Chapter 1

OBJECT REPRESENTATIONS

Hanan Samet, University of Maryland

Abstract A brief overview of interior-based and boundary-based object representations is presented. The emphasis is on the importance of aggregation of similarly-valued elements, which was recognized early on by Azriel Rosenfeld.

1. INTRODUCTION

Object representation has always played an important role in computer vision research, since the success of computer vision systems depends on the ability to process the data. This ability is often measured in terms of efficient execution, which is a function of the appropriateness of the underlying representation. Azriel Rosenfeld was one of the first to recognize this, as evidenced by his pioneering research reviewed below, and by Chapter 11 of *Digital Picture Processing*, an important text in the field, co-authored with Kak [47]. This chapter laid the groundwork for Samet's two texts on spatial data structures [58, 59].

Rosenfeld's earliest work was centered on the development of algorithms for geometric processing of images represented primarily by binary arrays. He developed some of the earliest algorithms for such important operations as connected component labeling [48], the principles of which still form the basis of current approaches.

Rosenfeld also recognized the importance of aggregation of similarly-valued image elements and the fact that it would lead to faster algorithms. This was manifested by his work on algorithms that use the medial axis transformation (MAT) [38, 45] and the region quadtree [61, 62]. The latter initiated the work of this author and others by developing the first algorithm for converting a region quadtree to a chain code representation [13]. This work led to the development of several algorithms for showing the interchangeability of the region quadtree with more tra-

ditional representations such as arrays and rasters [50, 52, 56, 69], as well as further work on chain codes [51, 75]. These results led to the work of Rosenfeld, Samet, and their students on using the region quadtree as the underlying representation in Geographic Information Systems (GIS) [63, 64, 65].

An important ramification of this work is that it provided the impetus to obtaining additional evidence to support Hunter's [26] result that aggregation techniques such as the region quadtree don't just save space, but, more importantly, lead to faster algorithms whose execution time depends on the number of blocks rather than the sizes of the blocks (for additional details, see the discussion in Section 3). This was demonstrated by Samet in his algorithm for connected component labeling [11, 53] which was a direct adaptation of the earlier algorithm by Rosenfeld [48]. Rosenfeld's student Dyer [12] obtained an analogous result for the genus or Euler number of an image, adapting the algorithm of Minsky and Papert [34]. In particular, Dyer showed that the genus can be computed in terms of the number of object quadtree blocks and their adjacencies rather than the number of object pixels and their adjacencies. This reinforced the principle that the number of quadtree blocks is the most important factor in the execution time rather than their sizes.

In a similar vein, Rosenfeld was one of the first to realize the utility of the pyramid representation [43]. This is a multiresolution representation introduced by Tanimoto and Pavlidis [73]. It is in contrast to the region quadtree, which is a variable-resolution representation. Each has its strengths. In particular, the region quadtree is useful for responding to location-based queries such as "given a location, what object is there?". On the other hand, for a feature-based query such as "given a feature, find its location", the region quadtree requires that we examine every block to see if the feature is present. The pyramid stores summary information in the nonleaf nodes of the data below them. Thus the summary can be used to determine if the tree structure should be descended further. If the feature is not noted as present in a node, it cannot be present in its subtrees and there is no need to descend the tree further. Rosenfeld and his colleagues developed many algorithms exploiting the pyramid for basic image processing operations [8, 24, 25, 39, 44, 46].

The above work has stimulated research in object representation in many diverse fields such as computer graphics, pattern recognition, and database management systems. In the rest of this chapter we provide a brief review of some of the most common representations and demonstrate the influence of Azriel Rosenfeld. We assume that the objects are connected although their environment need not be. The objects and

their environment are usually decomposed into collections of more primitive elements (termed *cells*) each of which has a location in space, a size, and a shape. These elements can either be subobjects of varying shape (e.g., a table consists of a flat top in the form of a rectangle and four legs in the form of rods whose lengths dominate their cross-sectional areas), or can have a uniform shape. The former yields an *object-based* decomposition while the latter yields an *image-based* or *cell-based* decomposition. Another way of characterizing these two decompositions is that the former decomposes the objects while the latter decomposes the environment in which the objects lie. Regardless of the characterization, the objects (as well as their constituent cells) can be represented either by their interiors or by their boundaries.

Each of the decompositions has its advantages and disadvantages. They depend, in part, on the nature of the queries that are posed to the database. The most relevant queries are *where* and *what*. They are stated more formally as follows:

1. Given an object, determine its constituent cells (i.e., their locations in space).
2. Given a cell (i.e., a location in space), determine the identity of the object (or objects) of which it is a member as well as the remaining constituent cells of the object (or objects).

The generation of responses to these queries is facilitated by building an index (i.e., the result of a sort) either on the objects or on their locations in space.

There are two fundamental methods of representing objects: by their interiors or by their boundaries. Many of the representations can be made more compact by aggregating similar elements. These elements are usually identically-valued contiguous cells (possibly adjacent to identically-oriented boundary elements), or even objects which, ideally, are in proximity. Our main focus in Sections 2–5 is on interior-based representations, while Section 6 gives a brief overview of some boundary-based representations. Concluding remarks are made in Section 7.

Section 2 examines representations based on collections of unit-size cells. An alternative class of interior-based representations of the objects and their environment removes the stipulation that the cells that make up the object collection be of unit size and permits their sizes to vary. The varying-sized cells are termed *blocks* and are the subject of Section 3. The representations described in Sections 2 and 3 assume that each unit-size cell or block is contained entirely in one or more objects.

A cell or block cannot be partially contained in two objects. This means that either each cell in a block belongs to the same object or

objects, or all of the cells in the block do not belong to any of the objects. Section 4 permits a cell or a block to be a part of more than one object, and does not require the cell or block to be contained in its entirety in these objects. In other words, a cell or a block may overlap several objects without being completely contained in them. This also has the effect of permitting the representation of collections of objects whose boundaries do not coincide with the boundaries of the underlying blocks, and also permitting them to intersect (i.e., overlap). Section 5 examines the use of hierarchies of space and objects which enable efficient responses to both queries 1 and 2.

2. UNIT-SIZE CELLS

The most common representation of the objects and their environment is as a collection of cells of uniform size and shape (termed *pixels* and *voxels* in two and three dimensions, respectively) all of whose boundaries (with dimensionality one less than that of the cells) are of unit size. Since the cells are uniform, there exists a way of referring to their locations in space relative to a fixed reference point (e.g., the origin of the coordinate system). An example of a specification of a location of a cell in space is a set of coordinate values that enable us to find it in the d-dimensional space of the environment in which it lies. It should be clear that the concept of the *location* of a cell in space is quite different from that of the *address* of a cell, which is the physical location (e.g., in memory, on disk, etc.), if any, where some of the information associated with the cell is stored.

In most applications (including the ones that we consider here), the boundaries (i.e., edges and faces in two and three dimensions, respectively) of the cells are parallel to the coordinate axes. In our discussion, we assume that the cells comprising a particular object are contiguous (i.e., adjacent), and that a different unique value is associated with each distinct object, thereby enabling us to distinguish between the objects. For example, Figure 1.1 contains three two-dimensional objects `A`, `B`, and `C` and their corresponding cells.

Interior-based methods represent an object o by using the locations in space of the cells that comprise o. Operations on the cells (i.e., locations in space) comprising o are facilitated by aggregating the cells of the objects into subcollections of contiguous identically-valued cells. The aggregation may be implicit or explicit, depending on how the contiguity of the cells that make up the aggregated subcollection is expressed.

An aggregation is *explicit* if the identities of the contiguous cells that form the object are hardwired into the representation. An example of

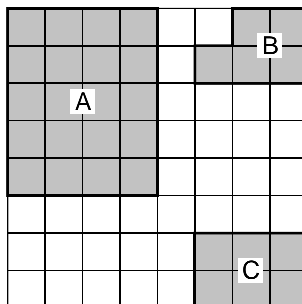

Figure 1.1: Example collection of three objects and the cells that they occupy.

an explicit aggregation is one that associates a set with each object o that contains the location in space of each cell that comprises o. In this case, no identifying information (e.g., the object identifier corresponding to o) is stored in the cells. Thus there is no need to allocate storage for the cells (i.e., no addresses are associated with them). One possible implementation of this set is a list. For example, consider Figure 1.1 and assume that the origin `(0,0)` is at the upper-left corner. Assume further that this is also the location of the pixel that abuts this corner. Therefore, the explicit representation of object `B` is the set of locations {`(5,1) (6,0) (6,1) (7,0) (7,1)`}. It should be clear that using the explicit representation, given an object o, it is easy to determine the cells (i.e., locations in space) that comprise it (query 1). Of course, even when using an explicit representation, we must still be able to access object o from a possibly large collection of objects, which may require an additional data structure such as an index on the objects (e.g., a table of object-value pairs where *value* indicates the entry in the explicit representation corresponding to *object*).

The fact that no identifying information as to the nature of the object is stored in the cell means that the explicit representation is not suited for answering the inverse query of determining the object associated with a particular cell at location l in space (i.e., query 2). Using the explicit representation, query 2 can be answered only by checking for the presence of l in the various sets associated with the different objects. This is time-consuming, as it may require that we examine all cells in each set.

