Object-based and Image-based Image Representations

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

October 23, 1998

1 Copyright ©1998 by Hanan Samet. These notes may not be reproduced by any means (mechanical, electronic, or any other) without the express written permission of Hanan Samet.
Object-based and Image-based Image Representations

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 (e.g., [7, 108, 107, 127]). The problem also arises in building databases to support them (e.g., [109]). We assume that the objects are connected\(^2\) 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., [46]).

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:

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. As we will see, it is important to note that there is a distinction between a location in space and the address where information about the location is stored, which, unfortunately, is often erroneously assumed to be the same.

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., [6, 110]).

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 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. Once again, we reiterate that 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. This distinction between the location in space of a cell and the address of a cell is important and we shall make use of it often.

In most applications (including most of 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,

\(^2\)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.
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. Note that, although not the case in this example, objects are allowed to overlap which means that a cell may be associated with more than one object. Here we assume, without loss of generality, that the volume of the overlap must be an integer multiple of the volume of a cell (i.e., pixels, voxels, etc.).

![Figure 1: Example collection of three objects and the cells that they occupy.](image)

The shape of an object o can be represented either by the interiors of the cells comprising o, or by the subset of the boundaries of those cells comprising o that are adjacent to the boundary of o. In other words, interior-based methods represent an object o by using the locations in space of the cells that comprise o, while boundary-based methods represent o by using the locations in space of the cells that are adjacent to the boundary of o. Operations on the cells (i.e., locations in space) comprising a particular object o are facilitated by aggregating the cells of the objects into subcollections of contiguous identically-valued cells, or contiguous identically-valued cells adjacent to the boundary with contiguous boundary elements having the same direction. The aggregation may be implicit or explicit depending on how the contiguity of the cells that make up the aggregated subcollection is expressed.

In general, interior-based representations make it very easy to calculate properties of an object such as its mass, and, depending on the nature of the aggregation process, to determine the value associated with any point (i.e., location) in the space covered by a cell in the object. On the other hand, boundary-based representations make it easy to obtain the boundary of an object.

Regardless of the representation that is used, the generation of responses to queries 1 and 2 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 want to be able to answer both types of queries with one representation. This is somewhat tricky, but, as we will show, it is doable.

The rest of this chapter is organized as follows. Section 1 discusses interior-based representations. These are the most prevalent and thus this discussion forms the main part of the chapter. Section 2 discusses boundary-based representations. This section is much briefer as the principal representation is the boundary model and the principal example of it is the winged-edge data structure which is discussed in Section ?? of Chapter ??.

The discussion in these sections shows how some 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. The representations can be made even more compact by only recording the differences between the elements. The use of difference-based compaction methods is the subject of Section 3.

As stated earlier, most of our discussion assumes that the objects can be decomposed into cells whose boundaries are orthogonal and parallel to the coordinate axes. Nevertheless, we also give a brief discussion of the representation of objects with arbitrary boundaries (e.g., edges of arbitrary slope, nonplanar faces, etc.).
In this case, the representation of the underlying environment is usually, but not always, based on a decomposition into cells whose boundaries are orthogonal and parallel to the coordinate axes.

1 Interior-based Representations

In this section we focus on interior-based representations. Section 1.1 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 a unit size and permits their sizes to vary. The varying-sized cells are termed blocks and are the subject of Section 1.2. The representations described in Sections 1.1 and 1.2 assume that each unit-size cell or block is contained entirely in one or more objects. The cell or block cannot be partially contained in two objects. This means that each cell in each block belongs to the same object or objects or that the cells do not belong to any of the objects. Section 1.3 permits a cell or a block to be a part of more than one object. 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 (i.e., they need not be orthogonal and parallel to the coordinate axes). Section 1.4 looks at the use of hierarchies of space and objects which enable efficient responses to both queries 1 and 2. Section ?? contains a summary.

1.1 Unit-size Cells

Interior-based representations aggregate identically-valued cells by recording their locations in space. We say that a representation that makes use of 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 (see Section ?? of Chapter ??). 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 [79] (see Chapter ??). 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 1.4.

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.

There are a number of ways of increasing the quality of the approximation. For example, the minimum bounding box may be rotated by an arbitrary angle so that the sides are still orthogonal while no longer having to be parallel to the coordinate axes (e.g., [19]). The box may also be replaced by a circle, sphere, ellipse, etc. (e.g., [19]). Alternatively, the box can be replaced by a polyhedron with either a fixed (e.g., [19, 39]) or an arbitrary number of sides (e.g., the P-tree [71] and the minimum bounding polybox [20], as well as a convex hull [19]). Restricting the sides (i.e., faces in dimension higher than 2) of the polyhedron to be parallel to the coordinate axes (termed an orthogonal polygon) enables simpler point-in-object tests (e.g., the vertex representation [37, 38] as used in [39] and discussed in greater detail in Section 3).

In the rest of this chapter we restrict our discussion to minimum bounding boxes that are rectangles with sides parallel to the coordinate axes, although, of course, the techniques we describe are applicable to other more general bounding objects. Nevertheless, in the interest of brevity, we use the term bounding box even though the terms minimum bounding box or minimum bounding object would be more appropriate.

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 as 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 1.4.

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 $8 \times 8$ portion of the space and described briefly below. 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](image_url)

Figure 2: The result of applying a number of different space-ordering methods to an $8 \times 8$ 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.
Choosing among the space-filling curves illustrated in Figure 2 is not easy as each one has its advantages and disadvantages. Below, we review a few of their desirable properties and show how some of the two-dimensional orderings satisfy them, leaving it to you to explore them further.

- The curve should pass through each location in space once and only once.

- The mapping from the higher-dimensional space to the integers should be relatively simple and likewise for the inverse mapping. This is the case for all but the Peano-Hilbert order (Figure 2d). For the Morton order (Figure 2c), the mapping is obtained by interleaving the binary representations of the coordinate values of the location of the cell. The number associated with each cell is known as its **Morton number**.

- The ordering should be stable. This means that the relative ordering of the individual locations is preserved when the resolution is doubled (e.g., when the size of the two-dimensional space in which the cells are embedded grows from $8 \times 8$ to $16 \times 16$) or halved assuming that the origin stays the same. The Morton and Cantor-diagonal (Figure 2e) orders are stable while the row (Figure 2a), row-prime (Figure 2b), Peano-Hilbert, and spiral (Figure 2f) orders are not stable (but see Exercises 1 and 2).

- Two locations that are adjacent (i.e., in the sense of a $(d-1)$-dimensional adjacency also known as 4-adjacent) in space are neighbors along the curve and vice versa. This is impossible to satisfy for all points at all space sizes. However, for the row-prime, Peano-Hilbert, and spiral orders, every element is a 4-adjacent neighbor of the previous element in the sequence while this is not the case for the other orders. This means that the row-prime, Peano-Hilbert, and spiral orders have a slightly higher degree of locality than the other orders.

- The process of retrieving the neighbors of a location in space should be simple.

- The order should be admissible. This means that at each point in the ordering, at least one 4-adjacent neighbor in each of the lateral directions (i.e., horizontal and vertical) must have already been encountered. This is useful in a number of algorithms (e.g., connected component labeling [32]). The row and Morton orders are admissible while the Peano-Hilbert order is not admissible. The row-prime, Cantor-diagonal, and spiral orders are admissible only if we permit the direction of the 4-adjacent neighbors to vary from point to point along the curve. For example, for the row-prime order, at points on odd rows, the previously encountered 4-adjacent neighbors are the western and northern neighbors, while at points on even rows, it is the eastern and northern neighbors.

The row order (Figure 2a) is of special interest to us because its mapping function is the one most frequently used by the multidimensional array (described in greater detail below), which is the most common access structure. Assuming that the first entry in the cell-address table indexed by the index has an index value of 0, location $(a,b)$ in an $8 \times 8$ collection of cells is mapped to index value $8b + a$. An alternative ordering that is also used in the multidimensional array is known as column order where the difference from row order lies in the order in which the various dimensions are scanned. In the column order, the $y$ coordinate value varies most rapidly while in the row order, the $x$ coordinate value varies most rapidly. The one to choose is arbitrary, although the row order is preferred as it yields a lexicographic ordering when array references are of the form $T[i,j]$ corresponding to an element in row $i$ and column $j$ of array $T$ (i.e., at point $(j,i)$).

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 gives a cell $c$ at a location $i$ 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 (see Section ?? of Chapter ?? as well as the above discussion on space orderings) 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 described above (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.

Figure 3 shows the contents of the array corresponding to the collection of two-dimensional objects and cells given in Figure 1. Each of the array elements $T[i, j]$ is labeled with an address that specifies where an identifier is stored that indicates the identity of the object (or objects) of which the corresponding cell at location $(j, i)$ is a member. We have labeled array elements corresponding to cells of object $O$ as $O_i$ and the array elements corresponding to cells that are not in any of the objects as $W_i$ using the suffix $i$ to distinguish between them in both cases.

![Figure 3: Array corresponding to the collection of objects and cells in Figure 1.](image)

The array is called a random access structure because the address associated with a location in space can be retrieved in constant time independent of the number of elements in the array and does not require any 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 (see Section ?? of Chapter ??).

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 [79] (see Chapter ??). 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 into 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 nonempty 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 Chapter ???. The second is to make use of one of the orderings of space shown in Figure 2 to obtain a mapping from the nonempty 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 nonempty cell is stored. In this discussion, we choose the second approach as often when the dimensionality of the space (i.e., $d$) gets large, the implementation of the first approach resorts to use of the second approach anyway.

At this point, it is worthwhile to comment on what exactly is stored in the nodes of the tree-like access structure that are associated with nonempty cells. First of all, as mentioned above, each node contains an address which indicates the physical location where the information about the object associated with the location in space corresponding to the nonempty cell is stored. As in the case of the array, we could store the object identifier $o$ in the node of the tree itself rather than allocating a separate address $a$ for $o$ thereby saving some space. In addition, each node of the tree must also store the index value (i.e., the result of the mapping) for the nonempty cell as unlike the array we do not have an entry for every cell which would have enabled us to avoid having to store the index value. Note that if we did not store this index value, we would not be able to tell if the node in the tree corresponds to a particular location through the use of a comparison. The mapping that we use must have an inverse so that given an index value we can determine the corresponding location in space of the cell that is associated with it.

\[
\begin{array}{cccccccccccc}
0 & 1 & 4 & 5 & 16 & 17 & 20 & 21 \\
2 & 3 & 6 & 7 & 18 & 19 & 22 & 23 \\
8 & 9 & 12 & 13 & 24 & 25 & 28 & 29 \\
10 & 11 & 14 & 15 & 26 & 27 & 30 & 31 \\
32 & 33 & 36 & 37 & 48 & 49 & 52 & 53 \\
34 & 35 & 38 & 39 & 50 & 51 & 54 & 55 \\
40 & 41 & 44 & 45 & 56 & 57 & 60 & 61 \\
42 & 43 & 46 & 47 & 58 & 59 & 62 & 63 \\
\end{array}
\]

Figure 4: Result of applying a Morton ordering to an $8 \times 8$ collection of cells.

We demonstrate the use of a tree access structure with the mapping that yields a Morton order (i.e., Figure 2c), and use it as an index in the binary search tree. Figure 4 shows the result of applying a Morton ordering to an $8 \times 8$ collection of cells. Figure 5 is the binary search tree for the collection of two-dimensional objects and cells given in Figure 1. Each node in the tree is labeled with the Morton order value of its corresponding cell $c$ and an address $a$ that specifies where an identifier is stored that indicates the identity of the object (or objects) of which $c$ is a member. The address $a$ in Figure 5 is the same as the one used in the corresponding cell in the array access structure in Figure 3. Notice that the empty cells (i.e., those labeled with $Wi$ in the array access structure in Figure 3 are not present in Figure 5.

Regardless of which tree access structure is used, we can always determine the object $o$ associated with a cell $c$ at location $l$. We just apply the mapping (Morton ordering in our example) to $l$ yielding $p$ (which is an
address or an offset value from a base location) and then search for \( p \) in the tree (a binary search tree in our example). If \( p \) is not in the tree, then \( c \) is an empty cell; otherwise, we retrieve the object identifier \( o \) associated with it. Recall that retrieving the object identifier \( o \) associated with cell \( c \) only yields a partial response to query 2 as we need to determine the remaining cells that comprise \( o \). This is achieved by accessing each of the cells adjacent to \( c \) and determining if they exist in the tree, and if so, then checking if they are associated with \( o \). If the answer is yes to both of the above, then their adjacent cells are examined by a recursive application of this process. Note that for most of the orderings we can determine the locations in space that are adjacent to \( c \) directly from the value of \( p \) without having to apply the mapping that yields the ordering or the inverse mapping (see Exercise 3). In fact, for some of the tree access structures (e.g., an MX quadtree \([69, 70, 108]\) which is a variant of the region quadtree, described in Section 1.2, that only merges empty cells) the adjacent cells can be determined by just using the structure of the tree (i.e., no arithmetic operations such as those needed for address computation are performed).

**Exercises**

1. In the text we said that the row order is not stable. This is true if the image grows by doubling or if its resolution is halved. Under what kinds of growth conditions is the row order stable?

2. In the text we said that the spiral order is not stable. This is true if the image grows by doubling or if its resolution is halved. Under what kinds of growth conditions is the spiral order stable?

3. Given a Morton ordering for a \( 2^n \times 2^n \) image, show how to determine the addresses associated with the locations that are adjacent to location \( c \) in the horizontal and vertical directions.

### 1.2 Blocks

An alternative class of representations of the objects and their environment removes the stipulation that the cells that make 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 that are 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 frequently 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. Of course, it goes without saying so, 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 permitted.

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. In essence, the decomposition 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 [104] (also see Section 3). The same techniques are applicable to higher-dimensional data as well.

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.). For example, Figure 6 is an arbitrary block decomposition for the collection of objects and cells given in Figure 1. We have labeled the blocks corresponding to object \( O \) as \( O_i \) and the blocks that are not in any of the objects as \( W_i \) using the suffix \( i \) to distinguish between them in both cases.

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 their various 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 1.

In particular, we 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. The result of the mapping is used 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 block with upper-left corner \( u \) is stored. Since we need to know the size of the block (i.e., the lengths of its sides), we also record this information along with the address in the node of the tree corresponding to the block. An alternative approach is to forgo the ordering and keep track of the upper-left corners of the blocks using one of a number of multidimensional access structures such as a point quadtree, k-d tree, MX quadtree, etc. (as described in Chapter ??). Again, each node in the tree contains the corresponding block’s size and the address where more information an be found about the object associated with the corresponding block.

As in the case of unit-size cells, regardless of which tree access structure is used, 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 and the block decomposition method to whose results it is being applied do not have the joint property that all of the cells in each block appear in consecutive order. In other words, given the cells in the block \( \epsilon \) with minimum and maximum values in the ordering, say \( u \) and \( v \), there exists at least one cell in block \( f \) distinct from \( \epsilon \) 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 (see Exercise 1).

It is also important to recall that this only yields a partial response to query 2 as we also want to determine the remaining blocks that comprise object \( o \). This is done by accessing the blocks adjacent to \( b \) and checking if they are associated with \( o \). If yes, then their adjacent blocks are examined by a recursive application of this process.

The adjacent blocks are found by using the size information about block \( b \) and the location in space of \( b \)'s upper-left corner to calculate points \( p_i \) that are adjacent to the faces (edges in two dimensions) of \( b \) and that must lie in the adjacent blocks. Next, we find the blocks \( b_i \) that contain \( p_i \). Depending on the sizes of \( b_i \) we may have to calculate additional neighboring points. This process can be quite tedious the less regularity exists in the decomposition process. For example, if all the blocks are of the same size, then there is just one adjacent block along each face. Otherwise, the number of adjacent blocks to \( b \) along face \( f \) can be as large as the surface area (length of an edge in two dimensions) of \( f \). In the rest of this section, we discuss various decomposition rules which differ in the number of restrictions that they pose on the size and placement of the blocks thereby simplifying the adjacency determination process.

A very simple decomposition rule is one that partitions a \( d \)-dimensional space having coordinate axes \( x_i \) into \( d \)-dimensional blocks by use of \( h_i \) hyperplanes that are parallel to the hyperplane formed by \( x_i = 0 \) \((1 \leq i \leq d)\). The result is a collection of \( \prod_{i=1}^{d} (h_i + 1) \) blocks. These blocks form a grid of irregular-sized blocks rather than congruent blocks. There is no recursion involved in the decomposition process. For example, Figure 7a is an example block decomposition using hyperplanes parallel to the \( x \) and \( y \) axes for the collection of objects and cells given in Figure 1. We term the resulting decomposition an irregular grid as the partition lines are at arbitrary positions in contrast to a uniform grid [50] where the partition lines are positioned so that all of the resulting grid cells are congruent.

The block decomposition resulting from the use of an irregular grid is handled by an explicit representation in the same way as the arbitrary decomposition. Finding a suitable implicit representation is a bit more complex as we must define an appropriate access structure. Although the blocks are not congruent, we can still impose an array access structure by adding \( d \) access structures termed linear scales. The linear scales indicate the position of the partitioning hyperplanes that are parallel to the hyperplane formed by \( x_i = 0 \) \((1 \leq i \leq d)\). Thus given a location \( l \) in space, say \((a,b)\) in two-dimensional space, the linear scales for the \( x \) and \( y \) coordinate values indicate the column and row, respectively, of the array access structure entry which corresponds

![Figure 7](image-url)
to the block that contains \( l \).

For example, Figure 7b is the array access structure corresponding to the block decomposition in Figure 7a, while Figures 7c and 7d are the linear scales for the \( x \) and \( y \) axes, respectively. In this example, the linear scales are shown as tables (i.e., array access structures). In fact, they can be implemented using tree access structures such as binary search trees, range trees, segment trees, etc. The representation described here is an adaptation for regions of the grid file [90] data structure for points (see Section ?? of Chapter ??).

Our implementation of the access structures for the irregular grid yields a representation that is analogous to an indirect uniform grid in the sense that given a cell at location \( l \) we need to make \( d + 1 \) array-like accesses (analogous to the two memory references involved with indirect addressing in computer instruction formats) to obtain the object \( o \) associated with it instead of just one array access when the grid is uniform (i.e., all the blocks are congruent and cell-sized). The first \( d \) accesses find the identity of the array element (i.e., block \( b \)) that contains \( l \), while the last access determines the object \( o \) associated with \( b \). Once we have found block \( b \), we examine the adjacent blocks to obtain the rest of the cells comprising object \( o \), thereby completing the response to query 2, by employing the same methods as we used for the array access structure for the uniform-sized cells. The only difference is that every time we find a block \( b \) in the array access structure associated with \( o \), we must examine \( b \)'s corresponding entries in the linear scales to determine \( b \)'s size so that we can report the cells that comprise \( b \) as parts of object \( o \).

Perhaps the most widely known decompositions into blocks are those referred to by the general terms quadtree and octree [107, 108]. 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 (see also Chapters ?? and ??), 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 [77] and region octrees [69, 85]. 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 8a is the block decomposition for the region quadtree corresponding to Figure 1. We have labeled the blocks corresponding to object \( A \) as \( Ai \) 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 \( 2^d \times 2^d \) satisfy the property that \( a \mod 2^d = 0 \) and \( b \mod 2^d = 0 \).

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 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 \emph{n} Figure 1 is given by the sets \( A = \{(0,0):2,(0,4):0,(1,4):0,(2,4):0,(3,4):0\}, B = \{(5,1):0,(6,0):1\}, \text{and }C = \{(5,6):0,(5,7):0,(6,6):1\}\), which correspond to blocks \( \{A1,A2,A3,A4,A5\}, \{B1,B2\}, \text{and } \{C1,C2,C3\}\), respectively.

An implementation of a region quadtree that makes use of an implicit representation is quite different. First of all, 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 is 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 9 corresponding to the collection of two-dimensional objects in Figure 1 whose quadtree block decomposition is given in Figure 8a). 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 [30]). 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 as this information can be inferred from knowledge of the size of the underlying space. This is because 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 such as a binary search tree, range tree, $B^+$-tree (e.g., [1]), etc. There are many possible mappings (e.g., Chapter 2 in [107]). 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 number. 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 9) in the order NW, NE, SW, SE. This ordering is also the same as the Morton ordering of the unit-sized cells given in Figure 2c.

As an example of the use of locational codes, consider the collection of objects whose quadtree block decomposition is given in Figure 8a. Assume an $8 \times 8$ space, a depth value ranging between 0 and 3 (i.e., requiring 2 bits), and an interleaving scheme that places the value of the $y$ coordinate in a more significant position than that of the $x$ coordinate. Also, assume an implementation where the empty blocks are given a locational code, although they are frequently excluded in other implementations (e.g., [55]). Figure 8b shows the correspondence of blocks with locational codes.

As mentioned above, the ordering that we used in our mapping is an instance of the Morton ordering. The Morton ordering (as well as the Peano-Hilbert ordering shown in Figure 2c) is particularly attractive for quadtree-like block decompositions because all cells within a quadtree block appear in consecutive positions in the ordering. Alternatively, the two orders exhaust a quadtree block before exiting it. Note, however, that unlike the Peano-Hilbert ordering, the Morton ordering does not traverse the blocks in a spatially contiguous manner (i.e., the result has the shape of the letter 'N' or 'Z' and is also known as N order [138] and Z order [91]). The spatial contiguity property has been shown to have a good effect on the efficiency of range queries [40]. Nevertheless, the Morton ordering is useful due to the ease with which the mapping and the inverse mapping can be constructed as well as its other desirable properties described in Section 1.

An order 5 $B^+$-tree access structure for these locational codes is given in Figure 10. It is important to note...
that although some of these tree-like access structures also form a containment hierarchy (e.g., the range tree and the $B^+$-tree), unlike the tree with fanout $2^d$, the space spanned by the union of the blocks spanned by their nonleaf nodes does not have to consist of similar blocks (e.g., nodes $P$ and $Q$ in Figure 10), nor does it have to be a single block (e.g., node $R$ in Figure 10), nor do the blocks that comprise it have to be contiguous (e.g., node $S$ in Figure 10).

Regardless of the access structure that is used, we determine the object $o$ associated with a cell at location $l$ by finding the block $b$ that covers $l$. Recall that this only yields a partial response to query $2$ as we also want to determine the remaining blocks that comprise object $o$. This is done by accessing the blocks adjacent to $b$ and checking if they are associated with $o$. If yes, then their adjacent blocks are examined by a recursive application of this process. This process is known as neighbor finding [105, 106]. The actual mechanics of this process depend on the nature of the access structure (for a detailed discussion, see Chapter 3 in [107]).

Using an access structure in the form of a tree with fanout $2^d$ means that we can locate the adjacent blocks by just using the structure of the tree. For the sake of this discussion, assume that $d = 2$ and that each node $p$ in the quadtree contains an additional pointer to its immediate ancestor. Here, we restrict ourselves to lateral neighbors (i.e., in the horizontal and vertical directions) that are of size greater than or equal to that of $p$. Now, to locate a lateral neighbor $q$ of $p$ we follow ancestor links until finding the nearest common ancestor of $p$ and $q$. Once the common ancestor is found, we descend along a path that retraces the ascending path with the modification that each step is a reflection of the corresponding prior step about an axis formed by the common boundary between the two nodes $p$ and $q$.

For example, when attempting to locate the eastern neighbor of node $A$ (i.e., node $G$) in the box in Figure 11, node $D$ is their nearest common ancestor, while the eastern edge of the block corresponding to node $A$ is the common boundary between node $A$ and its neighbor $G$. The main idea behind this neighbor finding method can be understood by examining more closely how the nearest common ancestor of node $p$ and its eastern neighboring node $q$ of greater than or equal size are located. In other words, as we ascend links, how do we know that we have encountered the nearest common ancestor and that it is time to descend links? The answer is obtained by observing that the nearest common ancestor has $p$ as one of the eastern-most nodes of one of its western subtrees, and $q$ as one of the western-most nodes of one of its eastern subtrees. Thus as long as an ancestor $r$ of $p$ is in a subtree that is an eastern son (i.e., NE or SE), we must ascend the tree at least one more level before locating the nearest common ancestor. C code for this process, as well as the necessary tables, is given in Figure 11. This technique has been shown to require an average of 4 links to be followed for each neighbor of equal size that is being sought [105]. If the neighbor is larger, then the algorithm is even faster as the descent step stops sooner [105, 111].

Neighbor finding can also be applied when using one of the alternative access structures that are based on finding a mapping from the domain of the blocks to a subset of the integers. In this case, neighbors are obtained by calculating their locational codes and then conducting a search for them in the relevant access structure. The calculation is made using binary arithmetic [107, 118]. The key idea (or trick) is to calculate the coordinate

![Figure 10: Result of applying a $B^+$-tree access structure to the locational codes of the collection of objects whose quadtree block decomposition is given in Figure 8.](image-url)
Finding Lateral Eastern Neighbor of Equal Size in the East Direction

node *LateralNeighbor(node *p, edge i)
    /* Locate an equal-sized edge-neighbor of node p in direction i. */
    {
        node *t;
        if (Adj(i, SonType(p)))
            t = LateralNeighbor(Father(p), i); /* case 1 */
        else t = Father(p); /* case 2 */
        return(Son(t, Reflect(i, SonType(p))));
    }

When \( i = \text{East} \)

Case 1

Case 2

\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{i (direction)} & \textbf{o (quadrant)} & \textbf{i (direction)} & \textbf{o (quadrant)} \\
\hline
N & T & T & F & F & N & T & T & F & F \\
E & F & T & T & E & F & T & T & F & F \\
S & F & F & T & T & S & F & F & T & T \\
W & T & T & T & F & F & T & T & T & F \\
\hline
\end{tabular}

\( \text{Adj}(i, o) \) \( \text{Reflect}(i, o) \)

Figure 11: The process of locating the eastern neighbor of node \( A \) (i.e., \( G \)). (a) Block decomposition; (b) the corresponding tree representation of the search process; (c) C code of the algorithm and the necessary tables.

values of the upper-left corner of the neighboring block without having to decompose the locational code into its individual coordinate axes components and then composing them to form the new locational code.

In some applications we may require finer (i.e., more) partitions along a subset of the dimensions due to factors such as sampling frequency (e.g., when the blocks correspond to aggregates of point data), while needing coarser (i.e., fewer) partitions along the remaining subset of dimensions. This is achieved by loosening the stipulation that the region quadtree results in \( 2^d \) congruent blocks at each subdivision stage, and replacing it by a stipulation that all blocks at the same subdivision stage (i.e., depth) \( i \) are partitioned into \( 2^{c_i} \) (\( 1 \leq c_i \leq d \)) congruent blocks. We use the term \( ATree \) [18] to describe the resulting structure.

For example, Figure 12a is the block decomposition for the \( ATree \) for Figure 1, while Figure 12b is the corresponding tree access structure. We have labeled the blocks corresponding to object \( O_i \) as \( O_i \) and the blocks that are not in any of the objects as \( W_i \) using the suffix \( i \) to distinguish between them in both cases. The nonleaf nodes are labeled with the partition axis or axes which must be the same for all nodes at the same level. Note that all blocks at a particular level of subdivision \( \epsilon \) are partitioned in the same way (i.e., into the same number of congruent blocks along the same axes).

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 imple-

17
Figure 12: (a) Block decomposition for the ATree corresponding to the collection of objects and cells in Figure 1 and (b) the corresponding tree access structure. The nonleaf nodes are labeled with the partition axis or axes.

The block decomposition resulting from the use of a bintree (as well as an ATree and a generalized bintree) can be implemented using an explicit representation in the same way as the region quadtree. Although we presented the bintree in terms of an implicit representation that uses a tree access structure with a fanout value of 2, it can be implemented with other access structures (e.g., $B^+$-tree, etc.) that are applied once a mapping has been found from the domain of the blocks to that of the integers (e.g., locational codes). We do not discuss such techniques here although it should be clear that they are quite easy to apply as long as we know the order in which the decomposition process cycles through the axes in the case of the bintree, or, in the case of the ATree, the identity of the partitions applied at each subdivision stage. For the generalized bintree the situation is a bit more complex as we need to keep track of either the order in which the partitions were applied for each block, or the size of each block.

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

Figure 13: (a) Block decomposition for the bintree corresponding to the collection of objects and cells in Figure 1 and (b) the corresponding tree access structure. The splits alternate between the \( y \) and \( x \) coordinate values with the first split being based on the \( y \) coordinate value.

Figure 14: (a) Block decomposition for the generalized bintree corresponding to the collection of objects and cells in Figure 1 and (b) the corresponding tree access structure. The nonleaf nodes are labeled with the partition axes.

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. For example, the explicit representation (as described in the start of this section in conjunction with the example in Figure 6) needs to record the coordinate values of the upper-left corner of each block as well as the lengths of its sides along the \( d \) coordinate axes instead of just the length of one side as is the case for the region quadtree.
When using the implicit representation we have a number of options. Since the decomposition process is recursive, we can use a tree access structure where at each level of the tree the positions of the partition lines are recorded. An example of such a structure is an adaptation of the point quadtree \([45]\) 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. For example, Figure 15a is the block decomposition for the point quadtree for Figure 1, while Figure 15b is the corresponding tree access structure. Nonleaf nodes are labeled with the location of the partition point. We have labeled the blocks corresponding to object \(O\) as \(O_i\) and the blocks that are not in any of the objects as \(W_i\) using the suffix \(i\) to distinguish between them in both cases.

![Diagram](image)

Figure 15: (a) Block decomposition for the point quadtree corresponding to the collection of objects and cells in Figure 1 and (b) the corresponding tree access structure. The \((x, y)\) coordinate values of the locations of the partition points are indicated next to the relevant nonleaf nodes.

As 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. Recall that in this case we resorted to the bintree which is an access structure with a fanout value of 2. The k-d tree \([16]\) is a similar 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. In fact, the k-d tree is the precursor of the bintree and in its adaptation to regions is defined in a similar manner in the sense that for \(d\)-dimensional data we cycle through the \(d\) axes every \(d\) levels in the k-d tree. The difference is that in the k-d tree, the positions of the partitions are arbitrary, whereas they are a result of a halving process in the case of the bintree.

For example, Figure 16a is the block decomposition for the k-d tree for Figure 1, while Figure 16b is the corresponding tree access structure. Nonleaf nodes are labeled with the name of the axis of the partition and the location of the partition line. We have labeled the blocks corresponding to object \(O\) as \(O_i\) and the blocks that are not in any of the objects as \(W_i\) using the suffix \(i\) to distinguish between them in both cases.

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. For example, Figure 17a is the block decomposition for the generalized k-d tree for Figure 1, while Figure 17b is the corresponding tree access structure. Nonleaf nodes are labeled with the name of the axis of the partition and the location of the partition line. We have labeled the blocks corresponding to object \(O\) as \(O_i\) and the blocks that are not in any of the objects as \(W_i\) using the suffix \(i\) to
Figure 16: (a) Block decomposition for the k-d tree corresponding to the collection of objects and cells in Figure 1 and (b) the corresponding tree access structure. The splits alternate between the y and x coordinate values with the first split being based on the y coordinate value. The locations of the splits are indicated next to the relevant nonleaf nodes.

distinguish between them in both cases.

Figure 17: (a) Block decomposition for the generalized k-d tree corresponding to the collection of objects and cells in Figure 1 and (b) the corresponding tree access structure. The nonleaf nodes are labeled with the partition axes and the partition values.

The generalized k-d tree is really an adaptation to regions of the adaptive k-d tree [52] and the dynamic pseudo k-d tree [94], as well as the disk-based k-d-B-tree [100] (see Section 1.4.3) the LSD tree [64]. The similarity lies in the fact that they also permit the partitions to take place on the various axes at an arbitrary
order and at arbitrary positions. Although all of these representations were originally developed for points, they can be also be easily adapted to handle regions.

The generalized k-d tree can also be regarded as a special case of the BSP tree (denoting Binary Space Partitioning) [53, 54] discussed in Section ?? of Chapter ?? of this book. 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 [31] 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.

