Applications of Spatial Data Structures

Computer Graphics, Image Processing, and GIS

Hanan Samet
UNIVERSITY OF MARYLAND

Applications of Spatial Data Structures

Computer Graphics, Image Processing, and GIS

Hanan Samet
UNIVERSITY OF MARYLAND

ADDISON - WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam
Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

This book is in the Addison-Wesley Series in Computer Science
Michael A. Harrison: Consulting Editor

Library of Congress Cataloging-in-Publication Data

Samet, Hanan.
Applications of spatial data structures: computer graphics, image processing, and GIS / by Hanan Samet.
p. cm.
Bibliography: p.
Includes index.
ISBN 0-201-50300-X
1. Data structures (Computer science) 2. Computer graphics.
I. Title.
QA76.9.D35S25 1989
006—dc19

89-30365
CIP

ABCDEFGHIJ-MA-89

Credits:
Thor Bestul created the cover art.

Figures 1.1, 2.3, 3.1, 4.1, 4.3, and 5.16 are from H. Samet and R. E. Webber, On encoding boundaries with quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 6*, 3(May 1984), 365–369. © 1984 IEEE. Reprinted by permission of IEEE.

Figures 1.2, 1.3, 1.17, 6.2, 6.4, 6.5, and 6.12 are from H. Samet and R. E. Webber, Hierarchical data structures and algorithms for computer graphics, Part I. Fundamentals, *IEEE Computer Graphics and Applications 8*, 3(May 1988), 48–68. © 1988 IEEE. Reprinted by permission of IEEE.

Figures 1.4 through 1.6, 1.9 through 1.14, 1.18, 3.3, 4.16, 4.28, 5.2, 5.17, 5.18, 6.1, 6.3, and 6.10 are from H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984), 187–260. Reprinted by permission of ACM.

Figures 1.15 and 4.24 are from H. Samet and R. E. Webber, Storing a collection of polygons using quadtrees, *ACM Transactions on Graphics 4*, 3(July 1985), 182–222. Reprinted by permission of ACM.

Figures 2.1, 4.22, 4.23, and 8.1 are from C. R. Dyer, A. Rosenfeld, and H. Samet, Region representation: boundary codes from quadtrees, *Communications of the ACM 23*, 3(March 1980), 171–179. Reprinted by permission of ACM.

Figures 2.2 and 8.2 through 8.19 are from H. Samet, Data structures for quadtree approximation and compression, *Communications of the ACM 28*, 9(September 1985), 973–993. Reprinted by permission of ACM.

Figures 2.4 and 4.13 are from C. A. Shaffer and H. Samet, Optimal quadtree construction algorithms, *Computer Vision, Graphics, and Image Processing 37*, 3(March 1987), 402–419. Reprinted by permission of Academic Press.
Continued on p. 507

PREFACE

The quadtree and octree are hierarchical data structures used to represent spatial data. They are based on the principle of recursive decomposition (similar to *divide and conquer* methods [Aho74]). This book focuses on the use of quadtree and octree representations of region data (in two and three dimensions, respectively) in applications in computer graphics, image processing, and geographic information systems (GIS), as well as computer vision, robotics, pattern recognition, solid modeling, and other areas. For a comprehensive treatment of related hierarchical representations of spatial data including points, lines, rectangles, regions, and volumes, see [Same90a].

To many people, the terms *quadtree* and *octree* have taken on a generic meaning synonymous with the term *hierarchical data structure*. Hierarchical data structures are useful because of their ability to focus on the interesting subsets of the data. This focusing results in an efficient representation and in improved execution times. Thus they are particularly convenient for performing set operations. Many of the operations described can often be performed as efficiently, or more so, with other data structures. Nevertheless, hierarchical data structures are attractive because of their conceptual clarity and ease of implementation. In addition, the use of some of them provides a spatial index. This is very useful in applications involving spatial databases.

This book is organized as follows. Chapter 1 introduces hierarchical data structures such as the quadtree and octree. It reviews their key properties, traces their history, and gives an overview of their use in representing point and line data, as well as three-dimensional regions. All of these topics are covered in much greater detail in [Same90a].

