Object-Based and Image-Based Representations of Objects by their Interiors*

Hanan Samet

Abstract

An overview is presented of object-based and image-based representations of objects by their interiors that can be used to answer two fundamental queries in image understanding: (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). Representations based on collections of unit-size cells and blocks are described that are designed to answer just one of these queries, while representations based on the use of hierarchies of space and objects are presented enabling the efficient response to both queries.

1: Introduction

The representation of spatial objects and their environment is an important issue in applications of computer graphics, computer vision, image processing, robotics, and pattern recognition as well as in building databases to support them (e.g., [25, 26]). We assume that the objects are connected[1] 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 or similar shapes 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. This distinction is commonly used to characterize algorithms in computer graphics (e.g., [8]). Regardless of the characterization, the objects (as well as their constituent cells) can be represented either by their interiors or by their boundaries. In this article our focus is on interior-based representations.

Each of the decompositions has its advantages and disadvantages. They depend primarily on the nature of the queries that are posed to the database. The most general queries ask *where*, *what*, *who*, *why*, and *how*. The ones that are relevant to our application are *where* and *what*. They are stated more formally as follows:

---

*This work was supported in part by the National Science Foundation under Grant IRI–92–16970. Appears in *Advances in Image Understanding - A Festschrift for Azriel Rosenfeld*, K. Bowyer and N. Ahuja, eds., IEEE Computer Society Press, Los Alamitos, CA, 1996, 316–332.

[1]Intuitively, this means that a d-dimensional object cannot be decomposed into disjoint subobjects so that the subobjects are not adjacent in a $(d-1)$-dimensional sense.

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).

Not surprisingly, the queries can be classified using the same terminology that we used in the characterization of the decomposition. In particular, we can either try to find the cells (i.e., their locations in space) occupied by an object or find the objects that overlap a cell (i.e., a location in space). If objects are associated with cells so that a cell contains the identity of the relevant object (or objects), then query 1 is analogous to retrieval by contents while query 2 is analogous to retrieval by location.

Queries 1 and 2 are the basis of two more general classes of queries. Query 1 is known as a *feature-based* (also *object-based*) query, while query 2 is known as a *location-based* (also *image-based* or *cell-based*) query. Query 2 is a special case of a wider range of queries known as *window queries* which retrieve the objects that cover an arbitrary region (often rectangular). These queries are used in a number of applications including geographic information systems (e.g., [1, 26]).

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. Ideally, we wish to be able to answer both types of queries with one representation. At times, more compact representations are desired in which case we make use of techniques that aggregate identically-valued contiguous cells, or even objects which, ideally, are in proximity. These queries and a number of different representations and indexes that facilitate responding to them are discussed in this article which is organized as follows. Section 2 examines representations based on collections of unit-size cells while Section 3 treats the case of blocks. Section 4 looks at the use of hierarchies of space and objects which enable efficient responses to both queries 1 and 2, while Section 5 contains concluding remarks.

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 the 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 a *location* of a cell in space is quite different from that of an *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 orthogonal to each other and parallel to the coordinate axes. In our discussion, we assume that the cells comprising a particular object are contiguous, or equivalently continuous (i.e., adjacent), and that a different unique value is associated with each distinct object thereby enabling us to distinguish between the objects. Depending on the underlying representation, this value may be stored with the cells. For example, Figure 1 contains three two-dimensional objects A, B, and C and their corresponding cells.

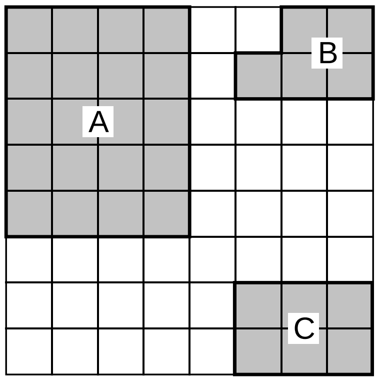

Figure 1: Example collection of three objects and the cells that they occupy.

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 identity of the contiguous cells that form the object is hardwired into the representation. An example of 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 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*). This index does not make use of the spatial coverage of the objects and thus may be implemented using conventional searching techniques such as hashing [17]. In this case, we will need $O(n)$ additional space for the index, where n is the number of different objects. We do not discuss such indexes here.