Figure 18a is the block decomposition for the puzzletree for Figure 1, while Figure 18b is the corresponding tree access structure. Notice that the puzzletree was created by compressing the successive initial partitions on \( x = 4 \) and \( x = 5 \) at depth 0 and 1, respectively, and likewise for the successive partitions on \( y = 6 \) and \( y = 2 \) at depth 2 and 3, respectively, in Figure 17. Nonleaf nodes are labeled with the name of the axis of the partition and the location of the partition line. We have labeled the blocks corresponding to object O as \( O_i \) and the blocks that are not in any of the objects as \( W_i \) using the suffix \( i \) to distinguish between them in both cases.

The fact that the puzzletree compresses all successive partitions on the same dimension means that for \( d = 2 \), the partitions cycle through the different dimensions so that a different dimension is tested at each level of the decomposition. Note that for \( d \geq 3 \), a requirement that the partitions cycle through the various dimensions before being permitted to repeat may prevent us from being able to use certain more perceptually appealing block combinations. For example, consider the three-dimensional four-legged chair in Figure 19a whose corresponding puzzletree has the block decomposition given in Figure 19b and a tree access structure given in Figure 19c. Here we see the impossibility of cycling through all of the dimensions in an alternating manner.

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 20a and 20b are the block decompositions resulting from the use of a bintree and a puzzletree, respectively, for this scene, while Figure 20c is the tree access structure corresponding to the puzzletree in Figure 20b. 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.
Constructing the puzzletree for a particular scene is a tricky matter. The problem is that on the one hand we want to decompose the scene into meaningful blocks. This requires some a priori high level knowledge about the semantic meaning of the scene’s constituent parts (e.g., a chair consists of four legs, a seat, and a back). On the other hand, we usually don’t usually have this semantic information readily available when processing the scene. In other words, ideally, we would like the puzzletree construction to be omniscient. As this is impossible we must resort to the use of heuristics about inferring meaningful two-dimensional blocks from easily derivable measures of one-dimensional block aggregations (see Exercise 6).

The puzzletree can be shown to require no more nodes than a region quadtree and often considerably less nodes. Of course, the construction process of the puzzletree is more complex than that of the region quadtree. In particular, the puzzletree is not unique and relies on heuristics to identify the dominant parts so that a perceptually meaningful decomposition is obtained when we start with a collection of cells. The drawback of the puzzletree (and other representations such as the point quadtree, k-d tree, and generalized k-d tree that do not use regular decomposition) is that since the placement of the partition lines is arbitrary, a height-balanced decomposition (i.e., one that does not require too much space) will require that all the data be examined prior to building the structure. Otherwise, the structure can be skewed in a manner analogous to the worst-case behavior of a binary search tree, which is a linear list.

Of course, access structures aside from space partitions (e.g., multidimensional trees or trees that partition the space one dimension at a time) could be used as well. In this case, we again need to find a mapping from the domain of the blocks to a subset of the integers (i.e., to one dimension), and then apply one of the familiar access structures (e.g., a binary search tree, range tree, B⁺-tree, etc.). Since the sizes of the blocks are arbitrary, it does not make sense to use a mapping that concatenates the sizes to the result of interleaving the binary representations of the coordinate values of the upper-left corner of each block as was done for the regular decompositions. Instead, we employ the same techniques that we used for an arbitrary decomposition as described at the start of this section — that is, we apply one of the orderings shown in Figure 2 to obtain a mapping from the coordinate values of the upper-left corner of each block to the integers.

R-trees and pyramids...hierarchies of objects and cells. Enable answering queries 2 and 1, respectively, without exhaustive search (see point 11 at end). In essence, the hierarchy enables us to answer the queries in log(side of space) and log(number of objects) time. Distinction is the same as that between object space and image space.

Exercises

1. Given an arbitrary decomposition where each block is represented by the position of its upper-left corner in a Morton ordering which is then used as an index in a binary search tree, show how to find the object o that is associated with a cell in location l.

2. Suppose that you have an environment of width 32 and length 64. What are the permissible sizes of the blocks after the first region quadtree subdivision step?

3. Suppose that you have an environment of width 16 and length 24. Can the first region quadtree subdivision step result in 2 blocks of size $8 \times 8$ and two blocks of size $8 \times 16$? If not, explain why.

4. Neighbor finding algorithms for diagonal directions, other representations, etc.

5. Why isn’t the generalized bintree a special case of the ATree?

6. The key to building a puzzletree is identifying the blocks. Below, we describe a variation of a technique suggested in [31] to choose between performing a vertical or a horizontal partition at each stage. We illustrate our discussion with the collection of cells and objects in Figure 21a. Decompose each row $j$, having length $L_j$, into maximal one-dimensional identically-valued blocks (i.e., each block is associated with the same object), and calculate the value $H_j = (\sum_i x_i^j) / L_j$ where $x_i$ denotes the width of the $i^{th}$ block in row $j$. Perform a similar calculation for the columns — that is, $V_j = (\sum_i y_i^j) / L_j$ where $y_i$ denotes the length of the $i^{th}$ block in column $j$ and $L$ is the width of column $j$. Clearly, a value of
$H_j = 1$ ($V_j = 1$) indicates that the row (column) consists of one solid block. Once these values have been computed, there are a number of ways to proceed, none of which are guaranteed to always yield the minimum number of leaf nodes. We choose the partitioning axis as the one that has the maximum $H_j$ or $V_j$ value (e.g., into columns in Figure 21a). Ties are broken in an arbitrary manner. Once we have chosen the partitioning axis, we partition so that adjacent columns (rows) with the maximum value $V_j$ ($H_j$) and identical values in each cell form a block. This process is termed the *linear homogeneity heuristic*. Figure 21a is the block decomposition of the puzzletree resulting from its application to a collection of cells and objects whose corresponding tree access structure is given in Figure 21b. Once the first level of partition has been chosen, the rest are chosen by adhering to the heuristic whenever possible. If successive application of this heuristic yields a partition on the same dimension, then the partitions are compressed. Is the puzzletree in Figure 18 constructed according to these rules?

![Diagram](image)

**Figure 21:** (a) Block decomposition of a puzzletree for a collection of cells and objects constructed according to the linear homogeneity heuristic, and (b) the corresponding tree access structure.

7. Devise an algorithm to construct a puzzletree that makes use of the linear homogeneity heuristic described in Exercise 6.

8. Once a puzzletree has been generated, it is often possible to transform it into another puzzletree. For example, let $a$, $b$, and $c$ be three node positions in a puzzletree which are occupied by nonleaf nodes such that, without loss of generality, $b$ and $c$ test the $x$ and $y$ coordinate values, respectively, $b$ has degree 2, $c$ is the only son of $b$ that is a nonleaf node, all the sons of $c$ are leaf nodes, and $b$ is a son of $a$. Therefore, by the definition of a puzzletree, the nonleaf node at position $a$ tests the $y$ coordinate value. One possible transformation interchanges the tests at positions $b$ and $c$ so that $c$ becomes a test on the $x$ coordinate value and $b$ becomes a test on the $y$ coordinate value. Once the interchange has occurred, we have two successive tests on the $y$ coordinate value at positions $a$ and $b$, and we can compress these two tests into one test. For example, referring to the tree access structure in Figure 18b, this transformation is applicable to the partitions on $x = 6$ and $y = 1$ at depth 2 and 3, respectively, in the rightmost subtree of the root. What properties must node $c$ satisfy (i.e., nature of the objects associated with it) so that this transformation does not result in an increase in the number of leaf nodes? Can it ever lead to a decrease in the number of leaf nodes?

9. At times, we can apply a technique, which we term *common partition elimination* (analogous to common subexpression elimination in compiler code optimization [3]), to yield more optimal partitions in terms of the number of blocks. This is the case when two sibling nonleaf nodes have at least one common partition. For example, consider the block decomposition and tree access structure in Figure 22. The results of the partition at the root that yields the regions $5 \leq x < 6$ and $6 \leq x < 8$ are both subdivided further at $y = 2$ and $y = 6$. Hence, they share the common subsequent partitions $0 \leq y < 2$, $0 \leq y < 2$. 

25
2 \leq y < 6, \text{ and } 6 \leq y < 8. \text{ The regions corresponding to these subsequent partitions can be merged if they are identically-valued (i.e., their blocks are associated with the same object or are not associated with any object). For example, the pair of blocks labeled W5 and W6 and the pair labeled C2 and C3 in Figure 22 can be merged into larger blocks labeled W4 and C1, respectively, as shown in Figure 18. Notice that the effect of the common partition elimination is the removal of the partition step at x = 6 from the root of the tree access structure, thereby reducing the size of the tree. The reason it is still present at depth 2 in Figure 18 is that the objects associated with blocks W3 and B1 are not the same as those associated with block B2. Give an algorithm COMPRESS to implement common partition elimination. In order to ease the expression of your solution, assume a recursive list representation [31] for the tree access structure where each node \( t \) in the tree is a list of three items \( (A, P, S) \) where \( A \) is the split axis, \( P \) is a list containing the split locations, and \( S \) is a list containing the nodes or values corresponding to the sons of \( t \). We refer to these items by the field names \( A, P, \) and \( S \). For example, the puzzletree corresponding to Figure 22 is \( (x \ (4 \ 5 \ 6) \ ((y \ (6) \ (A1 \ W1))) \ W2 \ (y \ (1 \ 2 \ 6) \ (W3 \ B1 \ W5 \ C2)) \ (y \ (2 \ 6) \ (B2 \ W6 \ C3))) \). The input to procedure COMPRESS are the two sibling nodes being merged, the split axis separating them, and the split axis value.

Figure 22: (a) An alternative block decomposition for a puzzletree corresponding to the collection of objects and cells in Figure 1 and (b) the corresponding tree access structure. The nonleaf nodes are labeled with the partition axes and the partition values.

10. Devise an algorithm COMPRESS_TREE for the technique described in Exercise 9 to eliminate a partition for an implementation of a puzzletree that uses a tree access structure. The input to the algorithm are the axis \( u \) being partitioned, the partition value \( p/0 \), and pointers to the two subtrees \( a \) and \( b \) corresponding to the two puzzletree nodes that are being merged. Assume that each nonleaf node has four fields: TEST, NUM_SONS, SONS[1:NUM_SONS], and VALUE[1:NUM_SONS] indicating the coordinate being tested, the number of sons, pointers to the sons, and the values of the coordinate being tested in increasing order. Note that in order to simplify the algorithms, V[NUM_SONS] is always initialized to \( \infty \). Each leaf node has a field called OBJECT which contains a pointer to the object associated with the block corresponding to the leaf node.

11. An alternative means of achieving common partition elimination to the one described in Exercise 9 is one that eliminates a partition step as well as introduces new partitions wherever the common partitions are not present [31]. For example, consider the block decomposition and tree access structure in Figure 22. In particular, focus on the partition at the root that yields the regions \( 5 \leq x < 6 \) and \( 6 \leq x < 8 \). They are both subdivided further at \( y = 2 \) and \( y = 6 \). Hence, they share the common partitions \( 0 \leq y < 2 \), \( 2 \leq y < 6 \), and \( 6 \leq y < 8 \). In this case, we modify the technique discussed in Exercise 9 to create a new partition wherever a partition is not present so that we partition the region \( 0 \leq y < 2 \) into \( 0 \leq y < 1 \) and \( 1 \leq y < 2 \). The result is shown in Figure 23. Give an algorithm called


REPARTITION to implement this approach. Assume the same recursive list representation for the nodes that was used in Exercise 9. Again, the inputs to procedure COMPRESS are the two sibling nodes being merged, the split axis separating them, and the split axis value.

12. Give an example of a puzzletree, say \( t \), for which common partition elimination using the method of Exercise 9 results in fewer leaf nodes while common partition elimination using the method of Exercise 11 results in more nodes. The block decomposition in \( t \) is such that all the sons of the two nodes in which the common partitions are found are leaf nodes. Also, \( t \) is constructed in accordance with the linear homogeneity heuristic.

13. Assume a puzzletree block decomposition where two sibling nodes \( a \) and \( b \) have common partitions, and that all the sons of \( a \) and \( b \) are leaf nodes. Prove that the result of common partition elimination using the method of Exercise 9 never requires more leaf nodes than the method of Exercise 11. In this exercise, the puzzletree need not necessarily be constructed in accordance with the linear homogeneity heuristic.

14. Assume a puzzletree block decomposition constructed in accordance with the linear homogeneity heuristic, and that two sibling nodes \( a \) and \( b \) have common partitions. Can you prove that the result of common partition elimination using the method of Exercise 9 never requires more leaf nodes than the method of Exercise 11?

15. Assume a puzzletree block decomposition that is not necessarily constructed in accordance with the linear homogeneity heuristic, and that two sibling nodes \( a \) and \( b \) have common partitions. Can you prove that the result of common partition elimination using the method of Exercise 9 never requires more leaf nodes than the method of Exercise 11?

1.3 Nonorthogonal Objects

In Section 1.1 we assumed that each unit-sized cell was contained entirely in one or more objects. In other words, no cell \( c \) was partially contained in one object \( o_1 \) and partially contained in another object \( o_2 \), or, even more generally, no cell \( c \) was partially contained in an object \( o_1 \) while the rest of \( c \) was not in any other object. Similarly, in Section 1.2 we stipulated that the block decomposition rules all required that each cell in each block belong to the same object or objects, or that the cells do not belong to any objects. Thus the situation in
One of the consequences of these restrictions is that we could not represent objects whose boundaries did not coincide with the boundaries of the underlying blocks. A more general statement of this consequence is that we could not represent polygonal objects whose sides were not parallel to the coordinate axes (i.e., nonorthogonal objects)\(^3\). The problem was that for such polygonal objects we could not, in general, obtain a decomposition so that each cell or block was completely contained in a single object or is not in any object. In this section, we loosen these restrictions. One result is that the boundaries of the objects can be of arbitrary orientation and order (i.e., linear, quadratic, cubic, etc.). Another result is that the amount of overlap between objects need not be such that the objects overlap a cell in its entirety.

In the case of the explicit representation, loosening these restrictions has the effect of freeing us from the need to represent the individual objects as a collection of unit-sized cells or rectangular blocks. Instead, collections of other shapes such as triangles, trapezoids, convex polygons, etc. (e.g., [80, 19]) could be used. The rationale for using these shapes is that they can capture the geometry of the object more accurately than the rectangular blocks, as well as simplify the computation of point-in-object tests in comparison to performing the test on the entire object. We do not discuss the explicit representation further in this section.

In the case of the implicit representation, loosening these restrictions enables the decoupling of the space partition of the underlying space induced by the decomposition process from the space partition induced by the objects. This means that in addition to keeping track of the identity of all objects of which a particular cell or block is a member (partially or totally), we must also keep track of its geometric description. This enables us to respond to query 2. The problem is that without the geometric description, given a specific location \(l\) in space, finding the cell \(c\) or block \(b\) in which it is contained may not be enough to determine the identity of the object (or objects) of which \(l\) is a member as well as the rest of the constituent cells of the object (or objects). Therefore, once we have located \(c\) or \(b\) we must check the geometric description of the objects associated with \(c\) or \(b\) to see which ones contain \(l\). By recording a pointer to the geometric description of the objects with each cell or block rather than the entire geometric description we save much storage as the geometric description is only recorded once. Notice that once we have located the object (or objects) containing \(l\), determining the remaining constituent cells of these objects is done in the same manner as described earlier (i.e., by use of connected component labeling).

The block decompositions described in Section 1.2 were all based on a decomposition rule that required that either all the cells in block \(b\) be elements of the same object (or objects) or all the cells in \(b\) are in none of the objects. In other words, a block had to be completely contained in an object or objects. This 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 a way to stop the decomposition process. A natural way is to place a restriction on the number of objects in which a block or part of the block is a member. This rule can be formally stated as stipulating that a block is decomposed whenever it is a member of more than \(T\) (\(T \geq 1\)) objects\(^4\). Such a rule is known as a bucket-like decomposition rule. 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 1.2 which required that either all the cells in \(b\) be 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 the rest of this section we assume two-dimensional data\(^5\). In particular, we focus on how to deal with collections of arbitrary polygonal objects thereby, of course, also including rectangles which are common in applications such as VLSI and CAD. In the case of polygons,

\(^3\)To be more precise, we could not represent nonrectilinear polygonal objects. Notice that strictly speaking our restrictions did not permit the representation of all orthogonal polygonal objects as such polygonal objects are only required to have boundaries that are orthogonal to each other while not requiring the boundaries to be parallel to the coordinate axes.

\(^4\)Notice that the decomposition rule is not one that that decomposes a block if it contains more than \(T\) objects. Such a condition may 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 objects in their entirety.

\(^5\)Our discussion is also applicable to data of arbitrary dimensionality.
the specific variant that is used also depends on whether the polygons themselves are the objects being de-
composed, 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 2.1).

It is straightforward to formulate bucket-like variants of the different block decomposition rules described
in Section 1.2. Without loss of generality, in the rest of this section we assume a quadtree-like regular decom-
position rule which decomposes the underlying space into four congruent blocks at each stage (i.e., level) of
decomposition. As in Section 1.2, the results of successive stages form a containment hierarchy. Again, as in
the previous section, once the decomposition process has terminated, the result is a set of disjoint blocks. Of
course, other decomposition rules such as those that, to name a few, do not require the blocks at a particular
stage to be congruent and similar at successive stages (e.g., a point quadtree), or do not subdivide into just two
parts (e.g., a k-d tree or bintree), or do not require that the blocks be orthogonal (e.g., a BSP tree) could also
be used. We do not discuss such decomposition rules in this section.

The bucket-like decomposition rule described above works fine 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. Overlapping polygons arise in a number of defense analysis problems. For example, trafficability (also
known as mobility) polygons are often overlapping [5]. In particular, frequently it is desired to find a path
through a set of overlapping (thereby connected) polygons. Overlapping polygons also arise in VLSI and
CAD applications such as building plans. Adjacent polygons are discussed in greater detail a bit later in this
section.

As an example of a block configuration which 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 instance of time when \( b \) contains a subset of \( T \) 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 of the blocks resulting from the decomposition
will be a part of \( T \) or fewer polygons. For example, assuming \( T = 2 \), the block containing the three polygons
in Figure 24a is not split upon insertion of the third polygon. In this case, the order of inserting the polygons
is immaterial.

![Figure 24: 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 condition or (b) the mutual intersection condition. (c) B is split when using the touching condition if the polygons are inserted in the order 1, 2, and 3 while B is not split if the polygons are inserted in the order 1, 3, and 2.](image)

A decomposition rule that prevents more decomposition than complete containment is one that requires the
mutual intersection of all of the polygons in \( b \) including the polygon \( p \) being inserted to be nonempty in order
to prevent decomposition. The mutual intersection condition is motivated by the desire to be able to deal with
the situation that 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 24b are mutually intersecting and thus the block is not split regardless of the order in which

\[ \text{Footnote 6: The mutual intersection condition is weaker than the complete containment condition in the sense that complete containment implies mutual intersection.} \]
the polygons are inserted into it. In fact, the order in which the polygons are inserted is always immaterial when using the mutual intersection condition.

A decomposition rule that prevents even more decomposition than mutual intersection, and the one we suggest using, 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 even just touch at their boundaries). For example, assuming \( T = 2 \), inserting polygons 1, 2, and 3 in this order in Figure 24c avoids splitting the block. On the other hand, inserting the polygons in the order 1, 3, and 2 in Figure 24c results in the block being split. Thus we see that the result of using the touching condition may be sensitive to the order in which the polygons are inserted. Notice that the touching condition would also not split the blocks in Figures 24a and 24b regardless of the order in which the polygons were inserted. This shows that the order in which the polygons are inserted is not always important for the touching condition.

It is interesting to observe that using a decomposition rule based on touching often prevents useful decompositions from taking place. For example, in Figure 24c it may be better to decompose the block. Although the mutual intersection condition is really a better condition than the touching condition in the sense of allowing more useful decomposition to take place, the use of mutual intersection is not suggested because its computation for the general case of polygons of arbitrary shape is expensive.

Perhaps the most common 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 example, a map of the US is a polygonal map where the polygons correspond to states and the edges of the polygons correspond to state lines. Similarly, a map of the world is a polygonal map where the polygons correspond to countries and the edges of the polygons correspond to borders of countries. Polygonal maps arise in many defense applications. For example, they are used for crop coverages, soil types, land usage, etc. In addition, they are also used to represent the projections of elevation models. For example, given a set of elevations such as a DTM (Digital Terrain Model) or a DEM (Digital Elevation Model), we often fit an interpolating surface consisting of triangles on it which is called a triangulation. The projection of the triangulation on the plane is also polygonal map. Thus the representation of a polygonal map is an important issue.

In the case of adjacent polygons, 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 [88, 89] known as the PMR quadtree (see also Section 2.1 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.

An alternative solution to that provided by the PMR polygon quadtree is to introduce an additional condition that halts the decomposition whenever the block \( b \) contains a single vertex \( v \) [98]. The motivation for this is rule is the fact that when a block \( b \) contains just one vertex \( v \), it is often the case that all of the edges in \( b \) meet at \( v \). Note that this rule does not preclude a block \( b \) from containing more than one vertex provided that \( b \) does not contain more than \( T \) polygons. In fact, this is quite a common occurrence since each polygon consists of at least three vertices! We term the result a vertex bucket polygon quadtree.

Once the space has been partitioned into blocks, we need to consider the representations of the polygons that make up each block. There are a number of 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 [98]. This

\[ ^{5} \text{The touching condition is weaker than the mutual intersection condition in the sense that mutual intersection implies touching.} \]
is motivated by the fact the 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 the union of its convex parts, and the second is as a difference of convex regions.

![Decomposition Examples](attachment:image.png)

Figure 25: (a) Example block consisting of two regions A and B. (b) The result of decomposing B into three convex regions. (c) The convex polygon C that must be subtracted from the convex hull of B to obtain B. (d) The convex difference trees (CDTs) corresponding to the polygons of the entire block.

Subdivision into a union of convex regions can be achieved easily by use of methods such as a triangulation. For example, polygon B in Figure 25a is decomposed into three convex regions (i.e., triangles) in Figure 25b. In fact, optimal convex decompositions can also be achieved by introducing new points (termed Steiner points as discussed in Section ?? of Chapter ??) [23] as well as without new points [59]. The disadvantage of a subdivision into a union of convex regions is that the number of resulting polygons can increase dramatically.

When the polygons are represented as a difference of convex regions, the number of resulting polygons is considerably smaller, but still larger than just leaving them alone. This is achieved by using the convex difference tree (CDT) representation for simple polygons [133]. In this case, a simple polygon is represented as a tree. The root of the tree contains the convex hull of the polygon. There is one son for each connected component of the difference between the original polygon and the convex hull. The sons are also convex difference trees which means that they contain the convex hull of their polygons. Thus we see that the representation process is recursive with the decomposition stopping once we get a connected component difference which is convex. As a simple example, the block in Figure 25a contains two polygons A and B. Since B is not convex, its CDT is formed by subtracting convex polygon C as shown in Figure 25c from the convex hull of B. The two CDTs corresponding to the polygons of the entire block are given in Figure 25d. The broken lines in Figures 25c and 25d show the convex hull of B.

The W-like shape in Figure 26a is a more complicated example of a CDT. The broken lines in Figure 26b show the various components in the CDT for Figure 26a, while Figure 26c is the actual tree. In particular, the root of Figure 26c corresponds to the convex hull of the W-like shape, while its left son corresponds to the convex hull of its upper connected region A, and its right son corresponds to the lower connected region B which is a triangle and hence is not decomposed further. The left son of the root has one son corresponding to its lower connected component C.

When the polygons in a block are represented by their CDT, it has been suggested [98] that the block be further subdivided on the basis of whether the total number of nodes in the CDT exceeds the splitting threshold $T$ of the underlying polygon decomposition rule (i.e., the number of polygons in which each block is permitted to be a member unless it possibly contains a vertex). Each time a block is decomposed, the CDT’s of all polygons in the new sub-blocks are calculated and all sub-blocks for which the total number of CDT nodes exceeds the splitting threshold are recursively subdivided.

Regardless of the value of $T$ that is used, when all of the polygons meet at a single vertex, the blocks that contain such vertices may be a part of many polygons thereby possibly complicating algorithms such as point location [98]. One way to reduce this number, and, more importantly, the number of polygons that can be a part of each block, is to modify the polygonal map $m$ by preprocessing it and applying a technique termed vertex enlargement to each vertex $v$ at which more than $T$ ($T \geq 3$) polygons are incident [98]. Note that we
Figure 26: (a) Example W-like shape, (b) the components that make up its convex different tree (CDT), and (c) the actual tree.

discuss this technique not so much for its utility, which is unclear, but for the way in which it reduces the complexity of the map in a systematic manner.

The basic operation in vertex enlargement is checking if each of the angles between the edges that meet at \( v \) in clockwise order is less than 180° (i.e., they form what is termed a convex corner). If this is the case, then \( v \) is replaced by a new small polygon at which all of these edges are incident (e.g., the polygonal map in Figure 27a is transformed to Figure 27c by the removal of vertex \( V \)). If this is not the case, then there is one angle that is greater than or equal to 180° and the new polygon contains \( v \), but now the degree of \( v \) has been reduced to two (e.g., the polygonal map of Figure 27d is transformed to Figure 27f by the replacement of the four polygons incident at \( V \) by just two polygons). As long as the new vertices of the newly-created polygons are sufficiently close to \( v \), the new edges of the newly-created polygons will not cross existing edges and the result will be a legal polygonal map. This process is described more precisely and formally as follows, assuming that \( v \) has degree \( d \):

1. For each polygon \( p \) in \( m \) with \( e \) edges such that successive adjacent edges \( e_1 \) and \( e_2 \) in clockwise order that are incident at \( v \) for which \( e_1 \) and \( e_2 \) form a convex corner: choose two points \( p_1 \) and \( p_2 \) on edges \( e_1 \) and \( e_2 \), respectively, so that the edge \( f \) between them does not cross any of the existing edges of the polygonal map. Replace \( p \) by a pair of polygons \( p' \) and \( p'' \). \( p' \) contains \( e + 1 \) edges and is formed by replacing the portions of the edges between \( p_1 \) and \( v \) and \( p_2 \) and \( v \) in \( p \) by a new edge \( f \) (e.g., polygon JKCD in Figure 27b). \( p'' \) is a triangle polygon formed by \( f \) and the edges between \( p_1 \) and \( v \) and \( p_2 \) and \( v \) (e.g., polygon VJK in Figure 27b).

2. For each polygon \( p \) with edges \( e_1 \) and \( e_2 \) incident at \( v \) such that there does not exist another polygon \( q \) with either edge \( e_1 \) or \( e_2 \) that does not have a convex corner at \( v \): remove edges \( e_1 \) and \( e_2 \) (e.g., edges VI and VJ in polygon VJK in Figure 27b whose result is given in Figure 27c whereas this condition is not satisfied for the same edges and polygon in Figure 27e).

3. If \( v \) is isolated (i.e., it has degree 0), then mark the polygon \( q \) of \( d \) edges in which \( v \) is contained as having an enlarged vertex, and remove \( v \) (e.g., vertex \( V \) in Figure 27c). Otherwise, \( v \) is a vertex of just two polygons one of which, say \( q \), has \( d + 1 \) edges where all pairs of adjacent edges form a convex angle, and both \( v \) and \( q \) are marked as corresponding to an enlarged vertex (e.g., vertex \( V \) and polygon VIJKL in Figure 27f).

Exercises

1. Why do we need to check for angles greater than 180° when checking for vertex enlargement?
2. Suppose that you apply vertex enlargement to a polygonal map. What is the minimum value of the maximum degree of any vertex in the resulting map and how would you achieve this situation?

*The rest of this discussion can be skipped on an initial reading.*
Figure 27: (a-c): Example of vertex enlargement where every pair of successive edges in clockwise order that are adjacent at vertex V form a convex corner. (a) Initial polygonal map; (b) intermediate result after adding edges around V; (c) final result after V has been removed. (d-f): Example of vertex enlargement where every pair of successive edges in clockwise order that are adjacent at vertex V form a convex corner with the exception of edges VE and VB. (d) Initial polygonal map; (e) intermediate result after adding edges around V; (f) final result after all but two of the edges incident at V have been removed.

3. Consider a polygonal map. Suppose that you apply vertex enlargement in conjunction with a bucketing decomposition rule that decomposes a block whenever it is a part of more than $T$ polygons. What is a reasonable value of $T$?

4. Suppose that you use a vertex bucket polygon quadtree (i.e., decomposition halts whenever a block is a part of no more than $T$ polygons or contains just one vertex). Can you guarantee that using such a rule, the decomposition process for $T = 1$ will always halt? No vertex enlargement has been applied.

5. Suppose that you preprocess every polygonal map by applying vertex enlargement to all vertices that have degree greater than $T$ ($T \geq 3$). Does this mean that we can now dispense with the stipulation that decomposition halts whenever a block contains just one vertex? In other words, does a decomposition rule that stipulates that decomposition halts whenever a block is a part of no more than $T$ polygons yield the same decomposition as one that also halts the decomposition when the block contains just one vertex?

6. Suppose that a polygonal map has been decomposed to form a vertex bucket polygon quadtree. Next, recursively subdivide the block if its CDT contains more than $T$ nodes. Prove that decomposition of the block based on the number of CDT nodes cannot lead to a reduction in the number of polygons in a node thereby making it impossible for a situation to arise that merging should be applied.

1.4 Hierarchical Interior-based Representations

Assuming the presence of an access structure, the implicit interior-based representations described in Sections 1 and 1.2 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 interior-based 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 interior-based 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) interior-based representations, while the latter is applicable to explicit (i.e., object-based) interior-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 [46], respectively.

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.

On the other hand, in the 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 use of the bounding box approximation has the drawback that the bounding boxes at a given level in the hierarchy are not necessarily disjoint. This can be overcome by decomposing the bounding boxes so that disjointness holds. 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.

It is very important to 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 [46].

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.

The rest of this section is organized as follows. We first show how to modify the implicit interior-based representations which are image-based, and then address the explicit interior-based representations which are object-based. We conclude with ...

### 1.4.1 Image-based Hierarchical Interior-based Representations

Our goal here is to be able take an object \( o \) as input and return the cells that it occupies (query 1) when using a representation that stores with each cell the identity of the objects of which it is a member. The most natural hierarchy that can be imposed on the cells to enable us to answer this query is one that aggregates every \( q \) cells regardless of the values associated with them into larger congruent blocks (unlike the aggregation of identically-valued cells into multi-dimensional blocks as in the region quadtree). This process is repeated recursively so that groups of \( q \) blocks are repeatedly aggregated into one block until there is just one block left. The value associated with the block \( b \) is the union of the names (i.e., object identifiers) of the objects associated with the cells or blocks that comprise block \( b \). The identity of the cells and blocks that are aggregated depends, in part, on how the collection of the cells is represented. For example, assuming a two-dimensional space, if the cells are represented as one long list consisting of the cells of the first row, followed by those of the second row, etc., then one possible aggregation combines every successive \( q \) cells. In this case, the blocks are really one-dimensional entities.

The process that we have just outlined can be described more formally as follows. We make the following assumptions:

- The blocks are rectangular with sides parallel to the coordinate axes.
- Each block contains \( q \) cells or \( q \) blocks so that, assuming \( d \) dimensions, \( q = \prod_{j=1}^{d} r_j \) where the block has width \( r_j \) for dimension \( j \) \( (1 \leq j \leq d) \) measured in cells or blocks depending on the level in the hierarchy in which the block is found.
- All blocks at a particular level in the hierarchy are congruent.
- There are \( M \) cells in the space and let \( n \) be the smallest power of \( q \) such \( q^n \geq M \).
- The space can be enlarged by adding \( L \) empty cells so that \( q^n = M + L \) and that each side of the space along dimension \( j \) is of width \( r_j^n \).

The hierarchy consists of the set of sets \( \{ C_i \} \) \( (0 \leq i \leq n) \) where \( C_i \) corresponds to the original collection of cells having \( M + L \) elements, \( C_{n-1} \) contains \( (M + L)/q \) elements corresponding to the result of the initial aggregation of \( q \) cells into \( (M + L)/q \) congruent blocks, and \( C_0 \) is a set consisting of just one element corresponding to a block of size \( M + L \). Each element \( e \) of \( C_i \) \( (0 \leq i \leq n - 1) \) is a congruent block whose value is the union of the values (i.e., sets of object identifiers) associated with the blocks of the \( q \) elements of \( C_{i+1} \). The value of each element of \( C_n \) is the object identifier(s) corresponding to the object(s) of which its cell is a member.