Most hierarchical data structures are trees. As such they are most often implemented using pointers. Chapter 2 discusses alternative implementations that do not make use of pointers and compares their storage requirements with an implementation that does use pointers. Chapter 3 contains a detailed presentation of how to perform

neighbor finding. This is an important technique in the efficient implementation of a number of algorithms that use quadtree and octree-like representations. It is used heavily in their construction (Chapter 4), computing geometric properties (Chapter 5), ray tracing (Chapter 7), and skeletons (Chapter 9). Chapters 2 and 3 may be skipped by readers who are interested only in applications.

The remaining chapters discuss the applications in greater detail. Chapter 4 shows the ease with which conversions can be made between the quadtree representation of two-dimensional regions and more conventional representations such as arrays, rasters, and boundary codes. The extension to three-dimensional regions is straightforward and is discussed only in the context of building an octree from a set of views of an object or a scene of three-dimensional objects. For details on converting between other representations of three-dimensional regions and octrees (e.g., boundary model [BRep], constructive solid geometry [CSG]), see Chapter 5 of [Same90a].

Chapter 5 examines the computation of geometric properties such as connected component labeling. Such operations arise in computer graphics (where it is known as polygon coloring or filling) and as a basic step in the processing of an image.

Chapter 6 discusses the implementation of set-theoretic operations, linear transformations, and other algorithms useful in computer graphics and image processing. Chapter 7 contains a detailed presentation of the use of hierarchical data structures in the display of graphical information. In particular, much attention is given to the problem of ray tracing and some to beam tracing. The radiosity method is covered with considerably less detail. Chapters 6 and 7 may be skipped by readers uninterested in computer graphics.

Chapter 8 treats the problem of image approximation and compression by use of hierarchical representations. This discussion is in the context of two-dimensional binary images. Chapter 9 examines the application of quadtree-like decomposition to the computation of skeletons. In particular, the concept of distance is formulated in the context of a quadtree, and its application is explored. Chapters 8 and 9 are somewhat specialized and may be omitted in the interest of time.

This book is designed to be used as a reference as well as the basis of a course on the implementation of a graphics, image processing, or geographic information system (GIS) based on quadtrees and octrees. It can also be used as a supplement in a general course on these topics.

There are a number of topics for which justice requires considerably more detailed treatment. Due to space limitations, I have omitted a detailed discussion of them and instead refer interested readers to the appropriate literature. The notion of a pyramid is presented only at a cursory level in Chapter 1 so that it can be contrasted with the quadtree. In particular, the pyramid is a multiresolution representation, whereas the quadtree is a variable resolution representation. Readers are referred to Tanimoto and Klinger [Tani80] and the collection of papers edited by Rosenfeld [Rose83a] for a more comprehensive exposition on pyramids. The use of quadtrees in finite element analysis is mentioned in Chapter 1; for more details, see Kela, Perucchio, and Voelcker [Kela86] and the references cited there. Similarly I discuss image

compression and coding only in the context of hierarchical data structures. This is done in Chapter 8.

For more details on early results involving these and related topics, consult the surveys by Nagy and Wagle [Nagy79], Peuquet [Peuq84], Requicha [Requ80], Srihari [Srih81], Samet and Rosenfeld [Same80d], Samet [Same84b], and Samet and Webber [Same88c, Same88d]. A number of excellent texts contain material related to the topics that I cover. Rosenfeld and Kak [Rose82a] should be consulted for an encyclopedic treatment of image processing. Mäntylä [Mänt87] has written a comprehensive introduction to solid modeling. Burrough [Burr86] provides a survey of geographic information systems. For a comprehensive view of the literature, see Rosenfeld's annual collection of references in the journal *Computer Vision, Graphics, and Image Processing* (e.g., [Rose88]).

Nevertheless, given the broad and rapidly expanding nature of the field, I am bound to have omitted significant concepts and references. In addition, at times, I devote a disproportionate amount of attention to some concepts at the expense of others. This is principally for expository purposes; I feel that it is better to understand some structures well rather than to give readers a quick runthrough of buzzwords. For these indiscretions, I beg your pardon and hope you nevertheless bear with me.