Query 2 can be answered more directly if we allocate an address a in storage for each cell c where an identifier is stored that indicates the identity of the object (or objects) of which c is a member. Such a representation is said to be *implicit* because in order to determine

the rest of the cells that comprise the object associated with c (and thus complete the response to query 2), we must examine the identifiers stored in the addresses associated with the contiguous cells and then aggregate the cells whose associated identifiers are the same. However, in order to be able to use the implicit representation, we must have a way of finding the address a corresponding to c, taking into account that there is possibly a very large number of cells, and then retrieving a.

Finding the right address requires an additional data structure, termed an *access structure*, like an index on the locations in space. An example of such an index is a table of cell-address pairs where *address* indicates the physical location in which the information about the object associated with the location in space corresponding to *cell* is stored. The table is indexed by the location in space corresponding to *cell*.

Such an access structure enables us to obtain the contiguous cells (as we know their locations in space) that comprise an object without having to examine all of the cells. This permits us to complete the response to query 2 with an implicit representation. The existence of an access structure also enables us to answer query 1 with the implicit representation, although this is quite inefficient. In particular, given an object o, we must exhaustively examine every cell (i.e., location l in space) and check if the address where the information about the object associated with l is stored contains o as its value. This is time-consuming, as it may require that we examine all the cells.

The multidimensional array (having a dimension equal to the dimensionality of the space in which the objects and the environment are embedded) is an access structure which, given a cell c at a location l in space, enables us to calculate the address a containing the identifier of the object associated with c. The implicit representation can also be implemented with access structures other than the array. This is an important consideration when many of the cells are not in any of the objects (i.e., they are empty). The problem is that using the array access structure is wasteful of storage, as the array requires an element for each cell regardless of whether the cell is associated with any of the objects. In this case, we choose to keep track only of the non-empty cells.

We have two ways to proceed. The first is to use one of several multidimensional access structures such as a point quadtree [14], k-d tree [5], MX quadtree [59], etc. as described in [59]. The second is to make use of an ordering of space to obtain a mapping from the non-empty contiguous cells to the integers. The result of the mapping serves as the index in one of the familiar tree-like access structures (e.g., binary search tree, range tree, B$^+$-tree, etc.) to store the address which indicates the

physical location where the information about the object associated with the location in space corresponding to the non-empty cell is stored.

3. BLOCKS

An alternative class of representations of the objects and their environment removes the stipulation that the cells making up the object collection be of unit size, and permits their sizes to vary. The resulting cells are termed *blocks* and are usually rectangular with sides parallel to the coordinate axes (this is assumed in our discussion unless explicitly stated otherwise). The volume (e.g., area in two dimensions) of the blocks need not be an integer multiple of that of the unit-size cells, although this is often the case. Observe that when the volumes of the blocks are integer multiples of that of the unit-size cells, we have two levels of aggregation in the sense that an object consists of an aggregation of blocks which are themselves aggregations of cells. We assume that all the cells in a block belong to the same object or objects. In other words, the situation that some of the cells in the block belong to object o_1 while the others belong to object o_2 (and not to o_1) is not allowed.

The collection of blocks is usually a result of a space decomposition process guided by a set of rules. There are many possible decompositions. When the decomposition is recursive, we have the situation that the decomposition occurs in stages and often, although not always, the results of the stages form a containment hierarchy. This means that a block b obtained in stage i is decomposed into a set of blocks b_j that span the same space. Blocks b_j are in turn decomposed in stage $i+1$ using the same decomposition rule. Some decomposition rules restrict the possible sizes and shapes of the blocks as well as their placement in space. Other decomposition rules do not require that the blocks be axis-parallel, while still others do not even require that the blocks be rectangular. In addition, the blocks may be disjoint or be allowed to overlap. Clearly, the choice is large. In the following, we briefly explore some of these decomposition processes.

The simplest decomposition rule is one that permits aggregation of identically-valued cells in only one dimension. It assigns a priority ordering to the various dimensions; fixes the coordinate values of all but one of the dimensions, say i; and then varies the value of the ith coordinate and aggregates all adjacent cells belonging to the same object into a one-dimensional block. This technique is commonly used in image processing applications where the image is decomposed into rows which are scanned from top to bottom, and each row is scanned from left to right while aggregating all adjacent pixels with the same value into a block.

The aggregation into one-dimensional blocks is the basis of *runlength encoding* [49]. Similar techniques are applicable to higher-dimensional data.

The drawback of the decomposition into one-dimensional blocks described above is that all but one side of each block must be of unit width. The most general decomposition removes this restriction along all of the dimensions, thereby permitting aggregation along all dimensions. In other words, the decomposition is arbitrary. The blocks need not be uniform or similar. The only requirement is that the blocks span the space of the environment. We assume that the blocks are disjoint, although this need not be the case. We also assume that the blocks are rectangular with sides parallel to the coordinate axes, although again this is not absolutely necessary, as there exist decompositions using other shapes as well (e.g., triangles).

The use of a decomposition into blocks of arbitrary size makes it somewhat cumbersome to determine all the blocks that comprise the object associated with location l. This process can be facilitated by imposing some regularity on the decomposition process and therefore on the size and placement of the resulting blocks. One of the earliest methods of aggregating unit-size cells with the same value (e.g., belonging to the same object) is the *medial axis transformation (MAT)* [6, 48]. The MAT of object o is formed as follows. For each unit-size cell c in o, find the largest square s_c of width w_c (i.e., radius $w_c/2$) centered at c that is contained in o. s_c is called a *maximal block* if it is not contained in the largest square $s_{c'}$ of any other cell c' in o. o is completely specified by the maximal blocks, their widths, and the unit-size cells at their centers — that is, the set of triples (c, s_c, w_c) — since any point in o lies completely inside one maximal block. The MAT of o is the set of these triples. We usually do not construct the MAT of the regions containing the cells that are not in any of the objects (i.e., the background of the image).

The MAT is compact as long as the objects have simple shapes. However, the MAT is somewhat wasteful of space, as some maximal blocks are contained in the unions of others. Furthermore, when the MAT is defined in the above manner, the widths of the blocks are restricted to being odd integers. This means that half the possible block widths are excluded. We can overcome this by redefining the maximal blocks to be anchored at the unit-size cell at one of their corners instead of at their centers.

Perhaps the most widely known decompositions into blocks are those referred to by the general terms *quadtree* and *octree* [58, 59]. They are usually used to describe a class of representations for two and three (or higher) dimensional data, respectively, that are the result of a recursive

decomposition of the environment (i.e., space) containing the objects into blocks (not necessarily rectangular) until the data in each block satisfies some condition (e.g., with respect to its size, the nature of the objects that comprise it, the number of objects in it, etc.). The positions and/or sizes of the blocks may be restricted or arbitrary. We shall see that quadtrees and octrees may be used with both interior-based and boundary-based representations.

There are many variants of quadtrees and octrees, and they are used in numerous application areas including high-energy physics, VLSI, finite-element analysis, and many others. Below, we focus on *region quadtrees* [30, 27] and *region octrees* [26, 28, 33]. They are specific examples of interior-based representations for two- and and three-dimensional region data (variants for data of higher dimension also exist), respectively, that permit further aggregation of identically-valued cells.

Region quadtrees and region octrees are instances of a restricted-decomposition rule where the environment containing the objects is recursively decomposed into four or eight, respectively, rectangular congruent blocks until each block is either completely occupied by an object or is empty (such a decomposition process is termed *regular*). For example, Figure 1.2a is the block decomposition for the region quadtree corresponding to Figure 1.1. We have labeled the blocks corresponding to object `O` as `Oi` and the blocks that are not in any of the objects as `Wi`, using the suffix `i` to distinguish between them in both cases. Notice that in this case, all the blocks are square, have sides whose size is a power of 2, and are located at specific positions. In particular, assuming an origin at the upper-left corner of the image corresponding to the environment containing the objects, the coordinate values of the upper-left corner of each block (e.g., (a, b) in two dimensions) of size $2^i \times 2^i$ satisfy the property that $a \bmod 2^i = 0$ and $b \bmod 2^i = 0$.

Hunter [26] has shown that the number of blocks in a region quadtree representation of a simple polygonal object (i.e., with non-intersecting edges and without holes) with perimeter p, measured in pixel-widths, that is embedded in a $2^q \times 2^q$ space is $O(p + q)$. In all but the most pathological cases (e.g., a small square of unit width centered in a large space), the q factor is negligible and thus the number of blocks is $O(p)$. An analogous result holds for three-dimensional data [32] (i.e., represented by a region octree), where perimeter is replaced by surface area, as well as for objects of higher dimensions d, for which it is proportional to the size of the $(d - 1)$-dimensional interfaces between the objects.

Algorithms that execute on a region quadtree representation of an object, instead of simply imposing an array access structure on the original collection of cells, usually have an execution time that is proportional

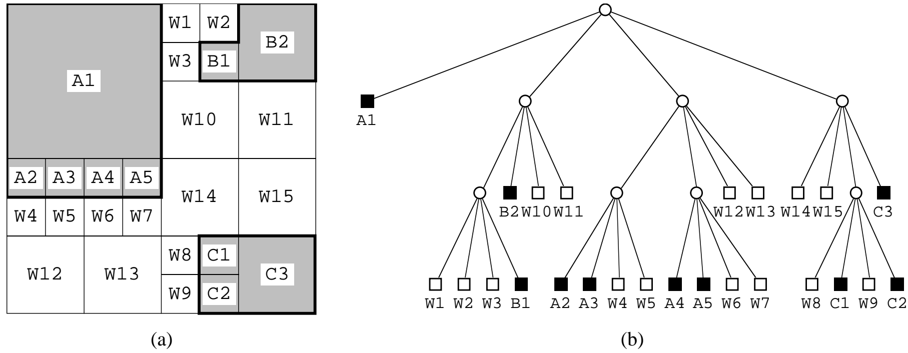

Figure 1.2: (a) Block decomposition and (b) its tree representation for the collection of objects and cells in Figure 1.1.

to the number of blocks in the region quadtree rather than the number of unit-size cells. Applying this space complexity result means that the use of the region quadtree representation, with an appropriate access structure, to solve a problem in d-dimensional space leads to a solution whose execution time is proportional to the $(d-1)$-dimensional space of the surface of the original d-dimensional image. On the other hand, use of the array access structure on the original collection of cells results in a solution whose execution time is proportional to the number of cells that comprise the image. Therefore, region quadtrees and region octrees act like dimension-reducing devices.