The resulting hierarchy is known as a **cell pyramid** [131] 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, although similar in shape. It is important to distinguish the cell 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 nonrectangular shapes (most notably, triangles in two dimensions [15, 108]).

---

\(^9\) Actually, the qualifier *cell* is rarely used. However, we use it here to avoid confusion with other variants of the pyramid which are based on a hierarchy of objects rather than cells as discussed in Section 1.4.2.
It is quite difficult to use the cell pyramid, in the form that we have described, to respond to query 1 and to the complete query 2 (i.e., to obtain all of the contiguous cells that make up the object associated with the query location) due to the absence of an access structure. This can be remedied by implementing a set of arrays $A_i$ in a one-to-one correspondence to $C_i \ (0 \leq i \leq n)$ where $A_i$ is a $d$-dimensional array of side length $r_j^i$ for dimension $j \ (1 \leq j \leq d)$. Each of the elements of $A_i$ corresponds to a $d$-dimensional block of side length $r_j^{n-i}$ for dimension $j \ (1 \leq j \leq d)$ assuming a total space of side length $r_j^n$. The result is a stack of arrays $A_i$, termed an array pyramid, which serves as an access structure to to collections $C_i \ (0 \leq i \leq n)$. The array pyramid is an instance of an implicit interior-based representation with an array access structure. Of course, other representations are possible through the use of alternative access structures (e.g., different types of trees).

We illustrate the array pyramid for two dimensions with $r_1 = 2$ and $r_2 = 2$. Assume that the space in which the original collection of cells is found is of size $2^n \times 2^n$. Let $C_n$ correspond to the original collection of cells. The hierarchy of arrays consists of the sequence $A_i \ (0 \leq i \leq n)$ so that elements of $A_i$ access the corresponding elements in $C_i$. We obtain $C_{n-1}$ by forming an array of size $2^{n-1} \times 2^{n-1}$ with $2^{2n-2}$ elements so that each element $e$ in $C_{n-1}$ corresponds to a $2 \times 2$ square consisting of 4 elements (i.e., cells) in $C_n$ and has a value consisting of the union of the names (i.e., labels) of the objects that are associated with these 4 cells. This process is applied recursively to form $C_i \ (0 \leq i \leq n-1)$ where $C_0$ is a collection consisting of just one element whose value is the set of names of all the objects associated with at least one cell. The arrays are assumed to be stored in memory using sequential allocation with conventional orderings (e.g., lexicographically), and are accessed by use of the $d$-dimensional coordinate values of the cells. For example, Figure 28 is the array pyramid for the collection of objects in Figure 1.

![Figure 28: Array pyramid for the collection of objects and cells in Figure 1. (a) Array $A_2$. (b) Array $A_1$. (c) Array $A_0$. The block decomposition in Figure 1 corresponds to Array $A_3$.](image)

Using the array pyramid, it is very easy to respond to query 1, as we just examine the relevant parts of the stack of arrays. For example, suppose that we want to determine the locations that comprise object $o$, and we use the array pyramid consisting of arrays $A_i \ (0 \leq i \leq n)$ in a two-dimensional space of size $2^n \times 2^n$ where the blocks are squares of side length $2^{n-i}$. We start with $A_0$, which consists of just one element $e$, and determine if $o$ is a member of the set of values associated with $e$. If it is not, then we exit and the answer is negative. If it is, then we examine the four elements in $A_1$ that correspond to $e$ and repeat the test. At this point, we know that $o$ is a member of at least one of them as otherwise $o$ could not have been a member of the set of values associated with element $e$ of $A_i$. This process is applied recursively to elements of $A_i$ that contained $o$ (i.e., the appropriate elements of $A_{i+1}$ are examined for $1 \leq j \leq n-1$) until encountering $A_n$ at which time the process stops. The advantage of this method is that elements of $A_{i+1}$ are not examined unless object $o$ is guaranteed to be a member of the set of values associated with at least one of them.

The array pyramid uses a sequence of arrays as an access structure. An alternative implementation is one that imposes an access structure in the form of a tree $T$ on the elements of the hierarchy $\{C_i\}$. One possible implementation is a tree of fanout $q$ where the root $T_0$ corresponds to $C_n$, nodes $\{T_{ij}\}$ at depth $i$ to $C_i \ (1 \leq i \leq n-1)$, while the leaf nodes $\{T_{nij}\}$ correspond to $C_n$. In particular, element $t$ in the tree at depth $j$ corresponds to element $e$ of $C_j \ (0 \leq j \leq n-1)$ and $t$ contains $q$ pointers to its $q$ sons in $T_{j+1}$ corresponding to the elements of $C_{j+1}$ that are contained in $e$. The result is termed a cell-tree pyramid. Figure 29 shows the cell-tree pyramid corresponding to the collection of objects and cells in the array in Figure 3. This example makes use of two-dimensional data with $r_1 = 2$ and $r_2 = 2$. In this case, notice the similarity between the cell-tree pyramid and the quadtree implementation that uses an access structure which is a tree with a fanout
of 4 (Figure 9).

Using the term *quadtree* in its most general sense (i.e., $d$-dimensional blocks whose sides need not be powers of two nor be of the same length), the cell-tree 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 the maximum depth of the tree — see the complete binary tree in Figure ?? of Section ?? of Chapter ??). Nevertheless, there are some very important differences. The first is that the quadtree is a variable-resolution representation while the cell-tree 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* [109]) which uses the nonleaf nodes to store information about the objects present in the cells and nodes below them (e.g., Figure 30). Note that both the cell-tree pyramid and the truncated-tree pyramid are instances of an implicit interior-based representation with a tree access structure.

Our definition of the pyramid was made in a bottom-up manner in the sense that we started with a block size and a space size. Next, we expanded the size of the space so that a hierarchy of congruent blocks at each
level and similar blocks at different levels could be formed. We can also define a variant of the pyramid where
the requirements of block congruence at each level and block similarity at different levels are relaxed. This is
a bit easier if we define the pyramid in a top-down manner as we can calculate the number of cells by which
the space needs to be expanded as the block sizes at the different levels are defined. It should be clear that
the congruence assumption is more restrictive than the similarity assumption. If we relax the assumption that
the blocks at different levels are similar, but retain the assumption that the blocks at each level are congruent,
then we must store at each level \( i \) the size of the block \( q_i \) (i.e., the values of the individual components \( r_{ij} \) of
\( q_i = \prod_{j=1}^{d} r_{ij} \) for dimension \( j \) (1 \( \leq j \leq d \)).

If we relax both the assumption that the blocks at the different levels are similar and the assumption that
the blocks at each level are congruent, then we are in effect permitting partitioning hyperplanes (i.e., lines in
two dimensions) at arbitrary positions. In this case, we get the most general pyramid if we use a top-down
definition as now we can have a different partition at each block at each level. In particular, this means that
if we assume a tree access structure, then we must store at each nonleaf node the positions of the partitioning
hyperplanes (i.e., lines in two dimensions). The result is an irregular grid (see Section 1.2) which can be im-
plemented as an array structure with pointers to the contained blocks in the immediately deeper level in the
tree plus a set of \( d \) linear scales so that the \( i^{th} \) linear scale indicates the position of the partitioning hyperplane
that is parallel to the hyperplane formed by \( x_i = 0 \) (1 \( \leq i \leq d \)). Recall from Section 1.2 that this is the basis
of the uniform grid. We term this structure an irregular cell-tree pyramid (assuming a tree access structure;
otherwise, we use the terms irregular array pyramid or irregular pyramid, as is appropriate).

Although the regularity of the irregular cell-tree pyramid makes it rather cumbersome to use, special cases
of it do find use and often are quite easy to implement. For example, the dynamically quantized pyramid
(DQP) [92, 125] (see Section ?? of Chapter ??) is a pyramid-like partition of two-dimensional space where
\( q = r_1 \times r_2 \) is defined so that \( r_1 = 2 \) and \( r_2 = 2 \) at all but the deepest level of the hierarchy where they
are free to vary. In addition, the positions of the partitioning hyperplane are free to vary across all levels. Figure ??
in Section ?? of Chapter ?? is an example of a DQP for \( n = 3 \). Notice the close similarity to a complete point
quadtree [45]. The DQP finds use in cluster detection as well as multidimensional histogramming.

It should be clear that even more general definitions can be given of pyramid structures than the one we
described where the requirements of congruent blocks at a given level and similar blocks at all levels were
relaxed. In particular, instead of an irregular grid at each block in each level, we could use any one of the other
recursive and nonrecursive decompositions described in Section 1.2 with the appropriate access structure. A
subset of such representations will be discussed at the end of Section 1.4.2.

Exercises

In Exercises 1–11 assume the use of an array pyramid over a \( 2^n \times 2^n \) space to store an object.

1. What is the maximum number of pyramid elements that must be examined in order to answer query 1
assuming that only one object is stored in the pyramid?

2. Suppose that you use an array pyramid over a \( 2^n \times 2^n \) space to store one object. We now examine the
cost of using the pyramid to respond to query 1. Define the efficiency of the pyramid, \( E_p \), to be the
ratio of the maximum number of pyramid elements examined to the total number of cells in the space
(i.e., the number of elements at the deepest level of the pyramid). \( E_p \) indicates the advantage of using.
the array pyramid hierarchy when answering query 1 with an implicit representation over the cost that
would be incurred had we not used a hierarchy and thereby examined every cell. Define the overhead
of the pyramid, \( O_p \), to be the ratio of the maximum number of pyramid elements examined to the total
number cells in the space that satisfy the query (i.e., are associated with the object whose constituent
cells query 1 is seeking). \( O_p \) indicates the overhead incurred in using the array pyramid hierarchy when
answering query 1 with an implicit representation instead of using an explicit representation. Compute
\( E_p \) and \( O_p \) for an object that occupies just one cell.

3. Compute \( E_p \) and \( O_p \) for an object that occupies the entire space (i.e., \( 2^n \times 2^n \) cells).

38
4. Compute \( E_p \) and \( O_p \) for a line-like object that is one cell wide, is horizontal, and spans the entire space.

5. Compute \( E_p \) and \( O_p \) for a line-like object that is one cell wide, and is diagonal so that one end is in the upper-left corner of the space while the other end is in the lower-right corner of the space.

6. Compute the worst case \( E_p \) and \( O_p \) for an object that occupies a square of size \( 2 \times 2 \) cells.

7. Compute the worst case \( E_p \) and \( O_p \) for an object that occupies a square of size \( 2^i \times 2^i \) cells (\( 1 \leq i \leq n - 1 \)).

8. Compute the worst case \( E_p \) and \( O_p \) for an object that occupies a square of size \( 2^{i+1} \times 2^{i+1} \) cells (\( 1 \leq i \leq n - 1 \)).

9. Compute the worst case \( E_p \) and \( O_p \) for an object that occupies a square of size \( 2^i \times 2^i \) cells (\( 1 \leq i \leq n - 1 \)).

10. Compute the worst case \( E_p \) and \( O_p \) for an object that occupies a square of size \( 2^{i+1} \times 2^{i+1} \) cells (\( 1 \leq i \leq n - 1 \)).

11. Compute the worst case \( E_p \) and \( O_p \) for an object that occupies a square of size \( 2^{i+2} \times 2^{i+2} \) cells (\( 1 \leq i \leq n - 1 \)).

12. Compute the worst case \( E_p \) and \( O_p \) for an object that occupies a square of size \( 2^{i+2} \times 2^{i+2} \) cells (\( 1 \leq i \leq n - 1 \)).

13. Compute the worst case \( E_p \) and \( O_p \) for an object that occupies a square of size \( 2^{i+2} \times 2^{i+2} \) cells (\( 1 \leq i \leq n - 1 \)).

1.4.2 Object-based Hierarchical Interior-based Representations

Our goal here is to be able to take a location \( a \) as input and return the objects in which \( a \) is a member (query 2) when using a representation that stores with each object the addresses of the cells that comprise it. The most natural hierarchy that can be imposed on the objects that would enable us to answer this query is one that aggregates every \( M \) objects (that are hopefully in close spatial proximity, although this is not a requirement) into larger objects. This process is repeated recursively until there is just one aggregated object left. Since the objects may have different sizes and shapes, it is not easy to compute and represent the aggregate object. Moreover, it is similarly difficult to test each one of them (and their aggregates) to determine if they contain \( a \) since each one may require a different test by virtue of the different shapes. Thus, it is useful to use a common aggregate shape and point-inclusion test to prune the search.

The common aggregate shape and point-inclusion test that we use assume the existence of a minimum enclosing box (termed a bounding box) for each object. This bounding box is part of the data associated with each object and aggregate of objects. In this case, we reformulate our object hierarchy to be in terms of bounding boxes. In particular, we aggregate the bounding boxes of every \( M \) objects into a box (i.e., block) of minimum size that contains them. This process is repeated recursively until there is just one block left. The value associated with the bounding box \( b \) is its location (e.g., the coordinate values of its diagonally opposite corners for two-dimensional data). It should be clear that the bounding boxes serve as a filter to prune the search for an object that contains \( a \).

Note that the nature of the aggregation (i.e., using bounding boxes), the number of objects that are being aggregated at each step (as well as whether it can be varied), and, most importantly, deciding which objects to aggregate is quite arbitrary although an appropriate choice can make the search process much more efficient. The decision as to which objects to aggregate assumes that we have a choice in the matter. It could be that the objects have to be aggregated in the order in which they are encountered. This could lead to poor search performance when the objects are not encountered in an order that correlates with spatial proximity. Of course, this is not an issue as long as we just have \( \leq M \) objects.

It should be clear that the issue of choice only arises if we know the identity of all the objects before starting the aggregation process (unless we are permitted to rebuild the hierarchy each time we encounter a new object.
or delete an object), and if we are permitted to reorder them so that objects in aggregate $i$ need not necessarily have been encountered prior to the objects in aggregate $i + 1$, and vice versa. This is not always the case (i.e., a dynamic versus a static database), although for the moment we do assume that we know the identity of all of the objects before starting the aggregation, and that we may aggregate any object with any other object. Observe also that the bounding boxes in the hierarchy are not necessarily disjoint. In fact, the objects may be configured in space in such a way that no disjoint hierarchy is possible. By the same reasoning, the objects themselves need not be disjoint.

The process that we have just outlined can be described more formally as follows. Assume that there are $K$ objects in the space and let $n$ be the smallest power of $M$ such that $M^n \geq K$. Assume that all aggregates contain $M$ elements with the exception of the last one at each level which may contain less than $M$ as $M^n$ is not necessarily equal to $K$. The hierarchy of objects consists of the set $D$ of sets \( \{ D_i \} \) \((0 \leq i \leq n)\) where $D_n$ corresponds to the set of bounding boxes of the individual objects, $D_{n-1}$ corresponds to the result of the initial aggregation of the bounding boxes of $M$ objects into $K/M$ aggregates of objects and consists of $K/M$ bounding boxes, and $D_0$ is a set containing just one element corresponding to the aggregations of all of the objects and is a bounding box that encloses all of the objects. We term the resulting hierarchy an object pyramid. Once again, we have a multiresolution representation as the original collection of objects is described at several levels of detail by virtue of the number of objects whose bounding boxes are grouped at each level. This is in contrast with the cell pyramid where the different levels of detail are distinguished by the sizes of the cells that comprise the elements at each level.

Searching an object pyramid consisting of sets $D_i$ \((0 \leq i \leq n)\) for the object containing a particular location $a$ (i.e., query 2) proceeds as follows. We start with $D_0$, which consists of just one bounding box $b$, and determine if $a$ is inside $b$. If it is not, then we exit and the answer is negative. If it is, then we examine the $M$ elements in $D_1$ that are covered by $b$ and repeat the test using their bounding boxes. Note that unlike the cell pyramid, at this point, $a$ is not necessarily included in the $M$ bounding boxes in $D_1$ as these $M$ bounding boxes are not required to cover the entire space spanned by $b$. In particular, we exit if $a$ is not covered by at least one of the bounding boxes at this level. This process is applied recursively to all elements of $D_j$ for $0 \leq j \leq n$ until all elements of $D_n$ have been processed at which time the process stops. The advantage of this method is that elements of $D_j$ \((1 \leq j \leq n)\) are not examined unless $a$ is guaranteed to be covered by at least one of the elements of $D_{j-1}$.

The bounding boxes serve to distinguish between occupied and unoccupied space, thereby indicating whether or not the search for the objects that contain a particular location (i.e., query 2) should proceed further. At a first glance, it would appear that the object pyramid is rather inefficient for responding to query 2 as in the worst case all of the bounding boxes at all levels must be examined. However, the maximum number of bounding boxes in the object pyramid, and hence the maximum number that will have to be inspected, is $\sum_{j=0}^{n} M^j \leq 2K$.

Of course, we may also have to examine the actual sets of locations associated with each object when the bounding box does not result in any of the objects being pruned from further consideration since the objects are not necessarily rectangular in shape (i.e., boxes). Thus using the hierarchy provided by the object pyramid results in at most an additional factor of two in terms of the number of bounding box tests while possibly saving many more tests. Therefore, the maximum amount of work to answer query 2 with the hierarchy is of the same order of magnitude to that which would have been needed had the hierarchy not been introduced.

As we can see, the way in which we introduced the hierarchy to form the object pyramid did not necessarily enable us to make more efficient use of the explicit interior-based representation to respond to query 2. The problem was that once we determined that location $a$ was covered by one of the bounding boxes, say $b$, in $D_j$ \((0 \leq j \leq n - 1)\), we had no way to access the bounding boxes comprising $b$ without examining all of the bounding boxes in $D_{j+1}$. This is easy to rectify by imposing an access structure in the form of a tree $T$ on the elements of the hierarchy $D$. One possible implementation is a tree of fanout $M$ where the root $T_0$ corresponds to the bounding box in $D_0$. $T_0$ has $M$ links to its $M$ sons \( \{ T_{1k} \} \) which correspond to the $M$ bounding boxes in $D_1$ that comprise $D_0$. The set of nodes \( \{ T_{ik} \} \) at depth $i$ correspond to the bounding boxes in $D_i$ \((0 \leq i \leq n)\), while the set of leaf nodes \( \{ T_{nk} \} \) correspond to $D_n$. In particular, node $t$ in the tree at depth $j$ corresponds to bounding box $b$ in $D_j$ \((0 \leq j \leq n - 1)\), and $t$ contains $M$ pointers to its $M$ sons in $T_{j+1}$ corresponding.
Table 1: Specifications for a collection of rectangles embedded in a $2^6 \times 2^6$ grid with an origin at the lower-left corner. The centroid values are truncated in order to be integers. The Peano-Hilbert order entry is based on coordinate values $x_{\text{centroid}}$ and $y_{\text{centroid}}$.

![Diagram of object-tree pyramid](image)

Figure 31: Specifications for a collection of rectangles embedded in a $2^6 \times 2^6$ grid with an origin at the lower-left corner. The object-tree pyramid to describe this structure.

Figure 32a is an example object-tree pyramid for a collection of 9 rectangle objects with $M = 3$ (and thus $n = 2$) whose specifications are given in Figure 31. The rectangles are assumed to be embedded in a $2^6 \times 2^6$ grid with an origin at the lower-left corner. Figure 32b shows the spatial extent of the bounding boxes of the nodes in Figure 32a, with broken lines denoting the bounding boxes corresponding to the leaf nodes. Note that the object-tree-pyramid is not unique. Its structure depends heavily on the order in which the bounding boxes for the individual objects are aggregated.

The object-tree pyramid that we have just described still has a worst case where we may have to examine all of the bounding boxes in $D_{j+1}$ that are contained in $b$. We use the term *object-tree pyramid* to describe this structure.
can arise in practice because *a* may be included in the bounding boxes of many objects (termed a *false hit*), as the bounding boxes are not disjoint, while *a* is contained in a much smaller number of objects. Equivalently, false hits are caused by the fact that a spatial object may be spatially contained in full or in part in several bounding boxes or nodes while being associated with just one node or bounding box.

However, the object-tree pyramid does guarantee that only the bounding boxes that contain *a* will be examined and no others unlike the object pyramid. Thus we have not improved on the worst-case of the object pyramid in that we may still have to examine $2K$ bounding boxes, although we have reduced its likelihood. It is interesting to observe that the object pyramid and the object-tree pyramid are instances of an explicit interior-based representation since it is still the case that associated with each object *o* is a set containing the addresses of the cells that comprise it. Note also that the access structure only facilitates the determination of the object associated with a particular cell and not which cells are contiguous. Thus the object-tree pyramid is not an instance of an implicit interior-based representation.

The decision as to which objects to aggregate plays an important factor in the efficiency of the object-tree pyramid in responding to query 2. The efficiency of the object-tree pyramid for search operations depends on its abilities to distinguish between occupied space and unoccupied space, and to prevent a node from being examined needlessly due to a false overlap with other nodes.

The extent to which these efficiencies are realized is a direct result of how well our aggregation policy is able to satisfy the following two goals. The first goal is to minimize the number of sons of node *p* that must be visited by the search. This goal is accomplished by minimizing the area common to the sons of *p* (termed *overlap*). The second goal is to reduce the likelihood that each node *p* is visited by the search. This is accomplished by minimizing the total area spanned by the bounding box of *p* (termed *coverage*). Another way of interpreting these goals is that they are designed to ensure that objects that are spatially close to each other are stored in the same node. Of course, at times, these goals may be contradictory. For example, consider the four bounding boxes in Figure 33a. The first goal is satisfied by the split in Figure 33b, while the second goal is satisfied by the split in Figure 33c.

![Figure 33](image-url)  
*(a) Four bounding boxes and the splits that would be induced, (b) by minimizing the total area of the covering bounding boxes of the two nodes and (c) by minimizing the area common to the covering bounding boxes of the two nodes.*

These goals could be satisfied by using trial-and-error methods that examine all possible aggregations and choose the one that yields the minimum amount of overlap or coverage among the constituent bounding boxes of the nodes as well as among the nodes at a given level (e.g., Exercise 45). The cost is clearly prohibitive. These trial-and-error methods can be made more intelligent by use of iterative optimization [56] (see Exercise 46). However, the cost is still too high.

The aggregation techniques described above take the space (i.e., volume) occupied by (termed *extent of*) the bounding boxes of the spatial objects into account. An alternative is to order the objects prior to performing the aggregation. However, in this case, the only choice that we may possibly have with respect to the identity of the objects which are aggregated is when the number of objects (or bounding boxes) that are being aggregated at each step is permitted to vary. The most obvious order, although not particularly interesting or useful, is one that preserves the order in which the objects were initially encountered (i.e., objects in aggregate *i* have been encountered before those in aggregate *i* + 1). The more common orders are based on proximity or on the values...
of a small set of parameters describing a common property that is hopefully related to the proximity (and to a lesser degree to the shape and extent) of the objects or their bounding boxes in one or all of the dimensions of the space in which they lie [73, 86, 103].

The most frequently used ordering technique is based on mapping the bounding boxes of the objects to a representative point in a lower, the same, or a higher dimensional space and then applying one of the space-ordering techniques described in Section 1 and shown in Figure 2 (see Exercise 47). We use the term object number to refer to the result of the application of space ordering. For example, two-dimensional rectangle objects can be transformed into one of the following representative points (e.g., [108]):

1. The centroid.
2. The centroid and the horizontal and vertical extents (i.e., the horizontal and vertical distances from the centroid to the relevant sides).
3. The \(x\) and \(y\) coordinate values of the two diagonally opposite corners of the rectangle (e.g., the upper-left and lower-right corners).
4. The \(x\) and \(y\) coordinate values of the lower-right corner of the rectangle and its height and width.

Once the \(K\) objects have been ordered, the hierarchy \(D\) is built in the order \(D_n, D_{n-1}, \ldots, D_1, D_0\) where \(n\) is the smallest power of \(M\) such that \(M^n \geq K\). \(D_n\) consists of the set of original objects and their bounding boxes. \(D_{n-1}\) is formed as follows. The first \(M\) objects and their corresponding bounding boxes form the first aggregate, the second \(M\) objects and their corresponding bounding boxes form the second aggregate, etc. \(D_{n-2}\) is formed by applying this aggregation process again to the set \(D_{n-1}\) of \(K/M\) objects and their bounding boxes. This process is continued recursively until we obtain the set \(D_0\) containing just one element corresponding to a bounding box that encloses all of the objects. Note that when the process is continued recursively, the elements of the sets \(D_i(0 \leq i \leq n-1)\) are ordered according to the time at which they were created.

There are a number of implementations of the object-tree pyramid. The Hilbert packed R-tree [73] is an object-tree pyramid that makes use of a Peano-Hilbert order. It is important to note that only the leaf nodes of the Hilbert packed R-tree are ordered using the Peano-Hilbert order. The nodes at the remaining levels are ordered according to the time at which they were created.

A slightly different approach is employed in the packed R-tree [103] which is another instance of an object-tree pyramid. The packed R-tree is based on ordering the objects on the basis of some criterion such as increasing value of the \(x\) coordinate or any of the space-ordering methods shown in Figure 2. Once this order has been obtained, the leaf nodes in the packed R-tree are filled by examining the objects in increasing order where each leaf node is filled with the first unprocessed object and its \(M-1\) nearest neighbors which have not yet been inserted in other leaf nodes. Once an entire level of the packed R-tree has been obtained, the algorithm is reapplied to add nodes at the next level using the same nearest neighbor criterion, terminating when a level contains just one node. The only difference between the ordering that is applied at the levels containing the nonleaf nodes from that used at the level of the leaf nodes is that in the former case we are ordering the bounding boxes while in the latter case we are ordering the actual objects.

There are two distinctions between the packed R-tree and the Hilbert packed R-tree. The first is that the packed R-tree construction process makes use of a proximity criterion in the domain of the actual data rather than the domain of the representative points which is the case of the Hilbert packed R-tree. This distinction is quite important as it means that the Hilbert packed R-tree construction process makes no attempt to reduce or minimize coverage and overlap which we shall soon see are the real cornerstones of the R-tree [63]. Therefore, as we point out below, this makes the Hilbert packed R-tree (and, to a lesser extent, the packed R-tree) much more like a B-tree that is constructed by filling each node to capacity.

The second distinction is that the nonleaf nodes of the packed R-tree are also formed by ordering the nodes at the immediately lower level according to a nearest neighbor criterion, while the nonleaf nodes of the Hilbert packed R-tree are formed by ordering the nodes at the immediately lower level according to the time at which
they were created. It is also interesting to note that no matter which of the implementations of the object-tree pyramid that we deploy, the ordering is only used to build the object-tree pyramid. The actual positions of the objects in the ordering (i.e., the object numbers) are not recorded in the object-tree pyramid.

Regardless of how the objects are aggregated, the object-tree pyramid is analogous to a height-balanced $M$-ary tree where only the leaf nodes contain data (objects in this case), and all of the leaf nodes are at the same level. Thus it is good for static data sets. However, in a dynamic environment where objects are added and deleted at will, the object-tree pyramid needs to be rebuilt either entirely or partially to maintain the balance, order, and node size constraints. In the case of binary trees, this issue is resolved by making use of a B-tree, or a $B^+$-tree if we wish to restrict the data (i.e., the objects) to the leaf nodes as is the case in our application. Below, we show how to use the $B^+$-tree to make the object-tree pyramid dynamic.

When the aggregation in the object-tree pyramid is based on ordering the objects, then the objects and their bounding boxes can be stored directly in the leaf nodes of the $B^+$-tree. We term the result an object $B^+$-tree. The key difference between the object $B^+$-tree and the object-tree pyramid is that the $B^+$-tree (and likewise the object $B^+$-tree) permits the number of objects and nodes that are aggregated at each step to vary (i.e., the number of sons per node). In the $B^+$-tree, they usually range between $\lceil M/2 \rceil$ and $M$ with the root having at least 2 sons unless it is a leaf node (see Section ?? of Chapter ??). The only modification to the $B^+$-tree definition is in the format of the nodes of the object $B^+$-tree. In particular, the format of each nonleaf node $p$ is changed so that if $p$ has $j$ sons, then $p$ contains the following 3 items of information for each son $s$:

1. A pointer to $s$.
2. The maximum object number associated with any of the children of $s$ (analogous to a key in the conventional $B^+$-tree).
3. The bounding box $b$ for $s$ (e.g., the coordinate values of a pair of diagonally opposite corners of $b$).

Notice that $j$ bounding boxes are stored in each node corresponding to the $j$ sons instead of just one bounding box as called for in the definition of the object-tree pyramid. This is done to speed up the point-inclusion tests necessary to decide which son to descend when executing query 2. In particular, it avoids a disk access when the nodes are stored on disk.

A leaf node $p$ in the object $B^+$-tree has a similar format with the difference that instead of having pointers to $j$ sons which are nodes in the tree, $p$ has $j$ pointers to records corresponding to the $j$ objects that it represents. Therefore, $p$ contains the following 3 items of information for each object $s$:

1. A pointer to the actual object corresponding to $s$.
2. The object number associated with $s$.
3. The bounding box $b$ for $s$ (e.g., the coordinate values of a pair of diagonally opposite corners of $b$).

Observe that unlike the object-tree pyramid, the object $B^+$-tree does store object numbers in both the leaf and nonleaf nodes in order to facilitate updates. The update algorithms (i.e., data structure creation, insertion, and deletion) for an object $B^+$-tree are identical to those for a $B^+$-tree with the added requirement of maintaining the bounding box information, while the search algorithms (e.g., query 2, window queries, etc.) are identical to those for an object-tree pyramid. The performance of the object $B^+$-tree for answering range queries is enhanced if the initial tree is built by inserting the objects in sorted order filling each node to capacity, subject to the minimum occupancy constraints, thereby resulting in a tree with minimum depth. Of course, such an initialization will cause subsequent insertions to be more costly as they will inevitably result in node split operations whereas this would not necessarily be the case if the nodes were not filled to capacity initially. The Hilbert $R$-tree [74] is an instance of an object $B^+$-tree that applies a Peano-Hilbert space ordering (Figure 2d) to the centroid of the bounding boxes of the objects.\(^{30}\)

---

\(^{30}\)As we will see, the name Hilbert $B$-tree is more appropriate as the update algorithms (e.g., insertion) do not make use of the bounding
Figure 34a is an example Hilbert R-tree for the collection of rectangle objects whose specifications are given in Figure 31 with \( m = 2 \) and \( M = 3 \). Figure 34b shows the spatial extent of the bounding boxes of the nodes in Figure 34a, with broken lines denoting the bounding boxes corresponding to the leaf nodes, and gray lines denoting the bounding boxes corresponding to the subtrees rooted at the nonleaf nodes. We have filled each node to capacity subject to the constraint that each node must have a minimal number of sons. We have also tried to aggregate objects in each node based on the proximity of their positions in the Peano-Hilbert order.

![Image of a diagram](image)

**Figure 34:** (a) Hilbert R-tree for a collection of rectangle objects whose specifications are given in Figure 31 with \( m=2 \) and \( M=3 \), and (b) the spatial extents of the bounding boxes.

Notice that there is much of overlap between the bounding boxes of the nodes. Rearranging the objects that are aggregated in each node can be used to alleviate this problem, but only to a very limited extent as the order of the leaf nodes must be maintained — that is, all elements of leaf node \( i \) must have a Peano-Hilbert order number that is less than all elements of leaf node \( i + 1 \). Thus all we can do is change the number of elements that are aggregated in the node subject to the node capacity constraints. For example, in Figure 34 we could aggregate objects \( E \) and 3 into \( R_3 \) and objects \( G \), 2, and \( D \) into \( R_4 \). However, the real problem is that objects 2 and \( F \) should be aggregated but this is impossible as their corresponding positions in the Peano-Hilbert order are so far apart (i.e., 598 and 3496, respectively). The problem is caused, in part, by the presence of objects with nonzero extent. This is in contrast to experimental results [73] which encourage ordering the objects on the basis of the Peano-Hilbert order of their centroids. The problem is that these experiments were conducted with objects with little or no extent (i.e., point-like objects or line segments with unit length).

When the objects are to be aggregated on the basis of their extent (i.e., the space occupied by their bounding boxes), then good dynamic behavior is achieved by making use of an R-tree [63]. An R-tree is a generalization of the object-tree pyramid where 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 \( M \) while it is always \( M \) for the object-tree pyramid. 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. Of the different variations on the object-tree pyramid that we discussed, the R-tree is the one that is used most frequently, especially in database applications.