My approach is an algorithmic one. Whenever possible, I have tried to motivate critical steps in the algorithms by a liberal use of examples. I feel that it is of paramount importance for readers to see the ease with which the representations can be implemented and used. In each chapter, except for the introduction (Chapter 1), I give at least one detailed algorithm using pseudo-code so that readers can see how the ideas can be applied. The pseudo-code is a variant of the ALGOL [Naur60] programming language that has a data structuring facility incorporating pointers and record structures. Recursion is used heavily. This language has similarities to C [Kern78], PASCAL [Jens74], SAIL [Reis76], and ALGOL W [Baue68]. Its basic features are described in the Appendix. However, the actual code is not crucial to understanding the techniques and it may be skipped on a first reading. The index indicates the page numbers where the code for each algorithm is found.

In many cases I also give an analysis of the space and time requirements of different data structures and algorithms. The analysis is usually of an asymptotic nature and is in terms of *big O* and Ω notation [Knut76]. The *big O* notation denotes an upper bound. For example, if an algorithm takes $O(\log_2 N)$ time, then its worst-case behavior is never any worse than $\log_2 N$. The Ω notation denotes a lower bound. As an example of its use, consider the problem of sorting N numbers. When I say that sorting is $\Omega(N \cdot \log_2 N)$, I mean that given any algorithm for sorting, there is some set of N input values for which the algorithm will require at least this much time.

At times I also describe implementations of some of the data structures for the purpose of comparison. In such cases counts such as the number of fields in a record are often given. These numbers are meant only to amplify the discussion. They are not to be taken literally, as improvements are always possible once a specific application is analyzed more carefully.

Each chapter contains a substantial number of exercises. Many of the exercises develop further the material in the text as a means of testing the reader's understanding, as well as suggesting future directions. When the exercise or its solution is not my own, I have preceded it with the name of its originator. The exercises have not been graded by difficulty. They rarely require any mathematical skills beyond the undergraduate level for their solution. However, while some of the exercises are quite straightforward, others require some ingenuity. Solutions, or references to papers that contain the solution, are provided for a substantial number of the exercises that do not require programming. Readers are cautioned to try to solve the exercises before turning to the solutions. It is my belief that much can be learned this way (for the student and, even more so, for the author). The motivation for undertaking this task was my wonderful experience on my first encounter with the rich work on data structures by Knuth [Knut73a, Knut73b].

An extensive bibliography is provided. It contains entries for both this book and the companion text [Same90a]. Not all of the references that appear in the bibliography are cited in the two texts. They are retained for the purpose of giving readers the ability to access the entire body of literature relevant to the topics discussed in them. Each reference is annotated with a key word(s) and a list of the numbers of the sections in which it is cited in either of the texts (including exercises and solutions). In addition, a name and credit index is provided that indicates the page numbers in this book on which each author's work is cited or a credit is made.

ACKNOWLEDGMENTS

Over the years I have received help from many people, and I am extremely grateful to them. In particular, Robert E. Webber, Markku Tamminen, and Michael B. Dillencourt have generously given me much of their time and have gone over critical parts of the book. I have drawn heavily on their knowledge of some of the topics covered here. I have also been extremely fortunate to work with Azriel Rosenfeld over the past ten years. His dedication and scholarship have been a true inspiration to me. I deeply cherish our association.

I was introduced to the field of spatial data structures by Gary D. Knott who asked "how to delete in point quadtrees." Azriel Rosenfeld and Charles R. Dyer provided much interaction in the initial phase of my research. Those discussions led to the discovery of the neighbor-finding principle. It is during that time that many of the basic conversion algorithms between quadtrees and other image representations were developed as well. I learned much about image processing and computer vision from them. Robert E. Webber taught me computer graphics, Markku Tamminen taught me solid modeling and representations for multiattribute data, and Michael B. Dillencourt taught me about computational geometry.

During the time that this book was written, my research was supported, in part, by the National Science Foundation, the Defense Mapping Agency, the Harry Diamond Laboratory, and the Bureau of the Census. In particular, I thank Richard

Antony, Y. T. Chien, Su-shing Chen, Hank Cook, Phil Emmerman, Joe Rastatter, Alan Saalfeld, and Larry Tokarcik. I am appreciative of their support.

