Algorithmica (1992) 8:39–54

Algorithmica

Linear-Time Border-Tracing Algorithms for Quadtrees[1]

Robert E. Webber[2] and Hanan Samet[1,3]

Abstract. In applications where the quadtree is used as an underlying object representation, a number of basic operations are implemented as a trace along the border of the object's region. A technique is presented that determines a way to shift any given scene (as well as its quadtree), so that the border of all the objects in the scene can be traversed in time proportional to the length of all the borders in the scene (or the number of blocks when the scene is represented as a quadtree). This determination is shown to be performed in time proportional to the length of all the borders in the scene. This allows the direct translation of a number of chain-code algorithms into quadtree algorithms without loss of asymptotic worst-case efficiency. This results in improved worst-case analyses of algorithms that convert chain codes into quadtrees and that perform connected component labeling of images represented as quadtrees.

Key Words. Quadtrees, Chain codes, Worst-case analysis, Connected component labeling.

1. Introduction. The quadtree [18], [17] is a hierarchical variable-resolution data structure designed for efficient manipulation of planar geometric objects. In essence it provides a technique for sorting the objects. It is frequently used in applications in computer cartography, computer graphics, computer vision, and robotics. Many operations that use the quadtree object representation are implemented as a trace of the border of the object's region. These algorithms are also used when objects are represented by their borders (e.g., using a chain code representation [3]).

Hunter and Steiglitz [5] have shown that the number of nodes in a quadtree is proportional to the size of the chain-code representation of the same object. Prior attempts to implement algorithms formulated for the chain code using the quadtree as the underlying representation (e.g., by neighbor finding) yield an inferior worst-case complexity (although the average-case complexities were similar for the two representations). In this paper, we demonstrate that, with some minor preprocessing of the quadtree, it is possible to achieve worst-case complexity of the quadtree implementation of a given algorithm to be asymptotically equivalent to the chain-code implementation of the same algorithm.

The rest of this paper is organized as follows. Section 2 contains background material. In particular, we explain the notation that we use. We also cite the main

[1] The support of the National Science Foundation under Grant IRI-88-02457 is gratefully acknowledged.
[2] Department of Computer Science, Middlesex College, The University of Western Ontario, London, Ontario, Canada N6A 5B7.
[3] Computer Science Department, University of Maryland, College Park, MD 20742, USA.

Received November 15, 1987; revised June 10, 1988, November 21, 1989, and July 9, 1990. Communicated by David P. Dobkin.

theorems from the quadtree literature that are used later. Sections 3, 4, and 5 show how to shift[4] a quadtree so that the border of the represented object can be followed in time proportional to the length of the object's perimeter. Although these sections are central to the results of this paper, they do not actually discuss quadtree algorithms. Instead, they focus on the computation of the relative position of the chain codes that describe the border of the objects in a scene and origin (i.e., the lower left-hand corner) of the quadtree being used to represent that scene. Once the position of the origin has been calculated, the normal chain code to quadtree algorithm [14] executes in linear time. If the data is already in the form of a quadtree, then use of a linear time quadtree shifting algorithm [21] leads to a linear conversion algorithm. Of course, the computation of the location of the origin must also be performed in linear time in order for the entire process to be accomplished in linear time.

As we mentioned above, Section 3 considers the basic question of how to determine the location for an arbitrary chain code so that it can be traversed in time proportional to its length. This is the central result. However, in order to apply it to border-following algorithms, a few more observations are necessary. This is the subject of Sections 4 and 5. Specifically, Section 4 generalizes the results of Section 3 to take into account the sequence of nodes that an algorithm would visit if it were traversing a perimeter corresponding to a specific chain code. Section 4 also considers the implications on the algorithm of Section 3 of nodes along the perimeter that have different sizes. Section 5 assumes that the position of the origin is calculated from a quadtree representation of the object instead of from a chain code representation. This is equivalent to permitting the elements of the chain code, which serves as the starting point of Sections 3 and 4, to be of varying length. Section 6 summarizes the significance of these results. It also discusses extensions of these results to various applications.

