CAR–TR–248
CS–TR–1752

DCR–86–05557
January 1987

HIERARCHICAL DATA STRUCTURES AND ALGORITHMS FOR COMPUTER GRAPHICS

Hanan Samet
Computer Science Department
University of Maryland
College Park, Maryland 20742

Robert E. Webber
Computer Science Department
Rutgers University, Busch Campus
New Brunswick, New Jersey 08903

ABSTRACT

An overview is presented of the use of hierarchical data structures and algorithms in computer graphics. These methods have found many applications in image rendering and solid modeling. While such data structures are not necessary for the processing of simple scenes, they are central to the efficient processing of large-scale realistic scenes. Although object-space hierarchies are discussed briefly, the main emphasis is on hierarchies constructed in the image space such as quadtrees and octrees.

The support of the National Science Foundation under Grant DCR–86–05557 is gratefully acknowledged.

webber d3 00 688

1. INTRODUCTION

Computer graphics applications require the manipulation of two distinct data formats: vector and raster (see Figure 1). The raster format enables the modeling of a graphics image as a collection of square cells of uniform size (called pixels). A color is associated with each pixel. In order to attain a maximum degree of flexibility, an attempt is made to model directly the addressability of the phosphors on the display screen so that each pixel corresponds to a phosphor. This format has also proven useful in computer vision since it corresponds to the digitized output of a tv camera. In contrast, instead of modeling the display screen directly, the vector format models the ideal geometric space that is to be represented on the display screen. Vector data consists of points, line segments, filled polygons, and polyhedral solids. In addition to processing these two formats of data directly, in computer graphics applications we are also concerned with the problem of conversion between these two formats. Closely related to the distinction between the raster format and the vector format is the distinction between image space and object space presented in an early classification of hidden-surface algorithms [Suth74].

Both data formats have obvious representations. These representations are minimal in the sense of just providing sufficient structure to allow updating. For the raster format, the obvious representation is as a two-dimensional array of color values. As an example, in Figure 1b, all elements of the array through which a line passes or that contain a point (i.e., shown shaded) are BLACK. For the vector format, the obvious representation is as a linked list of line segments (see Figure 2). Early work on the vector format extended the structure of this list by ordering the line segments around common vertices. For example, consider the winged-edge polyhedra representation [Baum72] as illustrated in Figure 3. While these representations are suitable for medium range applications, once the scene being modeled becomes significantly larger than the display grid, major logistic problems arise. There are two approaches [Suth74] to handle the logistics problems. One approach, based on object-space hierarchies [Clar76], is only discussed briefly in this paper. The other approach, based on image-space hierarchies, is typified by hierarchical data structures such as the quadtree and octree and is the subject of this paper.

In the remainder of this paper we review applications of hierarchical data structures such as the quadtree and octree in computer graphics. Section 2 contains a general discussion of their properties. Sections 3 and 4 describe algorithms using quadtrees and octrees respectively. Section 5 concludes with a brief discussion of some other applications of these data structures as well as hardware implementations. For more references and details on hierarchical data structures, see [Same84b, Same86b].

2. PROPERTIES OF QUADTREES AND OCTREES

In this section we discuss some fundamental properties of quadtrees and octrees. However, we first elaborate on the motivation for their development. As mentioned in Section 1, hierarchical data structures such as the quadtree and octree have their roots in attempts to overcome the problems that arise when the scene that is being modeled is more complex than the display grid (e.g., in size, precision, number of elements, etc.). These problems are solved by use of object-space hierarchies and image-space hierarchies, which are described in greater detail below. Next, we present a definition of the quadtree and octree, an examination of some of the more common ways in which they are implemented, and an explanation of the quadtree/octree complexity theorem. We conclude with a discussion of vector quadtree and vector octrees.

2.1. OBJECT-SPACE HIERARCHIES

Two kinds of logistic problems present themselves. The first problem is the communication between the user software and the graphics package, i.e., the number of procedure calls (or commands transmitted on a graphics channel) can become a bottleneck for the system. The second problem is in determining what subset of the scene is actually visible. For example, in a $512 \times 512 \times 512$ scene, only about 512×512 of it is actually visible at any given time. When the scene extends horizontally and vertically past the bounds of the viewing surface the problem is further aggravated. The first problem has been addressed, in part, by observing that the universe can be hierarchically organized into objects composed of subobjects which are in turn composed of other objects and so forth [Simo69]. This observation has been used as the basis for the organization of the user's interface to the data from the earliest graphics systems [Suth63, Gray67] to the most recent graphics package designs [Lant84, ANSI85].

Since the object hierarchy must be kept to solve the communication problem, it is tempting to use this hierarchy to solve the visible-subset problem. One way to adapt the object hierarchy to the visible-subset problem is through the notion of bounding objects. When determining whether or not an object is visible, it is common [Roge85] to surround the object (see Figure 4a) with a bounding box (see Figure 4b) or sphere (see Figure 4c). If the bounding object is not visible, then clearly the object being bounded is also not visible. This produces a major computational savings, since it is usually significantly easier to test for visibility of the bounding object than the visibility of the bounded object. However, this technique is not capable of dealing with the visible-subset problem when the number of objects is large. It has been noted [Clar76, Rubi80, Wegh84] that the objects being bounded need not be limited to the primitive objects of the scene; instead, bounding objects can also be placed around the complex objects formed by the different levels of the object hierarchy (see Figures 4d and 4e).

While this approach is easy to implement and can significantly improve execution time, its efficiency is based on the notion that the object hierarchy is a structural approximation of a balanced binary tree in the sense that objects in the hierarchy are expected to be locatable in time roughly proportional to the logarithm of the number of objects in the hierarchy. Of course, this is often not the case. This is because there are two kinds of levels in a natural hierarchy: those formed by a few unique objects and those formed by a large number of nearly identical objects [Simo69]. Levels formed in this second manner can be very flat and of little computational benefit. Even when the branching factor is reasonable, there is no guarantee that the natural hierarchy will be balanced in the algorithmic sense (see Figure 4f). Some attempts have been made to artificially structure the object space to avoid these problems (e.g., [Fuch83]); however, such attempts have problems handling dynamically changing scenes (due to preprocessing costs) as well as often having unfortunate worst-case results.

A related artificial object hierarchy is the strip tree [Ball81] (see Figure 5). In this case, we are dealing with an object consisting of a single curve. The curve is surrounded by a bounding rectangle two of whose sides are parallel to the line joining the endpoints of the curve (e.g., the bounding rectangle A in Figure 5 has two sides parallel to the line between points P and Q). The curve is then partitioned in two at one of the locations where it touches the bounding rectangle. Each subcurve is then surrounded by a bounding rectangle and the partitioning process is applied recursively. This process stops when the width of each strip is less than a predetermined value. The strip tree is implemented as a binary tree where each node contains eight fields. Four fields contain the x and y coordinates of the endpoints, two fields contain pointers to the two sons of the node, and two fields contain information about the width of the strip (i.e., W_L and W_R in Figure 5). The worst-case situations illustrated by this data structure are typical of the problems with computations on object hierarchies.

2.2. IMAGE-SPACE HIERARCHIES

A natural alternative to processing graphics commands in the object-space hierarchy is to organize the data around an image-space hierarchy. One problem with traditional image-space representations (i.e., two and three dimensional arrays) is that they require the user to fix the maximum resolution in advance. However, a hierarchical organization of the image space allows the resolution to vary with the complexity of the objects in various regions. Of course there are many ways to partition the image space (when it is viewed as a continuous plane/space) [Ahuj83, Bell83, Gibs82], but to easily interface with a Cartesian coordinate system and with the typical display device controller, a decomposition of the plane into square regions (and a space into cubical regions) is simplest. Two examples of non-square partitionings of the plane are given in Figure 6. In the following we discuss the organization of a planar image space (leaving consideration of the three-dimensional image space for a later section).

When justifying the usage of object-space hierarchies for image-space processing, we often refer to the property of area coherence, which means that objects tend to represent compact regions in the image space. Similarly, we might speak of object coherence as being a factor in image-space hierarchies, since regions that are close to each other tend to be parts of the same object. Thus, both types of hierarchies tend to approximate each other. However, for large-scale applications, the costs associated with the imprecision of these approximations can easily overshadow any benefits accrued from the explicit maintenance of just one of the hierarchies. Thus, when possible both hierarchies should be maintained. A definitive analysis of the merits of image space and object space hierarchies awaits a universally accepted model of "typical graphic data".

2.3. QUADTREE/OCTREE DEFINITION

One commonly used two-dimensional image-space hierarchy is typified by the quadtree data structure [Klin71, Same84b]. It is constructed in the following manner. We start with an image (whose binary array representation is given in Figure 7a) and check to see if it has a simple description and thus does not require any further hierarchical structuring. If this is not the case, then the image space is partitioned into four disjoint congruent square regions (called quadrants) whose union covers the original image space (see Figure 7b). Each of these new image spaces is treated as if it was isolated, and for each one the question is raised as to whether or not it has a simple description (resulting in Figure 7c). Of course, in this example, the stopping rule for the decomposition process is homogeneity (i.e., each square region is of one color). This decomposition technique is referred to as a regular decomposition to distinguish it from decomposition approaches that vary the size of the subregions formed from the original regions (see Figure 7d). While it is plausible to attempt to move the boundaries of the subregions inorder to distribute the complexity of the image more evenly, it is not clear how to do this in an effective way. The inherent simplicity of regular decomposition facilitates both its implementation and the analysis of its performance.

The test for determining whether or not an image space has a simple description is called the leaf criterion. It is called the leaf criterion because the spaces that satisfy it form the leaf nodes of the tree that represents the hierarchical structure. There are many variants on the quadtree data structure that only differ in what constitutes a satisfactory leaf criterion for the data structure. This is useful because it allows the construction of integrated graphic databases that handle a wide variety of data in an analogous manner [Rose82, Rose83, Same84c, Same84d, Same85g].

There are many plausible leaf criteria. When looking for a leaf criterion, we are really looking for a subset of the possible image spaces where the graphics tasks we want to process can be solved easily. It is also necessary that any arbitrary image space can eventually be decomposed into regions that satisfy the criterion. Thus, for example, if we were to store the vector data in the image space, we might hypothesize a criterion that stipulated that at most one line segment could appear in any leaf. However this in itself would be unsatisfactory, because there are images (for example, any image containing a vertex where at least two line segments meet) that cannot in general be partitioned (in a finite amount of time) into square regions where no region contains more than one line segment.

Although the above criterion is inadequate as a pure vector representation, a slight modification of it has been used [Shne81, Ayal85, Same84d]. The modification is to establish a maximum quadtree depth. Once the maximum depth is reached in the construction process, if the criteria is still not satisfied, then the region is simply represented by a pixel. This yields a mixed raster and vector representation where some information about the image can be lost. This representation is known as the edge quadtree [Shne81]. For example, Figure 8 is the edge quadtree corresponding to the vector data of Figure 1a. In this case, truncation at the maximum tree depth (i.e., 4) has occurred at the nodes containing vertices A, B, C, D, E, F, and G but not H.

The octree [Hunt78, Jack80, Meag82, Redd78] data structure is the three-dimensional analog of the quadtree. It is constructed in the following manner. We start with an image in the form of a cubical volume and determine if its description is sufficiently complex, in which case the volume is recursively subdivided into eight congruent disjoint cubes (called octants) until the complexity is sufficiently reduced. Of course, the leaf criteria differ depending on whether the data is of a raster format (consisting of three-dimensional voxels having a single color instead of two-dimensional pixels) or vector format (consisting of solids and planar or curved surfaces instead of polygons and edges). Figure 9a is an example of a simple three-dimensional object whose raster octree block decomposition is given in Figure 9b and whose tree representation is given in Figure 9c.

For the purposes of this paper, we will consider quadtrees and octrees constructed from two different leaf criteria (one for handling raster data and the other for handling vector data). For raster data, we will use the quadtree/octree built from the criterion that no space can contain data having more than one color. This works for raster data because the raster grid is built of singly-colored regions, and hence the hierarchy need never decompose to a level lower than that of these pixels. This structure has many interesting mathematical properties, some of which are reviewed briefly below after the discussion of implementation techniques.

2.4. QUADTREE/OCTREE DATA STRUCTURE IMPLEMENTATIONS

Besides consideration of the leaf criteria, the investigation of hierarchical data structures has also been concerned with how to encode the tree representing the hierarchy. In his treatise on data structures, Knuth [Knut75] mentions three general approaches to representing trees. Each of these approaches has been investigated by others with regards to the specific representation of quadtrees. In the following, we describe these three approaches and discuss their relative advantages and disadvantages. Our discussion is in the context of a quadtree; however, the extension to octrees is straightforward.

The first and most obvious quadtree encoding is as a tree structure that uses pointers. Figure 10 is the tree/pointer representation of the quadtree of Figure 7. Each internal node (often referred to as a GRAY node) requires four pointers (one for each of its subtrees). Clearly the leaf nodes do not need pointer fields. The size of a pointer field is the base 2 logarithm of the number of nodes in the tree. Each node also requires one bit of information to distinguish whether it is an internal node or a leaf. In order to describe quadtree algorithms, it is useful for each node to contain a father link; however, this is not necessary from an implementation viewpoint because in most tasks processing starts at the root and a stack of father links can be easily maintained. Pointers have also been proposed to connect nodes that represent neighboring regions [Hunt78, Hunt79a] but these are not necessary for the efficient processing of the quadtree. Most of the early implementations of quadtrees used the pointer approach while the next two approaches were considered later due to a perceived storage inefficiency of the pointer approach. However, the literature is often unclear as to exactly how the quadtree algorithms are coded.

The second approach makes use of the observation that the number of subtrees of a given node in the quadtree node is either four or zero. This makes it possible to represent a quadtree by listing the nodes encountered by a preorder traversal of the tree structure. For example, traversing the quadtree of Figure 10 in the order NW, NE, SW, and SE and letting G, B, and W denote non-terminal, solid, and empty nodes, respectively, results in the list GWGWWBBGWBWBB. It requires exactly one bit of overhead per node, which is used to distinguish between leaf nodes and internal nodes. Many simple algorithms, e.g., intersection/union and area calculation, are performed by preorder traversals of the quadtree and they can be efficiently implemented with this encoding. However, other algorithms can not be efficiently implemented by this encoding. For example, in order to visit the second subtree of a node, it is necessary to visit each node of the first subtree so that the location of the root of the second subtree can be determined. Nevertheless, this encoding is usable for some applications, e.g., archiving and facsimile transmission. Algorithms specific to this representation have been investigated by Kawaguchi *et al.* [Kawa80a, Kawa80b, Kawa83] who call it a *DF-expression* (because of the similarity between a preorder traversal and a depth-first expansion of the tree), and Oliver and Wiseman [Oliv83a, Oliv83b] who refer to it as a *treecode*.

The third approach is based on the use of locational codes (referred to as a Dewey-decimal encoding by Knuth [Knut75]). It was first proposed by Morton [Mort66] as an index to a geographical database. In the variant that we describe, each node is represented by a pair of numbers. The first number, termed a *locational code*, is composed of a concatenation of base 4 digits corresponding to directional codes that locate the node along a path from the root of the quadtree. The directional codes take on the values 0, 1, 2, 3 corresponding to quadrants NW, NE, SW, SE, respectively. The second number is the level of the tree at which the node is located. Assume that the root is at level 0. For example, the pair of numbers (312,3) are decoded as follows. 312 is the base 4 locational code and denotes a node at level 3 that is reached by a sequence of transitions, SE, NE, and SW, starting at the root. The overhead per node is 2 bits per level of depth of the node, plus the base 2 logarithm of the depth of the node in order to specify the level at which the node is found. Algorithms specific to this representation have been investigated by Gargantini *et al.* [Garg82a, Garg82b, Garg82c, Garg83] who call this representation a *linear quadtree* (because the addresses are keys in a linear list of nodes), and Oliver and Wiseman [Oliv83a, Oliv83b] who refer to it as a *leafcode*.

When using the linear quadtree encoding, it is possible to further reduce the storage requirements without substantially increasing the runtime requirements of the algorithms. In particular, there is no need to retain the internal nodes as the general quadtree structure only stores data in the tree's leaf nodes. Since the number of internal nodes is equal to one third of the number of leaf nodes minus one, this results in a significant space savings. Moreover, it is often remarked that the empty nodes (or nodes representing a background color) can also be eliminated from the node list. While it is correct that this does not excessively complicate the processing of the quadtree, it is unclear how useful it is. In the case of a binary raster image, the result is a reduction in the size of the quadtree to one half of its former size (assuming that, on the average, one half of the pixels are background). However, for multicolored raster data, the notion of a background color becomes less relevant and this compaction becomes, in turn, less useful. This approach can also be applied to vector data quadtrees. A related method draws an analogy to run encoding [Ruto68] where the locational codes of the leaf nodes are sorted and only the first element of each subsequence of blocks of the same color is retained [Lauz85]. This method cannot be easily applied to vector quadtrees.

It is interesting to compare the overhead of the pointer and linear quadtree encodings. By overhead of an encoding, we mean the size of that portion of the encoding that is used to represent the quadtree structure. We ignore the data fields that are stored at the leaf nodes, since they would be the same in either encoding. Our comparison is in the context of a static collection of nodes. In particular, we determine the maximum number of quadtree nodes that can be stored in a fixed amount of storage for an image of a given maximum leaf depth. This is achieved by computing the cutoff node count value that indicates when a pointer quadtree requires less space than a linear quadtree.

A quadtree internal node has 4 pointers whereas an external node does not require pointers. We distinguish between the two node types by using a bit that is stored in the pointer field of the node that points at the node being described, instead of in the node being described. Expressing the overhead of the pointer encoding as a function of the number of leaf nodes, we find that it is four thirds times the number of leaf nodes (i.e., four pointers per internal node) times the sum of 1 (the leaf flag bit) and the base 2 logarithm of four thirds the number of leaf nodes (i.e., the pointer size). On the other hand, the linear quadtree encoding requires a number of bits equal to the number of leaf nodes times the sum of twice the maximum leaf depth and the base 2 logarithm of the maximum leaf depth. If L is the number of leaf nodes and n is the maximum leaf depth, then the linear quadtree is more compact than the pointer quadtree when:

$$L\cdot(2\cdot n+\log_2(n)) < (4/3)\cdot L\cdot(1+\log_2((4/3)\cdot L))$$

or

$$2\cdot n+\log_2(n) < (4/3)\cdot(1+\log_2(4/3)+\log_2(L))$$

or

$$2^{(3/4)\cdot(2\cdot n+\log_2(n))-(1+\log_2(4/3))} < L$$

Table 1 shows the evaluation of the above formula for various values of depth n so that we can see where the tradeoff occurs. For example, a quadtree of depth 9 must contain at least 22,574 leaf nodes in order for the linear quadtree to be more compact than the corresponding pointer quadtree. Since the maximum number of leaf nodes at depth 9 is 262,144 (i.e., 4^9), we see that this means that the number of leaf nodes must be at least 8.6% of the maximum (i.e., the number of leaf nodes in a complete quadtree) in order for the linear quadtree to be more compact for images of depth 9. To put this in perspective, consider Figure 11 which is a map of a floodplain from a geographic

Table 1. Tradeoff leaf node count.

depth	tradeoff leaf node count	maximum number of leaf nodes
3	19	64
4	68	256
5	227	1024
6	736	4096
7	2337	16384
8	7306	65536
9	22574	262144
10	69100	1048576
11	209928	4194304
12	633807	16777220
13	1903591	67108860
14	5691899	268435500
15	16954100	1073742000

database [Same84d]. The number of leaf nodes in the quadtree of this map is approximately 2% of the maximum number of leaf nodes in a quadtree of depth 9. It would be more compactly represented by a pointer quadtree.

An alternative interpretation of this result is that the greater the compactness of the quadtree, the smaller the overhead of the pointer quadtree versus the linear quadtree. Of course, for an actual implementation, the above analysis must be modified to take into account the fact that the fields that correspond to the pointer, level, depth, and node type must lie on bit boundaries. Moreover, the entire encoding of a node must be further restricted to lie on a byte boundary. However, the interpretation remains the same.

For octrees, and higher dimensional data, the comparison is performed in the same manner. The difference is that now there are 8 pointers (or 2^d for d-dimensional data) for each internal node. Using the same notation - i.e., L is the number of leaf nodes and n is the maximum leaf depth, then the linear octree is more compact than the pointer octree when:

$$2^{(7/8)\cdot(3\cdot n+\log_2(n))-(1+\log_2(8/7))} < L$$

Interestingly, for higher dimensional data (e.g., three dimensions and above), the cutoffs are much closer to the maximum node counts which means that in all practical cases of such data, the pointer structure (e.g., pointer octree) will require less space than the linear structure (e.g., linear octree). For more details, see [Same86d].

The amount of storage required by quadtrees and octrees is directly proportional to the number of leaf nodes. One approach to reducing the number of leaf nodes in these data structures is the bintree [Know80, Tamm84a, Same85b]. In the following, we illustrate its application to the octree. The bintree is constructed by first dividing the cube into two half cubes (with respect to a plane parallel to the $y-z$ plane), then, if necessary, the half cubes into quarter cubes (with respect to a plane parallel to the $x-z$ plane), and finally if it is still necessary to subdivide the region, then it can be split further into cubes (by planes parallel to the $x-y$ plane). For example, see Figures 12a and 12b which are the block decomposition and tree representation, respectively, for the three-dimensional bintree corresponding to the octree of Figure 9 when using the octree coordinate system of Figure 12c. Comparing the pointer representations of bintrees and octrees, we find that the relative number of internal nodes has increased from 1/7 of the number of leaf nodes in the octree to one less than the number of leaf nodes in the bintree. However, the number of leaf nodes in the bintree is bounded from below by one fourth the number of leaf nodes in the octree and from above by the number of leaf nodes in the octree. Note that both bounds are attainable, and thus it is not always the case that the total number of nodes in the bintree is less than or equal to the total number of nodes in the octree. However, in the case of a linear bintree representation the extra internal nodes become irrelevant and the additional two bits required to distinguish the final level of the bintree is often overshadowed by the reduction in the number of leaf nodes. Another advantage of the bintree is that algorithms using it work for data of arbitrary dimensionality.

2.5. THE QUADTREE/OCTREE COMPLEXITY THEOREM

Most quadtree algorithms are simply preorder traversals of the quadtree and hence their execution time is generally a linear function of the number of nodes in the quadtree. Thus we are interested in the asymptotic analysis of the size of a quadtree more from the standpoint of its relevance to the execution-time analysis of quadtree algorithms than from the standpoint of the amount of storage that is actually required. Our discussion assumes a tree representation in the sense that the number of nodes in the quadtree includes the internal nodes. A key to the analysis of the execution time of quadtree algorithms is the following result on the size of quadtrees (henceforth referred to as the Quadtree Complexity Theorem [Hunt78, Hunt79a]), which states that:

> For a quadtree of depth q representing an image space of $2^q \times 2^q$ pixels where these pixels represent a region whose perimeter measured in pixel-widths is p, then the number of nodes in the quadtree cannot exceed $16 \cdot p - 11 + 16 \cdot q$.

Since under all but the most pathological cases (e.g., Figure 13), the region perimeter exceeds the base 2 logarithm of the width of the image space in which the region is presented, the Quadtree Complexity Theorem means that the size of the quadtree representation of a region is linear in the perimeter of the region. An alternative interpretation of this result is that for a given image, if the resolution doubles and hence the perimeter doubles (ignoring fractal effects), then the number of nodes will double. On the other hand, for the two-dimensional array representation, when the resolution doubles, the size of the array quadruples. Therefore, asymptotically, quadtrees are arbitrarily more compact than two-dimensional arrays; however, for moderate size applications, constant factors need to be scrutinized more carefully. Figure 14 illustrates the relative growth of the two representations for a simple triangular region.

In most tree structures, the number of nodes in the tree is dominated by the number of nodes at the deepest levels (assuming that the root is at the top). This is also true for quadtrees (e.g., Figure 10). The Quadtree Complexity Theorem follows from the realization that all nodes in the quadtree are either adjacent (including diagonal adjacencies) to the border between two regions or have a sibling with a subtree that contains a portion of the border. Thus at the deeper levels of a quadtree, the only nodes present are those that are very close to the border. From elementary geometry we know that the number of disjoint regions of a bounded size that can be within a bounded region of the perimeter is a linear function of the length of the perimeter. Although we might expect a typical image to have a lower constant of proportionality than the 16 of the Quadtree Complexity Theorem, we should expect it to have a size that is linear in its perimeter. In fact, Dyer [Dyer82] has shown that if square regions with sides that are powers of 2 in length, say 2^m, are randomly placed in the image space, then the expected size of the quadtree is linear in the length of the perimeter of the square. For example, the worst-case for a $2^q \times 2^q$ image requires $4 \cdot p + 16 \cdot (q - m) - 27$ nodes.

The Quadtree Complexity Theorem holds for three-dimensional data [Meag80] where perimeter is replaced by surface area, as well as higher dimensions for which, in all but pathological cases, it means that

> The size of the k-dimensional quadtree of a set of k-dimensional objects is proportional to the sum of the resolution and the size of the $(k-1)$-dimensional interfaces between these objects.

Aside from its implications on the storage requirements, the Quadtree Complexity

Theorem also directly impacts the analysis of the execution time of algorithms. In particular, most algorithms that execute on a quadtree representation of an image instead of an array representation have an execution time that is proportional to the number of blocks in the image rather than the number of pixels. In its most general case, this means that the application of a quadtree algorithm to a problem in d-dimensional space executes in time proportional to the analogous array-based algorithm in the $(d-1)$-dimensional space of the surface of the original d-dimensional image.

2.6. VECTOR QUADTREE DEFINITION

The other type of data that we want to represent is vector data. There are a number of useful leaf criteria [Same85c] for representing vector data using quadtrees. These criteria differ in the degree of the complexity of the image-space description versus the size of hierarchy (i.e., the number of nodes in the quadtree). Choosing between the criteria is a matter of analyzing constants on specific machines to determine whether we prefer a large number of simple leaf nodes or a smaller number of more complicated leaf nodes (where it is understood that the expense of processing a leaf is proportional to the complexity of the information stored in the leaf). In the following, we present a leaf criterion that results in many simple leaf nodes, but which minimizes the complexity of the description of algorithms; however, other leaf criteria may prove more useful for specific implementations. The criteria that we shall use for vector data is is termed a *PM_1 quadtree* [Same85c] and is defined as follows:

(1) There can be at most one vertex in an image space.
(2) If there is a vertex in the image space, then all line segments in the image space must share that vertex.
(3) If there are no vertices in the image space, then there can be at most one line segment passing through the image space.

For our purposes, vertices occur at the endpoints of line segments and at any location where two line segments intersect. A line segment consists of a set of q-edges where a *q-edge* is the maximal portion of a line segment that is contained within a given image space. Using such criteria, the image of Figure 1a is represented by the quadtree of Figure 15.

When a line segment passes through an image space, resulting in a q-edge, only its presence in the space is explicitly recorded [Nels86]. The intercepts of the q-edge with the border of the image space can be derived from the descriptor of the line segment that is associated with the q-edge. Thus all q-edges are specified with the same precision as the vertices of their corresponding line segments. The descriptor of the line segment is retained as long as at least one of its q-edges is still present. Thus fragments of line segments can be represented. This is important for it means that the representation is consistent - i.e., removal of a q-edge from an image space and its subsequent reinsertion into the same image space will result in the same line segment.

The Quadtree Complexity Theorem is also applicable to vector data. In this case, a suitable pixel width would be the size of the deepest leaf node needed to represent the structure. This maximum depth is a function of the closest approach between vertices and line segments that are not adjacent to the vertices. Alternatively, an upper bound on the depth can be constructed based on the precision with which the location of the vertices is specified [Same86a]. In either case, the bound on the number of nodes given by the Quadtree Complexity Theorem is excessively pessimistic for vector data.

It would be nice if the number of nodes of the quadtree was a function of the number of line segments in the image space (thus making the size of the image-space hierarchy comparable to the size of the object-space hierarchy for the same data). However, this is not the case because as the image space is subdivided, line segments are also subdivided. Thus information about a given line segment can exist in many nodes of the structure. In the worst case, the number of nodes in which information about a particular line segment can occur is proportional to the length of that line segment. This worst case is the one that is analyzed by the above adaptation of the Quadtree Complexity Theorem. However, this is not typical. In fact, it is reasonable to expect that the smallest leaf nodes that contain a given line segment occur near the endpoints of the line segment. Furthermore, as we examine parts of the line segment that are successively further from both endpoints, the size of the leaf nodes containing these parts of the line segment get larger. In other words, we expect the number of nodes contributed by a given line segment to be proportional to the base 2 logarithm of the length of the line segment.

2.7. VECTOR OCTREE

Just as the raster quadtree leaf criterion could be generalized to a raster octree leaf criterion, the vector quadtree leaf criteria can also be generalized to form vector octree leaf criteria to represent polyhedra. Octree data structures have been used where the octree decomposition was performed as long as the number of primitives in a leaf node exceeded a predefined bound [Glas84, Wyvi85, Jans86]. This approach has also been used in the context of the bintree representation of the octree [Kapl85]. However, it has the same problems as the analogous quadtree approach, i.e., there are some features that cannot be represented exactly by this approach (thus requiring a maximum depth truncation similar to the edge quadtree [Shne81, Ayal85, Same84d]). One way to avoid the information loss from a maximum-depth cutoff, is to permit a variable number of primitives to be associated with each octree leaf node. The vector octree analog [Ayal85, Carl85, Fuji85] of the vector quadtree consists of leaf nodes of type face, edge, and vertex, defined as follows. A face node is an octree leaf node that is intersected by exactly one face of the polyhedron. An edge node is an octree leaf node that is intersected by exactly one edge of the polyhedron. For our purposes, it is permissible to have more than two faces meet at a common edge. However, such a situation cannot arise when modeling solids with Eulerian operators [Baum72]. Nevertheless, it is plausible when three-dimensional objects are represented by their surfaces. A vertex node is an octree leaf node that is intersected by exactly one vertex of the polyhedron. The space requirements of the vector octree are considerably harder to analyze than those of the raster octree [Nava86a]. However, it should be clear that the vector octree for a given image is much more compact than the corresponding raster octree. For example, Figure 16b is a vector octree decomposition of the object in Figure 16a.

