

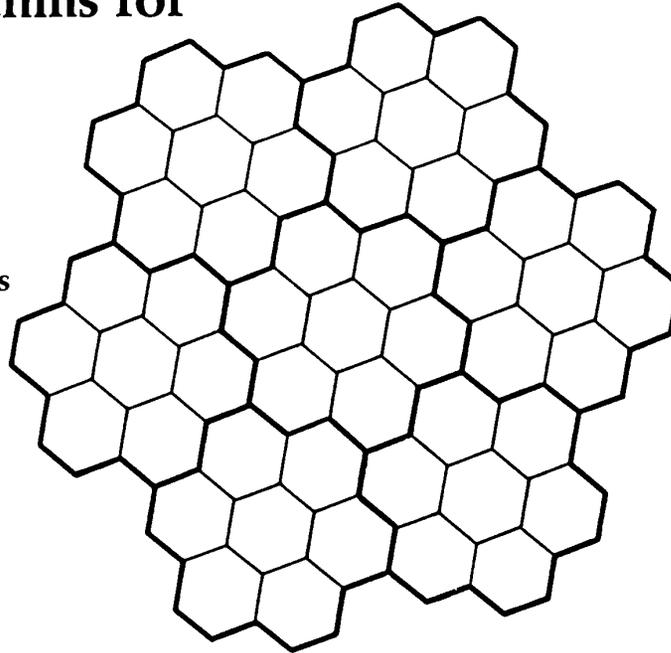
# Data Structures

## Hierarchical Data Structures and Algorithms for Computer Graphics

### Part I: Fundamentals

Hanan Samet  
University of Maryland

Robert E. Webber  
Rutgers University



This is the first part of a two-part overview of the use of hierarchical data structures and algorithms in computer graphics. In Part I, the focus is on fundamentals. Part II focuses on more advanced applications. Methods based on hierarchical data structures and algorithms have found many uses 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. Object-space hierarchies are discussed briefly, but the main emphasis is on hierarchies constructed in the image space, such as quadtrees and octrees.

**C**omputer 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. To attain maximum 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 television camera. In contrast, instead of modeling the display screen directly, the vector format models the

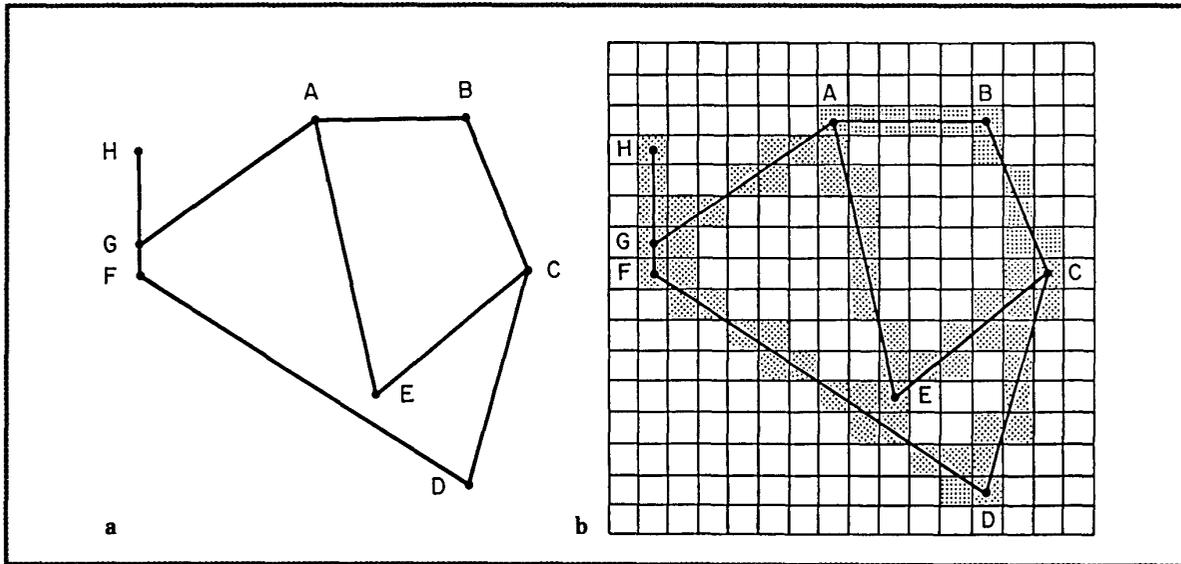


Figure 1. Example image represented in (a) a vector data format and (b) a raster data format.

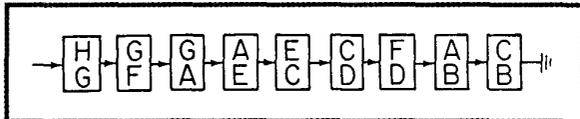


Figure 2. Linked list of records representing, by pairs of their endpoints, the line segments of Figure 1a.

ideal geometric space 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, computer graphics also involves 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.<sup>1</sup>

Both data formats have obvious representations, which are minimal in the sense of providing structure just sufficient 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 (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 polyhedral representation<sup>2</sup> illustrated in Figure 3. While these representations are suitable for medium-range applications, once the scene being modeled

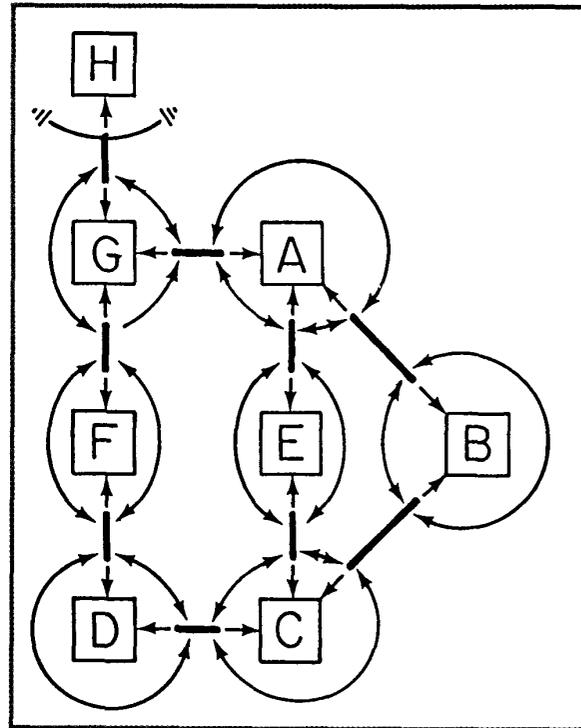
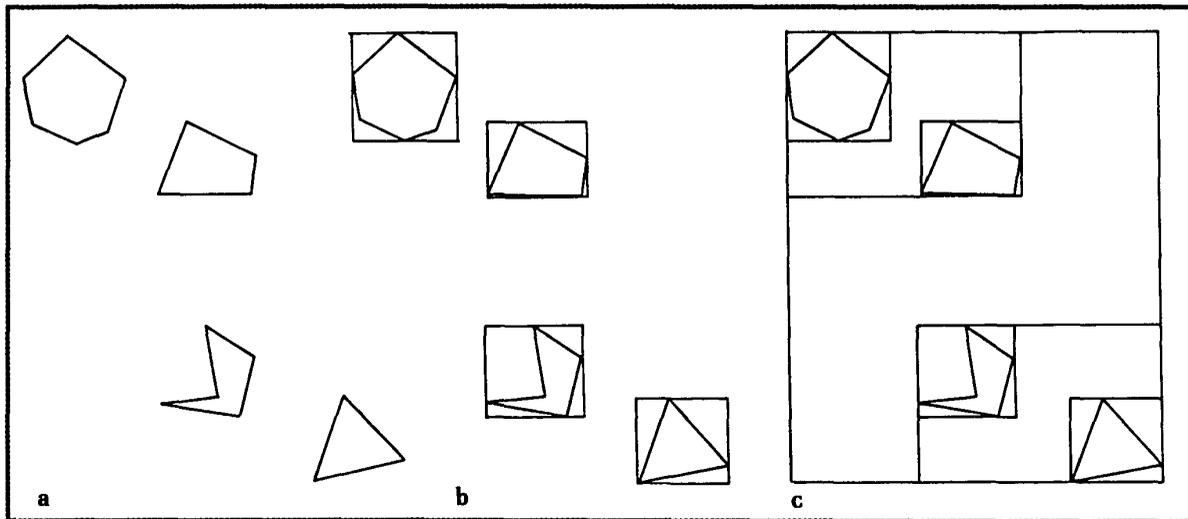


Figure 3. Winged-edge representation of the line segments and their endpoints of Figure 1a. The result is a graph with two types of nodes shown as squares and narrow solid rectangles. The squares correspond to endpoints of line segments while the rectangles correspond to the actual line segments. Each 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.



**Figure 4.** An example of the use of bounding objects: (a) unbounded objects for use in Figures 4b through 4c, (b) bounding boxes, (c) hierarchical bounding boxes.

becomes significantly larger than the display grid, major logistic problems arise.

There are two ways<sup>1</sup> to handle the logistics problems. One approach, based on object-space hierarchies,<sup>3</sup> is discussed only briefly in this article. 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 Parts I and II<sup>4</sup> of this article.

