorithms based on their approach and ours needs to be determined. Chien and Aggarwal [3] describe an efficient method for constructing an octree for an object from silhouettes of its three orthogonal views. Their method is similar to the method described here, though it provides much coarser results since they only use three axial views. The accuracy of the octree describing the object is improved if, in addition to the three orthogonal views, information from other views of the object is also used. Of course, the challenge lies in containing

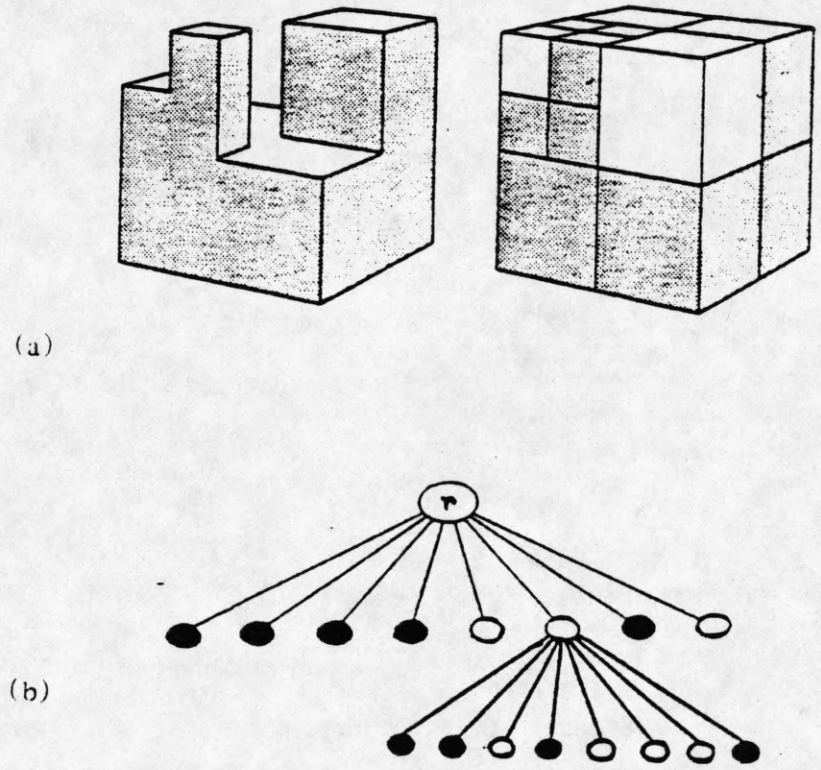


Figure 2

An object (a) and its octree representation (b).

the amount of computation while improving accuracy.

During the development of our image-to-octree algorithm, we felt the need for a octree-to-image display algorithm to visually monitor the accuracy of the octree as it evolves by assimilating object information present in successive silhouette views. We developed an algorithm for this octree-to-object transformation, which is the reverse transformation of constructing the octree from the object discussed in the first part of the report. The octree-to-image algorithm displays an object represented by a given octree as a line drawing in perspective with hidden lines removed. The line drawing can be displayed corresponding to any arbitrary viewpoint. We use this algorithm to display line drawings of the octrees derived by the octree generation algorithm. A visual comparison of the original objects with those depicted by the line drawing algorithm then serves as a useful test of the correctness of the octree generation algorithm.

We also present results on more precise evaluation of the performance of our octree generation algorithm. The performance measure used is the ratio of the true volume of the object to the volume represented by the octree generated. The variation of the accuracy of representation with changes in object orientation, object complexity, and allowed computation time are studied.

The algorithm described in this report is part of a three-dimensional representation, manipulation, and navigation system that we are developing. The common theme through the various components of the system is the use of octree representation. The problem addressed in this paper is that of initial acquisition of the occupancy information and construction of octree representation. Elsewhere, we have addressed the problem of updating the octree in response to motion of objects in the scene. The work on using the octrees for path planning and manipulation tasks is in progress.

In Section 2 we describe our octree generation algorithm. Section 3 discusses the performance of the algorithm. Section 4 describes the line drawing display algorithm and presents experimental results as line drawings of the objects represented by the generated octrees. Section 5 presents concluding remarks.

2. OCTREE GENERATION ALGORITHM

The algorithm described in this section constructs the octree without computing any projections or performing any intersection tests. The viewing directions are defined with reference to the universe cube and are restricted to be those providing one of the six "face" views, one of the twelve "edge" views, or one of the eight "corner" views of the universe cube. Although this allows only thirteen useful views (since views from any two opposite directions provide the same silhouette), the viewpoints are distributed widely in space and together provide significant information to construct a good approximation of the object.

Restricting the viewpoints in this manner allows us to find correspondences between the pixels in the two-dimensional silhouette image and the octants in the three-dimensional space that define the octree. The relationship between pixels and octants for an orthographic face view is easily derived so it is described first. Then the relationships between pixels and octants for orthographic edge and corner views are presented. Similar relationships would be difficult to obtain for an arbitrary viewpoint.

2.1. Face View

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. (See Figures 1,

3(a).) 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 3(a). 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, as in Figure 3(a). 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. Figure 3(b) shows the nodes assigned to the sub-quadrants.

A similar procedure is used for the other two face views, the only difference being in the labeling scheme for the image quadrants. For example, the labels for the upper left quadrant for the face y view are 7, 5, and for the face z view are 4, 0. (See Figure 4).

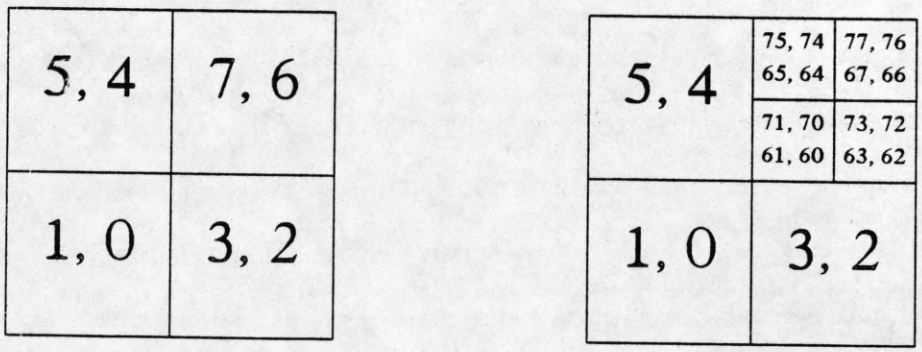


Figure 3

The labeling scheme for quadrants for the face x view. Each quadrant is assigned two labels (a) instead of one. Each time a quadrant is sub-divided, the sub-quadrants have twice as many labels (b).

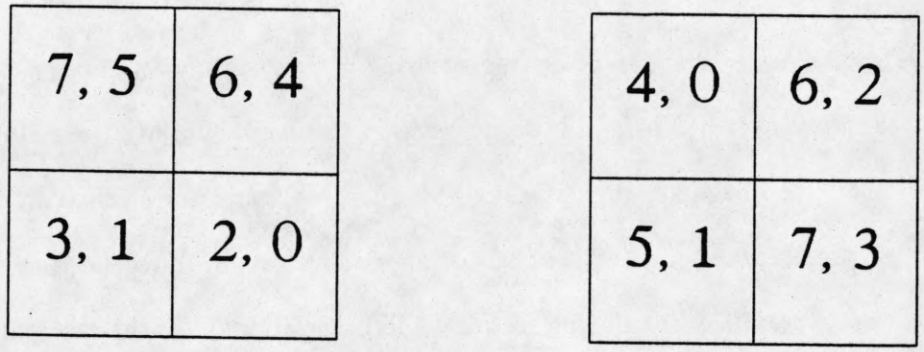


Figure 4

The labeling scheme for quadrants for the face y view (a), and face z view (b).

2.2. Edge View

An "edge view" of a cube is the view obtained when the line of sight bisects an edge of the universe cube and passes through the center of the cube. An edge view is labeled with the two adjacent octants of the universe cube each of which contains one half of the bisected edge. The octants of the cube in Figure 1 viewed from edge 3-7 would appear as shown in Figure 5. The silhouette of a cube in an edge view is longer in the horizontal direction by a factor of $\sqrt{2}$. Since the octree generation algorithm requires a square array, the elongated image from an edge view must be compressed into a square array. This is accomplished by resampling the digitized image with smaller sampling density along the horizontal direction.

The recursive procedure for constructing an octree from a square array of pixels representing an edge view is similar to procedures for constructing a quadtree. If the square array is all ones or all zeroes, then it is marked black or white, respectively. Otherwise it contains some ones and some zeroes and it is decomposed recursively in two different ways.

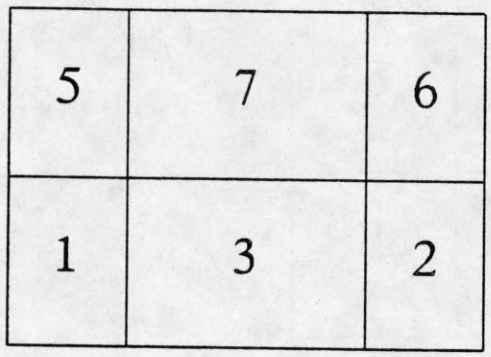


Figure 5

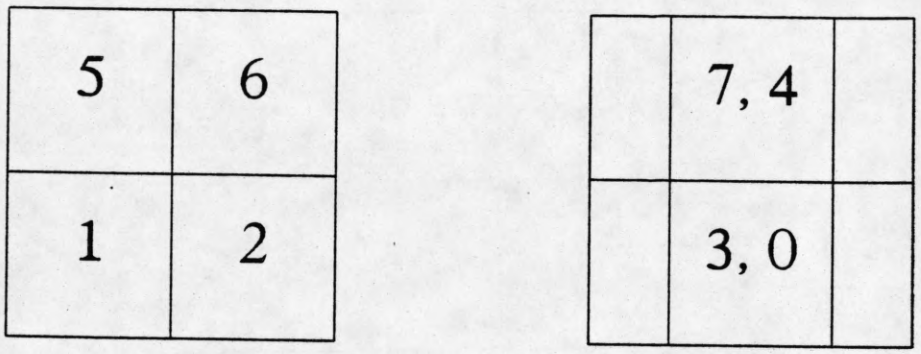
The cube in Figure 1 viewed from edge 3-7.

---

(1) It is decomposed into the usual four quadrants, each with one label. The labels depend on which edge is being viewed. For example, the labels for the four quadrants for the edge 3-7 view are given in Figure 6(a). If a quadrant contains both zeroes and ones then it is recursively decomposed.

(2) It is decomposed into two center squares and two margins (see Figure 6(b)). The center squares are the same size as the quadrants in the first decomposition step. The margins are half the width of the squares and are not used. Each center square has two labels. These are treated in a manner similar to the way the quadrants with two labels for the face view were treated. Whenever a node with one of the two labels is added to the octree, another node with the other label is also added. If a center square contains both zeroes and ones, then it is recursively decomposed.

Each time a quadrant or a center square is decomposed, both methods of decomposition described above are used, unless it is a 2×2 square in which case only the first method is used. At each recursive decomposition step the dimension of the quadrants examined is half that at the previous step. When the dimension of a quadrant is 2, it can only be decomposed according to the first method above (otherwise an image pixel would have to be split in half). To prevent



(a) (b)

Figure 6

The decomposition of the image array for the edge 3-7 view into 4 quadrants (a) and 2 center squares (b).

---

the introduction of nonexistent "holes" in the octree during this process, a center square of a 2×2 array is marked black if either one of the two quadrants intersecting with it is marked black. For example, if a silhouette image taken from edge 3-7 is decomposed down to a 2×2 square and the upper right quadrant (whose label is 6) is black, then, in addition to octant 6, octants 7 and 4, corresponding to the upper central square, will also be marked black.

Figure 7(a) shows the center squares decomposed into four quadrants (each quadrant inherits two labels from the parent square) and Figure 7(b) shows quadrant 6 decomposed into two center squares.

2.3. Corner View

A "corner view" is the view obtained when the line of sight intersects a corner of the universe cube and passes through the center of the cube. Each corner view is labeled according to the corner octant which is closest to the viewer. The silhouette of a cube viewed from one of its corners is a regular hexagon (see Figure 8(a)). Because the geometry of the corner view silhouette does not correspond naturally with the rectangular image plane, processing a corner

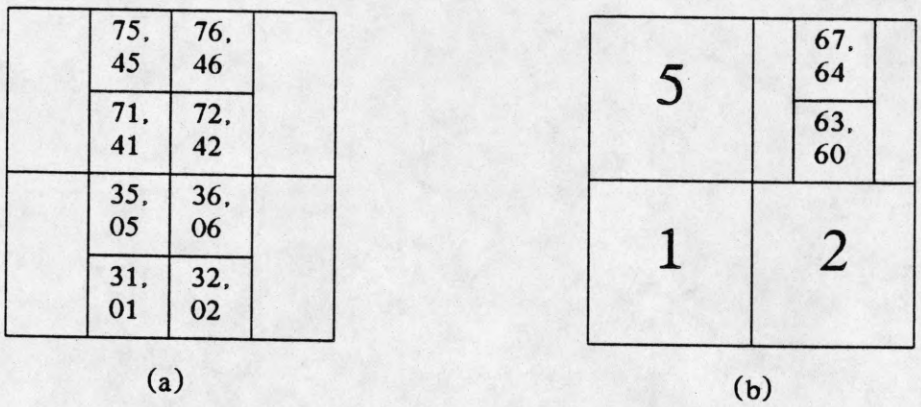


Figure 7

Further decomposition of center squares into quadrants (a), and of the upper right quadrant into center squares (b).

---

view is somewhat more complicated than processing face or edge views where the silhouette of the universe cube is a rectangle.

The silhouettes of the different octants of a cube are all regular hexagons whose regions of intersection are composed of equilateral triangles (see Figure 9). The silhouette of any octant is a union of a subset of these triangles. Thus, the occupancy of an octant can be inferred from the occupancy of an appropriate set of triangles. Occupancy of the universe cube can be inferred from the occupancy of the six major triangular cells (Figure 8(a)). To generate nodes at lower levels in the octree, the triangles of interest are the result of recursive triangular decomposition (see Figures 8(b) and 8(c)) of the six major triangles. Therefore, the processing of a corner view is done in two phases. First, six quadtrees are generated from the digitized image in such a way that each quadtree represents one of the six triangular sections of the regular hexagonal silhouette of the universe cube. Second, the octree is constructed from the six quadtrees. The quadtrees and octree are constructed recursively.

To construct the quadtrees, the universe hexagon is divided into six triangles labeled 0 to 5 as shown in Figure 8(a). Each triangle is recursively subdivided and its nodes are labeled 0 to

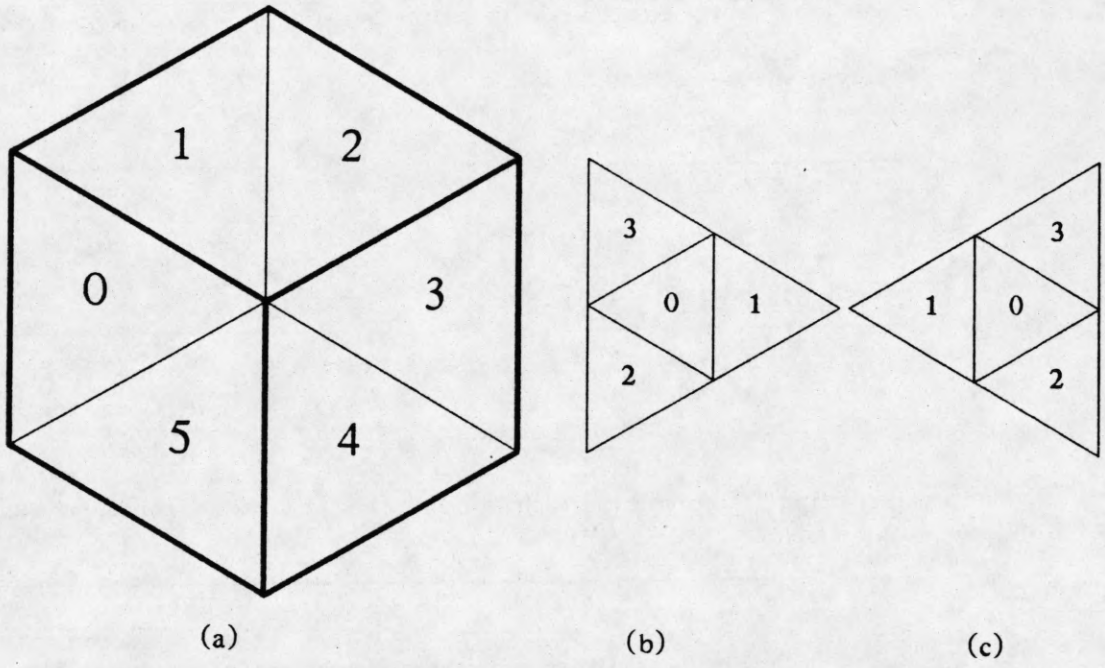


Figure 8

The division of the universe hexagon into six triangular sections (a). Each section is represented by a triangular quadtree. The labels of the quadtree nodes for each of the two orientations of a triangle (b and c).