Vector octree techniques have also been extended to handle curvilinear surfaces. Primitives including cylinders and spheres have been used in conjunction with a decomposition rule that limits the number of distinct primitives that can be associated with a leaf node [Fuji86a, Wyvi85]. Another approach [Nava86b] extends the concepts of face node, edge node, and vertex node to handle faces represented by biquadratic patches. The use of biquadratic patches enables a better fit with fewer primitives than can be obtained with polygonal faces, thus reducing the size of the octree. The difficulty in organizing curved surface patches by using octrees lies in devising efficient methods of calculating the intersection between a patch and an octree node. Observe that in this approach we are organizing a collection of patches in the image space, in contrast to decomposing a single patch in the parametric space by use of quadtree techniques as discussed in Section 3.8.4.

3. ALGORITHMS USING QUADTREES

In this section we describe how a number of basic graphics algorithms can be implemented using quadtrees. In particular, we discuss point location, object location, set operations, image transformations, scaling, transmission, quadtree construction, polygon coloring, display, and hidden surface algorithms. We also expand on the concept of neighbor finding which serves as a basis for many algorithms using quadtrees and octrees.

3.1. POINT LOCATION

Probably the simplest task to perform on raster data is to determine the color of a given pixel. In the traditional raster representation, this is achieved by exactly one array access. In the raster quadtree, this requires searching the quadtree structure. The algorithm starts at the root of the quadtree and uses the values of the x and y coordinates of the center of its block to determine which of the four subtrees contains the pixel. For example, if both the x and y coordinates of the pixel are less than the x and y coordinates of the center of the root's block, then the pixel belongs in the southwest subtree of the root. This process is performed recursively until a leaf is reached. It requires the transmission of parameters so that the center of the block corresponding to the root of the subtree currently being processed can be calculated. The color of that leaf is the color of the pixel. The execution time for the algorithm is proportional to the level of the leaf node containing the desired pixel.

Point location can also be performed without explicitly calculating the center of the block corresponding to each node encountered along the path. This calculation can be avoided by making use of the depth n of the pixel relative to that of the root and assuming that the southwestern-most pixel is at (0,0). This approach to pixel location is easiest to contemplate with respect to a quadtree representation that makes use of locational codes, although it is equally applicable to the pointer representation of quadtrees. The locational code for a leaf is formed by a process (described in Section 2.4) that is equivalent to interleaving the binary coordinates of the lower lefthand corner of the leaf. Here, coordinates are integer values ranging from 0 to 2^n-1 for a $2^n \times 2^n$ grid. When the leaf nodes are sorted by their locational codes (as is the case for a preorder traversal of the quadtree), the addresses of all descendants of a node, say P, lie between the address of P and the address of its immediate successor at the same level. Locating a pixel is done by first interleaving the binary representations of its coordinates to construct an address, say K, for a hypothetical leaf node corresponding to the pixel. This hypothetical leaf is located by performing a binary search on the sorted list of locational codes for the leaf nodes of the quadtree and returning the leaf node with the largest locational code value that is less than or equal to K. The execution time for the algorithm is proportional to the log of the number of leaf nodes in the tree (assuming key comparisons can be made in constant time). When a pointer representation is used, the pixel location algorithm is slightly different. In particular, we locate the appropriate leaf by descending the tree. The execution time is proportional to the level of the leaf node containing the desired pixel.

3.2. NEIGHBORING OBJECT LOCATION

The vector analog of the pixel-location task is the object-location operation where the x and y coordinates of the location of a pointing device (e.g., mouse, tablet, lightpen) must be translated into the name of the appropriate object. In order to handle this task, we must first determine the leaf that contains the indicated location. The first approach discussed in Section 3.1 can be adapted in a straightforward manner. The second approach, using the interleaved bits, is not immediately applicable since there is no underlying pixel level. Let us assume that the block corresponding to the root of the quadtree is the unit square and let us represent the values of the x and y coordinates of the pointing device as fixed length binary fractions. Now the bits of the binary fraction can also be viewed as representing the unsigned integer coordinates of a grid where the separation between neighboring grid points is the minimum resolution of the binary fraction. The equivalence to integer coordinates is straight-forward.

For vector data quadtrees, the leaf corresponding to the location of the pointing device serves as the starting point of the object-location algorithm. In essence, we wish to report the nearest primitive of the object description stored in the quadtree. If the leaf is empty, then we must investigate other leaf nodes. In fact, even if the leaf node is not empty, unless the location of the pointing device coincides with a primitive, it is possible that a nearer primitive might exist in another leaf. Such an algorithm has been developed for quadtree representations that use locational codes [Abel84b] as well as pointers [Ande83]. The latter is reported only for the case of point data; however, the treatment of vector data differs from point data only in the form of the formula used to calculate the distance from a point.

Using a pointer quadtree representation, finding the nearest primitive (also known as the *nearest neighbor problem*) is achieved by a top-down recursive algorithm. Initially, at each level of the recursion, we explore the subtree that contains the location of the pointing device, say P. Once the leaf containing P has been found, the distance from P to the nearest primitive in the leaf is calculated (empty leaf nodes have a value of infinity). Next, we unwind the recursion and, as we do so, at each level we search the subtrees that represent regions that overlap a circle centered at P whose radius is the distance to the closest primitive that has been found so far. When more than one subtree must be searched, the subtrees representing regions nearer to P are searched before the subtrees that are further away (since it is possible that a primitive in them might make it unnecessary to search the subtrees that are further away). For example, consider Figure 17 and the task of finding the nearest neighbor of P in node 1. If we visit nodes in the order NW, NE, SW, SE, then as we unwind for the first time, we visit nodes 2 and 3 and the subtrees of the eastern brother of 1. Once we visit node 4, there is no need to visit node 5 since node 4 contained A. However, we still visit node 6 which contains point B which is closer than A, but now there is no need to visit node 7. Unwinding one more level finds that due to the distance between P and B, there is no need to visit nodes 8, 9, 10, 11, and 12. However, node 13 must be visited as it could contain a point that is closer to P than B.

Sometimes it is not necessary to calculate the nearest neighbor as long as a "close" neighbor is found. For example, in a plotting application it is desired to minimize the amount of wasted pen motions (i.e., motions of the pen that do not involve drawing). In particular, we require a realtime algorithm in the sense that we want to minimize the total time required to both preprocess the drawing and to actually plot it. In such a case, it has been found useful to use a quadtree heuristic for calculating the nearest neighbor [Ande83]. For example, in Figure 17 such a heuristic might return A as the nearest neighbor of P even though B is closer. In this application, the only relevant data are the endpoints of the line segments. The heuristic is to use the primitive in the leaf containing the location of the pointing device (unless that leaf is empty in which case one of the neighboring nonempty leaf nodes is used).

3.3. SET-THEORETIC OPERATIONS AND IMAGE TRANSFORMATIONS

The basic set-theoretic operations on quadtrees were first described by Hunter and Steiglitz [Hunt78, Hunt79b] for pointer-based quadtrees. Gargantini [Garg83] and van Lierop [vanL84] later investigated these operations for linear quadtrees. Gargantini [Garg83] raises the issue of performing these operations on quadtrees that are not aligned. Hunter and Steiglitz [Hunt78, Hunt79b] and Peters [Pete85] consider the problem of performing an arbitrary linear transformation on an object represented by a quadtree. In this section we show how to perform set-theoretic operations on both aligned and unaligned quadtrees. We conclude with a demonstration that linear transformations on a quadtree are special cases of set-theoretic operations applied to quadtrees that are not aligned.

3.3.1. ALIGNED QUADTREES

Two quadtrees are said to be *aligned* when their root nodes correspond to the same region. Set-theoretic operations on aligned quadtrees are generally simpler than the equivalent operation on unaligned quadtrees. Of course, the complement operation, which is a unary operation, is trivially an aligned quadtree algorithm (since every quadtree is aligned with itself). The complement operation only makes sense as a set-theoretic operation when the quadtree in question represents a binary image (i.e., leaf nodes are either black or white). In the more general case of a quadtree with multicolored leaf nodes, the analogous operation is to uniformly replace the color of each of the leaf nodes by another color. When the color-to-color mapping is specified by an array indexed by the first color, then the cost of the transformation is simply the cost of visiting each node of the quadtree and creating a copy with the appropriate new data. Assuming that the color mapping is one-to-one and onto, then the quadtree's structure does not change. The simplest way to do this is to traverse the input quadtree in preorder, simultaneously building the resultant quadtree. If the mapping between colors is not invertible, then it may be necessary to merge some nodes in the resulting quadtree. However, this can be done naturally during the preorder traversal of the input tree. Thus, under either condition, the quadtree recoloring algorithm executes in time proportional to the size of the input quadtree.

A variant of the recoloring algorithm is the dithering (or halftoning) algorithm [Javi76]. The dithering task requires us to convert an image whose pixels are colored with varying intensities of gray into an image whose pixels are either black or white (while maintaining as much similarity to the original image as possible). A simple solution is to associate the colors black or white with each gray value. Better results can be achieved by varying the threshold that differentiates the black gray values from the white gray values in a pseudo-random manner with respect to the location of the pixel in the image. The traditional algorithm that is used to distribute thresholds across the image works particularly well with quadtrees because it is based on a matrix of threshold values that is a square matrix whose width is a power of two. This matrix is defined in terms of its four square submatrices as follows:

$$D_0 = 0 \qquad D_n = \begin{bmatrix} D_{n-1} & D_{n-1} + 2/2^{2n} \\ D_{n-1} + 3/2^{2n} & D_{n-1} + 1/2^{2n} \end{bmatrix}$$

In the above formulation we show a scalar term being added to a matrix. In this case, the scalar term is added to each element of the appropriate matrix.

The recursive definition of the dithering matrix D means that the dithering values can be computed as the block is recursively decomposed into its subblocks. Once a pixel sized block is encountered, it is thresholded with the appropriate dither value. The dynamic computation of the dithering values takes approximately the same amount of time as that required to access a precomputed dither matrix hence making the precomputation unnecessary. This means that we can use the largest possible dither matrix - i.e., D_n for a quadtree that corresponds to a $2^n \times 2^n$ image. See [Javi76] for a discussion of the significance of using dithering matrices of varying sizes.

For a binary image, set-theoretic operations such as union and intersection are quite simple to implement. For example, the intersection of two quadtrees yields a black node only when the corresponding regions in both quadtrees are black. Figure 18c is the result of the intersection of the quadtrees of Figures 18a and 18b. This operation is performed by simultaneously traversing three quadtrees. The first two trees correspond to the trees being intersected and the third tree represents the result of the operation. At each step in the traversal one of the following actions is taken:

(1) If either input quadtree node is white, then the output quadtree node is white.

(2) If both input quadtree nodes are black, then the output quadtree node is black.

(3) If one input quadtree node is black and the other input quadtree node is gray (i.e., an internal node), then the gray node's subtree is copied into the output quadtree.

(4) If both input quadtree nodes are gray, then the output quadtree node is gray, and these four actions are recursively applied to each pair of corresponding sons. Once the sons have been processed, we must check to see if they are all leaf nodes of the same color in which case a merge takes place (e.g., the sons of nodes B and E in Figures 18a and 18b respectively). Note that for the intersection operation, a merge of four black leaf nodes is impossible and thus we must only check for the

mergibility of white leaf nodes.

The worst-case execution time of this algorithm is proportional to the sum of the number of nodes in the two input quadtrees. Note that as a result of actions (1) and (3), it is possible for the intersection algorithm to visit fewer nodes than the sum of the nodes in the two input quadtrees.

The union operation is implemented easily by applying DeMorgan's law to the above intersection algorithm. For example, Figure 18d is the result of the union of the quadtrees of Figures 18a and 18b. When the set-theoretic operations are interpreted as Boolean operations, union and intersection become "or" and "and" operations, respectively. Other operations, such as "xor" and set-difference, are coded in an analogous manner with linear-time algorithms. Since all of these algorithms are based on preorder traversals, they will execute efficiently regardless of the way the quadtree is represented (e.g., pointers, locational codes, DF-expressions, etc.). We also observe that clipping is a special case of the intersection operation. In this case, one of the input quadtrees corresponds to a black region that represents the display screen's location and size, thereby making clipping easy to implement using quadtrees.

3.3.2. RECTILINEAR UNALIGNED QUADTREES AND SHIFT OPERATIONS

Implicit in the intersection algorithm given above is the assumption that both input quadtrees correspond to the same region (although the individual pixels can have different values). In this section we are interested in the situation where the quadtrees correspond to regions of the same size but their lower lefthand corners correspond to different positions. For example, consider the two 4×4 quadtrees given in Figures 19a and 19b whose lower lefthand corners are at locations (0,2) and (2,0) respectively. This alignment information is stored separately from the quadtree. Thus in order to translate or rotate a quadtree we only need to update the alignment information. However, when two quadtrees of differing alignment must be operated upon simultaneously (e.g., intersected), then the algorithm must take the differing alignments into consideration as it traverses the two quadtrees. Such quadtrees are termed *unaligned* quadtrees.

Processing unaligned quadtrees is simplified by the observation that if a square of size $w \times w$ (parallel to the x and y axes) is overlaid on a grid of squares such that each square is of size $w \times w$, then it can overlap at most four of those squares (see Figure 20a) and those four squares will be neighbors (i.e., they form a $2w \times 2w$ square). We will refer to this case as the *rectilinear unaligned-quadtree* problem. In the case where the $w \times w$ square is not parallel to the x and y axes, we have the *general unaligned-quadtree* problem. In that case, we observe that when an arbitrary square of size $w \times w$ is overlaid at an arbitrary orientation upon a grid of squares such that each square is of size $w \times w$, it can cover at most six grid squares (see Figure 20b). These six or less grid squares will lie within a $3w \times 3w$ square where the center square of the $3w \times 3w$ square is always one of the intersected squares.

To handle the rectilinear unaligned-quadtree intersection problem, we adopt the convention that the output quadtree will be aligned with the first quadtree. We refer to the first quadtree as the *aligned* quadtree and to the second quadtree as the *unaligned* quadtree. For example, intersecting the quadtrees in Figure 19a and 19b so that the quadtree of Figure 19a is the aligned quadtree yields the quadtree of Figure 19c. On the other hand, if the quadtree of Figure 19b is the aligned quadtree, then the result is represented by the quadtree of Figure 19d. When intersecting aligned quadtrees (see Section 3.3.1), we examined pairs of nodes that overlaid identical regions. In contrast, when intersecting rectilinear unaligned quadtrees, upon processing a node in the aligned quadtree, say A, we must inspect at most four nodes (say U_1, U_2, U_3, U_4) from the unaligned quadtree that overlap the corresponding region. Note that A corresponds to one of the shaded squares in Figure 20a while U_1, U_2, U_3, U_4 correspond to the overlapped grid cells. When A is not white, we may have to process the sons of A further. In this case, the four nodes from the unaligned quadtree that overlap a given son of A are chosen from the sons of U_1, U_2, U_3, and U_4. Thus an efficient recursive top-down algorithm for this version of the quadtree intersection problem can be easily implemented. The execution time of this algorithm is proportional to the sum of the sizes of the two input quadtrees and the size of the output quadtree. Note that this bound is slightly different than the bound obtained for the aligned intersection algorithm as in this case the size of the output quadtree is not bounded from above by the sum of the sizes of the two input quadtrees.

Shifting a quadtree can be viewed as a special case of the rectilinear unaligned-quadtree algorithm. In particular, suppose it is desired to shift a quadtree, say A, to the right by n units and up by m units. In such a case, a quadtree, say B, is created representing a black square whose width is the same as that of A and whose origin is n units to the left and m units below the origin of A. We now use B as the aligned quadtree and A as the unaligned quadtree in our rectilinear unaligned-quadtree algorithm. The resulting output quadtree will be a shifted version of quadtree A. For example, see Figure 21 where the 4×4 quadtree of Figure 21a is shifted to the right by 2 units and up by 1 unit. The position of the aligned quadtree relative to the unaligned quadtree is shown in Figure 21b using broken lines while Figure 21c is the resulting shifted quadtree. Following the analysis of the previous paragraph, a quadtree can be shifted an integer number of pixel widths in time linear with respect to the sizes of the original and resulting quadtrees.

3.3.3. GENERAL UNALIGNED QUADTREES AND ROTATIONS

The general unaligned-quadtree algorithm is analogous to the algorithm of Section 3.3.2 for rectilinear unaligned quadtrees. The only difference is that each node in the aligned quadtree can be overlapped by as many as six nodes in the unaligned quadtree (see Figure 20b). Just as shifting was a special case of the rectilinear unaligned-quadtree intersection algorithm, rotation is a special case of the general unaligned-quadtree intersection algorithm. In particular, suppose we wish to rotate a quadtree, say A, counterclockwise by m degrees. In such a case, a quadtree, say B, is created representing a black square whose width is the same as that of A but one that has been rotated by m degrees in the clockwise direction about the appropriate center. We now use B as the aligned quadtree and A as the unaligned quadtree in our general unaligned-quadtree algorithm. The resulting output quadtree will be a rotated version of quadtree A. In the following we describe the rotation of the quadtree of Figure 22a, termed A, by 16 degrees in a counterclockwise direction about its origin. Black block B has been rotated by 16 degrees in the clockwise direction about the origin of A (see Figure 22b). We use broken lines to depict the decomposition of B and solid lines to depict the decomposition of B.

The rotation algorithm proceeds by first determining if B is a terminal node by checking if the maximum of 6 nodes of equal size in A that cover it are of a the same color. If yes, then we are done. Otherwise, B is subdivided (it is a gray node) as are the relevant nodes in A. This process is repeated until either all nodes in B's trees are terminal or we have reached a maximum level of decomposition. In our example, the first subdivision is illustrated in Figure 22c and its result is given in Figure 22d. Notice the use of the "?" symbol to indicate that the block will be subdivided further. The NE quadrant in Figure 22d (corresponding to the block labeled B2 in Figure 22c) is white because the blocks in A that overlap it (just the two blocks labeled A2 and A4 in Figure 22c) are white. Prior to proceeding further, we should check to see if any of the subdivided blocks of B have four identically colored sons, in which case a merge must occur.

Next, we subdivide the blocks labeled with a "?" in Figure 22d as well as blocks A1, A2, A3, and A4 in Figure 22c to obtain Figure 22e. Again, we now check each of the newly obtained subblocks of B to see if they are covered by subblocks of A that are of the same color. In this case we find that this is true for blocks B5 and B6 in Figure 22e (i.e., by black subbblocks A5, A6, A7, A8, and A9), as well as blocks B7 and B8. The result is given in Figure 22f with "?" denoting that the block will be decomposed further. One more level of decomposition is depicted in Figure 22g and the resulting rotated quadtree is shown in Figure 22h. Checking if any of the subdivided blocks in B have identically colored sons finds that the four blocks of the NW son of the NW quadrant in Figure 22h should be merged as they are all white. At this point the resulting quadtree is at the same level as the original unrotated quadtree. Nodes labeled with a "?" can be assigned either black or white as is desired. This may cause more merging.

Since we are usually working in a digitized space, the rotation operation is not generally invertible. In particular, a rotated square usually cannot be represented accurately by a collection of rectilinear squares. However, when we rotate by 90°, then the rotation is invertible. For example, Figure 23b is the result of rotating Figure 23a by 90° counterclockwise. The algorithm traverses the tree in preorder and rotates the pointers at each node. For a counterclockwise rotation by 90° it is based on the following observations.

(1) All of the pixels in the NW quadrant of the image are in the SW quadrant;
(2) All the pixels in the NE quadrant of the image are in the NW quadrant;
(3) All the pixels in the SE quadrant of the image are in the NE quadrant;
(4) All the pixels in the SW quadrant of the image are in the SE quadrant;
(5) Each of the quadrants in its new position appears as if it had been locally rotated clockwise by 90-degrees.

Although the operations discussed in this and the previous subsections are presented for binary raster quadtrees, they can be extended in a straightforward manner to raster quadtrees that have multiple colors and to vector quadtrees. However, vector quadtree algorithms generally require more bookkeeping operations than the corresponding raster quadtree algorithms and consequently are more difficult to analyze.

3.4. SCALING QUADTREES AND MULTIRESOLUTION REPRESENTATIONS

Besides the traditional graphics operations of translation (shifting) and rotation, which are discussed above, there is also the scaling operation. To make an image represented by a quadtree half the size that it was originally, we need only create a new root and give that root three white (or empty in the case of vector quadtrees) sons and one son that was the original quadtree. To make the quadtree twice as big, we choose one of the subtrees to serve as the new root (e.g., the SW subtree) thus eliminating the remaining three subtrees. If a particular portion of the quadtree is to be doubled or halved in size, then a shift operation may have to be performed for the purpose of alignment. The above techniques can be applied to scaling by any power of two. Scaling by an arbitrary factor, say f, is handled by using the property that when a square, say S, of size $f \cdot w \times f \cdot w$ $(0 \leq f \leq 1)$ is placed on a grid of squares such that each square is of size w, then S can overlap no more than four grid squares. Note that arbitrary scaling is implemented in a manner similar to that used for the rectilinear unaligned-intersection problem.

Progressive transmission of images represented by quadtrees can be achieved by taking advantage of the above techniques for scaling by powers of two. Progressive transmission of an image enables the receiver to preview a reduced resolution version of the image before seeing it in its entirety. For example, using such a scheme for the triangle of Figure 14 means that we would first see Figure 14d, then Figure 14e, and finally Figure 14f. This facilitates browsing a database of images. One successful approach [Hill83, Kawa80a, Same85e, Sloa79] is to transmit the nodes of a raster quadtree in breadth-first order, so that large leaf nodes are seen first.

3.5. BOTTOM-UP NEIGHBOR FINDING

Many quadtree algorithms involve more work than just traversing the tree. In particular, in several applications we must perform a computation at each node that depends on the values of its adjacent neighbors. Thus we must be able to locate these neighbors. There are several techniques for achieving this result. One approach [Klin79] makes use of the coordinates and the size of the node whose neighbor is being sought in order to compute the location of a point in the neighbor and then performs an algorithm similar to that described in Section 3.1 for the point location problem. For a $2^n \times 2^n$ image, this can require n steps corresponding to the path from the root of the quadtree to the desired neighbor. An alternative approach, and the one we describe below, only makes use of father links and computes a direct path to the neighbor by following links in the tree. This method is termed *bottom-up neighbor finding* and has been shown to require an average of four links to be followed for each neighbor that is sought [Same82a, Same85f].

In this section we shall limit ourselves to neighbors in the horizontal and vertical direction that are of size equal to or greater than the node whose neighbor is being sought. For neighbors in the diagonal direction, see [Same82a]. Finding a node's neighbor in a specified horizontal or vertical direction requires us to follow father links until a common ancestor of the two nodes is found. Once the common ancestor is located, we descend along a path that retraces the previous path with the modification that each step is a reflection of the corresponding prior step about the axis formed by the common boundary between the two nodes. The general flow of such an algorithm is given in

Figure 24. For example, when attempting to locate the eastern neighbor of node A (i.e., node G) in Figure 24, node D is the common ancestor of nodes A and G, and the eastern edge of the block corresponding to node A is the common boundary between node A and its neighbor. The main idea behind bottom-up neighbor finding can be understood by examining more closely how the nearest common ancestor of a node, say A, and its eastern neighbor of greater than or equal size, say G, is located. In particular, the nearest common ancestor has A as one of the eastern-most nodes of one of its western subtrees, and G as one of the western-most nodes of one of its eastern subtrees. Thus as long as an ancestor X is in a subtree that is not an eastern son (i.e., NE or SE), we must ascend the tree at least one more level before locating the nearest common ancestor.

3.6. CONSTRUCTING QUADTREES

Before we can operate on images represented by quadtrees, we must first build the quadtrees. This involves being able to convert between a number of different data formats and the quadtree. In this section we briefly describe the construction of raster quadtrees from raster data and vector data. The construction of vector quadtrees from either type of data can be performed in an analogous manner.

The algorithm for building a raster quadtree from a two-dimensional array can be derived directly from the definition of the raster quadtree [Same80b]. When building a quadtree from raster data presented in raster scan order (i.e., the array is processed row by row) [Same81a] we use the bottom-up neighbor-finding algorithm described in Section 3.5 to move through the quadtree in the order in which the data is encountered. For example, considering the quadtree of Figure 7 as a 4×4 image means that its image elements are examined in the order indicated in Figure 25. Such an algorithm takes time proportional to the number of pixels in the image. Its execution time is dominated by the time necessary to check if nodes should be merged. This can be avoided by use of predictive techniques which assume the existence of a homogeneous node of maximum size whenever a pixel that can serve as an upper left corner of a node is scanned (assuming a raster scan from left to right and top to bottom). In such a case, merging is reduced and the algorithm's execution time is dominated by the number of blocks in the image [Shaf87] rather than by the number of pixels. However, this algorithm does require the use of an auxiliary data structure (implemented by a fixed-size array in [Shaf87]) of size on the order of the width of the image to keep track of all active quadtree blocks (i.e., blocks containing pixels that have not yet been encountered by the raster scanning process).

Building a raster quadtree from vector data is more complicated than from raster data. This is because a list of line segments has no inherent spatial ordering. A top-down algorithm for producing a raster quadtree from vector data takes as input a list of line segments. This list is recursively clipped against the region, say R, represented by the root of the current subtree of the quadtree. If no line segments fall within R, then a white leaf node is created. If R is of pixel size and contains at least one line segment, then a black leaf node is created. Otherwise, a gray node corresponding to R is created and the algorithm is recursively applied to each of its four children using the list that has been clipped.

Alternatively, we could use a bottom-up approach to building the raster quadtree from vector data. First, we must convert the line segments into a list of pixel-to-pixel steps (also known as chain codes [Free74]) using a traditional line-drawing algorithm [Bres65]. Next, we follow the path formed by the chain codes of the line segments creating black pixel-sized leaf nodes [Same80a]. This is done by using the bottom-up neighbor finding algorithm of Section 3.5 Average-case analysis for the execution time of the chain code to raster quadtree algorithm can be shown to be linear in the length of the chain code by using the analysis in [Same80a] in conjunction with the Quadtree Complexity Theorem. Moreover, by preprocessing the chain code, it has been shown that the worst-case analysis of this algorithm is also linear in the length of the chain code [Webb84]. Neighbor-finding methods have also been used to construct chain codes from quadtrees [Dyer80], as well as two-dimensional arrays in a row by row manner [Same84a].

3.7. POLYGON COLORING

Another raster operation that can be efficiently implemented in quadtrees using neighbor finding is the seed-filling approach to polygon coloring. The classic seed-filling algorithm [Roge85] has as its input a starting pixel location and a new color. The algorithm propagates the new color throughout the polygon containing the starting pixel location. When using arrays, this algorithm is coded by a recursive routine that checks if the color of the current pixel is equal to that of the original color of the start pixel. If yes, then its color is set to the new color and the algorithm is applied to each of the current pixel's four neighboring pixels (for a 4-connected region). The array implementation of this algorithm can be adapted to quadtrees by using bottom-up neighbor finding. Another approach to coloring a region is to color the border of the region and then move inward from smaller to larger quadtree nodes [Hunt78, Hunt79a]. This algorithm could also be implemented using bottom-up neighbor finding.

A more general version of polygon coloring is connected-component analysis. Here, the task is to take a binary image and recolor each of the distinct black regions so that each region has a unique color. The general approach is to traverse the quadtree in preorder and attempt to propagate different colors across the different regions. We discuss three techniques for propagating the colors. The first technique is to perform the quadtree-based seed-filling polygon-coloring algorithm described above whenever a new region is encountered during the traversal. The second technique consists of a three stage algorithm [Same81b]. The first stage propagates the color of a node to its southern and eastern neighbors. This may result in coloring a single connected component with more than one color in which case the equivalence of the two colors is noted. These equivalences are merged in the second stage. The third stage updates the colors of all nodes of the quadtree to reflect the result of the second stage. Often, the first and second stages can be combined into one stage [Same85b, Same86c]. The third technique [Webb84] is a modification of the second technique and avoids the second stage of merging equivalences. Each time the border of a new region is encountered, the preorder traversal is interrupted and the border of the region is traced and colored using bottom-up neighbor finding. At the end of the trace, the preorder traversal is resumed.

Both the second and third techniques described above use a special kind of neighbor finding, i.e., they perform a preorder traversal of a quadtree and are interested in some of the neighbors of each node in the traversal. For this approach top-down neighbor finding can be used to produce improved worst-case results [Same82b, Jack83, Webb84, Same85a]. Top-down neighbor finding is based on the observation that the neighbor of a node is either 1) a sibling of the node or 2) a child of a neighbor of the node's father. Thus, the neighbors of a node can be transmitted as parameters to the function that is performing the preorder traversal of the quadtree. The same idea can be used for efficiently calculating the perimeter of a region represented by a quadtree [Jack83].

3.8. QUADTREE HIDDEN-SURFACE ALGORITHMS

Probably one of the most basic graphics operations is the conversion of an internal model of a three-dimensional scene into a two-dimensional scene that lies on the viewplane for the purpose of display on a two-dimensional screen. This is known as the hidden-surface operation (also discussed as the visible-subset problem in Section 2.1). While there are many mappings that are abstractly possible between a three-dimensional space and a two-dimensional space, we are interested in a mapping that closely models classical optics. Such mappings are called projections (see Section 4.2 for more details). Each pixel of the viewplane determines a pyramid that is formed by the set of all rays originating at the viewpoint and intersecting the viewplane within the boundary of the pixel (see Figure 26). In the simplest case, a color is assigned to each pixel that corresponds to the color of the object that is closest to the viewpoint while also lying within the pixel's pyramid. The hidden-surface task [Suth74] can be conceptualized as a two-stage sorting process. The first stage sorts the surfaces into the different viewing pyramids (this is also known as a bucket sort). The second stage sorts the surface within a given viewing pyramid to determine the closest one to the viewpoint.