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 only be answered by checking for the presence of location l in space in the various sets associated with the different objects. This will be time-consuming as it may require that we examine all cells in each set. It should be clear that the explicit representation could also be classified as being *object-based* as it clearly lends itself only to retrieval on the basis of knowledge of the objects rather than of the locations of the cells in space. We shall make use of this characterization in Section 4.

Note that since the explicit representation consists of sets, there is no particular order for the cells within each set although an ordering could be imposed based on spatial proximity of the locations of the cells in space, etc. For example, the list representation of a set already presupposes the existence of an ordering. Such an ordering could be used to obtain a small, but not insignificant, decrease in the time (in an expected sense) needed to answer query 2. In particular, now whenever cell c is not associated with object o, we will be able to cease searching the list associated with o after having inspected half of the cells associated with o instead of all of them, which is the case when no ordering exists.

An important shortcoming of the use of the explicit representation, which has an effect somewhat related to the absence of an ordering, is the inability to distinguish between occupied and unoccupied cells. In particular, in order to detect that a cell c is not occupied by any object we must examine the sets associated with each object, which is quite time-consuming. Of course, we could avoid this problem by forming an additional set which contains all of the unoccupied cells, and examine this set first whenever processing query 2. The drawback of such a solution is that it slows down all instances of query 2 that involve cells that are occupied by objects.

We can avoid examining every cell in each object set, thereby speeding up query 2 in certain cases, by storing a simple approximation of the object with each object set o. This approximation should be of a nature that makes it easy to check if it is impossible for a location l in space to be in o. One such approximation is a minimum bounding box whose sides are parallel to the coordinate axes of the space in which the object is embedded. For example, for object `B` in Figure 1 such a box is anchored at the lower-left corner of cell `(5,1)` and the right corner of cell `(7,0)`. The existence of a box b for object o means that if b does not contain l, then o does not contain l either, and we can proceed with checking the other objects. This bounding box is usually a part of the explicit representation

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 right 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 where 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*. The index is really an ordering and hence its range is usually the integers (i.e., one-dimensional). When the data is multidimensional (i.e., cells in d-dimensional space where $d > 0$), it may not be convenient to use the location in space corresponding to the cell as an index since its range spans data in several dimensions. Instead, we employ techniques such as laying out the addresses corresponding to the locations in space of the cells in some particular order and then making use of an access structure in the form of a mapping function to enable the quick association of addresses with the locations in space corresponding to the cells. Retrieving the address is more complex in the sense that it can be a simple memory access or may involve an access to secondary or tertiary storage if virtual memory is being used. In most of our discussion, we assume that all data is in main memory, although, as we will see, a number of the representations do not rely on this assumption.

Such an access structure enables us to obtain the contiguous cells (as we know their locations in space) without having to examine all of the cells. Therefore, we will know the identities of the cells that comprise an object thereby enabling us to complete the response to query 2 with an implicit representation. In the rest of this section, we discuss a number of such access structures. However, before proceeding further, we wish to point out that the implicit representation could also be classified as being *image-based* as it clearly lends itself to retrieval on the basis of knowledge only of the cells rather than the objects. We shall make use of this characterization in Section 4.

The existence of an access structure also enables us to answer query 1 with the implicit representation although it 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 will be time-consuming as it may require that we examine all the cells.

There are many ways of laying out the addresses corresponding to the locations in space of the cells each having its own mapping function. Some of the most important ones for a two-dimensional space are illustrated in Figure 2 for an 8×8 portion of the space. To repeat, in essence, what we are doing is providing a mapping from the d-dimensional space containing the locations of the cells to the one-dimensional space of the range of index values (i.e., integers) to a table whose entries contain the addresses where information about the contents of the cells is stored. The result is an ordering of the space, and the curves shown in Figure 2 are termed *space-filling curves*.