---

3 according to one of the two schemes shown in Figure 8(b) and 8(c). The orientation of the triangle determines which labeling scheme is used. In both labeling schemes, quadtree node 0 is in the center and node 2 is in the lower corner. The recursive subdivision of a triangle into four sub-triangles continues until the distance between the centers of adjacent triangles is less than or equal to the distance between pixels in the digitized image. The image pixel nearest the center of a triangle at the lowest level in the quadtree determines whether the corresponding quadtree node is black or white. The color of a quadtree node above the lowest level is determined by the colors of its children. If the children of a quadtree node are all white or all black, then the parent node is assigned the same color as its children, and the children are removed from the tree. Otherwise, some children are white and some are black so the parent is assigned the color gray. Each quadtree is given the label (0 - 5) of the major triangle it

represents. Once the six quadtrees are constructed, the raw silhouette image data is no longer needed. The octree is generated directly from the quadtrees; quadtrees are used only as an intermediate step in the octree construction.

The octree construction is best explained by showing how the color of a particular octant is determined. Figure 9 shows the projection of octant 5 in relation to the projection of the universe cube viewed from octant 7. Octant 5 also projects as a hexagon and overlaps triangular quadtrees 0 and 1. Octant 5 is labeled white if the nodes of quadtrees 0 and 1 covered by the projection of octant 5 are all white. In this case the relevant nodes that need to be examined are: nodes 0,1,3 of quadtree 0 immediately below the root level, and nodes 0,1,2 of quadtree 1 immediately below the root level. If these nodes are not all white nor all black and the predetermined maximum depth for the octree has not been exceeded, then octant 5 is recursively subdivided and the appropriate children of these nodes in the quadtree at the next lower

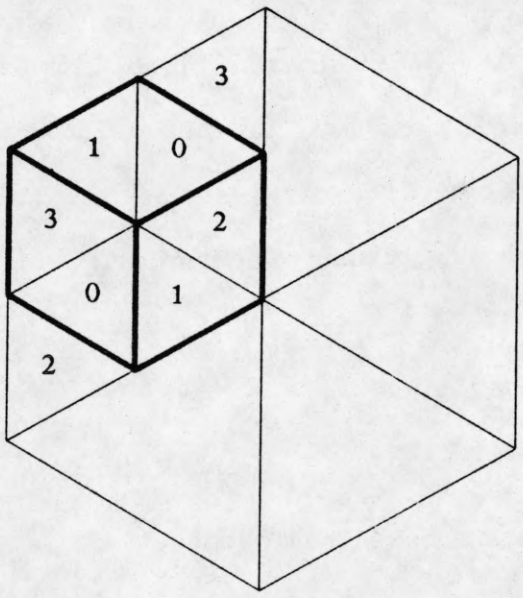


Figure 9

The projection of octant 5 (shown in bold lines) which overlaps triangular quadtrees 0 and 1.

level are examined. This process continues until all nodes are labeled black or white. The same procedure is used for the other octants, the only difference being in the set of quadtree children which needs to be examined.

2.4. Algorithm Details

This section describes in more detail the implementation of the octree generation algorithms. All of the algorithms described in this report were implemented in the C programming language [7]. Example C procedures which implement the face, edge, and corner views are given below along with an explanation for each one. The example routines are actual working C procedures although they have been simplified for clarity. The following C structure was used by all the routines to represent an octree:

```
typedef struct octree {
        char black;
        struct octree *child[8];
} OCTREE;
```

The first element of the OCTREE structure, "black", has value 1 if the corresponding octree node is a black leaf node; else "black" has value 0. The second element in the structure is an array of eight pointers to the subtrees representing the children of this node. White leaf nodes are not stored and their corresponding pointers are null (zero).

2.4.1. Face View

A simple C procedure, *face_x*, to generate the octree for the face x view is given in Figure 10. The digitized silhouette image is stored in the global array *image*.

The three parameters to the procedure are *dim*, *row*, and *col*. The variables *row* and *col* give the location of the upper leftmost pixel in the quadrant currently being examined, and *dim* gives the side length of the quadrant. Initially, *row* and *col* are zero and *dim* is the dimension of the square silhouette image which must be an integer power of two. This procedure

recursively calls itself, dividing *dim* in half at each recursive step, until *dim* = 2. The four pixels in the final 2 × 2 square correspond to the four quadrants of the image; for each pixel that has value 1, the corresponding two octree nodes are made black.

```
extern char image[512][512];

/*
 * face_x() returns a pointer to an OCTREE structure representing the occupied
 * space of the extended silhouette of a "face x" view of an object.
 */

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

   root = newtree();              /* newtree() returns a pointer to an octree */
   dim = dim/2;
   if (dim > 1) {                 /* recursively subdivide image quadrants */

        root->child[4] = face_x(dim, row, col);          /* upper left quadrant */
        root->child[5] = copy(root->child[4]);
        root->child[6] = face_x(dim, row, col+dim);      /* upper right quadrant */
        root->child[7] = copy(root->child[6]);
        root->child[0] = face_x(dim, row+dim, col);      /* lower left quadrant */
        root->child[1] = copy(root->child[0]);
        root->child[2] = face_x(dim, row+dim, col+dim);  /* lower right quadrant */
        root->child[3] = copy(root->child[2]);

   } else {                       /* examine image pixels in 2x2 square array */

        if (image[row][col]) {                           /* if pixel value is 1, */
           root->child[4] = new_black_child();           /*  then make children black */
           root->child[5] = new_black_child();
        }
        if (image[row][col+1]) {                         /* upper right pixel */
           root->child[6] = new_black_child();
           root->child[7] = new_black_child();
        }
        if (image[row+1][col]) {                         /* lower left pixel */
           root->child[0] = new_black_child();
           root->child[1] = new_black_child();
        }
        if (image[row+1][col+1]) {                       /* lower right pixel */
           root->child[2] = new_black_child();
           root->child[3] = new_black_child();
        }
   }
   return(compact(root));                                /* return compacted octree */
}
```

Figure 10

A C procedure to generate an octree for the face x view.

Procedures for the face y and face z views can be obtained by appropriately changing the integer labels of the octree children, or a general procedure can be defined by using variables as the labels of the octree children.

2.4.2. Edge View

A simple C procedure, *edge_37*, to generate the octree for the edge 3-7 view is given in Figure 11. As with the face x procedure, the digitized silhouette image is assumed to be stored in the global array *image*. The parameters supplied to the procedure are the same as supplied to *face_x*. The variable *dim* is the dimension of the current quadrant being examined and *row* and *col* are used to index the pixels in the image.

The image is recursively subdivided using both methods described in Section 2.2 until the quadrant is a 2×2 array. If either the upper left or the upper right pixel in this array has value 1, then the corresponding child 5 or 6, respectively, is added to the octree as a black leaf node and the variable *black* is set to 1, causing additional octree children 7 and 4 to be added. Similarly, if either the lower left or lower right pixel (corresponding to children 1 and 2, respectively) in the 2×2 square is black, then children 0 and 3 are also added to the octree as black leaf nodes.

By substituting variables for the octant labels, this procedure can be generalized so that it works for all edge views.

```
/*
 * edge_37() returns a pointer to an OCTREE structure representing the occupied
 * space in the extended silhouette of the edge 3-7 view of an object.
 */
OCTREE *edge_37(dim, row, col)
int dim, row, col;
{
   OCTREE *root;
   int black;

   root = newtree();              /* newtree() returns a pointer to an octree */
   dim = dim/2;
   if (dim > 1) {                 /* recursively subdivide image quadrants */
      root->child[5] = edge_37(dim, row, col);              /* upper left quadrant */
      root->child[6] = edge_37(dim, row, col+dim);          /* upper right quadrant */
      root->child[7] = edge_37(dim, row, col+dim/2);        /* upper center squares */
      root->child[4] = copy(root->child[7]);
      root->child[1] = edge_37(dim, row+dim, col);          /* lower left quadrant */
      root->child[2] = edge_37(dim, row+dim, col+dim);      /* lower right quadrant */
      root->child[3] = edge_37(dim, row+dim, col+dim/2);    /* lower center squares */
      root->child[0] = copy(root->child[3]);
   } else {                       /* examine image pixels in 2x2 square array */
      black = 0;
      if (image[row][col]) {              /* if pixel value = 1, then make child black */
         root->child[5] = new_black_child();     black = 1;
      }
      if (image[row][col+1]) {                                /* upper right pixel */
         root->child[6] = new_black_child();     black = 1;
      }
      if (black) {        /* if either child 5 or 6 is black, then make both 7 and 4 black */
         root->child[7] = new_black_child();
         root->child[4] = new_black_child();
      }
      black = 0;
      if (image[row+1][col]) {            /* if pixel value = 1, then make child black */
         root->child[1] = new_black_child();     black = 1;
      }
      if (image[row+1][col+1]) {                              /* lower right pixel */
         root->child[2] = new_black_child();     black = 1;
      }
      if (black) {        /* if either child 1 or 2 is black, then make both 3 and 0 black */
         root->child[3] = new_black_child();
         root->child[0] = new_black_child();
      }
   }
   return(compact(root));                          /* return compacted octree */
}
```

Figure 11

A C procedure to generate an octree for the edge 3-7 view.

2.4.3. Corner View

A simple C procedure, *corner_7*, to generate the octree for the corner 7 view of an object is given in Figure 12. This procedure is called with two parameters: *hex* and *treedepth*. The first parameter, *hex*, is an array of 6 pointers to the triangular quadtrees which together represent the silhouette image data. The quadtrees are represented by a QUADTREE structure which is analogous to the OCTREE structure for octrees defined earlier. The second parameter, *treedepth*, is an integer that controls the number of levels in the octree and prevents the octree from growing beyond a predetermined maximum depth, which is stored in the external variable *Maxdepth*. Without the *treedepth* parameter, this routine could easily get into an infinite loop. The value of *treedepth* is initially zero and is incremented each time the routine calls itself.

The array *octnode* lists the octree children in the order in which they are generated by this routine. The first element ('5' in this case) is the octant whose projection overlaps quadtrees 0 and 1; the second element is the octant which overlaps quadtrees 1 and 2; the third element overlaps quadtrees 2 and 3, and so on. By appropriately permuting the octant numbers in *octnode*, this routine will generate the octree from any other corner view.

The first six elements of *octnode* project close to the border of the hexagon in Figure 8(a) and so overlap only two quadtrees. The last two elements of *octnode* both project onto the center of the hexagon and overlap parts of all six quadtrees. For this reason, the last two elements are handled separately.

The *for* loop generates the first six octants in the list of octree nodes. The routines *allblack*, *notallwhite*, and *nextargs* are not shown; *allblack* returns true if the quadtree children overlapped by the projected octant are all black; *notallwhite* returns true if the quadtree children are not all white; *nextargs* stores the pointers to the six quadtree children in the array *newhex* which is used for the next recursive call.

```
extern int Maxdepth;
static char octnode[8] = { 5,4,6,2,3,1,7,0 };

/*
 * corner_7() returns a pointer to an OCTREE structure representing the occupied
 * space in the extended silhouette of the corner 7 view of an object.
 */

OCTREE *corner_7(hex,treedepth)
QUADTREE *hex[];
int treedepth;
{
   OCTREE *root;
   QUADTREE *newhex[6];
   int node, parent0, parent1;

   if (treedepth >= Maxdepth) {
        root = new_black_child();              /* return black node */
        return(root);
   }
   root = newtree();
   for (node=0; node<=5; node++) {             /* for each perimeter octree node */
        parent0 = node;                        /* find its two overlapping quadtrees */
        parent1 = (node + 1) % 6;
        if (allblack(hex,parent0,parent1))                    /* if both quadtrees all black, */
          root->child[octnode[node]] = new_black_child(); /* then make octree child black */
        else if (notallwhite(hex,parent0,parent1)) {          /* if quadtrees are not both white, */
            nextargs(hex,newhex,parent0,parent1);             /* then subdivide perimeter node */
            root->child[octnode[node]] = corner_7(newhex,treedepth+1);
        }
   }

/* next, check octree nodes which project onto center of hexagon */
   if (midallblack(hex)) {                                    /* if center of hexagon all black, */
        root->child[octnode[6]] = new_black_child();          /* then make center nodes black */
        root->child[octnode[7]] = new_black_child();
   } else if (notmidallwhite(hex)) {                          /* if center is not all white, */
        nextmidargs(hex,newhex);                              /* then subdivide center */
        root->child[octnode[6]] = corner_7(newhex,treedepth+1);
        root->child[octnode[7]] = copy(root->child[octnode[6]]);
   }
   return(compact(root));                                     /* return compacted octree */
}
```

Figure 12

A C procedure to generate an octree for the corner 7 view.

Finally, the two octants which project into the middle of the hexagon are generated. The routines *midallblack*, *notmidallwhite*, and *nextmidargs* are not shown but are analogous to the routines described above.

The only change necessary to make this routine work for another corner view is to permute the octree children labels in the array *octnode*. A general routine can be created by using a doubly subscripted array in place of *octnode*; the first subscript determining the corner being viewed, and the second subscript being the child label.

3. PERFORMANCE OF THE OCTREE GENERATION ALGORITHM

There are several aspects to the performance of any octree generation algorithm. First and foremost is the issue of the accuracy of the object shape captured by the octree generated. Next is the question whether any trade-off is possible between the accuracy of the representation obtained and the complexity of the computation performed to obtain it. Another aspect of the performance of the algorithm concerns the execution time, i.e., the efficiency with which the algorithm implements the necessary computations. In this section, we will discuss the performance of our octree generation algorithm described earlier.

3.1. Defining Accuracy

An algorithm of the kind described in this report that attempts to reconstruct an object shape from its silhouette views suffers from some inherent limitations. First, the algorithm cannot *detect* those three-dimensional features of the object that are lost during the projection process. For example, no surface concavities are registered. Thus, at its very best the algorithm suffers from such *detection errors* occurring during the data acquisition phase, and can provide a representation of only a *bounding* volume of the object. Then, there are the inaccuracies arising from the limitations of the representation itself. In our case, the octree of a given depth will have an associated error due to the 3-D spatial quantization. An increase in the allowed depth (resolution) of the octree will reduce the error, possibly to zero. Finally, there is the usual 2-D digitization error in obtaining the image. This error can also be reduced by using higher resolution images.

Beyond these general sources of inaccuracy are the specific features of our algorithm that may contribute to further approximations. In particular, our algorithm restricts the choice of the viewing directions to a predetermined set of thirteen directions. Since this restricts, in general, the amount of available information about the object, it is desirable to compare the shapes

of the object captured by the octree and the original object. In order to evaluate the performance of any given set of viewing directions, it is first necessary to define a measure that reflects the accuracy with which an object's volume (shape) can be approximated using silhouettes obtained from the given viewing directions. The approximation, of course, depends on the shape and orientation of the object viewed.

One possible measure of accuracy for a set of viewing directions is the ratio of the volume of the smallest object which could give rise to a given set of silhouettes to the volume of intersection of the extended silhouettes. This measure is a fraction since the volume of the intersection of extended silhouettes of an object contains that object. Even if the object is convex, the volume of the object is probably smaller than the volume of intersection. This worst-case definition means that if a given set of silhouettes has an accuracy measure of 90% then the volume of the actual object can be no less than 90% of the computed volume. Some restrictions must be placed on the object shape (like requiring it to be convex) to prevent the smallest object from having an arbitrarily small volume. Even with restrictions, however, the accuracy measure can be very low if only a few views are used. For example, there exist convex objects smaller than a unit cube which have unit squares as silhouettes when viewed along three orthogonal directions. The projection of a tetrahedron oriented so that its four vertices coincide with four vertices of the unit cube is a unit square when viewed along any direction perpendicular to a face of the unit cube. The tetrahedron would be represented as a cube by the algorithm since the intersection of extended silhouettes is a cube. The volume of the tetrahedron, however, is only one-third the volume of the cube. Since a tetrahedron inscribed in a unit cube is the smallest convex object whose three orthogonal silhouettes are unit squares, the accuracy measure for that set of three silhouettes is 33.3%.

The above definition of accuracy may be of only theoretical interest; the difficulty of finding the smallest object for each set of silhouettes makes this definition impractical. An alternate approach is to empirically measure the performance of a chosen set of viewing

directions on a suitably selected set of objects. The measure of accuracy for a given object is the observed ratio of the volume of the object to the volume of the intersection of the extended silhouettes of the object. For example, a sphere would have an accuracy measure equal to the ratio of its volume to the volume of the intersection of circular cylinders containing it, where the axes of the cylinders coincide with the viewing directions. Using this measure of accuracy, three orthogonal views of a sphere would yield an accuracy of 88.9%. The accuracy of nine face and edge views is approximately 98.7%.