The two parts are organized as follows. In Part I we review the basic fundamentals of hierarchical data structures and show how they are used in the implementation of some basic operations in computer graphics. The second section of Part I contains a general discussion of properties of the structures. The third section describes how some basic operations are performed using quadtrees. The performance of basic operations using octrees, where it differs from quadtrees, is discussed in Part II of the article (to appear in the July 1988 *CG&A*). Part II focuses on more advanced applications with a heavy emphasis on quadtree hidden-surface algorithms and such display methods as ray tracing and radiosity. More references and details on hierarchical data structures appear elsewhere.<sup>5-7</sup>

### Properties of quadtrees and octrees

In this section we discuss some fundamental properties of quadtrees and octrees. First, however, we elaborate on the motivation for their development. As mentioned earlier, hierarchical data structures such as the quadtree and octree have their roots in attempts to overcome problems that arise when the scene being modeled is more complex than the display grid (in size, precision, num-

ber of elements, etc.). The problems are solved with 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 quadtrees and vector octrees.

### Object-space hierarchies

Two kinds of logistic problems present themselves in scene modeling. First, communication between the user software and the graphics package—i.e., the number of procedure calls (or commands transmitted on a graphics channel)—can become a bottleneck for the system. The second problem is in determining what subset of the scene is actually visible. For example, in a  $512 \times 512 \times 512$  scene, only about  $512 \times 512$  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.<sup>8</sup> This observation has been used as the basis for the organization of the user's interface to the data from the earliest graphics systems<sup>9,10</sup> to the most recent graphics package designs.<sup>11,12</sup>

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<sup>13</sup> to surround the object (see Figure 4a) with a bounding box (see Figure 4b) or even a sphere. If the bounding object is not visible, then clearly the object being bounded is also not visible. This technique produces a major computational savings, since it is usually much easier to test for visibility of the bounding object than the visibility of the bounded object. However, the approach cannot deal with the visible-subset problem when the number of objects is large. Researchers<sup>3,14,15</sup> have noted that the objects being bounded need not be limited to the primitive objects of the scene; instead, bounding objects can also be placed around the complex objects formed by the different levels of the object hierarchy (see Figure 4c).

This approach is easy to implement and can greatly improve execution time. But 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, 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.<sup>8</sup> Levels formed in the 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. Some attempts have been made to structure the object-space artificially to avoid these problems,<sup>16</sup> but such attempts have problems handling dynamically changing scenes (due to preprocessing costs) as well as often producing worst-case results.

A related artificial object hierarchy is the strip tree.<sup>17</sup> Here 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. 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. The process stops when the width of each strip is less than a predetermined value. This approach to subdivision can be viewed as heuristically subdividing a curve near its points of maximum curvature and halting when the curve is essentially linear. The worst-case situations illustrated by this data structure are typical of the problems with computations on object hierarchies.

### 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., 2D and 3D arrays) is that they require the user to fix the maximum resolution in advance. However, a hierarchical

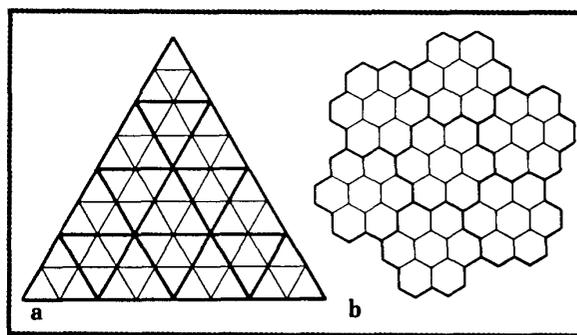


Figure 5. Examples of nonsquare partitionings of the plane: (a) equilateral triangles, (b) hexagons.

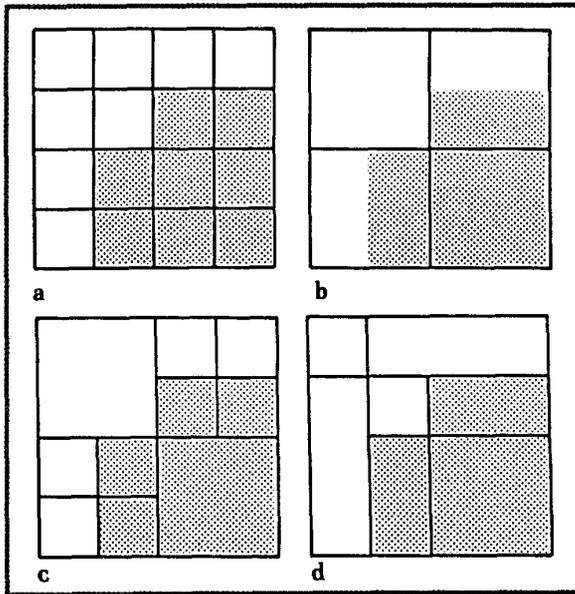
organization of the image space allows the resolution to vary with the complexity of the objects in various regions. Of course, there are many ways to partition the image space (when it is viewed as a continuous plane/space),<sup>18-20</sup> but to interface easily with a Cartesian coordinate system and with the typical display device controller, a decomposition of the plane into square regions (and a space into cubical regions) is simplest. Two examples of nonsquare partitionings of the plane are given in Figure 5. In the following, we discuss the organization of a planar image space (leaving consideration of the 3D image space for a later section).

When justifying the use 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.

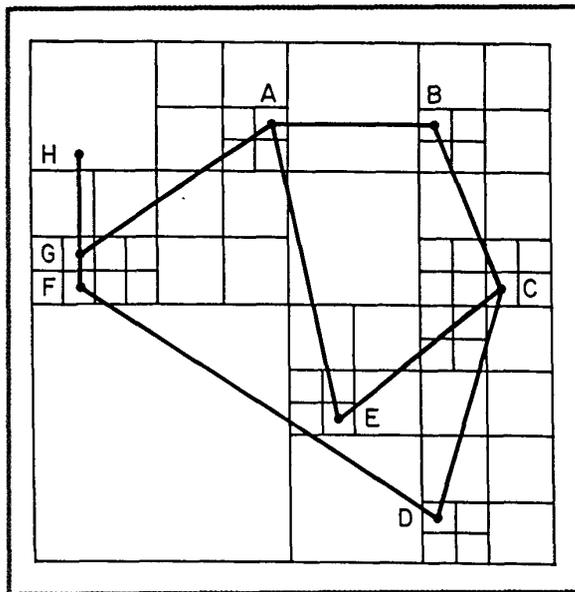
For large-scale applications, however, the costs associated with the imprecision of these approximations can easily overshadow any benefits accrued from the explicit maintenance of just one of the hierarchies. Thus, when possible, both hierarchies should be maintained. A definitive analysis of the merits of image-space and object-space hierarchies awaits a universally accepted model of "typical graphic data."

### Quadtree/octree definition

One commonly used 2D image-space hierarchy is typified by the quadtree data structure.<sup>5,21</sup> It is constructed in the following manner. We start with an image (whose binary array representation is given in Figure 6a) 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 6b). Each of these new image spaces is treated as if it



**Figure 6. Illustration of the quadtree decomposition process: (a) original image, (b) first level of decomposition, (c) second and final level of decomposition, (d) an example of an irregular decomposition.**



**Figure 7. The edge quadtree for the vector data of Figure 1a. The maximum level of decomposition is 4.**

were isolated, and each one is examined to determine whether or not it has a simple description (resulting in Figure 6c). 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 6d). While it is plausible to attempt to move the boundaries of the subregions to distribute the complexity of the image more evenly, how to do this effectively is not clear. 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, 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 differ only 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 analogously.<sup>22,23</sup>

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 stipulating 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.<sup>22,24,25</sup> The modification is to establish a maximum quadtree depth. Once the maximum depth is reached in the construction process, if the criterion is still not satisfied, then the region is simply represented by a pixel. The result is a mixed raster and vector representation in which some information about the image can be lost. This representation is known as the edge quadtree.<sup>24,26</sup> For example, Figure 7 is the edge quadtree corresponding to the vector data of Figure 1a. In this case, truncation at the maximum tree depth (4) has occurred at the nodes containing vertices A, B, C, D, E, F, and G, but not H.

The octree data structure<sup>27-30</sup> is the 3D analog of the quadtree. It is constructed as follows: 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 3D voxels having a single color, instead of 2D pixels) or vector format (consisting of solids and planar or curved surfaces, instead of polygons and edges). Fig-

ure 8a is an example of a simple 3D object whose raster octree block decomposition is given in Figure 8b and tree representation in Figure 8c.

In this article, we 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 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. The structure has many interesting mathematical properties, some of which are reviewed briefly below after the discussion of implementation techniques.

### 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<sup>31</sup> mentions three general approaches to representing trees. Each of these approaches has been investigated by others with regard to the specific representation of quadtrees. In the following, we describe these three approaches and discuss their relative advantages and disadvantages 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 9 is the tree/pointer representation of the quadtree of Figure 6. 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 indicate whether it is an internal node or a leaf. To describe quadtree algorithms, a father link is useful in each node; however, this is not necessary for implementation, 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,<sup>27,32</sup> but these are not necessary for the efficient processing of the quadtree. Most early implementations of quadtrees used the pointer approach, while the next two approaches were considered later because of a perceived storage inefficiency of the pointer approach. However, the literature is often unclear about 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. Thus a quadtree can be represented by listing the nodes encountered by a preorder traversal of the tree structure. For example,

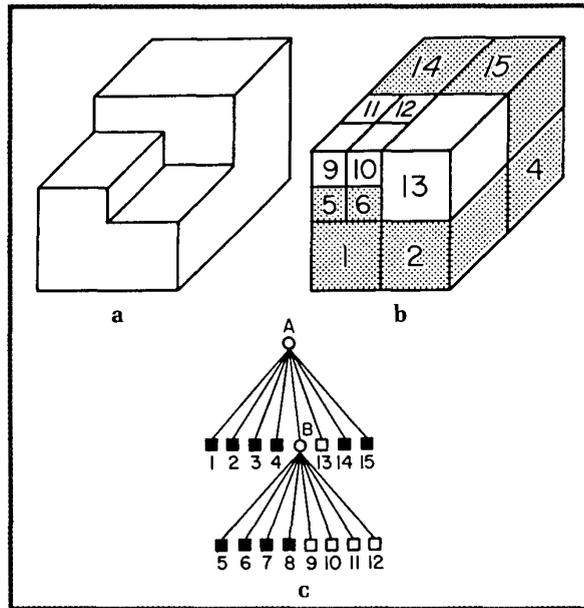


Figure 8. (a) Example 3D object, (b) its octree block decomposition, and (c) its tree representation.