Figure 35a is an example R-tree for a collection of rectangle objects whose specifications are given in Figure 31 with \( m = 2 \) and \( M = 3 \). Figure 35b shows the spatial extent of the bounding boxes of the nodes

---

Box information to reduce the coverage and overlap whereas this is the key to their operation in the R-tree. Although experiments for the Hilbert R-tree and the very closely related Hilbert packed R-tree [73] (which is the result of the initial Hilbert R-tree construction process given \( N \) objects) exhibit good behavior, this is primarily on account of the use of point-like data such as very small line segments. Experiments using objects with greater extent (e.g., longer line segments) may be different. The Hilbert R-tree is closely related to the Hilbert tree [82] which applies the same ordering to a set of points and then stores the result in a height-balanced binary tree (see also [134] which makes use of a Morton order and a 1-2 brother tree [93]).
in Figure 35a, with broken lines denoting the bounding boxes corresponding to the leaf nodes, and gray 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.

![Diagram of R-tree](image)

**Figure 35:** (a) R-tree for a collection of rectangle objects whose specifications are given in Figure 31 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.

Given that each R-tree node can contain a varying number of objects or bounding boxes, it is not surprising that the R-tree was inspired by the B-tree [26]. This means that nodes are viewed as analogous to disk pages. Thus the parameters defining the tree (i.e., \( m \) and \( M \)) are chosen so that a small number of nodes is visited during a spatial query (i.e., variants of query 2), which means that \( m \) and \( M \) are usually quite large.

The need to minimize the number of disk accesses also affects the format of each R-tree node. Recall that in the definition of the object-tree pyramid, each node \( p \) contains \( M \) pointers to \( p \)'s sons and one bounding box corresponding to the union of the bounding boxes of \( p \)'s sons. This means that in order to decide which of node \( p \)'s sons should be descended, we must access the nodes corresponding to these sons to perform the point-inclusion test. Each such access requires a disk I/O operation. In order to avoid these disk I/O operations, the format of R-tree node \( p \) is modified so that \( p \) contains \( k \) (\( m \leq k \leq M \)) pointers to \( p \)'s sons and the \( k \) bounding boxes of \( p \)'s sons instead of containing just one bounding box corresponding to the union of the bounding boxes of \( p \)'s sons as is the case for the object-tree pyramid. Recall that this format is also used in the definition of a node in the object \( B^+ \)-tree. Once again, we observe that the \( k \) point-inclusion tests do not require any disk I/O operations at the cost of being able to aggregate a smaller number of objects in each node since \( m \) and \( M \) are now smaller assuming that the page size is fixed.

As long as the number of objects in each R-tree leaf node is between \( m \) and \( M \), no action needs to be taken on the R-tree structure other than adjusting the bounding boxes when inserting or deleting an object. If the number of objects in a leaf node decreases below \( m \), then the node is said to underflow. In this case, the objects in the underflowing nodes must be reinserted, and bounding boxes in nonleaf nodes must be adjusted. If these nonleaf nodes also underflow, then the objects in their leaf nodes must also be reinserted. If the number of objects in a leaf node increases above \( M \), then the node is said to overflow. In this case, it must be split and the \( M+1 \) objects that it contains must be distributed in the two resulting nodes. Splits are propagated up the tree.

Underflows in an R-tree are handled in an analogous manner to the way they are dealt with in a B-tree (see Exercise 10). In contrast, the overflow situation points out a significant difference between an R-tree and a B-tree. Recall that overflow is a result of attempting to insert an item \( t \) in node \( p \) and determining that node \( p \)
is too full. In a B-tree, we usually don’t have a choice (but see Exercise 3) as to the node \( p \) that is to contain \( t \) since the tree is ordered. Thus once we determine that \( p \) is full, we must either split \( p \) or apply a rotation (also known as deferred splitting) process. On the other hand, in an R-tree, we can insert \( t \) in any node \( p \), as long as \( p \) is not full. However, once \( t \) is inserted in \( p \), we must expand the bounding box associated with \( p \) to include the space spanned by the bounding box \( b \) of \( t \). Of course, we can also insert \( t \) in a full node \( p \), in which case we must also split \( p \).

The need to expand the bounding box of \( p \) has an effect on the future performance of the R-tree, and thus we must make a wise choice with respect to \( p \). As in the case of the object-tree pyramid, the efficiency of the R-tree for search operations depends on its abilities to distinguish between occupied space and unoccupied space, and to prevent a node from being examined needlessly due to a false overlap with other nodes. Again, as in the object-tree pyramid, the extent to which these efficiencies are realized is a direct result of how well we are able to satisfy our goals of minimizing coverage and overlap. These goals guide the initial R-tree creation process as well subject to the previously mentioned constraint that the R-tree is usually built as the objects are encountered rather than waiting until all objects have been input.

Not surprisingly, these same goals also guide the node-splitting process. In this situation, one goal is to distribute the objects among the nodes so that the likelihood that the two nodes will be visited in subsequent searches will be reduced. This is accomplished by minimizing the total area spanned by the bounding boxes of the resulting nodes (equivalent to what we termed coverage). The second goal is to reduce the likelihood that both nodes are examined in subsequent searches. This goal is accomplished by minimizing the area common to both nodes (equivalent to what we termed overlap). Again, we observe that, at times, these goals may be contradictory.

A number of node-splitting policies have been proposed that take these goals into account. They are differentiated on the basis of their execution-time complexity and by the number of these goals that they attempt to meet. An easy way to see the different complexities is to look at the following three algorithms [63] all of which are based on minimizing the coverage. The simplest is an exhaustive algorithm that tries all possibilities. In such a case, the number of possible partitions is \( 2^M - 1 \) (see Exercise 11). This is unreasonable for most values of \( M \) (e.g., \( M = 50 \) for a page size of 1024 bytes).

The remaining two node-splitting algorithms have a common control structure that consists of two stages. The first stage “picks” a pair of bounding boxes \( j \) and \( k \) to serve as “seeds” for the two resulting nodes, while the second stage redistributes the remaining bounding boxes into the nodes corresponding to \( j \) and \( k \). The redistribution process tries to minimize the “growth” of the area spanned by \( j \) and \( k \). Thus the first and second stages can be described as “seed-picking” and “seed-growing”, respectively.

The first of these “seed-picking” algorithms is a quadratic cost algorithm (see Exercise 15) that first finds the two bounding boxes that would waste the most area were they to be in the same node. This is determined by subtracting the sum of the areas of the two bounding boxes from the area of the covering bounding box. These two bounding boxes are placed in the separate nodes, say \( j \) and \( k \). Next, the remaining bounding boxes are examined, and for each bounding box, say \( i \), \( d_{ij} \) and \( d_{ik} \) are computed, which correspond to the increases in the area of the covering bounding boxes of nodes \( j \) and \( k \), respectively, when \( i \) is added to them. Now, the bounding box \( r \) such that \( |d_{rij} - d_{rki}| \) is a maximum is found, and \( r \) is added to the node with the smallest increase in area. This process is repeated for the remaining bounding boxes. The motivation for selecting the maximum difference \( |d_{rij} - d_{rki}| \) is to find the bounding box having the greatest preference for a particular node \( j \) or \( k \).

The second of these “seed-picking” algorithms is a linear cost algorithm (see Exercise 16) that first finds the two bounding boxes with the greatest normalized separation along all of the dimensions, and splits along this axis (i.e., using a hyperplane perpendicular to it). The separations are normalized by dividing the actual separation by the width of the bounding box of the overflowing node along the corresponding axis. For example, assuming two-dimensional objects and that the origin is at the upper-left corner of the space, in the vertical dimension these are the bounding boxes with the maximum lower side and the minimum upper side. The remaining bounding boxes are processed in arbitrary order and placed in the node whose covering bounding box is increased the least by their addition. Empirical tests [63] showed that there was not much
difference between the three node-splitting algorithms in the performance of a window search query (i.e., in CPU time and in the number of disk pages accessed). Thus, the faster linear cost node-splitting algorithm was found preferable for this query even though the quality of the splits was somewhat inferior.

An alternative node splitting policy is based on minimizing the overlap. One technique which has a linear cost [4] applies $d$ partitions (one for each of the $d$ dimensions) to the bounding boxes in the node $t$ being split thereby resulting in $2^d$ sets of bounding boxes. In particular, we have one set for each face of the bounding box $b$ of $t$. The partition is based on associating each bounding box $o$ in $t$ with the set corresponding to the closest face along dimension $i$ of $b$. Once the $2^d$ partitions have been constructed (i.e., each bounding box $o$ has been associated with $d$ sets), select the partition that ensures the most even distribution of bounding boxes. In case of a tie, choose the partition with the least overlap. In case of another tie, choose the partition with the least coverage. For example, consider the four bounding boxes in Figure 36. The partition along the $x$ axis yields the sets $\{1, 2\}$ and $\{3, 4\}$ while the partition along the $y$ axis yields the sets $\{1, 3\}$ and $\{2, 4\}$. Since both partitions yield sets that are evenly distributed, we choose the one that minimizes overlap (i.e., along the $y$ axis).

![Figure 36: Example collection of rectangles demonstrating a linear node splitting algorithm that minimizes overlap rather than coverage.](image)

The algorithm is linear as it examines each bounding box once along each dimension (actually, it is $O(dM)$ for $M$ objects but $d$ is usually very small). Experiments with randomly generated rectangles [4] resulted in lower coverage and overlap than the linear and quadratic algorithms described above [63] that are based on minimizing the coverage. The window search query was also found to be about 16% faster with the linear algorithm based on minimizing overlap than the quadratic algorithm based on minimizing coverage. The drawback of this linear algorithm (i.e., [4]) is that it does not guarantee that the two nodes resulting from the partition will contain an equal number of bounding boxes. This is because the partitions are based on proximity to the borders of the bounding box of the node being split. In particular, when the data is not uniformly distributed, although the resulting nodes are likely to have little overlap (as they are likely to partition the underlying space into two equal areas), they will most likely contain an uneven number of bounding boxes (see Exercise 20).

Better decompositions in terms of less node overlap and lower storage requirements than those achieved by the linear and quadratic node-splitting algorithms have also been reported in [14] where three significant changes have been made to the R-tree construction algorithm including a different node-splitting strategy. An R-tree that is built using these changes is termed an *R*-tree [14][13]. These changes are described below. Interestingly, these changes also involve using a node splitting policy that, at times, tries to minimize both coverage and overlap.

The first change is the use of an intelligent object insertion procedure that is based on minimizing overlap in the case of leaf nodes, while minimizing the increase in area in the case of nonleaf nodes. The distinction between leaf and nonleaf nodes is necessary as the insertion algorithm starts at the root and must process non-leaf nodes first. Formally, each bounding box $o$ has two faces $f_{oi}$ and $f_{oih}$ that are parallel to the respective faces $f_{iol}$ and $f_{bih}$ of $b$ where $l$ and $h$ correspond to the low and high values of coordinate or dimension $i$. For each dimension $i$, there are two sets $S_{i,l}$ and $S_{i,h}$ corresponding to faces $f_{i,l}$ and $f_{i,h}$ of $b$, and the algorithm inserts $o$ into $S_{i,l}$ if $x_i(f_{iol}) < x_i(f_{iwh})$ and into $S_{i,h}$ otherwise.

11 Formally, each bounding box $o$ has two faces $f_{oi}$ and $f_{oih}$ that are parallel to the respective faces $f_{iol}$ and $f_{bih}$ of $b$ where $l$ and $h$ correspond to the low and high values of coordinate or dimension $i$. For each dimension $i$, there are two sets $S_{i,l}$ and $S_{i,h}$ corresponding to faces $f_{i,l}$ and $f_{i,h}$ of $b$, and the algorithm inserts $o$ into $S_{i,l}$ if $x_i(f_{iol}) < x_i(f_{iwh})$ and into $S_{i,h}$ otherwise.

12 The `*' is used to signify its "star"-like performance in comparison with R-trees built using the other node-splitting algorithms.
leaf nodes before encountering the leaf node where the object will ultimately be inserted. Thus we see that the bounding box $b$ for an object $o$ is inserted into the leaf node $p$ for whom the resulting bounding box has the minimum increase in the amount of overlap with the bounding boxes of $p$’s brothers (sons of nonleaf node $s$). This is in contrast to the R-tree where $b$ is inserted into the leaf node $p$ for whom the increase in area is a minimum (i.e., based on minimizing coverage). This part of the R*-tree object insertion algorithm is quadratic in the number of entries in each node (i.e., $O(M^2)$) for an order $(m, M)$ R*-tree as the overlap must be checked for each leaf node son $p$ of the selected nonleaf node $s$ with all of $p$’s $O(M)$ brothers (but see Exercise 23).

The second change is that when a node $p$ is found to overflow in an R*-tree, then instead of immediately splitting $p$ as is done in the R-tree, an attempt is made first to see if some of the objects in $p$ could possibly be more suited to being in another node. This is achieved by reinserting a fraction (30% has been found to yield good performance [14]) of these objects in the tree (termed forced reinsertion). Forced reinsertion is similar in spirit to rotation (also known as “deferred splitting”) in a conventional B-tree which was also a technique developed to avoid splitting a node (see Exercise 4).

Forced reinsertion is quite important as usually an R-tree is built by inserting the objects one-by-one as they are encountered in the input. Thus we don’t usually have the luxury of processing the objects in sorted order. This could lead to some bad decompositions in the sense that the redistribution stage may prefer one of the “seed” nodes over the other in a consistent manner (see Exercise 14). Of course, this can be overcome by taking into account the bounding boxes of all of the objects before building the R-tree; but now the representation is no longer dynamic. Forced reinsertion is a compromise in the sense that it permits us to periodically rebuild part of the R-tree as a means of compensating for some bad node placement decisions.

The third change involves the manner in which an overflowing node $p$ is split. Again, as in the original R-tree node-splitting algorithm, a two-stage process is used. The difference is in the nature of the stages. The process follows closely an approach presented in an earlier study of the R-tree [58] which did not result in the coining of a new name! (see Exercise 17). In particular, in contrast to the original R-tree node-splitting strategy [63] where the first stage “picks” two “seeds” for the two resulting nodes which are subsequently “grown” by the second stage, in the R*-tree (as well as in the approach described in [58]), the first stage determines the axis (i.e., hyperplane) along which the split is to take place, while the second stage determines the position of the split. In two dimensions, for example, the split position calculated in the second stage serves as the boundary separating the left (or an equivalent alternative is the right) sides of the bounding boxes of the objects that will be in the left and right nodes resulting from the split.

Note that the calculation of the split position in the second stage plays the same role as the redistribution step in the linear and quadratic cost R-tree node-splitting algorithms. It should also be clear that the two halves resulting from the axis split play the same role as the “seeds” in the original R-tree node-splitting algorithms. However, in the second stage, they do not “grow” in an independent manner in the sense that their constituent bounding boxes $b_i$ are determined by the relative positions of the corresponding faces of $b_i$ (e.g., in two dimensions, the sorted order of their left, right, top, or bottom sides).

This two-stage process is implemented by performing $2d$ sorts (two per axis) of the bounding boxes of the objects in the overflowing node $p$. For each axis $a$, the bounding boxes are sorted according to their two opposite faces that are perpendicular to $a$. The split axis is determined by examining the two sorted lists of possible split positions (i.e., faces of the bounding boxes) for each axis and choosing the axis having the split value for which the sum of the perimeters of the bounding boxes of the two resulting nodes is the smallest (but see Exercise 35) while still satisfying the constraint posed by $m$ and $M$. This factor is somewhat related to the desire to minimize the coverage of the bounding boxes of the resulting nodes (see Exercise 26).
The position of the split along the axis $a$ selected by the first stage is calculated by examining the two sorted lists of possible split positions (i.e., faces of the bounding boxes) for $a$ and choosing the split position for which the amount of overlap between the bounding boxes of the two resulting nodes is the smallest (but see Exercises 27 and 36) while still satisfying the constraint posed by $m$ and $M$. Ties are resolved by choosing the position which minimizes the total area of the resulting bounding boxes thereby reducing the coverage. Minimizing the overlap reduces the likelihood that both nodes will be visited in subsequent searches. Thus we see that the R*-tree’s node-splitting policy tries to address the issues of minimizing both coverage and overlap. Determining the split position requires $O(M)$ overlap computations when scanning the bounding boxes in sorted order. Algorithms that employ this sort-and-scan paradigm are well-known as plane-sweep techniques [9, 97, 120].

Figure 37b shows the space decomposition resulting from the use of the R*-tree in comparison to that resulting from the use of an R-tree that makes use of the linear cost (Figure 37c) and the quadratic cost (Figure 37d) node-splitting algorithm for a sample collection of 1700 line-shaped objects (Figure 37a) using $m = 20$ and $M = 50$. Notice the square-like boxes corresponding to nodes at the bottom level of the R*-tree and the small amount of overlap between them. It is quite clear from the figure that the combined criterion used by the R*-tree node-splitting algorithm that chooses the split which minimizes the sum of the perimeters of the bounding boxes of the two resulting nodes as well as their overlap seems to be working. Whether this is indeed the change in the definition that leads to this behavior is unknown (see Exercises 24 and 25).

Empirical studies have shown that use of the R*-tree node-splitting algorithm instead of the conventional linear and quadratic cost R-tree node-splitting algorithms leads to a reduction in the space requirements (i.e., improved storage utilization) ranging from 10 to 20% [14, 68] while requiring significantly more time to build the R*-tree [68]. The effect of the R*-tree node-splitting algorithms vis-a-vis the conventional linear and quadratic cost node-splitting algorithms on query execution time is not so clear due to the need to take factors such as paging activity, node occupancy, etc. into account [14, 68, 86].

Although the definition of the R*-tree makes three changes to the R-tree, the only real differences are in the way an overflowing node is split, and in the way the bounding boxes are redistributed in the two resulting nodes. In particular, the R-tree algorithms determine “seeds” while the R*-tree algorithm determines a split axis and an axis split value. The bounding boxes of the objects are redistributed about these “seeds” and axis, respectively. At this point, it is important to re-emphasize that the motivation for these redistribution strategies is to avoid the exhaustive search solution which looks at all possible partitions. The R*-tree technique of sorting the boundaries of the bounding boxes along each of the axes and then finding an approximation of the optimal split (with respect to a minimum sum of the perimeters of the resulting bounding boxes) along one of these axes is a heuristic that approximates the solution to the $d$-dimensional problem (i.e., optimal partitioning with minimal coverage or overlap) with an approximation of the optimal one-dimensional solution along one of the axes (see Exercise 35. *** QUESTION: I am not even sure that we are solving the one-dimensional problem in an optimal manner. I will need a proof showing that by looking at the bounding boxes in sorted order we really have gotten the minimal overlap or coverage. ASK MOUNT! *** The validity of this approximation decreases as $d$ (i.e., the dimensionality of the underlying space) increases since more splits are eliminated from consideration.

The remaining changes involving forced reinsertion and intelligent object insertion could have also been used in the R-tree construction algorithms. The evaluation of the R*-tree conducted in [14] involves all three of these changes. An evaluation of R-trees constructed using these remaining changes is also of interest (see Exercises 24 and 25).

One of the drawbacks of the R-tree (as well as its variants such as the R*-tree) is that as the node size (i.e., page size — that is, $M$) gets large, the performance starts to degrade. This is somewhat surprising as according to conventional wisdom, performance should increase with node size as the depth of the tree decreases thereby requiring fewer fewer disk accesses. The problem is that as the node size increases, operations on each node take more time. This is especially true if the operation involves search (e.g., finding the nearest object to a.
Figure 37: (a) A sample collection of 1700 line-shaped objects, and the space decomposition resulting from the use of (b) and R*-tree, and R-trees that make use of (c) linear cost and (d) quadratic cost node-splitting algorithms. All trees are order \( m=20, M=50 \).

This problem can be overcome by ordering the bounding boxes in each node using the same ordering-based aggregation techniques that were used to make the object-tree pyramid more efficient in responding to query 2. For example, we could order the bounding boxes by applying a Morton or Peano-Hilbert space ordering to their centroids. We term the result an ordered R-tree (see Exercise 52). Interestingly, the ordered R-tree can be viewed as a hybrid between an object B⁺-tree and an R-tree in the sense that nodes are ordered internally (i.e., their constituent bounding box elements) using the ordering of the object B⁺-tree, while they are ordered externally (i.e., vis-a-vis each other) using an R-tree.

Although the R-tree is height-balanced, the branching factor of each node is not the same. Recall that each node contains between \( m \) and \( M \) objects or bounding boxes. This has a number of drawbacks. First, it
means that the nodes are not fully utilized thereby causing the tree structure to be deeper than it would be had
the nodes been completely full. Therefore, the number of data objects in the leaf nodes of the descendants
of sibling nonleaf nodes is not the same and, in fact, can vary greatly, thereby leading in imbalance in
terms of the number of objects stored in different subtrees. This could have a detrimental effect on the effi-
ciency of retrieval. Second, satisfying the branching factor condition often requires compromising the goal
of minimizing total coverage, overlap, and perimeter. The packed R-tree and the Hilbert packed R-tree are some
ways to overcome this problem as they initially have a branching factor of \( M \) at all but the last node at each
level. However, they are not necessarily designed to meet our goals of minimizing total coverage, overlap,
and perimeter.

The S-tree [2] is an approach to overcome the above drawbacks of the R-tree and its packed variants by
trading off the height-balanced property in return for reduced coverage, overlap, and perimeter in the resulting
minimum bounding boxes. The S-tree has the property that each node that is not a leaf node or a penultimate
nodes (i.e., a node whose sons are all leaf nodes) has \( M \) sons. In addition, for any pair of sibling nodes (i.e.,
with the same parent) \( s_1 \) and \( s_2 \) with \( N_{s_1} \) and \( N_{s_2} \) objects in their descendants, respectively, we have that
\( p \leq N_{s_1}/N_{s_2} \leq 1/p \) (\( 0 < p \leq 0.5 \)). \( p \) is a parameter that is related to the skewness of the data and governs the
amount of tradeoff thereby providing a worst-case guarantee on the skewness of the descendants of the node.
This guarantee is fairly tight when \( p \) is close to 0.5, while it is quite loose when \( p \) is small. In other words,
when \( p = 0.5 \), the difference in the number of objects that will be found in the subtrees of sibling nodes is
within a factor of 2. This is not the case for an R-tree (see Exercise 53). In addition, the S-tree construction
process makes use of an overlap factor parameter \( o \) (ranging between 0 and 1) defined to be the area covered by
the intersection of the bounding boxes divided by the sum of the corresponding areas of the bounding boxes.

A shortcoming of all of the representations that are based on object hierarchies (i.e., including all of the R-
tree variants) is that when the objects are not hyper-rectangles, use of the bounding box approximation of the
object only eliminates some objects from consideration when responding to queries. In other words, the ac-
tual execution of many queries requires knowledge of the exact representation of the object (e.g., query 2). In
fact, the execution of the query may be quite complex using this exact representation. At times, these queries
may be executed more efficiently by decomposing the object further into smaller pieces and then executing
the query on the individual pieces, thereby overcoming the shortcoming associated with the approximation.
Some suggestions include a decomposition into shapes such as triangles, trapezoids, convex polygons, etc.
(e.g., [80, 19]) as mentioned in Section 1.3 when discussing the explicit representation of nonorthogonal ob-
jects. The TR* -tree [19] is an example of such a representation which decomposes each object in an R* -tree
into a collection of trapezoids. Of course, this technique is also applicable to other R-tree variants.

Exercises

1. Give two possible representative points for circle objects.

2. What is the difference between the definition of a node in an R-tree and a node in a B-tree?

3. In an R-tree we always have a choice with respect to the node in which an object is to be inserted. This
   is not always the case in a B-tree. When do you have a choice as to the B-tree node in which an item
   with key \( k \) can be inserted?

4. Explain why forced reinsertion is unlikely to be useful in a B-tree (i.e., why it would not necessarily
   enable us to avoid having to split a B-tree node)?

5. What is the maximum height of an R-tree of order \( m, M \) with \( N \) objects or bounding boxes (not
   nodes)?

6. What is the maximum number of nodes in an R-tree of order \( m, M \) with \( N \) objects or bounding boxes?

7. Given a set of \( M \) rectangles, devise an algorithm to compute their minimum bounding box. What is the
   execution time of the algorithm?
8. What is the total cost of inserting an object in an R-tree of order \((m, M)\) when no node splitting is needed?

9. Write an algorithm to insert an object into an R-tree of order \((m, M)\). You will have to choose one of the described techniques for splitting a full node with \(M + 1\) records.

10. Write an algorithm to delete an object from an R-tree.

11. Prove that there are \(2^M - 1\) possible partitions of a node in an R-tree of order \((m, M)\) with \(M + 1\) records into two nonempty subsets.

12. Explain why the “seeds picked” by the quadratic cost R-tree node-splitting algorithm tend to be small and far apart.

13. Suppose that the “seeds picked” by the quadratic cost R-tree node-splitting algorithm are small. Suppose further that there exists a bounding box \(r\) far away from one of the “seeds’’ \(s\) where \(d-1\) of the coordinate values of \(r\) are almost in agreement with those of \(s\). Show that \(r\) is highly likely to be the first to be distributed and will be distributed to the bounding box of \(s\).

14. Let \(b_1\) be the first bounding box to be redistributed by the quadratic cost R-tree node-splitting algorithm and suppose that it is redistributed to “seed” \(s\). Explain why the algorithm has a tendency to prefer “seed” \(s\) for the next bounding box \(b_2\) to be redistributed? What happens to the remaining bounding boxes in this case (i.e., after \(b_1\) and \(b_2\) have been redistributed)?

15. Prove that the quadratic cost R-tree node-splitting algorithm requires \(O(M^2)\) time for an order \((m, M)\) R-tree.

16. The “seed-picking” stage of the linear cost R-tree node-splitting algorithm finds the two bounding boxes with the maximum normalized separation along all of the dimensions. This is based on a definition of separation in terms of the maximum distance between the furthest of the parallel sides of two bounding boxes. Suppose that the definition of furthest was changed to be the maximum separation between the closest of the parallel sides of the two bounding boxes [58]. Is the cost of the resulting algorithm still linear (i.e., \(O(M^2)\)) for an R-tree of order \((m, M)\)?

17. Both the linear and quadratic cost R-tree node-splitting algorithms consist of a stage that “picks seeds” \(j\) and \(k\), and a stage that redistributes the remaining bounding boxes into the nodes corresponding to \(j\) and \(k\). The linear cost algorithm redistributes the remaining bounding boxes by inserting them into the node \(j\) or \(k\) whose bounding box is increased (i.e., “grows”) the least by their addition. This is based on the principle of minimizing coverage. An alternative approach “picks” an axis in the first stage for which the normalized separation between the “seeds” is a maximum. In addition, instead of “seed-growing”, an alternative redistribution algorithm is used that sorts the bounding boxes by their low value along the chosen split axis, and then assigns the first \(M/2\) bounding boxes to one node and the remaining bounding boxes to the other node [58]. In this case, there is no minimization of “growth” in the sense that once the sort has taken place, the redistribution just depends on the value of \(M\). Compare the performance of this alternative redistribution algorithm in conjunction with the following two approaches to “picking” the split axis:

1. The axis with the greatest normalized separation for the two “seeds” \(j\) and \(k\) selected by the quadratic cost algorithm. The separations are normalized by dividing the actual separation by the length of the side of the bounding box of \(j\) and \(k\) along the appropriate axes.

2. The variation on the technique used in the linear cost algorithm described in Exercise 16 that finds the two bounding boxes \(j\) and \(k\) with the maximum separation between the closest of the parallel sides of \(j\) and \(k\) [58].

18. What is the cost of the variation of the node-splitting algorithm that makes use of technique 2 in Exercise 17 for an R-tree of order \((m, M)\)?

\[\text{[58] Technique 1 is erroneously attributed to [58] in [14], whereas technique 2 is really the one described in [58].}\]
19. What is the drawback of using the alternative redistribution algorithm described in Exercise 17 which sorts the bounding boxes in an order \((m, M)\) R-tree by their low value along the chosen split axis, and then assigns the first \(M/2\) bounding boxes to one node and the remaining bounding boxes to the other node [58]?

20. In [4], experiments are described that show that a linear node splitting algorithm that is based on reducing overlap results in lower coverage and overlap than the linear and quadratic algorithms described in [63]. The experiments used randomly generated rectangles. In this case, the data is well-distributed throughout the underlying space and thus the algorithm will naturally result in creating partitions which have approximately the same number of bounding boxes. In contrast, this may not be the case for non-random data. Perform a similar comparison using real data. The comparison should be qualitative and quantitative. The qualitative comparison should look at the resulting space decomposition as in Figure 37. This can be achieved by choosing a sample data set and drawing the output. The quantitative comparison should tabulate coverage and overlap, storage utilization, the time for construction, as well as the times for intersection (i.e., determining all pairs of intersecting objects), window, and point queries.

21. In the study of the linear node splitting algorithm that is based on reducing overlap [4], all comparisons were made with a quadratic algorithm that was based on minimizing coverage (i.e., [63]). Conduct a similar study as in Exercise 20 for both random and real data using an R*-tree. This means that now our comparison uses a quadratic node splitting algorithm that attempts to reduce both overlap and coverage as this is the motivation for the R*-tree.

22. What is the total cost of inserting an object in an R*-tree when no node splitting is needed?

23. When inserting an object into an R*-tree we insert its bounding box \(b\) into the leaf node \(p\) for whom the resulting bounding box has the minimum increase in the amount of overlap with \(p\)'s brothers that are sons of nonleaf node \(s\). This algorithm is quadratic in the number of entries in each node. We can improve on this bound by observing that insertion of \(b\) in the bounding boxes of brother leaf nodes that are far from \(b\) are unlikely to result in minimizing the overlap. How would you take advantage of this to modify the insertion algorithm to avoid the quadratic cost and get an approximated minimum overlap solution?

24. Compare the performance of the R*-tree with R-trees that make use of the linear cost and quadratic cost node-splitting algorithms both of which are implemented with forced reinsertion. The comparison should be qualitative and quantitative. The qualitative comparison should look at the resulting space decomposition as in Figure 37. This can be achieved by choosing a sample data set and drawing the output. The quantitative comparison should tabulate coverage and overlap, storage utilization, the time for construction, as well as the times for intersection (i.e., determining all pairs of intersecting objects), window, and point queries. You should use real data, if possible.

25. Compare the performance of the R*-tree with R-trees that make use of the linear cost and quadratic cost node-splitting algorithms both of which are implemented with an intelligent object insertion procedure that is based on minimizing overlap in the case of leaf nodes, while minimizing the increase in area in the case of nonleaf nodes. Perform a similar comparison as in Exercise 24.

26. The R*-tree node-splitting rule tries to address both the issue of minimizing coverage and of minimizing overlap. Thus it does not make use of a rule that minimizes just one of these factors. Instead, it makes use of a two-stage node-splitting process where the first stage finds the axis along which the split takes place while the second stage calculates the position of the split. Given that we want a rule that minimizes coverage while still leading to a reduction in overlap, explain why it may be better for the first stage that chooses the split axis to try to minimize the sum of the perimeters of the bounding boxes of the resulting nodes rather than to try to minimize the total area spanned (i.e., the coverage) by their bounding boxes?

27. The first stage of the R*-tree node splitting algorithm determines the axis \(a\) having the split position \(s_1\) for which the sum of the perimeters of the bounding boxes of the two nodes is the smallest, while the
second stage calculates a new (possibly different) split position $s_2$ along axis $a$ for which the amount of overlap between their bounding boxes is the smallest. Why not use the split position $s_1$ and dispense with the second stage that searches for the split position that minimizes overlap? In other words, explain why $s_1$ may not necessarily minimize overlap? Give an example of this phenomenon.

28. Despite the results of Exercise 27, again use the axis having the split value for which the sum of the perimeters of the bounding boxes of the two nodes is the smallest but now also use this split value as the position of the split. Compare the performance of this variation with the conventional R*-tree definition.

29. Repeat Exercise 24 with the variation of the R*-tree definition given in Exercise 28.

30. Repeat Exercise 25 with the variation of the R*-tree definition given in Exercise 28.

31. Repeat Exercise 28 by varying the definition of the R*-tree to use the axis having the split value for which the amount of overlap between the bounding boxes of the two resulting nodes is the smallest and also use this split value as the position of the split.