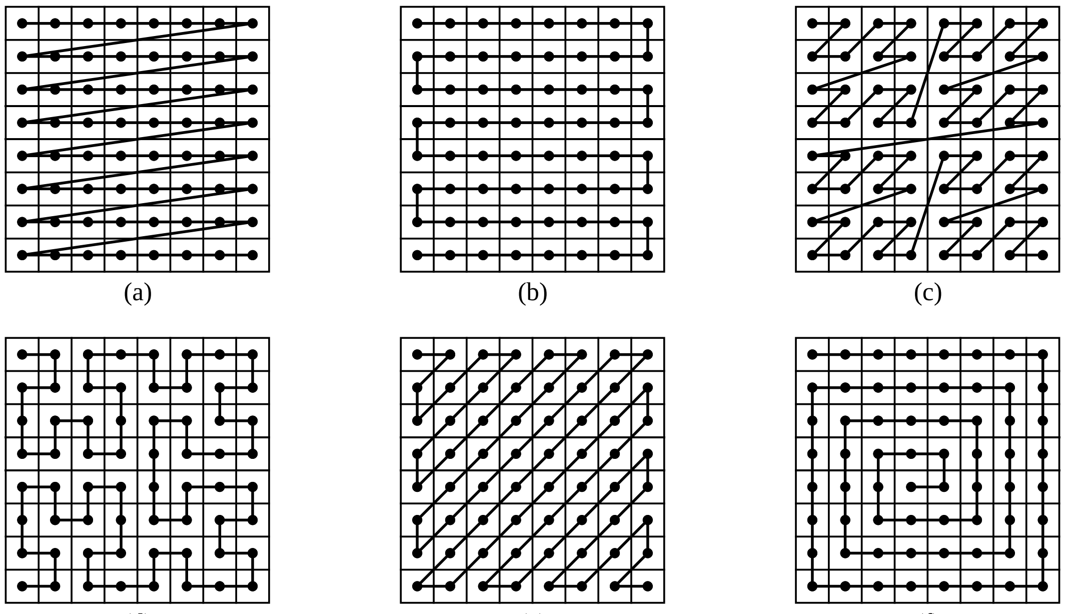

Figure 2: The result of applying a number of different space-ordering methods to an 8×8 collection of cells whose first element is in the upper-left corner: (a) row order, (b) row-prime order, (c) Morton order, (d) Peano-Hilbert order, (e) Cantor-diagonal order, (f) spiral order.

The multidimensional array (i.e., 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 array is only a conceptual multidimensional structure (it is not a multidimensional physical entity in memory) in the sense that it is a mapping of the locations in space of the cells into sequential addresses in memory. The actual addresses are obtained by the array access function which is based on the extents of the various dimensions (i.e., coordinate axes). The array access function is usually the mapping function for the row order illustrated in in Figure 2a (although, at times, the column order is also used). Thus the array enables us to implement the implicit representation with no additional storage except for what is needed for the array's descriptor. The descriptor contains the bounds and extents of each of the dimensions which are used to define the mapping function (i.e., they determine the values of its coefficients) so that the appropriate address can be calculated given the cell's location in space.

The array is called a *random access structure* as the address associated with a location in space can be retrieved in constant time independent of the number of elements in the array and doesn't require search. Note that we could store the object identifier o in the array element itself instead of allocating a separate address a for o thereby saving some space.

The array is an implicit representation because we have not explicitly aggregated all the contiguous cells that comprise a particular object. They can be obtained given a particular cell c at a location l in space belonging to object o by recursively accessing the array elements corresponding to the locations in space that are adjacent to l and checking if they are associated with object o. This process is known as depth-first connected component labeling (e.g., [21]).

Interestingly, depth-first connected component labeling could also be used to answer query 1 efficiently with an implicit representation if we add a data structure such as an index on the objects (e.g., a table of object-location pairs where *location* is one of the locations in space that comprise *object*). Thus given an object o we use the index to find a location in space that is part of o and then proceed with the depth-first connected component labeling as before. This index does not make use of the spatial coverage of the objects and thus it can be implemented using conventional searching techniques such as hashing [17]. In this case, we will need $O(n)$ additional space for the index where n is the number of different objects. We do not discuss such indexes here.

Of course, we could also answer query 2 with an explicit representation by adding an index which associates objects with locations in space (i.e., having the form location-objects). However, this would require $O(s)$ additional space for the index where s is the number of cells. The $O(s)$ bound assumes that only one object is associated with each cell. If we take into account that a cell could be associated with more than one object, then the additional storage needed is $O(ns)$ if we assume n objects. Since the number of cells s is usually much greater than the number of objects n, the addition of an index to the explicit representation is not as practical as extending the implicit representation with an index of the form object-location as described above. Thus it would appear that the implicit representation is more useful from the point of view of flexibility when taking storage requirements in to account.