2. Background.

The quadtree is a hierarchical data structure formed by recursively subdividing a geometric space. There are many criteria that can be used as a basis for determining whether or not to further subdivide a space. We shall work with the simplest and most common criterion,[5] i.e., that a square region is recursively subdivided into four square subregions until the resulting regions are homogeneous. The four square subregions correspond to the four quadrants (and hence are labeled *NW*, *NE*, *SW*, and *SE*). The leaf nodes of a quadtree built from such a subdivision process are shown in Figure 1.

We view a quadtree, say Q, as representing a grid of $2^q \times 2^q$ cells. The width

[4] By "shift" we mean a rigid movement in a direction parallel to either the x or y axis (or a combination of such movements).

[5] The regular recursive decomposition of squares into squares has many special properties that distinguish it from other tessellations [1]. The regular recursive decomposition of a square into subsquares was independently developed in many fields, e.g., pattern recognition [9], [10], robotics [12], and computer graphics [23], [22].

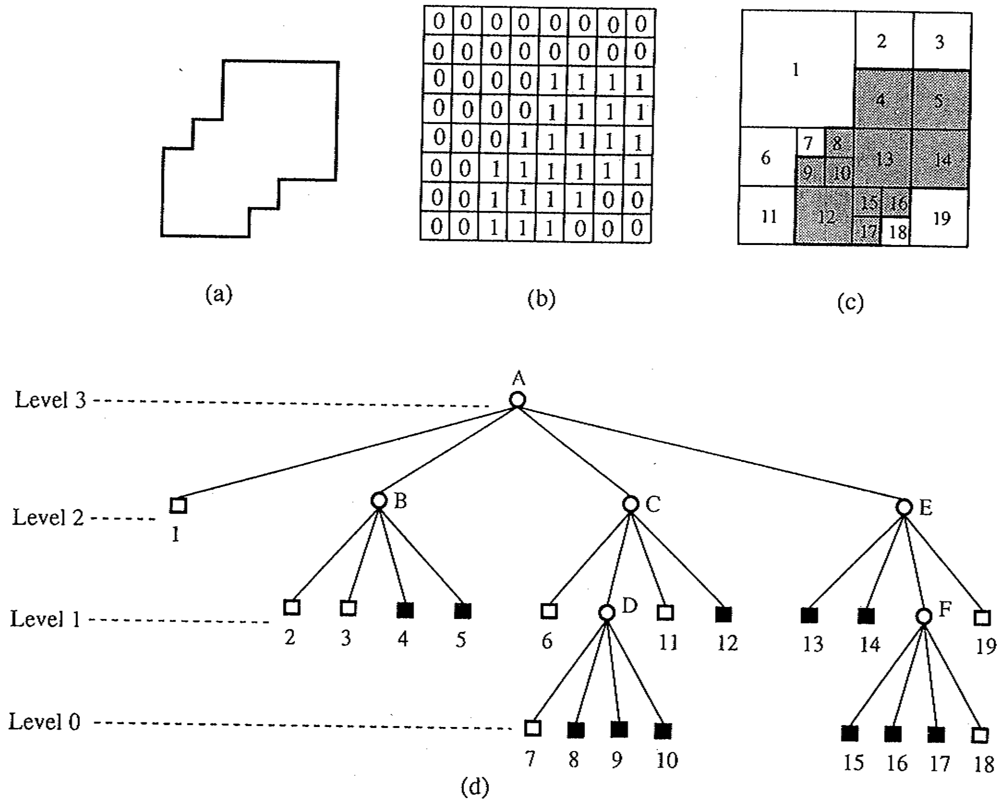

Fig. 1. (a) A region; (b) its binary array; (c) its maximal blocks; and (d) corresponding quadtree.

of this grid, i.e., 2^q, is also referred to as the width (denoted by $\mathscr{W}(Q)$) of the quadtree Q. The base two logarithm of $\mathscr{W}(Q)$ corresponds to the maximum depth of the quadtree Q and is denoted by $\mathscr{D}(Q)$. The actual number of nodes in Q is denoted by $|Q|$.

We can map a polygon on a grid represented by the quadtree. If we mark each cell of the grid that lies on the interior or border of the polygon as black and each cell of the grid that lies exterior to the polygon as white, then we have a digitization of that polygon onto that grid. The *perimeter length* of a quadtree is the length of the perimeter of the scene represented by the quadtree. When a quadtree represents a single polygon, then the length of the perimeter of the polygons, measured in grid cell widths, is called the perimeter length of the quadtree. We shall denote the perimeter length of a quadtree Q by $\mathscr{P}(Q)$.