The region quadtree can be viewed as a special case of the medial axis transformation (MAT). This can be seen by observing that the widths of the maximal blocks of the MAT are restricted to be powers of 2, and the positions of their centers are constrained to be centers of blocks obtained by a recursive decomposition of the underlying space into four congruent blocks. The notion of a MAT can be further adapted to the region quadtree by constraining only the positions of the centers of the blocks and not their sizes. The result is termed a *quadtree medial axis transform (QMAT)* [55, 57]. Thus the blocks of the QMAT can span a larger area, as long as they are square-shaped.

The blocks of the QMAT are determined in the same way as the maximal blocks in the MAT. For each block b in the region quadtree for object o, find the largest square s_b of width w_b centered at b that is contained in o. S_c is called a *maximal block* if it is not contained in the largest square $s_{b'}$ of any other block b' in o. Each object is completely specified by the set of maximal blocks that comprise it. The QMAT is the quadtree whose blocks are the ones that are occupied by objects and that have

a corresponding maximal block and an associated width. For example, Figure 1.3b is the block decomposition of the QMAT corresponding to the block decomposition of the region quadtree in Figure 1.3a. Notice that in order to permit larger maximal blocks, we assume that the area outside the array of unit-size cells corresponding to the space in which object o is embedded is also part of o (e.g., the area to the north, west, and northeast of block 1 in Figure 1.3a). It can be shown that the QMAT is unique [55, 58].

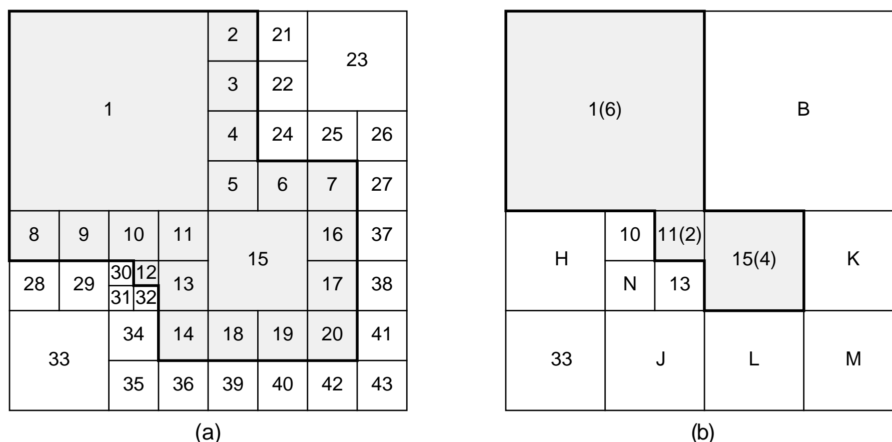

Figure 1.3: Block decomposition induced by (a) the region quadtree for an object, and (b) its corresponding quadtree medial axis transform (QMAT).

The traditional, and most natural, access structure for a region quadtree corresponding to a d-dimensional image is a tree with a fanout of 2^d (e.g., Figure 1.2b, corresponding to the collection of two-dimensional objects in Figure 1.1, whose quadtree block decomposition is given in Figure 1.2a). Each nonleaf node f corresponds to a block whose volume is the union of the blocks corresponding to the 2^d sons of f. In this case, the tree is a containment hierarchy and closely parallels the decomposition in the sense that they are both recursive processes and the blocks corresponding to nodes at different depths of the tree are similar in shape.

Answering query 2 using the tree access structure is different from using an array, where it is usually achieved by a table lookup having an $O(1)$ cost (unless the array is implemented as a tree, which is a possibility [9]). In contrast, query 2 is usually answered in a tree by locating the block that contains the location in space corresponding to

the desired cell. This is achieved by a process that starts at the root of the tree and traverses the links to the sons whose corresponding blocks contain the desired location. This process has an $O(m)$ cost, where the environment has a maximum of m levels of subdivision (e.g., an environment all of whose sides are of length 2^m).

There are several alternative access structures to the tree with fanout 2^d. They are all based on finding a mapping from the domain of the blocks to a subset of the integers (i.e., to one dimension) and then applying one of the familiar tree-like access structures (e.g., a binary search tree, range tree, B^+-tree, etc.). There are many possible mappings (e.g., [58]).

As the dimensionality of the space (i.e., d) increases, each level of decomposition in the region quadtree results in many new blocks, as the fanout value 2^d is high. In particular, it is too large for practical implementation of the tree access structure. In this case, an access structure termed a *bintree* [31, 66, 72] with a fanout value of 2 is used. The bintree is defined in a manner analogous to the region quadtree except that at each subdivision stage, the space is decomposed into two equal-sized parts. In two dimensions, at odd stages we partition along the y axis and at even stages we partition along the x axis.

The region quadtree, as well as the bintree, is a regular decomposition. This means that the blocks are congruent — that is, at each level of decomposition, all of the resulting blocks are of the same shape and size. We can also use decompositions where the sizes of the blocks are not restricted in the sense that the only restriction is that they be rectangular and be a result of a recursive decomposition process. In this case, the representations that we described must be modified so that the sizes of the individual blocks can be obtained. An example of such a structure is an adaptation of the point quadtree [14] to regions. Although the point quadtree was designed to represent points in a higher-dimensional space, the blocks resulting from its use to decompose space do correspond to regions. The difference from the region quadtree is that in the point quadtree, the positions of the partitions are arbitrary, whereas they are a result of a partitioning process into 2^d congruent blocks (e.g., quartering in two dimensions) in the case of the region quadtree.

As the dimensionality d of the space increases, each level of decomposition in the point quadtree results in many new blocks since the fanout value 2^d is high. In particular, it is too large for a practical implementation of the tree access structure. Therefore we use a k-d tree [5], which is an access structure having a fanout of 2 that is an adaptation of the point quadtree to regions. As in the point quadtree, although the k-d tree was designed to represent points in a higher-dimensional space,

the blocks resulting from its use to decompose space do correspond to regions.

The k-d tree can be further generalized so that the partitions take place on the various axes in an arbitrary order. In fact, the partitions need not be made on every coordinate axis. In this case, at each nonleaf node of the k-d tree, we must also record the identity of the axis that is being split. We use the term *generalized k-d tree* to describe this structure. The generalized k-d tree is really an adaptation to regions of the *adaptive k-d tree* [19] and the *LSD tree* [23], which were originally developed for points. It can also be regarded as a special case of the *BSP (Binary Space Partitioning) tree* [20]. In particular, in the generalized k-d tree, the partitioning hyperplanes are restricted to be parallel to the axes, whereas in the BSP tree they have arbitrary orientations. The BSP tree is used in computer graphics to facilitate viewing.

One of the shortcomings of the generalized k-d tree is the fact that we can only decompose the space into two parts along a particular dimension at each step. If we wish to partition a space into p parts along dimension i, then using the generalized k-d tree we must perform $p-1$ successive partitions on dimension i. Once these $p-1$ partitions are complete, we partition along another dimension. The *puzzletree* [10] is a further generalization of the k-d tree that permits the underlying space to be decomposed into two or more parts along a particular dimension at each step so that no two successive partitions use the same dimension. In other words, the puzzletree compresses all successive partitions on the same dimension in the generalized k-d tree.

4. ARBITRARY OBJECTS

In this section we loosen the restrictions on the cells, blocks, and objects in Sections 2 and 3 that prevented a cell or a block from being partially contained in several objects or that required a cell or a block to be contained in its entirety in a single object. In other words, a cell or a block may overlap several objects without being completely contained in these objects. Furthermore, we permit objects to be of arbitrary shapes rather than requiring their boundaries to be hyperplanes.

Loosening these restrictions enables the decoupling of the partition of the underlying space induced by the decomposition process from the partition induced by the objects. Moreover, the individual objects no longer need to be represented as collections of unit-sized cells or rectangular blocks. Instead, collections of other shapes such as triangles, trapezoids, convex polygons, etc. can be used. When there is no explicit relationship between the blocks and the objects that they contain

(e.g., containment), then in addition to keeping track of the identities of all objects (or parts of objects) which can be covered by a block or part of a block, we must also keep track of the geometric descriptions of both the cells or blocks and the objects. This enables us to respond to query 2.

In Section 3 we halted the decomposition process whenever a block was completely contained in an object or a set of objects. However, it is clear that this rule is impossible to satisfy in general, now that the boundaries of the objects need not coincide with the boundaries of the blocks that are induced by the decomposition process. Thus we need to find an alternative way of halting the decomposition process. There are two natural methods of achieving this result, and they are the subject of this section.