There are four approaches to this task that are relevant to this discussion. First, the three-dimensional scene can be viewed as a sequence of overlays of two-dimensional scenes each of which is represented by a quadtree. Second, quadtrees can be used to model the viewplane even when the three-dimensional scene consists of polygons of arbitrary orientation and placement in the three-dimensional space. This solution was first proposed by Warnock [Warn68, Warn69a] and is known as Warnock's algorithm. It is an image-space method. Warnock was actually interested in two versions of the hidden-surface task: 1) the basic hidden surface task and 2) the hidden-line task, which is an adaptation of the hidden surface task to a wireframe representation of a solid. In the process of developing a solution to the hidden surface problem, Warnock also made contributions [Warn69b] to light modeling that are beyond the scope of this paper. For expository purposes, we shall first describe the hidden-line operation in our discussion, and conclude with a description of its adaptation to the hidden-surface operation. Third, we present Weiler and Atherton's object-space hidden-surface algorithm [Weil77], which is analogous to Warnock's image-space algorithm. Weiler and Atherton also point out how image-space heuristics can be used to speed up object-space methods. Fourth, the parametric space of the surface of a three-dimensional object can be modeled by a quadtree. The first three approaches assume a vector data representation of a three-dimensional scene.

3.8.1. 2.5-DIMENSIONAL HIDDEN-SURFACE ELIMINATION

2.5-dimensional hidden-surface elimination is a technique devised to handle the display of three-dimensional scenes that are represented by a forest of quadtrees. It arises most commonly in applications in cel-based animation. A *cel* is a piece of transparent plastic on which a figure has been painted. A *scene* can be created by overlaying cels (see Figure 27). A given view of the scene can be constructed by first laying down the cel representing the background. On top of the background, cels are placed that represent objects in the foreground. When each cel is represented by a quadtree, scenes can be constructed easily. Cel-based scene construction is a simplification of the hidden-surface task and is described in greater detail below. It is simpler than the general three-dimensional problem because each object is restricted to be in just one cel. Thus we need not be concerned with problems resulting from situations such as object A occluding object B, object B occluding object C, and object C occluding object A. In other words, in our domain occlusion is transitive whereas it need not be so in an unrestricted three-dimensional domain (e.g., Figure 28).

2.5-dimensional hidden-surface elimination is equivalent to a sequence of set-union operations and can be implemented in a manner analogous to quadtree intersection as described in Section 3.3.1. In particular, starting with the quadtree corresponding to the backmost cel, while moving towards the front cel, perform successive overlays of the quadtrees of the cels encountered along the path [Kauf83]. Hunter [Hunt78] has shown that the total cost of this process is proportional to the sum of the number of nodes in all of the quadtrees of the cels.

While the algorithm given above is an optimal worst-case result, an algorithm with a better average-case performance is possible. In essence, if the cels are processed from front-to-back, then certain blocks in the intermediate quadtree can be marked as transparent thereby indicating that up to now nothing in the sequence of quadtree nodes corresponding to that location has been opaque. We also mark the internal nodes as opaque if all of their subtrees are opaque although their subtrees need not be the same color (i.e., correspond to the same object). Thus when traversing the intermediate quadtree and the next cel, say C, if the intermediate quadtree has an internal node that is marked opaque, then nothing in the corresponding subtree of cel C, say T, is visible, and hence T need not be traversed. Furthermore, when the root of the intermediate quadtree is marked opaque, then no more cels need be visited. Such actions have a potential of reducing the execution time of the 2.5-dimensional hidden-surface elimination task because subtrees corresponding to invisible regions need not be traversed. Of course, in a more flexible animation system, it is often desirable to overlay unaligned cels (i.e., unaligned quadtrees). This can be handled by using the techniques described in Section 3.3.2 and 3.3.3 for computing set operations on unaligned quadtrees.

3.8.2. WARNOCK'S ALGORITHM

The usage of the quadtree for modeling the viewplane during the hidden-surface operation was first described by Warnock [Warn68]. The quadtree is used to store the parts of the scene that are currently believed to be visible. The hidden-line operation is a derivative task of the hidden-surface operation. The difference between them is how the result of the visibility calculation is displayed. We will now describe the hidden-line operation in greater detail. In this case, the viewplane's quadtree consists of polygons formed by the visible edges of the objects in the three-dimensional scene. At most one edge is associated with each pixel. The edge, if any, that is associated with a pixel, corresponds to the one that passes through the pixel's region as part of the border of a polygon that is not occluded by another polygon which is closer to the viewpoint. In the following discussion we use the verb *color* to distinguish between edges of different polygons. In other words, a pixel is output (i.e., colored) if a visible edge passes through it.

The quadtree is used in the display process to rapidly select the pixels that need to be colored (these are the pixels through which visible edges of the scene pass). The quadtree is not built explicitly. Instead, the viewplane is recursively decomposed (traversed as if it were a quadtree) using an appropriate decomposition rule to yield a collection of disjoint square regions (i.e., leaf nodes). At each such region, drawing (i.e., coloring) commands for driving a display are output.

The type of quadtree decomposition rule that is used is analogous to the one devised by Hunter and Steiglitz [Hunt78, Hunt79a]. In particular, a pixel is represented by a boundary node if an edge of a polygon passes through it; otherwise it is an empty node. Empty nodes are merged to yield larger nodes while boundary nodes are not merged. Using this rule enables us to formulate the actions taken by Warnock's algorithm in terms of the following leaf node types and corresponding actions.

(1) At an empty leaf node, draw nothing since no lines pass through this region.
(2) At a leaf node corresponding to a pixel, draw a point representing the border of the polygon that occludes the upper lefthand corner of the pixel (if no such polygon exists, then draw nothing).
(3) At a leaf node corresponding to a collection of polygons, draw nothing since the existence of such a node means that one of the polygons occludes all the other polygons over this region.

At this point it is interesting to briefly explain the relationship between the hidden-surface and hidden-line tasks. The hidden-line problem is closely identified with the usage of vector displays and plotters. This caused Warnock to investigate edge quadtree-like decompositions [Warn69b]. On the other hand, the hidden surface problem is closely identified with the usage of raster devices. Although our treatment of the hidden-line problem assumes vector data, it results in the output of line-drawing commands at a raster/pixel level. By doing a bit more calculation, we can often recognize that a line will be visible without having to subdivide all the way down to the pixel level.

The algorithm given above for the hidden-line problem can be modified to handle the hidden-surface problem as well. The only modification which needs to be made here is that empty nodes which represent regions that are completely spanned by a polygon must now be colored with the color of that polygon, instead of being ignored (as happens in the hidden-line display process).

One problem with building quadtree decompositions of data presented as arbitrary collections of polygons in three-dimensional space is to determine when there is no need for further subdivision. For example, this situation arises when a node contains a collection of polygons where one polygon completely occludes the other polygons. This requires a sort of all the polygons in the node. Since occlusion is not in general transitive, sorting does not always work (recall Figure 28). If sorting fails due to non-transitivity or because the nearest polygon does not occlude the entire region, then further subdivision is needed to determine what is visible in the region corresponding to the node.

It is worthwhile to note that we have been assuming that the closest polygon's color was the most appropriate color for a pixel. However, clearly a pixel could contain small features that this approach would represent falsely. This general problem is referred to as antialiasing [Roge85]. Warnock handled the situation of a pixel that contained complicated features by pretending that the viewing pyramid for the pixel was a single ray passing through the pixel's upper lefthand corner. If this produces an approximation of the image that is too rough for a particular application, then classical antialiasing techniques [Roge85] (such as computing a weighted average of the visible intensities within a pixel) can be applied without altering the basic algorithm.

3.8.3. WEILER-ATHERTON'S ALGORITHM

Warnock's algorithm is an image-space hidden-surface algorithm. Weiler and Atherton [Weil77] developed an analogous object-space hidden-surface algorithm. The object space consists of a collection of polygons. It is interesting to note that Weiler and Atherton use image-space heuristics to speed up their object-space algorithm. Their object-space algorithm has the following structure:

(1) Order all the polygons by their smallest z-value (where the viewer is located at a z-value of minus infinity).

(2) Find the closest polygon, say C, to the viewer.

(3) Form two collections of polygons. The first collection contains those polygons whose projection overlaps (partially or totally) the projection of C (which we will call the *inner set*) The second collection contains those polygons whose projection is not entirely covered by the projection of C (which we will call the *outer set*). In cases where a polygon is in both the inner and outer sets, it is often convenient to clip that polygon against C and store the resulting polygons in the appropriate sets.

(4) Remove all polygons from the inner set that do not occlude part of C.

(5) If no polygons occlude C (i.e., the inner set is empty), then the hidden-surface problem has now been solved for C and proceed to solve the hidden-surface problem for the outer set.

(6) If there exist polygons that occlude C (i.e., the inner set is non-empty), then recursively go to step (2) and choose a "nearest" polygon from among the occluding polygons from the inner set of C. Upon return, process the outer set of C.

In order to reduce the number of polygons that have to be compared in step (4), Weiler and Atherton propose two preprocessing methods that are relevant to our study.

The first method recursively subdivides the image space (in the x and y directions) until the number of polygons in a given region, say R, drops below a specified threshold. Within region R, the basic algorithm, described above, is used. Note that at step (4), only the polygons in region R need to be considered. The second method is based on the observation that besides preprocessing by subdividing in the x and y directions, it might also be useful to subdivide in the z direction. In particular, after subdividing in the z direction, they propose to solve the hidden-surface problem for the backmost volume elements and then using this solution as part of the polygon list for the volume elements in the front. This back-to-front approach is also discussed in Section 3.8.1 in the context of 2.5 dimensional hidden-surface elimination. This last heuristic could be viewed as an octree method (see Section 4.2 for more details).

3.8.4. DISPLAYING CURVED SURFACES

In this paper, vector data is usually viewed as consisting of straight line segments and polygons. However, the quadtree paradigm also has proven useful to researchers interested in the manipulation of curved features such as surfaces. Curved surfaces are often represented by a collection of parametric bicubic surface patches [Mort85]. Curved surface representations are important in computer graphics applications because they are often more compact than polygonal representations and also because they enable the stipulation of continuity in the derivative of piecewise surface representations which is important for ray-tracing calculations (see Section 4.3).

One early approach to displaying such surfaces was developed by Catmull [Catm75]. The idea is to recursively decompose the patch into subpatches until the subpatches that are generated are so small that they only span the center of one pixel (or can be shown to lie outside the display region). The test for how many pixel centers are spanned by the patch (or whether or not the patch lies outside the display area) is based on the approximation of the patch by a polygon connecting the patch's corners. In our examples, patches are denoted by solid lines and their approximating polygons are denoted by broken lines.

As an example of the recursive decomposition of patches, consider Figure 29. Figure 29a shows a single patch with corners A, B, C, and D on a grid of pixel centers. We observe that quadrilateral ABCD which approximates the patch ABCD, contains more than one pixel center. Thus the patch must be decomposed. Figure 29b shows the decomposition of patch ABCD into quadrilateral patches AFIE, BGIF, CHIG, and DEIH. Since the quadrilateral approximations of each of these patches, again, span more than one pixel center, they must each be subdivided further as shown in Figure 29c. This time there is not enough detail in the figure to show the difference between the patch and its quadrilateral approximation. Note that in Figure 29c the quadrilateral approximation for patch JFKN contains only one pixel center and hence will not need to be subdivided further. Also, the quadrilateral approximation of patch MNLG contains no pixel centers and thus it too will not need to be subdivided further. However, the quadrilateral approximation of patch IJNM contains two pixel centers and hence will need to be subdivided further. The final decomposition of the original patch is shown in Figure 29d, and the raster image yielded by this decomposition is shown in Figure 29e.

As was observed by Catmull, the recursive decomposition approach to approximating the location of a patch can be generalized and thereby applied to other patch representations. Patch representations based on characteristic polyhedrons (e.g., Bezier and B-spline patches) [Mort85], allow these test decisions to be based on an approximation of each patch by the convex hull enclosing its control points (which is guaranteed to enclose the entire patch). This yields a more accurate result than Catmull's approximation which is only based on four corners of a patch.

As with Warnock's algorithm, Catmull's algorithm is oriented toward the generation of display commands. Thus it does not explicitly generate the quadtree structure although its processing follows the quadtree decomposition paradigm in the parametric space. Since the patches exist in three-dimensional space, more than one patch can span the same pixel center, the Catmull algorithm makes use of a z-buffer to keep track of the intensity/color of the patch that has most recently been found to be closest to the viewpoint. Basically, a z-buffer is a two-dimensional array that represents the displayed image. Each entry of the array contains a color and a depth. Initially, each pixel in the displayed image is black and at an infinite depth. Whenever a new color is to be assigned to a pixel, we first compare the depth of the location to which the pixel corresponds and if it is greater than the current depth in the z-buffer, then the assignment is ignored; otherwise, update the color and depth values in the z-buffer. Although, traditionally, the z-buffer has aliasing problems (i.e., it produces jagged borders between neighboring regions), these can be mitigated by using the rgb-α-z approach [Duff85].

Warnock's algorithm requires that the scene be completely specified at the time the algorithm is initiated. In contrast, the z-buffer enables elements of the scene to be processed in an arbitrary order. This permits elements to be added to the scene without having to reprocess elements of the scene that have been previously processed. In other words, at any time during its processing, the z-buffer represents what would be displayed if there were no further elements in the scene. The z-buffer is represented explicitly as a two-dimensional array. Such an array could be represented by a quadtree. This quadtree z-buffer representation might prove useful for generating line representations of the borders between surfaces but not for generating shaded surfaces [Posd82]. Note that raster quadtrees are seldom efficient for representing scenes including shaded surfaces since each pixel location on a shaded surface will have a slightly different color. However, if only the borders of the surfaces are represented, then the interior portions of the surfaces can be efficiently merged.

Catmull's display algorithm has been adapted to handle a constructive solid geometry [Requ80] representation of objects (i.e., objects composed as boolean combinations of primitive objects) for the case where the initial primitives are solids bordered by bicubic surfaces [Carl82]. Instead of subdividing down to the pixel level everywhere, the subdivision is performed only until it has generated subpatches that are mutually disjoint. Two subpatches can be viewed as disjoint when the interiors of the convex hulls of their respective control points are disjoint. While this approach helps determine the actual intersection between two subpatches, it does not address the problem of choosing which patches should be compared to determine the possible existence of an intersection. A vector octree approach to this problem [Nava86b] is mentioned in Section 4.2.3.

4. ALGORITHMS USING OCTREES

In this section we describe some usages of the octree in computer graphics. Due to space limitations and a desire to avoid repetition, we only discuss a few usages. We first review the execution of a number of basic operations using an octree including its construction. Next, we show how to apply the parallel and perspective projection methods to display the collection of objects that are represented by an octree. Implicit in this task is the solution of the hidden-surface problem in order to resolve the interaction between the objects in the scene modeled by an octree. We conclude with a discussion of image rendering (i.e., the problem of calculating what light falls on the view plane) by use of ray tracing and radiosity. Ray tracing models light as particles moving in the scene. The octree speeds up the determination of the objects that are intersected by rays emanating from the viewpoint. In contrast, radiosity models light as energy and seeks to determine a point at which its distribution is at equilibrium. This requires the derivation of a large set of linear equations. Using octrees can simplify the process of calculating the coefficients of these equations. This is especially true if rendering is to be done with respect to more than one viewpoint. The efficient solution of these equations is aided by use of heuristics, one of which is the adaptive recursive decomposition of the scene's surface analogous to that used by the algorithms of Warnock and Catmull (see Section 3.8.4).

4.1. BASIC OPERATIONS

The algorithms for performing basic computer graphics operations such as translation, rotation, scaling, and clipping on both raster and vector octrees are direct extensions of the algorithms discussed earlier for quadtrees. The techniques which were used in performing some of these operations (e.g., preorder traversal, rectilinear unaligned traversal, general unaligned traversal, bottom-up neighbor finding and top-down neighbor passing) can all be extended to deal with octrees once some additional bookkeeping information is maintained.

Building an octree is not an easy process from the point of view of the sheer amount of data in a three-dimensional image that must be examined. Clearly, the amount of work to construct a raster octree from a three-dimensional array representation of an image is quite costly due to the large number of primitive elements that must be inspected. If we start with an array representation, the conventional raster-scanning approach used to build quadtrees is computationally expensive because much time is spent detecting the mergibility of nodes. This can be alleviated, in part, by using the predictive techniques of Shaffer and Samet [Shaf87] (see Section 3.6). This method makes use of an auxiliary array whose storage requirements are as large as a cross-section of the image which may render the algorithm impractical. However, since this array is often quite sparse, this problem can be overcome by representing it by use of a linked list of blocks in a manner similar to that used by Samet and Tamminen [Same85b] for connected component labeling for images of arbitrary dimension. Alternatively, we can initially represent the data by using one of the more compact three-dimensional representations such as the boundary method or the CSG tree [Requ80].

The boundary method represents a three-dimensional object by its faces. The winged-edge representation [Baum72] referred to in Section 1, when applied to polyhedra, is one such representation. In order to create a boundary representation, we must first decompose the surface of the object into a collection of faces. The result is a graph whose edges correspond to the interconnections between the faces of the object. For example, the object in Figure 30a can be decomposed into the set of faces and interconnections shown in Figure 30b. There are a number of variants of this representation. They arise from the use of different methods of representing individual faces (which could be either polygons or curved surfaces), and different approaches of specifying the interconnection between adjacent faces. For example, we can view faces as meeting at either their borders or corners. Thus instead of a graph where the vertices represent faces and the edges represent their interconnection, we also have boundary methods where the vertices of the graph represent borders of a face or even the corners of a face. Tamminen and Samet [Tamm84b] describe a method for building a raster octree from a boundary representation by use of connectivity labeling.

Constructive Solid Geometry (CSG) methods represent rigid solids by decomposing them into primitive objects that are subsequently combined using variants on Boolean set operations such as union, intersection, and set-difference, and possibly geometric transformations (e.g., translation and rotation). These primitives are often in the form of basic solids such as cubes, parallelepipeds, cylinders, spheres, etc. A more fundamental primitive is a halfspace whose border can either be linear or non-linear. For example, a linear halfspace in three dimensions is given by the following inequality:

$$a \cdot x + b \cdot y + c \cdot z \geq d$$

CSG methods are usually implemented by a CSG tree which is a binary tree in which internal nodes correspond to geometric transformations and Boolean set operations while leaves correspond to the primitive objects (e.g., halfspaces). For example, the object in Figure 30a can be decomposed into three primitive solids whose CSG tree is shown in Figure 30c. Samet and Tamminen [Same85d] show how to build a bintree representation of a raster octree from a CSG tree (see also [Wood82]). These techniques are useful for conversion as well as display [Kois85, Morr85].

An even more fundamental problem than building the octree is the acquisition of the initial boundary data to form the boundary representation. One approach is to use a three-dimensional pointing device to create a collection of samples from the surface of the object. Having collected the point data, it is then necessary to interpolate a reasonable surface to join the point data. This interpolation can be achieved by triangulation. A triangulation in three-dimensional space is a maximal set of disjoint triangles that form a surface whose vertices are points in the original data set. There are many triangulation methods currently in use, both in two-dimensional spaces [Wats84] and three-dimensional spaces [Faug84]. They differ by how they determine which points are to be joined. For example, often it is desired to form compact triangles instead of long narrow ones. However, the problems of minimizing total edge length or maximizing the minimum angle pose difficult combinatorial problems. Posdamer [Posd82] has proposed to use the ordering imposed by an octree on a set of points as the basis for determining which points should be connected to form the triangles.

Posdamer's algorithm uses an octree whose leaf criterion is that no leaf can contain more than three points. The initial set of triangles is formed by connecting the points in the leaves that contain exactly three points. Whenever a leaf node contains exactly two points, these points are connected to form a line segment that is associated with the leaf node. This is the starting point for a bottom-up triangulation of the points by merging disjoint triangulations to form larger triangulations. The isolated points (i.e., leaf nodes that contain just one point) and isolated line segments are viewed as degenerate triangulations. The triangulation associated with a gray node is the result of merging the triangulations associated with each of its sons. By *merging* or *joining* two triangulations, we mean that a sufficient number of line segments is drawn between vertices of the two triangulations such that we get a new triangulation that contains the original two triangulations as sub-triangulations.

When merging the triangulations of the eight sibling octants, there are a number of heuristics that can be used to guide the choice of which triangulations are joined first. The order in which we choose the pair of triangulations to be joined is determined, in part, by the following factors. First, and foremost, it is preferred to merge triangulations that are in siblings that have a common face. If this is impossible, then triangulations in nodes that have a common edge are merged. Again, if this is not feasible, then triangulations in nodes that have a common vertex are merged. Within each preference, the triangulations that are closest according to some distance measure are merged first.

Alternative approaches to building an octree consist of taking a number of different views of an image [Chie86, Veen85] or even range data [Conn84]. This task can be viewed as the intersection of a collection of sweeps of two-dimensional silhouettes [Chie86]. When using such methods we must be careful that the views are adequate to describe the object in sufficient detail. In medical applications, it is feasible to be working with a collection of images representing slices of the three-dimensional object. Yau and Srihari [Yau83] show how to construct an k-dimensional octree-like representation from multiple $(k-1)$-dimensional cross-sectional images.

4.2. PARALLEL AND PERSPECTIVE PROJECTIONS

Once an octree has been constructed, it is natural to want to display it. The two display techniques used most commonly are the perspective projection and the parallel projection. The perspective projection is formed with respect to a viewpoint and a viewplane. In this case, all points lying on a given line through the viewpoint project onto the same point on the viewplane (see Figure 31a). A parallel projection can be defined as a special case of the perspective projection such that the viewpoint is at infinity (see Figure 31b).

For raster octrees, the most common display technique is the parallel projection [Doct81, Gill81]. The easiest parallel projection to perform on a raster octree is when the viewplane is parallel to one of the faces of a node in the tree. This situation is equivalent to the 2.5-dimensional hidden-surface task discussed in Section 3.8.1. A special case of the parallel projection technique that is of interest to engineers is the *isometric* projection. It has the property that the silhouette of a cube corresponding to the space spanned by root of an octree projects onto a regular hexagon which can be decomposed into six equilateral triangles. These triangles are decomposed further into triangular quadtrees in the process of determining what portions of the leaf nodes are visible for the purpose of display. A display algorithm based on this approach is reported by Yamaguchi *et al.* [Yama84].

Implicit in the task of displaying an octree is the solution of the hidden-surface problem for the interaction among the objects represented by the octree. Not surprisingly, since the octree imposes a spatial ordering on objects, the hidden-surface problem for scenes represented by octrees can be solved more efficiently than the general hidden-surface problem for arbitrary polygons. Note that any opaque object in the four front octants of an octree will occlude any opaque object in the back four octants. This property holds recursively within each of the suboctants. The process of displaying the scene is facilitated by the construction of a *display quadtree* which corresponds to a partial two-dimensional view of the scene. The display quadtree is updated as the nodes of the octree are traversed from back-to-front. Each opaque node, say P, that is encountered in this traversal *paints out* (i.e., overwrites) the previous view contained in a portion of the display quadtree that coincides with the projection of P. Of course, as indicated in the discussion of the 2.5-dimensional hidden-surface problem in Section 3.8.1, the nodes could also be processed from front-to-back, thereby allowing for the possibility of visiting fewer nodes.

Generalizations of the parallel projection to planes of arbitrary position and orientation are described by Meagher [Meag82] and Yau [Yau84]. Generalizations can also be made in a straightforward manner to compute perspective projections onto arbitrary planes as well. Another approach to the perspective projection task is to first transform the three-dimensional scene into a new three-dimensional scene whose parallel projection is the same as the corresponding perspective projection of the original scene. This approach was used on CSG trees by Koistinen *et al.* [Kois85], who then transformed the resulting CSG tree into a bintree for display by one of the parallel projection methods discussed above.

One drawback to displaying scenes represented by a raster octree is that there is little potential of using lighting models for the shading of the scene since adjacent faces of octree nodes meet at 90-degree angles. One approach at overcoming this drawback is described by Doctor and Torborg [Doct81]. They suggest that the amount by which to shade a face of a node can be calculated as a function of the number of the node's transparent neighbors. Thus a node on the corner of an object that is surrounded by empty space will be brighter than another node on the interior of a face of the object, since it has fewer transparent neighbors. This yields an interesting highlighting effect. More recently, this problem has been investigated by [Gord85, Chen85, Brig86].

4.3. RAY TRACING

Although the parallel and perspective projection display techniques are suitable for computer-aided design, realistic modeling of lighting effects generally requires using some variant of raytracing [Roge85]. Raytracing is an approximate simulation of how the light that is propagated through a scene lands on the image plane. This simulation is based on the classical optical notions of reflection (diffuse and specular) and refraction [Whit80]. Although the geometry of the reflection and refraction of "beams" of light from surfaces is straightforward, the formulation of the equations to model the intensity of the light as it leaves these surfaces is a recent development. The quality of the displayed image is a function of the appropriateness of the model represented by these equations and the precision with which the scene was represented. Nevertheless, the amount of time required to display a scene is heavily influenced by the cost of tracing the path of the rays of light as they move backward from the viewer's eye, through the pixels of the image plane, and out through the scene. For example, Whitted [Whit80] reports that as much as 95% of the total picture-generation time may be required to calculate points of intersection between rays of light and objects in a complex scene. Thus the motivation for using the octree in raytracing is to enable the calculation of more rays with a greater amount of accuracy. Since light-modeling equations rely on the availability of accurate information about the location of the normal to the surface at the point of its intersection with the ray, vector octrees are generally more appropriate than raster octrees. This is especially true for vector octrees that can represent curved, rather than planar, surfaces using either curved patches [Nava86b] or curved primitives [Wyvi85].

Octrees have been used to speed up intersection calculations for raytracing [Glas84, Fuji86a, Wyvi85, Kapl85, Jans86]. The basic speedup can be seen by examining the 22-sided polygon in Figure 32a. We use a quadtree instead of an octree in order to simplify the presentation. A naive raytracing algorithm would have to test the ray emanating from the viewpoint against each of these sides, sort the resulting intersections, calculate the reflected ray, and finally test the reflected ray to see if it intersects any other portion of the polygon. Figure 32b shows that a quadtree (octree)-based algorithm would perform the same calculation by visiting only 6 leaf nodes (i.e., nodes 1, 2, 14, 6, 3, and 4). Glassner [Glas84] describes a method to do this using a linear octree addressing scheme where the octree nodes are stored in a hash table rather than a list [Garg82b] or a B-tree [Rose83, Abel84a]. Thus, instead of using standard neighbor-finding techniques (either top-down or bottom-up) to move between the nodes that lie sequentially along a given ray, a neighboring node is located by calculating a point that would lie in the neighbor and then searching the octree for that point. This approach has also been applied to the pointer-based representation of octrees [Wyvi85, Fuji86a]. For an analogous approach using the bintree representation of octrees see [Kapl85].

Although searching for the node containing a particular point can be done very efficiently, standard neighbor-finding techniques should be faster for more complicated scenes. The use of both top-down and bottom-up neighbor-finding for raytracing on an octree is discussed by Jansen [Jans86]. However, more empirical results are required to evaluate the merits of the various neighbor-finding techniques for raytracing typical scenes. Nevertheless, the octree approach to raytracing seems promising. For example, Glassner [Glas84] reports that tracing 597,245 rays in a particular scene of 1,536 objects required 42 hours and 12 minutes using non-octree raytracing techniques, while only 2 hours and 57 minutes were required when using octrees. Another scene that was estimated to require 141 hours using non-octree methods was analyzed in 5 hours and 5 minutes using octrees.

4.4. RADIOSITY

While for many years ray tracing was the dominant approach to the realistic rendering of images, newer and different techniques have recently emerged. One such method is the radiosity approach [Gora84]. Instead of modeling light as particles bouncing around in a scene (as is done in raytracing), the radiosity approach models light as energy whose distribution tends toward a stable equilibrium. In other words, the radiosity approach treats light as if it were heat - i.e., light sources behave as sources of heat, and surfaces that reflect light behave as surfaces that reflect heat. In the subsequent discussion we use the terms *reflect, radiate,* and *emit* interchangeably to denote the light leaving a patch where a *patch* is a portion of a surface of an object in the scene. Although energy in the form of light and heat is normally viewed as a continuous flow, the radiosity method uses a discrete simulation of the flow so that an approximate rendering can be computed.

A scene is viewed as a collection of patches where the light emitted by the surface of a given patch, say Q, is either constant (e.g., for a light source) or is a linear combination of the light falling on Q from all the other patches. The simplifying assumption is made that the surfaces are Lambertian diffuse reflectors, i.e., light is reflected uniformly from the surface in all directions. This restriction can be lifted [Imme86] at the expense of greatly increasing the size of the problem. Of course, many patches do not contribute light to a particular patch because they are occluded by closer patches. In essence, radiosity converts the image rendering problem to one of solving a set of simultaneous linear equations. Each equation represents a portion of the discrete simulation of the light flow, i.e., the portion of the light from the rest of the scene that is eventually reflected by the patch. Furthermore, the equation for patch Q depend on which patches are visible from Q.