Many people helped me in the process of preparing the book for publication. Acknowledgments are due to Rene McDonald for coordinating the day-to-day matters of getting the book out and copyediting; to Scott Carson, Emery Jou, and Jim Purtilo for TROFF assistance beyond the call of duty; to Marisa Antoy and Sergio Antoy for designing and implementing the algorithm formatter used to typeset the algorithms; to Barbara Burnett, Michael B. Dillencourt, and Sandy German for help with the index; to Jay Weber for setting up the TROFF macrofiles so that I can keep track of symbolic names and thus be able to move text around without worrying about the numbering of exercises, sections, and chapters; to Liz Allen for early TROFF help; to Nono Kusuma, Mark Stanley, and Joan Wright Hamilton for drawing the figures; to Richard Muntz and Gerald Estrin for providing temporary office space and computer access at UCLA; to Sandy German, Gwen Nelson, and Janet Salzman for help in initial typing of the manuscript; to S. S. Iyengar, Duane Marble, George Nagy, and Terry Smith who reviewed the book; and to Peter Gordon, John Remington, and Keith Wollman at Addison-Wesley Publishing Company for their encouragement and confidence in this project.

Aside from the individuals already named, I have also benefited from discussions with many people over the past years. They have commented on various parts of the book and include Chuan-Heng Ang, Walid Aref, James Arvo, Thor Bestul, Sharat Chandran, Chiun-Hong Chien, Jiang-Hsing Chu, Leila De Floriani, Roger Eastman, Herbert Edelsbrunner, Christos Faloutsos, George (Gyuri) Fekete, Kikuo Fujimura, John Gannon, John Goldak, Erik Hoel, Liuqing Huang, Frederik W. Jansen, Ajay Kela, David Kirk, Per Åke Larson, Dani Lischinski, Don Meagher, David Mount, Randal C. Nelson, Glenn Pearson, Ron Sacks-Davis, Timos Sellis, Clifford A. Shaffer, Deepak Sherlekar, Li Tong, Brian Von Herzen, Peter Widmayer, and David Wise. I deeply appreciate their help.

CONTENTS

INTRODUCTION 1

There are numerous hierarchical data structuring techniques in use for representing spatial data. One commonly used technique that is based on recursive decomposition is the quadtree. It has evolved from work in different fields. Thus it is natural that a number of adaptations of it exist for each spatial data type. Its development has been motivated to a large extent by a desire to save storage by aggregating data having identical or similar values. However, we will see that this is not always the case. In fact, the savings in execution time that arise from this aggregation are often of equal or greater importance.

This chapter contains a brief overview of hierarchical data structures such as the quadtree and octree. The focus is on the representation of regions. The goal is to define the representations, and their key properties, that are used in the applications described in the remaining chapters. (For more details on these individual representations of spatial data, see [Same90a].)

The rest of this chapter is organized as follows. It starts with some basic definitions followed by an outline of key properties of the quadtree and octree data structures. Next some of the most important quadtree-based representations for point and line data, as well as three-dimensional regions, are reviewed. This is followed by a brief history of the development of the quadtree and octree. Finally, a description is given of an implementation of a quadtree (and octree) as a tree. Much of this chapter is a summary of Chapter 1 of [Same90a].

1.1 BASIC DEFINITIONS

Let us first define a few terms with respect to two-dimensional data. Assume the existence of an array of picture elements (termed *pixels*) in two dimensions. We use the term *image* to refer to the original array of pixels. If its elements are either black or white, it is said to be *binary*. If shades of gray are possible (i.e., gray levels), the

image is said to be a *gray-scale* image. In the discussion, we are primarily concerned with binary images. Assume that the image is on an infinite background of white pixels. The *border* of the image is the outer boundary of the square corresponding to the array.

Two pixels are said to be 4-*adjacent* if they are adjacent to each other in the horizontal or vertical direction. If the concept of adjacency also includes adjacency at a corner (i.e., diagonal adjacencies), then the pixels are said to be 8-*adjacent*. A set S is said to be *four-connected* (*eight-connected*) if for any pixels p, q in S there exists a sequence of pixels $p = p_0, p_1, \cdots, p_n = q$ in S, such that p_{i+1} is 4-adjacent (8-adjacent) to p_i, $0 \le i < n$.