Except for the sphere, the accuracy of a set of viewing directions for a given object is dependent on the object's orientation. In one orientation, the tetrahedron yields an accuracy of 33.3% for three orthogonal views; in another orientation, the accuracy is 100%. In fact, only two orthogonal views of the tetrahedron are necessary to represent it exactly. To obtain the average performance over all orientations, a Monte Carlo simulation experiment can be performed to measure the desired ratio of volumes over a large number of randomly chosen orientations. Then, for a given set of objects, the measure of accuracy for a set of viewing directions is the estimated expected value of the ratio of the object volume to the constructed volume for a randomly selected object at a randomly selected orientation.

3.2. Measuring Accuracy

How should the objects constituting the test set be chosen? One way to resolve this question is to use objects having shapes used as primitives for three-dimensional representations, e.g., generalized cones. A generalized cone is defined by a space curve spine and a planar cross section which is swept along the spine according to a sweeping rule. The sweeping rule determines how the cross section changes as it is translated along the spine. Figure 13 shows a sample of generalized cones used by Brooks [2] as primitive volume elements.

The measure of accuracy can then be computed as the average of the results of a large number of executions of the following three step procedure. First, an arbitrary object from the chosen set and a random orientation are selected. Second, the object is projected along each viewing direction to provide a set of silhouette images. Finally, the octree is constructed and the corresponding object volume computed. The ratio of the actual to the computed volume is the desired result for the chosen object and orientation.

In our experiments we used the geometric objects shown in Figure 13 with the following modifications. In place of 13(f) we used a circular cone; in place of 13(g) we used a regular cube; and in place of 13(i) we used a regular pyramid. We further added the sphere and a small cube to the set of objects giving us eleven objects on which to observe the performance of the octree generation algorithm. These objects were viewed at random orientations to determine an average accuracy resulting from the thirteen viewing directions described in this report. Figure 14 lists the objects used along with their mathematical definitions and volumes. The cube is the largest object in the list and all other objects fit inside it. This was done so that

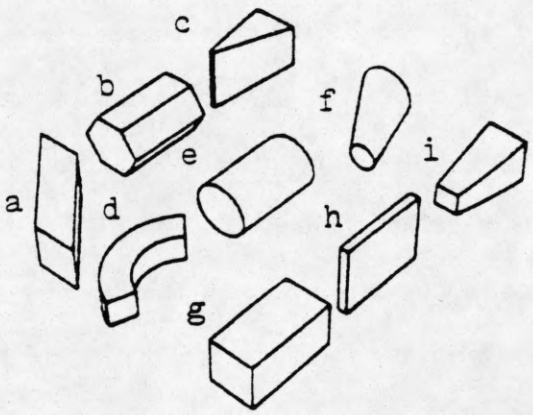


Figure 13

A selection of generalized cones as taken from [2]. A rectangular prism (a), an octagonal prism (b), a wedge (c), an arc (d), a cylinder (e), a partial cone (f), a rectangular solid (g), a slice (h), and a partial pyramid (i).

the silhouettes of all the objects would be guaranteed to fit on the simulated image screen.

For each object in the test set, one hundred random orientations were selected. For each orientation, the thirteen digitized silhouette images in the form of binary-valued square arrays were computed assuming that the object is placed with its center at the origin. The octree was then constructed from the thirteen silhouettes and the ratio of the object volume to the octree volume was computed.

The silhouettes were generated on the computer from mathematical definitions of the test objects and a simulated 128×128 digitized image was created. To determine the value of an image pixel, a line was constructed perpendicular to the image plane and passing through the center of the image pixel. If this line intersected the test object the pixel value was set to 1, else it was set to 0. This process was repeated for each of the 16,384 pixels to generate a digitized silhouette image.

Since the octree constructed from extended silhouettes represents an object larger than or equal to the actual object, the ratio of the object volume to the octree volume should always have a value less than or equal to 1. Due to digitization error, sometimes a pixel on the border of a silhouette is marked empty (set to zero) when it is actually partially covered by the silhouette. This can lead to lost volume in the object represented by the octree since the method of taking intersections of extended silhouettes always yields a smaller (or possibly the same size) octree as each new silhouette is processed. Once an octant is removed from the octree, it is never returned. This problem becomes more pronounced for smaller objects since the removal of a fixed chunk of the object represents a larger portion of its volume. To compensate for such digitization error, we "grow" the silhouette by locating every pixel with value 1 and setting its neighbors (in all eight directions) to 1. This process, in general, overcompensates for the digitization error and reduces the accuracy of the resulting octree, especially for small objects. But it at least guarantees that the octree will not represent an object smaller than the actual object.

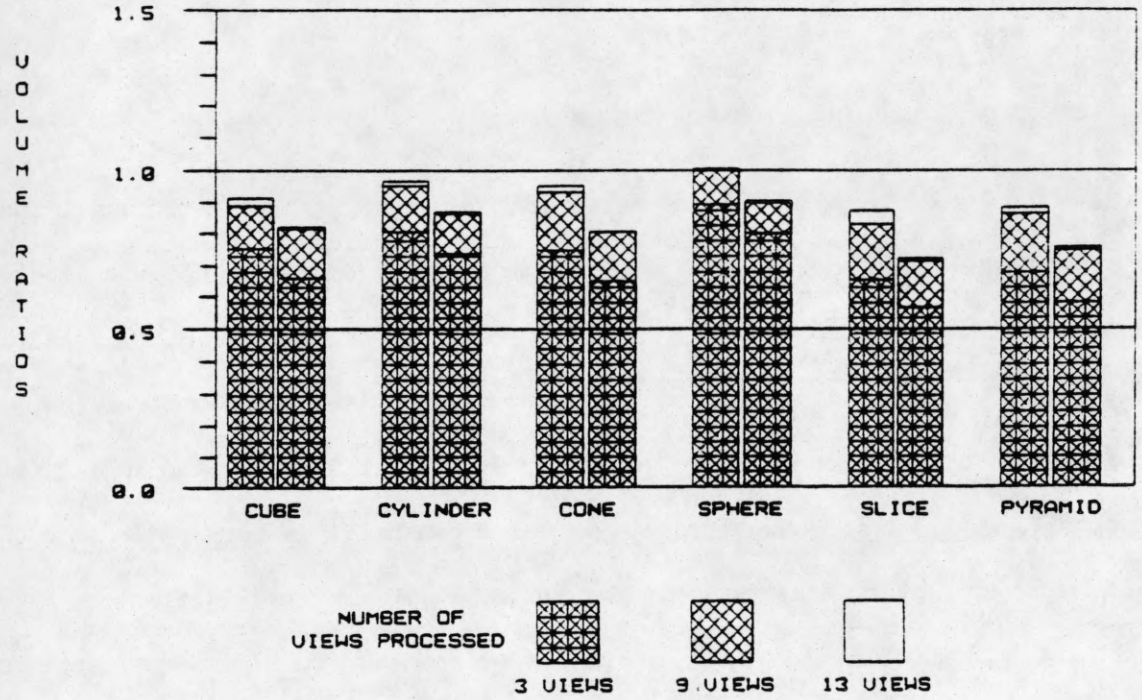
We have included a small cube in the set of test objects to assess the impact of silhouette expansion on accuracy. Since this cube has such a small volume ($\frac{1}{512}$ of the volume of the large cube), any lost volume due to digitization error will have a more noticeable effect on the accuracy measure.

Figure 15 shows the average accuracies over a sample of one hundred random orientations for each object. The results for both the original silhouettes and the expanded silhouettes, after preprocessing, are shown for comparison. Above each object's name are shown two bars. The left bar depicts the accuracy using silhouettes with no preprocessing, and the right bar depicts the accuracy using expanded silhouettes. The average accuracy for the entire set of test objects with no silhouette preprocessing is 93.7% and with silhouette expansion is 76.5%.

Objects Used in Performance Analysis		
Name	Definition	Volume
Cube	x, y, z *in* $[-1, 1]$	8.0
Cylinder	$x^2 + y^2 \leqslant 1$, and z *in* $[-1, 1]$	6.2832
Cone	$x^2 + y^2 \leqslant \frac{(z-1)^2}{4}$, and z *in* $[-1, 1]$	2.0944
Sphere	$x^2 + y^2 + z^2 \leqslant 1$	4.1888
Slice	x, z *in* $[-1, 1]$, and y *in* $[-0.25, 0.25]$	2.0
Pyramid	$x^2 \leqslant \frac{(z-1)^2}{4}$, and $y^2 \leqslant \frac{(z-1)^2}{4}$, and z *in* $[-1, 1]$	2.6667
Wedge	$x^2 \leqslant \frac{(z-1)^2}{4}$, and $y^2 \leqslant \frac{(z-1)^2}{4}$, and z *in* $[0, 0.8]$	0.6293
Octagonal Prism	x, y, z *in* $[-1, 1]$, and if $\sqrt{2}-1 \leqslant x \leqslant 1$ *then* $x-\sqrt{2} \leqslant y \leqslant -x+\sqrt{2}$; if $-\sqrt{2}+1 \leqslant x < \sqrt{2}-1$ *then* $-1 \leqslant y \leqslant 1$; if $-1 \leqslant x < -\sqrt{2}+1$ *then* $-x-\sqrt{2} \leqslant y \leqslant x+\sqrt{2}$	6.6274
Rectangular Prism	x *in* $[0, 1]$, and y, z *in* $[-1, 1]$, and $2y - 1 \leqslant z \leqslant 2y + 1$	2.0
Arc	$x, y \geqslant 0$, and $0.25 \leqslant x^2 + y^2 \leqslant 1$, and z *in* $[-0.25, 0.25]$	0.2945
Small Cube	x, y, z *in* $[-0.125, 0.125]$	0.0156

Figure 14

List of primitive objects used to test the accuracy of the octree generation algorithm.



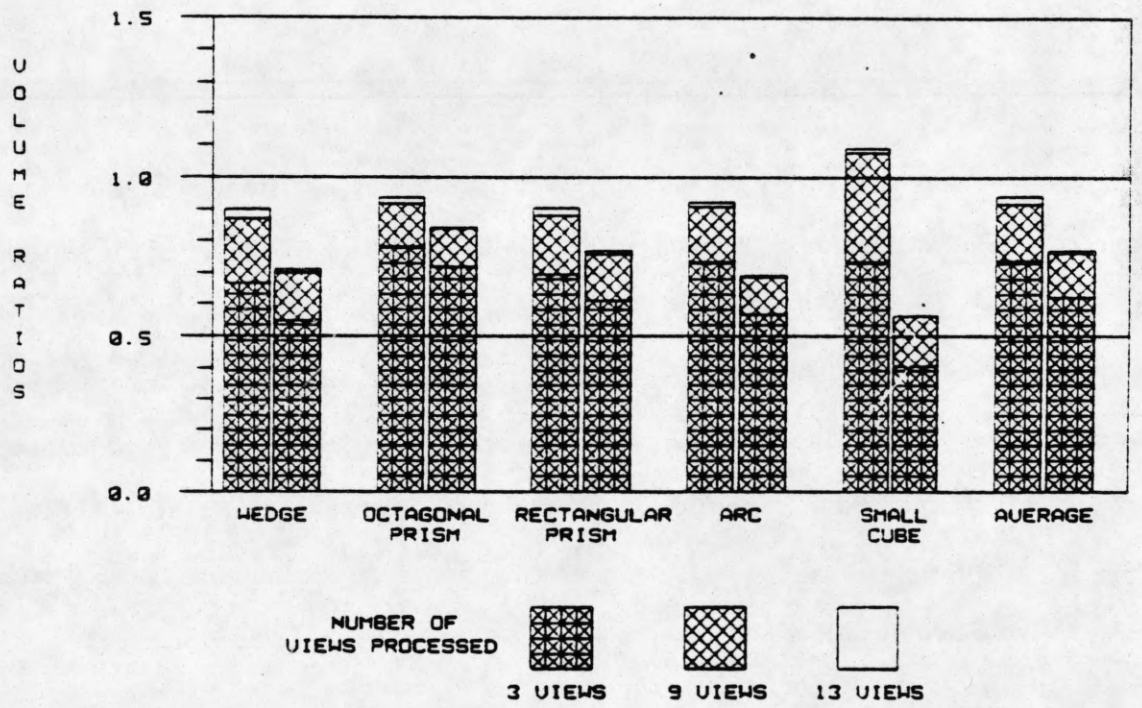


Figure 15

Ratios of the volume of a randomly oriented object to the volume represented by the octree generated from the object's silhouettes. For each object the left bar corresponds to the case of no preprocessing of silhouettes and the right bar corresponds to preprocessing.

3.3. Accuracy-Computation Trade-Off

In many cases it may suffice to have occupancy information which is even less detailed than that provided by the 13 directions. Of greater importance may be the speed at which the octree is generated. Under such conditions it is of interest to know how the accuracy degrades with a decrease in the number of directions. In Figure 15 we have indicated the ratios of actual volumes to the octree volumes corresponding to (1) only face views (3 directions), (2) face and edge views (9 directions), and (3) face, edge and corner views (13 directions). The successive increments in bar heights represent the additional accuracy contributed by the additional silhouettes. The graphs show that for all the objects tested, the additional six edge views contribute a significant amount of new information. The addition of the four corner views to the nine face and edge views, however, increases the accuracy only slightly.

3.3.1. Coarse-to-Fine Computation

The choice and order of the viewing directions used in our algorithm constitute a coarse-to-fine mechanism of acquisition of occupancy information. The face views act as coarse sensors of occupancy. Their spatial orthogonality contributes to independence of the information they provide. The additional edge views are well separated and reveal occupancy of the space half way between the face viewing directions. Thus, they provide finer grain occupancy information. The corner views further increase the density of viewing directions fairly isotropically since the corner viewing directions are located far away from the face and edge viewing directions. Fortunately, these sparsely located directions also allow efficient computation of the octree nodes.

Since the object orientation is random, there is no significance to the absolute direction of viewing. The incremental information provided by a new viewing direction is only a function of the location of the new direction relative to the previous directions. If (α, β) denotes the vector α degrees counterclockwise from the x-axis in the xy plane, and β degrees above the xy plane, then

the 13 directions used in our algorithm are the following: (0,0), (90,0), (0,90), (45,0), (-45,0), (0,45), (0,-45), (90,45), (90,-45), (45,35.3), (-45,35.3), (45,-35.3), (-45,-35.3). These directions are fairly, although not precisely, isotropic collectively, and so are the directions within each of the three subsets corresponding to the face, edge, and corner views. Therefore, the sequential use of the three subsets results in an efficient strategy to acquire occupancy information with increasing accuracy. The information acquisition can be stopped after using either the first one or two sets of viewing directions to obtain different trade-offs between accuracy and computation time. Using the three sets of viewing directions in different orders results in different, near optimal ways of acquiring the silhouette information in a coarse-to-fine manner. For example, if the order (face, edge, corner) is used, then one can stop after using 3, 3 + 6 = 9, or 3 + 6 + 4 = 13 directions; if the order (edge, face, corner) is used, then one can stop after 6, 9, or 13 directions. All different possible orders define the set of all near optimal ways of using different numbers of viewing directions. Given a specific number of viewing directions allowed, say 10, one can use the order (face, corner) or (corner, face) to select the viewing directions. Of course, some numbers of viewing directions, e.g., 8, do not allow near optimal trade-off of accuracy and computation time in the sense discussed here.

3.4. Stability

The measure used in the performance analysis of the derived octree representation is an average computed over a large number of random orientations of the objects. Thus, the results correspond to the expected fraction of the volume represented by the octree that is actually occupied by the object. *In practice*, we may derive the octree for a given *single* orientation of the object, which may be random. The question then arises as to how reliable the resulting representation is. In other words, how stable is the representation from orientation to orientation even though we know how the representation performs on an average over many orientations. Table 1 lists the observed maximum and minimum values of the measure over all 100 observations for each object

and for the case of no preprocessing. Also listed are the average values and the standard deviations of the values. Figure 16 shows the results graphically. Clearly, the smaller the standard deviation, and the smaller the differences among the maximum, minimum, and the average values, the better the stability of the derived octree representation. Table 2 and Figure 17 are analogous to Table 1 and Figure 16 but for the case with silhouette preprocessing. The improvement in the stability of the representation with increasing number of views can be seen in Tables 1 and 2 and in Figures 16 and 17. Figures 23 through 33 show a graphic display of the objects represented by the constructed octrees.

3.5. Complex Objects

An inadequacy of the above performance analysis is the simplicity of the objects analyzed. Originally, these objects were chosen because they are diverse and fairly powerful to serve as geometrical primitives to construct arbitrary objects. However, the construction entails *simultaneous* presence of these objects in the octree space, as components of the larger object whose octree representation is to be derived. Their relative configuration is determined by the complexity of the object to be constructed. For example, consider the object shown in Figure 18 consisting of the volumetric primitives of Figure 13. The spatial configuration of the components leads to their mutual occlusion when the object is viewed from an arbitrary direction. Thus, the already incomplete information in the silhouettes about the object is further confounded by self-occlusion. Of course, unlike surface concavities, the information about occupancy of regions self-occluded from a viewpoint may be recovered if other, allowed directions are more revealing. The availability of a large number of viewing directions assumes increased importance in this context.