The implicit representation can 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 or not the cell is associated with any of the objects. In this case, we choose only to keep track of the non-empty cells.

We have two ways to proceed. The first is to use one of a number of multidimensional access structures such as a point quadtree, k-d tree, MX quadtree, etc. as described in [24]. The second is to make use of one of the orderings of space shown in Figure 2 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 cells making up the object collection be of a 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, then 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 with a set of rules that guide it. 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. Some examples include:

- congruent blocks at each stage
- similar blocks at all stages
- all but one side of a block are unit-sized
- all sides of a block are of equal size
- all sides of each block are powers of two
- etc.

Other decomposition rules dispense with the requirement that the blocks be rectangular, while still others do not require that they be orthogonal. 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 and then fixes the coordinate values of all but one of the dimensions, say i, and then varies the value of the i^{th} 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* [22]. 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 as well as orthogonal although again this is not absolutely necessary as there exist decompositions using other shapes as well (e.g., triangles, etc.).

It is easy to adapt the explicit representation to deal with blocks resulting from an arbitrary decomposition (which also includes the one that yields one-dimensional blocks). In particular, instead of associating a set with each object o that contains the location in space of each cell that comprises o, we need to associate with each object o the locations in space and sizes of each block that comprises o. This can be done by specifying the coordinate values of the upper-left corner of each block and the sizes of its sides. This format is appropriate, and is the one we use, for the explicit representation of all of the block decompositions described in this section.

Using the explicit representation of blocks, both queries 1 and 2 are answered in essentially the same way as they were for unit-sized cells. The only difference is that for query 2 instead of checking if a particular location l in space is a member of one of the sets of cells associated with the various objects, we must check if l is covered by one of the blocks in the sets of blocks of the various objects. This is a fairly simple process as we know the location in space and size of each of the blocks.

An implementation of an arbitrary decomposition (which also includes the one that results in one-dimensional blocks) using an implicit representation is quite easy as long as the decomposition yields disjoint blocks, which is the case for all of the decompositions discussed in this section. Disjointness is important because it means that only one block can be associated with a location in space. Thus we can build an index based on an easily identifiable location in each block such as its upper-left corner. Therefore, we can, and do, make use of the same techniques that were presented in the discussion of the implicit representation for unit-sized cells in Section 2.

As in the case of unit-size cells, regardless of which tree access structure is used on the identifiable location, we determine the object o associated with a cell at location l by finding the block b that covers l. If b is an empty block, then we exit. Otherwise, we return o. Notice that the search for the block that covers l may be quite complex in the sense that the access structures may not necessarily achieve as much pruning of the search space as in the case of unit-sized cells. In particular, this is the case whenever the space ordering that is applied does not have the property that all of the cells in each block appear in consecutive order. In other words, given the cells in the block e with minimum and maximum values in the ordering, say u and v, there exists at least one cell in block f distinct from e which is mapped to a value w where $u < w < v$. Thus a search for the block b that covers l may require that we visit several subtrees of a particular node in the tree-like access structures.

Perhaps the most widely known decompositions into blocks are those referred to by the general terms *quadtree* and *octree* [23, 24]. They are usually used to describe a class of representations for two and three-dimensional data (and higher as well), 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. It is interesting to note that quadtrees and octrees may be used with both interior-based and boundary-based representations. Moreover, both explicit and implicit aggregations of the blocks are possible.

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* [15] and *region octrees* [14, 18]. They are specific examples of interior-based representations for two 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 3a is the block decomposition for the region quadtree corresponding to Figure 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, then 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 $b \bmod 2^i = 0$.