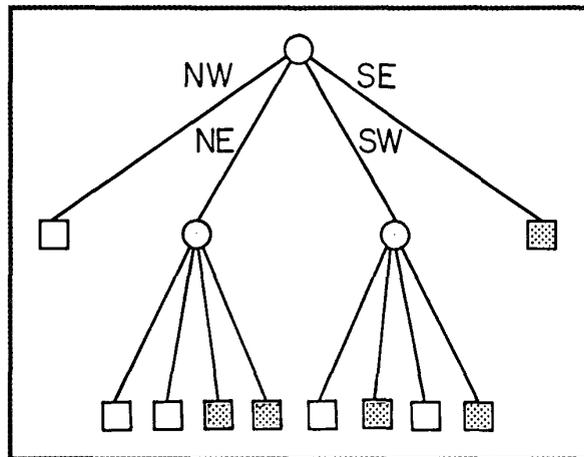


Figure 9. Pointer encoding of the quadtree of Figure 6. Internal nodes are represented by circular nodes. Terminal nodes are represented by square nodes whose contents correspond to the blocks in Figure 6.

traversing the quadtree of Figure 9 in the order NW, NE, SW, and SE and letting G, B, and W denote nonterminal, solid, and empty nodes, respectively, results in the list GWGWBBBGBWBWB. The approach requires exactly one bit of overhead per node, which is used to distinguish between leaf nodes and internal nodes.

Many simple algorithms—for example, 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 cannot be so efficiently implemented. For example, 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—for example, archiving and facsimile transmission. Algorithms specific to this representation have been investigated by Kawaguchi et al.,<sup>33-35</sup> 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,<sup>36,37</sup> 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<sup>31</sup>). It was first proposed by Morton<sup>38</sup> 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, and 3 corresponding to quadrants NW, NE, SW, and 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 reached by a sequence of transitions—SE, NE, and SW—starting at the root. The overhead per node is two bits per level of depth of the node, plus the base 2 logarithm of the depth of the node to specify the level at which the node is found. Gargantini<sup>39-42</sup> has investigated algorithms specific to this representation, which she calls a *linear quadtree*, because the addresses are keys in a linear list of nodes. Oliver and Wiseman<sup>36,37</sup> call it a *leafcode*.

When using the linear quadtree encoding, further reduction of the storage requirements is possible 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 stores only 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 nodes representing a background color<sup>39</sup> (or empty nodes) can also be eliminated from the node list. While this does not excessively complicate the processing of the quadtree, its usefulness is unclear. With 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,<sup>43</sup> 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.<sup>44</sup> This method cannot be easily applied to vector quadtrees.

The relative compactness of the pointer and the linear quadtree representations depends on the complexity of the scene being represented and on the application in which they are used. The attractiveness of the linear quadtree representation increases with the complexity of the scene. However, the choice is not clear cut, and is further complicated by the necessity for the various fields to land on byte boundaries. For 3D data (and data of even higher dimensions), the overhead of the internal nodes is less of a factor and hence the pointer representation is more compact for an even larger fraction of possible scenes.<sup>45</sup>

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.<sup>46-48</sup> Rather than splitting a region with respect to all the principal planes simultaneously, the bintree splits a region against only one plane at each level. For example, instead of splitting an octree node into eight subnodes, the bintree first splits the node into two subnodes along the *x-y* plane. Each of these subnodes is checked to see if it could be a valid leaf (i.e., if it represents a region of just one color). Each subnode that does not correspond to a leaf is then subdivided along the *y-z* plane. Finally, nodes that require further subdivision are subdivided along the *x-z* plane. This process is repeated in a cyclical manner until the appropriate maximum level of subdivision is attained.

In the best case, a region requiring one internal node and eight octree leaf nodes is represented by one internal node and two bintree leaf nodes. For the example described, in the worst case, a bintree of seven internal nodes and eight leaf nodes might be required. The average case is more difficult to define, so a pointer-based bintree representation may or may not be more compact than the corresponding pointer-based octree representation. However, with a linear bintree representation, the extra internal nodes become irrelevant and the need for two additional bits (for a 3D image) to represent the deepest 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 can be designed to work for data of arbitrary dimensionality.

### 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 for its relevance to the execution-time analysis of quadtree algorithms than for

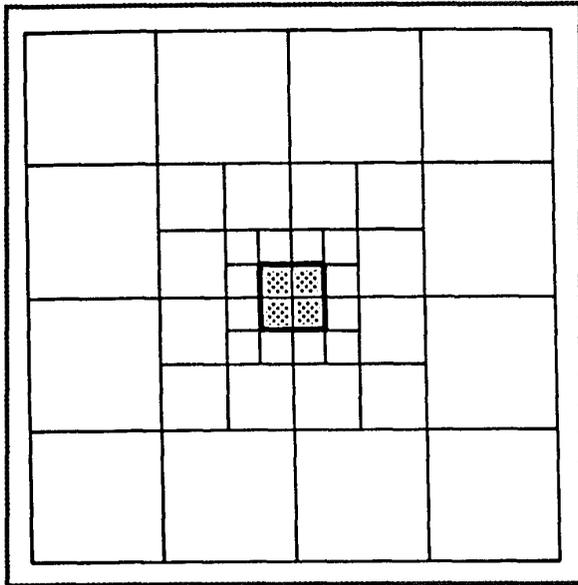


Figure 10. Example quadtree where the perimeter does not exceed the base 2 logarithm of the width of the image. The region in the image is assumed to consist of four pixels, each of unit width.

the amount of storage 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<sup>27,32</sup>), 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  $p$ , the number of nodes in the quadtree cannot exceed  $16 \cdot p - 11 + 16 \cdot q$ .

In all but the most pathological cases (see Figure 10 for an example) the region perimeter exceeds the base 2 logarithm of the width of the image space in which the region is presented. Therefore, the quadtree complexity theorem holds 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 2D array representation, when the resolution doubles, the size of the array quadruples. Therefore, asymptotically, quadtrees are arbitrarily more compact than 2D arrays; however, for moderate-size applications, constant factors need to be scrutinized more carefully. Figure 11 illustrates the relative growth of the two representations for a simple triangular region.

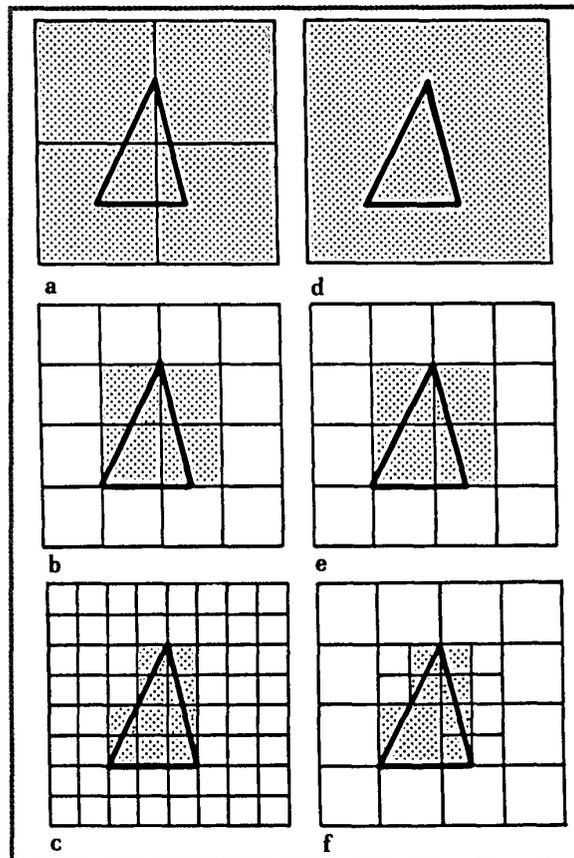
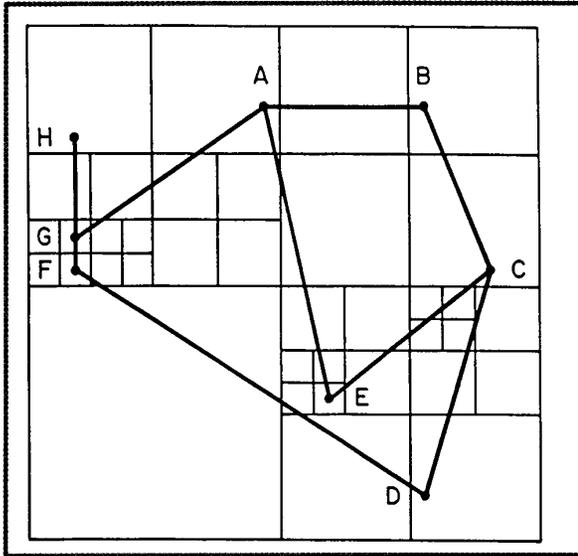


Figure 11. An illustration of the relative growth of the array and quadtree representations at different levels of resolution for a simple triangular region. Figures 11a through 11c are the array representations of the triangle at resolutions 1, 2, and 3. Figures 11d through 11f are the corresponding 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 is shown shaded). Note that the quadtree at resolution 1 (in Figure 11d) has just one node. The triangle overlaps each of the four blocks, and thus they have been merged.