To test the performance of our algorithm over more complex objects than shown in Figure 13, we conducted experiments with the object shown in Figure 18. Table 3 lists the results analogous to those given in Tables 1 and 2. Figure 19 is analogous to Figures 16 and 17. Figure 35 shows a graphic display of the self-occluding object in Figure 18 as represented by the constructed octree.

The upright rectangular solids have reproduced well without any staircase effect because these are oriented with their faces parallel to the faces of the universe cube. Parts of the curved surfaces of the circular cylinders are lost since these parts were occluded by the rectangular solids in all the silhouette views used.

Table 1

Stability characteristics of the generated octree with no silhouette preprocessing. The entries are the observed values of the accuracy measure -- the ratio of the actual to generated volume.

Object	Average			Standard Deviation			High Value			Low Value		
	Face	Face + Edge	Face + Edge + Corner	Face	Face + Edge	Face + Edge + Corner	Face	Face + Edge	Face + Edge + Corner	Face	Face + Edge	Face + Edge + Corner
Cube	0.751	0.884	0.909	0.102	0.041	0.031	0.996	0.987	1.018	0.601	0.810	0.844
Cylinder	0.801	0.947	0.960	0.052	0.021	0.015	0.969	0.992	0.991	0.738	0.910	0.931
Cone	0.743	0.931	0.948	0.074	0.039	0.027	0.904	1.003	1.000	0.617	0.866	0.892
Sphere	0.889	0.995	1.001	0.000	0.000	0.000	0.889	0.995	1.001	0.889	0.995	1.001
Slice	0.653	0.827	0.869	0.141	0.070	0.061	0.952	0.991	0.985	0.453	0.714	0.766
Pyramid	0.677	0.862	0.881	0.082	0.041	0.033	0.960	0.980	0.973	0.539	0.774	0.818
Wedge	0.665	0.867	0.893	0.071	0.037	0.036	0.894	0.966	0.988	0.529	0.784	0.820
Octagonal Prism	0.781	0.913	0.933	0.061	0.025	0.023	0.999	0.998	1.015	0.705	0.866	0.899
Rectangular Prism	0.693	0.875	0.899	0.074	0.043	0.036	0.912	0.979	0.992	0.543	0.791	0.814
Arc	0.728	0.905	0.919	0.097	0.044	0.035	1.018	1.015	1.001	0.590	0.828	0.833
Small Cube	0.730	1.076	1.090	0.094	0.066	0.070	0.950	1.303	1.303	0.566	0.982	0.999
Average	0.737	0.917	0.937	0.077	0.039	0.033	0.949	1.019	1.024	0.615	0.847	0.874

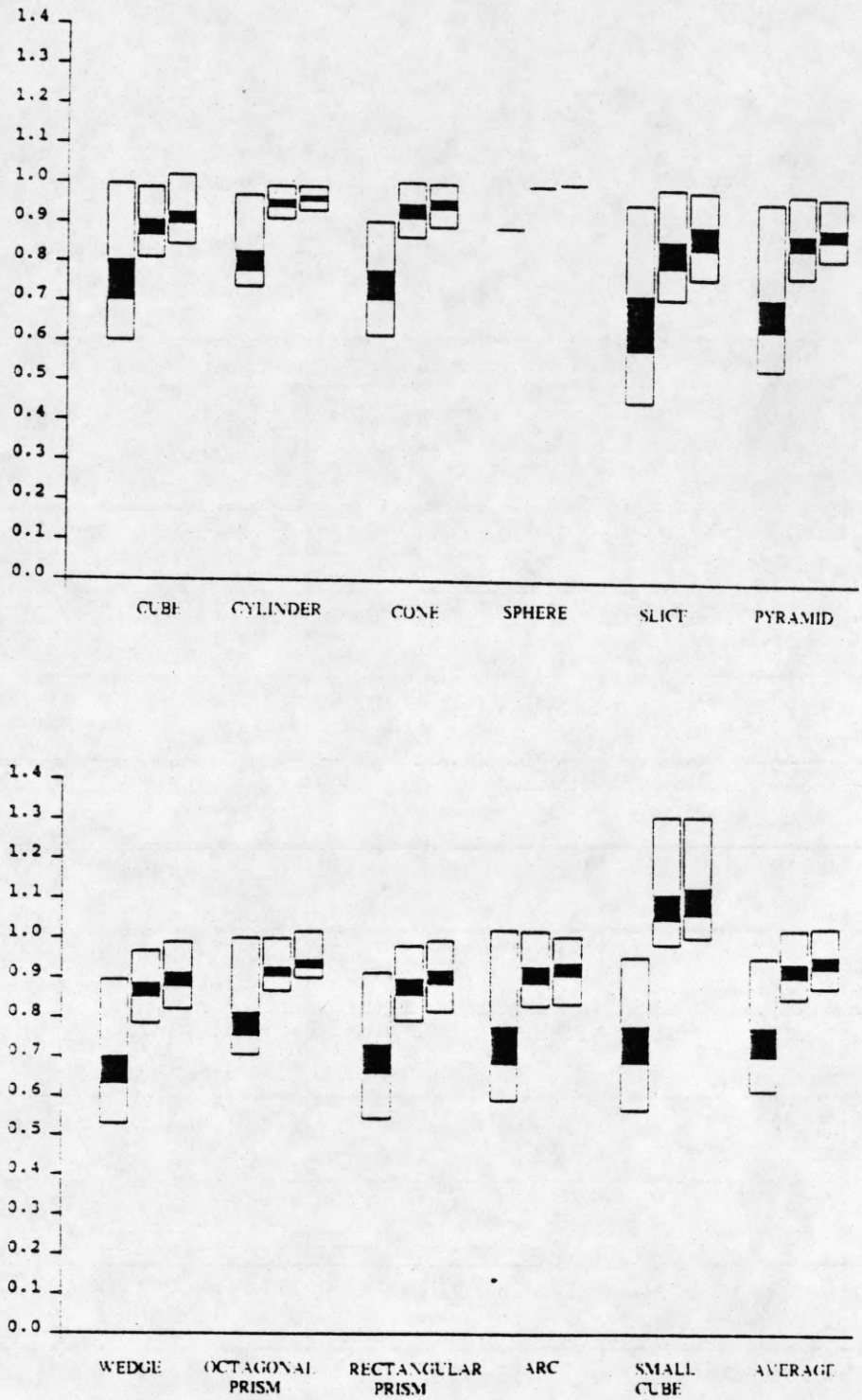


Figure 16

Graphical depiction of the stability characteristics of Table 1. For each object the three bars correspond, respectively, to 3, 9, and 13 views, with no silhouette preprocessing. Each bar extends from the minimum to the maximum observed measure value over the 100 observations made. The band in the bar is centered at the observed average value and has a thickness of the observed standard deviation.

Table 2

Stability characteristics of the generated octree with silhouette preprocessing. The entries are the observed values of the accuracy measure -- the ratio of the actual to generated volume.

Object	Average			Standard Deviation			High Value			Low Value		
	Face	Face + Edge	Face + Edge + Corner	Face	Face + Edge	Face + Edge + Corner	Face	Face + Edge	Face + Edge + Corner	Face	Face + Edge	Face + Edge + Corner
Cube	0.659	0.811	0.820	0.080	0.035	0.027	0.858	0.900	0.911	0.556	0.742	0.765
Cylinder	0.736	0.862	0.865	0.051	0.017	0.016	0.906	0.914	0.914	0.670	0.827	0.839
Cone	0.650	0.802	0.805	0.061	0.026	0.024	0.769	0.867	0.873	0.543	0.750	0.758
Sphere	0.799	0.896	0.898	0.000	0.000	0.000	0.799	0.896	0.898	0.799	0.896	0.898
Slice	0.568	0.711	0.720	0.123	0.055	0.052	0.843	0.828	0.846	0.394	0.615	0.637
Pyramid	0.582	0.746	0.756	0.061	0.033	0.025	0.738	0.829	0.816	0.459	0.680	0.699
Wedge	0.546	0.695	0.704	0.056	0.027	0.027	0.755	0.771	0.776	0.448	0.622	0.659
Octagonal Prism	0.717	0.836	0.837	0.064	0.024	0.019	0.903	0.900	0.899	0.639	0.788	0.806
Rectangular Prism	0.612	0.753	0.762	0.079	0.033	0.028	0.830	0.858	0.831	0.473	0.683	0.701
Arc	0.566	0.683	0.686	0.068	0.028	0.028	0.732	0.758	0.773	0.465	0.636	0.638
Small Cube	0.400	0.561	0.561	0.044	0.037	0.038	0.518	0.698	0.665	0.335	0.502	0.505
Average	0.621	0.759	0.765	0.062	0.029	0.026	0.787	0.838	0.836	0.526	0.704	0.719

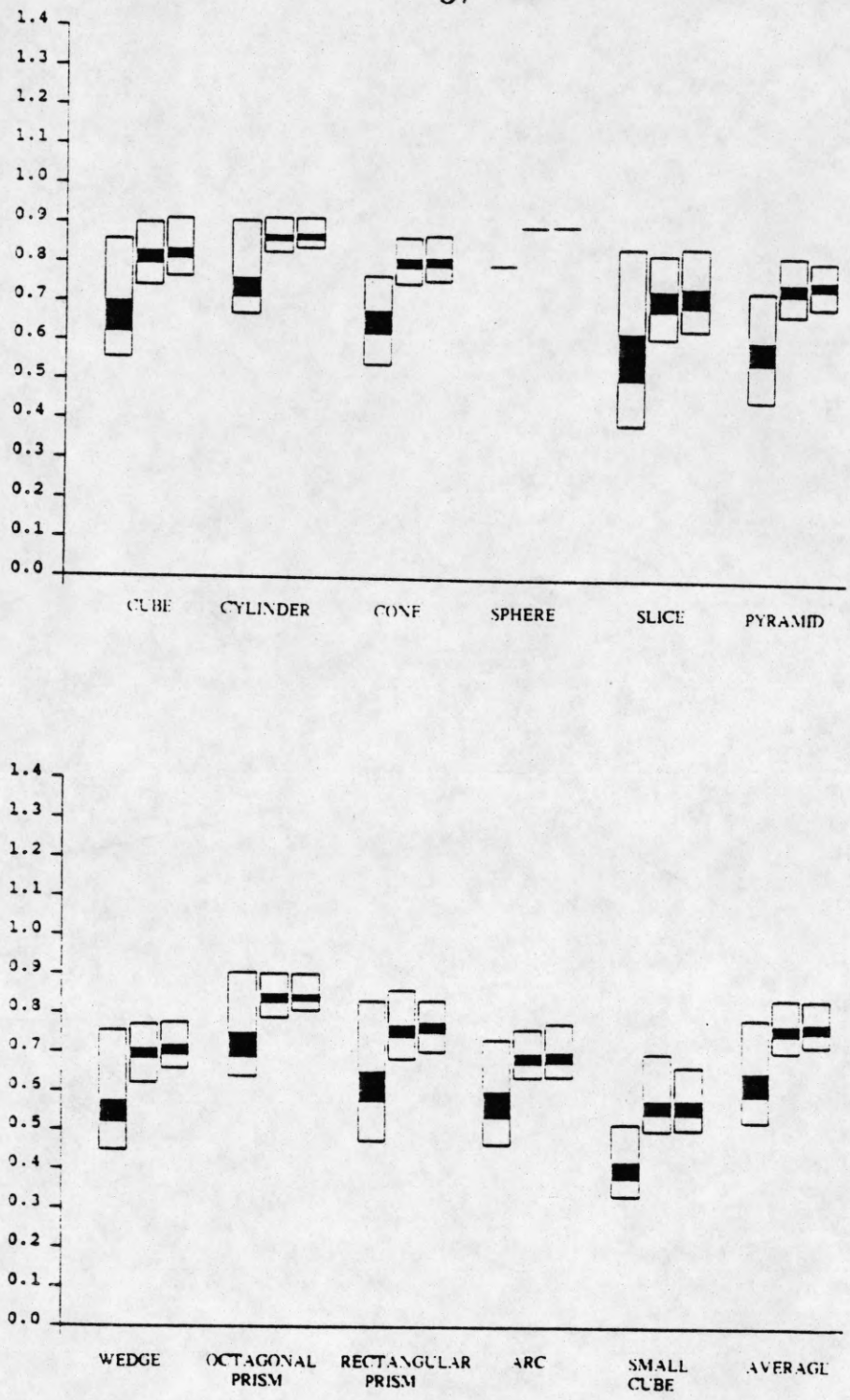


Figure 17

Graphical depiction of the stability characteristics of Table 1. For each object the three bars correspond, respectively, to 3, 9, and 13 views, with silhouette preprocessing. Each bar extends from the minimum to the maximum observed measure value over the 100 observations made. The band in the bar is centered at the observed average value and has a thickness of the observed standard deviation.

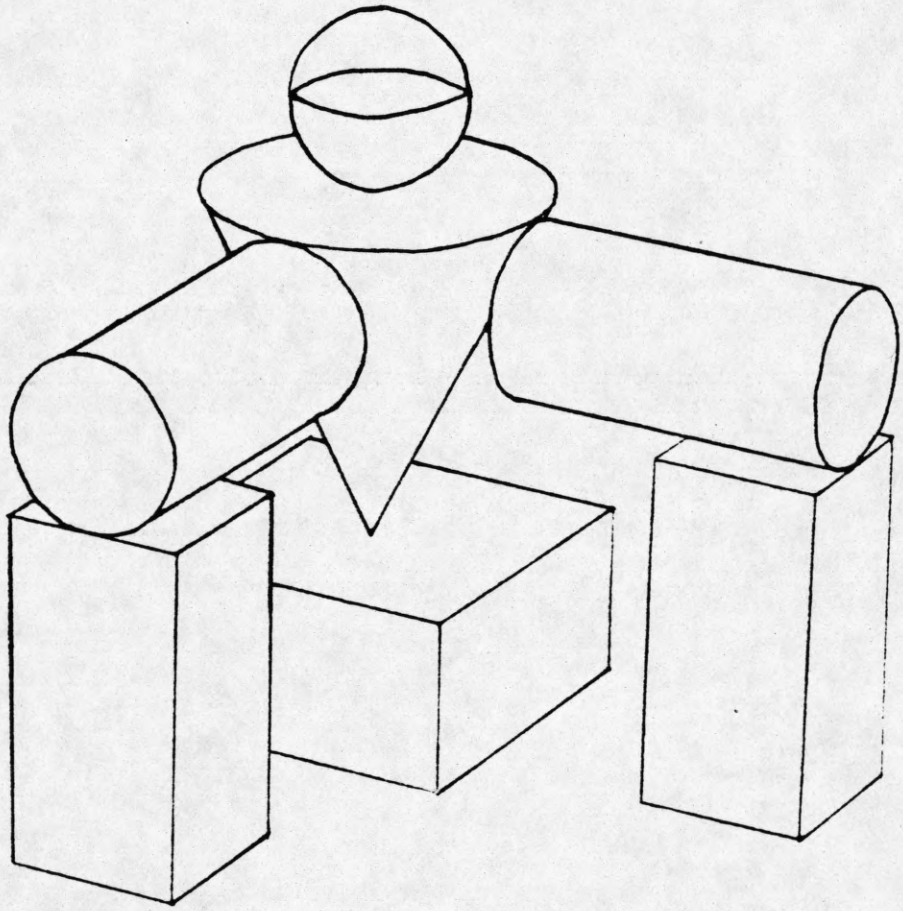


Figure 18

A self-occluding object used in testing the octree generation algorithm.

Table 3

Stability characteristics of the generated octree for the object shown in Figure 18. The entries are the observed values of the accuracy measure -- the ratio of the actual to generated volume.