32. Repeat Exercise 24 with the variation of the R*-tree definition given in Exercise 28.

33. Repeat Exercise 25 with the variation of the R*-tree definition given in Exercise 31.

34. As we saw, the R*-tree algorithms try to minimize the coverage, perimeter, and overlap in the node insertion process. Each of these factors is taken into account separately in different stages of the algorithm. An alternative approach is to define an objective function which takes these three factors, with appropriate weights, into account simultaneously and to try to minimize it. This objective is applied when determining which node is to contain the new object, as well as in the node-splitting process. Investigate the use of this approach by defining some factors and compare your results with the R*-tree node insertion algorithm that we described.

35. Given a set of $M$ bounding boxes of two-dimensional objects, suppose that you want to subdivide the set into two nonempty sets so that the sum of the perimeters of their bounding boxes is minimized. For two-dimensional data, the approach that is used in the R*-tree node-splitting algorithm decomposes this problem into four plane-sweep subproblems, one in each of the two dimensions and two sweep directions, as follows. Sort the left sides of all of the bounding boxes and then sweep a vertical line across this set stopping at every left boundary $l_i$, treating it as the separator between two sets $L$ and $R$ containing all of the bounding boxes whose left sides are $\leq l_i$ and those whose left sides are $> l_i$, respectively. Compute the sum of the perimeters of the bounding boxes of $L$ and $R$ retaining the position if it is smaller than the previous minimum. Repeat this sort and sweep process for the right sides of the bounding boxes and likewise for the top and bottom sides of the bounding boxes. Is this approach guaranteed to find the partitioning for which the sum of the perimeters of their bounding boxes is a minimum? In other words, is it possible that a better partition can be obtained where, without loss of generality, it is not the case that one of the nodes contains all of the bounding boxes whose left (right) sides are $\leq l_i$ and the other node contains all of the bounding boxes whose left (right) sides are $> l_i$? Note that the R*-tree algorithm only looks at $dM$ of the number of possible partitions, which is much smaller (see Exercise 11). Prove that this guarantee hold or give a counterexample. If the guarantee does not hold, can you obtain an optimal solution by the use of an auxiliary data structure such as a segment tree [17] or an interval tree [34, 35, 36] to keep track of the bounding boxes that are still active during the sweep process?

36. Repeat Exercise 35 for the problem of subdividing a set of $M$ bounding boxes into two nonempty sets so that instead of minimizing the sum of their perimeters, we now try to minimize the area common to their bounding boxes.

37. Write an algorithm to perform a point query in an R-tree.

38. Consider an R-tree built for objects whose centroids and extents are uniformly distributed in the unit space and point queries that are also uniformly distributed over the same unit space. Prove that the expected number of R-tree nodes visited by a point query is equal to the sum of the $d$-dimensional volumes of the bounding boxes of all of the nodes.
39. Write an algorithm to perform a window query in an R-tree.

40. Using the same environment as in Exercise 38, consider a rectangular window query to retrieve all objects whose bounding boxes have a nonempty intersection with window \( w \) having width \( w_i \) in dimension \( i \). Prove that the expected number of R-tree nodes visited is \( \sum_{j=1}^{n} \prod_{i=1}^{m} (b_{ji} + w_i) \) where \( b_j \) is the bounding box of R-tree node \( q_j \) and \( b_{ji} \) is the width of \( b_j \) along dimension \( i \).

41. Write an algorithm to find the nearest object to a point in an R-tree.

42. Write an algorithm to solve the rectangle intersection problem for an R-tree.

43. Prove that the worst-case execution time of the rectangle intersection problem for \( N \) rectangles using an R-tree is \( O(N^2) \).

44. What is the expected cost of the rectangle intersection problem for \( N \) rectangles using an R-tree?

45. What is the cost of an algorithm to build a packed R-tree that starts out by examining the \( K \) objects and finding the \( M \) objects with the smallest total bounding box, followed by the next \( M \) remaining objects with the smallest bounding box, until \( K/M \) bounding boxes have been obtained. This process is repeated recursively until there is just one box left [103].

46. Write an algorithm to build a packed R-tree for a collection of two-dimensional objects that minimizes the total area covered by the bounding boxes at all of the levels.

47. Measure the performance of point queries and window queries (in terms of the number of R-tree nodes visited) on packed R-trees for two-dimensional rectangle objects of your choice which have been transformed into the four representative point transformations described in the text and sorted using the row, Morton, and Peano-Hilbert orders. Generate the queries for uniformly distributed points and likewise for windows whose centroids and horizontal and vertical extents are drawn from a uniform distribution. Compare your results with those for an R-tree constructed using the R*-tree, linear cost, and quadratic cost node-splitting algorithms where the objects are inserted in the same order as in the packed R-tree for the corresponding space-ordering technique.

48. Write an algorithm to perform a point query in an ordered R-tree.

49. Write an algorithm to perform a window query in an ordered R-tree.

50. Write an algorithm to find the nearest object to a point in an ordered R-tree.

51. Write an algorithm to solve the rectangle intersection problem for an ordered R-tree.

52. Implement an ordered R-tree using the different node-splitting algorithms (i.e., R*-tree, linear cost, and quadratic cost) and compare its performance with the corresponding nonordered versions. Implement the ordered R-tree by applying the Morton and Peano-Hilbert space orderings to their representative points. Use the following two types of representative points for the bounding boxes:

   1. The centroid.
   2. The centroid and the horizontal and vertical extents (i.e., the horizontal and vertical distances from the centroid to the relevant sides).

   The comparison should include the time for construction, as well as the times for intersection (i.e., determining all pairs of intersecting objects), window, point, and nearest object queries. You should use real data, if possible (e.g., a collection of rectangle objects or lines).

53. Consider an R-tree where each node contains between \( m \) and \( M \) objects or bounding boxes. The S-tree is designed to provide a guarantee on the relative difference in the number of objects found in the subtrees of two sibling nodes. In particular, for skewness factor \( p \), given a pair of sibling nodes \( s_1 \) and \( s_2 \) with \( N_{s_1} \) and \( N_{s_2} \) objects in their descendants, respectively, we have that \( p \leq N_{s_1}/N_{s_2} \leq 1/p \) (0 < \( p \) ≤ 0.5). Show that the ratio \( N_{s_1}/N_{s_2} \) can get arbitrary large in an R-tree.

56
1.4.3 Disjoint Object-based Hierarchical Interior-based Representations

In our descriptions of the object pyramid and the object-tree pyramid in Section 1.4.2 we observed 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. The fact that each object is only associated with one node while being contained in possibly many bounding boxes (e.g., in Figure 35, rectangle 1 is contained in its entirety in \( R_1, R_2, R_3, \) and \( R_5 \)) means that query 2 may often require several nonleaf nodes to be visited before determining the object that contains \( a \). This problem also arises in the R-tree as seen in the following example.

Suppose that we wish to determine the identity of the rectangle object(s), in the collection of rectangles given in Figure 35 that contains point \( Q \) at coordinate values \((22, 24)\). We first determine that \( Q \) is in \( R_0 \). Next, we find that \( Q \) can be in both or either of \( R_1 \) or \( R_2 \), and thus we must search both of their subtrees. Searching \( R_1 \) first, we find that \( Q \) could only be contained in \( R_3 \). Searching \( R_3 \) does not lead to the rectangle that contains \( Q \) even though \( Q \) is in a portion of rectangle \( D \) that is in \( R_3 \). Thus, we must search \( R_2 \) and we find that \( Q \) can only be contained in \( R_5 \). Searching \( R_5 \) results in locating \( D \), the desired rectangle. The drawback of the R-tree as well as other representations that make use of an object pyramid is that unlike those based on the cell pyramid they do not result in a disjoint decomposition of space. Recall that the problem is that an object is only associated with one bounding bounding box (e.g., rectangle \( D \) in Figure 35 is associated with bounding box \( R_5 \), yet it overlaps bounding boxes \( R_1, R_2, R_3, \) and \( R_5 \)). In the worst case, this means that when we wish to respond to query 2 (e.g., given a point, determining the containing rectangle in a rectangle database, or an intersecting line in a line segment database, etc. in the two-dimensional space from which the objects are drawn), we may have to search the entire database. Thus what we need is a hierarchy of disjoint bounding boxes.

An obvious way to overcome this drawback is to use one of the hierarchical image-based representations described in Section 1.4.1. Recall that these representations made use of a hierarchy of disjoint cells that completely spanned the underlying space. The hierarchy consists of a set of sets \( \{ C_j \} \) (0 ≤ \( j \) ≤ \( n \)) where \( C_j \) corresponds to the original collection of cells, and \( C_0 \) corresponds to one cell. The hierarchies differed in the number and size of the constituent cells at the different depths, although each hierarchy was usually a containment hierarchy in the sense that a cell at depth \( i \) usually contained all of the cells below it at depth \( i + 1 \). The most general hierarchy that was described was the irregular pyramid. In this section, we focus on the irregular cell-tree pyramid variant as we assume a tree access structure.

A simple way to adapt the irregular cell-tree pyramid to our problem is to overlay the decomposition induced by \( C_{n-1} \) (i.e., the next to the deepest level) on the bounding boxes \{ \( b_i \) \} of the objects \{ \( o_i \) \} thereby decomposing the bounding boxes and associating each part of the bounding box with the covering cell of the irregular cell-tree pyramid. Note that we use the hierarchy at the next to the deepest level (i.e., \( C_{n-1} \)) rather than the hierarchy at the deepest level (i.e., \( C_n \)) as the deepest level contains the original collection of unit-sized cells and thus does not correspond to any aggregation. The cells \( c_{j,k} \) at the remaining levels \( j \) (0 ≤ \( j \) ≤ \( n - 2 \)) are formed in the same way as in the irregular cell-tree pyramid — that is, they contain the union of the objects associated with the cells comprising \( c_{j,k} \). Using our terminology, we term the result an irregular bounding-box cell-tree pyramid or bounding-box cell-tree pyramid for short. It should be clear that the depth of the bounding-box cell-tree pyramid is one less than that of the corresponding irregular cell-tree pyramid.

The definition of the irregular cell-tree pyramid as well as the other hierarchical image-based representations stipulates that each cell is contained in its entirety in one or more objects. Equivalently, a cell cannot be partially in object \( o_1 \) and partially in object \( o_2 \). The same restriction also holds for block decompositions which are not hierarchical (see Section 1.2). In contrast, in the case of the bounding-box cell-tree pyramid, the fact that the bounding boxes are just approximations of the objects enables us to relax this restriction in the sense that we allow a cell (or a block in the case of the block decompositions of Section 1.2) to contain parts of the bounding boxes of several objects. In other words, cell \( b \) can be partially occupied by part of the bounding box \( b_1 \) of object \( o_1 \), by part of the bounding box \( b_2 \) of object \( o_2 \), and may even be partially empty.

The bounding-box cell-tree pyramid is a hierarchy of grids, albeit that the grid sizes are permitted to vary
in an arbitrary manner between levels. This definition is still overly restrictive in the sense that we want to be able to aggregate a varying but bounded number of cells at each level (in contrast to a predefined number) so that we can have a height-balanced dynamic structure in the spirit of the B-tree. We also wish to use a hierarchy that makes use of a different block decomposition rule (e.g., a k-d tree, generalized k-d tree, point quadtree, bintree, region quadtree, etc.) instead of a grid as in the case of the irregular cell-tree pyramid.

Our solution is equivalent to a marriage of the bounding-box cell-tree pyramid hierarchy with one of the block decompositions described in Section 1.2. This is done by choosing a value $M$ for the maximum number of cells (actually blocks) that can be aggregated and a block decomposition rule (e.g., a generalized k-d tree). As we are propagating the identity of the objects associated with the bounding boxes up the hierarchy rather than the occupied space occupied by them, we use an object-based variant of the block decomposition rule. This means that a block is decomposed whenever it contains the bounding boxes of more than $M$ objects. Note that the occupied space is implicit to the block decomposition rule and thus need not be explicitly propagated up the hierarchy.

It should be clear that each object’s bounding box can only appear once in each block as the objects are continuous. If more than $M$ of the bounding boxes overlap each other in block $b$, then there is no point in attempting to decompose $b$ further as we will never be able to find subblocks $b_i$ of $b$ so that each of $b_i$ has $M$ or fewer bounding boxes. Observe also that although the block decompositions yield a partition of space into disjoint blocks, the bounding boxes at the lowest level of the hierarchy may not necessarily be disjoint. For example, consider a database of line segment objects and the situation of a vertex where five of the line segments meet. It is impossible for the bounding boxes of the line segments to be disjoint.

The object-based variants of the block decomposition rules are quite different from their image-based counterparts that were discussed in Section 1.2 which based the decomposition on whether the space spanned by the block was completely covered by an object. It is important to reiterate that the blocks corresponding to the leaf nodes do not represent hyper-rectangular aggregates of identically-valued unit-sized cells as in the conventional pyramid. Instead, they represent hyper-rectangular aggregates of bounding boxes of objects or pieces thereof.

Without loss of generality, assuming a generalized k-d tree, the hierarchy of sets $\{H_j\}$ ($1 \leq j \leq n$) is defined as follows. $H_0$ consists of one block. $H_1$ consists of a subset of the nodes of a generalized k-d tree decomposition $Z$ with no more than $M$ members whose corresponding blocks span the entire underlying space. $H_2$ is formed by removing from $Z$ all nodes corresponding to members of $H_1$ and their ancestors, and then applying the same rule that was used to form $H_1$ to each of the blocks in $H_1$ with respect to $Z$. In other words, $H_2$ consists of generalized k-d tree decompositions of the blocks $h_{1k}$ ($1 \leq k \leq M$) that comprise $H_1$. Each element of $H_2$ contains no more than $M$ blocks for a a maximum of $M^2$ blocks. This process is repeated at each successive level up to the leaf level at depth $n − 1$. The nodes at the leaf level contain the bounding boxes of the objects or parts of the bounding boxes of the objects. The pyramid means that the hierarchy must be height-balanced with all leaf nodes at the same level, and that the cells at depth $j$ are disjoint and that they span the space covered by the cells at the immediately lower level at depth $j + 1$.

We term the resulting data structure a generalized k-d tree cell-tree pyramid on account of the use of the generalized k-d tree as the building block of the pyramid, although it is more commonly known as a $k$-$d$-B-tree [100] on account of the similarity of the node structure to that of a B-tree. If we would have used the point quadtree or the bintree as the building blocks of the hierarchy, then we would have termed the result a point quadtree cell-tree pyramid or a bintree cell-tree pyramid, respectively. It is interesting to note that the k-d-B-tree was originally developed for storing point-like objects (see the discussion in Section ?? of Chapter ??) although the extension to objects with extent is relatively straightforward as shown here (but see Exercise 1).

Figure 38 is an example of one possible k-d-B-tree for the collection of rectangles whose specifications are given in Figure 31. Broken lines denote the leaf nodes, and thin lines denote the space spanned by the subtrees rooted at the nonleaf nodes. Of course, other variations are possible since the k-d-tree is not unique. This particular tree is of order $(2,3)$ although in general it is not possible to always guarantee that all nodes will have a minimum of 2 entries, nor is the minimum a part of the definition of the k-d-B-tree. Notice that rectangle object D appears in three different nodes, while rectangle objects A, B, E, and G appear in two different
nodes. Observe also that the example uses a partition scheme that cycles through the axes in the order \(x, y, x, y, \ldots\) although, as we shall see below, this cycling is not guaranteed to hold once objects are inserted and deleted.

![Figure 38: (a) \(k\)-d-B-tree for a collection of rectangle objects whose specifications are given in Figure 31 with \(m=2\) and \(M=3\), and (b) the spatial extents of the nodes. Notice that only in the leaf nodes are the bounding boxes minimal.](image)

Our definition of the structure was given in a top-down manner. In fact, the structure is built in a bottom-up manner by inserting the objects one-at-a-time. Initially, the hierarchy contains just one node corresponding to the bounding box of the single object. As each additional object \(o\) is processed, we insert \(o\)'s bounding box \(b\) into all of the leaf nodes which overlap it. If any of these nodes become too full, then we split these nodes using an appropriate block decomposition rule and determine if the parent is not too full so that it can support the addition of a son. If not, then we recursively apply the same decomposition rule to the parent. The process stops at the root in which case overflow will usually cause the hierarchy to grow by one level.

For example, consider the insertion of rectangle object \(R\) with a lower left corner at \((22, 29)\) and an upper-right corner at \((38, 32)\) in the \(k\)-d-B-tree in Figure 38. \(R\) overlaps leaf nodes \(R_6\) and \(R_7\) and its insertion causes them to overflow. \(R_6\) is split into nodes \(R_{61}\) and \(R_{62}\) at \(y = 32\), while \(R_7\) is split into nodes \(R_{71}\) and \(R_{72}\) at \(y = 29\). These four nodes (i.e., \(R_{61}, R_{62}, R_{71}, \) and \(R_{72}\)) now become the sons of node \(R_2\) which causes \(R_2\) to overflow. \(R_2\) is split at \(x = 26\) into nodes \(R_{21}\) containing nodes \(R_{61}\) and \(R_{62}\) and \(R_{22}\) containing nodes \(R_{71}\) and \(R_{72}\).

These two nodes (i.e., \(R_{21}\) and \(R_{22}\)) replace \(R_2\) as the sons of the root node \(R_0\) along with \(R_1\) and \(R_3\) which now causes \(R_0\) to overflow. At this point, we split \(R_0\) along \(x = 22\) into two nodes \(R_A\) and \(R_B\), where \(R_A\) consists of node \(R_1\) and \(R_B\) consists of nodes \(R_{21}, R_{22},\) and \(R_3\). This causes the height of the tree to grow by one level thereby resulting in the creation of a new root node \(R_Z\) as shown in the \(k\)-d-B-tree given in Figure 39. Notice that the resulting tree is no longer order \(2,3\).

The process that we have described is analogous to that used in a B-tree upon overflow. The difference is that there are a number of ways to deal with overflow, in the sense of where the node and the underlying space are to be partitioned. The most straightforward one is to split the node along the partition line that corresponds to the partition at the root of the \(k\)-d tree of the underlying space that is spanned by the node that overflowed. In fact, this is the strategy that was pursued in the example illustrated in Figure 39. In particular, we split node \(R_0\) at \(x = 22\) so that the newly created root node \(R_Z\) has one son \(R_A\) containing \(R_1\), and a second son \(R_B\) containing \(R_{21}, R_{22},\) and \(R_3\). The advantage of splitting the node in this way is that once the node has been split, we do not need to worry about repartitioning nodes at lower levels in the tree. Interestingly, the same split strategy is also used in the LSD tree [64].

Unfortunately, an undesirable side-effect of the split strategy described above is that the resulting tree may be unbalanced heightwise thereby possibly causing it to be deeper than necessary. The LSD tree tries to ad-
dress the issue of balance by having two types of nodes with varying capacities that do not contain objects (see Section ?? of Chapter ?? for more details). In the k-d-B-tree, the issue of balance can be overcome by choosing another partition line as well as an axis for partitioning the overflowing node. The drawback of such a solution is that this may force the repartitioning of nodes at deeper levels in the hierarchy, and the reinsertion of the regions corresponding to the new partitions in the appropriate nodes.

In order to see the need for repartitioning, let us return to the example showing the result of inserting rectangle object $H$ into the k-d-B-tree in Figure 38. Assume that all splits at levels deeper than the root have been made as in the earlier discussion, and that we are now processing the root $R_0$ which has overflowed (see Figure 39b). Now, instead of partitioning $R_0$ at $x=22$, we introduce a new partition at $x=26$ yielding nodes $R_A$ and $R_B$. Again, this causes the height of the tree to grow by one level thereby resulting in the creation of a new root node $R_Z$.

The difference from the partition that was performed at $x=22$ is that we must now also repartition some of the nodes at deeper levels in the tree. In particular, we have to split node $R_3$ into two parts: $R_{31}$ which belongs in $R_A$ and $R_{32}$ which belongs in $R_B$. Therefore, $R_A$ consists of $R_1, R_21$, and $R_{31}$, while $R_B$ consists of $R_{22}$ and $R_{32}$. At the next level of the tree we see that a further ramifications of the split of $R_3$ is that $R_8$ has been split into two parts $R_{81}$ and $R_{82}$. The effect of this split is that $R_{81}$ is now a part of $R_{31}$, while $R_{82}$ is now a part of $R_{32}$ which also contains $R_9$. Figure 40 shows the result of pursuing this alternative option.

When choosing the partition line and axis as in Figure 40b we must take care so that the newly-created nodes do not overflow. For example, suppose that we choose to partition the overflowing node $R_0$ in Figure 40b at line $l$ at $y=15$ (instead of at $x=22$ or at $x=26$ as in Figure 40b). Figure 41 shows the result of this.
partition at the level of node R0. Again, let RA and RB represent the newly-created nodes corresponding to the area below and above $y = 15$, respectively. Therefore, all elements of R0 that are below $l$ (e.g., none in Figure 39b) are inserted into RA, while all elements of R0 that are above $l$ (e.g., regions R21 and R22 in Figure 39b) are inserted into RB. In addition, the elements of R0 whose regions are intersected by $l$ (e.g., R1 and R3) must be split and new regions created which are inserted into RA and RB.

The result is that RA contains 2 regions while RB contains 4 regions. However, each node can contain a maximum of 3 entries. Therefore, RB has overflowed which means that RB must also be split thereby causing the k-d-B-tree to grow by yet an additional level. With the appropriate choice of partition lines, this process can be applied repeatedly to create much larger k-d-B-trees. Clearly, this is undesirable and we should choose another partition line and axis. In fact, it is not difficult to construct another example where both of the newly-created nodes have overflowed (see Exercise 2).

*** ADD STUFF HERE ***

In fact, when the k-d-B-tree is used to represent objects (as here), we may find that repartitions that occur at deeper levels of the tree may result in nodes overflowing again thereby possibly causing the reinsertion process to require several iterations of going up and down the tree\(^{14}\).

HORSESHOE BUSINESS MAY BE WRONG because the object has one bounding box which is split up as the space is partitioned. We do not create new bounding boxes for the object. This would only be the case for the R+-tree rather than the k-d-B-tree.

\(^{14}\) This problem does not arise when k-d-B-trees are used to represent point objects (see Exercise 1).
All of the example partitionings of the overflowing k-d-B-tree node that we described resulted in the growth of the k-d-B-tree by one level. We propose to avoid this growth by making use of a technique analogous to rotation (also known as deferred splitting) in B-trees and R-trees (as discussed in Section 1.4.2). The idea is that instead of partitioning the overflowing node \( R \) into just two nodes resulting in the growth of the height of the tree, we repartition the space \( S \) spanned by \( R \) several times (as long as \( R \) does not overflow), and then rebuild the k-d-B-tree in a top-down manner. Notice that in this case, no new node is added when the process starts. Of course, we may have to perform many reinsertions which may in turn result in the addition of nodes. We use the term k-d-B-tree re-splitting to describe this overflow strategy.

As an example of the use of k-d-B-tree re-splitting, let us return again to the example showing the result of inserting rectangle object \( H \) in the k-d-B-tree in Figure 38. Assume that all splits at levels deeper than the root have been made as in the earlier discussion, and that we are now processing the root \( R_0 \) which has overflowed (see Figure 39). We now repartition \( R_0 \) at \( x = 26 \) into a left half which is partitioned further at \( y = 21 \) into two halves \( T_1 \) and \( T_2 \), and a right half \( T_3 \). In this case, the height of the tree stays the same. \( T_1 \) is split into a left half \( T_4 \) at \( x = 22 \) which is similar to \( R_4 \), and a right half which is partitioned further at \( y = 32 \) into the two halves \( R_{61} \) and \( R_{62} \). \( T_2 \) is split at \( x = 22 \) into a left half \( T_5 \) which is similar to \( R_5 \) and a right half \( T_8 \) which is similar to \( R_8 \). \( T_3 \) is split at \( y = 21 \) into a bottom half \( T_9 \) which is similar to \( R_9 \) and a top half which is partitioned further at \( y = 29 \) into the two halves \( R_{71} \) and \( R_{72} \).

Figure 42 shows the result of using the k-d-B-tree re-splitting overflow strategy. Notice that k-d-B-tree re-splitting required that we repartition a number of the original nodes. In particular, \( R_5 \) was split into two parts: \( R_{51} \) which was inserted into \( T_4 \) and \( R_{52} \) which was inserted into \( T_5 \). Similarly, \( R_8 \) was split into two parts: \( R_{81} \) which was inserted into \( T_8 \) and \( R_{82} \) which was inserted into \( T_9 \). In this case, the reinsertion was simple as these nodes were absorbed into the existing nodes as the subobjects that comprised them were identical. In other words, the merging did not cause the merged node to contain more objects and hence it could not overflow thereby guaranteeing no more reinsertion. For example, \( R_{82} \) consists of just a part of \( G \) and thus can be grafted onto the part of \( G \) that comprises \( T_9 \) (the old \( R_9 \)). In general, using k-d-B-tree re-splitting can become quite complex (as is also the case with rotation in conventional B-trees), and thus its use is not recommended.

BEGIN COMMENT Should talk about object-oriented representation where we no longer store the identity of the object with each cell. In other words, we cannot directly look up the location address and give the object associated with it. In this case we also have to make use of the space hierarchy to answer query 2. This
Figure 42: (a) k-d-B-tree resulting from the insertion of rectangle object H into the k-d-B-tree of Figure 38 when using the k-d-B-tree re-splitting overflow strategy, and (b) the spatial extents of the nodes. Notice that only in the leaf nodes are the bounding boxes minimal.

has to be somehow woven into the discussion. END COMMENT

The bounding-box cell-tree pyramid is good for answering query 2. However, it only acts as a partition of space. It does not distinguish between occupied and unoccupied space. Thus in order to determine whether a particular location \(a\) is occupied by one of the objects associated with cell \(c\), we need to check each of the objects associated with \(c\), which can be time-consuming especially if \(M\) is large. We can speed this process by modifying the definition of the bounding-box cell-tree pyramid so that a bounding box is stored in each node \(r\) in the hierarchy, regardless of \(r\)'s depth, that covers the bounding boxes of the cells that comprise \(r\). Thus associated with each node \(r\) is the union of the objects associated with the cells comprising \(r\) as well as a bounding box of the union of their bounding boxes. We term the result a disjoint object pyramid. It should be clear that the depth of the bounding-box cell-tree pyramid is one less than that of the corresponding irregular cell-tree pyramid.

A key difference between the disjoint object pyramid and the irregular cell-tree pyramid, and to a lesser extent the bounding-box cell-tree pyramid, is that the elements of the hierarchy of the disjoint object pyramid are also parts of the bounding boxes of the objects rather than just the cells that make up the objects which is the case for both the bounding-box and irregular cell-tree pyramid. The representation of the disjoint object pyramid, as well as the bounding-box cell-tree pyramid, is also much simpler as they both just decompose the objects until a a criterion involving the number of objects that are present in the block is satisfied rather than one based on the homogeneity of the block. This results in avoiding some of the deeper levels of the hierarchy that are needed in the irregular cell-tree pyramid.

There are many variants of the disjoint object pyramid. They differ according to which of the block decomposition rules described in Section 1.2 is used. They are usually referred to by the general term \(R^+\)-
tree[41, 119, 126] on account of the similarity to the R-tree since they both store a hierarchy of bounding boxes. However, the block decomposition rule is usually left unspecified although a generalized k-d tree block decomposition rule is often suggested. An alternative is not to use any decomposition rule in which case each node is just a collection of blocks as in Figure 6.

$R^+$-trees are built in the same incremental manner as any of the bounding-box cell-tree pyramids that we described (e.g., the k-d-B-tree, etc.). Again, as each additional object $o$ is processed, we insert $o$’s bounding box $b$ into all of the leaf nodes which overlap it. If any of these nodes become too full, then we split these nodes using the appropriate block decomposition rule and determine if the parent is not too full so that it can support the addition of a son. If not, then we recursively apply the same decomposition rule to the parent. The process stops at the root in which case the $R^+$-tree may grow by one level. The difference from the method used in the bounding-box cell-tree pyramids is that we also propagate the minimum bounding box information up the hierarchy. The entire process is analogous to that used in a B-tree upon overflow. The difference is that, at times as we showed in the k-d-B-tree insertion example in Figure 40, the decomposition at a nonleaf node may result in the introduction of a new partition that may force the repartitioning of nodes at deeper levels in the $R^+$-tree.

Figure 43 is an example of one possible $R^+$-tree for the collection of rectangles whose specifications are given in Figure 31. Broken lines denote the bounding boxes corresponding to the leaf nodes, and gray lines denote the bounding boxes corresponding to the subtrees rooted at the nonleaf nodes. In this case, we simply took the k-d-B-tree of Figure 38 and added bounding boxes to the nonleaf nodes. This particular tree is of order (2,3) although in general it is not possible to always guarantee that all nodes will have a minimum of 2 entries. Notice that rectangle D appears in three different nodes, while rectangles A, B, E, and G appear in three different nodes. Of course, other variants are possible since the $R^+$-tree is not unique.

The cell tree of Günther [60, 61] is similar to the $R^+$-tree. The difference is that the nonleaf nodes of the cell tree are convex polyhedra instead of bounding rectangles. The sons of each node, say $P$, form a binary space partition (BSP) [54] of $P$ (see Section ?? of Chapter ??). The cell tree is designed to deal with polyhedral data of arbitrary dimension. As in the $R^+$-tree, the polyhedral data that is being represented may be stored in more than one node.

$R^+$-tree stuff here to show similarity to cell pyramid in that it is a way to avoid the false hits. So, start the discussion with a motivation being avoiding the false hits. Result is disjoint decomposition. The k-d-B-tree may be a better way to start the discussion. Mention cell tree in passing as an application to polyhedra.
Another alternative to the R-tree in dealing with rectangles is the R\(^+\)-tree [41, 119, 126], an extension of the k-d-B-tree [100] (see Section ?? of Chapter ??). The motivation for the R\(^+\)-tree is to avoid overlap among the bounding rectangles. In particular, all bounding rectangles (i.e., at levels other than the leaf) are nonoverlapping. Thus, each rectangle is associated with all the bounding rectangles that it intersects. The result is that there may be several paths starting at the root to the same rectangle. This will lead to an increase in the height of the tree. However, retrieval time is sped up.

It is interesting to note that the decomposition into blocks induced by the R\(^+\)-tree is similar to the way a region quadtree would be used to represent a collection of rectangles. An extension of the k-d-B-tree, the R\(^+\)-tree has a drawback: B-tree performance guarantees are no longer valid. For example, pages are not guaranteed to be half full without very complicated record insertion and deletion procedures. Nevertheless, empirical tests by Faloutsos, Sellis, and Roussopoulos [41] reveal reasonable behavior in comparison to the conventional R-tree.

These tests were coupled with a limited analysis of the behavior of the two data structures when used to represent one-dimensional intervals of equal length by transforming them to points in two dimensions using representation (2) of Section ?? of Chapter ??.

Exercises

1. We discussed several methods of dealing with an overflowing node in a k-d-B-tree. One way to ensure greater balance in the tree was to partition the overflowing node at a position other than the one that corresponds to the partition at the root of the k-d tree of the underlying space spanned by the node. The drawback of this solution was that we may have to repartition some of the nodes at deeper levels of the tree. We pointed out that these repartitions may lead to overflow possibly resulting in an infinite loop. Why is this not a problem for point objects?

2. Consider a k-d-B-tree. Give an example partition line and axis whose use to split an overflowing node \( f \) with a capacity \( c \) (i.e., \( f \) has \( c + 1 \) regions) yields two newly-created region pages \( f_1 \) and \( f_2 \) that also overflow.

3. Compare the amount of information that must be stored in k-d-B-tree and R\(^+\)-tree nodes. Do they both require the same amount of storage?