The process of deriving the equations (i.e., the determination of the values of their coefficients) that describe the interactions among the patches was the computational bottleneck in the initial presentation of radiosity [Gora84, Cohe85]. Deriving the equations is straightforward although the exact mechanics [Gora84, Cohe85] are beyond the scope of this survey. However, part of this process can be facilitated, in part, by observing that if two patches, say Q and R, are mutually invisible , then the coefficient of the term in the equation of Q (or R) associated with R (or Q) will be zero. The physical interpretation of the concept of mutual invisibility is that light emitted by one of the patches cannot reach the other patch without first being reflected by yet a third patch. A geometric interpretation of this concept is that two patches, say Q and R, are mutually invisible only if there does not exist a pair of points p_Q on Q and p_R on R such that a straight line can be drawn between them without intersecting a third patch or passing through the interior of an object in the scene.

The determination of which terms in the equations have zero coefficients corresponds to a hidden-surface problem among the patches. It must be solved separately for each patch - i.e., if we have M patches, then we must solve the hidden-surface interactions among each of the $O(M^2)$ combinations of patches. Worse, we need to solve these problems not just for a point on a patch, but for every point on the surface of the patch. The solution of these problems could be eased by using a data structure such as the octree to organize the elements of the scene - i.e., the three-dimensional space occupied by the patches. This simplifies the determination of which patches are hidden with respect to the other patches thereby yielding the zero coefficients.

Unfortunately, the application of radiosity to the rendering of more complicated scenes results in a marked increase in the number of equations necessary to model the scene. This has led to a shift of the computational bottleneck so that it is now associated with the problem of solving the simultaneous equations. Nevertheless, recursive subdivision can still be used. Instead of recursively subdividing the three-dimensional space occupied by the patches, Cohen *et al.* [Cohe86] recursively subdivide the surfaces of the patches. This subdivision takes place in the parametric space of the patch in a manner similar to Catmull's algorithm (see Section 3.8.4). As with Catmull's algorithm, recursive subdivision of a patch (described below) does not actually require the construction of a quadtree. Instead, the data is simply aggregated in a manner that is equivalent to applying a particular leaf criterion to the organization of the surface of a scene.

It is interesting to observe that in order to determine the rough flow of light through a scene, the number of patches needed to model the objects in the scene is considerably smaller than the number of patches needed to depict features of the scene that are caused by the actual flow of light through the scene (e.g., shadow boundaries). For example, suppose that we are modeling a scene that corresponds to a room containing boxes. In this case, a rather coarse grid can be used to represent the surface of the room. However, the accurate representation of the shadows that the boxes cast on the walls will usually require a much finer grid. This is especially true for area light sources (e.g., fluorescent tubes that cannot be modeled accurately as point light sources) that cause varying shadow intensities. Of course, the more patches used to represent a scene, the more expensive is the solution process required to solve the corresponding set of equations since there are more equations with a concomitant increase in terms. In particular, the number of equations is proportional to the number of patches which can potentially lead to a quadratic number of interpatch relations. It should be noted that we don't know how many patches will be needed to represent the results of the radiosity calculation until after it has been performed. Early work on radiosity simply guessed the maximum number. However, with more complicated scenes the guesses are overly pessimistic, thereby apparently resulting in needlessly inefficient algorithms. Recursive subdivision performed in an adaptive manner avoids this problem.

Cohen *et al.* [Cohe86] propose a two-step algorithm to reduce the number of equations that must be solved simultaneously. The basic approach is to first solve the set of simultaneous equations corresponding to the light flow among the patches that are used to model the surfaces of the scene. In the second step, patches whose intensity value computed by the first step differs greatly from that of their neighbors are subsequently decomposed into smaller subpatches, termed *elements*, via a regular recursive decomposition (i.e., equal surface area). The rationale for further subdivision is the assumption that the intensity variance in the scene is a continuous function meaning that sharp discontinuities are an artifact of undersampling the intensity function - i.e., the grid was too coarse. The result is that the scene consists of a collection of patches (each corresponding to a small portion of the surface of the scene) where each patch is represented by a quadtree whose leaves are elements. The leaf criterion used to construct the quadtree is one that is based on the absolute intensity difference, across the portion of the surface approximated by the leaf, being below a given threshold.

Now, instead of deriving a new set of equations to represent the interactions between all the elements of each of the patches, the new set of equations assumes that only one patch has been decomposed and the remaining patches are treated as if they have a constant intensity value - i.e., the one computed in the first step. This is equivalent to an assumption that the cumulative effect of elements of patch Q on other patches is approximately the same as that of Q. In other words, for each collection of elements corresponding to a particular patch, a set of simultaneous equations is derived based on the individual variable intensity values of the elements in the collection and treating other patches in the scene as having constant intensity. This greatly reduces the number of equations that need to be solved with only a modest reduction in the accuracy of the solution.

The approach of Cohen *et al.* described above has several advantages. First of all, by applying adaptive decomposition to the individual patches it prevents the size of the set of linear equations (i.e., the number of terms) from growing quadratically. Second, it assumes that the decomposition of a particular patch, say Q, into elements does not change the total amount of light that is reflected by Q and that is therefore incident on the other patches. This means that after solving the problem of determining the light flow with the initial set of patches, the individual behavior of the light flow within a patch can be solved independently of the individual behavior within the other patches. In fact, the result is an asymmetric relation between the effects of patches and elements of patches. For each element in a patch, we compute the effect of the light from the remaining patches. However, the effect of individual elements of patch Q on patch R is taken collectively - i.e., the fact that Q has been decomposed into elements has no effect on the amount of light reflected by R.

As an example, Cohen *et al.* [Cohe86] report that application of these techniques to a scene whose objects required 58 patches and whose optical features (e.g., caused by shadow boundaries) required 1135 elements took 22.49 minutes to derive its radiosity equations and 1.10 minutes to solve them. However, instead of using the adaptive approach, a simple decomposition of the same scene into 829 patches required 90.10 minutes to derive the equations and 6.36 minutes to solve them.

Of course, even though the solution of the radiosity equations is a major part of the image rendering process, there still remain other issues to be considered. In particular, once the radiosity equations have been solved, we must still render the scene from a particular viewpoint. The scene described in the previous example required 14.67 minutes to render 1135 elements and 14.16 minutes to render the 829 patches. In order to improve the rendering time, it is necessary to use data structures that facilitate the solution of the standard hidden-surface problem. In Section 4.2 we suggested that the octree is an appropriate data structure for this problem.

5. CONCLUDING REMARKS

An overview of the use of hierarchical data structures, such as the quadtree and the octree, in computer graphics applications has been presented. This is a rapidly moving area of research which can be expected to yield further improvements in performance of traditional graphics algorithms in the future. In many cases, aside from a potential savings in space requirements, methods that incorporate these techniques also produce significant savings in the execution time of the algorithms. Of course, these data structures are used in applications other than computer graphics, some of which are described below. In addition, we briefly mention some hardware implications of their use.

Variants of quadtrees are used to represent points, lines, and areas in a geographic information system [Same84d]. This enables the data to be handled in an integrated manner and permits answering queries that involve combinations of the different data types. For example, it is easy to answer a query of the form "find all roads passing through swampland in Florida that pass through cities with over 10,000 inhabitants." They also have been applied in finite element mesh generation [Yerr83].

An important advantage of quadtrees and octrees is that it is easy to update them to reflect changes in the scene that they are representing. Thus it is natural that they would prove useful in the representation of scenes that change over time due to the motion of objects within the scene. Ahuja and Nash [Ahuj84] represent motion by updating an octree structure as the object is moved. Alternatively, Samet and Tamminen [Same85d] view a changing three-dimensional scene as a four-dimensional object and use a four-dimensional bintree to represent the space-time object. Besides using octrees to represent motion, they also can be used to plan motion. Kambhampati and Davis [Kamb86] have developed a multiresolution path-planning heuristic for two-dimensional motion using quadtrees that could easily be extended to three-dimensional motion using octrees. Fujimura and Samet [Fuji86b] use a similar approach to do path planning in the presence of moving obstacles.

Many graphics displays accept filled rectangles as a display primitive (e.g., [Whel82]) which means that the speed of displaying the quadtree is proportional to the number of nodes in the displayed region. On the other hand, pure raster displays would require the user to decompose the rectangle into pixels. A central goal that arises when designing graphics display primitives is to minimize the number of bits that need to be transferred while representing a given primitive. A general fill-rectangle primitive requires the specification of location, height, and width information in addition to color information. In contrast, a special purpose quadtree processor that expects a series of quadtree leaf nodes, only requires the specification of the width and color of the leaf nodes; the location can be derived from the position of the leaf in the list. Such an approach has been taken with at least one MC68000-based graphics display [Milf84, Will85]. A more aggressive approach to quadtree hardware is to design a parallel computer where individual processors are connected like nodes in a quadtree [Warn68, Linn73, Kush82, Dipp84]. One such device that has proven useful in image processing is the pyramid machine [Mill85]. Meagher [Meag84] describes an octree machine. Dew *et al.* [Dew85] discuss mapping an octree approach to CSG evaluation onto a systolic array computer. It should be noted that much of this work takes advantage of the interconnections within the hierarchy, but does not attempt to efficiently balance the workload among a restricted number of processors.

There still remain open questions about recursive hierarchical data structures for tasks in computer graphics. For example, the usage of quadtrees and octrees is often motivated by intuitive notions about the behavior of typical graphics data. However, this intuition still requires formalization. Furthermore, although much attention has been devoted to the development of hierarchical data structures, there has been relatively little work done comparing them. Comparisons based on more than a few "typical" examples would be a welcome contribution to this domain.

REFERENCES

1. [Abel84a] - D.J. Abel, A B^+-tree structure for large quadtrees, *Computer Vision, Graphics, and Image Processing 27*, 1(July 1984), 19-31.

2. [Abel84b] - D.J. Abel and J.L. Smith, A simple approach to the nearest-neighbor problem, *The Australian Computer Journal 16*, 4(November 1984), 140-146.

3. [Ahuj83] - N. Ahuja, On approaches to polygonal decomposition for hierarchical image representation, *Computer Vision, Graphics, and Image Processing 24*, 2(November 1983), 200-214.

4. [Ahuj84] - N. Ahuja and C. Nash, Octree representations of moving objects, *Computer Vision, Graphics, and Image Processing 26*, 2(May 1984), 207-216.

5. [ANSI85] - American National Standards Institute Committee X3H31, American National Standard for the Functional Specification of the Programmer's Hierarchical Interactive Graphics Standard (PHIGS), ANSI Standard X3H31/85-05 X3H3/85-21, American National Standards Institute, New York, February 85.

6. [Ande83] - D.P. Anderson, Techniques for reducing pen plotting time, *ACM Transactions on Graphics 2*, 3(July 1983), 197-212.

7. [Ayal85] - D. Ayala, P. Brunet, R. Juan, and I. Navazo, Object representation by means of nonminimal division quadtrees and octrees, *ACM Transactions on Graphics 4*, 1(January 1985), 41-59.

8. [Ball81] - D.H. Ballard, Strip trees: A hierarchical representation for curves, *Communications of the ACM 24*, 5(May 1981), 310-321 (see also corrigendum, *Communications of the ACM 25*, 3(March 1982), 213).

9. [Baum72] - B.G. Baumgart, Winged-edge polyhedron representation, STAN-CS-320, Computer Science Department, Stanford University, Palo Alto, CA, 1972.

10. [Bell83] - S.B.M. Bell, B.M. Diaz, F. Holroyd, and M.J. Jackson, Spatially referenced methods of processing raster and vector data, *Image and Vision Computing 1*, 4(November 1983), 211-220.

11. [Bres65] - J. E. Bresenham, Algorithm for computer control of digital plotter, *IBM Systems Journal 4*, 1(1965), 25-30.

12. [Brig85] - S. Bright and S. Laflin, Shading of solid voxel models, *Computer Graphics Forum 5*, (1986), 131-137.

13. [Carl85] - I. Carlbom, I. Chakravarty, and D. Vanderschel, A hierarchical data structure for representing the spatial decomposition of 3-D objects, *IEEE Computer Graphics and Applications 5*, 4(April 1985), 24-31.

14. [Carl82] - W.E. Carlson, An algorithm and data structure for 3D object synthesis

using surface patch intersections, *Computer Graphics 16,* 3(July 1982), 255-264 (also *Proceedings of the SIGGRAPH'82 Conference,* Boston, July 1982).

15. [Catm75] - E. Catmull, Computer display of curved surfaces, *Proceedings of the Conference on Computer Graphics, Pattern Recognition, and Data Structure,* Los Angeles, May 1975, 11-17.

16. [Chen85] - L.S. Chen, G.T. Herman, R.A. Reynolds, and J.K. Udupa, Surface shading in the cuberille environment, *IEEE Computer Graphics and Applications 5,* 12(December 1985), 33-41.

17. [Chie86] - C.H. Chien and J.K. Aggarwal, Identification of 3-d objects from multiple silhouettes using quadtrees/octrees, *Computer Vision, Graphics, and Image Processing 36,* 2/3(November/December 1986), 256-273.

18. [Clar76] - J. H. Clark, Hierarchical geometric models for visible surface algorithms, *Communications of the ACM 19,* 10(October 1976), 547-554.

19. [Cohe85] - M.F. Cohen and D.P. Greenberg, The hemi-cube, *Computer Graphics 19,* 3(July 1985), 31-40 (also *Proceedings of the SIGGRAPH'85 Conference,* San Francisco, July 1985).

20. [Cohe86] - M.F. Cohen, D.P. Greenberg, D.S. Immel, and P.J. Brock, An efficient radiosity approach for realistic image analysis, *IEEE Computer Graphics and Applications 6,* 3(March 1986), 26-35.

21. [Conn84] - C.I. Connolly, Cumulative generation of octree models from range data, *Proceedings of the International Conference on Robotics,* Atlanta, March 1984, 25-32.

22. [Dew85] - P.M. Dew. J. Dodsworth, and D.T. Morris, Systolic array architectures for high performance CAD/CAM workstations, in *Fundamental Algorithms for Computer Graphics,* R.A. Earnshaw, Ed., Springer-Verlag, Berlin, 1985, 659-694.

23. [Dipp84] - M. Dippe and J. Swensen, An adaptive subdivision algorithm and parallel architecture for realistic image synthesis, *Computer Graphics 18,* 3(July 1984), 149-158 (also *Proceedings of the SIGGRAPH'84 Conference,* Minneapolis, July 1984)

24. [Doct81] - L.J. Doctor and J.G. Torborg, Display techniques for octree-encoded objects, *IEEE Computer Graphics and Applications 1,* 1(July 1981), 39-46.

25. [Duff85] - T. Duff, Compositing 3-D rendered images, *Computer Graphics 19,* 3(July 1985), 41-44 (also *Proceedings of the SIGGRAPH'85 Conference,* San Francisco, July 1985).

26. [Dyer80] - C.R. Dyer, A. Rosenfeld, and H. Samet, Region representation: boundary codes from quadtrees, *Communications of the ACM 23,* 3(March 1980), 171-179.

27. [Dyer82] - C.R. Dyer, The space efficiency of quadtrees, *Computer Graphics and Image Processing 19,* 4(August 1982), 335-348.

28. [Faug84] - O.D. Faugeras, M. Hebert, P. Mussi, and J.D. Boissonnat, Polyhedral approximation of 3-d objects without holes, *Computer Vision, Graphics, and Image Processing 25,* 2(February 1984), 169-183.

29. [Free74] - H. Freeman, Computer processing of line-drawing images, *ACM Computing Surveys 6,* 1(March 1974), 57-97.

30. [Fuch83] - H. Fuchs, G.D. Abram, and E.D. Grant, Near real-time shaded display of rigid objects, *Computer Graphics 17,* 3(July 1983), 65-72 (also *Proceedings of the SIGGRAPH'83 Conference*, Detroit, July 1983).

31. [Fuji86a] - A. Fujimoto, T. Tanaka, and K. Iwata, ARTS: Accelerated ray-tracing system, *IEEE Computer Graphics and Applications 6,* 4(April 1986), 16-26.

32. [Fuji85] - K. Fujimura and T.L. Kunii, A hierarchical space indexing method, *Proceedings of Computer Graphics'85*, Tokyo, 1985, T1-4, 1-14.

33. [Fuji86b] - K. Fujimura and H. Samet, A hierarchical strategy for path planning among moving obstacles, Computer Science TR-1736, University of Maryland, College Park, MD, November 1986.

34. [Garg82a] - I. Gargantini, An effective way to represent quadtrees, *Communications of the ACM 25,* 12(December 1982), 905-910.

35. [Garg82b] - I. Gargantini, Linear octtrees for fast processing of three dimensional objects, *Computer Graphics and Image Processing 20,* 4(December 1982), 365-374.

36. [Garg82c] - I. Gargantini, Detection of connectivity for regions represented by linear quadtrees, *Computers and Mathematics with Applications 8,* 4(1982), 319-327.

37. [Garg83] - I. Gargantini, Translation, rotation, and superposition of linear quadtrees, *International Journal of Man-Machine Studies 18,* 3(March 1983), 253-263.

38. [Gibs82] - L. Gibson and D. Lucas, Vectorization of raster images using hierarchical methods, *Computer Graphics and Image Processing 20,* 1(September 1982), 82-89.

39. [Gill81] - R. Gillespie and W.A. Davis, Tree data structures for graphics and image processing, *Proceedings of the Seventh Conference of the Canadian Man-Computer Communications Society,* Waterloo, Canada, June 1981, 155-161.

40. [Glas84] - A.S. Glassner, Space subdivision for fast ray tracing, *IEEE Computer Graphics and Applications 4,* 10(October 1984), 15-22.

41. [Gora84] - C.M. Goral, K.E. Torrance, D.P. Greenberg, and B. Battaile, Modeling the interaction of light between diffuse surfaces, *Computer Graphics 18,* 3(July 1984), 213-222 (also *Proceedings of the SIGGRAPH'84 Conference*, Minneapolis, July 1984).

42. [Gord85] - D. Gordon and R. A. Reynolds, Image space shading of three-dimensional objects, *Computer Vision, Graphics, and Image Processing 29,* 3(March 1985), 361-376.

43. [Gray67] - J.C. Gray, Compound data structure for computer aided design: a survey, *Proceedings of the 22nd National Conference of the ACM*, 1967, 355-365.

44. [Hill83] - F.S. Hill, Jr. , S. Walker, Jr., and F. Gao, Interactive image query system using progressive transmission, *Computer Graphics 17*, 3(July 1983), 323-330 (also *Proceedings of the SIGGRAPH'83 Conference*, Boston, July 1983).

45. [Hunt78] - G.M. Hunter, Efficient computation and data structures for graphics, Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.

46. [Hunt79a] - G.M. Hunter and K. Steiglitz, Operations on images using quad trees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 2(April 1979), 145-153.

47. [Hunt79b] - G.M. Hunter and K. Steiglitz, Linear transformation of pictures represented by quad trees, *Computer Graphics and Image Processing 10*, 3(July 1979), 289-296.

48. [Imme86] - D.S. Immel, M.F. Cohen, and D.P. Greenberg, A radiosity method for non-diffuse environments, *Computer Graphics 20*, 4(August 1986), pp. 133-142 (also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, August 1986).

49. [Jack80] - C.L. Jackins and S.L. Tanimoto, Oct-trees and their use in representing three-dimensional objects, *Computer Graphics and Image Processing 14*, 3(November 1980), 249-270.

50. [Jack83] - C.L. Jackins and S.L. Tanimoto, Quad-trees, oct-trees, and k-trees - a generalized approach to recursive decomposition of Euclidean space, *IEEE Transactions on Pattern Analysis and Machine Intelligence 5*, 5(September 1983), 533-539.

51. [Jans86] - F.W. Jansen, Data structures for ray tracing, *Data Structures for Raster Graphics* (F.J. Peters, L.R.A. Kessener, and M.L.P. van Lierop, Eds.), Springer Verlag, Berlin, 1986, 57-73.

52. [Javi76] - J.F. Javis, C.N. Judice, and W.H. Ninke, A survey of techniques for the image display of continuous tone images on a bilevel display, *Computer Graphics and Image Processing 5*, 1(March 1976), 13-40.

53. [Kamb86] - S. Kambhampati and L.S. Davis, Multiresolution path planning for mobile robots, *IEEE Journal of Robotics and Automation 2*, 3(September 1986), 135-145.

54. [Kapl85] - M.R. Kaplan, Space-tracing: a constant time ray-tracer, SIGGRAPH'85 Tutorial on the Uses of Spatial Coherence in Ray-Tracing, San Francisco, ACM, July 1985.

55. [Kauf83] - A. Kaufman, D. Forgash, and Y. Ginsburg, Hidden surface removal using a forest of quadtrees, *Proceedings of Conference on Image Processing, Computer Graphics, and Pattern Recognition*, Beer-Sheva, Israel, June 1983, 85-89.

56. [Kawa80a] - E. Kawaguchi and T. Endo, On a method of binary picture representation and its application to data compression, *IEEE Transactions on Pattern Analysis and Machine Intelligence 2*, 1(January 1980), 27-35.

57. [Kawa80b] - E. Kawaguchi, T. Endo, and M. Yokota, DF-expression of binary-valued picture and its relation to other pyramidal representations, *Proceedings of the Fifth International Conference on Pattern Recognition*, Miami Beach, December 1980, 822-827.

58. [Kawa83] - E. Kawaguchi, T. Endo, and J. Matsunaga, Depth-first expression viewed from digital picture processing, *IEEE Transactions on Pattern Analysis and Machine Intelligence 5*, 4(July 1983), 373-384.

59. [Klin71] - A. Klinger, Patterns and Search Statistics, in *Optimizing Methods in Statistics*, J.S. Rustagi, Ed., Academic Press, New York, 1971, 303-337.

60. [Klin79] - A. Klinger and M.L. Rhodes, Organization and access of image data by areas, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 1(January 1979), 50-60.

61. [Know80] - K. Knowlton, Progressive transmission of grey-scale and binary pictures by simple, efficient, and lossless encoding schemes, *Proceedings of the IEEE 68*, 7(July 1980), 885-896.

62. [Knut75] - D.E. Knuth, *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1975.

63. [Kois85] - P. Koistinen, M. Tamminen, and H. Samet, Viewing solid models by bintree conversion, *Proceedings of the EUROGRAPHICS'85 Conference*, C.E. Vandoni, Ed., North-Holland, 1985, 147-157.

64. [Kush82] - T. Kushner, A. Wu, and A. Rosenfeld, Image processing on ZMOB, *IEEE Transactions on Computers 31*, 10(October 1982), 943-951.

65. [Lant84] - K.A. Lantz and W.I. Nowicki, Structured graphics for distributed systems, *ACM Transactions on Graphics 3*, 1(January 1984), 23-51.

66. [Lauz85] - J.P. Lauzon, D.M. Mark, L. Kikuchi, and J.A. Guevara, Two-dimensional run-encoding for quadtree representation, *Computer Vision, Graphics, and Image Processing 30*, 1(April 1985), 56-69.

67. [Linn73] - J. Linn, General methods for parallel searching, Technical Report 81, Digital Systems Laboratory, Stanford University, Stanford, CA, May 1973.

68. [Meag80] - D. Meagher, Octree encoding: a new technique for the representation, the manipulation, and display of arbitrary 3-d objects by computer, Technical Report IPL-TR-80-111, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, New York, October 1980.

69. [Meag82] - D. Meagher, Geometric modeling using octree encoding, *Computer Graphics and Image Processing 19*, 2(June 1982), 129-147.

70. [Meag84] - D. Meagher, The Solids Engine: a processor for interactive solid modeling, *Proceedings of the NICOGRAPH '84 Conference*, Tokyo, November 1984.

71. [Milf84] - D.J. Milford and P.C. Willis, Quad encoded display, *IEE Proceedings 131*, E3(May 1984), 70-75.

72. [Mill85] - R. Miller and Q.F. Stout, Pyramid computer algorithms for determining geometric properties of images, *Proceedings of the Symposium on Computational Geometry*, Baltimore, June 1985, 263-269.

73. [Morr85] - D.T. Morris and P. Quarendon, An algorithm for direct display of CSG objects by spatial subdivision, *Fundamental Algorithms for Computer Graphics*, R.A. Earnshaw, Ed., Springer-Verlag, Berlin, 1985, 725-736.

74. [Mort85] - M.E. Mortenson, *Geometric Modeling*, John Wiley and Sons, New York, 1985.

75. [Mort66] - G.M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, IBM Ltd., Ottawa, Canada, 1966.

76. [Nava86a] - I. Navazo, Contribucio a les tecniques de modelat geometric d'objectes poliedrics usant la codificacio amb arbres octals, Ph.D. dissertation, Escola Tecnica Superior d'Enginyers Industrials, Department de Metodes Informatics, Universitat Politechnica de Barcelona, Barcelona, Spain, January 1986.

77. [Nava86b] - I. Navazo, D. Ayala, and P. Brunet, A geometric modeller based on the exact octree representation of polyhedra, Escola Tecnica Superior d'Enginyers Industrials, Department de Metodes Informatics, Universitat Politechnica de Barcelona, Barcelona, Spain, January 1986.

78. [Nels86] - R.C. Nelson and H. Samet, A consistent hierarchical representation for vector data, *Computer Graphics 20*, 4(August 1986), 197-206 (also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, August 1986).

79. [Oliv83a] - M.A. Oliver and N.E. Wiseman, Operations on quadtree-encoded images, *Computer Journal 26*, 1(February 1983), 83-91.

80. [Oliv83b] - M.A. Oliver and N.E. Wiseman, Operations on quadtree leaves and related image areas, *Computer Journal 26*, 4(November 1983), 375-380.

81. [Pete85] - F.J. Peters, An algorithm for transformations of pictures represented by quadtrees, *Computer Vision, Graphics, and Image Processing 32*, 3(December 1985), 397-403.

82. [Posd82] - J.L. Posdamer, Spatial sorting for sampled surface geometries, *Proceedings of SPIE - Biostereometrics'82 361*, San Diego, August 1982.

83. [Redd78] - D.R. Reddy and S. Rubin, Representation of three-dimensional objects, CMU-CS-78-113, Computer Science Department, Carnegie-Mellon University, Pittsburgh, April 1978.

84. [Requ80] - A.A.G. Requicha, Representations of rigid solids: theory, methods, and systems, *ACM Computing Surveys 12*, 4(December 1980), 437-464.

85. [Roge85] - D.R. Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company, New York, NY, 1985.

86. [Rose82] - A. Rosenfeld, H. Samet, C. Shaffer, and R.E. Webber, Application of hierarchical data structures to geographical information systems, Computer Science TR-1197, University of Maryland, College Park, MD, June 1982.

87. [Rose83] - A. Rosenfeld, H. Samet, C. Shaffer, and R.E. Webber, Application of hierarchical data structures to geographical information systems phase II, Computer Science TR-1327, University of Maryland, College Park, MD, September 1983.

88. [Rubi80] - S.M. Rubin and T. Whitted, A 3-dimensional representation for fast rendering of complex scenes, *Computer Graphics 14*, 3(July 1980), 110-116 (also *Proceedings of the SIGGRAPH'80 Conference*, Seattle, July 1980).

89. [Ruto68] - D. Rutovitz, Data structures for operations on digital images, in *Pictorial Pattern Recognition*, G.C. Cheng et al., Eds., Thompson Book Co., Washington D.C., 1968, 105-133.

90. [Same80a] - H. Samet, Region representation: quadtrees from boundary codes, *Communications of the ACM 23*, 3(March 1980), 163-170.

91. [Same80b] - H. Samet, Region representation: quadtrees from binary arrays, *Computer Graphics and Image Processing 13*, 1(May 1980), 88-93.

92. [Same81a] - H. Samet, An algorithm for converting rasters to quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 3*, 1(January 1981), 93-95.

93. [Same81b] - H. Samet, Connected component labeling using quadtrees, *Journal of the ACM 28*, 3(July 1981), 487-501.

94. [Same82a] - H. Samet, Neighbor finding techniques for images represented by quadtrees, *Computer Graphics and Image Processing 18*, 1(January 1982), 37-57.

95. [Same82b] - H. Samet and R.E. Webber, On encoding boundaries with quadtrees, Computer Science TR-1162, University of Maryland, College Park, MD, February 1982.

96. [Same84a] - H. Samet, Algorithms for the conversion of quadtrees to rasters, *Computer Vision, Graphics, and Image Processing 26*, 1(April 1984), 1-16.

97. [Same84b] - H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984), 187-260.

98. [Same84c] - H. Samet, A. Rosenfeld, C.A. Shaffer, R.C. Nelson, and Y.G. Huang, Application of hierarchical data structures to geographical information systems Phase III, Computer Science TR-1457, University of Maryland, College Park, MD, November 1984.

99. [Same84d] - H. Samet, A. Rosenfeld, C.A. Shaffer, and R.E. Webber, A geographic information system using quadtrees, *Pattern Recognition 17,* 6 (November/December 1984), 647-656.

100. [Same85a] - H. Samet, A top-down quadtree traversal algorithm, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7,* 1(January 1985), 94-98.

101. [Same85b] - H. Samet and M. Tamminen, Efficient component labeling of images of arbitrary dimension, Computer Science TR-1480, University of Maryland, College Park, MD, February 1985.

102. [Same85c] - H. Samet and R.E. Webber, Storing a collection of polygons using quadtrees, *ACM Transactions on Graphics 4,* 3(July 1985), 182-222 (also *Proceedings of Computer Vision and Pattern Recognition 83,* Washington, DC, June 1983, 127-132; and University of Maryland Computer Science TR-1372).

103. [Same85d] - H. Samet and M. Tamminen, Bintrees, CSG trees, and time, *Computer Graphics 19,* 3(July 1985), 121-130 (also *Proceedings of the SIGGRAPH'85 Conference,* San Francisco, July 1985; and University of Maryland Computer Science TR-1472).

104. [Same85e] - H. Samet, Data structures for quadtree approximation and compression, *Communications of the ACM 28,* 9(September 1985), 973-993 (also University of Maryland Computer Science TR-1209).