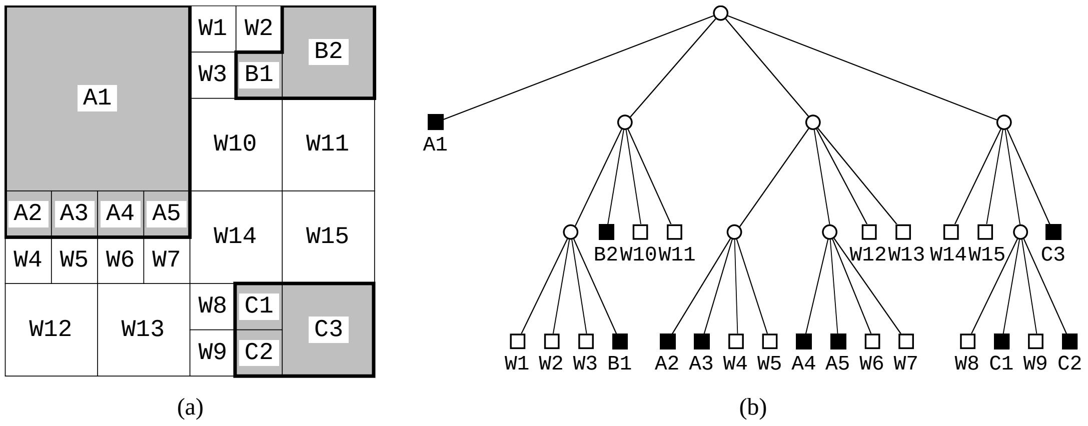

Figure 3: (a) Block decomposition and (b) its tree representation for the collection of objects and cells in Figure 1.

A region quadtree can be implemented using an explicit representation by associating a set with each object o that contains its constituent blocks. Each block is are specified by numbers corresponding to the coordinate values of its upper-left corner and the size of one of its sides. These numbers are stored in the set in the form $(a,b):d$ where (a,b) and d correspond to the coordinate values of the upper-left corner and depth, respectively, of the block. For example, the explicit representation of the collection of blocks n Figure 1 is given by the sets `A={(0,0):2,(0,4):0,(1,4):0,(2,4):0,(3,4):0}`, `B={(5,1):0,(6,0):1}`, and `C={(5,6):0,(5,7):0,(6,6):1}`, which correspond to blocks {A1,A2,A3,A4,A5}, {B1,B2}, and {C1,C2,C3}, respectively.

A region quadtree implementation that makes use of an implicit representation is quite different. First, we allocate an address a in storage for each block b which stores an identifier that indicates the identity of the object (or objects) of which b is a member. Second, it is necessary to impose an access structure on the collection of blocks in the same way as the array was imposed on the collection of unit-sized cells. Such an access structure enables us to determine easily the value associated with any point in the space covered by a cell without resorting to exhaustive search. Note that depending on the nature of the access structure, it's not always necessary to store the location and size of each block with a.

There are many possible access structures. Interestingly, using an array as an access structure is not particularly useful as it defeats the rationale for the aggregation of cells into blocks unless, of course, all the blocks are of a uniform size in which case we have the analog of a two-level grid.

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 3b corresponding to the collection of two-dimensional objects in Figure 1 whose quadtree block decomposition is given in Figure 3a). Each leaf node in the tree corresponds to a different block b and contains the address a in storage where an identifier is stored that indicates the identity of the object (or objects) of which b is a member. As in the case of the array, where we could store the object identifier o in the array element itself instead of allocating a separate address a for o, we could achieve the same savings by storing o in the leaf node of the tree. 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 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 [5]). 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).

Using a tree with fanout 2^d as an access structure for a regular decomposition means that there is no need to record the size and location of the blocks. This information can be inferred from knowledge of the size of the underlying space as the 2^d blocks that result from each subdivision step are congruent. For example, in two dimensions, each level of the tree corresponds to a quartering process that yields four congruent blocks (rectangular here, although a triangular decomposition process could also be defined which yields four equilateral triangles; however, in such a case, we are no longer dealing with rectangular cells). Thus as long as we start from the root, we know the location and size of every block.

There are a number of 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., [23]). The simplest is to use the same technique that we applied to the collection of blocks of arbitrary size. In particular, we can apply one of the orderings of space shown in Figure 2 to obtain a mapping from the coordinate values of the upper-left corner u of each block to the integers.

Since the size of each block b in the region quadtree can be specified with a single number indicating the depth in the tree at which b is found, we can simplify the representation by incorporating the size into the mapping. One mapping simply concatenates the result of interleaving the binary representations of the coordinate values of the upper-left corner (e.g., (a, b) in two dimensions) and i of each block of size 2^i so that i is at the right. The resulting number is termed a *locational code* and is a variant of the Morton order (Figure 2c). Assuming such a mapping and sorting the locational codes in increasing order yields an ordering equivalent to that which would be obtained by traversing the leaf nodes (i.e., blocks) of the tree representation (e.g., Figure 3b) in the order `NW`, `NE`, `SW`, `SE`.