1. Restrict the number of blocks that can cover an object (i.e., parts of it); this is termed *coverage-based splitting* [7] (Section 4.1).
2. Restrict the number of objects (or parts of objects) that can be covered by a block or part of a block; this is termed *density-based splitting* [7] (Section 4.2).

4.1 COVERAGE-BASED SPLITTING

A common way of implementing coverage-based splitting is to set the number of blocks that can cover an object to 1. In particular, this block is usually the smallest possible block that contains the object. For example, the MX-CIF quadtree [29] is a quadtree-like regular decomposition rule which decomposes the underlying space into four congruent blocks at each stage (i.e., level) of the decomposition process so that each object is associated with its minimum enclosing quadtree block. Figure 1.4 is an MX-CIF quadtree where we see that more than one object is associated with some of the nodes in the tree (e.g., the root and its `NE` son).

Since there is no limit on the number of objects that are associated with a particular block, an additional decomposition rule may be provided to distinguish between these objects. For example, in the case of the MX-CIF quadtree, a one-dimensional analog of the two-dimensional decomposition rule is used. In particular, all objects that are associated with a given block b are partitioned into two sets: those that overlap the vertical (horizontal) axis passing through the center of b. Objects that overlap the center of b are associated with the horizontal axis. Associated with each axis is a one-dimensional MX-CIF quadtree (i.e., a binary tree) where each object o is associated with the node that corresponds to o's minimum enclosing interval.

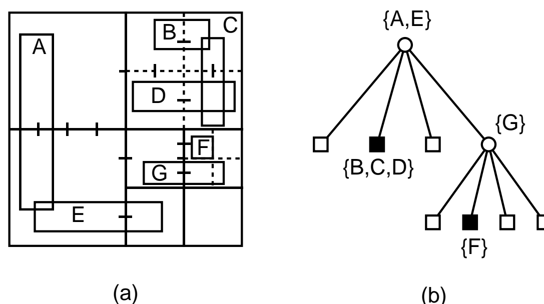

Figure 1.4: (a) Block decomposition induced by the MX-CIF quadtree for a collection of rectangle objects and (b) its tree representation.

The drawback of restricting the number of blocks that can cover an object o to 1 (i.e., o's minimum enclosing quadtree block) is that the block tends to be rather large, on the average. In particular, in the worst case, a small object that straddles the top-level split partition will have the root as its minimum enclosing quadtree block (e.g., objects `A` and `E` are associated with the root in the MX-CIF quadtree in Figure 1.4).

An easy way to overcome this drawback is to decompose o's minimum enclosing quadtree block (or the result of another space decomposition method such as a bintree, etc.) into smaller blocks, each of which minimally encloses some portion of o (or, alternatively, some portion of o's minimum bounding box). Orenstein [37] proposes the *error-bound* and *size-bound* methods for deciding how much decomposition to perform to yield the blocks covering object o.

In the error-bound method, the minimum enclosing block is recursively split until level L is reached, where 0 is the level of the root and L is a user-specified level threshold. In the size-bound method, the minimum enclosing block is recursively split in a breadth-first manner, and the process is terminated once a threshold number S of blocks that collectively cover the space spanned by o has been obtained. For both methods, once enough splitting has been performed, the blocks that collectively cover o are shrunk so that each one minimally encloses the portion of o inside it, and sibling blocks which both cover o are coalesced. Orenstein [37] suggests that the error-bound method generally leads to better query performance than the size-bound method, and this has been confirmed by experiments [7].

Frank and Barrera [16, 17] and Ulrich [74] take a different approach, by retaining the restriction that each object can be covered by just one block. One solution is to expand the size of the space that is spanned

by each quadtree block b of width w by a factor p ($p > 0$) so that the expanded block is of width $(1+p) \cdot w$. In this case, an object would be associated with its minimum enclosing expanded quadtree block. It can be shown that the radius of the minimum bounding volume for any object o that is associated with a block b of width w (on account of overlapping the horizontal or vertical axes that pass through the center of b while not lying within the expanded quadtree block of one of b's subblocks) must be larger than $p \cdot w/4$. The terms *cover fieldtree* [16, 17] and *loose quadtree* [74] are used to describe the resulting structure. We use these terms interchangeably.

Ulrich [74] advocates setting the block expansion factor p to 1. He argues that using block expansion factors much smaller than 1 increases the likelihood that the minimum enclosing expanded quadtree block is large, while letting the block expansion factor be much larger than 1 results in the areas spanned by the expanded quadtree blocks being too large, thereby having much overlap. For example, letting $p = 1$ and considering the collection of rectangle objects in Figure 1.4a, the only difference between the corresponding MX-CIF and loose quadtrees is that rectangle object `E` is associated with the `SW` son of the root of the loose quadtree instead of with the root of the MX-CIF quadtree, as shown in Figure 1.4b.

The loose quadtree and the cover fieldtree are similar to the quadtree medial axis transform (QMAT) [55] (see Section 3), where the factor p is bounded by 2 (i.e., it is less than 2) [54, 55]. In this case, for a black block b of width w in a region quadtree, $p \cdot w/2$ represents the distance from the border of b to the nearest point which is on a boundary between a black and a white block in the structure. The QMAT is used as an alternative to the region quadtree representation of an image in that it is an attempt to reduce the sensitivity of the storage requirements of the region quadtree to the position of the origin with respect to which it is built. The QMAT can also be used as a representation for a collection of objects, as is done with a loose quadtree and a cover fieldtree.

Frank and Barrera [16, 17] also propose shifting the positions of the centroids of blocks at successive levels of subdivision by half the width of the block that is being subdivided as yet another alternative approach to overcoming the problem of the minimum enclosing quadtree block of o being much larger than o (e.g., o's minimum bounding volume). Figure 1.5 shows an example of such a subdivision. The result is termed a *partition fieldtree* by Frank and Barrera [16, 17] and is also similar to the *overlapped pyramid* of Burt [8]. Note that the boundaries of blocks at different levels of decomposition never coincide. Moreover, blocks at

a given level of decomposition l do not form a refinement of the blocks at the previous level of decomposition $l-1$.

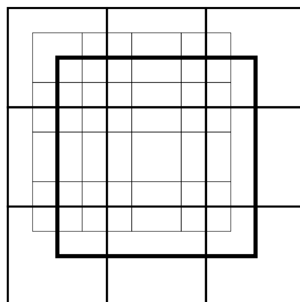

Figure 1.5: Example of the subdivision induced by a partition fieldtree.

An important property of the partition fieldtree is that the relative size of the minimum enclosing quadtree block for object o is bounded with respect to the size of the minimum bounding volume v of o. In particular, it can be shown that objects of the same size can be stored in nodes corresponding to larger blocks of at most three possible sizes depending on their positions — that is, in the worst case, o will be stored in a block that is 2^3 (i.e., 8) times o's size (i.e., maximum width along the axes) [17]. On the other hand, in the case of the cover fieldtree, the bound depends on the value of p. In particular, it can be shown that it is 2 for $p = 1$ [74] and 8 for $p = 1/4$. For values of p greater than $1/4$, the partition fieldtree yields a tighter fit than the cover fieldtree.

At this point, it is appropriate to review the distinction between the cover and partition fieldtrees. In both cases, the goal is to expand the area spanned by the subblocks of a block in order to reduce the size of the minimum enclosing quadtree block b when an object o overlaps the axes that pass through the center of b. In the case of the cover fieldtree, the area spanned by the four subblocks is expanded, while in the case of the partition fieldtree, the number of covering subblocks is enlarged by offsetting their positions while retaining their sizes — that is, a set of 3×3 blocks of size $u/2^{1+1} \times u/2^{i+1}$ at level $i+1$ spans a block of size $u/2^i$ at level i (assuming a universe of width u and a root at level i). The result in both cases is that the subblocks span an area that overlaps the partition lines, which is something that would not be possible in a conventional quadtree (e.g., the region quadtree, MX-CIF quadtree, or any quadtree where the blocks are disjoint), since the edges of the blocks at successive levels of decomposition must coincide (i.e., they are collinear).

4.2 DENSITY-BASED SPLITTING

Density-based splitting is formally specified as stipulating that a block is decomposed whenever it is covered by more than T ($T \geq 1$) objects (or parts of them). Notice that this decomposition rule is not one that decomposes a block if it contains more than T objects. Such a rule might be difficult to satisfy in the sense that there would be very little decomposition after an initial level, as blocks usually contain portions of many objects rather than many entire objects. Such rules are characterized as being *bucket-like*. The case $T = 1$ corresponds to halting the decomposition whenever each cell in the block b is an element of the same object o or is in no object. In other words, any cell c in b is either in o or in no object. Observe that even for the $T = 1$ case, the result is different from the block decomposition rule described in Section 3, which required that either all the cells in b are elements of the same object or all the cells in b are in none of the objects.

There are many variants of bucket-like block decomposition rules, depending in part on the nature of the objects being represented. Without loss of generality, in this section we assume two-dimensional data. In particular, we focus on how to deal with collections of arbitrary polygonal objects. The specific variant that is used depends on whether the polygons themselves are the objects being decomposed, which is the case in this section, or if the decomposition rule is based on the primitive objects that comprise the polygon (e.g., its vertices and/or edges), which is the case in the variants described in Section 6.