105. [Same85f] - H. Samet and C.A. Shaffer, A model for the analysis of neighbor finding in pointer-based quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7,* 6(November 1985), 717-720 (also University of Maryland Computer Science TR-1432).

106. [Same85g] - H. Samet, A. Rosenfeld, C.A. Shaffer, R.C. Nelson, Y-G. Huang, and K. Fujimura, Application of hierarchical data structures to geographic information systems: phase IV, Computer Science TR-1578, University of Maryland, College Park, MD, December 1985.

107. [Same86a] - H. Samet, C.A. Shaffer, and R.E. Webber, Digitizing the plane with cells of non-uniform size, to appear in *Information Processing Letters* (also University of Maryland Computer Science TR-1619).

108. [Same86b] - H. Samet, Bibliography on quadtrees and related hierarchical data structures, in *Data Structures for Raster Graphics,* F.J. Peters, L.R.A. Kessener, and M.L.P. van Lierop, Eds., Springer Verlag, Berlin, 1986, 181-201.

109. [Same86c] - H. Samet and M. Tamminen, A general approach to connected component labeling of images, Compute Science TR-1649, University of Maryland, College Park, MD, August 1986 (see also *Proceedings of the IEEE Computer Vision and Pattern*

Recognition Conference, Miami Beach, June 1986, 312-318).

110. [Same86d] - H. Samet and R.E. Webber, A comparison of the space requirements of multi-dimensional quadtree-based file structures Computer Science TR-1711, University of Maryland, College Park, MD, September 1986.

111. [Shaf87] - C.A. Shaffer and H. Samet, Optimal quadtree construction algorithms, to appear in *Computer Vision, Graphics, and Image Processing.*

112. [Shne81] - M. Shneier, Two hierarchical linear feature representations: edge pyramids and edge quadtrees, *Computer Graphics and Image Processing 17*, 3(November 1981), 211-224.

113. [Simo69] - H.A. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, MA, 1969.

114. [Sloa79] - K.R. Sloan Jr. and S.L. Tanimoto, Progressive refinement of raster images, *IEEE Transactions on Computers 28*, 11(November 1979), 871-874.

115. [Suth63] - I.E. Sutherland, Sketchpad, a man-machine communication system, *Proceedings of the Spring Joint Computer Conference*, Detroit, MI, May 1963, 329-346.

116. [Suth74] - I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, A characterization of ten hidden-surface algorithms, *ACM Computing Surveys 6*, 1(March 1974), 1-55.

117. [Tamm84a] - M. Tamminen, Comment on quad- and octtrees, *Communications of the ACM 27*, 3(March 1984), 248-249.

118. [Tamm84b] - M. Tamminen and H. Samet, Efficient octree conversion by connectivity labeling, *Computer Graphics 18*, 3(July 1984), pp. 43-51 (also *Proceedings of the SIGGRAPH'84 Conference*, Minneapolis, July 1984).

119. [Tarj75] - R.E. Tarjan, Efficiency of a good but not linear set union algorithm, *Journal of the ACM 22*, 2(April 1975), 215-225

120. [vanL84] - M.L.P. van Lierop, Transformations on pictures represented by leafcodes, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1984.

121. [Veen85] - J. Veenstra and N. Ahuja, Octree generation from silhouette views of an object, *Proceedings of the International Conference on Robotics*, St. Louis, March 1985, 843-848.

122. [Warn68] - J.E. Warnock, A hidden line algorithm for halftone picture representation, Computer Science Department TR 4-5, University of Utah, Salt Lake City, May 1968.

123. [Warn69a] - J.E. Warnock, The hidden line problem and the use of halftone displays, in *Pertinent Concepts in Computer Graphics - Proceedings of the Second*

University of Illinois Conference on Computer Graphics, M. Faiman and J. Nievergelt, Eds., University of Illinois Press, Urbana, Illinois, March 1969, 154-163.

124. [Warn69b] - J.E. Warnock, A hidden surface algorithm for computer generated half tone pictures, Computer Science Department TR 4-15, University of Utah, Salt Lake City, June 1969.

125. [Wats84] - D.F. Watson, and G.M. Philip, Systematic triangulations, *Computer Vision, Graphics, and Image Processing 26*, 2(May 1984), 217-223.

126. [Webb84] - R.E. Webber, Analysis of quadtree algorithms, Ph.D. dissertation, TR-1376, Computer Science Department, University of Maryland, College Park, MD, March 1984.

127. [Wegh84] - H. Weghorst, G. Hooper, and D.P. Greenberg, Improved computational methods for ray tracing, *ACM Transactions on Graphics 3*, 1(January 1984), 52-69.

128. [Weil77] - K. Weiler and P. Atherton, Hidden surface removal using polygon area sorting, *Computer Graphics 11*, 2(Summer 1977), 214-222 (also *Proceedings of the SIGGRAPH'77 Conference*, San Jose, California, July 1977).

129. [Whel82] - D.S. Whelan, A rectangular array filling display system architecture, *Computer Graphics 16*, 3(July 1982), 147-153 (also *Proceedings of the SIGGRAPH'82 Conference*, Boston, July 1982).

130. [Whit80] - T. Whitted, An improved illumination model for shaded display, *Communications of the ACM 23*, 6(June 1980), 343-349.

131. [Will85] - P. Willis and D. Milford, Browsing high definition color pictures, *Computer Graphics Forum 4*, (1985), 203-208.

132. [Wood82] - J.R. Woodwark and K.M. Quinlan, Reducing the effect of complexity on volume model evaluation, *Computer-aided Design 14*, 2(1982), 89-95.

133. [Wyvi85] - G. Wyvill and T.L. Kunii, A functional model for constructive solid geometry, *The Visual Computer 1*, 1(July 1985), 3-14.

134. [Yama84] - K. Yamaguchi, T.L. Kunii, K. Fujimura, and H. Toriya, Octree-related data structures and algorithms, *IEEE Computer Graphics and Applications 4*, 1(January 1984), 53-59.

135. [Yau83] - M. Yau and S.N. Srihari, A hierarchical data structure for multidimensional digital images, *Communications of the ACM 26*, 7(July 1983), 504-515.

136. [Yau84] - M. Yau, Generating quadtrees of cross-sections from octrees, *Computer Vision, Graphics, and Image Processing 27*, 2(August 1984), 211-238.

137. [Yerr83] - M.A. Yerry and M.S. Shepard, A modified quadtree approach to finite element mesh generation, *IEEE Computer Graphics and Applications 3*, 1(January/February 1983), 39-46.

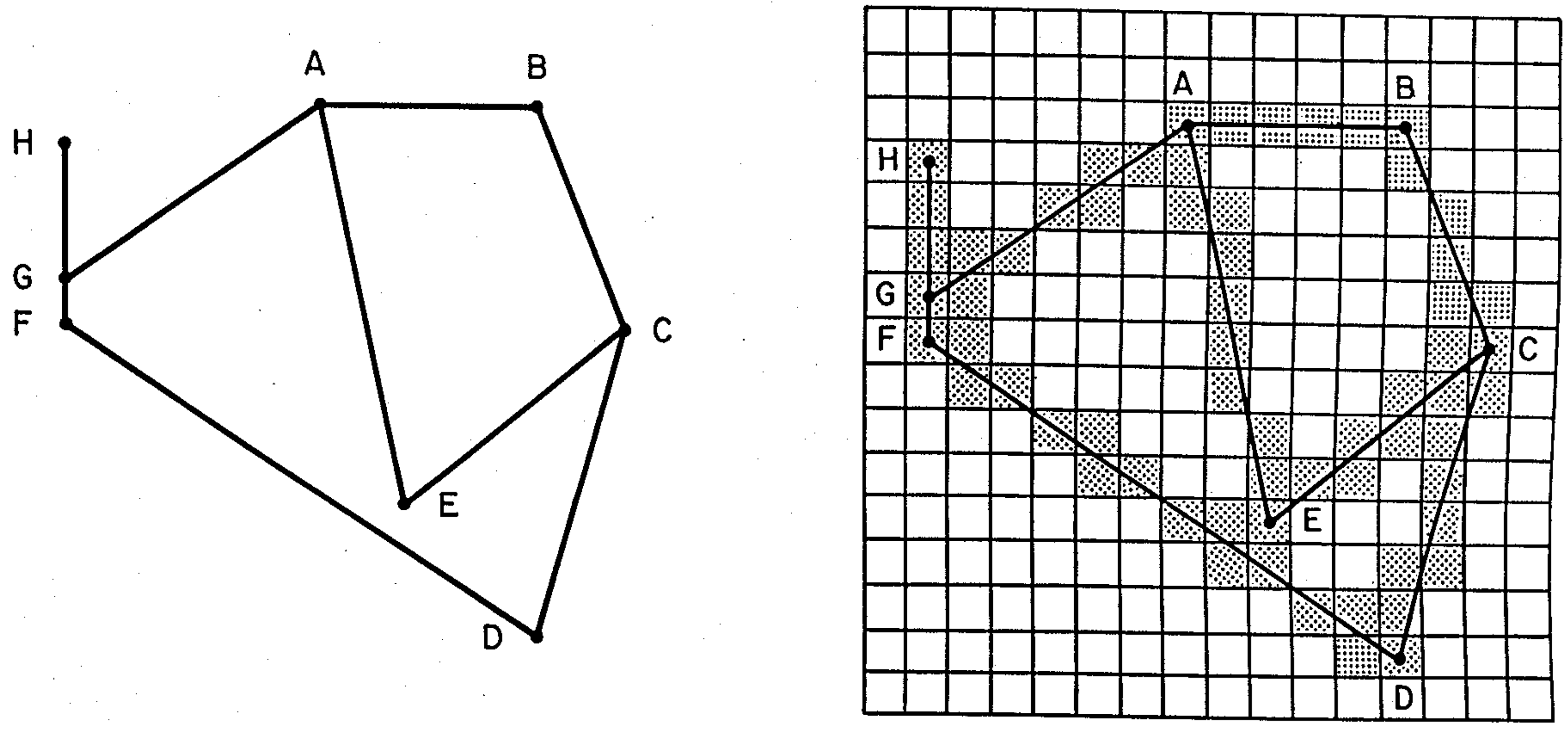


Figure 1. Example image represented in (a) a vector data format; and (b) a raster data format.

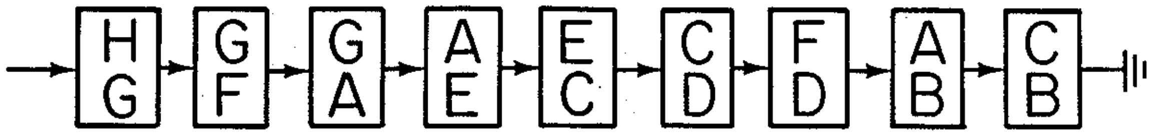


Figure 2. Linked list of records representing, by pairs of their endpoints, the line segments of Figure 1a.

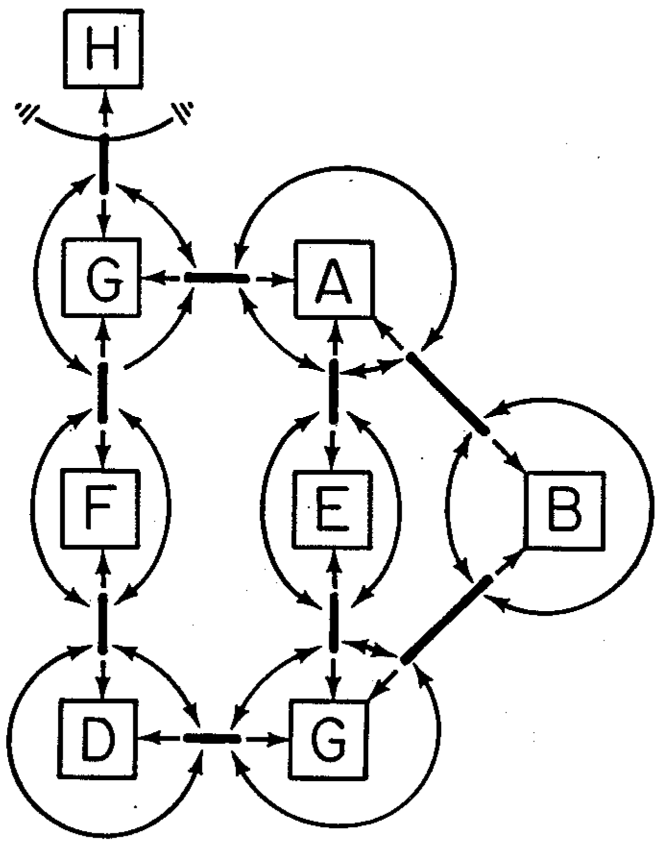
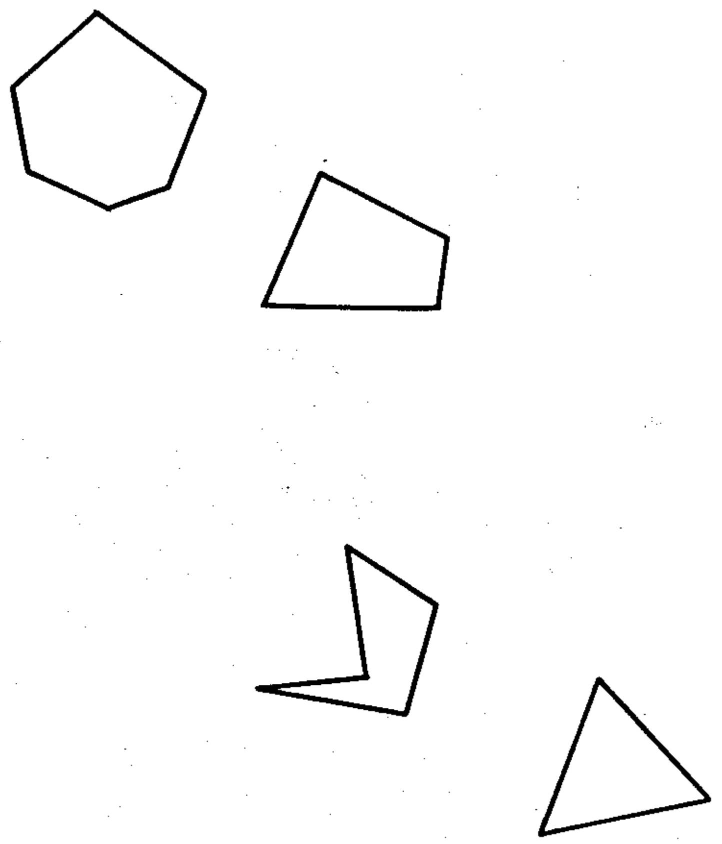
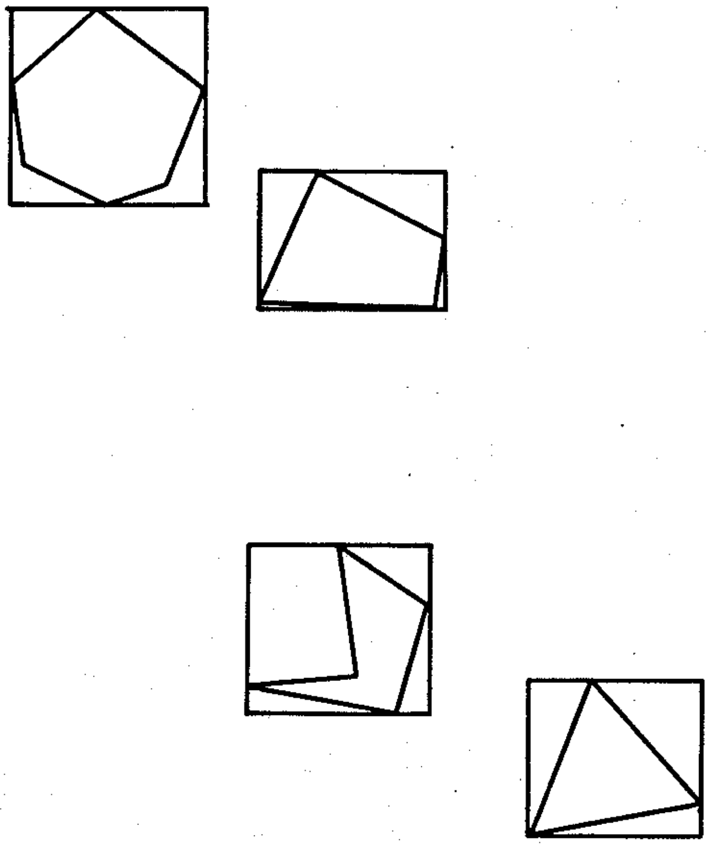


Figure 3. Winged-edge representation of the line segments and their endpoints of Figure 1a. The result is a graph with two types of nodes shown as squares and narrow solid rectangles. The squares correspond to endpoints of line segments while the rectangles correspond to the actual line segments. Each arrow denotes an edge in the graph between two nodes. Edges can exist between two line segments and also from line segments to their endpoints.

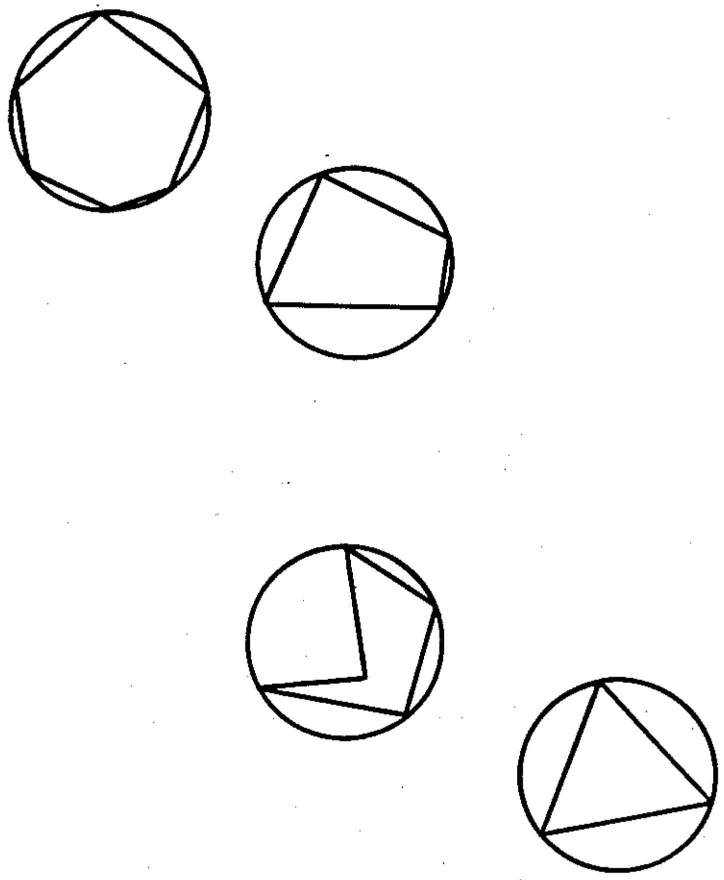


(a) Unbounded objects for use in parts (b)-(f).

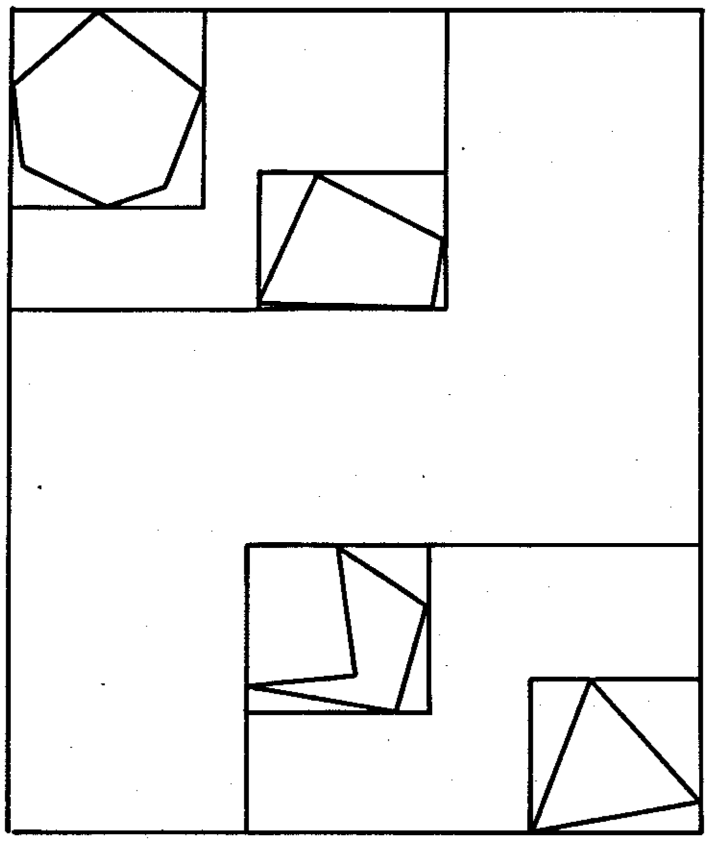
Figure 4. An example of the use of bounding objects.



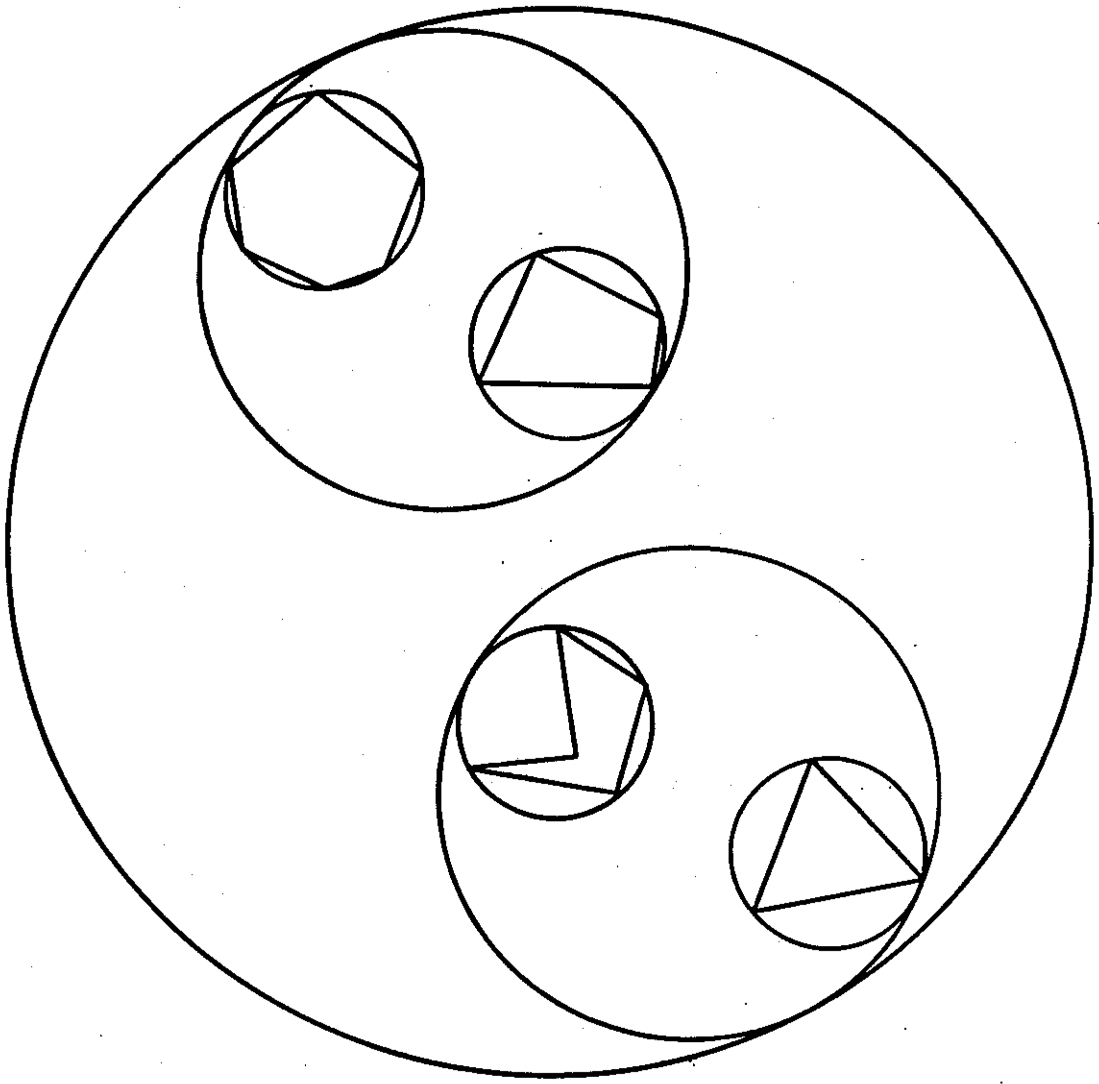
(b) Bounding boxes.



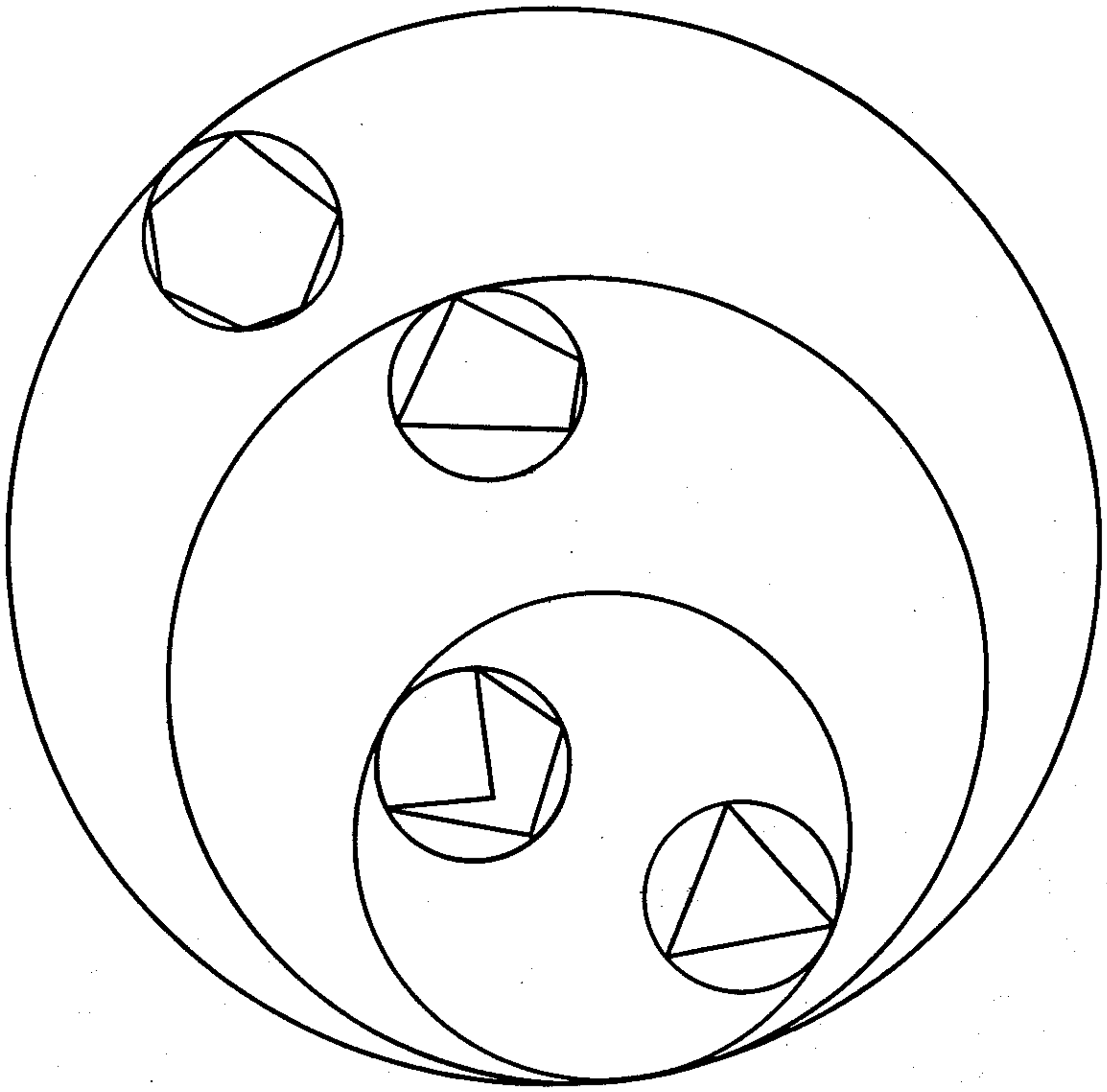
(c) Bounding circles.



(d) Hierarchical bounding boxes.



(e) Hierarchical bounding circles that are balanced.



(f) Unbalanced bounding circles.