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 a practical implementation of the tree access structure. In this case, an access structure termed a *bintree* [16, 27, 30] 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 [7] 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 [4] 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 at an arbitrary order, and, 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* [10] and the *LSD tree* [13] which were originally developed for points. It can also be regarded as a special case of the *BSP tree* (denoting *Binary Space Partitioning*) [11]. 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 an arbitrary orientation. 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 a dimension i, then we must perform $p-1$ successive partitions on dimension i. Once these $p-1$ partitions are complete, we partition along another dimension. The *puzzletree* [6] is a further generalization of the k-d tree that decomposes the space 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.

The puzzletree is motivated by a desire to overcome the rigidity in the shape, size, and position of the blocks that result from the bintree (and to an equivalent extent, the region quadtree) partitioning process (because of its regular decomposition). In particular, in many cases, the decomposition rules ignore the homogeneity present in certain regions on account of the need to place the partition lines in particular positions as well as a possible limit on the number of permissible partitions along each dimension at each decomposition step. Often, it is desirable for the block decomposition to follow the perceptual characteristics of the objects as well as reflect their dominant structural features.

For example, consider a front view of a scene containing a table and two chairs. Figures 4a and 4b are the block decompositions resulting from the use of a bintree and a puzzletree, respectively, for this scene, while Figure 4c is the tree access structure corresponding to the puzzletree in Figure 4b. Notice the natural decomposition in the puzzletree of the chair into the legs, seat, and back, and of the table into the top and legs. On the other hand, the blocks in the bintree (and to a greater extent in the region quadtree, although not shown here) do not have this perceptual coherence.

4: Hierarchical Representations

Assuming the presence of an access structure, the implicit representations described in Sections 2 and 3 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, the explicit representation that we described is 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 whether the hierarchy is 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 (i.e., image-based) representations, while the latter is applicable to explicit (i.e., object-based) representations. Thus, we see again that the distinction is the same as that used in computer graphics to distinguish between algorithms as being image-space or object-space [8], respectively.

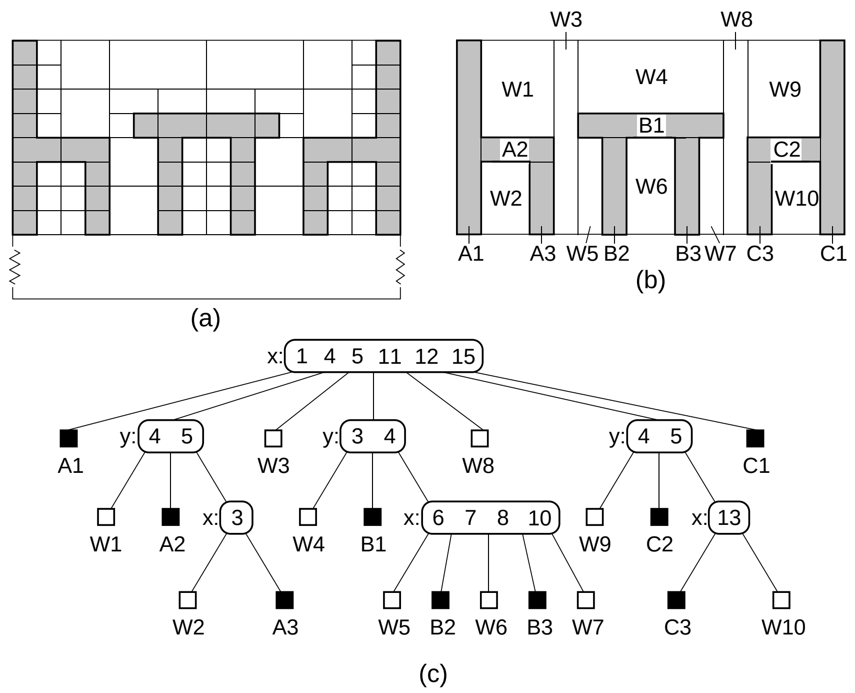

Figure 4: Block decomposition for the (a) bintree and (b) puzzletree corresponding to the front view of a scene containing a table and two chairs; (c) is the tree access structure for the puzzletree in (b).