It is straightforward to formulate bucket-like variants of the different block decomposition rules described in Section 3. Without loss of generality, in the rest of this section we assume a quadtree-like regular decomposition rule which decomposes the underlying space into four congruent blocks at each stage (i.e., level) of decomposition. As in Section 3, the results of successive stages form a containment hierarchy, and once the decomposition process has terminated, the result is a set of disjoint blocks. Of course, other decomposition rules such as those that do not require the blocks to be congruent at a particular stage and similar at successive stages (e.g., the point quadtree [14]), or do not subdivide into just two parts as do the k-d tree [5] and bintree [31, 66, 72] (e.g., the puzzletree [10]), or do not require that the blocks be axis-parallel (e.g., a BSP tree [20]) could also be used.

The bucket-like decomposition rule described above works well when the polygons are disjoint, and we use the term *bucket polygon quadtree* to describe the result. However, when the polygons are permitted to overlap or are adjacent, problems can arise in the sense that there exist

polygon configurations for which the decomposition will never halt. In this case, our goal is to prevent as much unnecessary decomposition as possible, but at the same time to minimize the amount of useful decomposition that is prevented from taking place.

As an example of a block configuration for which the bucket polygon quadtree would require an infinite amount of decomposition, consider the situation when $Q > T$ polygons in block b are arranged so that polygon p_i is *completely contained* in polygon p_{i+1} for $1 \leq i < Q$. Now, let us examine the instant of time when b contains T of the polygons. In this case, upon insertion of the $T+1^{st}$ polygon, we are better off not even starting to decompose b, which now has $T+1$ polygons, as we can never reach the situation that all the blocks resulting from the decomposition will be part of T or fewer polygons. For example, assuming $T = 2$, the block containing the three polygons in Figure 1.6a is not split upon insertion of the third polygon. In this case, the order of inserting the polygons is immaterial.

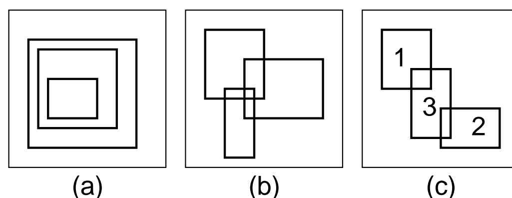

Figure 1.6: Examples of collections of three overlapping polygons and their effect on their containing block B, which is not split when using (a) the completely contained decomposition rule or (b) the mutual intersection rule. (c) B is split when using the touching rule if the polygons are inserted in the order 1, 2, 3, while B is not split if the polygons are inserted in the order 1, 3, 2.

A decomposition rule that prevents more decomposition than one that halts the decomposition only upon complete containment is one that requires the mutual intersection of all of the polygons in block b, including the polygon p being inserted, to be nonempty in order to prevent decomposition [68]. This mutual intersection rule is motivated by the desire to be able to deal with the situation in which many edges of the polygons have a common endpoint (e.g., in a polygonal subdivision of the plane where the polygons are adjacent but not overlapping). For example, assuming $T = 2$, the polygons in the block in Figure 1.6b are mutually intersecting and thus the block is not split regardless of the order in which the polygons are inserted into it. In fact, the order in which

the polygons are inserted is always immaterial when using the mutual intersection rule.

A decomposition rule that prevents even more decomposition than the mutual intersection rule is one that splits a block b that contains at least T polygons before inserting polygon p if p does not touch all of the polygons already contained in b. We say that two polygons p_1 and p_2 *touch* if p_1 and p_2 have at least one point in common (i.e., the polygons may overlap or just touch at their boundaries). For example, assuming $T = 2$, inserting polygons `1`, `2`, `3` in this order in Figure 1.6c avoids splitting the block. On the other hand, inserting the polygons in the order `1`, `3`, `2` in Figure 1.6c results in the block being split. Thus we see that the result of using the touching rule may be sensitive to the order in which the polygons are inserted.

It is interesting to observe that using a decomposition rule based on touching often prevents useful decompositions from taking place. For example, in Figure 1.6c it may be better to decompose the block. Nevertheless, although the mutual intersection rule is really a better decomposition rule than the touching rule in the sense of allowing more useful decomposition to take place, the use of the mutual intersection rule is not suggested because its computation for the general case of polygons of arbitrary shape is expensive.

Perhaps the most common type of collection of polygons is one where all the polygons are adjacent. This is called a *polygonal map* and results in the partition of the underlying image into connected regions. It arises in many cartographic applications such as maps. For such data, use of a bucket-like decomposition rule such as the bucket polygon quadtree can still result in an infinite amount of decomposition. However, the case of an infinite amount of decomposition due to more than T polygons being contained in each other is not possible, as the polygons do not overlap. Nevertheless, infinite decomposition is still possible if the polygonal map has a vertex at which more than T polygons are incident. An interesting way to overcome this problem is to use a variant of a bucket polygon quadtree, developed originally for line segment objects [35, 36], known as the *PMR quadtree* (see also Section 6 for a discussion of its use for line segment objects). It decomposes the block just once if it is a part of more than T polygons. We term the result a *PMR polygon quadtree*. Such a rule means that the shape of the resulting tree depends on the order in which the polygons are inserted into it.

Once the space has been partitioned into blocks, we need to consider the representations of the polygons that make up each block. There are several possible methods. The first is to leave them alone and just associate with each block a list of the polygons that overlap it. The elements

of this list are usually pointers to a polygon table which contains the full geometric description of each polygon (e.g., a list of vertices or edges). Of course, we could also apply some spatial sorting technique to the polygons in each block (e.g., by the locations of their centroids, etc.). The second is to decompose them into a collection of convex regions [40]. This is motivated by the fact that operations on convex polygons are more efficient than operations on general or simple polygons (e.g., point location). In particular, there are two choices for the decomposition into convex regions. The first is to represent each polygon in the block by a union of convex regions, and the second is as a difference of convex regions.

5. HIERARCHICAL REPRESENTATIONS

Assuming the presence of an access structure, the implicit representations are good for finding the objects associated with a particular location or cell (i.e., query 2), while requiring that all cells be examined when determining the locations associated with a particular object (i.e., query 1). In contrast, explicit representations are good for query 1, while requiring that all objects be examined when trying to respond to query 2. In this section, we focus on representations that enable both queries to be answered without possibly having to examine every cell.

This is achieved by imposing containment hierarchies on the representations. The hierarchies differ depending on whether the hierarchy is of space (i.e., the cells in the space in which the objects are found), or of objects. In the former case, we aggregate space into successively larger-sized chunks (i.e., blocks), while in the latter, we aggregate objects into successively larger groups (in terms of the number of objects that they contain). The former is applicable to implicit (usually image-based) representations, while the latter is applicable to explicit (usually object-based) representations.

The basic idea is that in image-based representations we propagate objects up the hierarchy, with the occupied space being implicit to the representation. Thus we retain the property that associated with each cell is an identifier indicating the object of which it is a part. In fact, it is this information that is propagated up the hierarchy so that each element in the hierarchy contains the union of the objects that appear in the elements immediately below it.

The resulting hierarchy is known as a *pyramid* [73] and is frequently characterized as a *multiresolution* representation since the original collection of objects is described at several levels of detail by using cells that have different sizes, though they are similar in shape. Figure 1.7

shows the pyramid corresponding to the collection of objects and cells in Figure 1.1 and the labels in Figure 1.2a. In this case, we are aggregating 2×2 cells and blocks. Notice the similarity between the pyramid and the region quadtree implementation that uses an access structure which is a tree with a fanout of 4 (Figure 1.2b).

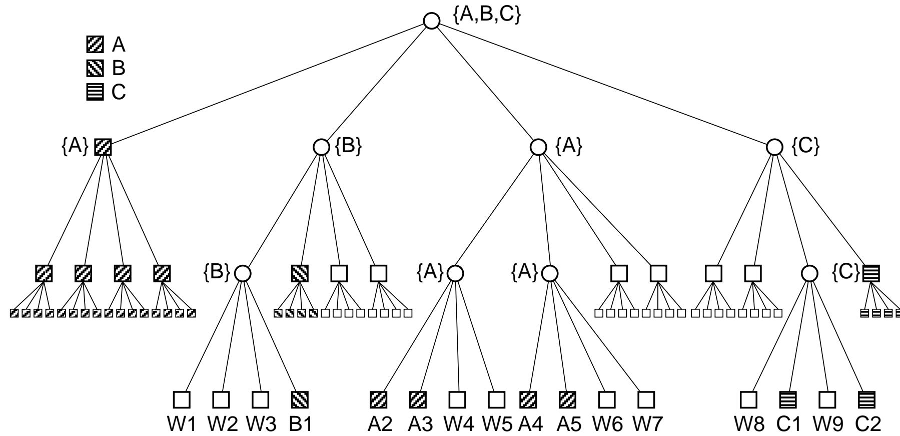

Figure 1.7: Pyramid for the collection of objects and cells in Figure 1.1.

Nevertheless, it is important to distinguish the pyramid from the region quadtree, which, as we recall, is an example of an aggregation into square blocks where the basis of the aggregation is that the cells have identical values (i.e., are associated with the same object, or objects if object overlap is permitted). The region quadtree is an instance of what is termed a *variable-resolution* representation, which, of course, is not limited to blocks that are square. In particular, it can be used with a limited number of non-rectangular shapes (most notably, triangles in two dimensions [4, 59]).