A black *region*, or black four-connected *component*, is a maximal four-connected set of black pixels. The process of assigning the same label to all 4-adjacent black pixels is called *connected component labeling* (see Chapter 5). A white *region* is a maximal *eight-connected* set of white pixels defined analogously. The complement of a black region consists of a union of eight-connected white regions. Exactly one of these white regions contains the infinite background of white pixels. All the other white regions, if any, are called *holes* in the black region. The black region, say R, is surrounded by the infinite white region, and R surrounds the other white regions, if any.

A pixel is said to have four edges, each of which is of unit length. The *boundary* of a black region consists of the set of edges of its constituent pixels that also serve as edges of white pixels. Similar definitions can be formulated in terms of rectangular blocks, all of whose pixels are identically colored. For example, two disjoint blocks, P and Q, are said to be 4-*adjacent* if there exists a pixel p in P and a pixel q in Q such that p and q are 4-adjacent. Eight-adjacency for blocks (as well as connected component labeling) is defined analogously.

1.2 PROPERTIES OF QUADTREES AND OCTREES

The term *quadtree* is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. They can be differentiated on the following bases:

1. The type of data they are used to represent.
2. The principle guiding the decomposition process.
3. The resolution (variable or not).

Currently they are used for point data, areas, curves, surfaces, and volumes. The decomposition may be into equal parts on each level (i.e., regular polygons and termed a *regular decomposition*), or it may be governed by the input. In computer graphics this distinction is often phrased in terms of image-space hierarchies versus object-space hierarchies, respectively [Suth74]. The resolution of the decomposition (i.e., the number of times that the decomposition process is applied) may be fixed

beforehand, or it may be governed by properties of the input data. Note that for some applications we can also differentiate the data structures on the basis of whether they specify the boundaries of regions (e.g., curves and surfaces) or organize their interiors (e.g., areas and volumes).

The first example of a quadtree representation of data is concerned with the representation of two-dimensional binary region data. The most studied quadtree approach to region representation, called a *region quadtree* (but often termed a *quadtree* in the rest of this chapter), is based on the successive subdivision of a bounded image array into four equal-sized quadrants. If the array does not consist entirely of 1s or entirely of 0s (i.e., the region does not cover the entire array), it is subdivided into quadrants, subquadrants, and so on, until blocks are obtained that consist entirely of 1s or entirely of 0s; that is, each block is entirely contained in the region or entirely disjoint from it. The region quadtree can be characterized as a variable resolution data structure.

As an example of the region quadtree, consider the region shown in Figure 1.1a represented by the $2^3 \times 2^3$ binary array in Figure 1.1b. Observe that the 1s correspond to picture elements (i.e., pixels) in the region, and the 0s correspond to picture elements outside the region. The resulting blocks for the array of Figure 1.1b are shown in Figure 1.1c. This process is represented by a tree of degree 4 (i.e., each nonleaf node has four sons).

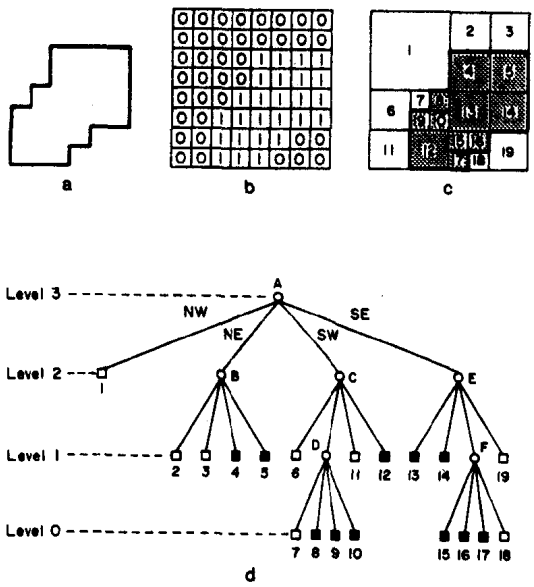

Figure 1.1 An example (a) region, (b) its binary array, (c) its maximal blocks (blocks in the region are shaded), and (d) the corresponding quadtree