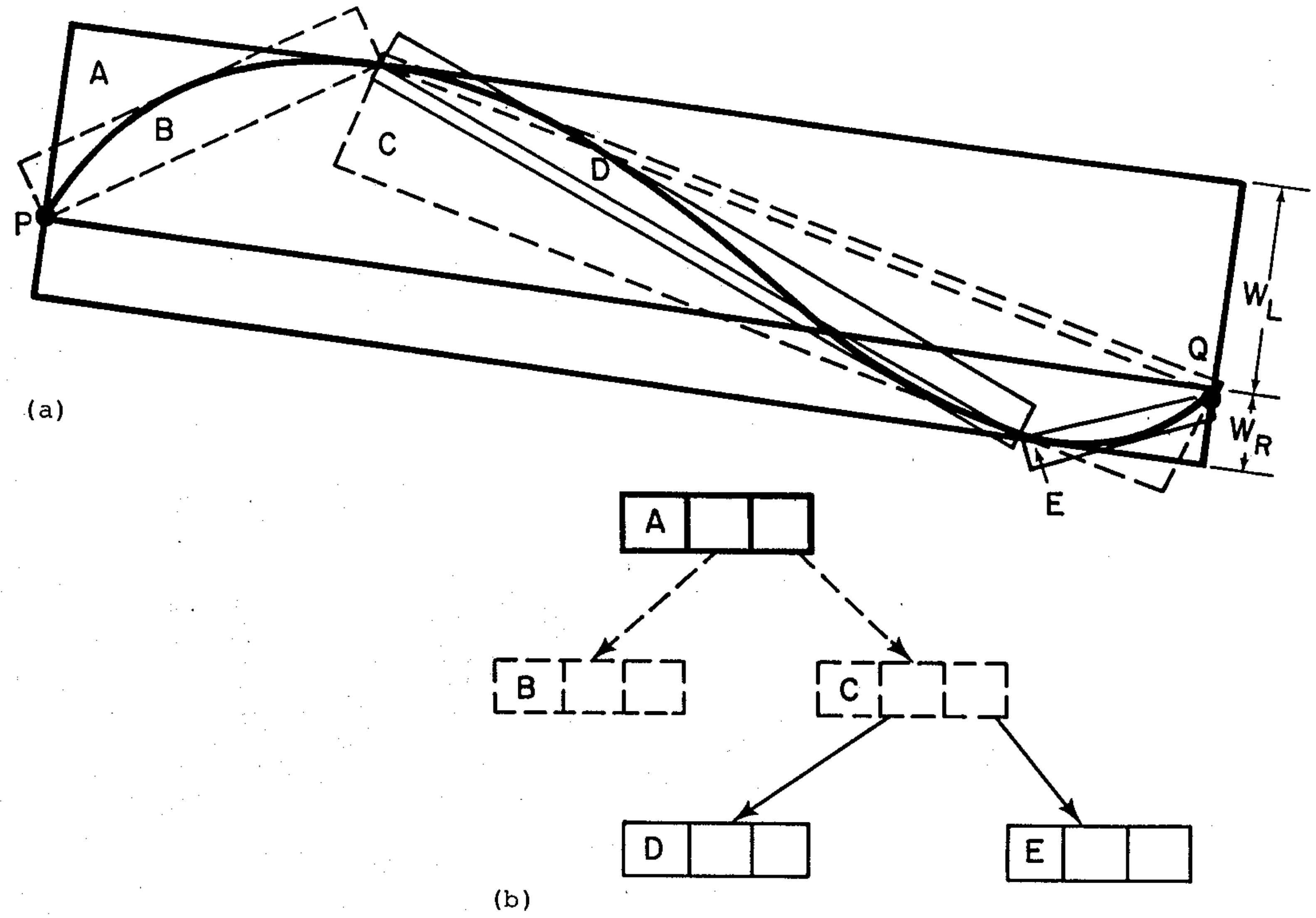
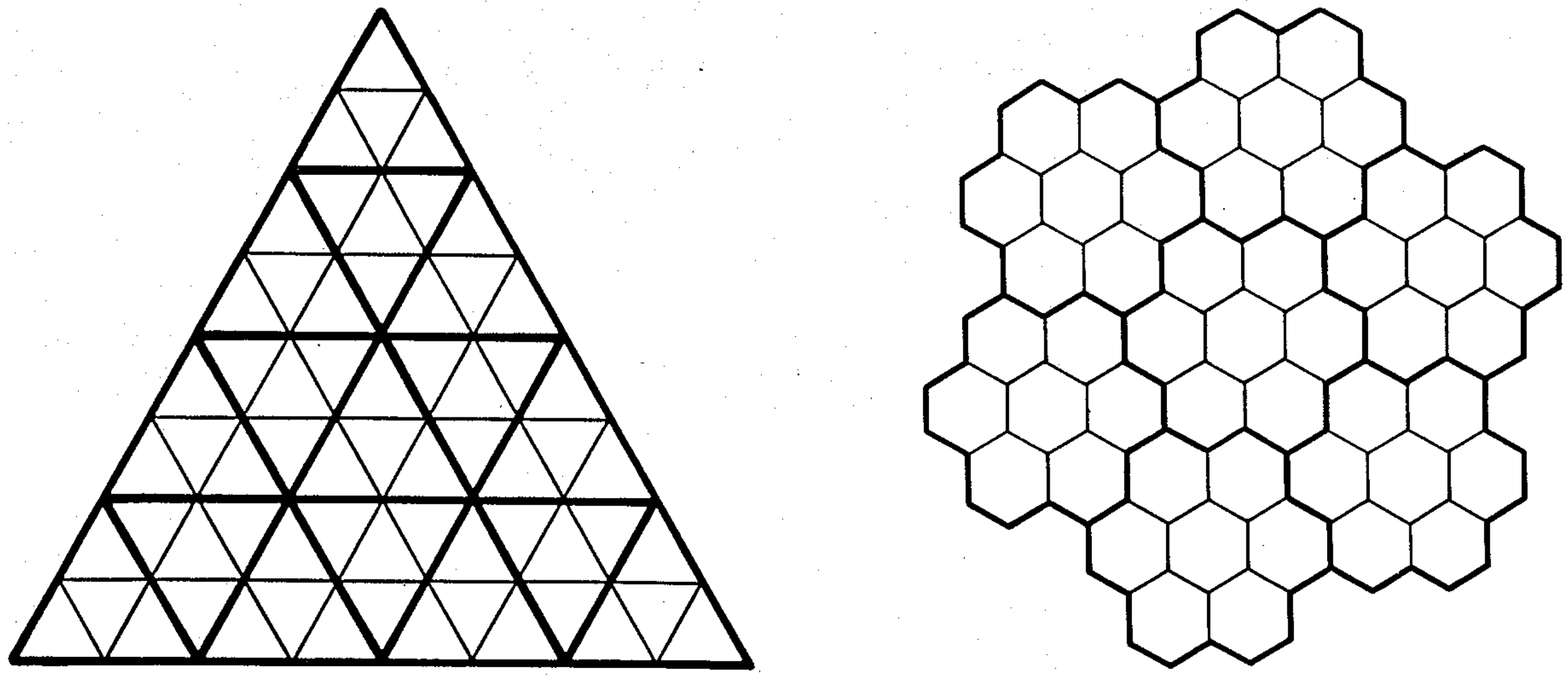
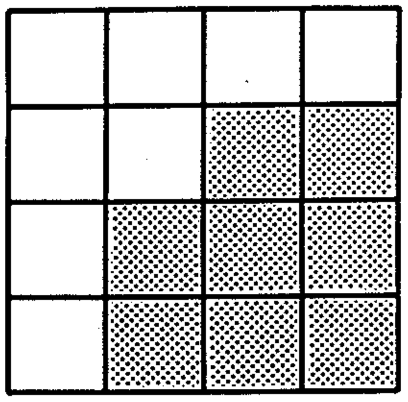


Figure 5. A curve between points P and Q. (a) Its decomposition into a hierarchy of strips and (b) the corresponding strip tree.

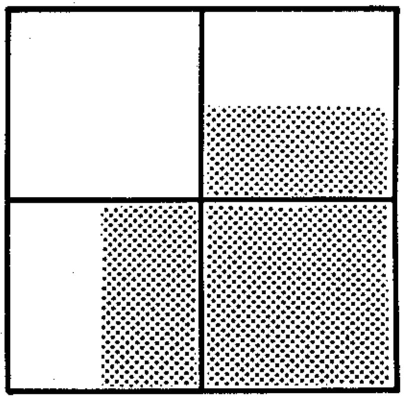

(a)

(b)

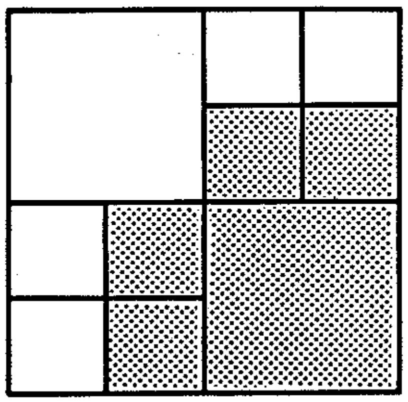
Figure 6. Examples of non-square partitionings of the plane.
(a) Equilateral triangles.
(b) Hexagons.



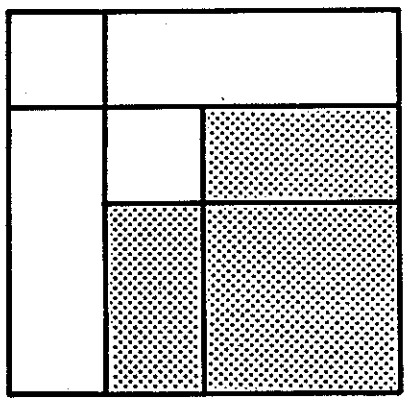
(a) Original image.



(b) First level of decomposition.



(c) Second and final level of decomposition.



(d) Example of an irregular decomposition.

Figure 7. Illustration of the quadtree decomposition process.

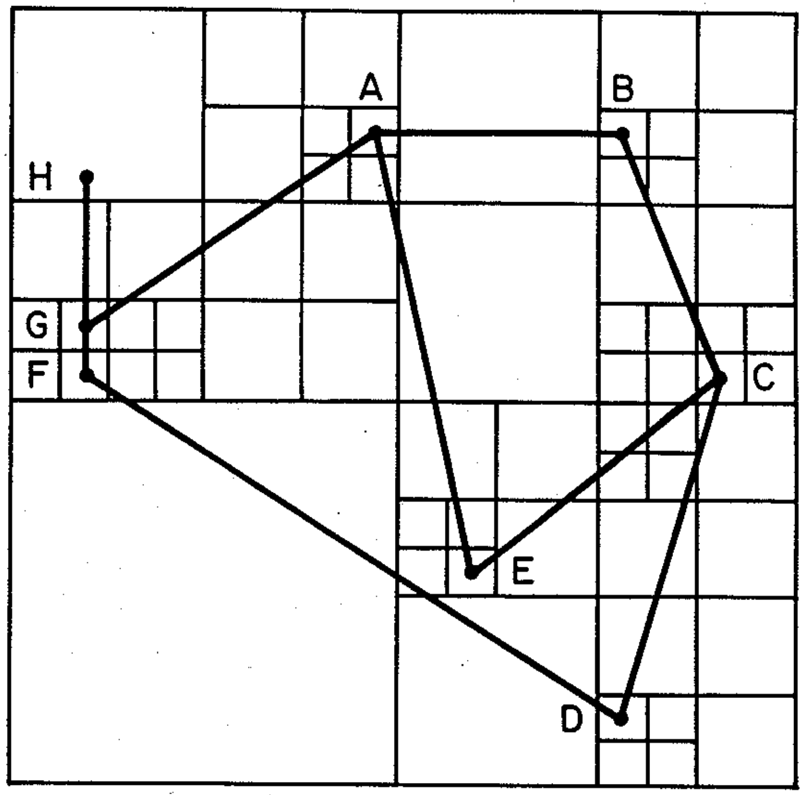
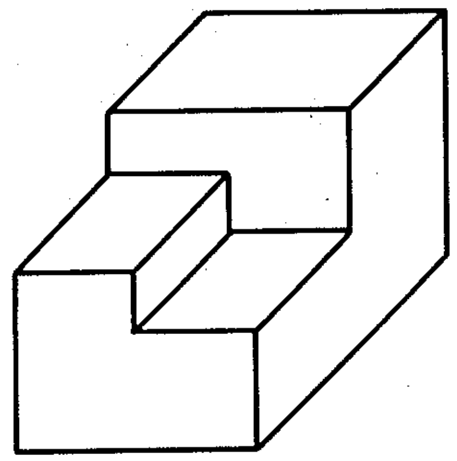
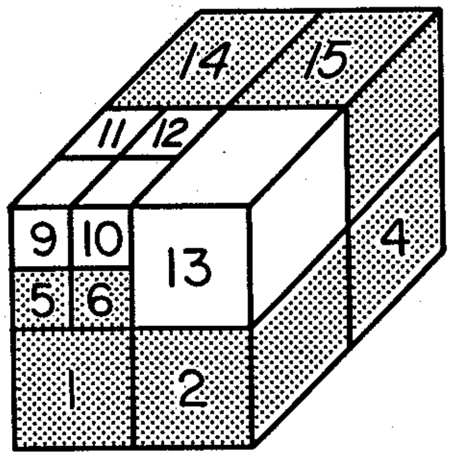
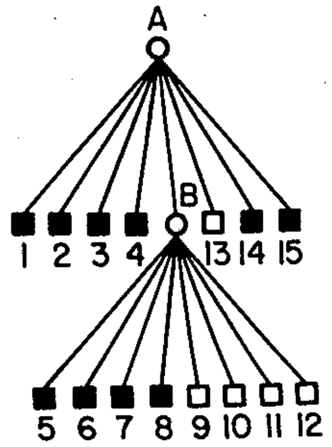


Figure 8. The edge quadtree for the vector data of Figure 1a. The maximum level of decomposition is 4.


(a)



(b)



(c)

Figure 9. (a) Example three-dimensional object; (b) its octree block decomposition; and (c) its tree representation.

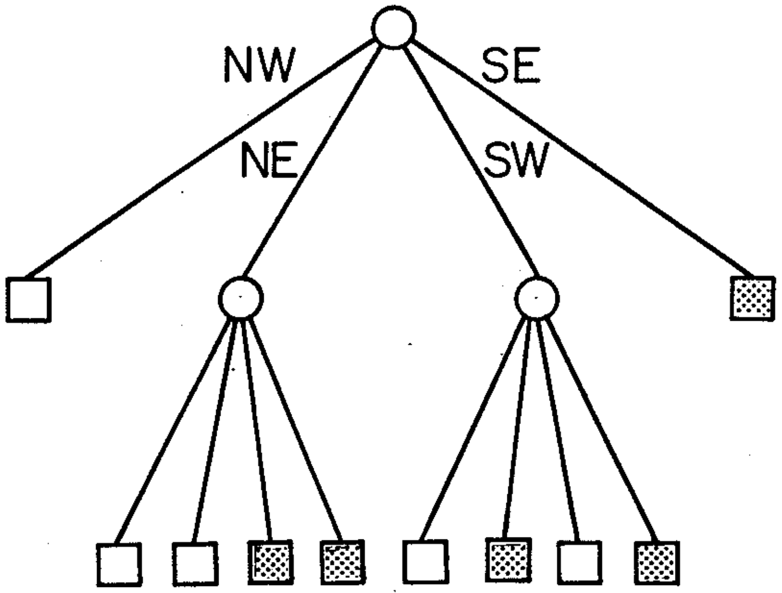


Figure 10. Pointer encoding of the quadtree of Figure 7. Internal nodes are represented by circular nodes. Terminal nodes are represented by square nodes whose contents correspond to the blocks in Figure 7.

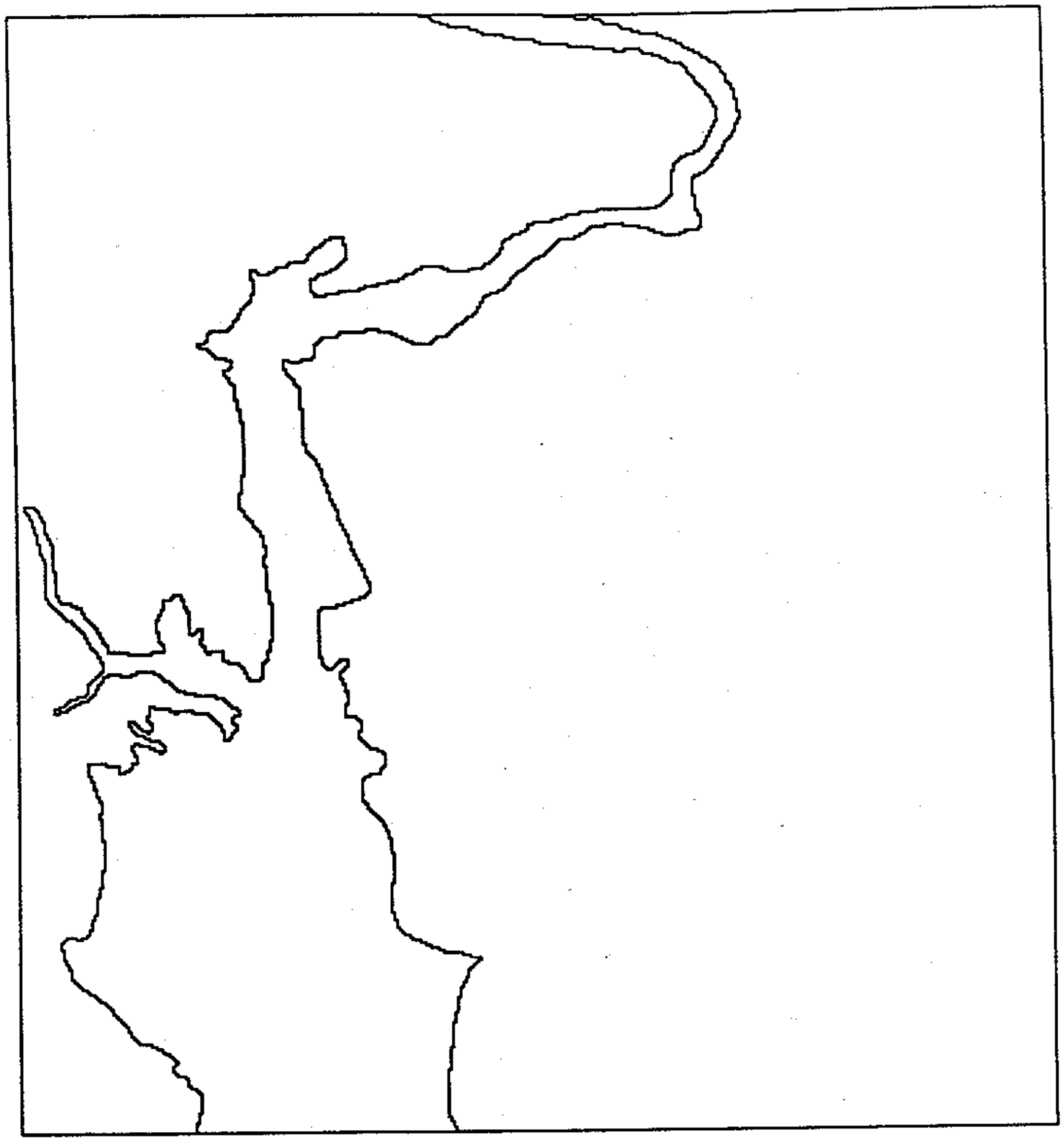


Figure 11. An example map whose encoding using a pointer quadtree is more efficient than its encoding using a linear quadtree.

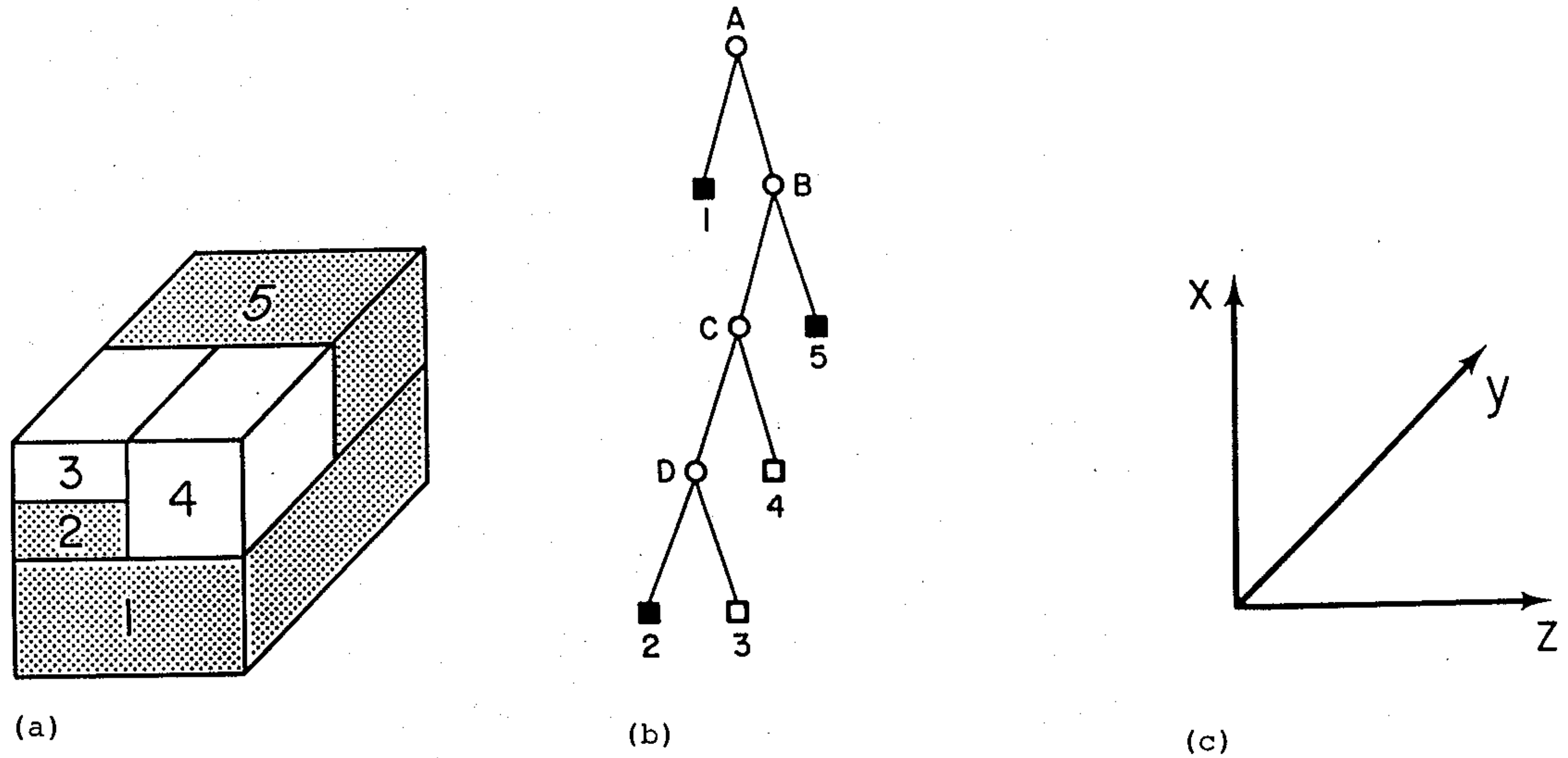


Figure 12. (a) Block decomposition and (b) tree representation of the three-dimensional bintree corresponding to the object of Figure 9 when using the octree coordinate system of (c).

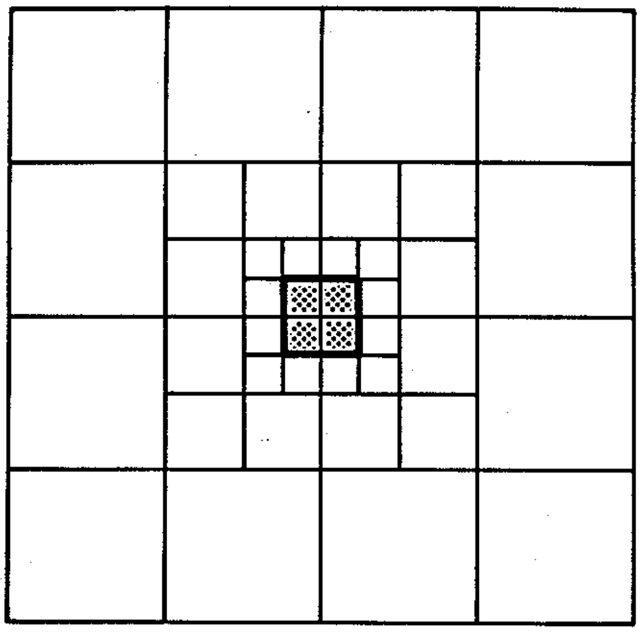


Figure 13. Example quadtree where the perimeter does not exceed the base 2 logarithm of the width of the image. The region in the image is assumed to consist of four pixels each of unit width.

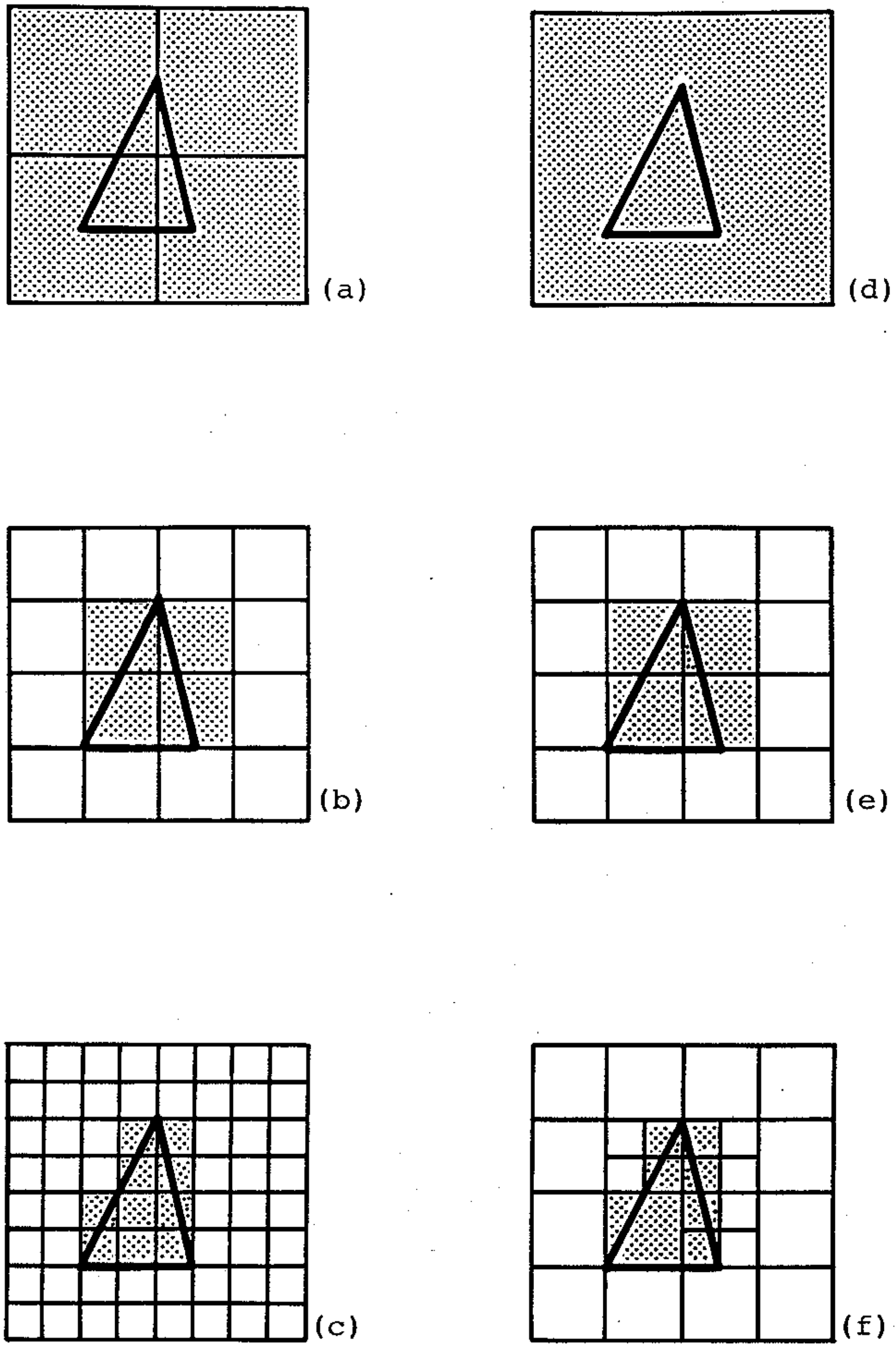


Figure 14. An illustration of the relative growth of the array and quadtree representations at different levels of resolution for a simple triangular region. (a)-(c) are the array representations of the triangle at resolutions 1, 2, and 3, while (d)-(f) are the corresponding quadtree representations at the same resolutions. Whenever any part of a square or node partially overlaps the interior of the triangle, the node or square is treated as being in the region (and shown shaded). Note that the quadtree at resolution 1 (i.e., in (d)) has just one node as the triangle overlaps each of the four blocks and thus they have been merged.

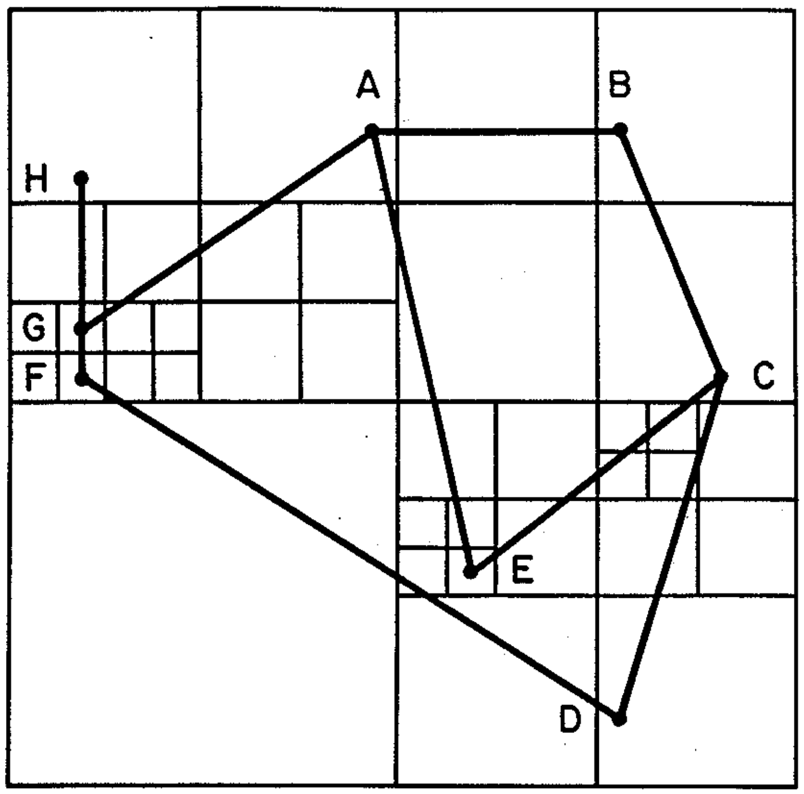
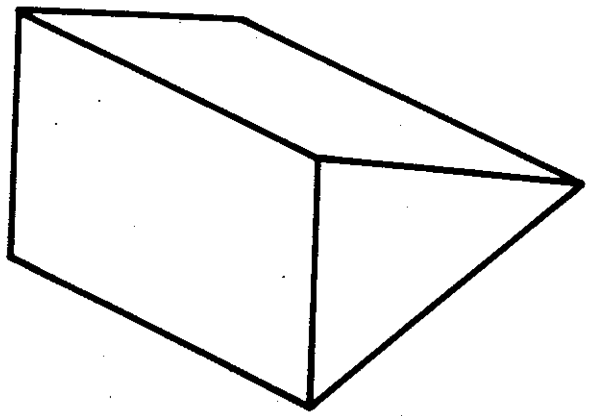


Figure 15. A vector data quadtree corresponding to the image of Figure 1a.