Converting a chain code to a quadtree in time proportional to the size of the chain code requires that the size of the quadtree be bounded from above by some constant times the size of the chain code. This result, called *Tree Complexity Bound Theorem*, was demonstrated by Hunter and Steiglitz [4], [6]. Specifically, they showed that:

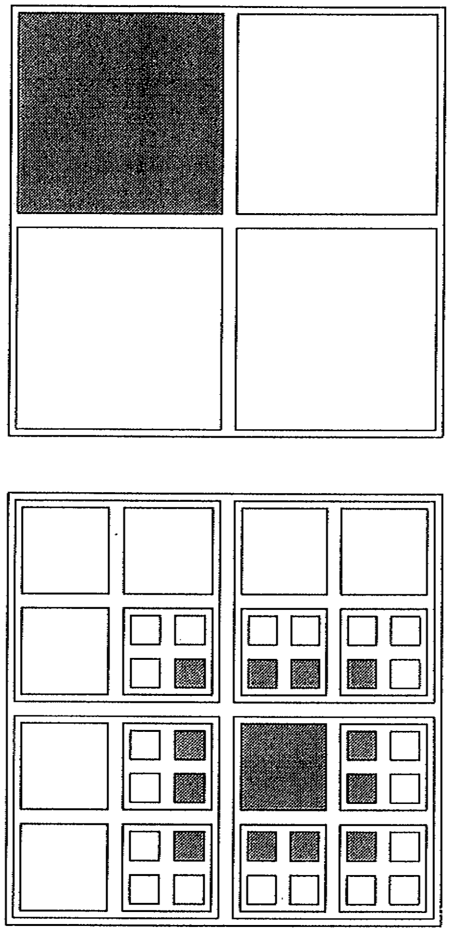
Fig. 2. Two quadtrees representing a 4×4 square in an 8×8 image.

The size of the quadtree representation Q of a digitized polygon is bounded from above by $16 \cdot \mathscr{P}(Q) - 11 + 16 \cdot \mathscr{D}(Q)$.

Since in general $\mathscr{P}(Q) \gg \mathscr{D}(Q)$, this result indicates that the size of a quadtree is $\mathbf{O}(\mathscr{P}(Q))$. This result is intuitively motivated by the observation that most of the nodes in a quadtree are at the maximum depth and lie along the perimeter of the objects being represented.

Although the Tree Complexity Bound Theorem yields a convenient upper bound, there is no corresponding lower bound result. Consider Figure 2, which shows two quadtrees built from the same polygon but with different relative offsets. We see that by shifting the polygon by three cells to the right and down, the number of the nodes in the tree can increase from 5 to 53. These differences can be arbitrarily large and depend on the origin of the quadtree. Dyer [2] found that, on the average, the number of nodes in a quadtree representing a randomly positioned rectangle is bounded both from above and below by a constant times the length of the perimeter of the rectangle. Assuming that a typical scene consists of a collection of randomly positioned rectangles, it would appear that there is little expectation of being able to significantly improve, via shifting, the size of a quadtree that represents a complicated scene.

Nevertheless, Li *et al.* [11] have developed a dynamic programming algorithm for determining the origin of a quadtree so as to minimize the number of nodes that represent a given scene. The worst-case time complexity for applying their algorithm to an arbitrary quadtree Q is $\mathbf{O}(\mathscr{W}(Q)^2 \cdot \mathscr{D}(Q))$. This represents a significant amount of work, especially in light of the fact that little improvement will result in general. A key observation made in the analysis of their algorithm was that

> The number of nodes representing subtrees of width 2^q is not changed by shifting the quadtree's center to the north, south, east, or west by the amount 2^r when $r \geq q$.

This observation plays an important role in the algorithm that we present in Section 5.

Algorithms that manipulate quadtrees are often implemented as tree traversals. The two most common types of tree traversals are top-down traversals and border-following traversals. Top-down traversal algorithms have been extensively studied [8], [16]. Such algorithms are based on viewing a task as a composition of the result of computing the task at each of the subtrees of the current node.