In the tree representation, the root node corresponds to the entire array. Each son of a node represents a quadrant (labeled in order NW, NE, SW, SE) of the region represented by that node. The leaf nodes of the tree correspond to those blocks for which no further subdivision is necessary. A leaf node is said to be black or white depending on whether its corresponding block is entirely inside (i.e., it contains only 1s) or entirely outside the represented region (i.e., it contains no 1s). All nonleaf nodes are said to be gray (i.e., its block contains 0s and 1s). Given a $2^n \times 2^n$ image, the root node is said to be at level n while a node at level 0 corresponds to a single pixel in the image.[1] The region quadtree representation for Figure 1.1c is shown in Figure 1.1d. The leaf nodes are labeled with numbers, and the nonleaf nodes are labeled with letters. The levels of the tree are also marked.

The region quadtree is easily extended to represent three-dimensional binary region data, and the resulting data structure is called a *region octree* (termed an *octree* in the rest of this chapter). We start with a $2^n \times 2^n \times 2^n$ object array of unit cubes (termed *voxels* or *obels*). The octree is based on the successive subdivision of an object array into octants. If the array does not consist entirely of 1s or entirely of 0s, it is subdivided into octants, suboctants, and so on, until cubes (possibly single voxels) are obtained that consist of 1s or of 0s; that is, they are entirely contained in the region or entirely disjoint from it.

This subdivision process is represented by a tree of degree 8 in which the root node represents the entire object and the leaf nodes correspond to those cubes of the array for which no further subdivision is necessary. Leaf nodes are said to be black or white (alternatively, full or void) depending on whether their corresponding cubes are entirely within or outside the object, respectively. All nonleaf nodes are said to be gray. Figure 1.2a is an example of a simple three-dimensional object, in the form of a staircase, whose octree block decomposition is given in Figure 1.2b and whose tree representation is given in Figure 1.2c.

At this point, it is appropriate to justify the use of a quadtree decomposition into squares. Of course, there are many planar decomposition methods. Squares are used because the resulting decomposition satisfies the following two properties:

1. It yields a partition that is an infinitely repetitive pattern so that it can be used for images of any size.
2. It yields a partition that is infinitely decomposable into increasingly finer patterns (i.e., higher resolution).

A quadtree-like decomposition into four equilateral triangles (Figure 1.3a) also satisfies these criteria. However, unlike the decomposition into squares, it does not have a uniform orientation—that is, all tiles with the same orientation cannot be mapped into each other by translations of the plane that do not involve rotation or

[1] Alternatively, we can say that the root node is at depth 0 while a node at depth n corresponds to a single pixel in the image. In this book both concepts of level and depth are used to describe the relative position of nodes. The one that is chosen is context dependent.

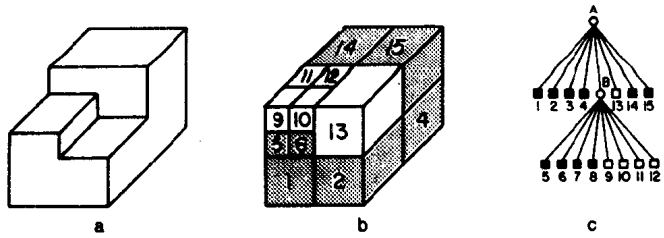

Figure 1.2 (a) Example three-dimensional object, (b) its octree block decomposition, and (c) its tree representation

reflection. In contrast, a decomposition into hexagons (Figure 1.3b) has a uniform orientation, but it does not satisfy property 2. For more details on the properties of decompositions see Bell, Diaz, Holroyd, and Jackson [Bell83] (and Section 1.4 of [Same90a]).