(a)



(b)

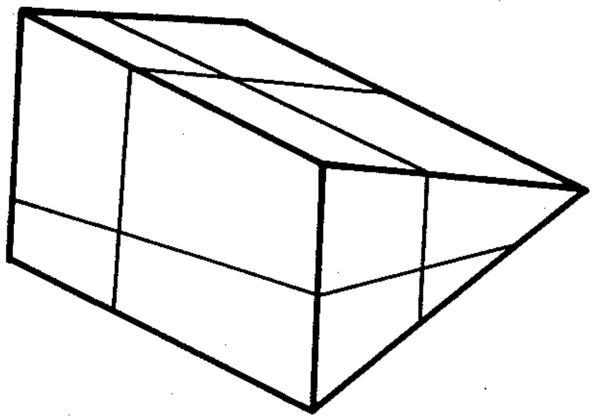


Figure 16. (a) Example three-dimensional object and (b) its corresponding vector octree.

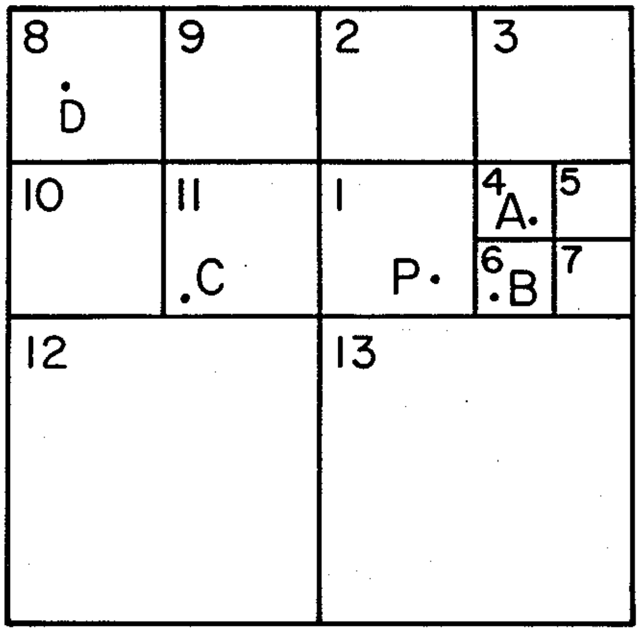
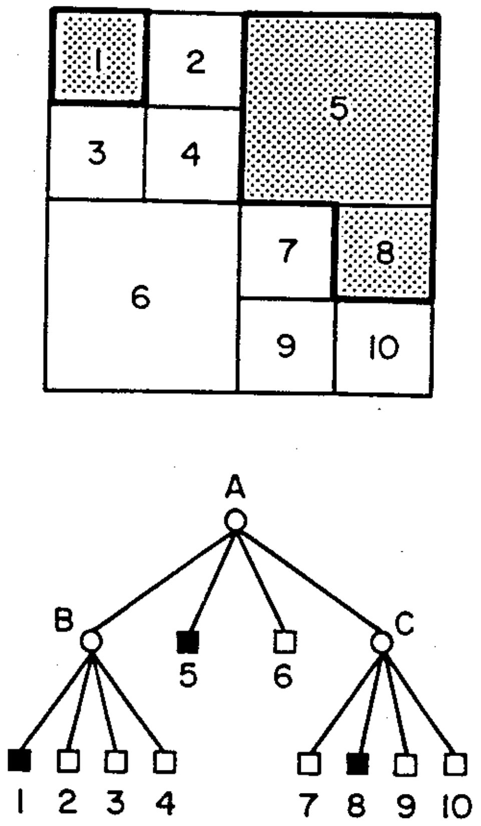
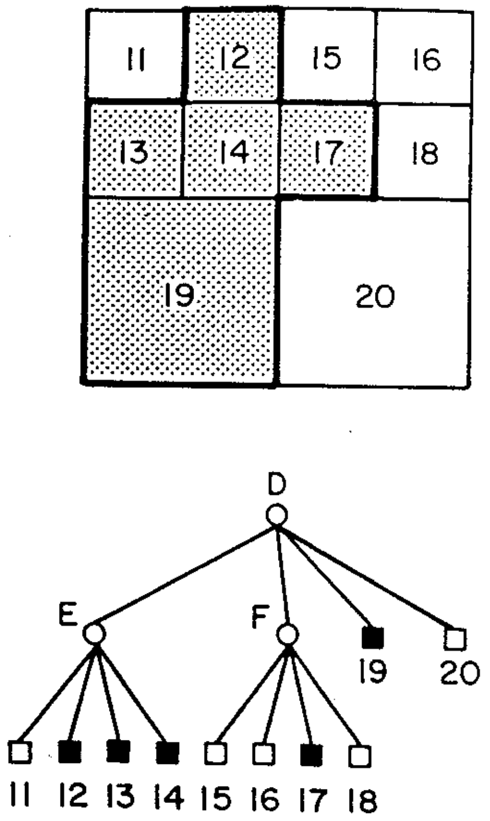


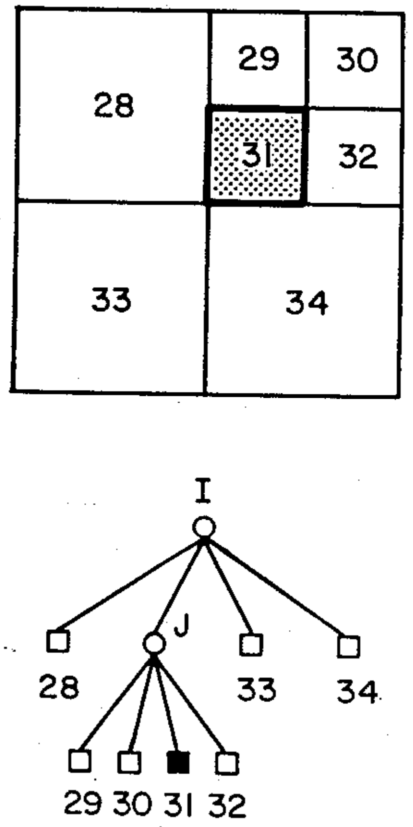
Figure 17. Example illustrating the neighboring object problem. P is the location of the pointing device. The nearest object is represented by point B in node 6.



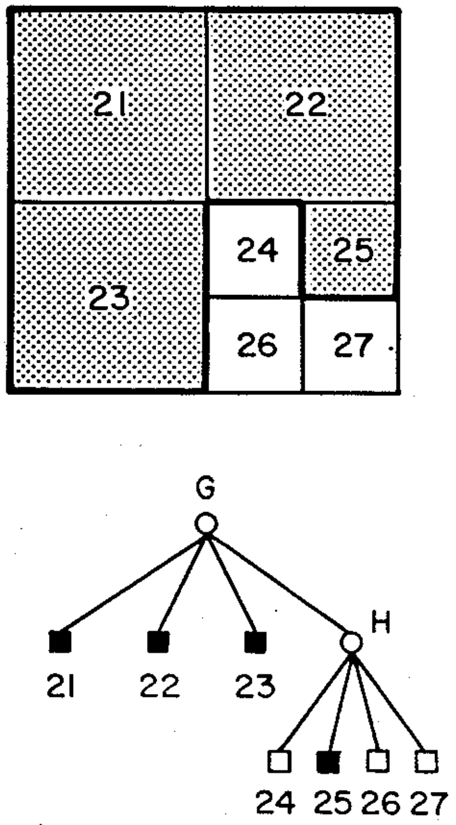
(a) Sample image and its quadtree.



(b) Sample image and its quadtree.



(c) Intersection of the images in (a) and (b).



(d) Union of the images in (a) and (b).

Figure 18. Example of set-theoretic operations.

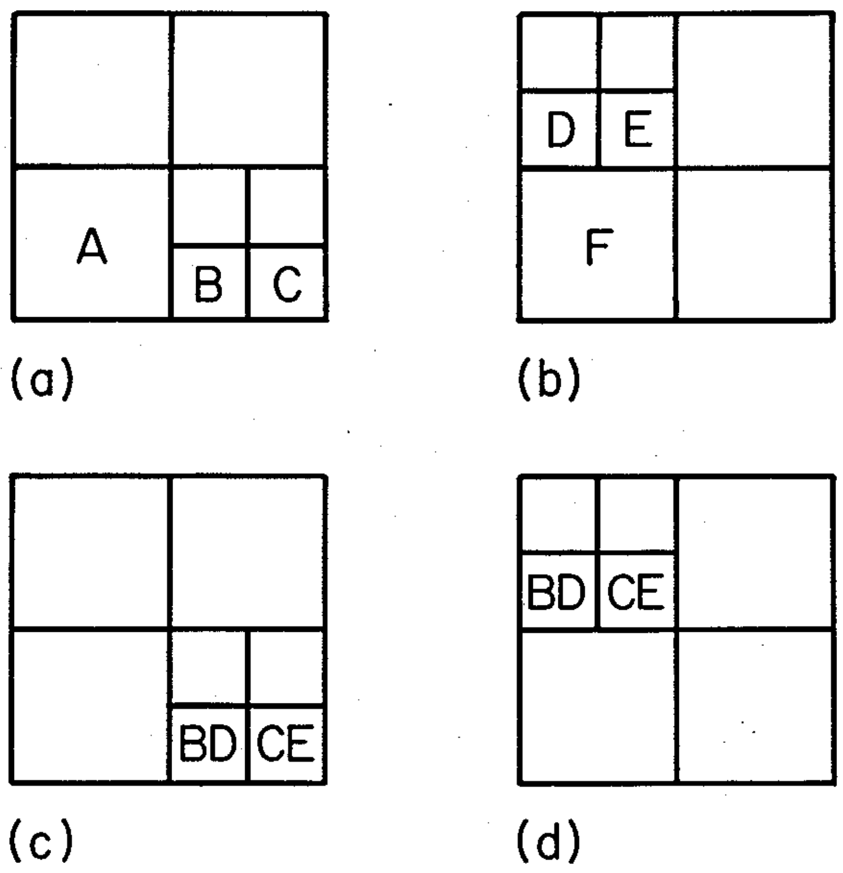
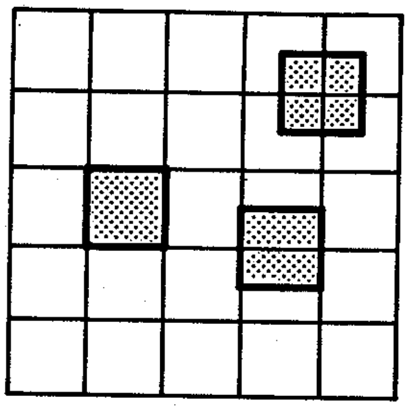


Figure 19. Example of rectilinear unaligned-quadtree intersection.
(a) A 4x4 quadtree with a lower lefthand corner at (0,2).
(b) A 4x4 quadtree with a lower lefthand corner at (2,0).
(c) The intersection of (a) and (b) with (a) as the aligned quadtree.
(d) The intersection of (a) and (b) with (b) as the aligned quadtree.

(a)



(b)

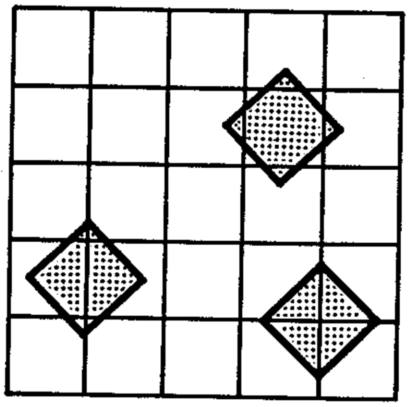


Figure 20. Examples showing how many squares can be overlapped when a square of size WxW is overlaid on a grid of squares such that each square is of size WxW so that the square and the grid are (a) rectilinearly unaligned; (b) generally unaligned.

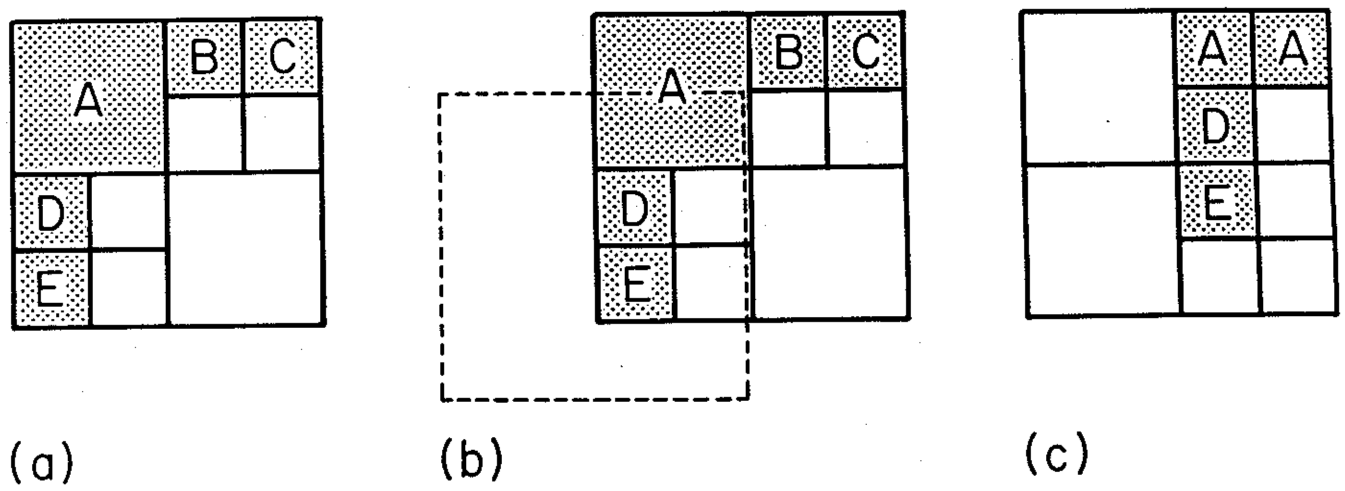
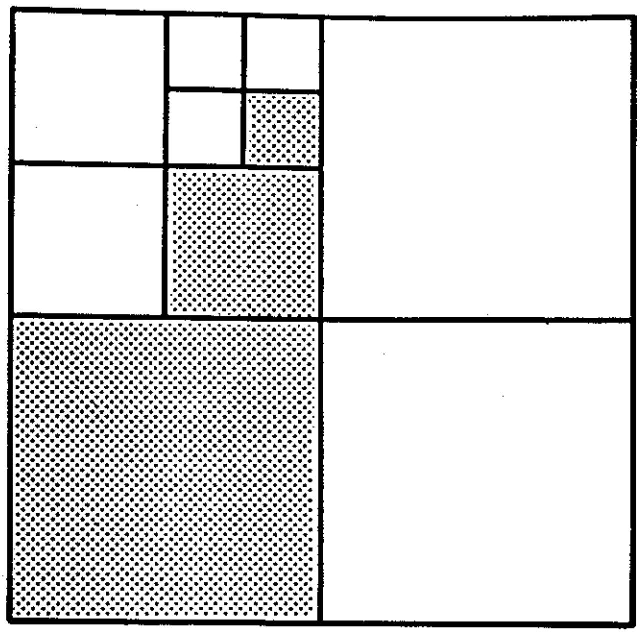
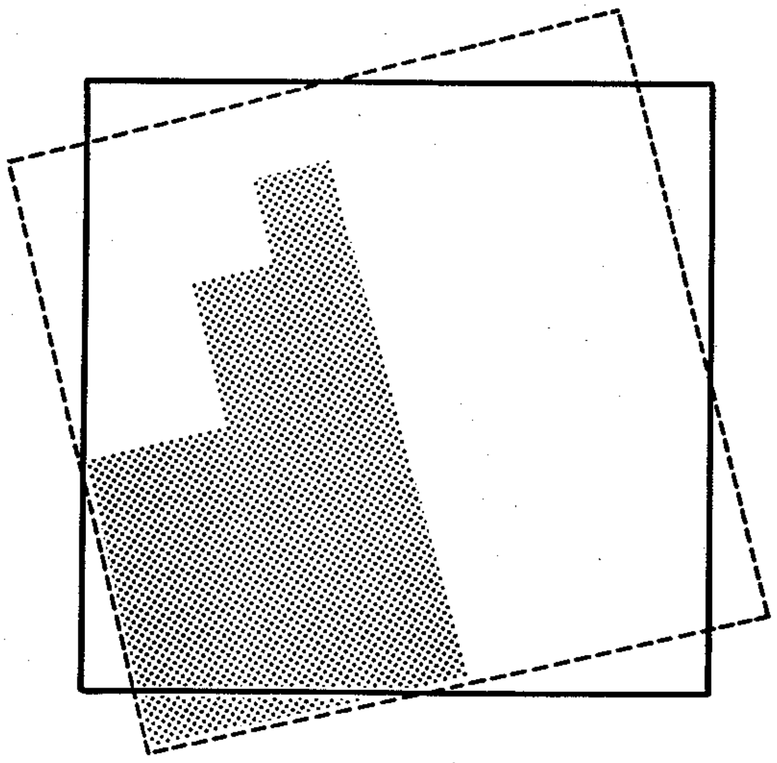


Figure 21. Example of shifting a 4x4 quadtree by 2 units to the right and 1 unit up.
(a) Original quadtree.
(b) Relative position of the two quadtrees that are being intersected.
The BLACK quadtree is shown with broken lines.
(c) The result of shifting the quadtree of (a).

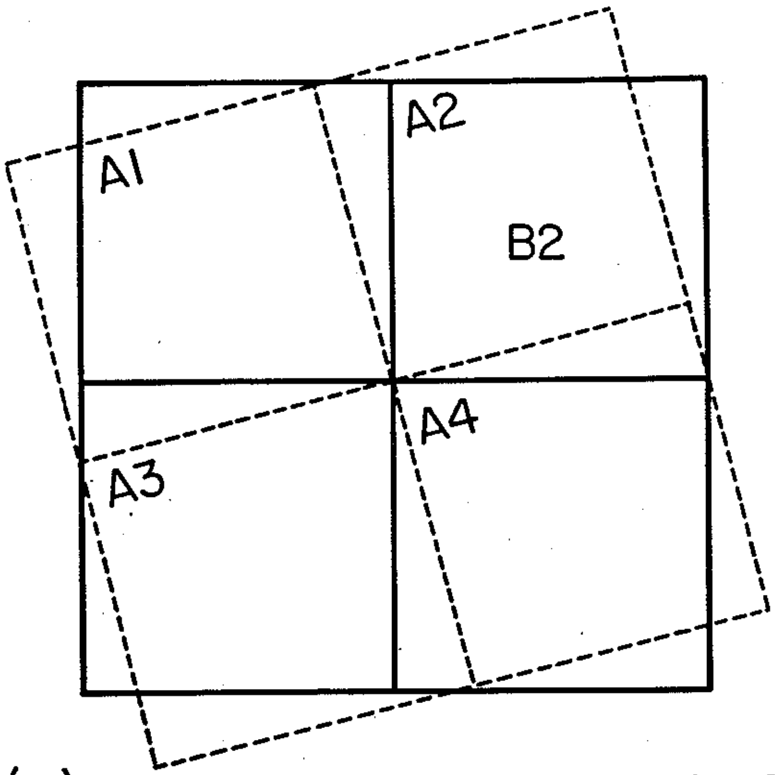


(a) Sample quadtree.

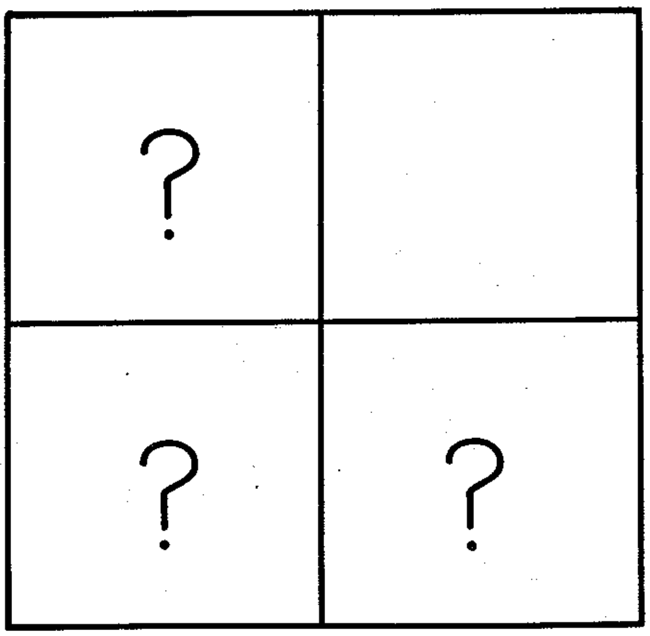


(b) Rotation of (a) by 16 degrees in a counterclockwise direction about its origin.

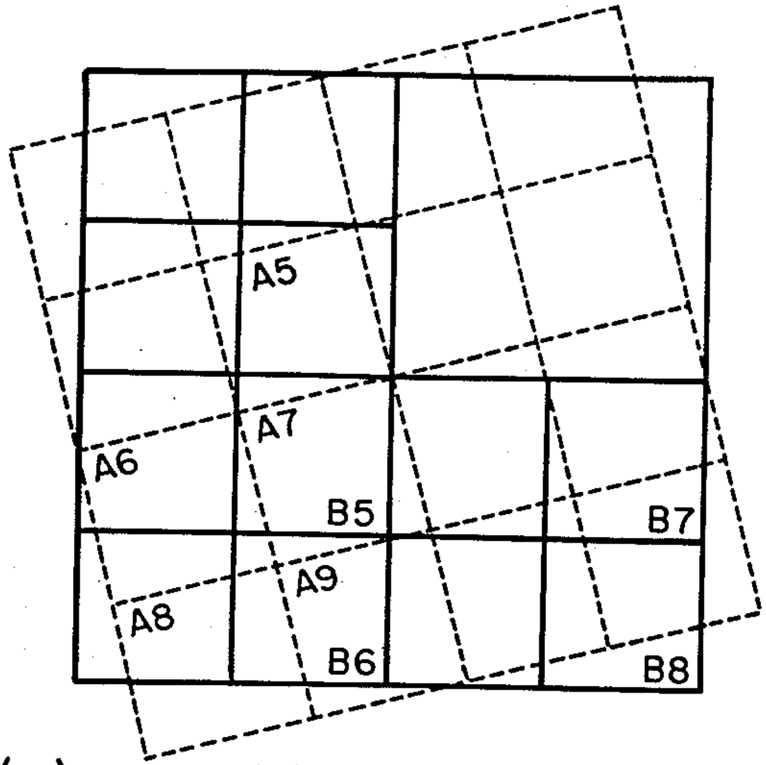
Figure 22. Example of rotation. Broken lines depict the decomposition of the unaligned quadtree and solid lines depict the decomposition of the aligned quadtree.



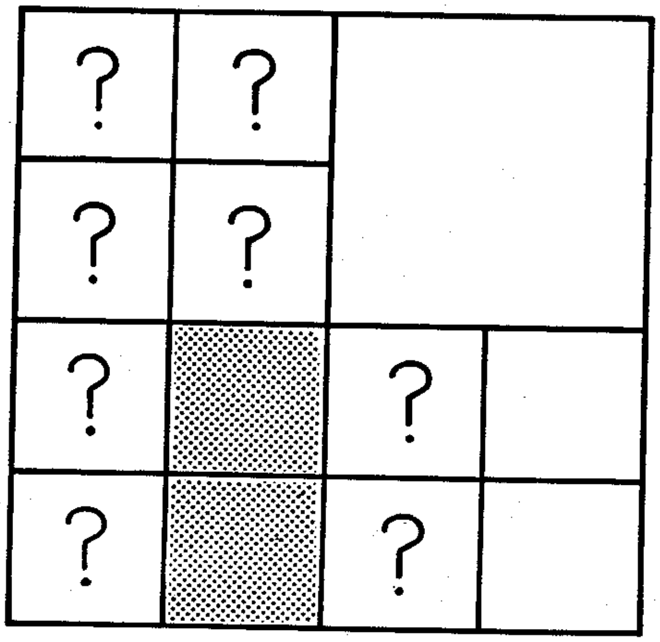
(c) Decomposition after the first level of subdivision.



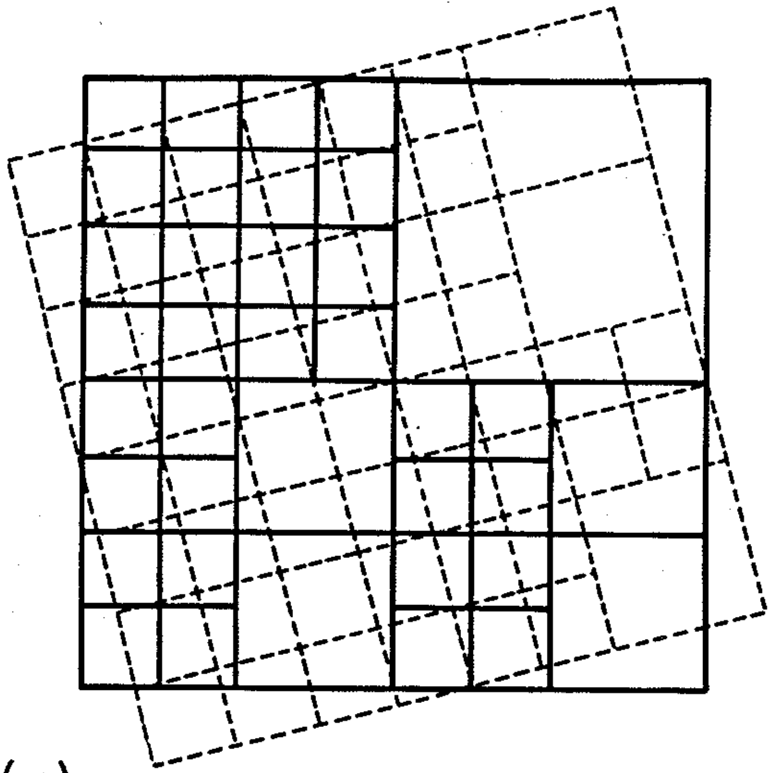
(d) Rotated quadtree with one level of subdivision.



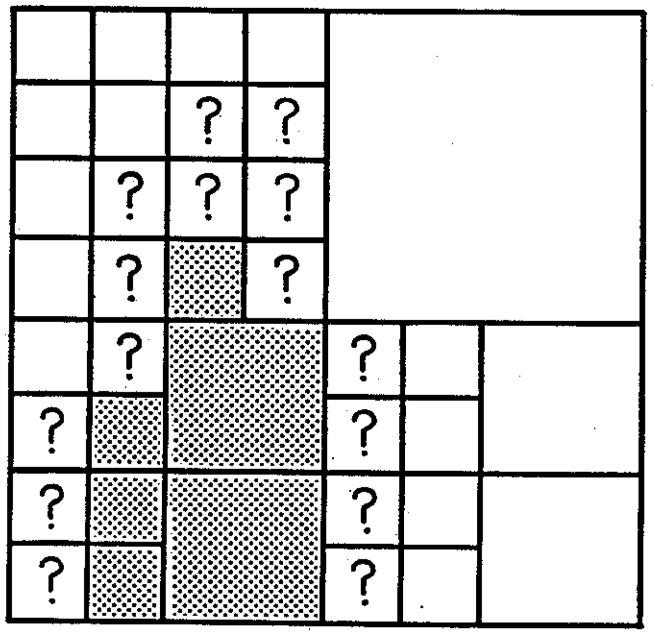
(e) Decomposition after the second level of subdivision.