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 (see Figure 9). 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



**Figure 12.** A vector data quadtree corresponding to the image of Figure 1a.

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. Dyer<sup>49</sup> has verified such expectations for randomly placed rectangles where a factor of 4 rather than 16 was found.

The quadtree complexity theorem applies to 3D data<sup>50</sup> where perimeter is replaced by surface area, as well as to higher dimensions for which, in all but pathological cases, it holds 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 has a direct impact on 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.

### Vector quadtree definition

The other type of data that we want to represent is vector data. There are a number of useful leaf criteria<sup>51</sup> for representing vector data using quadtrees. These criteria differ in the degree of the complexity of the image-space description versus the size of the hierarchy (i.e., the number of nodes in the quadtree). Choosing between the criteria is a matter of analyzing constants on specific computers 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 criterion that we shall use for vector data is termed a  $PM_1$  quadtree and is defined as follows:

- There can be at most one vertex in an image space.
- If there is a vertex in the image space, then all line segments in the image space must share that vertex.
- 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 12.

When a line segment passes through an image space, resulting in a  $q$ -edge, only its presence in the space is explicitly recorded.<sup>52</sup> 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; that is, 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.<sup>53</sup> 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 were a function of the number of line segments in the image space (thus making the size of the image-space hierarchy comparable with 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, it is not typical. In fact, usually 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 farther from both endpoints, the sizes 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.

### 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 polyhedrons. 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.<sup>54-56</sup> This approach has also been used in the context of the bintree representation of the octree.<sup>57</sup> However, it has the same problems as the analogous quadtree approach; that is, there are some features that cannot be represented exactly (thus they require a maximum-depth truncation similar to the edge quadtree<sup>22,24-26</sup>).

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<sup>25,58-61</sup> 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, having more than two faces meet at a common edge is permissible, although the situation cannot arise when modeling solids with Eulerian operators.<sup>2</sup> Nevertheless, it is plausible when 3D 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.<sup>62</sup> However, it should be clear that the vector

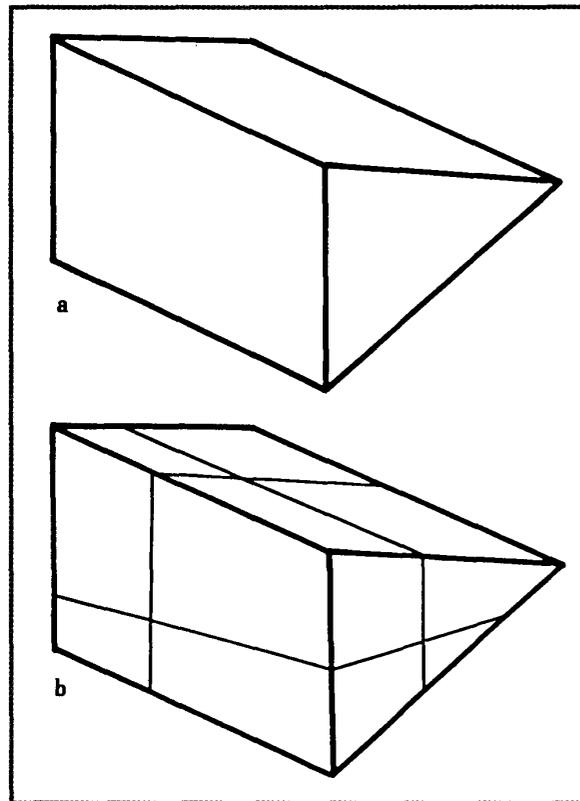


Figure 13. (a) Example 3D object and (b) its corresponding vector octree.

octree for a given image is much more compact than the corresponding raster octree. For example, a vector octree decomposition of the object in Figure 13a is shown in Figure 13b.

Vector octree techniques have also been extended to handle curvilinear surfaces. Primitives including cylinders and spheres have been used with a decomposition rule that limits the number of distinct primitives that can be associated with a leaf node.<sup>55,63</sup> Another approach<sup>64</sup> extends the concepts of face node, edge node, and vertex node to handle faces represented by biquadratic patches. Biquadratic patches enable 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.<sup>4</sup>

### Algorithms using quadtrees

Now 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, and polygon coloring. We also explain the concept of neighbor finding, which serves as a basis for many algorithms using quadtrees and octrees.

### Point location

Probably the simplest task to perform on raster data is determining the color of a given pixel. In the traditional raster representation, this task is accomplished by exactly one array access. In the raster quadtree, it 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 using 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 the section on quadtree/octree data structure implementations) that is equivalent to interleaving the binary coordinates of the lower left-hand 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 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. A pixel is located 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 this 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.

### 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. To handle this task, we must first determine the leaf that contains the indicated location. The first approach discussed in the section above 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 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 straightforward.

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<sup>65</sup> as well as pointers.<sup>66</sup> The latter is reported only for point data; however, the treatment of vector data differs from point data only in the formula used to calculate the distance from a point.

Using a pointer quadtree representation, the nearest primitive is found by a top-down recursive algorithm (the operation is also known as the *nearest neighbor problem*). 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. As we do so, at each level we search the subtrees that represent regions overlapping 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 farther away (since it is possible that a primitive

in them might make it unnecessary to search the subtrees that are farther away).

Consider, for example, Figure 14 and the task of finding the nearest neighbor of P in node 1. If we visit nodes in the order NW, NE, SW, and 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. Nevertheless, we still visit node 6, containing point B, which is closer than A, but now there is no need to visit node 7. Unwinding one more level reveals that because of 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 calculating the nearest neighbor is not necessary, as long as a "close" neighbor is found. For example, in a plotting application we want to reduce the wasted pen motions (motions of the pen that do not involve drawing). In particular, we require a real-time algorithm, in the sense that we want to minimize the total time required both to preprocess the drawing and actually plot it. In such a case, a quadtree heuristic for calculating the nearest neighbor has been found useful.<sup>66</sup> For example, in Figure 14 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).

### Set-theoretic operations and image transformations

The basic set-theoretic operations on quadtrees were first described by Hunter and Steiglitz<sup>27,67</sup> (see also Shneider<sup>68</sup>) for pointer-based quadtrees. Gargantini<sup>42</sup> raises the issue of performing these operations on linear quadtrees that are not aligned. Hunter and Steiglitz<sup>27,67</sup> and Peters<sup>69</sup> consider the related 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. Our implementation of the transformations uses the same technique as that of Meagher<sup>30</sup> for shifting and rotating images represented by octrees (see Jackins and Tanimoto<sup>28</sup> for a related approach), although he does not use the analogy with the set-theoretic operations.

Alternative implementations compute the transformed location of each black node in the original quadtree (octree) and insert it in the new quadtree (octree).<sup>70-72</sup> The worst-case analysis of their execution time is not as good as that of the methods that we discuss, although, in practice, the actual execution times seem to

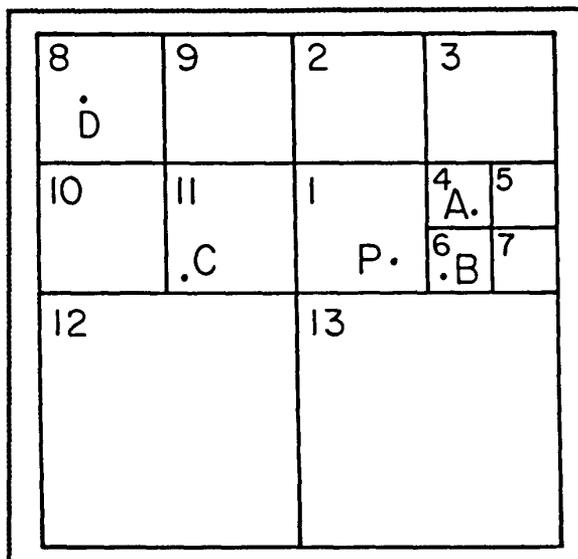
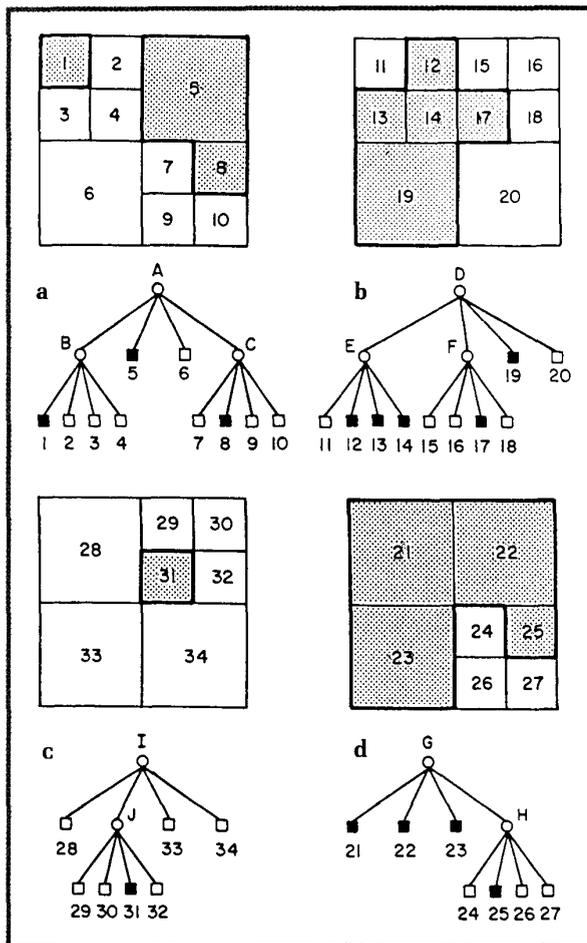


Figure 14. Example illustrating the neighboring object problem: P is the location of the pointing device. The nearest object is represented by point B in node 6.

be dominated by implementation-dependent constant factors. Van Lierop<sup>73</sup> and Walsh<sup>74</sup> discuss algorithms for linear quadtrees that have aspects of both of these alternatives, while Yamaguchi et al.<sup>75</sup> do the same for linear octrees.