The pyramid can be viewed as a complete region quadtree (i.e., where no aggregation takes place at the deepest level, or, equivalently, all leaf nodes with no sons are at maximum depth in the tree). The difference is that in the case of the region quadtree, the nonleaf nodes serve only as an access structure. Unlike the pyramid, they do not include any information about the objects present in the nodes and cells below them. This is why the region quadtree, like the array, is not useful for answering query 1. Of course, we could also devise a variant of the region quadtree (termed a *truncated-tree pyramid* [60]) which uses the nonleaf nodes to store information about the objects present in the cells and nodes below

them. Note that both the pyramid and the truncated-tree pyramid are instances of an implicit representation with a tree access structure.

On the other hand, in object-based representations we propagate the space occupied by the objects up the hierarchy, with the identities of the objects being implicit to the representation. Thus we retain the property that associated with each object is a set of locations in space corresponding to the cells that make up the object. Actually, since this information may be rather voluminous, it is often the case that an approximation of the space occupied by the object is propagated up the hierarchy rather than the collection of individual cells that are spanned by the object. The approximation is usually the minimum bounding box for the object that is customarily stored with the explicit representation. Therefore, associated with each element in the hierarchy is a bounding box corresponding to the union of the bounding boxes associated with the elements immediately below it.

The R-tree [22] is an example of an object hierarchy which is used especially in database applications. The number of objects or bounding boxes that are aggregated in each node is permitted to range between $m \leq \lceil M/2 \rceil$ and M. The root node in an R-tree has at least two entries unless it is a leaf node, in which case it has just one entry corresponding to the bounding box of an object. The R-tree is usually built as the objects are encountered rather than waiting until all objects have been input.

Figure 1.8a is an example R-tree for a collection of rectangle objects with $m = 2$ and $M = 3$. Figure 1.8b shows the spatial extents of the bounding boxes of the nodes in Figure 1.8a, with heavy lines denoting the bounding boxes corresponding to the leaf nodes, and broken lines denoting the bounding boxes corresponding to the subtrees rooted at the nonleaf nodes. Note that the R-tree is not unique. Its structure depends heavily on the order in which the individual objects were inserted into (and possibly deleted from) the tree.

The drawback of the R-tree (and any representation based on an object hierarchy) is that we may have to examine all of the bounding boxes at all levels when attempting to determine the identity of the object o that contains location a (i.e., query 2). This is caused by the fact that the bounding boxes corresponding to different nodes may overlap (i.e., they are not disjoint). The fact that each object is associated with only one node while being contained in possibly many bounding boxes (e.g., in Figure 1.8, rectangle `1` is contained in its entirety in `R1`, `R2`, `R3`, and `R5`) means that query 2 may often require several nonleaf nodes to be visited before determining the object that contains a. This can be overcome by decomposing the bounding boxes so that disjointness holds (e.g., the

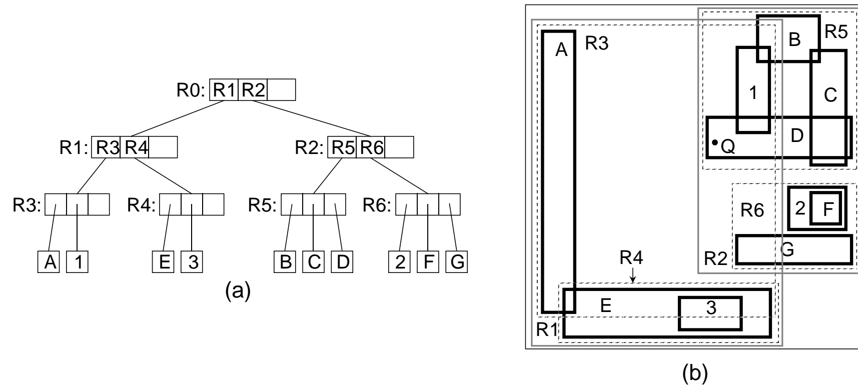

Figure 1.8: (a) R-tree for a collection of rectangle objects with m=2 and M=3, and (b) the spatial extents of the bounding boxes. Notice that the leaf nodes in the index also store bounding boxes, although this is only shown for the nonleaf nodes.

k-d-B-tree [42] and the R$^+$-tree‘[70]), which is a type of space hierarchy in the spirit of the pyramid, albeit not a regular decomposition. The drawback of this solution is that an object may be associated with more than one bounding box, which may result in the object being reported as satisfying a particular query more than once. For example, suppose that we want to retrieve all the objects that overlap a particular region (i.e., a window query) rather than a point as is done in query 2.

Note that the presence of the hierarchy does not mean that the alternative query (i.e., query 1 in the case of a space hierarchy and query 2 in the case of an object hierarchy) can be answered immediately. Instead, obtaining the answer usually requires that the hierarchy be descended. The effect is that the order of the execution time needed to obtain the answer is reduced from linear to logarithmic. Of course, this is not always the case. For example, the fact that we are using bounding boxes for the space spanned by the objects rather than the exact space occupied by them means that we do not always have a complete answer when reaching the bottom of the hierarchy. In particular, at this point we may have to resort to a more expensive point-in-polygon test [15].

It is worth repeating that the only reason for imposing the hierarchy is to facilitate responding to the alternative query (i.e., query 1 in the case of a space hierarchy on the implicit representation, and query 2 in the case of an object hierarchy on the explicit representation). Thus the base representation of the hierarchy is still usually used to answer the original query, because often, when using the hierarchy, the inherently

logarithmic overhead incurred by the need to descend the hierarchy may be too expensive (e.g., when using the implicit representation with the array access structure to respond to query 2). Of course, other considerations such as space requirements may cause us to modify the base representation of the hierarchy, with the result that it will take longer to respond to the original query (e.g., the use of a tree-like access structure with an implicit representation). Nevertheless, as a general rule, in the case of the space hierarchy we use the implicit representation (which is the base of this hierarchy) to answer query 2, while in the case of the object hierarchy we use the explicit representation (which is the base of this hierarchy) to answer query 1.

6. BOUNDARY-BASED REPRESENTATIONS

Boundary-based representations are more amenable to the calculation of global shape properties (e.g., perimeter, extent, etc.). Not surprisingly, the nature of the boundaries plays an important role in the representation that is chosen. Often, the boundary elements of the objects are constrained to be hyperplanes (e.g., polygons in two dimensions and polyhedra in three dimensions) which may in addition be constrained to be axis-parallel. Much of the following presentation is in the context of such constraints unless explicitly stated otherwise, although we will also discuss the more general case.

Assuming that these two constraints hold, a simple representation is one that records the locations of the different boundary elements associated with each cell of each object and their natures (i.e., their orientations and the locations of the cells to which they are adjacent). For example, in two dimensions, the boundary elements are the sides of the cells (i.e., unit vectors), while in three dimensions, the boundary elements are the faces of the cells (i.e., squares of unit area, with directions normal to the object). Boundary-based representations aggregate identically-valued cells whose boundary elements have the same direction, rather than just identically-valued cells as is done by interior-based representations. In two dimensions, the aggregation yields boundary elements which are vectors whose lengths can be greater than 1.

Whichever boundary-based representation is used, and regardless of whether any aggregation takes place, the representation must also enable the determination of the connectivity between individual boundary elements. The connectivity may be implicit or explicit (e.g., by specifying which boundary elements are connected). Thus we notice that this distinction (i.e., implicit vs. explicit) between boundary-based represen-

tations is different from the one used with interior-based representations which was based on the nature of the specification of the aggregation.

As an example of a boundary-based representation, let us consider two-dimensional objects for which the boundary elements are vectors. The location of a vector is given by its start and end vertices. An object o has one more boundary (i.e., a collection of connected boundary elements) than it has holes. Connectivity may be determined implicitly by ordering the boundary elements $e_{i,j}$ of boundary b_i of o so that the end vertex of vector v_j corresponding to $e_{i,j}$ is the start vertex of vector v_{j+1} corresponding to $e_{i,j+1}$. The result of applying such an ordering when identically-valued cells whose boundary elements have the same direction are aggregated yields a representation known as the *polygon representation*. This term is also used to describe the representation of arbitrary objects whose boundaries need not be axis-parallel.

In two dimensions, the most general example of a nonpolygonal object boundary is the curvilinear line segment. Straight line segments with arbitrary slopes are less general. A curvilinear line segment is often approximated by a set of line segments termed a *polyline*. In order to comply with our assumption that the objects are comprised of unit-sized cells (i.e., pixels), we digitize the line and then mark the pixels through which it passes. An alternative is to classify the pixels on the basis of the slope of the part of the line that passes through them. One such representation is the chain code [18], in which case the slopes are restricted to four or eight principal directions. The chain code is of particular interest when the slopes are restricted to the four principal directions, as this is what is obtained when the boundaries of the objects are parallel to the coordinate axes.

In dimensions higher than 2, the relationship between the boundary elements associated with a particular object is more complex, as is its expression. Whereas in two dimensions we have only one type of boundary element (i.e., an edge or a vector consisting of two vertices), in $d > 2$ dimensions, given our axis-parallel constraint, we have $d - 1$ different boundary elements (e.g., faces and edges in three dimensions). As we saw, in two dimensions, the sequence of vectors given by a polygon representation is equivalent to an implicit specification of the boundary, by virtue of the fact that each boundary element of an object can be adjacent to only two other boundary elements. Thus consecutive boundary elements in the representation are implicitly connected. This is not possible in $d > 2$ dimensions; assuming axis-parallel objects comprised of unit-size d-dimensional cells, there are 2^{d-1} different adjacencies per boundary element. Therefore, it is difficult to adapt the polygon representation to data of dimensionality greater than 2. Of course, it can

be used to specify a spatial entity comprised of a sequence of edges in d-dimensional space which forms a cycle (i.e., the starting vertex is the same as the final vertex). However, the spatial entity need not be planar.

