COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING 50, 29–49 (1990)

Set Operations for Unaligned Linear Quadtrees*

CLIFFORD A. SHAFFER

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061

AND

HANAN SAMET

Computer Science Department, Center for Automation Research and Institute for Advanced Studies, University of Maryland, College Park, Maryland 20742

Received July 1, 1988; revised May 1, 1989

An algorithm is presented that performs set operations (e.g., union or intersection) on two unaligned images represented by linear quadtrees. This algorithm seeks to minimize the number of nodes that must be searched for or inserted into the disk-based node lists that represent the trees. Windowing and matching operations can also be cast as unaligned set functions; these operations can then be solved by similar algorithms. © 1990 Academic Press, Inc.

1. INTRODUCTION

Creating the union or intersection of two maps (sometimes referred to as *polygon overlay*) is a fundamental operation performed by Geographic Information Systems (GIS). These operations allow the user to determine spatial relationships between objects without requiring that such relationships be explicitly stored by the database. For example, a map of all wheatfields above 100 ft in elevation can be produced by intersecting a wheatfield map with an elevation map whose regions representing elevations below 100 ft are WHITE (or empty) and whose regions above 100 ft are non-WHITE. The resulting map would have non-WHITE regions wherever the corresponding regions of both input maps are non-WHITE. The quadtree data structure has been used successfully as the underlying representation for maps in several experimental GIS [18, 24, 11, 3]. One variant, named the *region quadtree* [17] decomposes an image into homogeneous blocks. If the image is all one color, then it is represented by a single block. If not, then the image is decomposed into quadrants, sub-quadrants, ..., until each block is homogeneous. Figure 1 illustrates the region quadtree. The region of Fig. 1(a) is represented by a binary pixel array in Fig. 1b. The resulting quadtree block decomposition is shown in Fig. 1c, and the corresponding tree structure in Fig. 1d. For the sake of brevity, we will use the generic term "quadtree" to refer specifically to the region quadtree. When a quadtree is represented by means of a tree structure with explicit pointers from a node to its children, it is referred to as a *pointer-based quadtree*.

Maps are sufficiently large that they normally cannot be represented in the computer's primary memory. The *linear quadtree* technique [5, 1] has gained much use since it allows the storage of quadtrees in such a way that they may be efficiently manipulated when stored in disk files. The linear quadtree represents the position of

*The support of the National Science Foundation under Grant ISI-8802457 is gratefully acknowledged.

0734-189X/90 $3.00

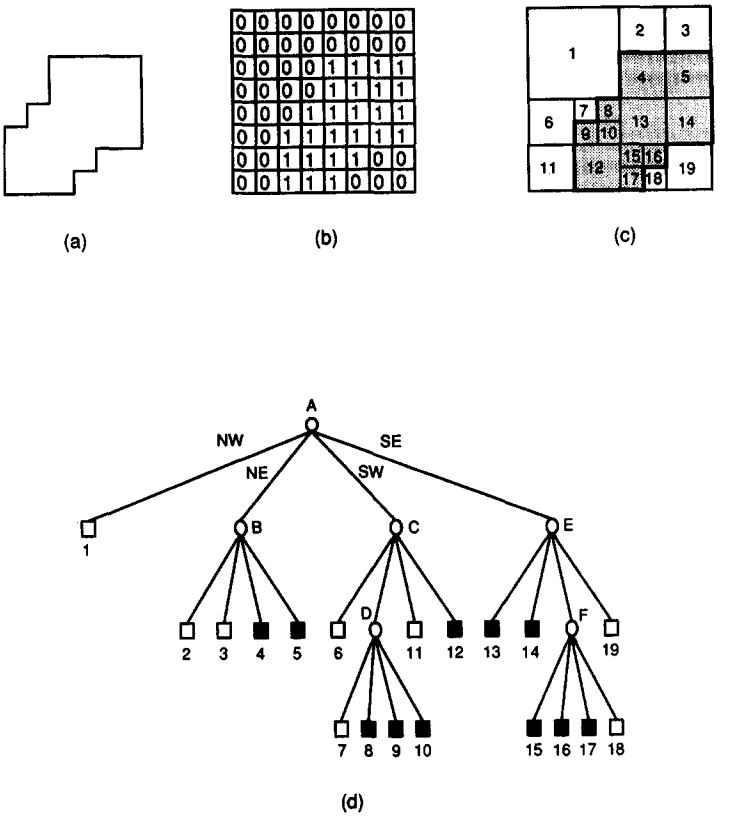

FIG. 1. A region, its binary array, its maximal blocks, and the corresponding quadtree: (a) region; (b) binary array; (c) block decomposition of the region in (a); blocks in the region are shaded; (d) quadtree representation of the blocks in (c).

each leaf node by use of a locational code corresponding to a sequence of directional codes that locate the leaf along a path from the root of the tree, as illustrated in Fig. 2. These locational codes can also be derived by interleaving the bits from the binary representation of the (*row*, *col*) coordinates of the blocks' upper left pixel. These locational codes will be referred to as *Morton sequences*, after the originator of their first use with GIS [12]. The resulting collection of quadtree leaf nodes is usually stored as a list sorted in increasing order of locational codes; this ordering is referred to as *Morton sequence ordering*. Such an ordering is useful because it reproduces the order in which the leaf nodes of the pointer-based quadtree would be visited by a depth-first traversal. Thus, this technique converts the pointer-based tree representation into a sorted list of leaf nodes which can be manipulated by traditional indexing schemes such as the B-tree [4].