### 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 makes sense only 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 method is to traverse the input quadtree in preorder, simultaneously building the resultant quadtree. If the mapping between colors is not invertible, then merging some nodes in the resulting quadtree may be necessary. However, this can be done naturally during the preorder traversal of the input tree.



**Figure 15. Example of set-theoretic operations.** Figures 15a and 15b show sample images and their quadtrees. Figure 15c shows the intersection of the images in Figures 15a and 15b; Figure 15d shows their union.

Thus, under either condition, the quadtree recoloring algorithm executes in time proportional to the size of the input quadtree.

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. The intersection of the quadtrees of Figures 15a and 15b results in Figure 15c. 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:

- If either input quadtree node is white, then the output quadtree node is white.

- If both input quadtree nodes are black, then the output quadtree node is black.
- 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.
- 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 15a and 15b respectively). Note that for the intersection operation, a merge of four black leaf nodes is impossible, and thus we must check only for the mergibility of white leaf nodes.

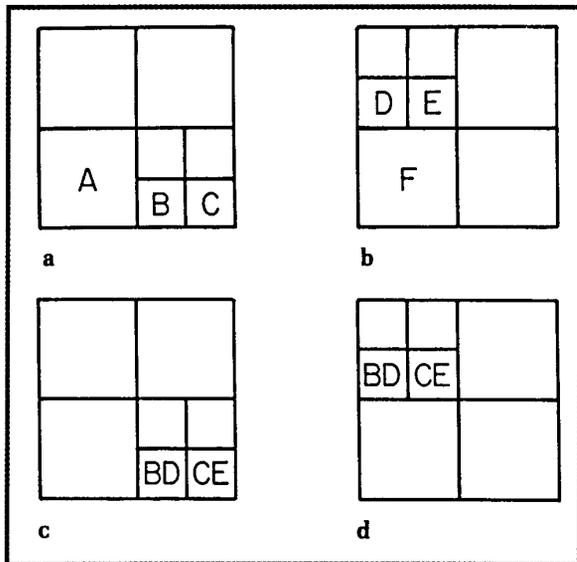
The worst-case execution time of this algorithm is proportional to the sum of the number of nodes in the two input quadtrees. Note that as a result of the first and third actions, it is possible for the intersection algorithm to visit fewer nodes than the sum of the nodes in the two input quadtrees.

The union operation is implemented easily by applying DeMorgan's law to the above intersection algorithm. For example, Figure 15d is the result of the union of the quadtrees of Figures 15a and 15b. 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).

Note also 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.

#### **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 left-hand corners correspond to different positions. For example, consider the  $4 \times 4$  quadtrees shown in Figures 16a and 16b, whose lower left-hand corners are at locations (0,2) and (2,0) respectively. This alignment information is stored separately from the quadtree. Thus, to translate or rotate a quadtree, we need only to update the alignment information. However, when two quadtrees of differing alignment must be operated upon simultaneously (e.g.,

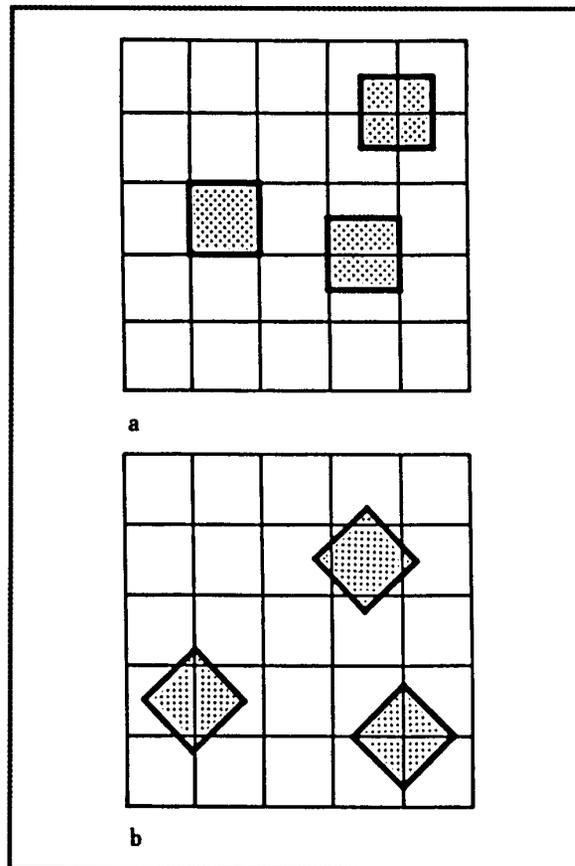


**Figure 16.** Example of rectilinear unaligned-quadtree intersection: (a) a  $4 \times 4$  quadtree with a lower left-hand corner at  $(0,2)$ , (b) a  $4 \times 4$  quadtree with a lower left-hand corner at  $(2,0)$ , (c) the intersection of Figures 16a and 16b with Figure 16a as the aligned quadtree, (d) the intersection of Figures 16a and 16b with Figure 16b as the aligned quadtree.

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 17a) and those four squares will be neighbors (i.e., they form a  $2w \times 2w$  square). We refer to this as the *rectilinear unaligned-quadtree problem*. When 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 17b). These six or fewer 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.

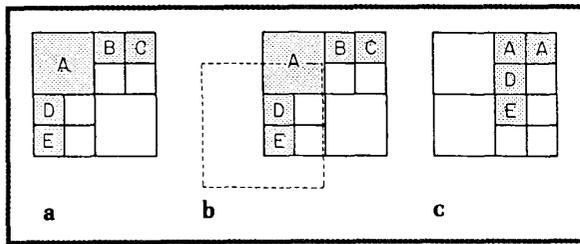
To handle the rectilinear unaligned-quadtree intersection problem, we adopt the convention that the output quadtree will be aligned with the first quadtree. We refer to the first quadtree as the aligned quadtree and to the second quadtree as the unaligned quadtree. For example, intersecting the quadtrees in Figures 16a and 16b so that the quadtree of Figure 16a is the aligned quadtree



**Figure 17.** Examples showing how many squares can be overlapped when a square of size  $w \times w$  is overlaid on a grid of squares such that each square is of size  $w \times w$ , so that the square and the grid are (a) rectilinearly unaligned and (b) generally unaligned.

yields the quadtree of Figure 16c. On the other hand, if the quadtree of Figure 16b is the aligned quadtree, then the result is represented by the quadtree of Figure 16d.

When intersecting aligned quadtrees (see section on aligned quadtrees above), 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 17a 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 eas-



**Figure 18.** Example of shifting a  $4 \times 4$  quadtree by two units to the right and one 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 Figure 18a.

ily 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 from 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*. This technique is the same as that used by Meagher<sup>30</sup> for shifting images represented by octrees. See, for example, Figure 18, where the  $4 \times 4$  quadtree of Figure 18a is shifted to the right by two units and up by one unit. The position of the aligned quadtree relative to the unaligned quadtree is shown in Figure 18b using broken lines, while Figure 18c 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.

#### General unaligned quadtrees and rotations

The general unaligned-quadtree algorithm is analogous to the algorithm discussed above 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 17b). 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. The method we describe is the same as

that used by Meagher<sup>30</sup> for rotating images represented by octrees, although he does not draw the analogy with the unaligned set 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*.