Silhouette Preprocessing	Average			Standard Deviation			High Value			Low Value		
	Face	Face + Edge	Face + Edge + Corner	Face	Face + Edge	Face + Edge + Corner	Face	Face + Edge	Face + Edge + Corner	Face	Face + Edge	Face + Edge + Corner
Without	0.542	0.821	0.864	0.091	0.038	0.037	0.732	0.921	0.991	0.373	0.757	0.802
With	0.438	0.626	0.640	0.071	0.024	0.021	0.665	0.691	0.710	0.308	0.578	0.608

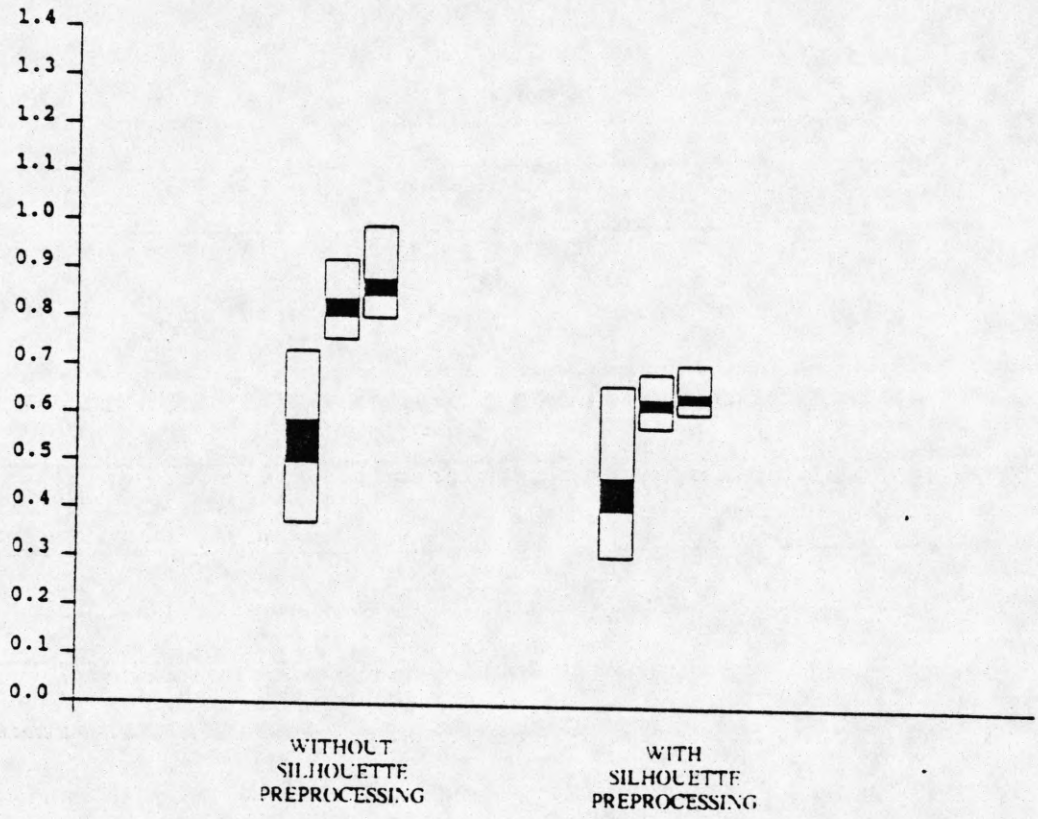


Figure 19

Graphical depiction of the stability characteristics of the self-occluding object shown in Figure 18. The three bars correspond to the 3, 9, and 13 views. Each bar extends from the minimum to the maximum observed value over the 100 observations made. The band in the bar is centered at the observed average value and has a thickness of the observed standard deviation. The three bars on the left show the results with no silhouette preprocessing and the three bars on the right show the results with silhouette preprocessing.

3.6. Experimental Details

The algorithms were implemented in C on a VAX 11/780 computer running the Berkeley 4.2 version of the UNIX operating system. After a Gould 9000 computer became available, the programs were converted to run on that machine since it was several times faster and had twice the virtual memory. We experienced a five-fold increase in speed when the same programs were run on the Gould. A VICOM computer was used to acquire and process 512 × 512 images. A Pascal program running under the VERSAdos operating system on the VICOM was used to perform the resampling of the images showing edge views. After thresholding the image to produce a silhouette, the image was transferred over a high-speed, dedicated DMA channel to the VAX, where the image data was used as input to the octree generating programs.

Some simple modifications were made to the edge view algorithm to improve its efficiency. For example, some pixels in the silhouette image array (especially those toward the middle) are examined several times by the recursive procedure since center squares overlap with quadrant squares. Some of these repeated pixel examinations can be avoided if these pixels are contained in large regions of uniform intensity values. Thus, if two horizontally adjacent quadrants have pixel values which are all zeroes or all ones, then the overlapping center square need not be examined. The necessary changes were incorporated into our algorithm so that a center square was only examined if the overlapping horizontally adjacent quadrants were not uniformly ones or zeroes.

The C programs were timed using test data in the form of 64 × 64 arrays representing binary images of varying complexity. The average cpu time spent in the octree generation procedures was recorded and plotted as a function of the number of nodes in the octree. (See Figure 20.) The octree generation times for a single face view and a single edge view are plotted separately. The data for the edge view do not include the time to resample the rectangular image into a square array. The graph shows that the execution time increases linearly with the number of nodes in the octree for a fixed image size.

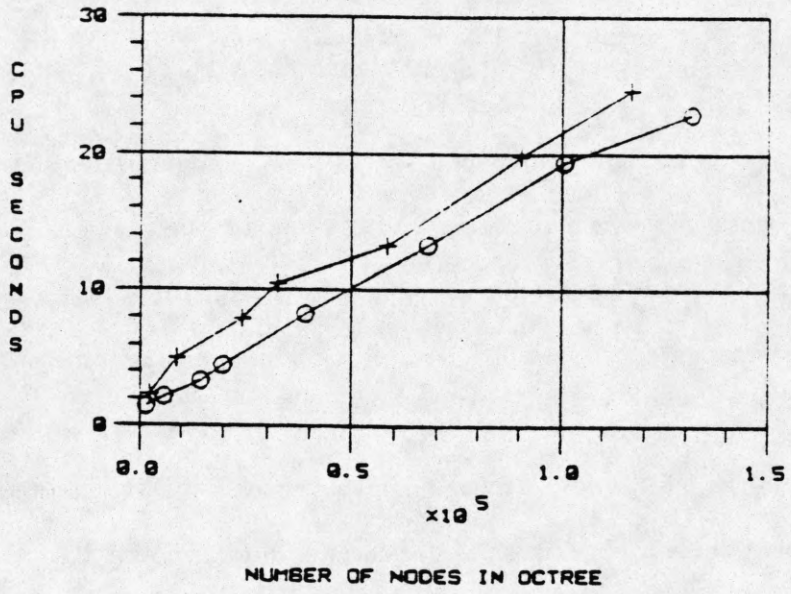


Figure 20

o represents the octree generation time for a face view
+ represents the octree generation time for an edge view

The accuracy of the octree increases as its depth increases. For the face views there is a direct relationship between the depth of the octree and the image resolution. The octants at the lowest level in an octree of depth d correspond directly to individual pixels in a $2^d \times 2^d$ image. So, for example, an octree of depth 6 can capture a single pixel resolution of 64×64 face views. If the image resolution is finer than the maximum allowed tree depth, then octants at the lowest level in the tree represent square regions of image pixels instead of individual pixels. When the square regions are not of uniform value, some criteria must be used to determine the color of the representative octree node. In our case, we chose to label the node black if at least ¼ of the pixels were black. This resulted in a good approximation without being too conservative. Alternatively, one could label the octree node black if any one of the pixels was black but this would vastly overestimate the size of the object.

The depth of the generated octree in our experiments is determined by a parameter under user control. In all the experiments described earlier in this section, we set the tree depth so that the deepest node corresponds to a pixel in the face views. The same depth value is used for the edge and corner views. On the VAX, we could not use silhouette views larger than 128 × 128 since octrees of depth greater than 7 were too large for the virtual memory space of the VAX. On the Gould machine, which has a larger virtual memory space, we successfully created a depth 8 octree from 256 × 256 silhouette images of a coffee cup (see Figure 34).

4. LINE DRAWING GENERATION ALGORITHM

During our work on the octree generation algorithm, it was necessary to monitor the accuracy of the octree representation constructed at different stages of development. First, we did this by printing each node in the octree with its associated "black" or "grey" label ("white" or empty nodes were not stored), and then 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 which the octree represented. This section describes an algorithm we developed for this purpose. The algorithm produces a line drawing of an object represented by an octree. The object is drawn in perspective with hidden lines removed. An alternative method of displaying the object represented by an octree is described by Meagher [8,9]. His algorithm produces a surface display from octree after hidden surface removal. However, surface displays depend upon 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 which can be easily drawn.

The algorithm consists of the following steps. First, the octree is traversed, visiting octants in order of increasing distance to the viewer. For each black leaf node, graphics information is collected and stored in a "graphics node". (To avoid confusion with octree nodes, we will 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. 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, 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 also 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 in perspective 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 which projects, in general, as a hexagon. The numbering schemes for the corners and edges of a projected cube are given in Figure 21. 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 graphics nodes closer to the beginning of the list. Since graphics nodes closer to the beginning of the list are closer to the viewer, any overlap represents part of a graphics node which should be hidden and is therefore removed.

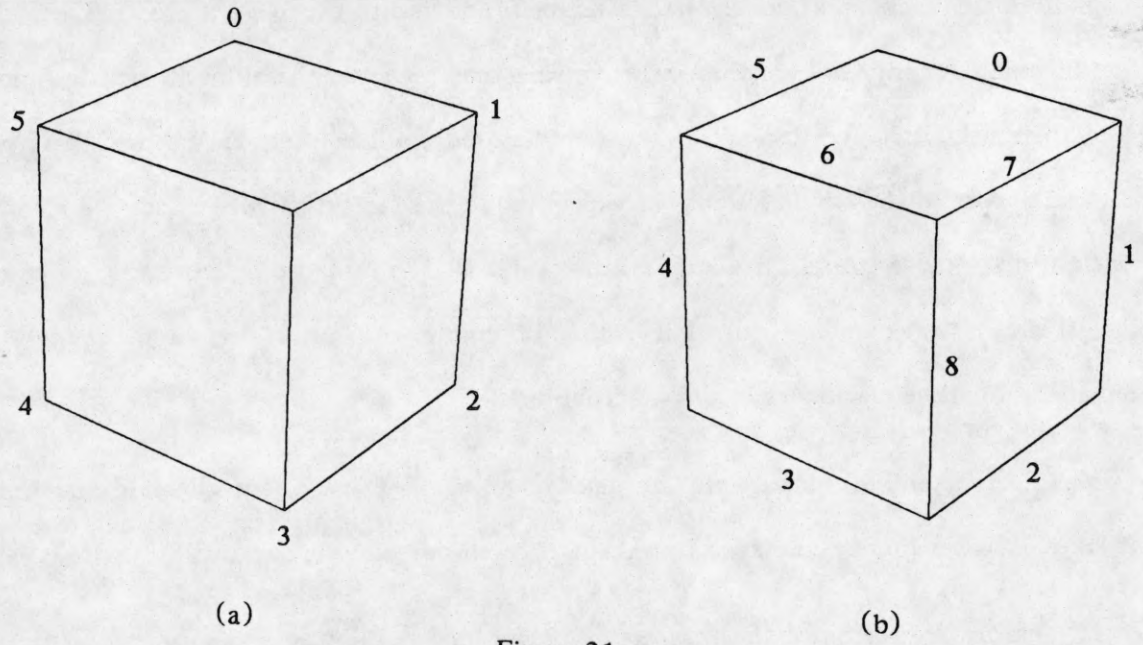


Figure 21

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

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 re-labeling the octants.

4.1. Elimination of "Cracks"

A problem unique to line drawing from octree 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. 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 if 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 which is created at the time the black octant is first encountered.

Only black octants have associated graphics nodes, and a black octant which 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 perspective 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.

4.2. Elimination of Hidden Lines

After cracks are removed and the perspective projections are calculated, the hidden lines are removed [4,10] using a straightforward 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 computation time to eliminate hidden lines is proportional to the square of the number of graphics nodes.

To carry out the intersection tests, a modified Cohen-Sutherland clipping algorithm is used. (See Figure 22.) The line which contains an edge of a projected octant defines two half planes, one of which contains the projection and one which does not. The six edges define six half planes which 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 which contain that point.

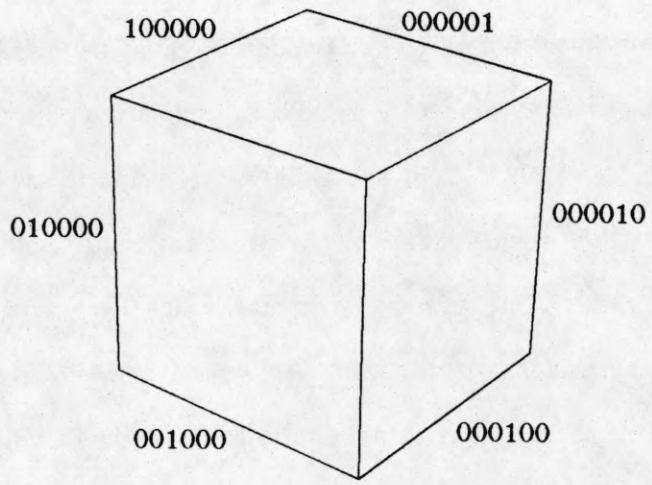


Figure 22

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

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 non-zero. 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.

4.3. 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. So 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 which 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 (for the perspective projection), 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.