4. What are the minimum and maximum heights of an R\(^+\)-tree of order \((m, M)\) with \( N \) objects (not nodes)?

5. Write an algorithm to insert an object into an R\(^+\)-tree assuming a generalized k-d tree block decomposition rule.

6. Analyze the expected cost of inserting an object into an R\(^+\)-tree.

7. Write an algorithm to delete an object from an R\(^+\)-tree assuming a generalized k-d tree block decomposition rule.

8. Write an algorithm to perform a point query in an R\(^+\)-tree.

9. Write an algorithm to perform a window query in an R\(^+\)-tree.
10. Prove that the worst-case execution time of the rectangle intersection problem for \( N \) rectangles using a \( \mathbb{R}^+ \)-tree is \( O(N^2) \).

11. What is the expected cost of the rectangle intersection problem for \( N \) rectangles using a \( \mathbb{R}^+ \)-tree?

2 Boundary-based Representations

Boundary-based representations are more amenable to the calculation of properties pertaining to shape (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 orthogonal to the coordinate axes. 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 location of the different boundary elements associated with each cell of each object and their nature (i.e., their orientation and the locations of the cells to which they are adjacent). For example, in two dimensions, the boundary elements are just 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 a direction equal to the 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 done by interior-based representations. In two dimensions, the aggregation yields boundary elements which are vectors whose length can be more than one.

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 representations 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 in which case the boundary elements are vectors. The location of the 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 the vector \( v_j \) corresponding to \( e_{i,j} \) is the start vertex of the 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 orthogonal.

In two dimensions, the most general example of a nonorthogonal object boundary is the curvilinear line segment. Straight line segments with an arbitrary slope are less general. Curvilinear line segments are 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 [51] in which case the slopes are restricted to the 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 orthogonal and parallel to the coordinate axes. The chain code is discussed in more detail in Section 3.

In dimensions higher than two, the relationship between the boundary elements associated with a particular object is more complex, as is its expression. Whereas in two dimensions we just have one type of a boundary element (i.e., an edge or a vector consisting of two vertices), in \( d > 2 \) dimensions, given our orthogonality and hyper-planarity constraints, 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
only be adjacent to two other boundary elements. Thus consecutive boundary elements in the representation are implicitly connected. This is not possible in $d > 2$ dimensions as, assuming orthogonal objects comprised of unit-size $d$-dimensional cells, there are $2^{d-1}$ different adjacencies (i.e., types of connectivities) per boundary element. Therefore, it is difficult to adapt the polygon representation to data of dimensionality greater than two. 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 the higher dimensions we do have a choice between an explicit and an implicit boundary-based representation. The boundary model (also known as BRep [13, 108]) is an example of an explicit boundary-based representation. 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. 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 $o$. Although this representation is quite general, it is easy to constrain it to handle orthogonal objects. We do not discuss this further here. For more details on the boundary model, see Section ?? of Chapter ??.

Constructive Solid Geometry (CSG) [99] 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). When the primitive instances 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 boundaries 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 $o$. Again, although these representations are quite general, it is easy to constrain them to handle orthogonal objects.

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., [46]) 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 their relationship 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 that obtained by using a quadtree, an octree, as well as variants of k-d trees, and then imposing an appropriate access structure on the blocks. Such methods are reviewed in Section 2.1. The second is 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 structures on the bounding boxes. They are presented in Section 2.2. Not surprisingly, the distinction between these representations is the same as that made earlier between image-based and object-based methods, respectively.

The boundary representations discussed above and in Sections 2.1 and 2.2 assume that the boundaries of the objects lie on hyperplanes. This is not the case in many applications. In particular, often the boundary is a surface that is highly irregular. The key feature is that we are dealing with a part of the boundary of the object which is like a 2.5-dimensional image such as terrain. This is the subject of Section 2.3.
2.1 Image-based Boundary Representations

The simplest decomposition into blocks is obtained by using a region quadtree decomposition of the underlying space. Associated with each block $b$ is the relevant information about the portion of the boundary $e$ of the objects of which $b$ is a part or the portion of the boundary $e$ of the environment (i.e., the underlying space) in which the objects are embedded that is shared by $b$. This adaptation of the region quadtree serves as the basis of the line quadtree [114, 115].

The line quadtree partitions the underlying space via a recursive decomposition technique that successively subdivides the space until obtaining blocks (possibly single pixels) that have no boundary elements passing through their interior. With each block, a code is associated that indicates which of its four sides forms a boundary (not a partial boundary) of any single object (including the underlying space). Thus, instead of distinguishing blocks on the basis of whether they are part of the background and on the identity of the object in which they lie, we use the nature of the boundary information.

For example, Figure 44b is the block decomposition corresponding to the space partition given in Figure 44a. Notice the use of heavy lines to denote the existence of a complete border on an edge $e$ of a particular block $b$. The absence of a heavy line means that the edge $e$ of $b$ is not completely on the border of a block. In particular, $e$ can be partially on the border. The extent of its presence on a border is determined by examining its neighbor $c$ along the edge. In particular, if $c$ along $e$ is of equal size to that of a side of $b$, then $e$ is not an edge of an object or of the underlying space.

![Figure 44](image)

Figure 44: (a) Example partition of the plane into 6 objects (i.e., regions), (b) the block decomposition corresponding to its line quadtree, and (c) the tree access structure with blocks for the nonleaf nodes instead of the actual tree. All heavy lines indicate that an edge of the block is on the border of an object or the underlying space.

A number of access structures are possible. Using a tree access structure with fanout 4 is particularly attractive as we can store the boundary information in a hierarchical manner (i.e., in the nonleaf nodes). In essence, wherever a nonleaf node does not form a T-junction with any of the boundaries of its descendants along a particular edge $e$, then $e$ is marked as being adjacent to a boundary by use of a heavy line. Figure 44c shows the blocks corresponding to all of the nodes (i.e., both leaf and nonleaf) in the tree access structure with fanout 4 for the line quadtree in Figure 44a. Notice the use of heavy lines to denote the edges that are along the boundary including that of the underlying space.

In most applications, the stipulation that the objects and their environment are composed of a collection of cells of uniform size and shape (i.e., pixels and voxels) is overly restrictive and hence we waive it in the rest of this discussion. Moreover (unlike Section 1), unless otherwise stated, we no longer require that the boundaries of the objects (i.e., edges and faces in two and three dimensions, respectively) are orthogonal to each other and parallel to the coordinate axes.

In the rest of this section, unless specifically stated otherwise, we assume two-dimensional data. Instead
of objects, we assume that the data is a collection of connected straight line segments that partition the underlying space into a collection of polygons termed a \textit{polygonal map} (or \textit{map} for short). Below, we discuss a number of quadtree-like decompositions of the underlying space into 4 congruent blocks. Of course, other decompositions are possible (e.g., into noncongruent blocks as well as a varying number of blocks), but, in the interest of space, we do not discuss them further here. In addition, we do not dwell on the nature of the access structures as the options are the same as those described in Section 1.2.

Note that the line quadtree is also applicable to polygonal maps. In this case, the line segments are orthogonal and parallel to the coordinate axes. In fact, the line quadtree is really an approximation of an arbitrary polygonal map where the edges of the polygonal map are approximated by lines in the 4 principal directions. An alternative approximation is provided by the \textit{MX quadtree} \cite{69,70}. In this case, the edges of the polygonal map are approximated by the pixels through which they pass.

For two-dimensional data such as a polygonal map, the MX quadtree is built by digitizing the line segments and labeling each unit-sized cell (i.e., pixel) through which it passes as of type \textit{boundary}. The remaining pixels are marked \textsc{white} and are merged, if possible, into larger and larger quadtree blocks as done in the region quadtree. Figure 45b is the MX quadtree for the collection of line segment objects in Figure 45a. A drawback of the MX quadtree is that it associates a thickness with a line. Also, it is difficult to detect the presence of a vertex whenever five or more line segments meet.

![Figure 45](image)

\textbf{Figure 45:} (a) Collection of line segments, (b) its MX quadtree, and (c) its edge quadtree.

The MX quadtree has the property (known as the \textit{Quadtree Complexity Theorem} \cite{69}) that the number of blocks needed to represent a simple polygon (i.e., with nonintersecting edges and without holes) is $O(p + q)$ for a $2^q \times 2^q$ image with perimeter $p$ measured in pixel-widths. In all but the most pathological cases (e.g., a small square of unit width centered in a large image), the $q$ factor is negligible and thus the number of blocks is $O(p)$.

The Quadtree Complexity Theorem also holds for three-dimensional data \cite{84} (i.e., represented by a MX 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 these objects. These results also hold for the region quadtree and region octree. The most important consequence of the Quadtree Complexity Theorem is that it means that most algorithms that execute on a quadtree representation of an image (with an appropriate access structure), instead of one that simply imposes an array access structure on the original collection of cells, usually have an execution time that is proportional to the number of blocks in the image rather than the number of unit-sized cells.

In its most general case, the Quadtree Complexity Theorem means that the use of the quadtree representation, with an appropriate access structure, in solving a problem in $d$-dimensional space will lead 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, quadtrees and octrees act like dimension-reducing devices.

The edge quadtree \([122, 136]\) is a refinement of the MX quadtree based on the observation that the number of blocks in the decomposition can be reduced by terminating the subdivision whenever the block contains a single curve that can be approximated by a single straight line. Thus it is defined for collections of line segments that are curves as well as straight lines. For example, Figure 45c is the edge quadtree for the collection of line segment objects in Figure 45a. Applying this process leads to quadtrees in which long edges are represented by large blocks or a sequence of large blocks. However, small blocks are required in the vicinity of corners or intersecting edges. Of course, many blocks will contain no edge information at all.

The PM quadtree family \([88, 116]\) (see also edge-EXCELL \([128]\)) represents an attempt to overcome some of the problems associated with the edge quadtree in the representation of a polygonal map. In particular, the edge quadtree is an approximation because vertices are represented by pixels. Also, when the lines are curves, the decomposition stops when they can be approximated by a single straight line for a predefined tolerance. There are a number of variants of the PM quadtree. These variants are either vertex-based or 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 obtaining a subset that is sufficiently simple so that it can be organized by some other data structure.

The PM\(_1\), PM\(_2\), and PM\(_3\) quadtrees \([116]\) are vertex-based. The PM\(_1\) quadtree is the simplest variant. 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. The fact that the structure is defined in a top-down manner means that each block is maximal. It should be clear that each block contains at most one vertex. For example, Figure 46a is the PM\(_1\) quadtree corresponding to the collection of line segment objects in Figure 45a.

![Figure 46](image)

Figure 46: (a) PM\(_1\) quadtree, (b) PM\(_2\) quadtree, and (c) PM\(_3\) quadtree for the collection of line segments of Figure 45a.

The number of blocks in a PM\(_1\) quadtree depends on the following three factors:

1. The minimum separation between two vertices.
2. The minimum separation between a vertex and a line segment.
3. The minimum separation between two nonintersecting line segments in a block that contains no vertices.

Given a polygonal map whose vertices are drawn from a grid (say \(2^m \times 2^m\)), and where line segments are not permitted to intersect at points other than the grid points (i.e., vertices), it can be shown that these factors imply that the maximum depth of any leaf node in a tree access structure for the PM\(_1\) quadtree is bounded from above by \(4m + 1\) \([112]\). This is useful if we want to represent the collection of blocks using one of the alternative access structures to a tree that were discussed in Section 1.2. Recall that 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.). The depth bound enables a determination of the maximum amount of storage that will be necessary for the locational code of each block.

The amount of storage necessary to represent the PM₁ quadtree can be reduced by removing some of the factors on which its depth depends. For example, we can remove the dependence on the minimum separation between two nonintersecting line segments in a block that contains no vertices by modifying the definition of the PM₁ quadtree so that a block is not split if all of the line segments that it contains intersect at a vertex outside of the block. In other words, we modify the decomposition rule so 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 regardless of its location. The result is termed a PM₂ quadtree and is illustrated in Figure 46b.

The amount of storage necessary to represent the PM₁ quadtree can be further reduced by removing the dependence on the minimum separation between a vertex and a line segment, so that partitioning only occurs when a block contains more than one vertex. The result is termed a PM₃ quadtree and is illustrated in Figure 46c. The PM₃ quadtree is the same as a PR quadtree (see Section ?? of Chapter ??) which is a representation for point data. As we see, the line segments play no role in the decomposition process. The PM₃ quadtree shows why we say that the PM₁, PM₂, and PM₃ quadtrees are vertex-based.

The PM₁ quadtree has been adapted to higher dimensions. In particular is has been used for three-dimensional images (e.g., [8] and the references cited in [108]) containing polyhedra. The decomposition criteria are such that no block 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 region octree.

The PM₁ quadtree also serves as the basis of the PM-CSG tree [139]. In this case the decomposition criteria are such that no block contains more than one CSG primitive. This representation does not handle a pure CSG representation as it does not permit a full complement of CSG operations. Instead, it uses a subset consisting of union and subtraction of sets (i.e., set difference). The set intersection operation is not allowed as in this case its constituent primitives could not be separated so that each one is in only one block. Moreover, the set union operation is not general in that it requires that the two operand sets be disjoint. Thus it is more like a set addition operation. An alternative physical analogy is the the operations of set addition and subtraction correspond to gluing and cutting, respectively. For more details, see [107, Section 5.5.2].

As mentioned above, the different variants of the PM quadtree that we discussed were all vertex-based. They differed in the degree to which the edges played in the decomposition process from some (PM₁), to little (PM₂), to none (PM₃). A drastic alternative is to remove completely the dependence of the decomposition on the vertices and just take into account the edges that make up the polygonal map. As we saw in the PM quadtrees, it is impossible for all of the blocks to contain just one line segment. Thus one approach to defining an edge-based decomposition is to stipulate that each block is split until it contains no more that $T$ line segments. In this case, $T$ acts like a bucket capacity and we term the result a bucket PM quadtree. The drawback of such a representation is that the decomposition does not stop in the neighborhood of a vertex $v$ with degree greater than $T$ (i.e., more than $T$ edges are incident at $v$).

There are a number of possible options. The first is analogous to ignoring the problem by setting an upper limit on the level of decomposition. The second option is to formulate bucket-like analogs of the vertex-based PM quadtrees where the bucketing condition involves the number of line segments that pass through the block. However, this is not as simple as it appears. The problem is that if we add the proviso that a block is not split if it has $T$ or less line segments, then we may violate the vertex condition. For example, a natural bucket analog of the PM₁ quadtree is one that decomposes the underlying space until each block $b$ contains no more than $T$ line segments or or if all of the line segments in $b$ meet at the same vertex $v$ which is in $b$. The bucket analog of the PM₂ quadtree would be defined similarly with the added proviso that block $b$ can contain more than $T$ line segments if they all meet at the same vertex $v$ and $v$ need not be in $b$. There is no logical bucket analog of the PM₃ quadtree as the number of line segments in the block plays no role in the decomposition process.

The shortcoming of these bucket decomposition rules is that even though a block $b$ may contain several vertices, no decomposition takes place when $b$ has $T$ or less line segments. Thus the bucket analogs must be
defined more directly in terms of the number of vertices that they permit. In other words, the primary condition is one that limits the number of vertices in each block with the bucket capacity, in terms of the number of line segments, being a secondary condition.

Using such a condition priority, we define the bucket-like analogs of the PM\textsubscript{1} and PM\textsubscript{2} quadtrees as follows. A bucket PM\textsubscript{1} quadtree contains at most one vertex \( v \) per block and no line segments that do not meet at \( v \). Each block that contains no vertices can contain at most \( T \) line segments. A bucket PM\textsubscript{2} quadtree is defined using a similar rule with the added proviso that a block \( b \) that contains no vertices can contain more than \( T \) line segments if they all meet at the same vertex \( v \) where \( v \) is not in \( b \). It is meaningless to define a bucket-like analog of the PM\textsubscript{3} quadtree as the number of line segments does not enter into the PM\textsubscript{3} quadtree decomposition rule and thus we do not do so.

The third option is similar to the first option in the sense that it is motivated by the rationale that since an infinite level of decomposition is possible, why not decompose the bucket PM quadtree just once each time an edge is inserted into a block that contains more than \( T \) line segments? Of course, the tree will still have a maximum level of decomposition; however, it will only be attained if there is really much insertion activity in the blocks that are being repeatedly decomposed rather than just because we happened to insert a line segment having a vertex whose degree exceeds \( T \). We say that the decomposition is based on a probabilistic splitting rule. We term the result a PMR quadtree \cite{[88]}, an edge-based variant of the PM quadtree.

It should be clear that a PMR quadtree block is permitted to contain a variable number of line segments. The PMR quadtree is constructed by inserting the line segments one-by-one into an initially empty structure consisting of one block. Each line segment is inserted into all the blocks that it intersects or occupies in its entirety. During this process, the occupancy of each block 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, then the block is split \textit{once}, and only once, into four blocks of equal size.

Figure 47\textsubscript{e} is the PMR quadtree for the collection of line segment objects in Figure 45\textsubscript{a} with a splitting threshold value of 2. The 9 line segments, labeled \( a \)–\( i \), are inserted in alphabetic order. It should be clear that the shape of the PMR quadtree for a given polygonal map is not unique; instead it depends on the order in which the line segments are inserted into it. In contrast, the shapes of the PM\textsubscript{1}, PM\textsubscript{2}, and PM\textsubscript{3} quadtrees, as well as their bucket analogs, are unique. Figure 47\textsubscript{a}–\textsubscript{e} shows some of the steps in the process of building the PMR quadtree of Figure 47\textsubscript{e}. In each part of Figure 47\textsubscript{a}–\textsubscript{e}, the line segment that caused the subdivision is denoted by a thick line, while the gray regions indicate the blocks where a subdivision has taken place.

The insertion of line segments \( c \), \( e \), \( g \), \( h \), and \( i \) cause the subdivisions in parts \( a \), \( b \), \( c \), \( d \), and \( e \), respectively, of Figure 47. The insertion of line segment \( i \) causes three blocks to be subdivided (i.e., the SE block in the SW quadrant, the SE quadrant, and the SW block in the NE quadrant). The final result is shown in Figure 47\textsubscript{e}. Note the difference from the PM\textsubscript{1} quadtree in Figure 46 — that is, the NE block of the SW quadrant is decomposed.
in the PM\textsubscript{1} quadtree while the SE block of the SW quadrant is not decomposed in the PM\textsubscript{1} quadtree.

A line segment is deleted from a PMR quadtree by removing it from all the blocks that it intersects. During this process, the occupancy of the block and its siblings is checked to see if the deletion causes the total number of line segments in them to be less than the predetermined splitting threshold. If the splitting threshold exceeds the occupancy of the block and its siblings, then they are merged and the merging process is reapplied to the resulting block and its siblings. Notice the asymmetry between the splitting and merging rules.

Notice that the splitting threshold is a different concept from the bucket capacity used in the definition of the bucket PM\textsubscript{1} and PM\textsubscript{2} quadtrees although it plays the same role. Thus it should be clear that the number of line segments in a PMR quadtree block can exceed the value of the splitting threshold. In fact, it can be shown [108, Exercise 4.75, p. 268] that the maximum number of line segments in a PMR quadtree block is bounded by the sum of the splitting threshold and the depth of the block (i.e., the number of times the original space has been decomposed to yield this block). Interestingly, the number of line segments in a quadtree block can also exceed the bucket capacity $T$ for the PM\textsubscript{3} quadtree and for both the PM\textsubscript{1} and PM\textsubscript{2} quadtrees, as well as their bucket analogs since the presence of a vertex in a block $b$ results in the possibility that $b$ (as well as other blocks for the PM\textsubscript{2} quadtree) contains more than $T$ line segments.

Once the decomposition into blocks has been obtained, we also need to decide how to organize the line segments that pass through the block. This is a relatively difficult issue to which not much attention has been paid in the past. However, the right choice has a dramatic effect on the performance of geometric operations as the blocks are often treated as buckets of nontrivial sizes (e.g., 32, 64, and even higher values). This is especially the case when the buckets take on the role of disk pages. Thus the buckets provide an initial sort of the underlying space. However, once we access the appropriate bucket we don’t want to have to execute a sequential search to find the relevant data when the bucket can contain many elements. For example, some possibilities for line segments include:

1. Sort the lines on the basis of their nearness to one of the axes.
2. Sort the lines that meet at a vertex in clockwise or counterclockwise order.
3. Represent the portion of the line that passes through the block by its midpoint and store it in a point representation such as a PR quadtree.

The PMR quadtree is very good for answering queries such as finding the nearest line to a given point [66] (see [67] for an empirical comparison with hierarchical object representations such as the R-tree and R$^+$-tree). It is preferred over the PM\textsubscript{1} quadtree (as well as the PM\textsubscript{2}, PM\textsubscript{3}, MX, and edge quadtrees) as 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\textsubscript{1} quadtree. This is important since four blocks are created at each subdivision step. Thus when many subdivision steps that occur in the PM\textsubscript{1} quadtree result in creating many empty blocks, the storage requirements of the PM\textsubscript{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 will increase.

Using a random image model and geometric probability, it has been shown [83], theoretically and empirically using both random and real map data, that for sufficiently high values of the splitting threshold (i.e., $\geq 4$), the number of blocks in a PMR quadtree is asymptotically proportional to the number of line segments and is independent of the maximum level of decomposition. In contrast, using the same model, the number of nodes in the PM\textsubscript{1} quadtree is product of the number of lines and the maximal level of decomposition (i.e., $n$ for a $2^n \times 2^n$ image). The same experiments and analysis for the MX quadtree confirmed the results predicted by the Quadtree Complexity Theorem which is that the number of blocks is proportional to the total length of the line segments.

All variants of the the PM quadtree (including the PMR quadtree and the three-dimensional variants) are characterized as employing spatial indexing because with each block the only information that is stored is
whether or not the block is occupied by the object or part of the object that is stored there. This information is usually in the form of a pointer to a descriptor of the object. For example, in the case of a collection of line segment objects in the PMR quadtree of Figure 47, the shaded blocks only record the fact that the line segments (i.e., a, b, and e) cross it or pass through it. The part of the line segment that passes through the block (or terminates within it) is termed a q-edge. Each q-edge in the block is represented by a pointer to a record containing the endpoints of the line segment of which the q-edge is a part [88]. This pointer is really nothing more than a spatial index and hence the use of this term to characterize this approach. Thus no information is associated with the shaded block as to what part of the line (i.e., q-edge) crosses it. This information can be obtained by clipping [47] the original line segment to the block. This is important for often the precision necessary to compute these intersection points is not available.

**Exercises**

In Exercises 1—4 assume the use of the vertex bucket polygon quadtree defined in section 1.3. In this case, decomposition halts whenever a block is a part of no more than $T$ polygons or contains just one vertex.

1. Consider a vertex bucket polygon quadtree with $T = 2$. Prove or disprove that the vertex bucket polygon quadtree decomposition is the same as one of the vertex-based PM quadtree decomposition rules for line segments.

2. Prove or disprove that the vertex bucket polygon quadtree decomposition rule yields the same decomposition as a PM$_N$ quadtree decomposition rule for line segments.

3. Prove or disprove that the vertex bucket polygon quadtree decomposition rule yields the same decomposition as a bucket PM$_1$ quadtree decomposition rule for line segments.

4. Prove or disprove that the vertex bucket polygon quadtree decomposition rule yields the same decomposition as a bucket PM$_2$ quadtree decomposition rule for line segments.

5. Do polygonal maps that correspond to Delaunay triangulations (e.g., [137]) behave better in the sense that their quadtree decompositions require less blocks than the quadtree decompositions of other triangulations of the same set of points? In other words, do they have some properties that avoid some of the bad degenerate cases of many decompositions? For example, the Delaunay triangulation of a given set of points satisfies the minimum angle property which means that it maximizes the minimum interior angle in the entire triangulation thereby avoiding the presence of long and thin triangles. Such triangles are poor for interpolating values of points that are not vertices of triangles since the vertices of the containing triangle may not necessarily be the closest ones to the point being interpolated. This property can be stated more formally as follows: The Delaunay triangulation produces the lexicographically largest possible nondecreasing sequence of angles in any triangulation of a given set of points. Thus if $a_1 \leq a_2 \leq \cdots \leq a_i \leq \cdots$ is the sequence of angles in a Delaunay triangulation and if $b_1 \leq b_2 \leq \cdots \leq b_i \leq \cdots$ is the sequence of angles in another triangulation of the same set of points, then there exists an $i$ such that $a_i > b_i$. Consider the various PM quadtree decompositions based on line segments as well as the polygon decomposition rule that stipulates that decomposition halts whenever a block is a part of no more than $T$ polygons or contains just one vertex (i.e., the vertex bucket polygon quadtree discussed in Section 1.3).

6. What are some issues in the use of PM quadtrees to represent collections of line segments in threedimensional space? In other words, can you use, for example, a PM$_1$ quadtree to represent a collection of line segments in three-dimensional space where we recursively decompose the underlying space until each block contains at most one line segment or all of the line segments meet at a vertex within the block?
2.2 Object-based Boundary Representations

In Sections 1.4.2 and 1.4.3 we already examined two hierarchical representations (i.e., the R-tree and the \( R^+ \)-tree) that propagate object approximations in the form of bounding boxes. In both of these cases, query 2 is facilitated by imposing a tree access structure on the elements of the hierarchy. In this section we review a number of related hierarchical representations of the boundaries of object. All of these representations are also accompanied by a tree access structure so that query 2 can be executed efficiently. Of course, other access structures could also be used but they are not discussed here.

When using the R-tree and \( R^+ \)-tree as an object approximation, the sides of the bounding boxes must be parallel to the coordinate axes of the space from which the objects are drawn. In contrast, the strip tree is a hierarchical representation of a single curve that successively approximates segments of it with bounding boxes that does 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.

![Figure 48: A curve and its decomposition into strips.](image)

The strip tree data structure is supplemented by an access structure in the form of a binary tree whose root represents the bounding rectangle of the entire curve. For example, consider Figure 48 where the curve between points \( P \) and \( Q \), at locations \((x_P, y_P)\) and \((x_Q, y_Q)\), respectively, is modeled by a strip tree. The rectangle associated with the root, \( A \) in this example, corresponds to a rectangular strip, that encloses the curve, whose sides are parallel to the line joining the endpoints of the curve (i.e., \( P \) and \( Q \)). The curve is then partitioned in two at one of the locations where it touches the bounding rectangle (these are not tangent points as the curve only needs to be continuous; it need not be differentiable). Each subcurve is then surrounded by a bounding rectangle and the partitioning process is applied recursively. This process stops when the width of each strip is less than a predetermined value. Figure 49 shows the binary tree corresponding to the decomposition into strips in Figure 48a.

![Figure 49: Strip tree corresponding to Figure 48.](image)

Figure 48 is a relatively simple example. In order to be able to cope with more complex curves such as those that arise in the case of objects rather than curves, the relationship between the bounding box and the underlying curve needs some further elaboration. In particular, closed curves and curves that extend past their endpoints require some special treatment. The general idea is that these curves are enclosed by boxes which are split into two rectangular strips, and from now on the strip tree is used as before. For example, see Figure 50.

For a related approach, see the arc tree [62]. The arc tree starts out by decomposing the curve into segments of equal length thereby forming the deepest level in the hierarchy. Successive levels of the hierarchy are built by aggregating pairs, pairs of pairs, etc. An access structure in the form of a complete binary tree (i.e., a one-
Figure 50: Strip tree representation of (a) a closed curve, and (b) a curve that extends past its endpoints.

dimensional pyramid) is imposed on the hierarchy. Thus the result is analogous to imposing a one-dimensional uniform grid on the curve where the width of each one-dimensional grid cell (i.e., interval) is based on the total length of the curve.

Basing the aggregation on curve length means that closed curves need no special treatment when represented by an arc tree. Each subarc is approximated by a bounding ellipse instead of a bounding box. The foci of the ellipses are placed at the endpoints of each subarc and the principal axis is as long as the subarc. This means that all subarcs lie completely within each ellipse thereby obviating the need for special treatment for subarcs that extend past their endpoints. The drawback of the arc tree is that we need to be able to compute the length of an arc which may be quite complex (e.g., if we have a closed form for the curve, then we need an elliptical integral).

The manner in which we described the strip tree leads to it being characterized as a top-down approach to curve approximation. The strip tree can also be built by a bottom-up process where the curve is assumed to be represented by a sequence of points. The construction process simply pairs up successive points and forms their strips which are bounding boxes. It then pairs up the strips forming their strips which are also bounding boxes. This process is applied repeatedly until obtaining one strip. It is interesting to note that the resulting approximations are not as good as those obtained by the top-down method.

Burton [21] defines a structure termed BS PR (Binary Searchable Polygonal Representation) which is closely related to the strip tree obtained by the bottom-up construction process in the sense that the BS PR is also a bottom-up approach to curve approximation. Once again, the primitive unit of approximation is a box; however, unlike the strip tree, in the case of the BS PR all boxes are upright (i.e., they have a single orientation).

The curve to be approximated by the BS PR is decomposed into a set of simple sections where each simple section corresponds to a segment of the curve which is monotonic in both the \( x \) and \( y \) coordinate values of the points comprising it. The tree is built by combining pairs of adjacent simple sections to yield compound sections. This process is repeatedly applied until the entire curve is approximated by one compound section. Thus, we see that leaf nodes correspond to simple sections and nonleaf nodes correspond to compound sections. For a curve with \( 2^n \) simple sections, the corresponding BS PR has \( n \) levels. Thus we see that unlike the strip tree which is constructed in a top-down manner, the BS PR does not allow for a variation in the approximation at the lowest level since the initial bounding boxes are determined by the simple sections (i.e., the monotonic curves).

As an example of a BS PR, consider the regular octagon in Figure 51a with vertices \( A - H \). It can be decomposed into four simple sections: \( ABCD, DEF, GHI, \) and \( HA \). Figure 51b shows a level 1 approximation to the four simple sections consisting of rectangles \( AIDN, DJFN, HMFK, \) and \( AMHL \), respectively. Pairing up adjacent simple sections yields compound sections \( AIJF \) corresponding to \( AIDN \) and \( DJFN \), and \( AFKL \) corresponding to \( HMFK \) and \( AMHL \) (see Figure 51c). More pairing yields the rectangle for compound section \( IJKL \) (see Figure 51d). The resulting BS PR tree is shown in Figure 51e. Using the BS PR, Burton shows how to perform point in polygon determination and polygon intersection. These operations are implemented by tree searches and splitting operations.

Strip trees are useful in applications that involve search and set operations. For example, suppose that we
wish to determine whether a road crosses a river. Using a strip tree representation for these features, answering this query requires that we perform an intersection of the corresponding strip trees. Three cases are possible as is shown in Figure 52. Figures 52a and 52b correspond to the answers NO and YES, respectively, while Figure 52c requires us to descend further down the strip tree. Notice the distinction between the task of detecting the possibility of an intersection and the task of computing the actual intersection, if one exists. The strip tree is well-suited to the former task. Other operations that can be performed efficiently by using the strip tree data structure include the computation of the length of a curve, areas of closed curves, intersection of curves with areas, point membership, etc.

The strip tree can also be generalized to represent boundaries of three-dimensional objects (i.e., their surface). A good example of such a generalization is the prism tree of Ponce and Faugeras [96]. It is a hierarchical representation of an approximation that is used to model the surfaces of three-dimensional objects that are polyhedra of genus 0 (i.e., they have no holes). The prism tree is best understood by examining its predecessor, termed a polyhedral approximation [42].

To simplify the discussion, consider the construction of a polyhedral approximation of a two-dimensional object as in Figure 53a. The object is initially modeled by a triangle with vertices that lie on the object’s boundary. Assume for this example that the object is convex. Thus, no part of the triangle lies outside the object. For each side of the triangle, say $E$, find a point on the object’s boundary, say $M$, whose distance
from $E$ is a local maximum. If the distance from $M$ to $E$ is greater than a predefined threshold, say $\epsilon$, then repeat this step for the two new sides of the triangle formed by $M$ and the endpoints of $E$. This process is repeated for all the new triangles until all of the approximations represented by their sides are within $\epsilon$, or the number of decomposition steps exceeds a predefined value. For example, Figure 53b is a possible polyhedral approximation for the object in Figure 53a.

![Figure 53: A two-dimensional prism tree: (a) object, (b) polyhedral approximation, (c) tree representation, (d) prism tree.](image)

The polyhedral approximation provides a hierarchical representation of the boundary. The results of the process of successive subdivision of the boundary can be used by imposing a tree access structure on it where, with the exception of the root node, each node represents a straight line and the associated segment of the boundary. For example, Figure 53c is the tree structure corresponding to the polyhedral approximation of Figure 53b. Notice that the root has three sons while all remaining nodes have two sons. Often, the exact description of the segment of the boundary corresponding to the node is difficult to manipulate (e.g., detecting intersections between boundaries of adjacent objects). Hence, with each node we also associate a quadrilateral approximation of the corresponding boundary segment. The resulting approximation is the two-dimensional version of the prism tree.