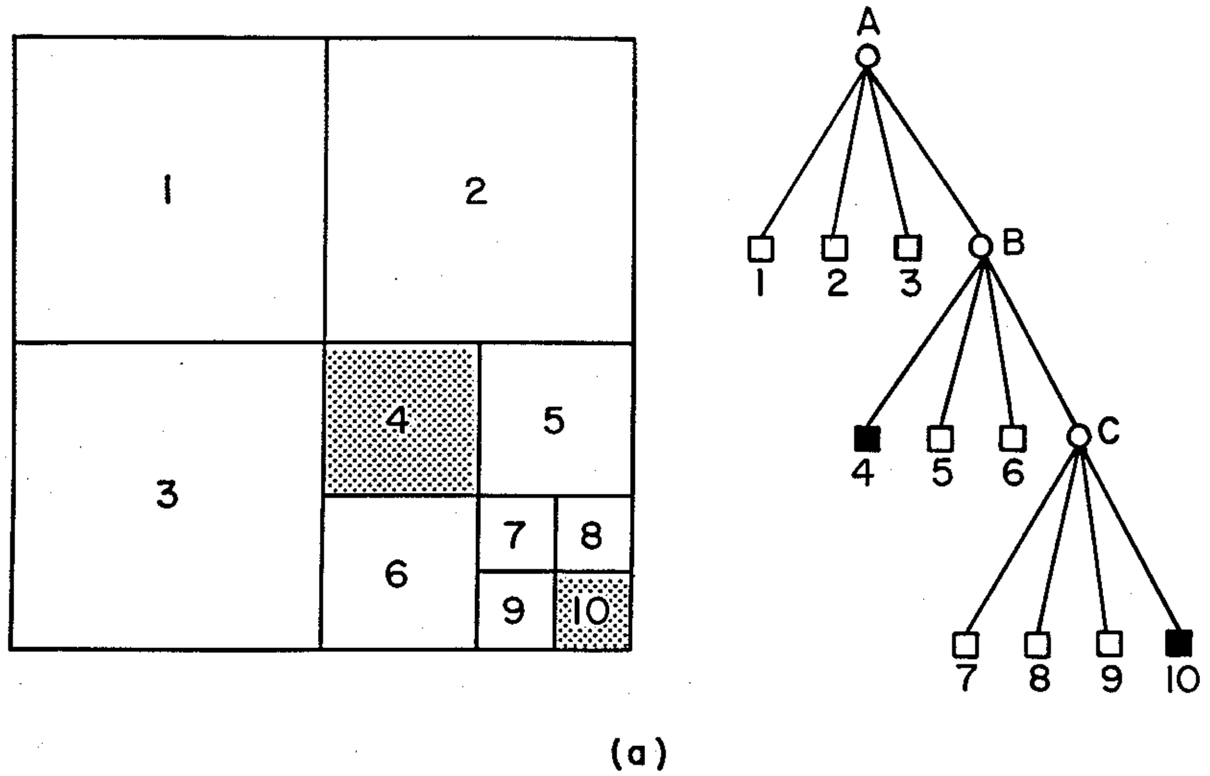
(f) Rotated quadtree with two levels of subdivision.



(g) Decomposition after the third level of subdivision.



(h) Rotated quadtree with three levels of subdivision.



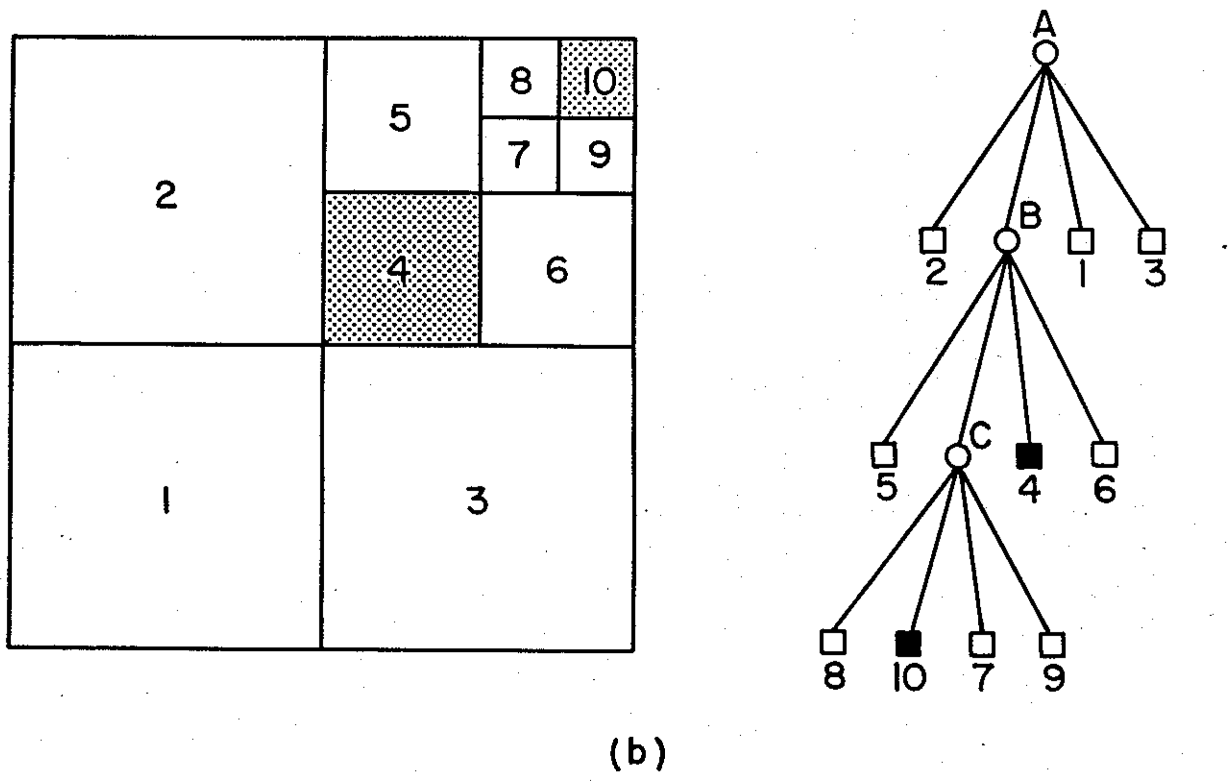
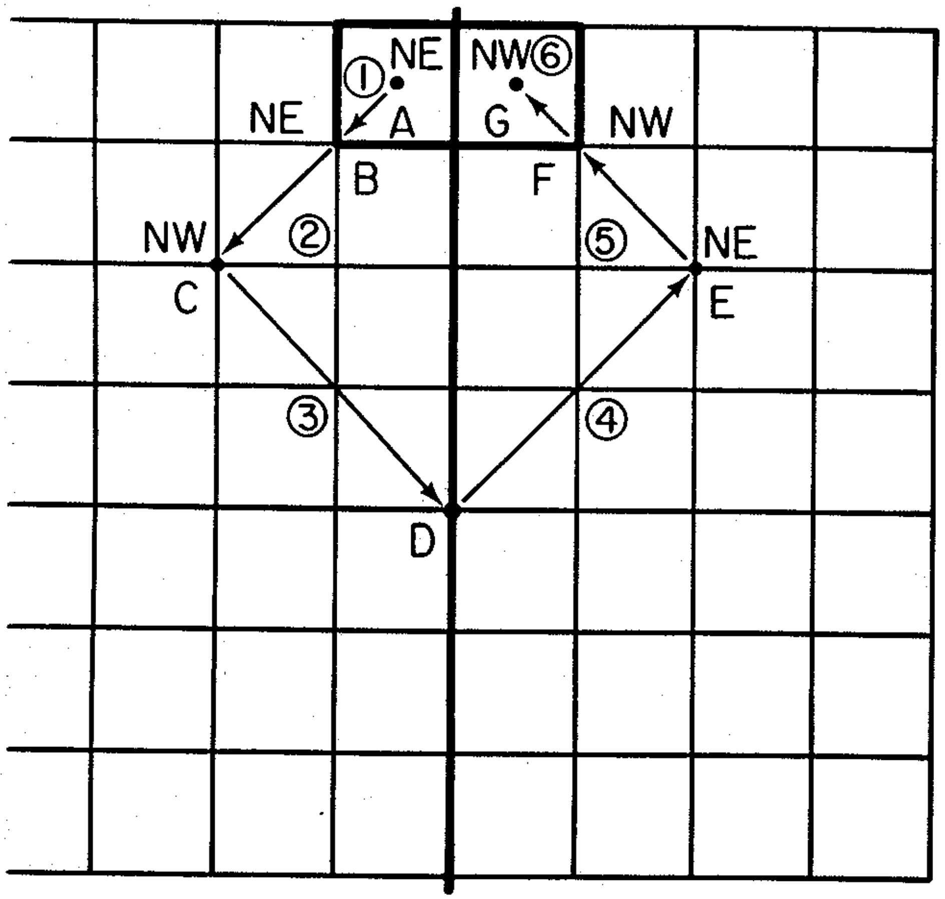
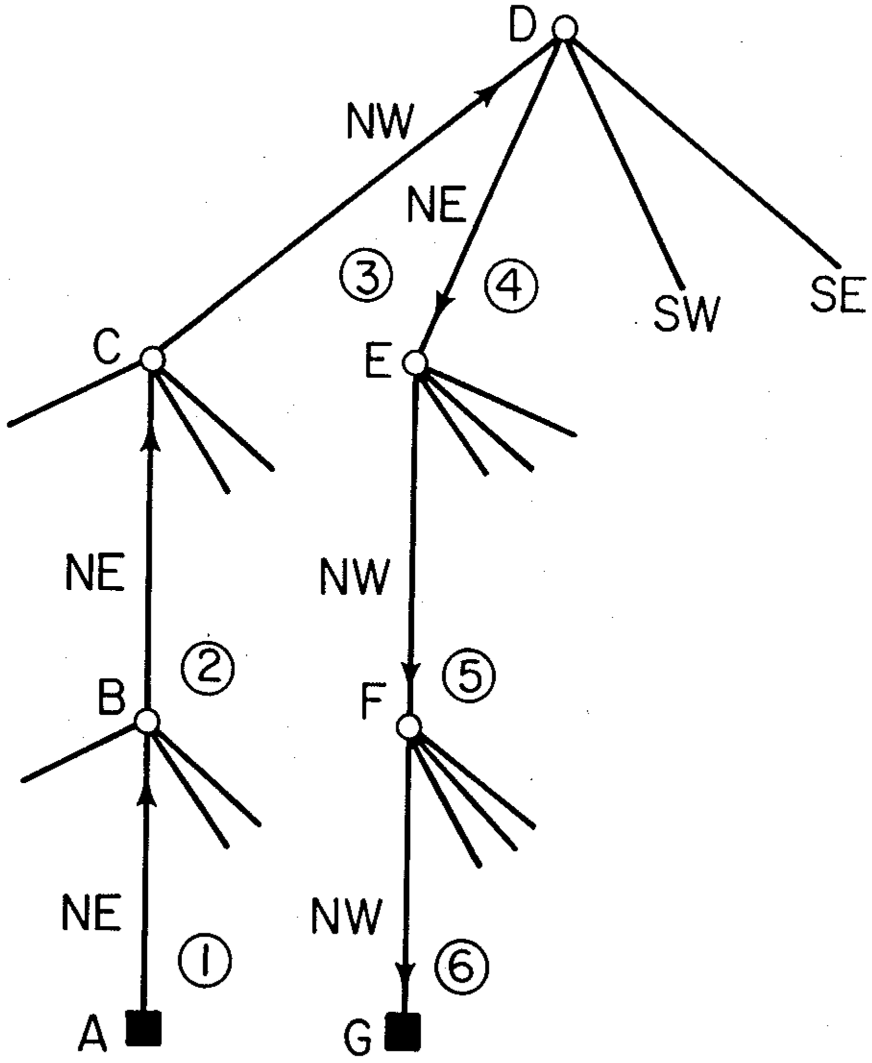


Figure 23. Rotating (a) by 90° counterclockwise yields (b).



(a) Block decomposition.



(b) Tree representation.

Figure 24. The process of locating the eastern neighbor of node A (i.e., G).

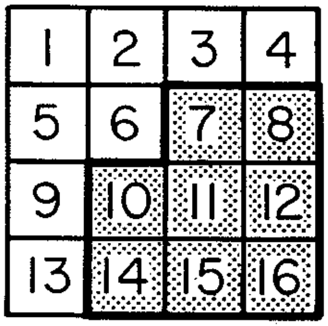


Figure 25. Raster scanning order for the image of Figure 7.

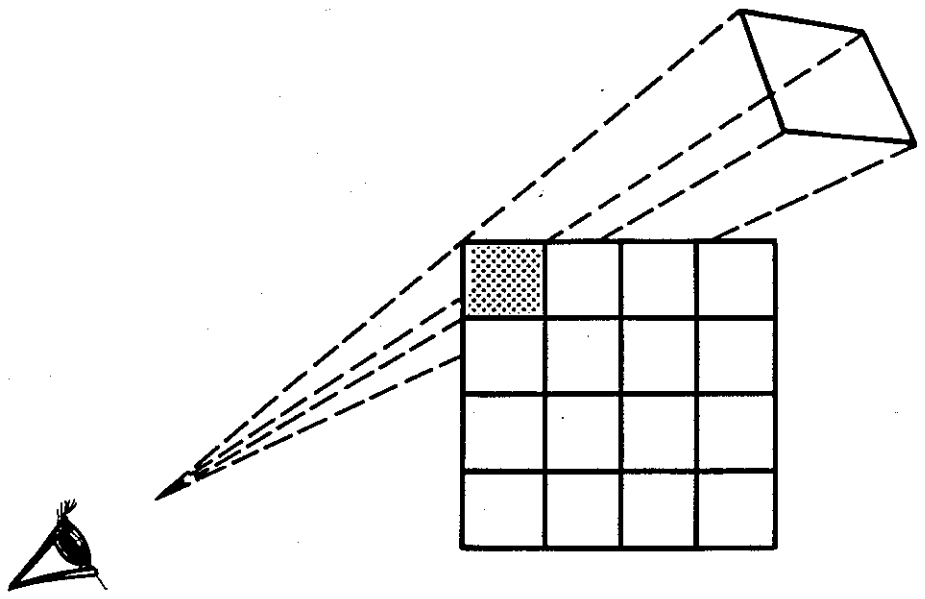


Figure 26. The viewing pyramid associated with the black pixel (shown shaded) in the viewplane.

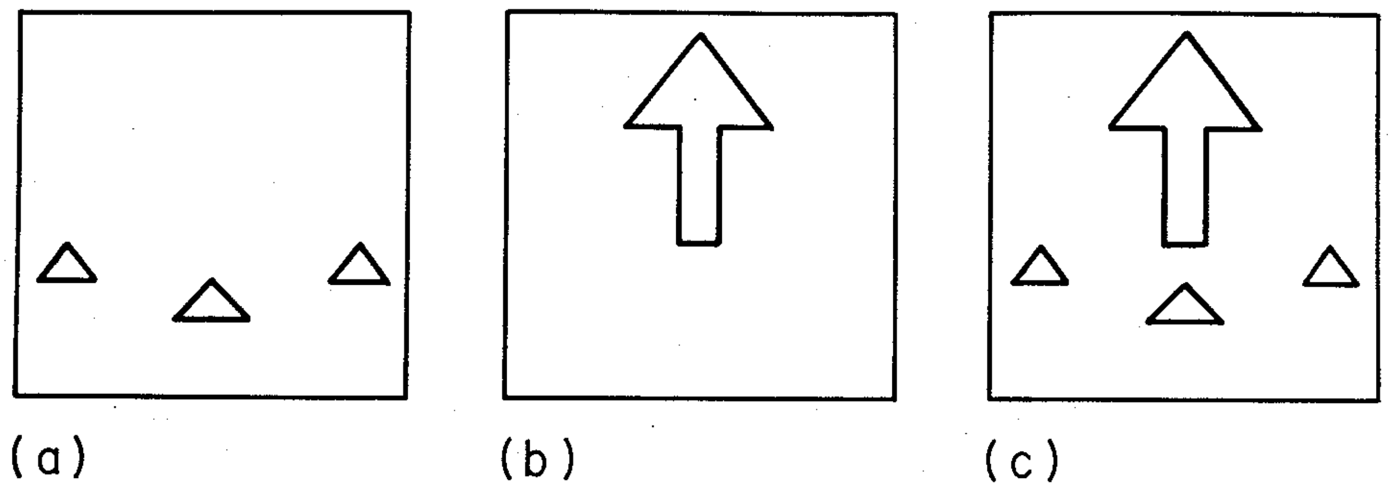


Figure 27. Example of scene creation via cel overlay.
(a) Plant cel.
(b) Tree cel.
(c) Possible overlay of (b) on (a).

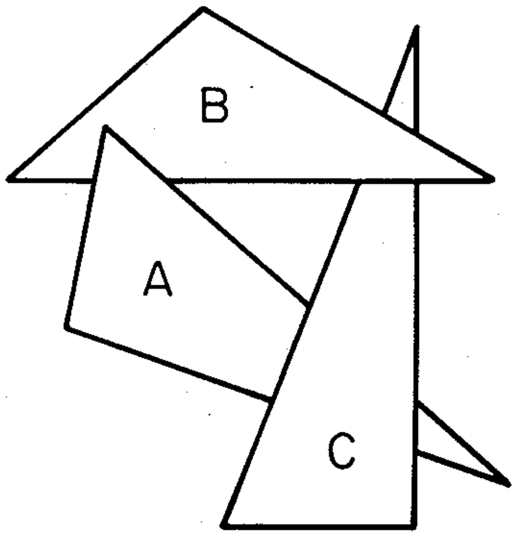
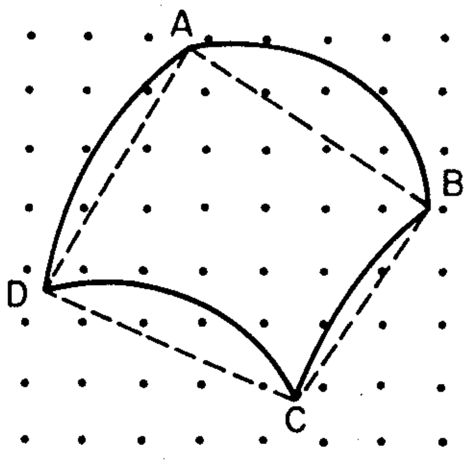
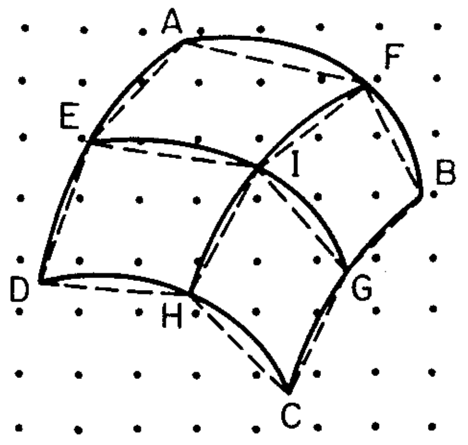


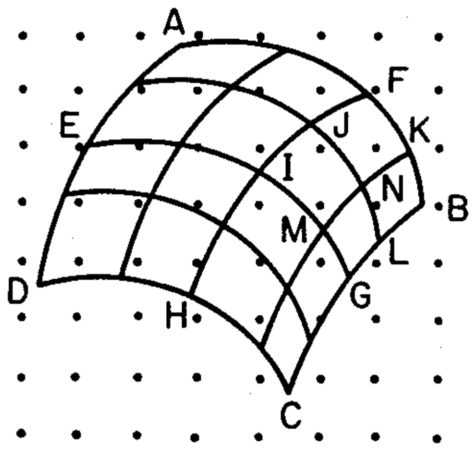
Figure 28. Example of a three-dimensional image where occlusion is not transitive.



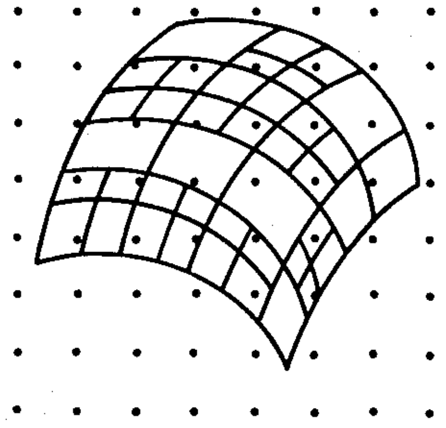
(a) A single patch.



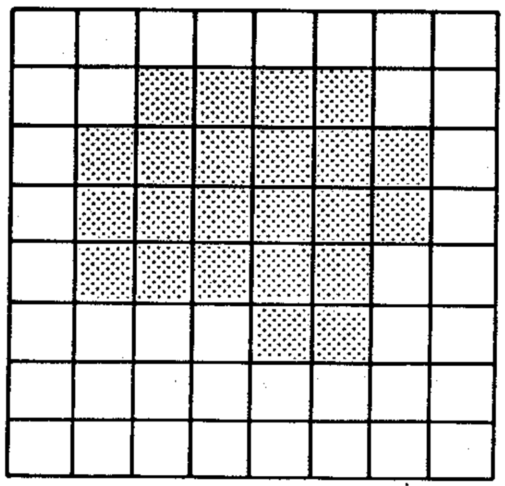
(b) Decomposition of (a) into four patches.



(c) Decomposition of (b) into sixteen patches.



(d) Final decomposition such that each patch contains no more than one pixel center.



(e) The raster image corresponding to the decomposition.

Figure 29. An example of the use of recursive decomposition into patches for the display of curved surfaces. Where space permits (i.e., in (a) and (b)), patches are denoted by solid lines and their approximating polygons are denoted by broken lines.

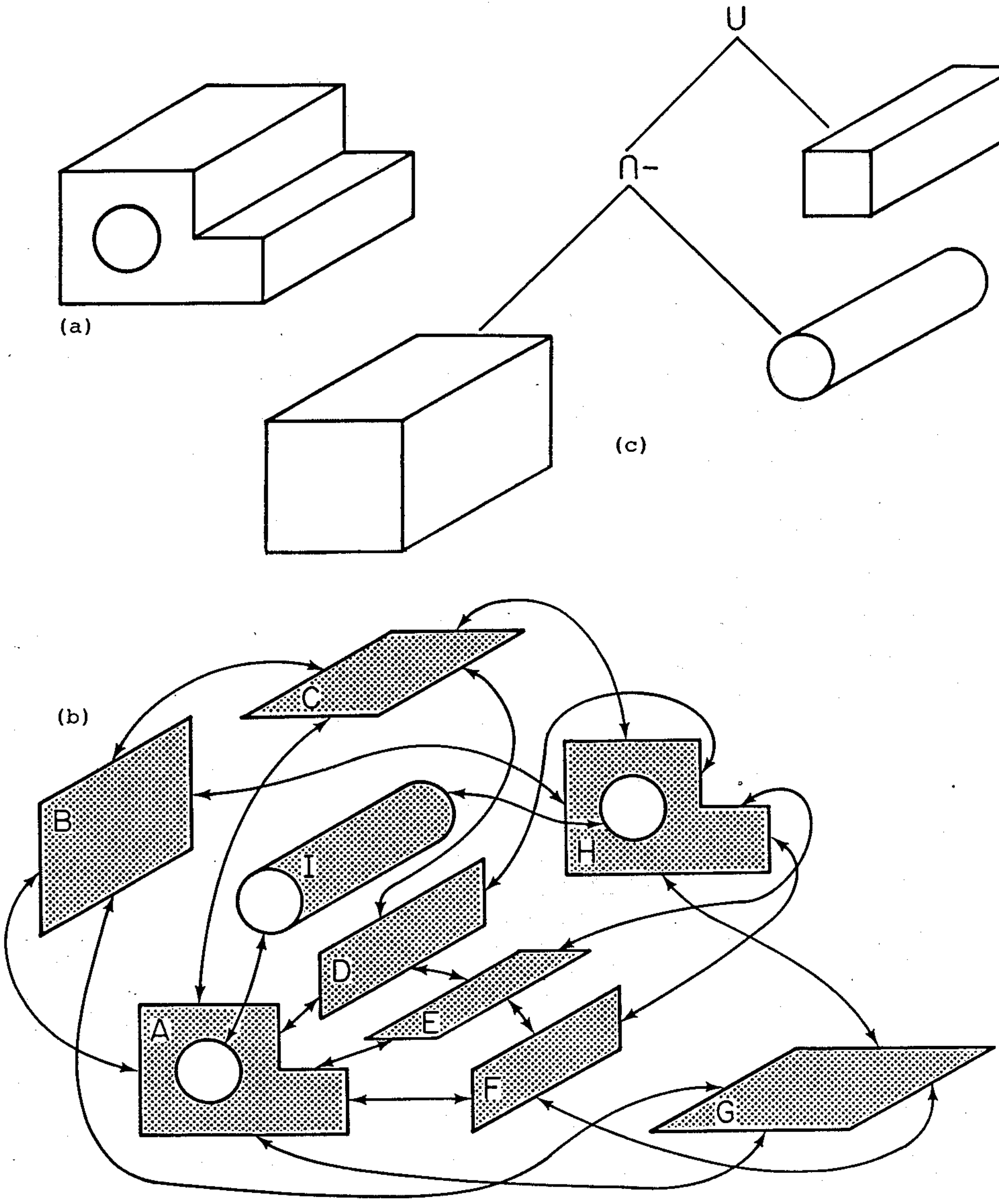


Figure 30. (a) A three-dimensional object; (b) its boundary representation; and (c) its CSG tree.

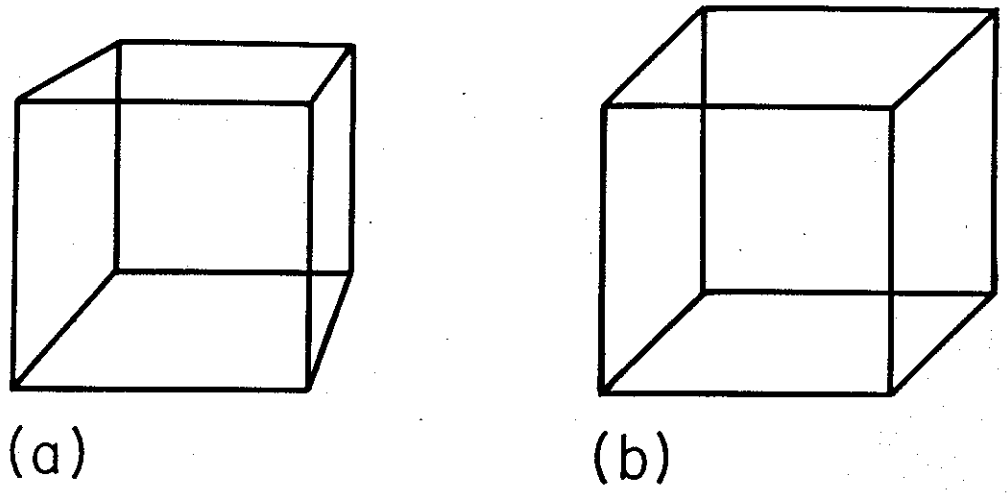


Figure 31. (a) Perspective projection of a cube.
(b) Parallel projection of a cube.

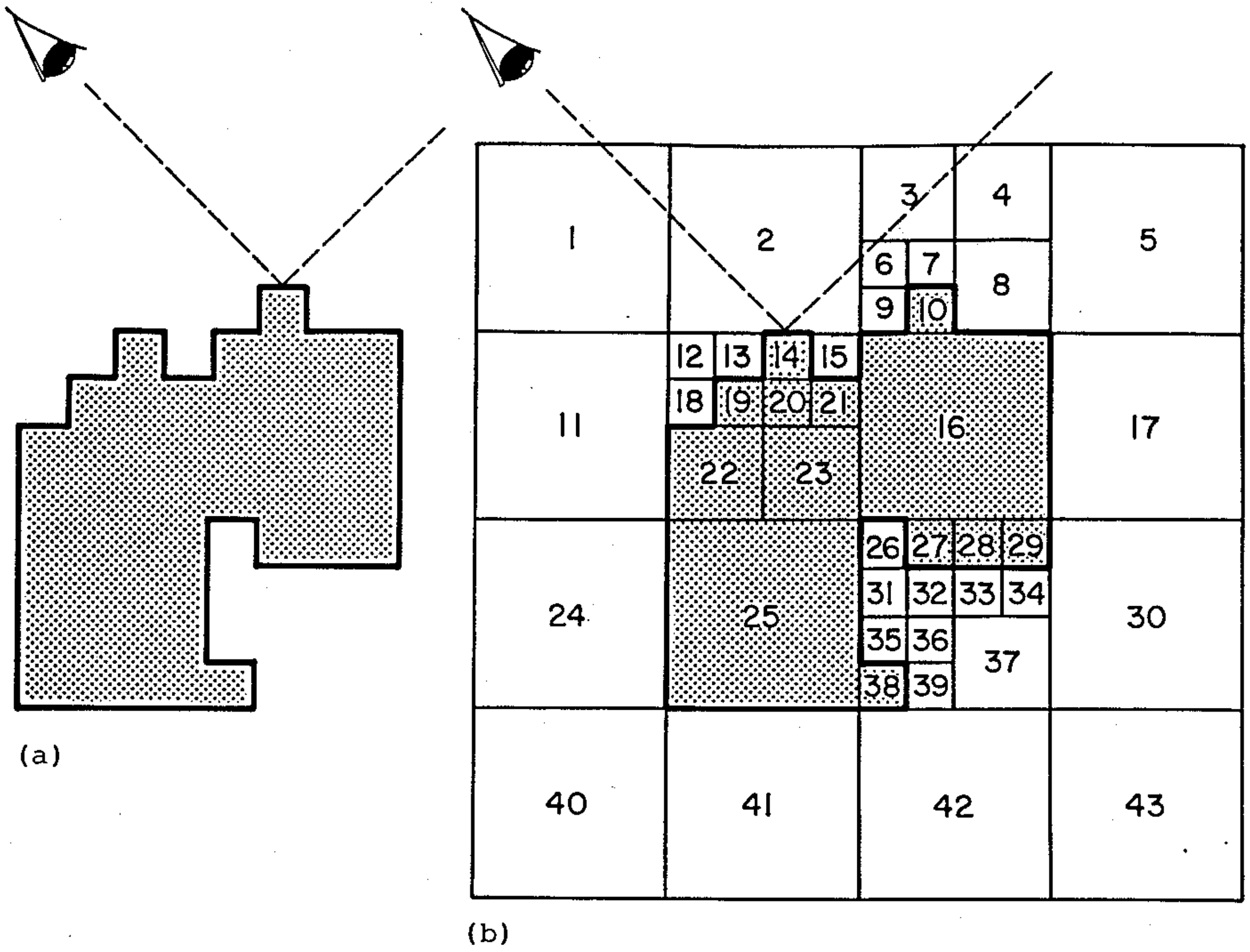


Figure 32. (a) Example polygon and (b) its corresponding quadtree with a ray shown to emanate from the viewpoint and reflect from the object.