4.4. Representation of Graphics Information

To facilitate integer instead of floating point representations, the side length of an octant at the lowest level in the octree (that is, 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 (that is, 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; and, 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;
```

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.

4.5. Performance of the Line Drawing Generation Algorithm

The octree generation algorithm followed by the line drawing generation algorithm should provide a display of the original object. Any differences between an object and its line drawing represent the approximations and errors involved in octree generation, and thus serve as a quick method of evaluating the performance of the octree generation algorithm.

4.5.1. Example Line Drawings

Figures 23 through 33 show the line drawings generated by our algorithm for the octrees of the test objects in Figure 14. Figure 34 is a line drawing generated from an octree that was obtained from gray level, silhouette images of a coffee cup. Figure 35 shows the drawing for the octree of the self-occluding object in Figure 18. Figures 36 and 37 show two different perspective views of 8 cubes. Figure 38 shows an object whose silhouette is a diamond when viewed from any face of the universe cube which contains it. The line drawing algorithm was executed on a VAX and the output sent to a QMS laser printer.

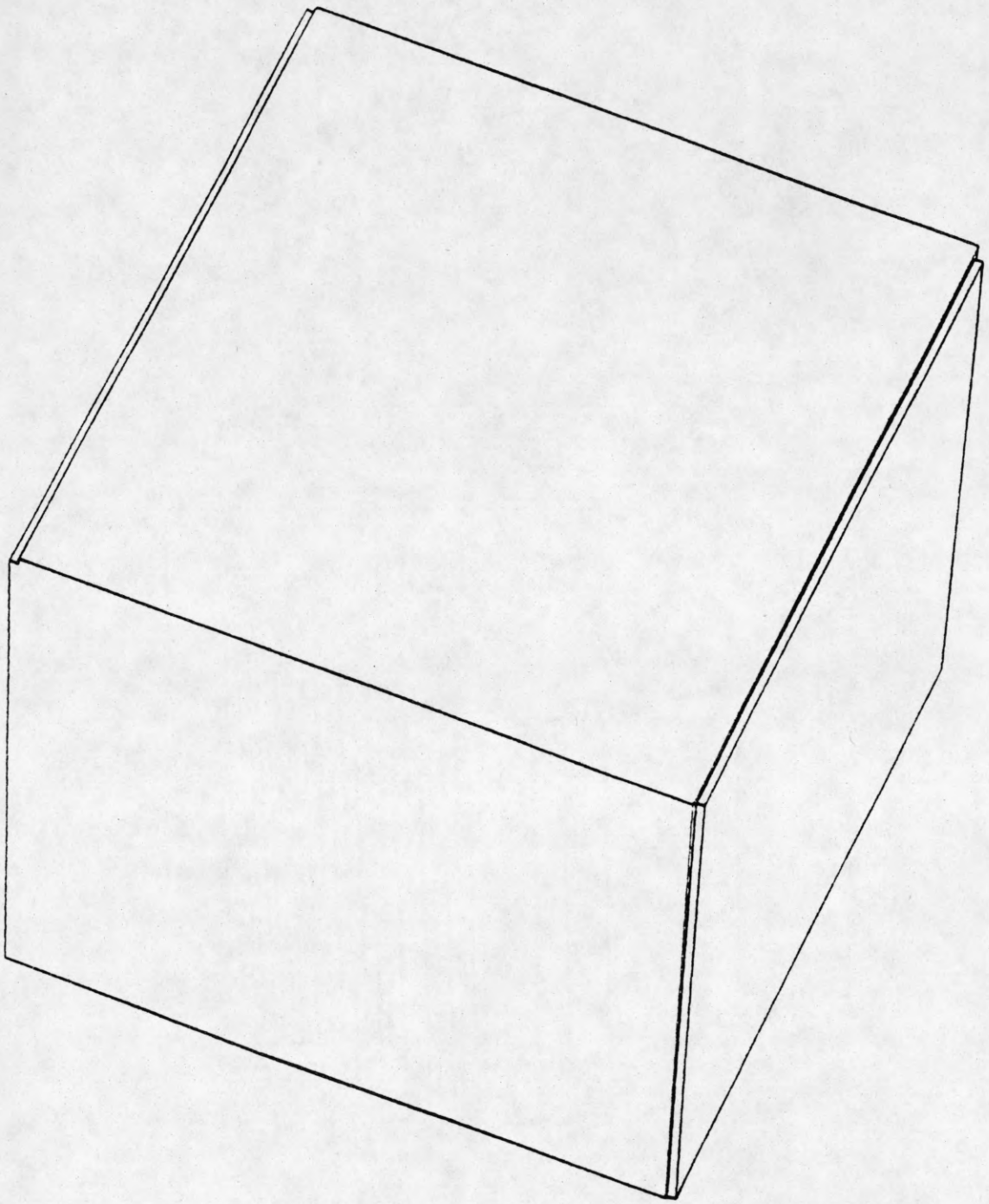


Figure 23

The line drawing for the derived octree representation of the cube in Figure 14.

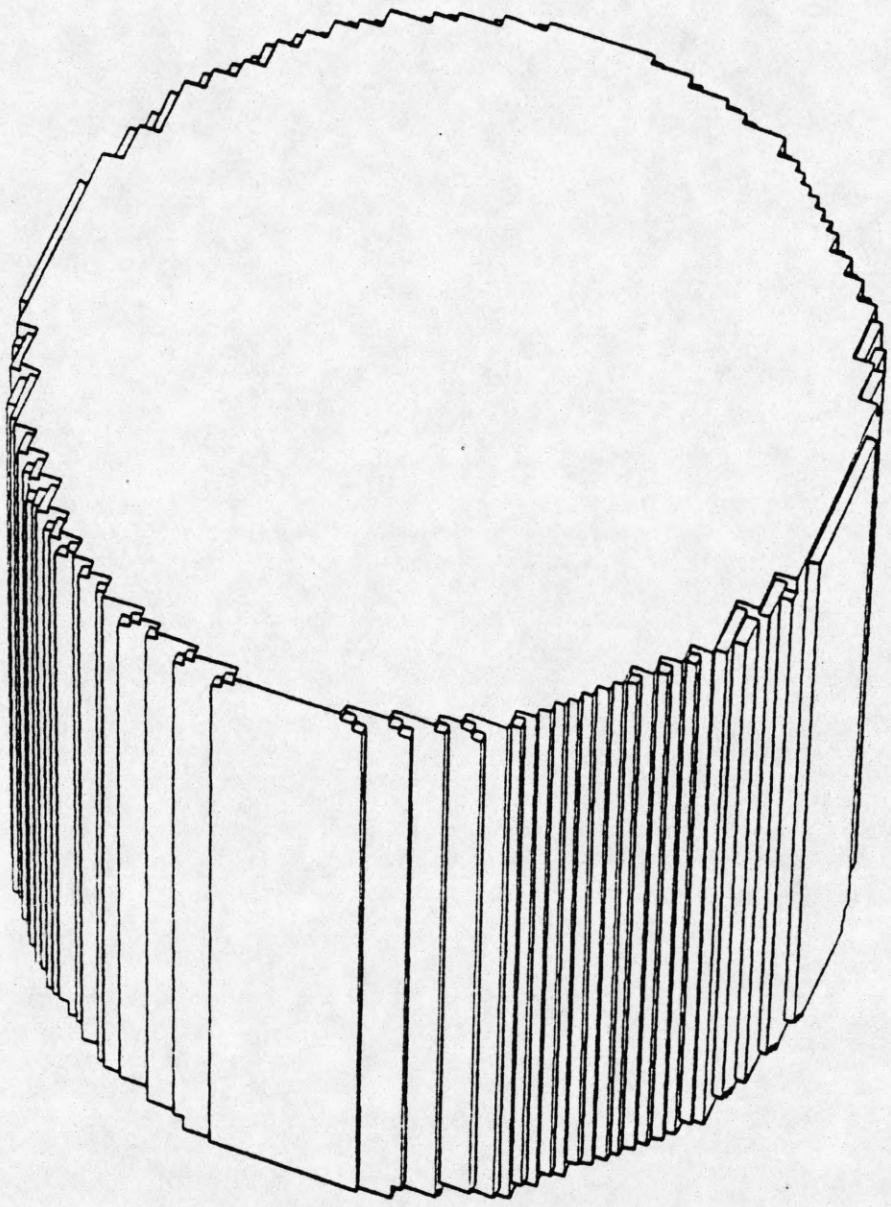


Figure 24

The line drawing for the derived octree representation of the cylinder in Figure 14.

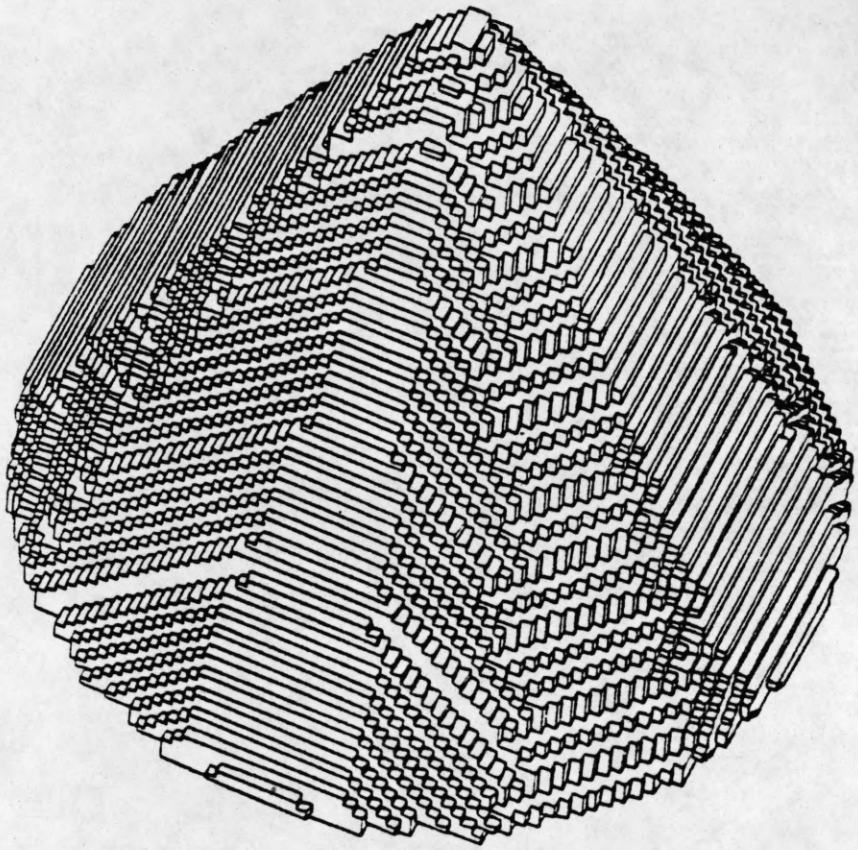


Figure 25

The line drawing for the derived octree representation of the cone in Figure 14.

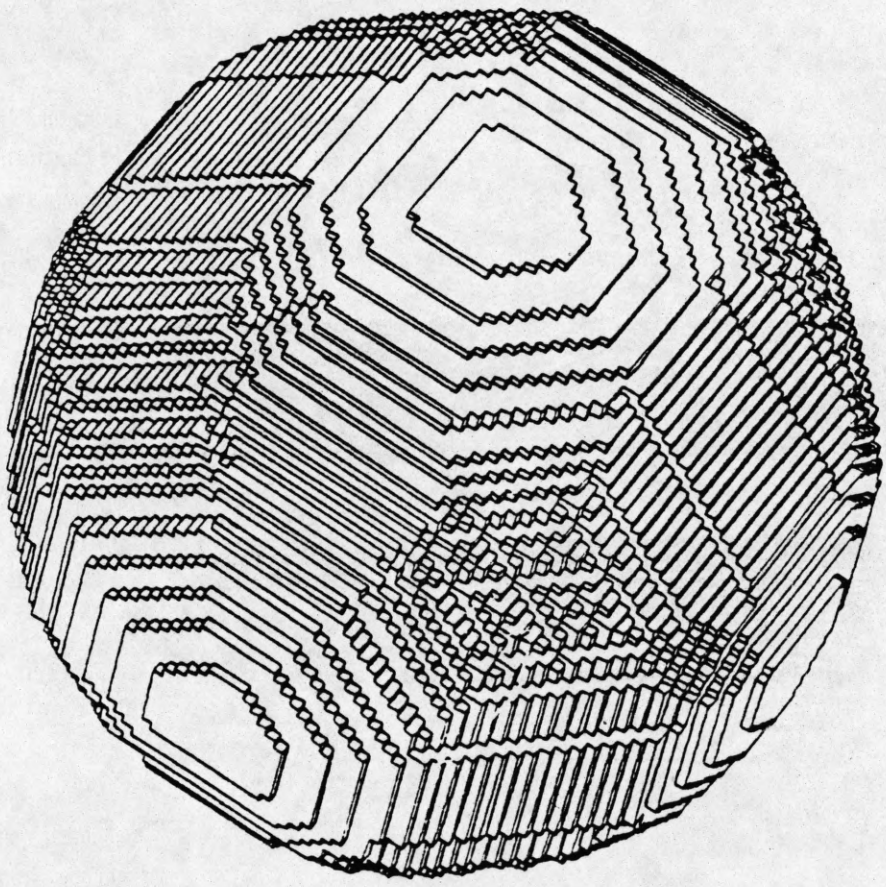


Figure 26

The line drawing for the derived octree representation of the sphere in Figure 14.

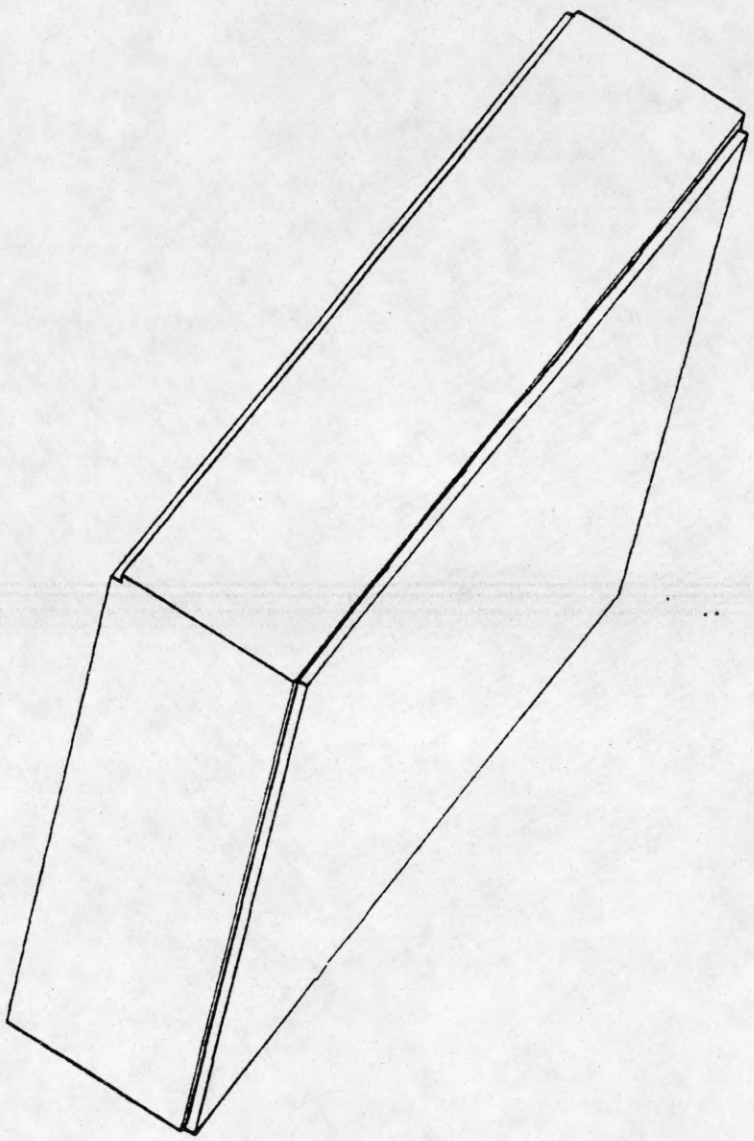


Figure 27

The line drawing for the derived octree representation of the slice in Figure 14.

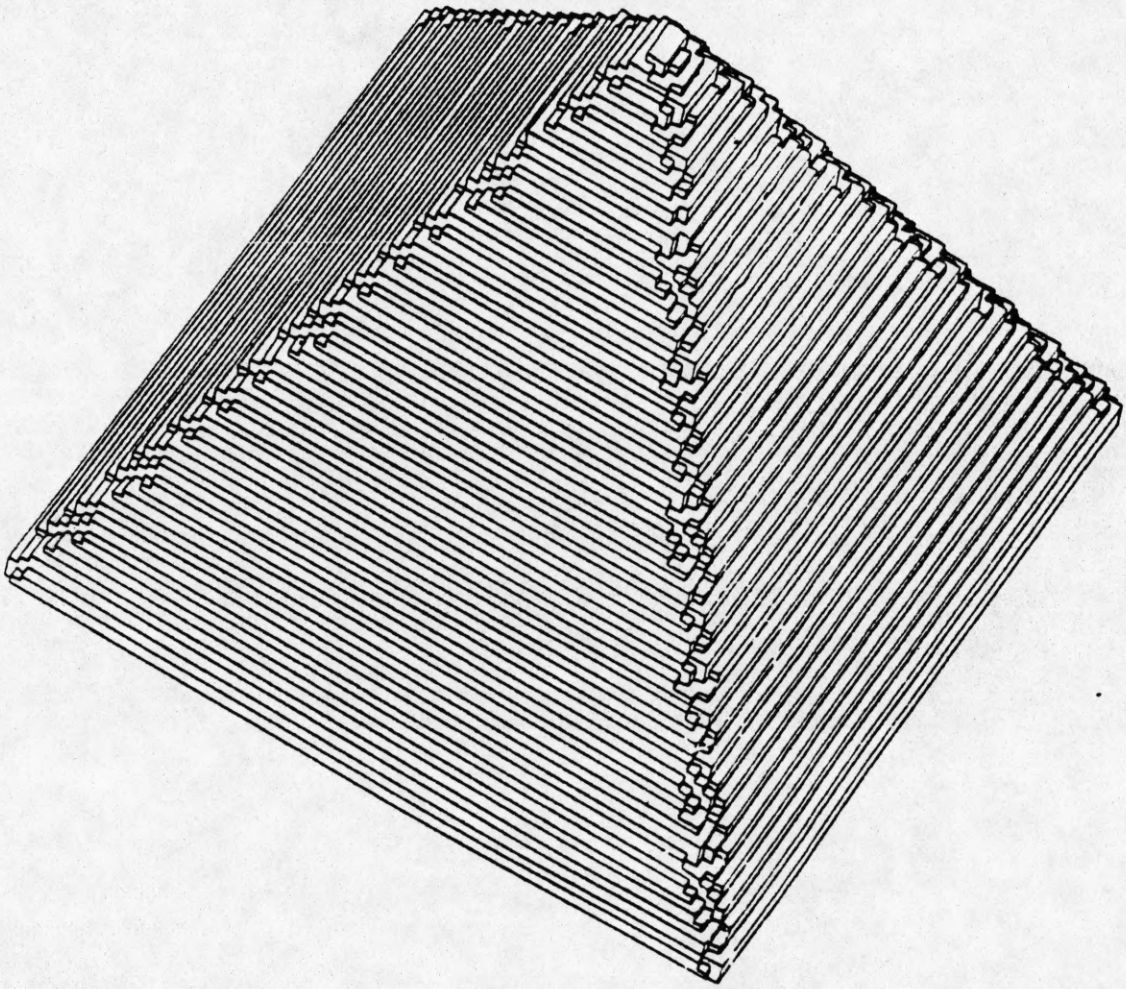


Figure 28

The line drawing for the derived octree representation of the pyramid in Figure 14.

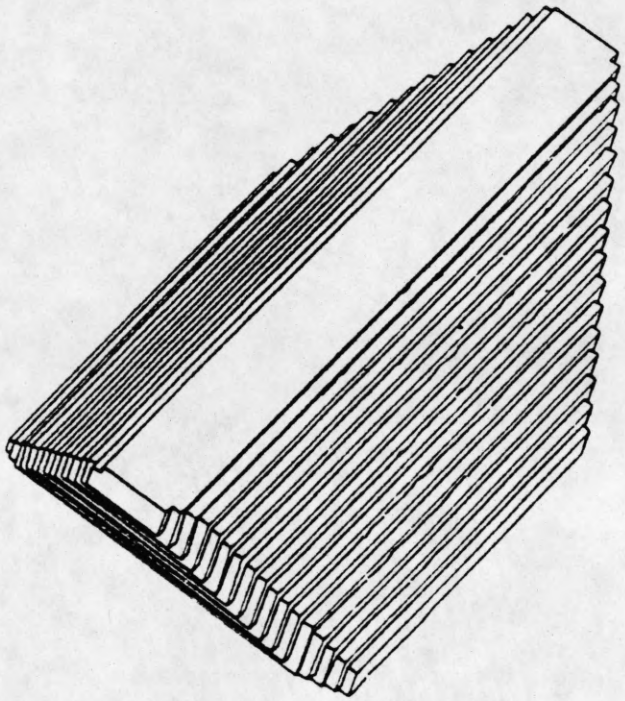


Figure 29

The line drawing for the derived octree representation of the wedge in Figure 14.

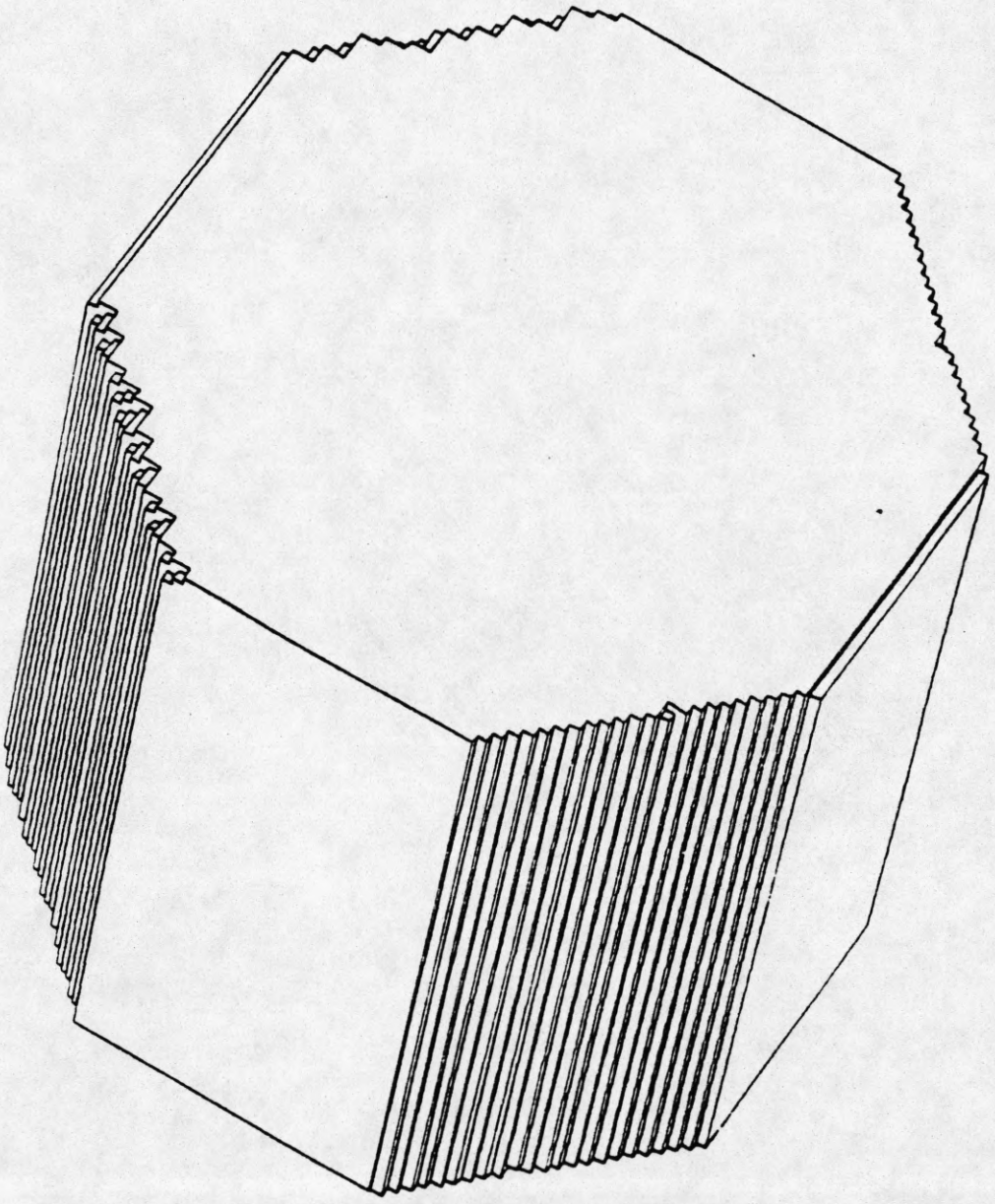


Figure 30

The line drawing for the derived octree representation of the octagonal prism in Figure 14.

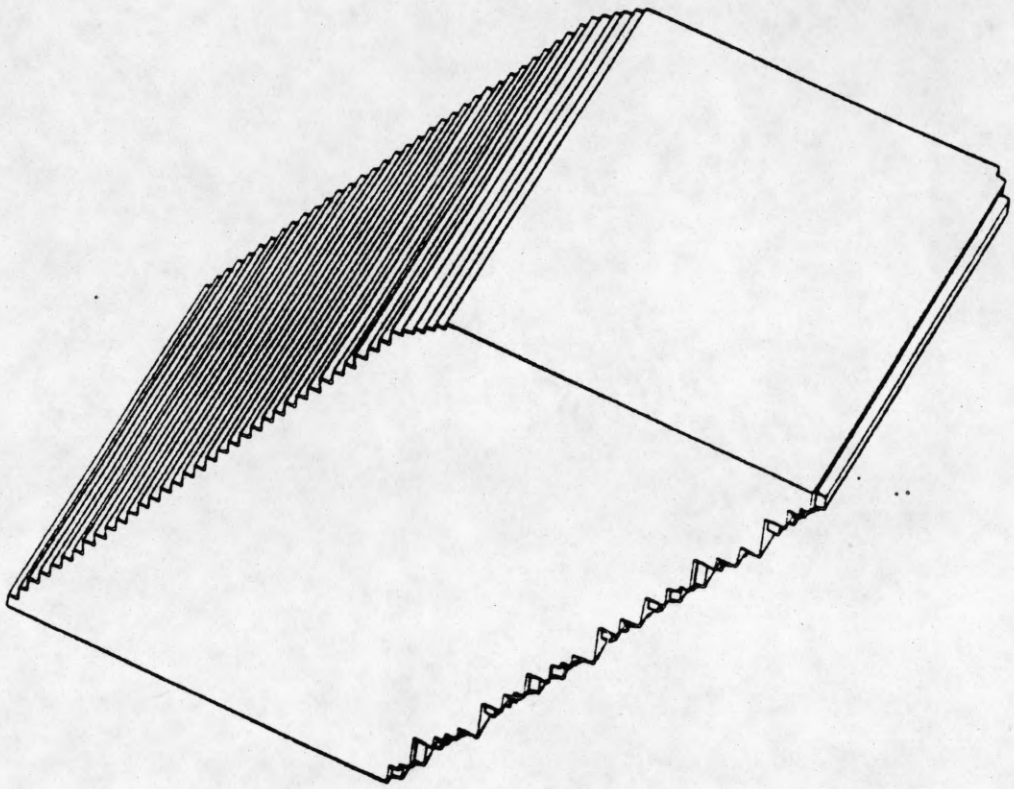


Figure 31

The line drawing for the derived octree representation of the rectangular prism in Figure 14.

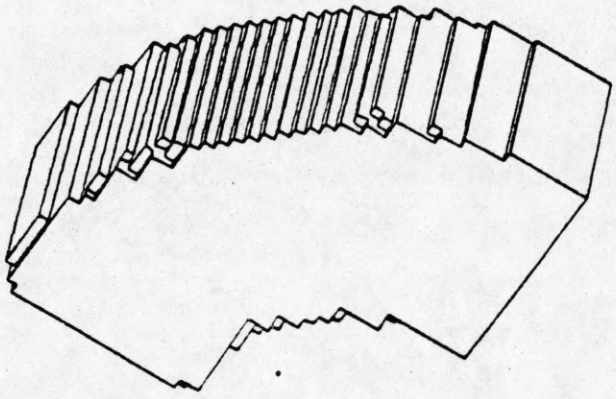


Figure 32

The line drawing for the derived octree representation of the arc in Figure 14.

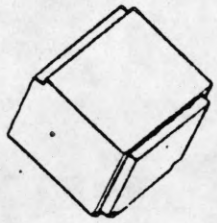


Figure 33

The line drawing for the derived octree representation of the small cube in Figure 14.

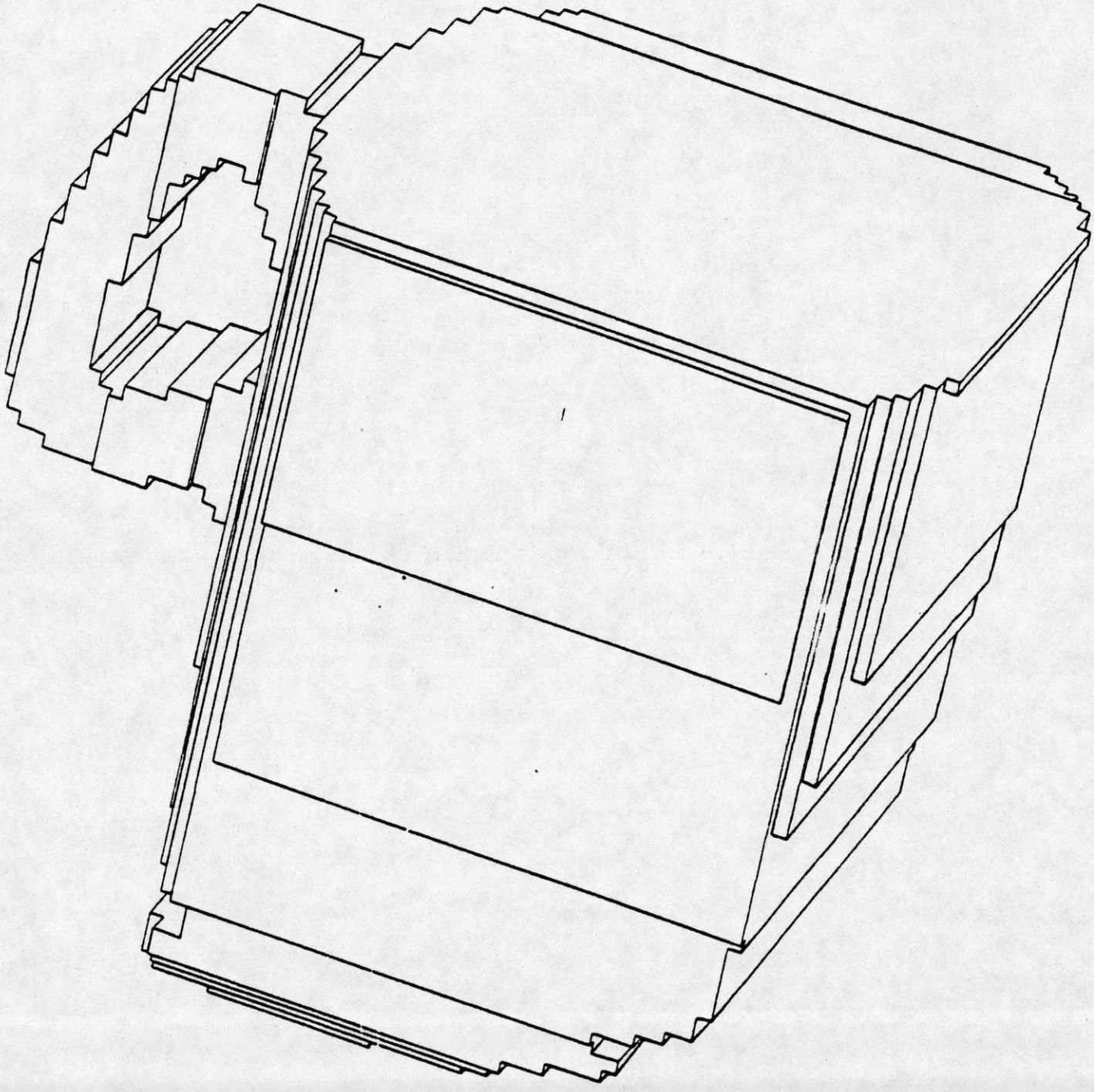


Figure 34

A line drawing of a coffee cup.

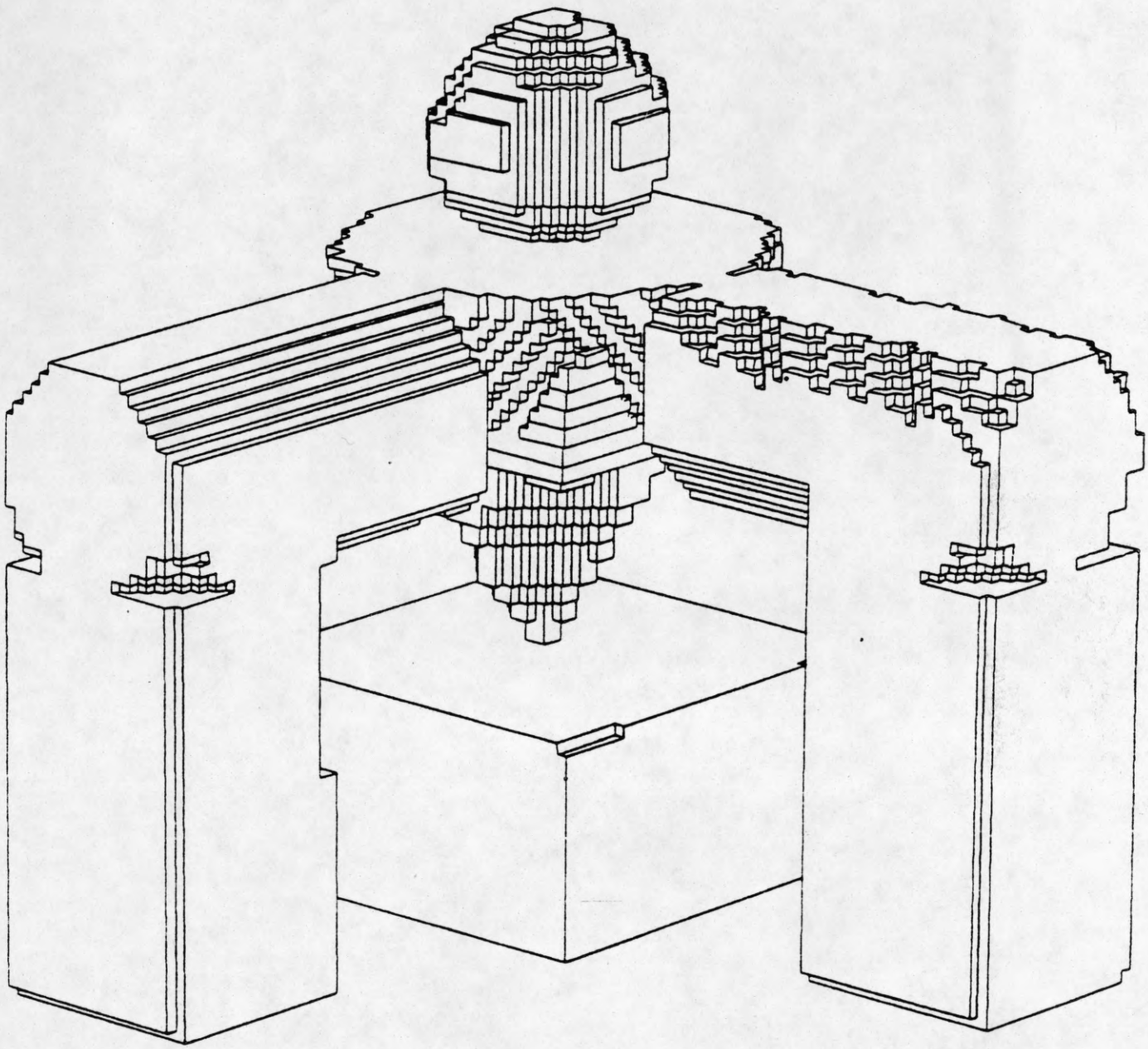


Figure 35

The line drawing for the derived octree representation of the self-occluding object in Figure 18.

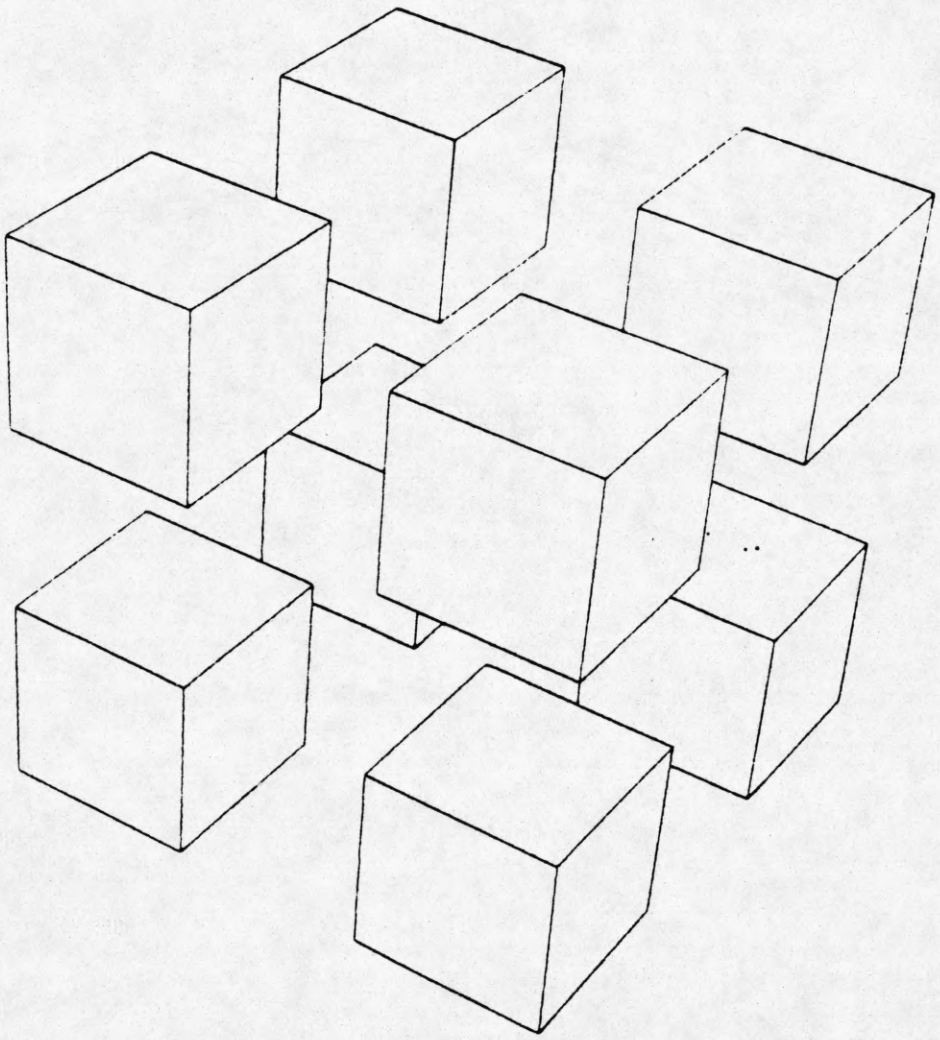


Figure 36

A line drawing of eight cubes.

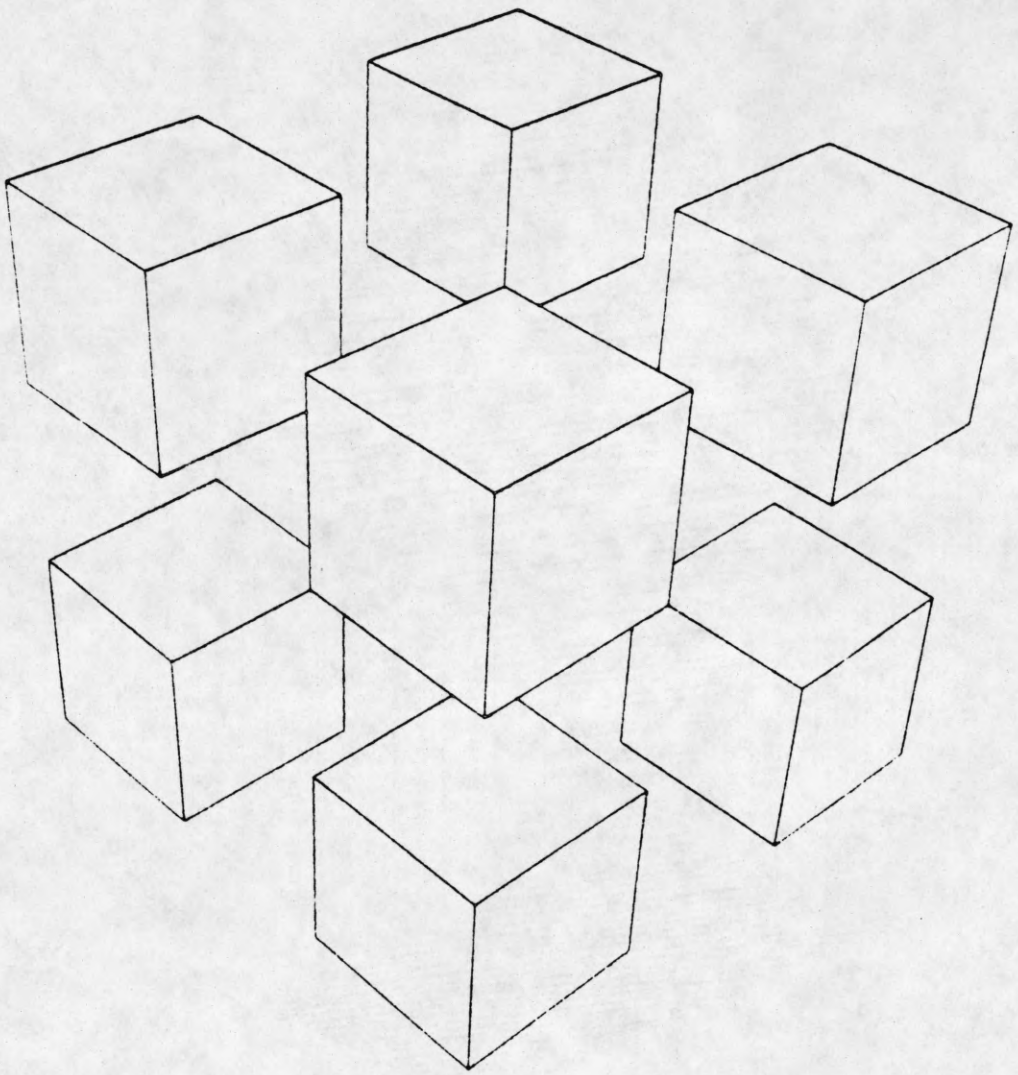


Figure 37

Another view of the eight cubes in Figure 36.

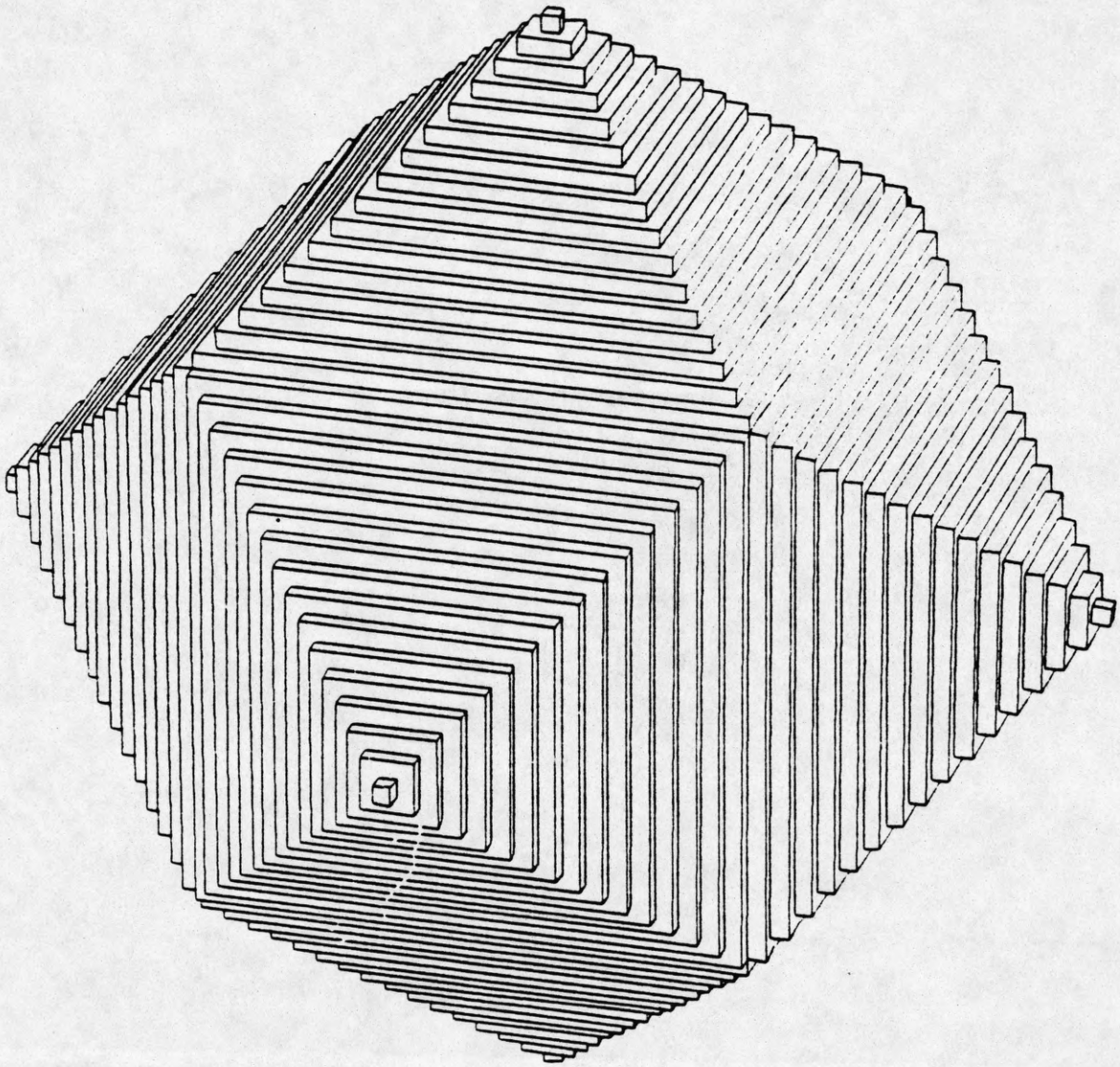


Figure 38

A line drawing of a diamond-shaped object.

4.5.2. Execution Time of the Line Drawing Generation Algorithm