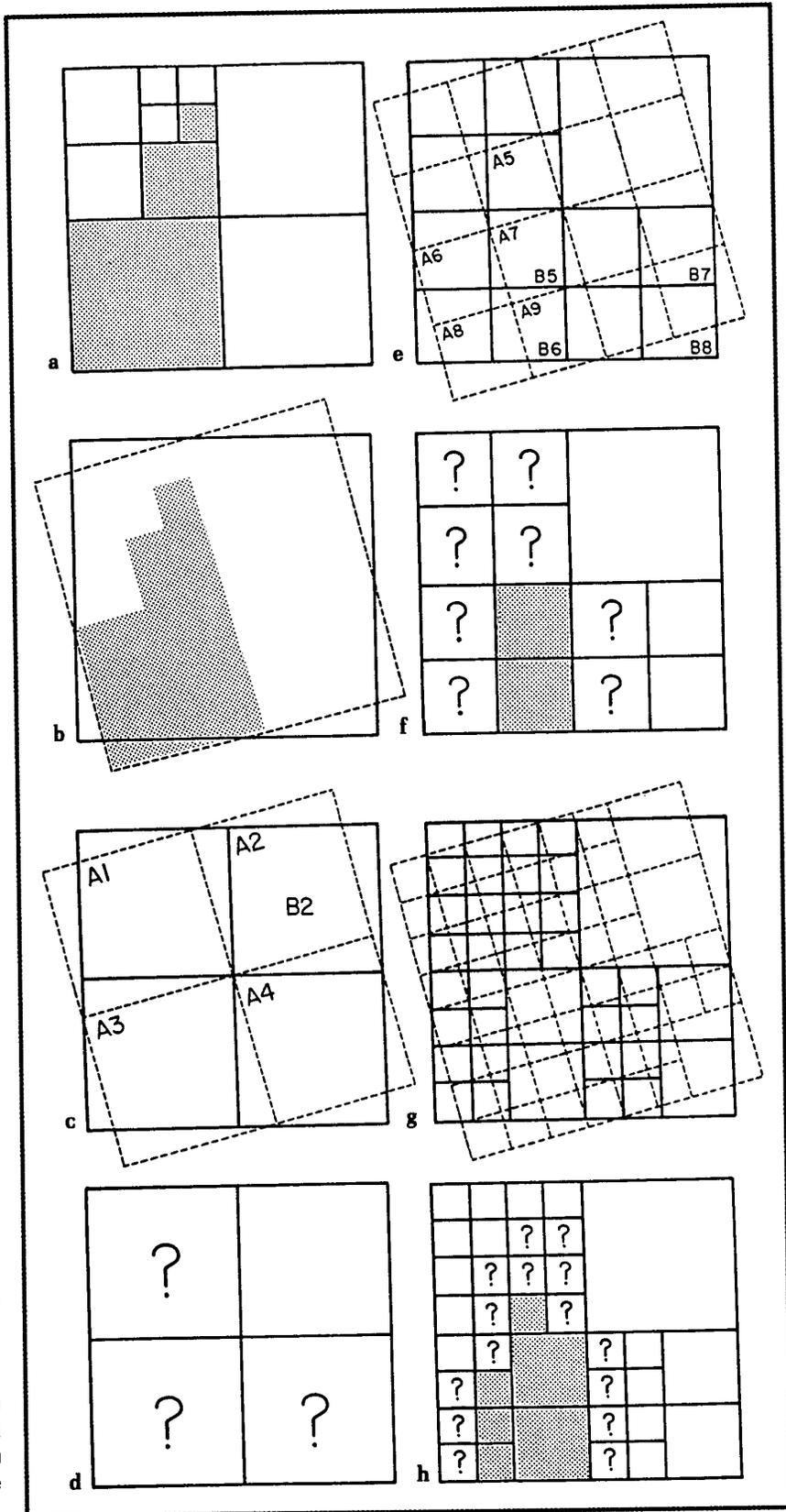
In the following we describe the rotation of the quadtree of Figure 19a, termed *A*, by 16 degrees in a counterclockwise direction about its origin. Black block *B* has been rotated by 16 degrees in the clockwise direction about the origin of *A* (see Figure 19b). We use broken lines to depict the decomposition of *A* and solid lines to depict the decomposition of *B*.

The rotation algorithm first determines whether *B* is a terminal node by checking whether the maximum of six nodes of equal size in *A* that cover it are of the same color. If they are, 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 19c, and its result is given in Figure 19d. Notice the "?" symbol that indicates the block will be subdivided further. The NE quadrant in Figure 19d (corresponding to the block labeled *B2* in Figure 19c) is white because the blocks in *A* that overlap it (just the two blocks labeled *A2* and *A4* in Figure 19c) are white. Before 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 "?" in Figure 19d as well as blocks *A1*, *A2*, *A3*, and *A4* in Figure 19c to obtain Figure 19e. Again, we now check each of the newly obtained subblocks of *B* to see if they are covered by subblocks of *A* of the same color. In this case we find that this is true for blocks *B5* and *B6* in Figure 19e (i.e., they are covered by black subblocks *A5*, *A6*, *A7*, *A8*, and *A9*), as well as blocks *B7* and *B8*. The result is given in Figure 19f, with "?" denoting that the block will be decomposed further. One more level of decomposition is depicted in Figure 19g and the resulting rotated quadtree is shown in Figure 19h. Checking to determine whether any of the subdivided blocks in *B* have identically colored sons reveals that the four blocks of the NW son of the NW quadrant in Figure 19h 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 partic-

Figure 19. Example of rotation. Broken lines depict the decomposition of the unaligned quadtree and solid lines depict the decomposition of the aligned quadtree: (a) sample quadtree, (b) rotation of Figure 19a by 16 degrees in a counterclockwise direction about its origin, (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.



ular, a rotated square usually cannot be represented accurately by a collection of rectilinear squares. However, when we rotate by 90 degrees, then the rotation is invertible. In such a case, the algorithm traverses the tree in preorder and rotates the pointers at each node. For a counterclockwise rotation by 90 degrees, the NW, NE, SE, and SW sons become SW, NW, NE, and SE sons, respectively, at each level of the quadtree.

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.

### 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 so 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 11, we would first see Figure 11d, then Figure 11e, and finally Figure 11f. The scheme facilitates browsing a database of images. One successful approach<sup>33,76-78</sup> is to transmit the nodes of a raster quadtree in breadth-first order, so that large leaf nodes are seen first.

### 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<sup>79</sup> uses 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 the section on point location. 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, makes use of only father links and computes a direct path to the neighbor by following links in the tree. This method is called *bottom-up neighbor finding* and has been shown to require an average of four links to be followed for each neighbor sought.<sup>80,81</sup>

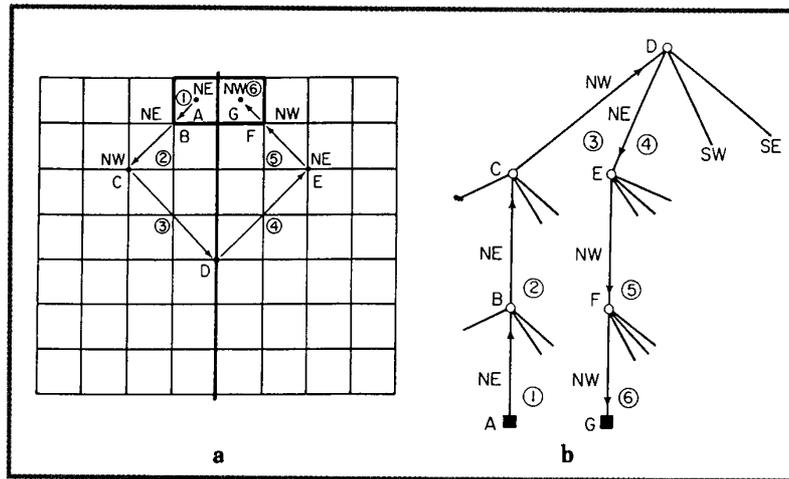
Here we limit ourselves to neighbors in the horizontal and vertical direction that are of a size equal to or greater than the node whose neighbor is being sought. Neighbors in the diagonal direction have been handled elsewhere.<sup>80</sup> Finding a node's neighbor in a specified horizontal or vertical direction requires us to follow father links until a common ancestor of the two nodes is found. Once it has been 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 20. For example, when attempting in Figure 20 to locate the eastern neighbor of node A (the neighbor is node G), 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 in Figure 20, and its eastern neighbor of greater or equal size, 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. Similar techniques are used to find neighbors in octrees.<sup>82</sup> The difference is that there are 26 different directions.

### Constructing quadtrees

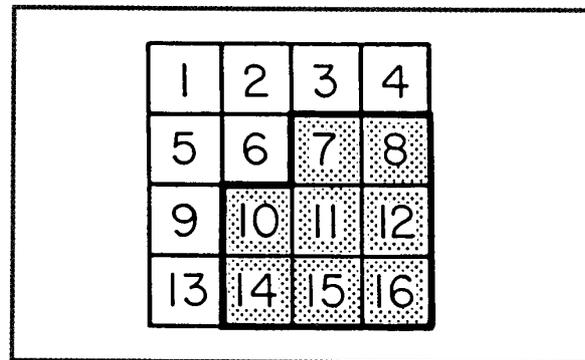
Before we can operate on images represented by quadtrees, we must first build the quadtrees. The process requires conversion between a number of different data formats and the quadtree. Here we briefly describe the construction of raster quadtrees from raster data and vector data. The construction of vector quadtrees from either type of data can be performed in an analogous manner.

**Figure 20. The process of locating the eastern neighbor of node A (i.e., node G): (a) block decomposition, (b) tree representation.**



The algorithm for building a raster quadtree from a 2D array can be derived directly from the definition of the raster quadtree.<sup>83</sup> When building a quadtree from raster data presented in raster scan order (i.e., the array is processed row by row),<sup>84</sup> we use the bottom-up neighbor-finding algorithm to move through the quadtree in the order in which the data is encountered. For example, considering the quadtree of Figure 6 as a 4 × 4 image means that its image elements are examined in the order indicated in Figure 21. 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 whether nodes should be merged. This can be avoided by predictive techniques that 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<sup>85,86</sup> rather than by the number of pixels. However, this algorithm does require an auxiliary data structure (which can be implemented by a fixed-size array<sup>85,86</sup>) of a 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. Other-



**Figure 21. Raster scanning order for the image of Figure 6.**

wise, 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.

As an alternative 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<sup>87</sup>) using a traditional line-drawing algorithm.<sup>88</sup> Next, we follow the path formed by the chain codes of the line segments creating black pixel-size leaf nodes.<sup>89</sup> This is done with the bottom-up neighbor-finding algorithm.