Nevertheless, in higher dimensions we do have a choice between an explicit and an implicit boundary-based representation. The boundary model (also known as *BRep* [3, 59]) is an example of an explicit boundary-based representation. In particular, observe that in three dimensions, the boundary of an object with planar faces is decomposed into a set of faces, edges, and vertices. The result is an explicit model based on a combined geometric and topological description of the object. The topology is captured by a set of relations that indicate explicitly how the faces, edges, and vertices are connected to each other. For example, the object in Figure 1.9a can be decomposed into the set of faces having the topology shown in Figure 1.9c. The geometry of the faces can be specified by use of appropriate geometric entities (e.g., planes in the case of polyhedra). In d dimensions, the boundary of object o would be decomposed into d sets s_i $(0 \leq i < d)$ where s_i contains all constituent i-dimensional elements of o. This forms the basis of the boundary model and is illustrated by the winged-edge [3] and quad-edge [21] data structures. Although this representation is quite general, it is easy to constrain it to handle axis-parallel objects.

Constructive Solid Geometry (CSG) [41] is another example of an implicit representation that is applicable to objects of arbitrary dimensionality. Although it is usually thought of as an interior-based representation, it also has a boundary-based interpretation. In the interior-based formulation, primitive instances of objects are combined to form more complex objects by use of geometric transformations and regularized Boolean set operations (e.g., union, intersection). The representation is usually in the form of a tree where the leaf nodes correspond to primitive instances and the nonleaf nodes correspond to Boolean operations. For example, the object in Figure 1.9a can be decomposed into three primitive solids with the CSG tree shown in Figure 1.9b, where the operation `A∩−B` denotes set difference. The key is that this representation is procedural. Thus, it indicates how an item can be constructed — that is, what operations are necessary. Often these operations have physical analogs (e.g., drilling). A disadvantage of the CSG representation is that it is not unique. In particular, often there are several ways of constructing an object (e.g., from different primitive elements).

When the primitive instances in CSG are halfspaces and the objects have planar faces, the result is an implicit boundary-based representation. In this case, the boundary of a d-dimensional object o consists of a collection of hyperplanes in d-dimensional space (i.e., the infinite bound-

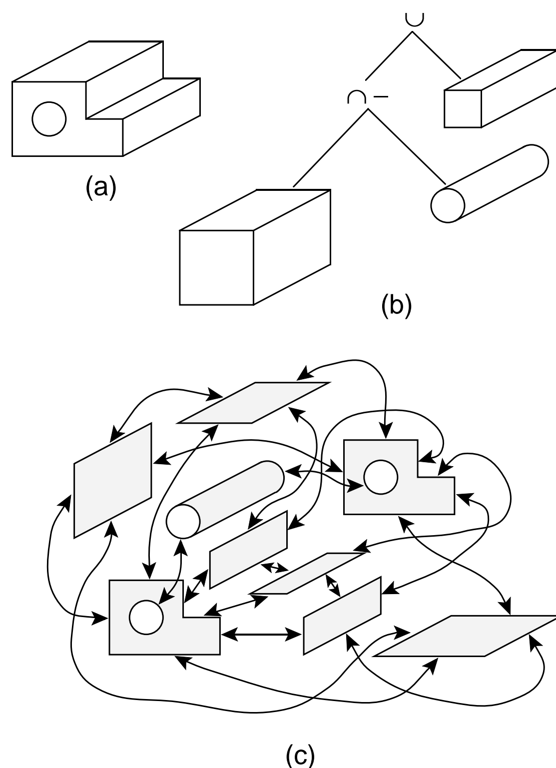

Figure 1.9: (a) A three-dimensional object, (b) its CSG tree, and (c) its boundary model.

aries of regions defined by the inequality $\sum_{i=0}^{d} a_i x_i \geq 0$ where $x_0 = 1$). We have one halfspace for each $(d-1)$-dimensional boundary element of o. Both of these representations (i.e., primitive instances of objects and halfspaces) are implicit, because the object is determined by associating a set of regular Boolean set operations with the collection of primitive instances (which may be halfspaces) the result of which is the object. Although these representations are quite general, it is easy to constrain them to handle axis-parallel objects.

It is usually quite simple to determine the cells that make up an object (i.e., query 1) when using boundary-based representations since the boundaries are usually associated with the individual objects. In contrast, one of the principal drawbacks of boundary-based representations is the difficulty of determining the value associated with an arbitrary point of the space given by a cell (i.e., query 2) without testing each boundary element using operations such as point-in-polygon tests

(e.g., [15]) or finding the nearest boundary element. The problem is that these representations are very local in the sense that generally they just indicate which boundary element is connected to which other boundary element rather than the relationships of the boundary elements to the space that they occupy. Thus if we are at one position on the boundary (i.e., at a particular boundary element), we don't know anything about the rest of the boundary without traversing it element-by-element.

This situation can be remedied in two ways. The first is by aggregating the cells that make up the boundary elements of the objects and their environment into blocks such as those obtained by using a quadtree, an octree, or a k-d tree variant, and then imposing an appropriate access structure on the blocks. The PM quadtree family [35, 67] (see also edge-EXCELL [71]) is an example of the first remedy and is usually applied to polygonal maps (recall Section 4.2). There are several variants of the PM quadtree: vertex-based and edge-based. They are all built by applying the principle of repeatedly breaking up the collection of vertices and edges (forming the polygonal map) until a subset is obtained that is sufficiently simple that it can be organized by some other data structure. PM quadtrees [67] are vertex-based. We illustrate the PM_1 quadtree. Its decomposition rule stipulates that partitioning occurs as long as a block contains more than one line segment, unless the line segments are all incident at the same vertex which is also in the same block (e.g., Figure 1.10a).

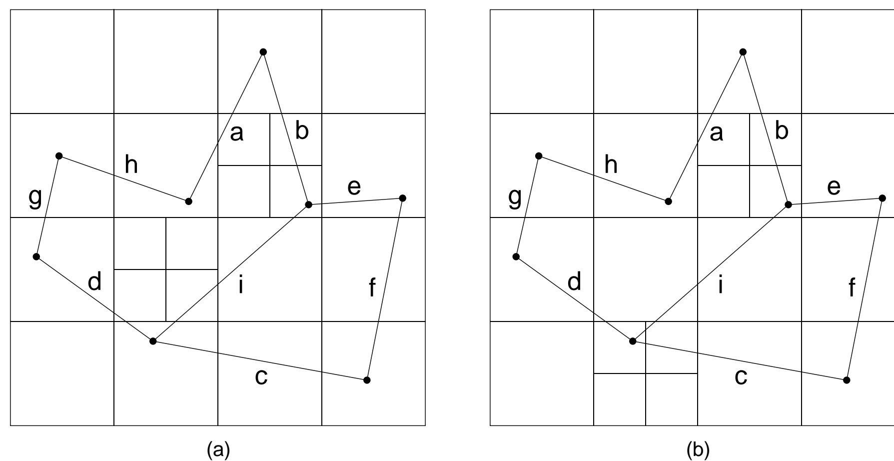

Figure 1.10: (a) PM_1 quadtree and (b) PMR quadtree for a collection of line segments.

A similar representation has been devised for three-dimensional objects (e.g., [1] and the references cited in [59]). The decomposition criteria are such that no node contains more than one face, edge, or vertex unless the faces all meet at the same vertex or are adjacent to the same edge. This representation is quite useful since its space requirements for polyhedral objects are significantly smaller than those of a conventional octree.

The PMR quadtree [35] is an edge-based variant of the PM quadtree. It makes use of a probabilistic splitting rule. A node is permitted to contain a variable number of line segments. A line segment is stored in a PMR quadtree by inserting it into the nodes corresponding to all the blocks that it intersects. During this process, the occupancy of each node that is intersected by the line segment is checked to see if the insertion causes it to exceed a predetermined *splitting threshold*. If the splitting threshold is exceeded, the node's block is split *once*, and only once, into four equal quadrants.

For example, Figure 1.10b is the PMR quadtree for the collection of line segment objects in Figure 1.10a with a splitting threshold value of 2. The line segments are inserted in alphabetic order (i.e., `a`–`i`). It should be clear that the shape of the PMR quadtree depends on the order in which the line segments are inserted. Note the difference from the PM_1 quadtree in Figure 1.10a — that is, the `NE` block of the `SW` quadrant is decomposed in the PM_1 quadtree, while the `SE` block of the `SW` quadrant is not decomposed in the PM_1 quadtree.

The PMR quadtree is preferred over the PM_1 quadtree since it results in far fewer subdivisions. In particular, in the PMR quadtree there is no need to subdivide in order to separate line segments that are very 'close' or whose vertices are very 'close,' which is the case for the PM_1 quadtree. This is important since four blocks are created at each subdivision step. Thus when many subdivision steps that occur in a PM_1 quadtree result in creating many empty blocks, the storage requirements of the PM_1 quadtree are considerably higher than those of the PMR quadtree. Generally, as the splitting threshold is increased, the storage requirements of the PMR quadtree decrease, while the time necessary to perform operations on it increases.