Figure 39 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. We would normally expect the execution time to increase roughly as the square of the number of octree nodes since the hidden line removal subroutine searches through the linked list of graphics nodes for each node in the list, looking for overlapping nodes. But the overwhelming factor determining the execution time is not the search time but the time to compute intersections and remove hidden lines from overlapping graphics nodes. Hence, 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.

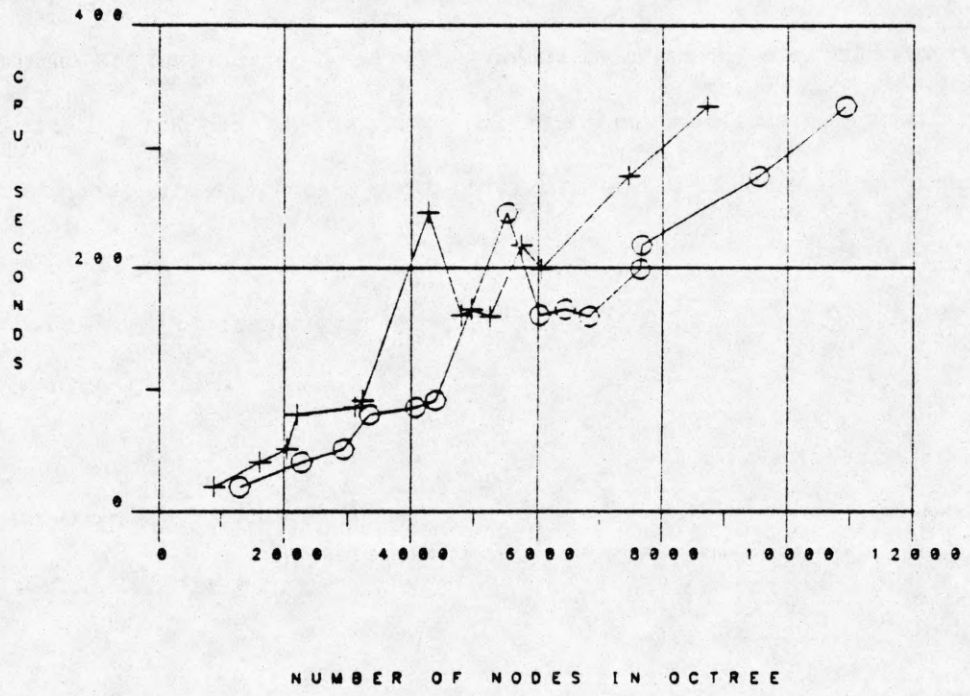


Figure 39

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.

5. SUMMARY

We have presented an algorithm to generate the octree representation of an object from silhouette images taken from a set of thirteen viewing directions. These viewing directions are parallel to three orthogonal faces, six face-diagonals, and four long-diagonals of an upright cube. Each silhouette of an object is first extended into a cylinder parallel to the viewing direction, and the corresponding octree is constructed. An intersection is performed on the octrees generated from the silhouettes to obtain an octree representing the space occupied by the object. The octree for each silhouette image is computed efficiently by a recursive quadtree decomposition of the image, and identification of the occupied octree nodes from a table listing corresponding pairs of image windows and octree nodes. In actual applications, the requirements of the 13 images may be met very simply by placing cameras in fixed positions in a cubical room, namely, at centers of walls, edges and corners, all pointing at the room center and taking orthographic images. We have also run performance tests on the accuracy of the octree and concluded that thirteen silhouette views can provide enough information for a good approximation of the object. The three sets of viewing directions (face, edge, and corner) act as coarse-to-fine, information acquisition probes. Fewer than 13 directions may be used to reduce computation time in exchange for reduced accuracy of the representation generated.

Although the general view algorithm allows an arbitrary viewpoint, this generality requires an explicit computation of the volume of intersection for determining the octree nodes corresponding to extended silhouettes. The corresponding intersection tests are more complex and may require greater computation time than the direct construction of octree nodes from image pixels used in our approach. Moreover, since silhouette images taken from viewing directions which are widely spaced yield more information, in general, than do silhouette images which are close together, and since the thirteen viewing directions used in our algorithm are distributed widely about the entire octree space, it is unlikely that a large amount of additional information will be obtained by using a silhouette taken from a viewpoint which falls at an

intermediate position. The results of our experiments bear out this expectation, where the accuracy for the thirteen views used in our algorithm is over 90% (with no silhouette preprocessing). Thus, there may be only a marginal gain in accuracy by using the general view algorithm, especially considering the inherent limitations discussed earlier of any shape-from-silhouette reconstruction algorithm. Given that the octree representation is useful only as a coarse occupancy map (for applications such as rough path planning), and is not intended as a representation of fine shape details, the accuracy provided by the thirteen viewing directions may suffice. This may be particularly important if the general-view algorithm turns out to be more expensive than our algorithm in processing silhouette views. We have not so far been able to obtain a copy of the general viewpoint algorithm of Shneier et al. [5, 12] so we are currently unable to compare the results of our algorithm with the general viewpoint algorithm. It remains to be determined how much more accuracy the general viewpoint algorithm can provide and what the cost of the additional accuracy is in terms of execution speed.