Average-case analysis for the execution time of the chain-code-to-raster-quadtree algorithm is linear in the length of the chain code, as shown by analysis<sup>89</sup> in conjunction with the quadtree complexity theorem. Moreover, preprocessing the chain code shows that the worst-case analysis of this algorithm is also linear in the length of the chain code.<sup>90</sup> Neighbor-finding methods

have also been used to construct chain codes from quadtrees,<sup>91</sup> as well as 2D arrays in a row-by-row manner.<sup>92</sup>

### 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<sup>13</sup> 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 whether the color of the current pixel is equal to that of the original color of the start pixel. If it is, 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.<sup>27,32</sup> 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.<sup>93</sup> 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. Such 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.<sup>48,94</sup>

The third technique<sup>90</sup> 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 use a special kind of neighbor finding; that is, they perform a preorder traversal of a quadtree and require the examination of 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.<sup>90,95-97</sup> Top-down neighbor finding is based on the observation that the neighbor of a node is either a sibling of the node or a child of a neighbor of the node's father. Thus, the neighbors of a node can be transmitted as parameters to the function 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.<sup>96</sup>

### Concluding remarks

We have presented an overview of the fundamentals behind the use in computer graphics of such hierarchical data structures as the quadtree and the octree. More advanced applications with an emphasis on the octree and display methods will be discussed in a companion article to appear in the July issue of *CG&A*.<sup>4</sup> ■

### Acknowledgment

The support of the National Science Foundation under grant DCR\86\05557 is gratefully acknowledged.

### References

1. I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *ACM Computing Surveys*, Mar. 1974, pp. 1-55.
2. B.G. Baumgart, "Winged-Edge Polyhedron Representation," Tech. Report STAN-CS-320, Computer Science Dept., Stanford Univ., Stanford, Calif., 1972.
3. J.H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *CACM*, Oct. 1976, pp. 547-554.
4. H. Samet and R.E. Webber, "Hierarchical Data Structures and Algorithms for Computer Graphics, Part II: Applications," To appear in *CG&A*, July 1988.
5. H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, June 1984, pp. 187-260.
6. 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, pp. 181-201.
7. H. Samet, *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*, to appear, 1989.
8. H.A. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, Mass., 1969.
9. I.E. Sutherland, "Sketchpad, A Man-Machine Communication System," *Proc. SJCC*, Detroit, 1963, pp. 329-346.
10. J.C. Gray, "Compound Data Structure for Computer Aided Design: A Survey," *Proc. 22nd Nat'l Conf. ACM*, ACM, New York, 1967, pp. 355-365.
11. K.A. Lantz and W.I. Nowicki, "Structured Graphics for Distributed Systems," *ACM Trans. Graphics*, Jan. 1984, pp. 23-51.
12. American National Standards Institute Committee X3H31, American National Standard for the Functional Specification of the Programmer's Hierarchical Interactive Graphics System (PHIGS), ANSI Standard X3H31/85-05 X3H31/85-21, American National Standards Institute, New York, 1985.
13. D.R. Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill, New York, 1985.
14. S.M. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics (Proc. SIGGRAPH)*, July 1980, pp. 110-116.
15. H. Weghorst, G. Hooper, and D.P. Greenberg, "Improved Computational Methods for Ray Tracing," *ACM Trans. Graphics*, Jan. 1984, pp. 52-69.

16. H. Fuchs, G.D. Abram, and E.D. Grant, "Near Real-Time Shaded Display of Rigid Objects," *Computer Graphics (Proc. SIGGRAPH)*, July 1983, pp. 65-72.
17. D.H. Ballard, "Strip Trees: A Hierarchical Representation for Curves," *CACM*, May 1981, pp. 310-321 (see also corrigendum, *CACM*, Mar. 1982, pp. 213).
18. N. Ahuja, "On Approaches to Polygonal Decomposition for Hierarchical Image Representation," *Computer Vision, Graphics, and Image Processing*, Nov. 1983, pp. 200-214.
19. 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*, Nov. 1983, pp. 211-220.
20. L. Gibson and D. Lucas, "Vectorization of Raster Images Using Hierarchical Methods," *Computer Graphics and Image Processing*, Sept. 1982, pp. 82-89.
21. A. Klinger, "Patterns and Search Statistics," in *Optimizing Methods in Statistics*, J.S. Rustagi, ed., Academic Press, New York, 1971, pp. 303-337.
22. H. Samet, A. Rosenfeld, C.A. Shaffer, and R.E. Webber, "A Geographic Information System Using Quadrees," *Pattern Recognition*, Nov./Dec. 1984, pp. 647-656.
23. C.A. Shaffer, H. Samet, and R.C. Nelson, "QUILT: A Geographic Information System Based on Quadrees," Tech. Report TR-1885, Computer Science Dept., Univ. of Maryland, College Park, Md., 1987.
24. M. Shneier, "Two Hierarchical Linear Feature Representations: Edge Pyramids and Edge Quadrees," *Computer Graphics and Image Processing*, Nov. 1981, pp. 211-224.
25. D. Ayala, P. Brunet, R. Juan, and I. Navazo, "Object Representation by Means of Nonminimal Division Quadrees and Octrees," *ACM Trans. Graphics*, Jan. 1985, pp. 41-59.
26. J.E. Warnock, "A Hidden Surface Algorithm for Computer Generated Half Tone Pictures," Tech. Report TR 4-15, Computer Science Dept., Univ. of Utah, Salt Lake City, 1969.
27. G.M. Hunter, "Efficient Computation and Data Structures for Graphics," doctoral dissertation, Princeton Univ., Princeton, N.J., 1978.
28. C.L. Jackins and S.L. Tanimoto, "Oct-Trees and Their Use in Representing Three-Dimensional Objects," *Computer Graphics and Image Processing*, Nov. 1980, pp. 249-270.
29. D. Meagher, "Geometric Modeling Using Octree Encoding," *Computer Graphics and Image Processing*, June 1982, pp. 129-147.
30. D.R. Reddy and S. Rubin, "Representation of Three-Dimensional Objects," Tech. Report CMU-CS-78-113, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, 1978.
31. D.E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, Mass., 1975.
32. G.M. Hunter and K. Steiglitz, "Operations on Images Using Quad Trees," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Apr. 1979, pp. 145-153.
33. E. Kawaguchi and T. Endo, "On a Method of Binary Picture Representation and Its Application to Data Compression," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Jan. 1980, pp. 27-35.
34. E. Kawaguchi, T. Endo, and M. Yokota, "DF-Expression of Binary-Valued Picture and Its Relation to Other Pyramidal Representations," *Proc. Fifth Int'l Conf. Pattern Recognition*, CS Press, Los Alamitos, Calif., 1980, pp. 822-827.
35. E. Kawaguchi, T. Endo, and J. Matsunaga, "Depth-First Expression Viewed from Digital Picture Processing," *IEEE Trans. Pattern Analysis and Machine Intelligence*, July 1983, pp. 373-384.
36. M.A. Oliver and N.E. Wiseman, "Operations on Quadtree-Encoded Images," *Computer J.*, Feb. 1983, pp. 83-91.
37. M.A. Oliver and N.E. Wiseman, "Operations on Quadtree Leaves and Related Image Areas," *Computer J.*, Nov. 1983, pp. 375-380.
38. G.M. Morton, "A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing," IBM Ltd., Ottawa, Canada, 1966.
39. I. Gargantini, "An Effective Way to Represent Quadrees," *CACM*, Dec. 1982, pp. 905-910.
40. I. Gargantini, "Linear Octrees for Fast Processing of Three-Dimensional Objects," *Computer Graphics and Image Processing*, Dec. 1982, pp. 365-374.
41. I. Gargantini, "Detection of Connectivity for Regions Represented by Linear Quadrees," *Computers and Mathematics with Applications*, Vol. 8, No. 4, 1982, pp. 319-327.
42. I. Gargantini, "Translation, Rotation, and Superposition of Linear Quadrees," *Int'l J. Man-Machine Studies*, Mar. 1983, pp. 253-263.
43. 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, pp. 105-133.
44. 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*, Apr. 1985, pp. 56-69.
45. H. Samet and R.E. Webber, "A Comparison of the Space Requirements of Multi-Dimensional Quadtree-Based File Structures" Tech. Report TR-1711, Computer Science Dept., Univ. of Maryland, College Park, Md., 1986.
46. K. Knowlton, "Progressive Transmission of Grey-Scale and Binary Pictures by Simple, Efficient, and Lossless Encoding Schemes," *Proc. IEEE*, July 1980, pp. 885-896.
47. M. Tamminen, "Comment on Quad- and Octrees," *CACM*, Mar. 1984, pp. 248-249.
48. H. Samet and M. Tamminen, "Efficient Component Labeling of Images of Arbitrary Dimension," Tech. Report TR-1480, Computer Science Dept., Univ. of Maryland, College Park, Md., 1985. Also to be published in *IEEE Trans. Pattern Analysis and Machine Intelligence*.
49. C.R. Dyer, "The Space Efficiency of Quadrees," *Computer Graphics and Image Processing*, Aug. 1982, pp. 335-348.
50. D. Meagher, "Octree Encoding: A New Technique for the Representation, the Manipulation, and Display of Arbitrary 3-D Objects by Computer," Tech. Report IPL-TR-80-111, Image Processing Laboratory, Rensselaer Polytechnic Inst., Troy, N.Y., 1980.
51. H. Samet and R.E. Webber, "Storing a Collection of Polygons Using Quadrees," *ACM Trans. Graphics*, July 1985, pp. 182-222. Also published in *Proc. IEEE Computer Vision and Pattern Recognition Conf.*, CS Press, Los Alamitos, Calif., 1983, pp. 127-132.
52. R.C. Nelson and H. Samet, "A Consistent Hierarchical Representation for Vector Data," *Computer Graphics (Proc. SIGGRAPH)*, Aug. 1986, pp. 197-206.
53. H. Samet, C.A. Shaffer, and R.E. Webber, "Digitizing the Plane with Cells of Non-Uniform Size," *Information Processing Letters*, Apr. 1987, pp. 369-375.
54. A.S. Glassner, "Space Subdivision for Fast Ray Tracing," *CG&A*, Oct. 1984, pp. 15-22.
55. G. Wyvill and T.L. Kunii, "A Functional Model for Constructive Solid Geometry," *The Visual Computer*, July 1985, pp. 3-14.
56. F.W. Jansen, "Data Structures for Ray Tracing," in *Data Structures for Raster Graphics*, F.J. Peters, L.R.A. Kessener, and M.L.P. van Lierop, eds., Springer-Verlag, Berlin, 1986, pp. 57-73.
57. M.R. Kaplan, "Space-Tracing: A Constant Time Ray-Tracer," *Uses of Spatial Coherence in Ray-Tracing*, tutorial notes, SIGGRAPH 85, ACM, New York, 1985.
58. I. Carlbom, I. Chakravarty, and D. Vanderschel, "A Hierarchical Data Structure for Representing the Spatial Decomposition of 3-D Objects," *CG&A*, Apr. 1985, pp. 24-31.
59. K. Fujimura and T.L. Kunii, "A Hierarchical Space Indexing Method," *Proc. Computer Graphics 85*, Springer-Verlag, Tokyo, 1985, pp. 21-33.
60. K.M. Quinlan and J.R. Woodwork, "A Spatially-Segmented Solids Database—Justification and Design," *Proc. CAD 82 Conf.*, Brighton, UK, 1982, pp. 126-132.
61. D.J. Vanderschel, "Divided Leaf Octal Trees," research note, Schlumberger-Doll Research, Ridgefield, Conn., 1984.