The prime motivation for the development of the quadtree is the desire to reduce the amount of space necessary to store data through the use of aggregation of homogeneous blocks. As we will see in subsequent chapters, an important by-product of this aggregation is the reduction of the execution time of a number of operations (e.g., connected component labeling and component counting). However, a quadtree implementation does have overhead in terms of the nonleaf nodes. For an image with B and W black and white blocks, respectively, $4 \cdot (B + W)/3$ nodes are required. In contrast, a binary array representation of a $2^n \times 2^n$ image requires only 2^{2n} bits; however, this quantity grows quite quickly. Furthermore if the amount of aggregation is minimal (e.g., a checkerboard image), the quadtree is not very efficient.

The worst case for a quadtree of a given depth in terms of storage requirements occurs when the region corresponds to a checkerboard pattern, as in Figure 1.4. The amount of space required is obviously a function of the resolution (i.e., the number of levels in the quadtree), the size of the image (i.e., its perimeter), and its positioning in the grid within which it is embedded. As a simple example, Dyer [Dyer82] has shown

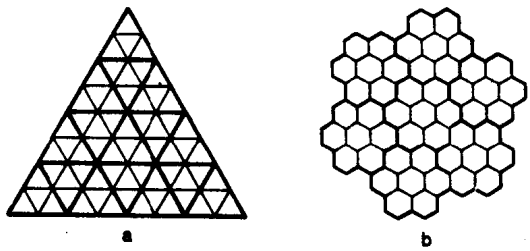

Figure 1.3 Example of nonsquare partitionings of the plane: (a) equilateral triangles and (b) hexagons

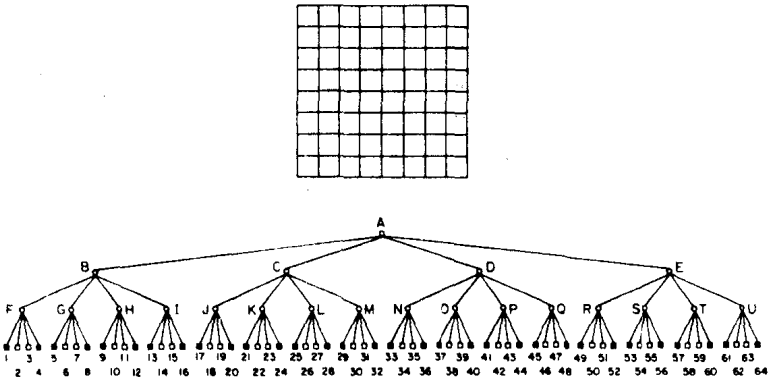

Figure 1.4 A checkerboard and its quadtree

that arbitrarily placing a square of size $2^m \times 2^m$ at any position in a $2^n \times 2^n$ image requires an average of $O(2^{m+2} + n - m)$ quadtree nodes. An alternative characterization of this result is that the amount of space necessary is $O(p + n)$ where p is the perimeter (in pixel widths) of the block.

Dyer's $O(p + n)$ result for a square image is merely an instance of the following theorem due to Hunter and Steiglitz [Hunt78, Hunt79a] who obtained the same result for simple polygons (i.e., polygons with nonintersecting edges and without holes). In fact, this result has been observed to hold in arbitrary images (see [Rose82b] for empirical results in a cartographic environment).

Theorem 1.1 The quadtree corresponding to a polygon with perimeter p embedded in a $2^n \times 2^n$ image has a maximum of $24 \cdot n - 19 + 24 \cdot p$ (i.e., $O(p + n)$) nodes. □

Hunter and Steiglitz represent a polygon by a three-color variant of the quadtree. It has three types of nodes: interior, boundary, and exterior. A node is said to be of type *boundary* if an edge of the polygon passes through it. Boundary nodes are not subject to merging. *Interior* and *exterior* nodes correspond to areas within, and outside, respectively, the polygon and can be merged to yield larger nodes. The resulting quadtree is analogous to the MX quadtree representation of point data described in Section 1.3, and this term will be used to describe it. In particular, boundary nodes are analogous to black nodes, while interior and exterior nodes are analogous to white nodes.

Figure 1.5 illustrates a sample polygon and its MX quadtree. One disadvantage of the MX quadtree representation for polygonal lines is that a width is associated with them, whereas in a purely technical sense these lines have a width of zero. Also, shifting operations may result in information loss. For more appropriate representations of polygonal lines, see Chapter 4 of [Same90a].