We have also developed a display algorithm to produce a line drawing of an object from its octree. The object is drawn in perspective with cracks and hidden lines removed. The line drawing produced may be used to check the performance of the octree generation algorithm by comparing the original and the represented object.

One of the inherent weaknesses of using silhouette images to obtain 3-D information is that surface concavities cannot be identified. A subject worthy of study is the use of range information, in addition to the silhouettes, to help identify these concavities. The range information could be obtained from such sources as sonar devices or laser range finders.

Several applications of this research suggest themselves. One possibility is the construction of a working system of thirteen cameras to monitor the objects in a room. The resulting octree representation of the occupied space in the room might be used to guide a robot. Another possibility is to point cameras at a spot above a conveyor belt and as objects pass through the

octree space defined by the camera viewing directions, perform some task based on the volume or shape of the octree approximated objects. If cameras are a scarce or precious resource, one could experiment with using multiple or movable mirrors (in conjunction with a single camera) to obtain the silhouette images. Finally, one could experiment with mounting a camera on a moving vehicle to obtain sequential images of an object or an environment for the purpose of maintaining a representation of the workspace, planning paths, and locating objects.

REFERENCES

1. N. Ahuja and C. Nash, Octree representations of moving objects, *Computer Vision, Graphics, and Image Processing, 26*, 1984, 207-216.

2. R. Brooks, Symbolic reasoning among models and 2-D images, *Artificial Intelligence, 17*, (1981) 285-348.

3. C. H. Chien and J. K. Aggarwal, A volume/surface octree representation, *Seventh International Conference on Pattern Recognition*, July 30 - August 2, 1984.

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

5. T. H. Hong and M. Shneier, Describing a robot's workspace using a sequence of views from a moving camera, unpublished manuscript, National Bureau of Standards.

6. C. L. Jackins and S. L. Tanimoto, Oct-trees and their use in representing three-dimensional objects, *Computer Graphics and Image Processing, 14*, 1980, 249-270.

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

8. D. Meagher, Efficient Synthetic Image Generation of Arbitrary 3-D Objects, *Proc. IEEE Conf. on Pattern Recognition and Image Processing*, Las Vegas, Nevada, June 14-17, 1982, p. 473.

9. D. Meagher, Geometric Modeling Using Octree Encoding, *Computer Graphics and Image Processing*, vol. 19, 1982, p.129.

10. W. Newman and R. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, 1979.

11. W. Osse and N. Ahuja, Efficient octree representation of moving objects, *Proc. 7th Int. Conf. on Pattern Recognition*, Montreal, Canada, July 30 - August 2, 1984, 821-823.

12. M. Shneier, E. Kent, and P. Mansbach, Representing workspace and model knowledge for a robot with mobile sensors, *Proc. Seventh Int. Conf. on Pattern Recognition*, Montreal, Canada, July, 1984, 199-202.

13. J. Veenstra and N. Ahuja, Octree Generation of an Object from Silhouette Views, *1985 IEEE Int. Conf. on Robotics and Automation*, St. Louis, Missouri, March 25-28, 1985, pp. 843-848.