62. I. Navazo, *Contribucio a les tecniques de modelat geometric d'objectes poliedrics usant la codificacio amb arbres octals*, doctoral dissertation, Escola Tecnica Superior d'Enginyers Industrials, Universitat Politecnica de Barcelona, Barcelona, Spain, 1986.
63. A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray-Tracing System," *CG&A*, Apr. 1986, pp. 16-26.
64. I. Navazo, D. Ayala, and P. Brunet, *A Geometric Modeller Based on the Exact Octree Representation of Polyhedra*, Escola Tecnica Superior d'Enginyers Industrials, Universitat Politecnica de Barcelona, Barcelona, Spain, 1986.
65. D.J. Abel and J.L. Smith, "A Simple Approach to the Nearest-Neighbor Problem," *Australian Computer J.*, Nov. 1984, pp. 140-146.
66. D.P. Anderson, "Techniques for Reducing Pen Plotting Time," *ACM Trans. Graphics*, July 1983, pp. 197-212.
67. G.M. Hunter and K. Steiglitz, "Linear Transformation of Pictures Represented by Quad Trees," *Computer Graphics and Image Processing*, July 1979, pp. 289-296.
68. M. Shneier, "Calculations of Geometric Properties Using Quad-trees," *Computer Graphics and Image Processing*, July 1981, pp. 296-302.
69. F.J. Peters, "An Algorithm for Transformations of Pictures Represented by Quadtrees," *Computer Vision, Graphics, and Image Processing*, Dec. 1985, pp. 397-403.
70. N. Ahuja and C. Nash, "Octree Representations of Moving Objects," *Computer Vision, Graphics, and Image Processing*, May 1984, pp. 207-216.
71. W. Osse and N. Ahuja, "Efficient Octree Representation of Moving Objects," *Proc. Seventh Int'l Conf. Pattern Recognition*, CS Press, Los Alamitos, Calif., 1984, pp. 821-823.
72. J. Weng and N. Ahuja, "Octrees of Objects in Arbitrary Motion: Representation and Efficiency," *Computer Vision, Graphics, and Image Processing*, Aug. 1987, pp. 167-185.
73. M.L.P. van Lierop, "Geometrical Transformations on Pictures Represented by Leafcodes," *Computer Vision, Graphics, and Image Processing*, Jan. 1986, pp. 81-98.
74. T.R. Walsh, "Efficient Axis-Translation of Binary Digital Pictures by Blocks in Linear Quadtree Representation," *Computer Vision, Graphics, and Image Processing*, Mar. 1988, pp. 282-292.
75. K. Yamaguchi, T.L. Kuniti, K. Fujimura, and H. Toriya, "Octree-Related Data Structures and Algorithms," *CG&A*, Jan. 1984, pp. 53-59.
76. F.S. Hill, Jr., S. Walker, Jr., and F. Gao, "Interactive Image Query System Using Progressive Transmission," *Computer Graphics (Proc. SIGGRAPH)*, July 1983, pp. 323-330.
77. H. Samet, "Data Structures for Quadtree Approximation and Compression," *CACM*, Sept. 1985, pp. 973-993.
78. K.R. Sloan, Jr., and S.L. Tanimoto, "Progressive Refinement of Raster Images," *IEEE Trans. Computers*, Nov. 1979, pp. 871-874.
79. A. Klinger and M.L. Rhodes, "Organization and Access of Image Data by Areas," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Jan. 1979, pp. 50-60.
80. H. Samet, "Neighbor Finding Techniques for Images Represented by Quadtrees," *Computer Graphics and Image Processing*, Jan. 1982, pp. 37-57.
81. H. Samet and C.A. Shaffer, "A Model for the Analysis of Neighbor Finding in Pointer-Based Quadtrees," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Nov. 1985, pp. 717-720.
82. H. Samet, "Neighbor Finding in Images Represented by Octrees," Tech. Report. TR-1968, Computer Science Dept., Univ. of Maryland, College Park, Md., 1988.
83. H. Samet, "Region Representation: Quadtrees from Binary Arrays," *Computer Graphics and Image Processing*, May 1980, pp. 88-93.
84. H. Samet, "An Algorithm for Converting Rasters to Quadtrees," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Jan. 1981, pp. 93-95.
85. C.A. Shaffer, *Application of Alternative Quadtree Representations*, doctoral dissertation and Tech. Report TR-1672, Computer Science Dept., Univ. of Maryland, College Park, Md., 1986.
86. C.A. Shaffer and H. Samet, "Optimal Quadtree Construction Algorithms," *Computer Vision, Graphics, and Image Processing*, Mar. 1987, pp. 402-419.
87. H. Freeman, "Computer Processing of Line-Drawing Images," *ACM Computing Surveys*, Mar. 1974, pp. 57-97.
88. J.E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems J.*, Vol. 4, No. 1, pp. 25-30.
89. H. Samet, "Region Representation: Quadtrees from Boundary Codes," *CACM*, Mar. 1980, pp. 163-170.
90. R.E. Webber, *Analysis of Quadtree Algorithms*, doctoral dissertation and Tech. Report TR-1376, Computer Science Dept., Univ. of Maryland, College Park, Md., 1984.
91. C.R. Dyer, A. Rosenfeld, and H. Samet, "Region Representation: Boundary Codes from Quadtrees," *CACM*, Mar. 1980, pp. 171-179.
92. H. Samet, "Algorithms for the Conversion of Quadtrees to Rasters," *Computer Vision, Graphics, and Image Processing*, Apr. 1984, pp. 1-16.
93. H. Samet, "Connected Component Labeling Using Quadtrees," *J. of the ACM*, July 1981, pp. 487-501.
94. H. Samet and M. Tamminen, "A General Approach to Connected Component Labeling of Images," Tech. Report TR-1649, Computer Science Dept., Univ. of Maryland, College Park, Md., 1986. Also published in *Proc. IEEE Computer Vision and Pattern Recognition Conf.*, CS Press, Los Alamitos, Calif., 1986, pp. 312-318.
95. H. Samet and R.E. Webber, "On Encoding Boundaries with Quad-trees," Tech. Report TR-1162, Computer Science Dept., Univ. of Maryland, College Park, Md., 1982.
96. C.L. Jackins and S.L. Tanimoto, "Quad-Trees, Oct-Trees, and k-Trees—A Generalized Approach to Recursive Decomposition of Euclidean Space," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Sept. 1983, pp. 533-539.
97. H. Samet, "A Top-Down Quadtree Traversal Algorithm," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Jan. 1985, pp. 94-98.



**Hanan Samet** is a professor of computer science at the University of Maryland, where he serves as the director of the graduate program in computer science. He is also a member of the Computer Vision Laboratory of the Center for Automation Research and has an appointment in the University of Maryland Institute for Advanced Computer Studies. His research interests are data structures, computer graphics, geographic information systems, computer vision, robotics, programming languages, artificial intelligence, and database management systems.

Samet received his BS in engineering from UCLA and his MS in operations research and MS and PhD in computer science from Stanford University. He is a senior member of IEEE and is a member of ACM.

Samet can be reached at the Computer Science Department, University of Maryland, College Park, MD 20742.



**Robert E. Webber** is an assistant professor of computer science at Rutgers University. His research interests are image synthesis, geographic information systems, analysis of algorithms, and discrete geometry.

Webber received his BS, MS, and PhD in computer science from the University of Maryland. He is a member of ACM, IEEE, and NCGA.

Webber can be contacted at the Department of Computer Science, Rutgers University, Busch Campus, New Brunswick, NJ 08903.