Sometimes a top-down algorithm is inefficient. For example, although the chain code for a quadtree can be viewed as the merger of the chain codes corresponding to each of its subtrees, such an approach requires an auxiliary structure for merging the chain codes of neighboring subtrees as well as space to store all the chain codes. On the other hand, a bottom-up approach exemplified by a border-following algorithm can construct the chain codes on the fly as it traverses the borders of each of the regions in a scene.

Border-following algorithms require that we be able to locate neighbors efficiently. In particular, we must be able to get from one leaf node to the neighboring leaf node on one of its sides, a process called *neighbor finding*. Samet and Shaffer [14], [19] show that on the average 3.5 quadtree links are dereferenced to get from a leaf node to its neighbor using neighbor finding.[6]

Neighbor-finding in a quadtree corresponds to finding the shortest path between two leaf nodes in the graph representing the structure of the quadtree itself. This shortest path ascends the links of the quadtree until it reaches the nearest common ancestor of both the node and its neighbor and then descends to the neighbor.

[6] It has been noted [7], [16], [20] that some restricted kind of neighbor-finding can be done using top-down algorithms because the neighbors of a node Q are either children of the parent of Q or children of a neighbor of a parent of Q. Thus top-down algorithms can be implemented so that at each node the value computed is a function of the node itself and the value of its immediate neighbors of equal or greater size. However, this approach does not yield a general border-following algorithm such that its worst-case execution time is linear in the length of the perimeter.

The exact mechanics of general neighbor-finding are described elsewhere [14], [15]. Neighbor-finding takes time proportional to the length of the path from the node to the nearest common ancestor and back down to the neighbor.

In the worst case, neighbor finding in a quadtree Q could be $2 \cdot \mathscr{D}(Q)$ or $\mathbf{O}(\log \mathscr{P}(Q))$. For a severely unbalanced tree it can be as large as $\mathbf{O}(|Q|)$. Thus, the worst-case analysis for a perimeter-following algorithm based on neighbor-finding would be $\mathbf{O}(\mathscr{P}(Q) \cdot \log \mathscr{P}(Q))$ or $\mathbf{O}(|Q|^2)$. In the rest of this paper we show how to improve on this result by shifting the location of the origin of the scene before constructing the quadtree.

3. Positioning when All Nodes Adjacent to the Border Are of Unit Size.

Before considering the positioning of a polygon (e.g., Figure 3), let us consider a simpler case. In Figure 4, we have a simple path that snakes back and forth across the middle of a 16×16 grid. In this section, we assume that every cell in the original grid through which the chain code corresponding to the path passes is a leaf node in the quadtree, i.e., none of the cells merge to form larger quadtree blocks. We use the term *links* to refer to the individual parts of the path and polygon that crosses each grid line.

The cost of traversing such a path can be broken up into two parts: the cost of finding all the horizontal neighbors and the cost of finding all the vertical neighbors. The cost of finding a neighbor in a quadtree is the number of links that must be traversed in order to find it. For the path in Figure 4, the cost of finding the horizontal neighbor is always the same, i.e., $2 \cdot \log 16$ which is 8. The cost of finding the vertical neighbor is a bit more complex to calculate. One of the vertical neighbors has a cost of 8 associated with it; two of them cost 6; four of them cost 4; and eight of them cost 2. The average cost of finding a vertical neighbor for our example is $\sum_{j=1}^{q} (2^{q-j} \cdot 2 \cdot j)/(2^q - 1) \leq 4$.

The optimal positioning for the path of Figure 4 is shown in Figure 5. The cost of finding vertical neighbors is unchanged, but the cost of finding each of the

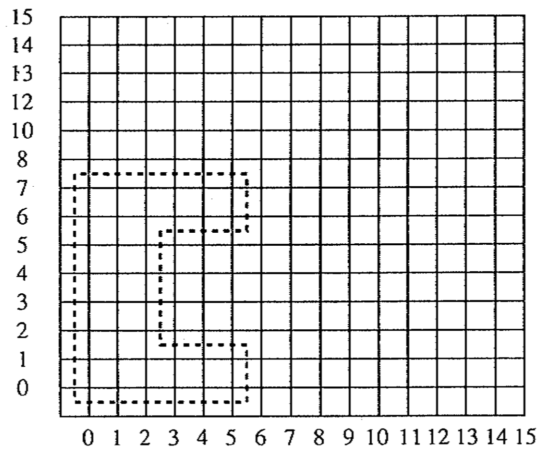

Fig. 3. A polygon.