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 member. 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* [31] 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 5 shows the pyramid corresponding to the collection of objects and cells in Figure 1 and the labels in Figure 3a. In this case, we are aggregating 2×2 cells and blocks. Notice the similarity between the pyramid and the quadtree implementation that uses an access structure which is a tree with a fanout of 4 Figure 3b).

Nevertheless, it is important to distinguish the pyramid from the 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). Hence the quadtree is an instance of what is termed a *variable resolution* representation, which, of course, is not limited to rectangular blocks that are square. In particular, it can be used with a limited number of other non-rectangular shapes (most notably, triangles in two dimensions [3, 24]).

The pyramid can be viewed as a complete quadtree (i.e., where no aggregation takes place at the deepest level, or, equivalently, all leaf nodes with zero sons are at maximum depth in the tree). Nevertheless, there are some very important differences. The first is that

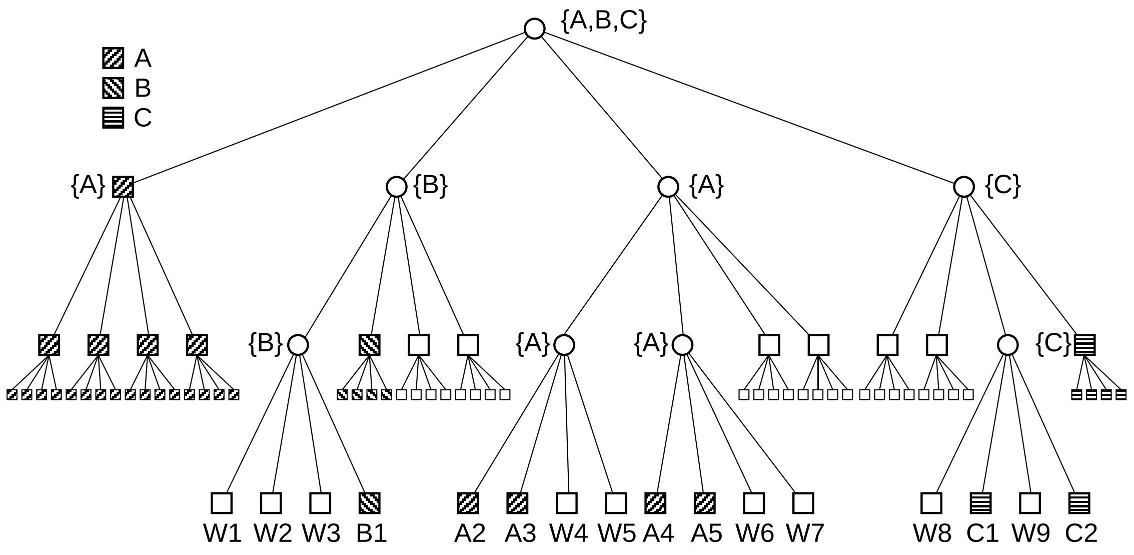

Figure 5: Pyramid for the collection of objects and cells in Figure 1.

the quadtree is a variable-resolution representation while the pyramid is a multiresolution representation. The second, and most important, difference is that in the case of the quadtree, the nonleaf nodes just serve as an access structure. They do not include any information about the objects present in the nodes and cells below them. This is why the quadtree, like the array, is not useful for answering query 1. Of course, we could also devise a variant of the quadtree (termed a *truncated-tree pyramid* [25]) 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 identity 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 [12] is an example of an object hierarchy which finds use 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 6a is an example R-tree for a collection of rectangle objects with $m = 2$ and $M = 3$. Figure 6b shows the spatial extent of the bounding boxes of the nodes in Figure 6a, 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.