For example, Figure 53d is the two-dimensional prism tree corresponding to the polyhedral approximation given in Figure 53b. Notice that the shape of a trapezoid for a given boundary segment, say $S$, depends on the lines that approximate the segments that are adjacent to $S$. To model three-dimensional objects, the polyhedral approximation uses tetrahedra and prisms (actually truncated pyramids) instead of triangles and quadrilaterals, respectively.

### 2.3 Surface-based Boundary Representations

Although the prism tree definition is in terms of polyhedral objects, it can also be used to represent objects whose surfaces need not lie on hyperplanes. We restrict ourselves to three-dimensional objects. The prism tree represents the entire surface of an object. However, in many applications, we are just interested in representing an arbitrary surface rather than an entire object (i.e., a partial boundary). In this case, we are really dealing with a 2.5-dimensional image — that is, for each pair $(x,y)$ corresponding to the projection of the surface on the $x$-$y$ plane, there corresponds a unique value of $z$ (also known as an elevation value). In the rest of this section, we give a brief overview of hierarchical representations of 2.5-dimensional images in the context of processing topographic data (i.e., terrains). The data consists of elevation values sampled at various locations on the plane.

The goal is to reconstruct the surface. This reconstruction is facilitated by choosing a sampling process and representation that adapts to changes in the terrain. The most common way of sampling the data is to use a uniformly-spaced rectangular or triangular grid (known as a gridded digital terrain model or DTM). In this case, we do not have to know where the special features are as the sampling process is given a resolution which ensures that any point on the surface is within a bounded distance of a sample point. An alternative method, which is more compact, is capable of capturing point and line features (e.g., peaks, pits, passes, ridges,
and valleys) exactly by explicitly recording the elevations of the points corresponding to their location on the surface.

In both cases, the actual surface can be approximated by a network of planar nonoverlapping triangles. Interpolation is used to obtain the elevation at nonsample locations. In the former, all triangles are of equal size and hence the interpolation uses the nearest samples. The drawback of this method is that some maxima and minima will be missed if they do not occur at sample points. In the latter, known as a Triangular Irregular Network (TIN) [95], there is a high likelihood that the triangles are thin and elongated. This has the drawback that interpolation of the elevation of the nonsample point \( s \) using the vertices of the triangle containing \( s \) will not necessarily use the closest vertices to \( s \). In other words, it is preferable that the triangles be as equiangular as possible. The Delaunay triangulation [29, 137] is one approach to achieve such a goal.

When the amount of data is large, the triangular network approach becomes unwieldy in terms of storage requirements. In this case, there are two possible solutions. The first is a pyramid-like approach that represents the surface at different predefined levels of precision (i.e., multiple resolution). The second approach, and the one we focus on in this section, represents different parts of the surface at different levels of resolution. Representations based on such an approach are usually characterized as being hierarchical. The hierarchical methods that are commonly used are based on either triangulations or rectangular decompositions.

Hierarchical triangular decomposition methods are differentiated on the basis of whether the decomposition is into three (termed ternary) or four (termed quaternary) sub-triangles. Ternary triangular decompositions are formed by taking an internal point of one of the triangles, say \( T \), and joining it to the vertices of \( T \) (e.g., Figure 54a). Quaternary triangular decompositions are formed by joining three points, each on a different side of a given triangle (e.g., Figure 54b). Hierarchical triangulations are represented by trees where the root corresponds to the initial enclosing rectangle; the out degree is three or four depending on the type of decomposition. In the case of a ternary triangular decomposition, each triangle is adjacent to at most one triangle on each side. In contrast, in the case of a quaternary triangular decomposition, each triangle may be adjacent to a number of triangles along a side.

![Figure 54: Example triangular decompositions: (a) ternary, (b) quaternary.](image)

Hierarchical triangulations result in the approximation of a surface, say \( S \), by planar triangular patches whose vertices are a subset of the data points that define \( S \). For each such patch, an approximation error is computed that is usually the maximum error of the data points with projections on the \( x-y \) plane overlapping the projection of the patch on the \( x-y \) plane. If the approximation error exceeds a predefined tolerance, then the patch is subdivided further. The resulting surface depends on the nature of the decomposition.

In the case of a ternary triangular decomposition, the approximated surface described by the triangulation is usually continuous at every level. However, the triangles are often thin and elongated since the point at which the triangle is decomposed is internal to the triangle. Thus, equiangularity is generally not satisfied. Nevertheless, the elongation can be alleviated, in part, by making use of the triacon hierarchy of Dutton [33] which creates an alternating containment hierarchy so that each level of decomposition is fully contained in the level that was created two steps previously. The field tree [48, 49] treats a quadtree decomposition into squares on the plane in a similar manner.

Ternary triangular decomposition is used to replace triangular faces by tetrahedra in the construction of the
three-dimensional prism tree [42, 96] with a slight variation to correct for edges between adjacent triangles for which the deviation from the actual surface is too large (termed an adjustment step). In particular, a new sample point \( M \) is added so that the space on the surface spanned by two adjacent noncoplanar triangles (e.g., Figure 55a) is now spanned by four adjacent noncoplanar triangles (e.g., Figures 55b and 55c). This adjustment tends to replace a pair of adjacent thin and elongated triangles by four triangles which are closer to being equiangular. There are two choices for the adjustment depending on whether the elevation of \( M \) is higher than that of the vertices of the adjacent noncoplanar triangles (e.g., Figure 55b) or lower (e.g., Figures 55c). The ternary triangular decomposition is usually used when the surface is defined at points that are randomly located. De Floriani, Falcidieno, Nagy, and Pienovi [27, 28] discuss its use for surface interpolation as well as serving as a data compression mechanism.

![Figure 55](Image)

**Figure 55:** Illustration of the adjustment step in the construction of a prism tree that replaces (a) two adjacent triangles whose common edge elevation deviates too much from the actual surface with (b) four triangles thereby deleting the edge.

In the case of a quaternary triangular decomposition, each triangle can be adjacent to a number of triangles on each of its sides. The decomposition may be regular (e.g., [12, 44]) or based on the actual data (e.g., [57]). Thus, the approximated surface defined on it is generally not continuous (see more details below in the discussion of hierarchical rectangular decompositions) unless all of the triangles are uniformly split — that is, the resulting tree is a complete quadtree. If the initial approximating triangle is equilateral and the triangles are always subdivided by connecting the midpoints, then equiangularity holds and the interpolation is ideal. The quaternary triangular decomposition is especially attractive when the data points are drawn from a uniformly-spaced grid.

Hierarchical rectangular decompositions are similar to hierarchical triangulations that are based on a quaternary decomposition. Of course, the difference is that we do not consider a ternary rectangular decomposition. Thus we often drop the qualifier “quaternary” when discussing them. Hierarchical rectangular decompositions are used when the data points are the vertices of a rectangular grid. In this case, a rectangle is split by choosing an internal point and joining it to its projections on the four sides of the rectangle. When the data is uniformly spaced, the result is analogous to a region quadtree.

For example, Carlson [22] subdivides a surface into rectangular patches for use in algorithms that perform intersection of sculptured surfaces. Schmitt, Barsky, and Du [117] use adaptive subdivision for surface fitting in conjunction with a parametric piecewise bicubic Bernstein-Bezier surface. They start with a rough approximating surface and recursively refine it where the data is poorly approximated. Chen and Tobler [24] evaluate the use of different approximations in each rectangular patch in terms of numerical accuracy, execution time, and storage.

Rectangular decompositions have the same continuity problem as the quaternary triangular decompositions. We discuss them together below using the term quaternary decomposition. In particular, in a quaternary decomposition each triangle or rectangle can be adjacent to several triangles or rectangles on each of its sides since the triangles and rectangles are always subdivided by connecting the midpoints of their sides. The disadvantage of quaternary decompositions is that when we apply a planar triangulation to the three-dimensional points (i.e., the vertices with their elevation values) in the triangular decomposition or fit a patch (e.g., by using a ruled surface [46]) to them in the rectangular decomposition, the resulting surface is not continuous (termed the alignment problem) unless all of the triangles or rectangles in the underlying decomposition have been split uniformly — that is, the resulting quadtree (whether triangular or rectangular) is always complete.
Figures 56a and 56b illustrate the alignment problem for a quaternary decomposition on the plane that makes use of triangles and rectangles, respectively. Notice the cracks (shown shaded) which are the result of the discontinuity in the interpolated surfaces corresponding to the triangles and rectangles.

The alignment problem can be resolved for both of the quaternary decompositions. In the case of triangles, one solution is to simply use the interpolated point instead of the true point [11]. For example, in Figure 56a, the elevation of point H used in triangles 2, 3, and HON would be replaced by the interpolated elevation at the midpoint of the edge between B and C. This is undesirable as we are giving up some of the original sample points. The same problem would arise in the case of rectangles where we would replace the elevation of point R in patches 3 and 5 in Figure 56b by the interpolated elevation at the midpoint of the edge between H and L.

Instead, it is better to decompose triangle ABC, as shown in Figure 57a, into a hierarchy of triangles so that there are triangles in ABC adjacent to edge BC that are of the same size as triangles 1, 2, 3, and 4 in triangle BCF [43]. This is achieved by adding the points p, q, r, s, t, and u, as well as the edges between them and the vertices along edge BC (i.e., B, G, H, I, and C).

The newly-added points are midpoints of known edges or midpoints of newly-created edges lying between endpoints or midpoints of known edges. In the strictest sense, the former do not represent new samples (since they lie on existing edges), while the latter do. In both cases, their elevation values are obtained by applying linear interpolation to the elevation values at the endpoints of these known or newly-created edges. For example, q is obtained by applying linear interpolation to the elevations at A and B. In contrast, both u and t are obtained by applying linear interpolation to the interpolated elevation just computed at q and the elevations at B and H, respectively.

Observe that a similar process must be applied to triangle HON in Figure 57a so that there are triangles in HON adjacent to edges HO and HN that are of the same size as triangles 2 and 5 in triangle BOH and triangles 3 and 6 in triangle HNC, respectively. This is achieved by adding the point v as well as the edges between it and vertices L and K of triangles BOH and HNC, respectively.

Unfortunately, we are not yet through in the sense that we may also have to apply the same process to the other edge neighbors of triangle BCF (i.e., the ones adjacent along edges BF and CF). This is the case if their atomic triangles that share edges BM and CJ are larger than triangles 1 and 4, respectively.
Figure 57: Example illustrating how to resolve the alignment problem for a quaternary decomposition corresponding to Figure 56 that makes use of (a) a triangular decomposition, and (b) a rectangular decomposition.

A similar technique can be applied to a quaternary rectangular decomposition thereby obtaining a hierarchy of rectangles so that there are adjacent rectangles of equal size along the edges (see Figure 57b). For example, in Figure 57b there are rectangles in rectangle IFDG adjacent to edge IF that are the same size as rectangles 2, 5, and 6 in IFDG. As in the quaternary triangular decomposition, a number of new points have been added. We decompose these points into two sets: those that lie on existing edges (e.g., r, t, u, and w) and those that do not (e.g., v, x, y, and z). The former, as in the case of the triangles, do not represent new samples since they are midpoints of known edges. However, their elevation values depend on the type of patch that is being fit through them.

The patch is defined by the four curves that make up its boundary. The simplest patch arises when the curves are linear in which case the resulting patch is a ruled surface. Of course, more complex curves such as quadratics could be used thereby leading to more complex surface patches. When the curves are linear, the elevation values of all newly-added points that lie on existing edges are obtained by applying linear interpolation to the elevation values at the endpoints of the edges for which they serve as midpoints. For example, r is obtained by interpolating the elevations at D and F, while w is obtained by applying linear interpolation to the elevation at F and the interpolated elevation just computed at r.

The newly-added points that do not lie on existing edges represent new samples. Thus we must also obtain an elevation value for them. There are two choices for some of them since they fall at the midpoints of edges between midpoints of both a horizontal and a vertical edge. We choose to take the average of the two interpolated values. For example, v is obtained by taking the average of the values obtained by applying linear interpolation to the elevation values at L and t and the elevation values at u and r. The same technique is applied to obtain the elevation value at z once we have obtained the elevation values at x, y, and z.

At this point, let us briefly compare the two different quaternary decompositions. As we saw, both methods suffer from the alignment problem when we try to fit a surface through their vertices. However, as we demonstrated, this problem can be overcome through the addition of the appropriate triangles and rectangles. For triangles, we obtained a planar surface as we were able to use a planar triangulation. For rectangles, we obtained a nonplanar surface as it is impossible to fit a plane through the four vertices of each rectangle.

It could be argued that a nonplanar surface is a better approximation of a nonplanar surface than a planar one, thereby making the rectangle more attractive than the triangle. Of course, we could also fit a nonplanar
surface through the vertices of the triangles. However, it is well-known that the more sample points that are used in a surface patch, the better is the approximation of the underlying surface. Thus, using such reasoning implies that the rectangle is preferable to the triangle.

An argument could also be made that the planar approximation of the surface provided by the triangle is preferable to the nonplanar approximation provided by the rectangle. This is especially true if ease of interpolation is an issue. In fact, we could also obtain a planar approximation using rectangles by splitting each rectangle into two triangles by adding a diagonal edge between two opposing vertices. There are two choices as shown in Figure 58. As long as the four vertices are not coplanar, one will result in a ridge-like planar surface (Figure 58a), while the other will result in a ridge-like planar surface (Figure 58b). We usually choose the diagonal edge whose angle is the most obtuse.

![Figure 58: Example illustrating the ambiguity that results when each rectangle is split into two triangles by adding a diagonal edge between (a) vertices B and C, and (b) vertices A and D.](image)

Interestingly, the alignment problem when using rectangles can also be overcome by applying a planar triangulation to the result of transforming the underlying rectangular quadtree to a restricted quadtree [135, 123, 124]. The restricted quadtree has the property that all rectangles that are edge neighbors are either of equal size or of ratio 2:1. Given an arbitrary quadtree decomposition, the restricted quadtree is formed by repeatedly subdividing the larger nodes until the 2:1 ratio holds. For example, Figure 59a is the restricted quadtree corresponding to the rectangular decomposition in Figure 56b. Note that the SE quadrant of Figure 56b had to be decomposed once in contrast to twice when building the hierarchy of rectangles in Figure 57b. This method of subdivision is also used in finite element analysis as part of a technique called h-refinement [76] to adaptively refine a mesh that has already been analyzed, as well as to achieve element compatibility.

![Figure 59: (a) Restricted quadtree corresponding to the rectangular decomposition in Figure 56b, and the result of triangulating it using (b) a four-triangle rule as well as (c) an eight-triangle rule.](image)

The alignment problem is overcome by triangulating the rectangles in the restricted quadtree uniformly using one of the following two rules:

1. Decompose each atomic rectangle into four triangles, or one triangle per edge, unless the edge borders a smaller rectangle. In that case, two triangles are formed along the edge (e.g., Figure 59b).
2. Decompose each atomic rectangle into eight triangles, or two triangles per edge, unless the edge borders a smaller rectangle. In that case, two triangles are formed along the edge (e.g., Figure 59c).

The decomposition into eight triangles is preferred [135] as it avoids problems when displaying (i.e., shading [101]) the resulting object (see also [107, Section 7.1.6]).

### 3 Difference-based Compaction Methods

Some interior-based and boundary-based representations (regardless of whether they are implicit or explicit) can be made more compact by making use of an ordering on the data whether it is the locations (i.e., the cells, blocks, boundary elements, etc.) or values that they record (if one exists). In particular, instead of just recording the actual values of the cells and/or the actual locations of the cells, such methods record the change in the values of the cells and/or the locations of the cells where the value changes. The ordering is used to reconstruct the original association of locations with values (i.e., query 1) or values with their locations (i.e., query 2) whenever we want to answer one of our basic queries of determining what locations are associated with a particular feature (i.e., query 1) or what value is associated with a particular location (i.e., query 2). In other words, these compaction methods are not designed to facilitate responding to either of queries 1 or 2.

Naturally, this reconstruction process is costly as it requires traversing the entire representation (i.e., all the objects). However, this is not necessarily a problem when execution of the operations inherently requires that the entire representation be traversed (e.g., Boolean set operations, data structure conversion, erosion, dilation, etc.). On the other hand, operations which require the ability to randomly access a cell in the representation (e.g., neighbor finding) are inefficient in this case unless an additional access structure is imposed on the compact representation. However, note that if the representation records relative changes in the values rather than the actual values, then an access structure may not be of much use since we still need the reconstruction process to determine the value associated with a particular cell.

Runlength encoding [104] is an example of a method that can be applied to make an interior-based representation such as the array more compact. The actual locations of the cells are not recorded. Instead, contiguous identically-valued cells are aggregated into one-dimensional rectangles for which only the value associated with the cells comprising the rectangle and the length of the rectangle are recorded. The one-dimensional rectangles are ordered in raster-scan order which means that row 1 is followed by row 2, etc. The same technique is applicable to higher dimensional data in which case the one-dimensional rectangles are ordered in terms of the rows, planes, etc.

For example, consider the object in Figure 60a which is embedded in an 8 × 8 image array. The cells that comprise the object are shown shaded. Assuming an origin at the upper-left corner and that the image is binary with ‘B’ and ‘W’ denoting foreground and background, respectively, we have a runlength encoding of (W, 8), (W, 8), (W, 4), (B, 4), (W, 4), (B, 4), (W, 3), (B, 5), (W, 2), (B, 6), (W, 2), (B, 4), (W, 2), (B, 3), (W, 3). Given the location of any cell c in one-dimensional rectangle r, the value associated with c can be computed by referring to the location of the starting position of the entire representation and accumulating the lengths of the one-dimensional rectangles encountered before the position of r in the ordering.

Runlength encoding can be made even more compact by just recording the change in the value associated with the cell rather than the value itself. We use the term change-based runlength encoding to describe this method thereby distinguishing it from the previous variant which we will refer to as value-based runlength encoding when the need arises.15 Change-based runlength encoding is especially attractive in the case of a binary image where we have just two possible values (i.e., 0 and 1 or B and W, respectively, in our examples).

In this case, once we know the value associated with the first cell in each row, we don’t need to keep track of the remaining values in the row. Thus we just record the value of the first cell in each row followed by a sequence of numbers corresponding to the lengths of the one-dimensional rectangles of alternating values. For example, given the image in Figure 60a we have the following change-based runlength encoding: (W, 8),
Runlength encoding can also be adapted to deal with rectangular aggregates (i.e., blocks) in higher dimensions. Once again, we assume that the locations of the blocks are not recorded. If the blocks are of a uniform size and orientation, then the adaptation is easy as it is analogous to a grid of cells of uniform size and we apply the same methods as before. If the blocks are not of a uniform size, then the problem is more complex. Recall, that since we do not record the locations of the blocks, we must make use of the ordering of the blocks. Such an ordering exists if there is a regularity to the manner in which the blocks of differing sizes are created. Of course, in such a case we must specify the block sizes. A block decomposition that results from a regular decomposition of space such as a region quadtree, region octree, bintree, etc. has such an ordering.

As an example, consider two-dimensional data represented by a region quadtree (i.e., a regular decomposition into 4 congruent rectangular blocks) where we ignore the imposed access structure (i.e., the tree). Its ordering is obtained in the same manner as the block decomposition — that is, recursively decompose the embedding space into four congruent blocks (labeled NW, NE, SW, SE in this order) until each block is either completely occupied by an object or is empty at which time their size and value are placed in the list. For example, assuming a list of pairs \((a, b)\) where \(a\) and \(b\) correspond to the value and size (i.e., power of two), respectively, of each block, the region quadtree in Figure 60b corresponding to the object in Figure 60a would be represented as \((W, 2), (W, 1), (B, 1), (B, 1), (W, 1), (W, 0), (B, 0), (B, 0), (B, 0), (W, 0), (W, 1)\).

This ordering can also be obtained by associating with each block \(b_i\) a unique base 4 number \(m_i\) formed by interleaving the bits in the binary representation of the \(x\) and \(y\) coordinate values of \(b_i\’s\) upper-left corner (termed a Morton number \([87, 107]\)), assuming an origin at the upper-left corner of the image, and then sorting the \(m_i\) values in increasing order. The resulting order is the same as the Morton order (see Sections 1.1 and 1.2 as well as Figure 2c). We term this representation a Morton size ordering.

Assuming a Morton size ordering representation, given the location of the upper-left corner of any cell \(c\), and the size (must be a power of two) of the space \(S\) in which the objects are embedded, then the value associated with \(c\) can be computed by an accumulation process that starts at the location of the upper-left corner of \(S\). This is done by reconstructing the embedding space using the principle that each time a block of width \(2^m\) is encountered, it must be followed by three more blocks \(\{b_1, b_2, b_3\}\) each of width \(2^m\) or satisfying the constraint that the area spanned by the constituent blocks of \(b_i\ (1 \leq i \leq 3)\) has a total width of \(2^m\). As each block \(b\) of the Morton size ordering is added, we check if \(b\) contains \(c\) and report \(b\’s\) value as that of \(c\) when the containment test is satisfied.

From a theoretical standpoint, other orderings than the Morton order could be applied to the blocks (e.g., see Figure 2) thereby resulting in different block orders than that in the Morton size ordering. For example, we could try to use the Peano-Hilbert order given in Figure 2d to obtain a Peano-Hilbert size ordering. However, in this case, determining the value associated with a particular block or cell may be rather complex (see Figure 60).
It is interesting to observe that an analogous representation to that of the Morton size ordering can be obtained by not completely ignoring the tree-like nature of the block decomposition process which is a tree having the property that each node has either four sons or is one of the blocks. The ordering is obtained by traversing the resulting tree in the order NW, NE, SW, SE and forming a list of the values of the nodes or blocks encountered. Nonleaf nodes have a value that is different from those associated with the blocks (say 'I' denoting a nonleaf node). In this case, there is no longer a need to record the size of the individual blocks as in the Morton size ordering since we can determine it by an accumulation process that keeps track of how many nonleaf nodes have been encountered in the traversal that starts at the root of the tree. This representation is also known as a DF-expression [75]. For example, the DF-expression of the region quadtree in Figure 60b is given by IWIWBBIWIVBBBWBIBBIBBWW.

The DF-expression stores the structure of the tree along with the values of the leaf nodes. Thus, in the case of a binary image, it may be a bit wasteful in its storage requirements as it requires two bits to encode each of the possible values while there are really just three different values. An alternative is to separate the structure of the tree from that of the values of the leaf nodes by using two separate bit strings [130] termed a linear tree and a color table. The resulting structure is known as an S-tree [72]. The linear tree corresponds to the structure of the tree and uses 0 and 1 to represent nonleaf and leaf nodes, respectively. In the case of a binary image, the color table makes use of 0 and 1 to represent the values of the leaf nodes, while more than one bit is needed for each value when the image has a greater range of possible values. In order to simplify our examples, in the rest of this discussion, we assume, without loss of generality, that the image is binary. Both the linear trees and the color tables are formed by a preorder traversal of the underlying quadtree. For example, the S-tree corresponding to the region quadtree in Figure 60b has 010111010111111011111110011111110011111101111 as its linear tree, and 00011001110111111011 as its color table.

The space saving provided by the S-tree should be quite clear. For example, in the case of a binary image, the S-tree contains one bit (in the linear tree) for each nonleaf node in the original tree, while containing two bits (one in the linear tree and one in the color table) for each leaf node in the original tree. The tradeoff is in the amount of time necessary to determine the value associated with a particular cell in the image as now the process takes twice as much time as in the DF-expression representation since two lists must be traversed in tandem (i.e., the linear tree and the color table). Again, we note that the S-tree could also be used for nonbinary images with even greater space saving relative to the DF-expression.

The principal drawback of the S-tree, as well as the DF-expression, is the fact that determining the value associated with a particular cell may require that the entire representation be examined. This is quite time-consuming especially for large images whose S-tree or DF-expression representation may span many pages. In this case, the S-tree is decomposed into several string segments that are stored on the different pages (e.g., we can distribute the bits of the string corresponding to the linear tree so that each group of consecutive \( m \) bits is on a separate page). This drawback can be overcome by identifying each string segment by the node \( N \) (leaf or nonleaf) in the original tree corresponding to the first element in the segment and then imposing an access structure on these node identifiers. This is the basis of the \( S^+ \)-tree [72] which uses a \( B^+ \)-tree as the access structure with the values of these node identifiers serving as the keys.\(^{16}\)

The simplest way of representing the first node \( N \) in each segment is to use the sequence of directional codes in the original tree corresponding to the path from the root of the tree to \( N \). In this case, we can use the codes 0, 1, 2, and 3 corresponding to NW, NE, SW, and SE, respectively. Transitions are made in the \( B^+ \)-tree on the basis of comparisons from left to right using these key values where for two keys \( i \) and \( j \), \( j > i \) whenever \( j \) is greater in value than \( i \) once the two strings have been made the same length by padding the shorter one on the right with digits having the value 0 such that the longer string is equal. Note that if \( i \) is a substring of \( j \), then \( j > i \). Also, observe that the identifier consisting of the empty string corresponds to the root of the tree which is not necessarily empty.

For example, the linear tree of the S-tree corresponding to the region quadtree in Figure 60b could be

\[^{16}\text{The original formulation of the S-tree and S}^+\text{-tree makes use of a bintree [78, 113, 129] (described in Section 1.2) but this makes no difference here.}\]
decomposed into three segments (of nearly equal size) on three consecutive pages as follows:

1. 010111101 with color table 000110 and no identifier (i.e., the root is the first node on the page).
2. 0111111 with color table 011101 and identifier 21. This is the portion of the tree starting with the NE son of the SW son of the root.
3. 011011111 with color table 111100 and identifier 3. This is the portion of the tree starting with the SE son of the root.

The resulting $S^+$-tree is given in Figure 61. The identifier values are used as the index values in the nonleaf nodes of the $B^+$-tree. Note that the actual keys (i.e., the values of the identifiers) are also stored in the leaf nodes of the $S^+$-tree, which are linked, in order to facilitate operations on the actual data stored in the nodes such as traversing the image in its entirety. In addition, observe that we have varied slightly the definition of the $B^+$-tree given in Section ?? of Chapter ?? in that in our implementation if a key value $k$ is stored in a nonleaf node between pointers to subtrees $p$ and $q$, then all key values in $p$ are less than $k$, and all key values in $q$ are greater than or equal to $k$.

![Figure 61: An $S^+$-tree corresponding to the region quadtree in Figure 60b. 'id', 'lt', and 'ct' denote the identifier, linear tree, and color table, respectively, for each leaf node in the tree. Note the presence of horizontal links in the leaf nodes of the tree in order to facilitate sequential access.](image)

The Morton size ordering did not record the locations of the blocks. Instead, it just recorded their values and sizes in a particular order. As we saw, this information was sufficient to determine the value associated with any block or cell in the image given its location. We can achieve a similar effect by just recording the locations of the blocks instead of their sizes, while still recording their values. In this case, we record the location of the block’s upper-left corner and use the same ordering as in the Morton size ordering. In other words, its ordering is obtained in the same manner as the block decomposition — that is, recursively decompose the embedding space into four congruent blocks (labeled NW, NE, SW, SE in this order) until each block is either completely occupied by an object or is empty at which time the location of the upper-left corner of the block and the block’s value are placed in the list. We term the resulting representation a Morton location ordering. The presence of the ordering enables us to determine the size of the block.

For example, assuming a list of pairs $(a, b)$ where $a$ and $b$ correspond to the value and location, respectively, of each block, the Morton location ordering representation of the region quadtree in Figure 60b corresponding to the object in Figure 60a is $(W, (0, 0)), (W, (4, 0)), (W, (6, 0)), (B, (4, 2)), (B, (6, 2)), (W, (0, 4)), (W, (2, 4)), (B, (3, 4)), (B, (2, 5)), (B, (3, 5)), (W, (0, 6)), (B, (2, 6)), (B, (4, 4)), (B, (6, 4)), (B, (4, 6)), (B, (5, 7)), (W, (6, 6))$.

The size of each block in the Morton location ordering is determined by making use of its Morton number and that of the block that immediately follows it in the ordering. In particular, given the location of the upper-left corner of any block $b$ in the set, and the size $M$ (must be a power of two) of the space $S$ in which the objects are embedded, the size of $b$ is determined by taking the logarithm to the base 4 of the difference between the Morton numbers of the immediately following block in the ordering and $b$. The Morton number of the block following the last block in the ordering is $M^2$.

[^17]: This is in contrast to the other definition which stipulates that if a key value $k$ is stored in a nonleaf node between pointers to subtrees $p$ and $q$, then all key values in $p$ are less than or equal to $k$, and all key values in $q$ are greater than $k$.
The key to understanding why the Morton location ordering works is to observe that the Morton ordering is a space-filling curve that accounts for all of the pixels in each block before moving to the next block in the block sequence. Thus the Morton numbers of two successive blocks \( A \) and \( B \) in the sequence (where \( B \) follows \( A \)) differ by the size of \( A \) since the Morton number of a block corresponds to the first pixel in the block (i.e., the pixel in the upper-left corner of the block). For example, rewriting the Morton location ordering representation given above of the region quadtree in Figure 60b corresponding to the object in Figure 60a yields (\( \mathcal{W}, 0 \)), (\( \mathcal{W}, 16 \)), (\( \mathcal{W}, 20 \)), (\( \mathcal{B}, 24 \)), (\( \mathcal{B}, 28 \)), (\( \mathcal{W}, 32 \)), (\( \mathcal{B}, 36 \)), (\( \mathcal{B}, 37 \)), (\( \mathcal{B}, 38 \)), (\( \mathcal{W}, 40 \)), (\( \mathcal{B}, 44 \)), (\( \mathcal{B}, 48 \)), (\( \mathcal{B}, 52 \)), (\( \mathcal{B}, 56 \)), (\( \mathcal{B}, 57 \)), (\( \mathcal{B}, 58 \)), (\( \mathcal{W}, 59 \)), (\( \mathcal{W}, 60 \)).

Given the location of the upper-left corner of any cell \( c \), determining the value associated with \( c \) using a Morton location ordering is very easy. We simply construct the Morton number \( m \) corresponding to \( c \) and find the block \( B \) in the ordering with the largest Morton number that is less than or equal to \( m \). This query can be executed even more efficiently if we impose an access structure on the Morton numbers such as a \( \mathbb{B}^+ \)-tree as now our search for the Morton number takes logarithmic time instead of being a sequential search. The ability to impose the access structure on the location field is an advantage of the Morton location ordering over the Morton size ordering, although, of course, the Morton location ordering does require more space than the Morton size ordering.

Both the Morton size and location orderings record the values and sizes or locations of each block in the image. This may be wasteful if the same value is associated with consecutive blocks in the image. One way to reduce the storage requirements is to just keep track of the blocks where the values change. This is not possible in the case of the Morton size ordering since only the sizes of the blocks are recorded and thus there is no easy way to identify the locations of the blocks where the values change. However, in the case of the Morton location ordering, such an approach does work since we can determine the size of the individual blocks in the ordering by making use of their Morton numbers and their relative position in the ordering. Thus all we need to do is to keep track of the locations where the values change (i.e., the location of the first block in the ordering associated with each value) as well as the values. Such an adaptation of runlength encoding is termed two-dimensional runlength encoding [81]. As in the case of the Morton location ordering, a sufficient location is the address of the upper-left corner of the block.

For example, assuming an origin at the upper-left corner of the space in which the objects are embedded, the two-dimensional runlength encoding of the region quadtree in Figure 60b is given by (\( \mathcal{W}, (0, 0) \)), (\( \mathcal{B}, (4, 2) \)), (\( \mathcal{W}, (0, 4) \)), (\( \mathcal{B}, (3, 4) \)), (\( \mathcal{W}, (0, 6) \)), (\( \mathcal{B}, (2, 6) \)), (\( \mathcal{W}, (5, 7) \)). Notice the similarity of this approach to the polygon representation of the boundary of an object which just records the locations of the vertices (i.e., the starting point of each vector).