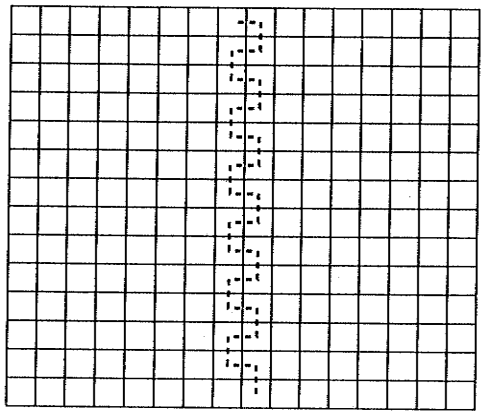
Fig. 4. A snake-like path.

horizontal neighbors is now 2. Note that it is not possible to find a way to shift this path so that the cost of finding each vertical neighbor is also 2. However, observe that the cost is 2 whenever the neighbor (horizontal or vertical) computation crosses an even-numbered grid line (using the numbering conventions of Figure 3). Thus it is possible to find a position for the origin so that at least half of the neighbors can be computed with cost 2. Indeed, all that needs to be done to do this is to count the number of neighbors whose computation requires crossing "odd numbered" grid lines and those that require crossing "even numbered" grid lines, and then determine whether or not it is necessary to shift the chain code by 1 in order to place the majority of the links on the "even numbered" grid lines.

Applying the above techniques to Figure 3 requires more work. Of the remaining links that we have not set to cost 2 (which is less than or equal to half the total number of links in the given direction), every other grid line has a cost of 4. Thus

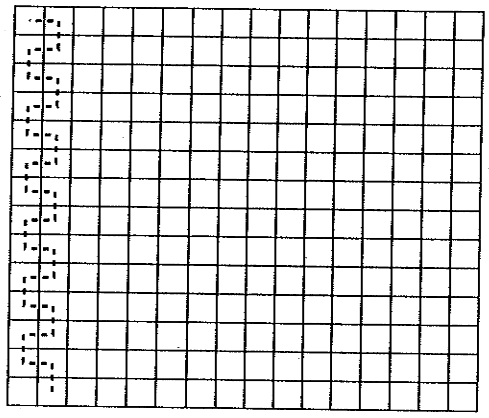
Fig. 5. Optimal positioning of the path in Figure 4.

particular, instead of using 2^k links of unit length to represent the border of a node of width 2^k, we use one link that is marked as being 2^k long.

Now, let us consider how to incorporate a link of length 2^k in the algorithm of Section 4. In particular, we notice that a link of length 2^k crosses each of 2^k consecutive "grid lines" exactly once. Thus it plays no role in determining the shifts for the passes corresponding to the frequency of the costs $2, 4, 6, \ldots, 2 \cdot k$ of the neighbor-finding operation. Moreover, for all passes dealing with neighbor-finding costs of $2 \cdot (k + 1)$ and higher, a link of length 2^k counts as one crossing in the group appropriate to its location.

Extending this observation to a collection of links of varying lengths corresponding to borders along leaf nodes of the same lengths, we see that we can start with $\mathscr{D}(Q)$ separate lists where list k contains links of length 2^{k-1} $(k \geq 1)$. When calculating whether or not it is necessary to make a shift of length 2^{k-1} to ensure that the larger group of links gets the cost $2 \cdot k$, it is only necessary to examine the links in list k and the links that remain from the immediately preceding pass. Using the fact that the number of variable size links is proportional to the number of quadtree nodes, it can be shown that the amount of work being done to calculate the new position is $\mathbf{O}(|Q|)$.

The analysis of the quadtree-to-quadtree variation is completed by showing that we can shift the quadtree Q into the location of the quadtree Q_{EPB} in time proportional to $|Q| + |Q_{EPB}|$ [21]. This results in a positioning algorithm that executes in worst-case time $\mathbf{O}(|Q| + |Q_{EPB}|)$. Thus the entire extended-perimeter path-balancing operation on a quadtree Q can be performed in worst-case execution time $\mathbf{O}(|Q| + |Q_{EPB}|)$.