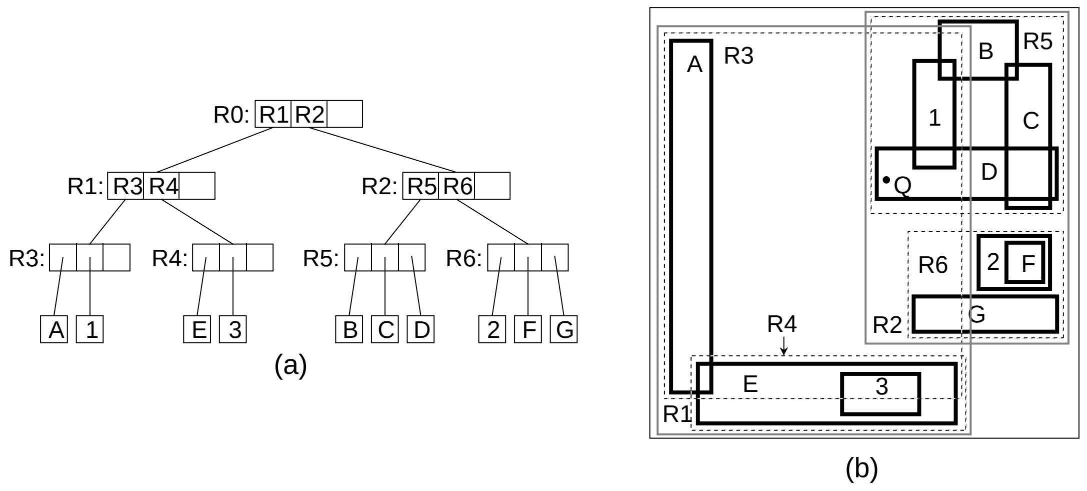

Figure 6: (a) R-tree for a collection of rectangle objects with m=2 and M=3, and (b) the spatial extents of the bounding boxes. Notice that the leaf nodes in the index also store bounding boxes although this is only shown for the nonleaf nodes.

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 was 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 only associated with one node while being contained in possibly many bounding boxes (e.g., in Figure 6, 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 k-d-B-tree [20] and the R^+-tree'[29]) which is a form of a 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 being linear to being 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 [8].

Furthermore, 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, usually, the base representation of the hierarchy is still 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 basis of this hierarchy) to answer query 2, while in the case of the object hierarchy, we use the explicit representation (which is the basis of this hierarchy) to answer query 1.

5: Concluding Remarks

We have reviewed a number of image-based and object-based representations of objects by their interiors with a focus on answering queries 1 and 2. For more details about some of these representations, see [23, 24]. Of course, there are also representations based on the boundaries of the objects, such as vectors and chain codes (e.g., [9]), which find much use. Although we do not go into great detail here about these representations, it is important to point out that one of the principal drawbacks of boundary-based representations is the difficulty in determining the value associated with an arbitrary point of the space given by the cell (i.e., query 2) without testing each boundary element using operations such as point-in-polygon tests (e.g., [8]) or finding the nearest boundary element. The problem is that these representations generally just indicate which boundary element is adjacent to which other boundary element rather than their relationship to the space that they occupy. This situation can be remedied by imposing an access structure such as an appropriate variant of a quadtree or octree which provides a way to index the boundary elements. For example, a variant of the PM quadtree (e.g., [28, 19]) can be used for a polygon representation in two dimensions and a PM octree (e.g., [2]) can be used similarly for three dimensions.

References

[1] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 265–272, Nashville, TN, April 1990. (Also *Proceedings of the Fifth Brazilian Symposium on Databases*, Rio de Janeiro, Brazil, April 1990, 15–26).

[2] 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.

[3] 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.

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

[5] 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.

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

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

[8] 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.

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

[10] 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.

[11] 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).

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

[13] A. Henrich, H. W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional point and non-point data. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, P. M. G. Apers and G. Wiederhold, eds., pages 45–53, Amsterdam, The Netherlands, August 1989.

[14] G. M. Hunter. *Efficient computation and data structures for graphics.* PhD thesis, Princeton University, Princeton, NJ, 1978.

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

[16] 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.

[17] D. E. Knuth. *The Art of Computer Programming vol. 3, Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

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

[19] 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).

[20] 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.

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

[22] D. Rutovitz. Data structures for operations on digital images. In *Pictorial Pattern*

Recognition, G. C. Cheng et al., ed., pages 105–133. Thompson Book Co., Washington, DC, 1968.

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

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

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

[26] H. Samet and W. G. Aref. Spatial data models and query processing. In *Modern Database Systems, The Object Model, Interoperability and Beyond*, W. Kim, ed., pages 338–360. ACM Press and Addison-Wesley, New York, NY, 1995.

[27] 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.

[28] 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*, Washington, DC, June 1983, 127–132; and University of Maryland Computer Science TR–1372).

[29] 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.

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

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