The second remedy proceeds by aggregating the boundary elements of the objects themselves by using variants of bounding boxes to yield successively coarser approximations, and then imposing an appropriate access structure on the bounding boxes. In Section 5 we examined two hierarchical representations (i.e., the R-tree and, to a lesser extent, the R^+-tree) that propagate object approximations in the form of bounding rectangles. In this case, the sides of the bounding rectangles had to be

parallel to the coordinate axes of the space from which the objects are drawn. The *strip tree* [2] is another example of such an approach. It is a hierarchical representation of a single curve that successively approximates segments of it with bounding rectangles that, unlike the R-tree and R$^+$-tree, do not require that the sides be parallel to the coordinate axes. The only requirement is that the curve be continuous; it need not be differentiable.

7. CONCLUDING REMARKS

We have presented a brief overview of several object representations, classified in several ways, principally on whether they are based on interiors or boundaries of objects. We have also tried to stress the importance of aggregation of similarly-valued elements, which was recognized early on by Azriel Rosenfeld. In fact, much of the research described in this overview might not have been carried out had not the author, as well as many others, been motivated by Rosenfeld's early contributions.

Acknowledgments

I am indebted to Gisli R. Hjaltason and William C. Cheng for their help in drawing the figures. I have also greatly benefitted from discussions with Azriel Rosenfeld. The support of the National Science Foundation under Grants IRI-97-12715, EIA-99-00268, and IIS-00-86162 is gratefully acknowledged.

References

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

[2] D. H. Ballard. Strip trees: a hierarchical representation for curves. *Communications of the ACM*, 24(5):310–321, May 1981. Also corrigendum, *Communications of the ACM*, 25(3):213, March 1982.

[3] B. G. Baumgart. A polyhedron representation for computer vision. In *Proceedings of the 1975 National Computer Conference*, volume 44, pages 589–596, Anaheim, CA, May 1975.

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

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[6] H. Blum. A transformation for extracting new descriptors of shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, Cambridge, MA, 1967.

[7] F. Brabec, G. R. Hjaltason, and H. Samet. Indexing spatial objects with the pk-tree. Submitted, 2001.

[8] P. J. Burt, T. Hong, and A. Rosenfeld. Segmentation and estimation of image region properties through cooperative hierarchical computation. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(12):802–809, December 1981.

[9] R. A. DeMillo, S. C. Eisenstat, and R. J. Lipton. Preserving average proximity in arrays. *Communications of the ACM*, 21(3):228–231, March 1978.

[10] A. Dengel. Self-adapting structuring and representation of space. Technical Report RR-91-22, Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Germany, September 1991.

[11] M. B. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM*, 39(2):253–280, April 1992. Also Corrigendum, *Journal of the ACM*, 39(4):985, October 1992; University of Maryland Computer Science TR-2303.

[12] C. R. Dyer. Computing the Euler number of an image from its quadtree. *Computer Graphics and Image Processing*, 13(3):270–276, July 1980.

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

[14] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[15] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, second edition, 1990.

[16] A. Frank. Problems of realizing LIS: storage methods for space related data: the field tree. Technical Report 71, Institut für Geodäsie und Photogrammetrie, ETH, Zurich, Switzerland, June 1983.

[17] A. U. Frank and R. Barrera. The Fieldtree: a data structure for geographic information systems. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, editors, *Design and Implementation of Large Spatial Databases — First Symposium, SSD'89*, pages 29–44, Santa Barbara, CA, July 1989. Also Springer-Verlag Lecture Notes in Computer Science 409.

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

[19] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[20] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics*, 14(3):124–133, July 1980. Also *Proceedings of the SIGGRAPH'80 Conference*, Seattle, WA, July 1980.

[21] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985. Also *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing*, pages 221–234, Boston, April 1983.

[22] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, Boston, June 1984.

[23] A. Henrich, H. W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional point and non-point data. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings of the 15th International Conference on Very Large Databases (VLDB)*, pages 45–53, Amsterdam, The Netherlands, August 1989.

[24] T. H. Hong, K. A. Narayanan, S. Peleg, A. Rosenfeld, and T. Silberberg. Image smoothing and segmentation by multiresolution pixel linking: further experiments and extensions. *IEEE Transactions on Systems, Man, and Cybernetics*, 12(5):611–622, September/October 1982.

[25] T. H. Hong, M. Shneier, and A. Rosenfeld. Border extraction using linked edge pyramids. *IEEE Transactions on Systems, Man, and Cybernetics*, 12(5):660–668, September/October 1982.

[26] G. M. Hunter. *Efficient computation and data structures for graphics*. PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.

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

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

[29] G. Kedem. The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference*, pages 352–357, Las Vegas, NV, June 1982.

[30] A. Klinger. Patterns and search statistics. In J. S. Rustagi, editor, *Optimizing Methods in Statistics*, pages 303–337. Academic Press, New York, 1971.

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

[32] D. Meagher. Octree encoding: a new technique for the representation, manipulation, and display of arbitrary 3-D objects by computer. Electrical and Systems Engineering IPL-TR-80-111, Rensselaer Polytechnic Institute, Troy, NY, October 1980.

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

[34] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, 1969.

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

[36] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proceedings of the ACM SIGMOD Conference*, pages 270–277, San Francisco, May 1987.

[37] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 294–305, Portland, OR, June 1989.

[38] J. L. Pfaltz and A. Rosenfeld. Computer representation of planar regions by their skeletons. *Communications of the ACM*, 10(2):119–122, February 1967.

[39] M. Pietikäinen and A. Rosenfeld. Image segmentation by texture using pyramid node linking. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(12):822–825, December 1981.

[40] A. Rappoport. Using convex differences in hierarchical representations of polygonal maps. In *Proceedings of Graphics Interface'90*, pages 183–189, Halifax, Newfoundland, Canada, May 1990.

[41] A. A. G. Requicha and H. B. Voelcker. Solid modeling: a historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, 2(2):9–24, March 1982.

[42] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 10–18, Ann Arbor, MI, April 1981.

[43] A. Rosenfeld. Quadtrees and pyramids. In *Proceedings of the 5th International Conference on Pattern Recognition*, pages 802–811, Miami Beach, December 1980.

[44] A. Rosenfeld. Some useful properties of pyramids. In A. Rosenfeld, editor, *Multiresolution Image Processing and Analysis*, pages 2–5. Springer-Verlag, Berlin, West Germany, 1984.

[45] A. Rosenfeld. Axial representations of shape. *Computer Vision, Graphics, and Image Processing*, 33(2):156–173, February 1986.

[46] A. Rosenfeld. Pyramid algorithms for finding global structures in images. *Information Sciences*, 50(1):23–24, January 1990.

[47] A. Rosenfeld and A. C. Kak. *Digital Picture Processing*. Academic Press, New York, second edition, 1982.

[48] A. Rosenfeld and J. L. Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM*, 13(4):471–494, October 1966.

[49] D. Rutovitz. Data structures for operations on digital images. In G. C. Cheng, R. S. Ledley, D. K. Pollock, and A. Rosenfeld, editors, *Pictorial Pattern Recognition*, pages 105–133. Thompson Book Co., Washington, DC, 1968.

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

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

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

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

[54] H. Samet. Distance transform for images represented by quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(3):298–303, May 1982. Also University of Maryland Computer Science TR-780.

[55] H. Samet. A quadtree medial axis transform. *Communications of the ACM*, 26(9):680–693, September 1983. Also corrigendum, *Communications of the ACM*, 27(2):151, February 1984.

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

[57] H. Samet. Reconstruction of quadtrees from quadtree medial axis transforms. *Computer Vision, Graphics, and Image Processing*, 29(3):311–328, March 1985.

[58] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[59] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[60] H. Samet. Spatial data structures. In W. Kim, editor, *Modern Database Systems, The Object Model, Interoperability and Beyond*, pages 361–385. ACM Press and Addison-Wesley, New York, 1995.

[61] H. Samet and A. Rosenfeld. Quadtree structures for region processing. In L. S. Baumann, editor, *Proceedings of the ARPA Image Understanding Workshop*, pages 36–41, Los Angeles, November 1979. Also SAIC Technical Report SAI-80-974-WA.

[62] H. Samet and A. Rosenfeld. Quadtree representations of binary images. In *Proceedings of the 5th International Conference on Pattern Recognition*, pages 815–818, Miami Beach, December 1980.

[63] H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber. Quadtree region representation in cartography: experimental results. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(6):1148–1154, November/December 1983.

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

[65] H. Samet, C. A. Shaffer, R. C. Nelson, Y. G. Huang, K. Fujimura, and A. Rosenfeld. Recent developments in linear quadtree-based geographic information systems. *Image and Vision Computing*, 5(3):187–197, August 1987.

[66] H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):579–586, July 1988.

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

[68] C. A. Shaffer. *Application of alternative quadtree representations.* PhD thesis, Computer Science Department, University of Maryland, College Park, MD, June 1986. Also Computer Science TR-1672.

[69] C. A. Shaffer and H. Samet. Optimal quadtree construction algorithms. *Computer Vision, Graphics, and Image Processing*, 37(3):402–419, March 1987.

[70] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the First International Conference on Expert Database Systems*, pages 353–364, Charleston, SC, April 1986.

[71] M. Tamminen. The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica*, 1981. Also Mathematics and Computer Science Series No. 34.

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

[73] S. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.

[74] T. Ulrich. Loose octrees. In M. DeLoura, editor, *Game Programming Gems*, pages 444–453. Charles River Media, Rockland, MA, 2000.

[75] R. E. Webber and H. Samet. Linear-time border-tracing algorithms for quadtrees. *Algorithmica*, 8(1):39–54, 1992. Also University of Maryland Computer Science TR 2309.