Given the location of the upper-left corner of any cell \( c \) in the image, determining the value associated with \( c \) using a two-dimensional runlength encoding is done in the same manner as with the Morton location ordering. Once again, we simply construct the Morton number \( m \) corresponding to \( c \) and find the block \( B \) in the ordering with the largest Morton number that is less than or equal to \( m \). As in the case of the Morton location ordering, this query can be executed even more efficiently if we impose an access structure on the Morton numbers of the blocks such as a \( \mathbb{B}^+ \)-tree as now our search for the Morton number takes logarithmic time instead of being a sequential search. We can also determine the locations and sizes of all of the blocks in the image using techniques similar to those used to compute the size of the blocks in the Morton location ordering (see Exercise 16).

As in the array representation, two-dimensional runlength encoding can be made more compact by just recording the change in the value associated with the block rather than the value itself. In fact, if the image is binary, then we only need to record the value of the first block in the ordering as all the rest of the blocks in the ordering will alternate in value (although we did not do so in our example). Another alternative to two-dimensional runlength encoding when the image is binary is to just record the locations and sizes of all blocks of a given color. The locations and sizes of the remaining blocks can be determined by a process similar to that used in determining the locations and sizes of the blocks of the same color in the two-dimensional runlength encoding (see Exercise 23).

The same principles used in adapting runlength encoding to interior-based representations can also be used
to a limited extent with some boundary-based representations. For example, in two dimensions, a common boundary-based representation is the chain code [51] which indicates the relative locations of the cells that comprise the boundary of object \( o \). In particular, it specifies the direction of each boundary element and the number of adjacent unit-sized cells that make up the boundary element. Given the location of the first cell in the sequence corresponding to the boundary of \( o \), we can determine the boundary of \( o \) by just following the direction of the associated boundary element. Thus there is no need to record the actual cells or their locations, as now \( o \) is uniquely identified. Therefore, in the chain code, the locations of the boundary elements (i.e., the cells that are adjacent to them) are given implicitly while the nature of the boundary elements is given explicitly (i.e., the direction of the boundary element). Each object has a separate chain code. Of course, an object with holes (e.g., a doughnut) has several chain codes (i.e., 2 for a doughnut). In particular, it has one more chain code than holes. In the rest of this discussion, we assume, without loss of generality, that the objects do not have holes.

In order to see how the chain code is constructed, consider the object in Figure 60a which, as we recall, is embedded in an \( 8 \times 8 \) image array. Assume boundary elements in the form of unit vectors in the four principal directions (corresponding to the boundaries of the cells) so that the interior of the object is always to the right of the unit vectors. Without loss of generality, represent the directions by integer numbers \( i \) ranging from 0 to 3 corresponding to a unit vector having direction \( 90\cdot i \) degrees. In this case, starting from the lower-left corner of the object in Figure 60a and moving clockwise so that the interior of the object will be to the right of the unit vectors, the chain code for the boundary of the object is \( 1^30^3^0^1^0^1^4^0^4^1^2^3^1^2^3^1^2 \) where the exponent is used as a shorthand notation to indicate the length (i.e., multiplicity) of the individual boundary elements in terms of the number of unit vectors comprising them. Notice the similarity of this approach to that used in obtaining the change-based runlength encoding from the value-based runlength encoding discussed earlier for the interior-based representations.

The chain code as described by us does not just record all of the possible changes that could be taking place in that it also records absolute values. In particular, only the relative change of the locations of the cells adjacent to the boundary elements was recorded, while the absolute values of the directions of the boundary elements were recorded. Of course, we could also record the relative change in the directions of the boundary elements. We have just two possible changes in direction corresponding to a transition to the right (denoted as positive) or to the left (denoted as negative). A change in direction of 180 degrees is meaningless as it would mean that the adjacent cell had zero area. Thus we have achieved compression as now our directional component only requires one bit (corresponding to two possible choices) instead of two (corresponding to four possible choices) as is the case for the conventional chain code. Using such an implementation, and assuming that the first element of the chain code must contain the direction, the chain code for the object in Figure 60a becomes \( 1^3^1^4^4^4^1^2^3^1^2^3^1^2 \). Once again, the exponent is used as a shorthand notation to indicate the length (i.e., multiplicity) of the individual boundary elements in terms of the number of unit vectors comprising them.

Using any of the variants of the chain code that we described, determining the value associated with a particular cell \( c \) is not easy. The problem is that the chain code does not provide a mechanism to find the closest part of the boundary to the location of \( c \) without reconstructing the boundary. In other words, there is no correlation between the boundary segments and their locations. We only know their absolute directions or directions relative to the immediately preceding boundary element. We do not know their starting and ending positions without reconstructing the entire boundary, which is possible as we do know the starting position of the boundary. This process is analogous in spirit to the accumulation process used to determine the value associated with a cell in the case of the runlength encoding with the difference is that now we are accumulating lengths of the boundary with the same direction rather than lengths of sequences of consecutive pixels with the same value. Once the boundary has been reconstructed, we can find the value by use of a point-in-polygon test (e.g., [46]), although it is a rather tedious task without the aid of an index (recall Section 2).

Unfortunately, as we mentioned earlier, the chain code cannot be used for data in higher dimensions than two for the same reasons that the polygon representation could not be used. The problem is that regardless of the dimension of the underlying space, the boundary elements lie on hyperplanes whose dimension is one less than the space in which they are embedded, and in these cases no obvious natural order for traversing them.
Nevertheless, even though boundary elements cannot be easily ordered on the basis of connectivity, the fact that vertices serve as locations where change takes place can be used as the basis of a data structure, known as the vertex representation [121], for the interiors of regions that is similar in spirit to runlength encoding. The regions are not restricted to polygons. In particular, they can also be collections of rasters.

The vertex representation was originally developed for storing and processing VLSI masks [121]. It was recently extended to handle orthogonal objects of arbitrary dimension [37, 38]. It applies techniques similar to that of runlength encoding across several dimensions to orthogonal objects that are represented using blocks of arbitrary size located at arbitrary positions. Its use results in decomposing space into multidimensional blocks (not necessarily disjoint) which are aggregated using techniques such as CSG (see Section ?? of Chapter ??) by attaching weights to the blocks. These weights enable the blocks to be manipulated in a manner analogous to that which is achieved through the use of regularized set-union and set-difference operations in CSG. However, instead of using combining elements that are blocks of finite area or halfspaces as in CSG, the vertex representation makes use of vertices which serve as tips of infinite cones.

Each cone is equivalent to an unbounded object formed by the intersection of \( d \) orthogonal halfspaces that are parallel to the \( d \) coordinate axes passing through a particular point, termed the vertex, which is the tip of the cone. As stated above, in some sense, the vertex representation resembles CSG where the primitive elements are halfspaces. In particular, the vertex plays the role of a CSG primitive and the region that it represents is equivalent to the intersection of the halfspaces that pass through it. However, unlike CSG, the vertex representation of an object is unique (but see Exercise 29). The vertices have signs (termed weights and defined below) associated with them indicating whether the space spanned by their cones is included in, or excluded from, the object being modeled. The space spanned by the union of the cones defines the object being modeled. It is interesting to observe that the cones corresponding to the vertices are like infinite blocks thereby slightly resembling a region octree in the sense that the object being modeled is also defined by the union of the blocks. However, unlike the region octree, since the blocks need not be disjoint, the vertex representation of a particular object (i.e., the constituent blocks) is invariant under translation.

Ignoring for the moment the way in which the weights and vertices are determined (but see below), Figure 62 is an example of a vertex representation for an orthogonal polygon. Notice that the vertices labeled with a weight of +1 correspond to infinite cones that are “included” while those labeled with a weight of -1 correspond to infinite cones that are “excluded”.

Figure 62: Example of a vertex representation for an orthogonal polygon

Now, let us turn to a more formal definition of the vertex representation. The vertex representation is primarily designed to represent orthogonal scalar fields (i.e., objects whose boundaries are parallel to the coordinate axes in our application). A scalar field is simply a function that maps points of the domain which is a \( d \)-dimensional space to scalar values. An orthogonal scalar field is a scalar field where regions of the domain space which are mapped to the same value are delimited by faces orthogonal to the coordinate axes. Thus each cone is a scalar field. The overall idea behind the use of the vertex representation for modeling orthogonal polygons is to view the polygons as scalar fields where points inside the polygons are mapped to 1, and
points outside the polygons are mapped to 0.

A vertex\(^{18}\) \(v\) is defined as a pair \((p, \alpha)\) where \(p\) is the location of \(v\) (i.e., it is a point) and \(\alpha\) is a nonzero integer value known as the weight of \(v\). The weight \(\alpha\) is used to determine the value of the scalar field anchored at location \(p\). In particular, a vertex at a location \(p = (p_1, p_2, \ldots, p_d)\) with weight \(\alpha\) has the effect of adding \(\alpha\) to the mapping of all points \(q = (q_1, q_2, \ldots, q_d)\) such that \(q_i \geq p_i\) for all \(i = 1 \ldots d\) (see Figure 63aa). As a consequence, we can tell if a point \(q\) is inside or outside the polygon by adding the weights of all vertices that contribute to its mapping — that is, a vertex at point \(p\) contributes to the mapping of \(q\) if and only if \(p_i \leq q_i\) for all \(i = 1 \ldots d\) (see Figure 63ab). Therefore, given a particular cell \(c\) having a lower-left corner at location \(l\), determining the value associated with \(c\) is a simple process consisting of finding all vertices whose corresponding cones contain \(l\) and adding up their weights. An alternative way of characterizing this process is to say that we add the weights of all vertices whose location is dominated (e.g., [97]) by \(l\). The process of determining the relevant cones could be made efficient by making use of an index on the vertices such as a variant of a quadtree, k-d tree, etc. (see Chapter ??).

![Figure 63: (a) A vertex (black dot) and some of the points influenced by it (black squares). (b) The point (black square) is only influenced by the vertices shown as black dots.](image)

The weights enable us to represent orthogonal polygons as a collection of vertices by using negative weight values at some of the vertices. In particular, a set of vertices \(V = \{v_1, v_2, \ldots, v_n\}\) induces a scalar field \(Q_V = \sum_{i=1}^{n} Q_{v_i}\). In other words, the scalar field induced by a set of vertices is the sum of the scalar field induced by each vertex separately. This means that the operation of addition is well-defined for vertices. For example, recall Figure 62 which is the vertex representation for a simple polygon.

At this point, it is important to ask how we determine the locations that comprise the vertices and their weights. Intuitively, the vertices are the locations where different boundary elements meet and hence the boundary changes (i.e., it bends by 90 degrees). There are three methods of identifying the vertices. The first method is visual and is the natural and intuitive way. Once the location \(l\) of a vertex \(v\) has been identified, its weight may be determined by analyzing how the scalar field changes in the immediate neighborhood of \(l\). For instance, in two dimensions the scalar field must be evaluated immediately to the northeast, northwest, southeast and southwest of \(l\). Once this is done, it is possible to determine the weight of \(v\) by inspection (we discuss how to perform the actual computation later on). Of course, more neighbors must be examined in three dimensions and higher. As we can see, this method is simple in two dimensions. However, it becomes increasingly more complex in three dimensions and higher as it requires us to be able to visualize the object which is not an easy task, especially in four dimensions and higher.

The second method is very similar to the first method except that it is more formal. In particular, it consists of interpreting the vertex representation as a differential code for rasters and just retaining the locations with a nonzero weight. To see why this works, we make two important observations. These observations are in the context of two-dimensional data but the extensions to higher dimensional data are not difficult. Our first observation is that if a vertex \(p\) is placed at a certain point of the domain space, then its weight represents the difference between the sum of the fields induced by all other vertices that have \(p\) inside their cones and the

\(^{18}\)Note that we use the term “vertex” to refer to a mathematical entity that does not necessarily correspond to the common definition of “vertex” in Geometry.
actual value of the field inside the cone having its tip at \( p \). For example, in Figure 64, \( p \) is inside the cones of vertices \( a, b, c, \) and \( d \). Therefore, if we want to map the shaded region to the value \( S \), then we must add to the representation a vertex placed at \( p \) with weight \( \frac{S}{wgt(a) - wgt(b) - wgt(c) - wgt(d)} \), where \( wgt(v) \) refers to the weight of vertex \( v \).

![Figure 64: Vertices that influence the scalar field at a point \( p \)](image)

Our second observation is based on a definition of the weight of a cell \( c \) corresponding to a pixel as being the weight of a vertex placed at the lower-left corner of \( c \). Using this definition, we observe that the weight of a vertex that lies in the center of a \( 2 \times 2 \) block of pixels is only dependent on the values of the pixels of the block. This observation is aided by considering the four adjacent cells (i.e., pixels) with values \( a, b, c, \) and \( d \) of a two-dimensional raster as depicted in Figure 65. The cell with value \( a \) is influenced by vertices lying in regions \( R_1 \) and \( R_2 \), the cell with value \( b \) by vertices lying in region \( R_2 \), and the cell with value \( c \) by vertices lying in regions \( R_2 \) and \( R_3 \). Thus, if we want the upper-right cell to have a value of \( D \), then a vertex should be placed at point \( p \) with weight equal to the difference between \( D \) and the weights of all vertices in regions \( R_1, R_2, \) and \( R_3 \). These weight sums can be rewritten in terms of \( a, b, c, \) and \( d \) by observing that:

- \( A = \text{sum}(R_1) + \text{sum}(R_2) \),
- \( B = \text{sum}(R_2) \),
- \( C = \text{sum}(R_2) + \text{sum}(R_3) \).

From this we have that:

- \( \text{sum}(R_1) = A - B \)
- \( \text{sum}(R_2) = B \)
- \( \text{sum}(R_3) = C - B \).

Subtracting these values from that of \( D \) leads to the weight for the vertex at \( p \) being given by:

\[
\text{wgt}(D) = D - (A - B) - B - (C - B) = D + B - A - C
\]

The importance of this second observation lies in the fact that it permits us to encode the images by computing the weight at a particular pixel (e.g., the one at the upper-right cell in a \( 2 \times 2 \) block) without having to keep track of the weight sums over all possible regions. In particular, the weight at the upper-right corner only depends on the values of the pixels at the lower-left, lower-right, upper-right, and upper-left corners. Moreover, using this result, we see that when we apply the differential coding scheme to a general raster, vertices (i.e., points with nonzero weights) will be generated only at certain positions. In order to see this, consider a two-dimensional raster representing a monochromatic image, where black pixels are mapped to 1 and white pixels to 0. We examine the 16 possible configurations of \( 2 \times 2 \) pixels focussing on a pixel at position \( D \) using
the labeling given in Figure 65. In particular, vertices will be generated in just 10 out of the 16 configurations as depicted in Figure 66. By inspection, we notice that those regions which are uniformly black or white, as well as regions containing strictly horizontal or vertical boundaries do not generate vertices. In summary, not surprisingly, we find that only groups of pixels near corners of features generate nonzero codes (i.e., vertices).

![Figure 65: Example of how the interpretation of the vertex representation as a differential code for rasters is used to determine the weight of the vertex at p to be $D + B - A - C$.](image)

The third method is procedural. It is best understood by examining the construction of the vertex representation of hyperrectangles in $d$ dimensions. The idea is to sweep a hyperrectangle in $d - 1$ dimensions along the $d^{th}$ axis. The process starts out with the equivalent of a hyperrectangle in one dimension ($x$) which is an interval. In particular, it is the scalar field that maps the interval $[a_1, b_1]$ to 1 and all other points to 0. This field is represented with two vertices: one at $x = a_1$ with weight +1 and another at $x = b_1$ with weight -1.

Next, we transform this one-dimensional interval into a two-dimensional interval by adding to each vertex a second coordinate ($y$) equal to $a_2$. This results in a two-dimensional scalar field where the region of space being mapped to 1 is bounded by line $y = a_2$ but extends towards $+\infty$ along the $y$ axis. Thus, we need to add two other vertices with coordinates $(a_1, b_2)$ and $(b_1, b_2)$ and weights $-1$ and $+1$, respectively, which will provide another boundary adjacent to line $y = b_2$.

This idea can be extended easily to higher dimensions. If we start with a set $L$ of vertices representing a hyperrectangle in $(d-1)$-dimensional space, then we may represent a hyperrectangle defined in $d$-dimensional space by a set $L'$ obtained through the following process:

1. Copy $L$ to $L'$ adding a new coordinate to each vertex, say, $a_d$.
2. Append $L$ to $L'$ adding a new coordinate to each vertex, say, $b_d$, where $b_d > a_d$, and with weights changed from +1 to $-1$ and vice-versa.

Figure 67 shows how this process can be repeated in order to build a vertex representation for a hyperrectangle in three dimensions — i.e., a parallelepiped.

This method can also be applied to objects that need not be hyperrectangles. In particular, it can be applied to arbitrary rasters by using the sweep method to construct one-dimensional vertex representations for
successive rows in the image followed by sweeping the rows along the columns and subtracting the weights of vertices in corresponding positions. The same technique is applied to higher dimensional data by applying the sweep-subtraction paradigm to the planes.

This procedure is best understood by examining the result of its application to the two-dimensional raster given in Figure 68a. Figure 68b is the result of the application of the algorithm to the individual rows, while Figure 68c is the final vertex representation. Figure 68a is similar to the result obtained using traditional run-length encoding which only applies compression in one dimension, while the vertex representation applies compression in the remaining dimensions as well as seen in Figure 68c. Since we are constructing a differential code, we must assume a reference value of 0 for the pixels that are outside the region spanned by the two-dimensional raster and on its boundary. Thus, when we process the first pixel of a row (as well as an entire row in the sweep of rows phase), we assume a (nonexistent) previous pixel (or entire row) with a value of 0.

It should be clear that the algorithm that we described for the hyperrectangles is a subset of general algorithm given above in that the starting point of the one-dimensional interval corresponds to the cone that is included in the object while the ending point of the one-dimensional interval corresponds to the cone that is excluded from the object. The general algorithm determines the weights of the locations that comprise the vertices in a systematic manner by traversing the locations in each row, followed by the rows that make up each plane, etc. This order is known as inverse lexicographic order and is illustrated in Figure 69 for the vertices a, b, c, and d which would be visited in the order c d a b.

This traversal process is facilitated by storing the vertices in a list in the order in which they are visited (i.e., inverse lexicographic order). For example, in two dimensions, they are sorted first on the basis of their y coordinate value and then by their x coordinate value. This representation is termed a vertex list and it has been used as a basis of a number of algorithms for operations on the vertex representation including set oper-
The utility of the vertex list representation is that it permits these algorithms to employ the plane-sweep paradigm (or line-sweep in two dimensions) [97]. For example, in two dimensions, this is compatible with the sweeping of a line perpendicular to the $y$ axis in the direction of increasing $y$ coordinate values.

As an example of the utility of implementing the vertex representation using a vertex list, we examine the conversion of rasters to a vertex representation implemented as a vertex list. This is given below by the following recursive algorithm called $\text{raster\rightarrow vlist}$ with parameters $d$ and $R$ corresponding to the $d$-dimensional raster $R$ being processed.

1. If $d = 1$, then for each pair of adjacent pixels $R(i)$, $R(i + 1)$ at locations $i$ and $i + 1$, respectively, along the least significant coordinate value, compute $\text{weight} = R(i + 1) - R(i)$. Output a vertex at location $i$ if $\text{weight}$ is nonzero.

2. Otherwise (i.e., $d > 1$), for each pair of consecutive $(d - 1)$-dimensional rasters $R(i)$ and $R(i + 1)$ at $i$ and $i + 1$, respectively, along the $d^{th}$ coordinate value, do:
   
   (a) Compute vertex lists $L_i$ and $L_{i+1}$ for $(d - 1)$-dimensional rasters $R(i)$ and $R(i + 1)$, respectively, by recursive invocation of procedure $\text{raster\rightarrow vlist}$ (i.e., steps 1 and 2).

   (b) Compute $S = \text{add}(L_{i+1}, L_i, -1)$ which subtracts the two vertex lists.

   (c) Make all vertices of $S$ $d$-dimensional by appending $i$ as the value of the $d^{th}$ coordinate value to their positions and output them.

Given a particular cell $c$ having a lower-left corner at location $l$, determining the value associated with $c$ is achieved by finding all vertices whose corresponding cones contain $l$ and adding up their weights. This process requires us to take into account the fact that we are dealing with a vertex list implementation of the vertex representation which means that we need to process the list in a particular order. A simple approach would partially reconstruct the raster in an analogous manner to that used by chain-based runlength encoding (i.e., a process of accumulating changes in values) by scanning the vertex list element-by-element adding up the weights of vertices that are “dominated” by $l$. A more elegant (and general in terms of arbitrary dimensions) way of describing this process is to use a procedure that works in the inverse manner as the $\text{raster\rightarrow vlist}$ procedure given above (see Exercises 31 and 32).

The vertex list implementation of the vertex representation demonstrates why we say that the vertex representation is a variant of runlength encoding. In particular, observe that the only pixels for which values are recorded in the vertex list are those at which the pixel values change. The facts that the vertex list implementation of the vertex representation orders the pixels in raster-scan order (i.e., the pixel values in the image are ordered by rows) and that only changes in the values are recorded demonstrate why we say that the vertex representation is a variant of runlength encoding. The main difference from the process of runlength encoding for
the multidimensional array described earlier that was only applied to the rows is that in the case of the vertex representation we are not restricted to blocks of a fixed size (e.g., squares or congruent blocks). Of course, this is not quite true as we restrict ourselves to congruent blocks of infinite size (anchored at the vertices). Is essence, the vertex representation is a combination of an implicit interior-based representation and an implicit boundary-based representation in the sense that both interiors and boundaries of regions are represented implicitly through the aid of a single vertex which is the tip of a cone. Moreover, only the changes in the value are recorded. Therefore, we see that using the vertex list implementation of the vertex representation, we have an ordering of the pixels which is analogous to a raster-scan order (i.e., the pixel values in the image are ordered by rows).

The main difference from the process of runlength encoding for the multidimensional array described earlier that was only applied to the rows is that in the case of the vertex representation we are not restricted to blocks of a fixed size (e.g., squares or congruent blocks). Of course, this is not quite true as we restrict ourselves to congruent blocks of infinite size (anchored at the vertices). Is essence, the vertex representation is a combination of an implicit interior-based representation and an implicit boundary-based representation in the sense that both interiors and boundaries of regions are represented implicitly through the aid of a single vertex which is the tip of a cone.

**Exercises**

1. Give an algorithm to construct the value-based runlength encoding for an $M \times M$ image represented by an array.

2. Give an algorithm to determine the value associated with a cell $c$ in an $M \times M$ image represented by a value-based runlength encoding.

3. Give an algorithm to construct the change-based runlength encoding for an $M \times M$ binary image represented by an array.

4. Give an algorithm to determine the value associated with a cell $c$ in an $M \times M$ binary image represented by a change-based runlength encoding.

5. Give the Morton size ordering of the blocks of the region quadtree in Figure 8 in the form of a list of ordered pairs $(a, b)$ where $a$ and $b$ correspond to the value and size (i.e., power of 2) of each block. Assume that the tree is traversed in the order NW, NE, SW, SE.

6. Give an algorithm to determine the value associated with a cell $c$ having an upper-left corner at location $(x, y)$ in an $M \times M$ image ($M$ is a power of 2) represented by a Morton size ordering.

7. Give an algorithm to determine the value associated with a cell $c$ having an upper-left corner at location $(x, y)$ in an $M \times M$ image ($M$ is a power of 2) represented by a Peano-Hilbert size ordering.

8. Give the DF-expression for the region quadtree in Figure 8 assuming a traversal of the tree in the order NW, NE, SW, SE.

9. Give an algorithm to determine the value associated with a cell $c$ having an upper-left corner at location $(x, y)$ in an $M \times M$ image ($M$ is a power of 2) represented by a DF-expression.

10. Give an algorithm to determine the value associated with a cell $c$ having an upper-left corner at location $(x, y)$ in an $M \times M$ image ($M$ is a power of 2) represented by an S-tree.

11. The original formulation of the $S^+$-tree [72] uses a slightly different representation for the identifier corresponding to the first node $N$ in each segment $S$. If the node is the root of the tree (which is not necessarily empty), then the identifier is empty. If the tree consists of just one leaf node and no non-leaf nodes, then the identifier is 1. Otherwise, the identifier consists of a sequence of directional codes corresponding to the path from the root of the tree to $N$ where each code is represented by a variable length sequence of the two binary digits 0 and 1 as follows:
1. NW: 0.
2. NE: 01.
3. SW: 011.
4. SE: 0111.

This string is equivalent to the linear tree encoding of a tree $T$ containing $N$ having the property that all leaf nodes at positions preceding $N$ are of the same color $R$, where $R$ is different from the color of $N$. This encoding of the identifier is termed the linear prefix of $S$. Of course, the first page has no linear prefix. Give an algorithm to build an $S^+$-tree using this representation of the identifier corresponding to the first node $N$ in each segment $S$.

Assume the following implementation for the $S^+$-tree [72]. In order for each segment to be self-contained, corresponding parts of the linear tree and the color table are stored on the same page, as is the linear prefix. Each page is assumed to be of the same size. Since the goal is to utilize as much of the available storage on each page as possible, the linear prefix and the linear tree are represented as one consecutive bit string while the color table is represented as another bit string. These two bit strings are stored in the page as stacks which grow towards each other. Use of the stacks means that two top of stack pointers must be stored with each page to indicate the positions of their last elements as some of the positions in the page may be unoccupied. In order to distinguish between the linear prefix and the linear tree, the length of the linear prefix is also recorded with each page.

12. In the original presentation of the $S^+$-tree [72] each page was required to terminate with a leaf node. Is this absolutely necessary?

13. In the original presentation of the $S^+$-tree, it is described as a variant of a prefix $B^+$-tree [?] (see Section ?? of Chapter ??) whereas, as we have seen, it is really just a $B^+$-tree. Using the definition of an $S^+$-tree in terms of a linear prefix given in Exercise 11, explain how such a characterization could have been made. [Hint: Think about the possible interpretations of the values that serve as the keys in the two data structures.]

14. Why can’t we compute the size of each block in the Morton location ordering directly from the locations of successive blocks in the list instead of having to use the difference between their Morton numbers? For example, why couldn’t we use the maximum or minimum differences between the corresponding $x$ and $y$ coordinate values of their location entries?

15. Suppose that you use a Peano-Hilbert location ordering. How can you determine the size of each block? Assume that for each block $b$ we record the location of the pixel in $b$’s upper-left corner.

16. Give an algorithm to determine the locations and sizes of all blocks of the same value in a two-dimensional runlength encoding given the locations of the upper-left corners of two successive blocks $b_1$ and $b_2$ with Morton numbers $c_1$ and $c_2$, respectively. The result includes all blocks starting with $b_1$ up to but not including $b_2$.

17. Suppose that in the Morton location ordering instead of recording the location of each block’s upper-left corner we record the location of its lower-right corner (i.e., the location of the last pixel in the block). How would you compute the size of each block?

18. Suppose that the two-dimensional run encoding sequence were defined to consist of the last element of each subsequence of blocks of the same value rather than the first element. Give the resulting encoding of the region quadtree in Figure 60b. Assume that the tree is traversed in the order NW, NE, SW, SE.

19. Suppose that the two-dimensional run encoding sequence were defined to consist of the last element of each subsequence of blocks of the same value rather than the first element. Give a procedure similar to that given in Exercise 16 to reconstruct the Morton numbers of the blocks of the quadtree given the Morton numbers of two successive blocks. If this is not possible, then how can you redefine the Morton number associated with each block so that it will work? Again, the inputs are the locations of
the upper-left corners of two successive blocks $b_1$ and $b_2$ with Morton numbers $c_1$ and $c_2$, respectively.

20. Give the two-dimensional runlength encoding for the region quadtree in Figure 8. Assume that the tree is traversed in the order NW, NE, SW, SE.

21. Give an algorithm to determine the value associated with a cell $c$ having a upper-left corner at location $(x, y)$ in an $M \times M$ image ($M$ is a power of 2) represented by a two-dimensional runlength encoding.

22. Suppose that you base the two-dimensional runlength encoding on the Peano-Hilbert location ordering instead of the Morton location ordering. Give an algorithm to determine the value associated with a block $b$ having an upper-left corner at location $(x, y)$ in an $M \times M$ image ($M$ is a power of 2) represented by this variant of the two-dimensional runlength encoding.

23. Consider a binary image for which only the locations and sizes of the black blocks are recorded. Give an algorithm to determine the locations and sizes of the white blocks.

24. Give an algorithm to determine the value associated with a cell $c$ having an upper-left corner at location $(x, y)$ in an $M \times M$ image ($M$ is a power of 2) for which only the locations and sizes of the black blocks are recorded.

25. Suppose that you use a chain code that records the relative change in the directions of the boundary elements. Prove that the number of positive transitions is always three more than the number of negative transitions. Assume that the object has no holes.

26. Suppose that you use a chain code that makes use of unit vectors in the four principal directions so that the interior of the object is always to the right of the unit vectors. Assume that the directions are represented by integer numbers $i$ ranging from 0 to 3 corresponding to a unit vector having direction $90 \cdot i$ degrees. Prove that the sum of the lengths of the boundary elements encoded by 0 is equal to those encoded by 2. Similarly, prove that the sum of the lengths of the boundary elements encoded by 1 is equal to those encoded by 3.

27. Assume that orthogonal polygons are scalar fields where points inside the polygons are mapped to 1, and points outside the polygons are mapped to 0. Given a vertex representation of a two-dimensional orthogonal polygon, prove that any vertex $p$ at a position given by $(x_0, y_0)$ such that no other point $(x, y)$ on the boundary of the polygon satisfies $(x < x_0) \land (y < y_0)$ will have weight equal to +1.

28. How does the vertex representation deal with polygons containing holes?

29. In the text we stated that the vertex representation of an orthogonal polygon is unique. How do you reconcile this statement with the fact that the following object can have two different vertex representations depending on how its boundary is interpreted? In particular, interpreting the boundary as in (a) yields

```
  x  y
  0  1
  1  1
  1  1
```

the vertex representation in (b), whereas interpreting it as in (c) results in the vertex representation given in (d).
30. It has been observed [37] that, in two dimensions, for a given simple orthogonal polygon (i.e., non-self intersecting), the weights of adjacent vertices along the border of the polygon always alternate between +1 and −1 as they are visited in clockwise or counter-clockwise order. Prove that this is true.

31. Given a two-dimensional raster image represented by a vertex list, give an algorithm to determine the value associated with a location \( c \). The algorithm should be an inverse of the procedure \( \text{raster}_\text{to}_\text{vlist} \) used to convert a raster to a vertex representation implemented as a vertex list.

32. Given an arbitrary \( d \)-dimensional raster image represented by a vertex list, give an algorithm to determine the value associated with a location \( c \). The algorithm should be an inverse of the procedure \( \text{raster}_\text{to}_\text{vlist} \) used to convert a raster to a vertex representation implemented as a vertex list.

33. Assuming a vertex representation implemented as a vertex list, give an algorithm to perform set-theoretic operations such as union and intersection.

34. The vertex list implementation of the vertex representation orders the vertices in inverse lexicographic order. This is a variant of row order (recall the discussion in Section 1.1 and Figure 2a). Suppose that we order the vertices using another ordering such as the Morton or the Peano–Hilbert orders. How would this affect the implementation of some of the operations on vertex lists. For example, discuss its effect on the algorithm for set-theoretic operations in Exercise 33.

35. How would you extend the vertex list representation to deal with polygons whose faces are not orthogonal?
References

Each reference is followed by a key word(s). It is also followed by a list of the sections in which it is referenced. The format is D or A followed by the section number. D corresponds to this book while A correspond to [107]. D.P and A.P denote the appropriate preface and L denotes the appendix describing the pseudo-code language. All references that are cited in the solutions to the exercises are associated with the section in which the exercise is found.


[34] H. Edelsbrunner. Dynamic rectangle intersection searching. Institute for Information Processing 47, Technical University of Graz, Graz, Austria, February 1980. [rectangles] D.1.4


[71] H. V. Jagadish. Spatial search with polyhedra. In Proceedings of the Sixth International Conference on Data Engineering, pages 311–319, Los Angeles, CA, February 1990. [rectangles; range searches; polyhedral search regions; polyhedral bounding rectangles; P-trees] D.1.1