6. Concluding Remarks. In this paper, we have presented a new quadtree transformation called path-length balancing. It allows us to place a worst-case linear upper bound on the execution time of quadtree algorithms that require the border of a region to be followed. For example, converting from a chain code to a quadtree (and vice versa) can be performed in time proportional to the number of nodes in the transformed quadtree (alternatively in time proportional to the length of the chain code representation of the objects). Previously this bound was only achieved on the average [14].

As another example, consider connected component labeling [13]. An algorithm can be devised that traverses the quadtree in a top-down order. Each pair of nodes of differing colors imply the presence of a boundary. Thus, when such a pair is encountered, the top-down traversal is interrupted and this boundary is followed with all nodes along it being assigned the same label if they have not been labeled already. Once a boundary has been fully followed, the top-down traversal continues. A second top-down traversal propagates the colors of the nodes along the boundary inwards. This is facilitated by transmitting each node's neighbors as part of the traversal [16]. The key to the execution time analysis of this method is that use of the path length balancing can be achieved in time proportional to the number of nodes in the quadtree. For more details about this algorithm, including the extension of path balancing to quadtrees containing many disconnected regions, see [24].

Techniques such as those described in this paper serve to demonstrate that the cost of quadtree algorithms is not significantly influenced by the maximum depth of the quadtree. Instead, the cost of these algorithms depends on the number of nodes in the quadtree. In particular, we have seen that use of an asymptotic worst-case analysis does not yield significantly different results from an average case analysis when measuring the performance of quadtree algorithms.

References

[1] N. Ahuja. On approaches to polygonal decomposition for hierarchical image representation. *Computer Graphics, Vision, and Image Processing*, **24** (1983), 200–214.

[2] C. R. Dyer. Space efficiency of region representation by quadtrees. *Computer Graphics and Image Processing*, **19** (1982), 335–348.

[3] H. Freeman. Computer processing of line-drawing images. *ACM Computing Surveys*, **6** (1974), 57-97.

[4] G. M. Hunter. *Efficient computation and data structures for graphics.* Ph.D. thesis, Department of Electrical Engineering and Computer Science, Princeton University, (1978).

[5] G. M. Hunter and K. Steiglitz. Liner transformation of pictures represented by quadtrees. *Computer Graphics and Image Processing*, **10** (1979), 289–296.

[6] G. M. Hunter and K. Steiglitz. Operations on images using quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **1** (1979), 145–153.

[7] C. L. Jackins and S. L. Tanimoto. Quad-trees, oct-trees, and k-trees—a generalized approach to recursive decomposition of euclidean space. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **5** (1983), 533–539.

[8] E. Kawaguchi, T. Endo, and M. Yokota. Depth-first expression viewed from digital picture processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **5** (1983), 373–384.

[9] A. Klinger. Patterns and search statistics. In *Optimizing Methods in Statistics*, (J. S. Rustagi, ed.) Academic Press, New York, (1971).

[10] A. Klinger and C. R. Dyer. Experiments in picture representation using regular decomposition. *Computer Graphics and Image Processing*, **5** (1976), 68–105.

[11] M. Li, W. Grosky, and R. Jain. Normalized quadtrees with respect to translation. *Computer Graphics and Image Processing*, **20** (1982), 72–81.

[12] N. J. Nilsson. A mobile automaton: an application of artificial intelligence techniques. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 509–520, Washington, D.C., (1969).

[13] A. Rosenfeld and A. C. Kak. *Digital Picture Processing, 2nd ed.* Academic Press, New York, (1982).

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

[15] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, **16** (1984), 187–260.

[16] H. Samet. A top-down quadtree traversal algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **7** (1985), 94–98.

[17] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, (1989).

[18] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, (1989).

[19] H. Samet and C. A. Shaffer. A model for the analysis of neighbor finding in pointer-based quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **7** (1985), 717–720.

[20] H. Samet and R. E. Webber. On encoding boundaries with quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **6** (1983), 365–369.

[21] H. Samet and R. E. Webber. Hierarchical data structures and algorithms for computer graphics. *IEEE Computer Graphics and Applications*, **8(3)** (1988), 48–68.

[22] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, **6** (1974), 1–55.

[23] J. E. Warnock. A hidden surface algorithm for computer generated halftone pictures. Technical Report 4-15, Computer Science Department, University of Utah (1969).

[24] R. E. Webber. *Analysis of Quadtree Algorithms.* Ph.D. thesis, Computer Science Department, University of Maryland (1983).