The algorithm for computing the union or intersection of aligned quadtrees, i.e., quadtrees whose image size, pixel size, origin, and rotation angle with respect to the coordinate axes are the same, is quite straightforward [9, 26]. In fact, it was one of the "classic" quadtree algorithms that sparked initial interest in the quadtree's use. Its time complexity is proportional to the sum of the number of blocks in the two

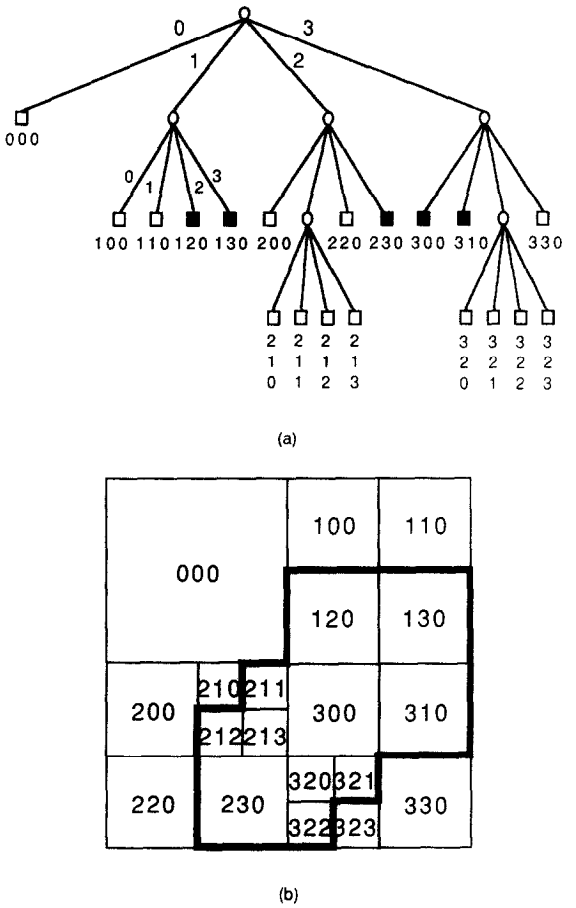

FIG. 2. Morton code addressing for the quadtree of Fig. 1: (a) the tree structure with the corresponding directional codes at each branch; (b) the resulting block decomposition with each node labeled by its upper left pixel's Morton code.

input trees. In this paper, we show that set functions for unaligned quadtrees, i.e., quadtrees whose image size and origins may differ, can be computed in the time required for a single access to each node in the two input trees plus a single write for each node in the output tree. In particular, we present an algorithm for generating the linear quadtree that is the result of computing the intersection of a pair of unaligned linear quadtrees. In Section 3, we will further generalize the operation by describing a method for performing set operations on maps with differing rotation angles with respect to the coordinate axes.

Most of the algorithms presented here, while modifiable for implementation with pointer-based quadtrees, are described using linear quadtree operations. One advantage of this approach is that we can discuss these algorithms in terms of searches and insertions into an abstract data type (conceptually a sorted list), rather than lower level tree manipulation functions. Our linear quadtree algorithms access the node list through two functions, OUTPUT(*tree*,*row*,*col*,*level*,*value*) and FIND(*tree*,*node*,*row*,*col*). OUTPUT inserts into linear quadtree *tree* a node with

level *level*, upper left corner at (*row*,*col*), and node value *value*. All algorithms discussed here output nodes strictly in Morton sequence order. FIND locates in *tree* the leaf node containing (*row*,*col*), returning a description of this node in *node*.

The remainder of this paper is organized as follows. Section 2 contains our unaligned set operation algorithm. Section 3 shows how to adapt this algorithm to map windowing. Section 4 adapts our algorithm to template matching. Section 5 presents our conclusions.

2. COMPUTING THE INTERSECTION OF UNALIGNED MAPS

In this section we present an algorithm to compute the intersection of two images represented by linear quadtrees. Other set functions such as union or difference can be handled in an analogous manner. The intersection of two quadtrees representing images with the same grid size, same map size, and same origin can be implemented by a simple depth-first traversal of both trees in parallel. Each node of the first image is compared with the corresponding node(s) in the second image. Algorithm 1 of the Appendix, named SIMPLE_INTERSECT, illustrates this procedure. For clarity, it has been formulated for operation on pointer-based quadtrees.

Little work has appeared on set operations between unaligned quadtrees (i.e., quadtrees that have compatible grid sizes, but differing origins and/or map sizes). Earlier approaches to unaligned quadtree intersection involved translating one of the images to be aligned with the other, and then performing aligned intersection [6]. This section presents an algorithm for unaligned map intersection that performs a FIND operation on each node of the two input quadtrees at most once, and performs at most one OUTPUT operation for each node in the resulting quadtree. Image translation will be discussed in Section 3.

The unaligned intersection algorithm works as follows. We select one of the input quadtrees to be the *aligned* quadtree, named QA. The other input quadtree is the *unaligned* quadtree, named QU. The nodes in the aligned quadtree are processed in Morton order. For each node N of QA, the various nodes of QU that overlap N are located. The algorithm starts with N's upper left pixel and finds the node of QU, say M, that covers that corner. Next, it determines the largest block, say B, that is contained in both N and M. Finally, it computes the appropriate set operation (intersection in this case) for the values of N and M, sets B to its result, and transmits B to an output function which is responsible for creating the nodes in the final quadtree. This processing sequence is repeated for the remaining unprocessed pixels in N by proceeding in Morton order. The key to making the algorithm efficient is to minimize the number of FIND operations performed on QU as well as the number of OUTPUT operations.

As an example, let the simple block decomposition shown in Fig. 3a be QU. Let the square region within the broken lines, labeled N, be the first node from QA and assume that it is BLACK. The algorithm compares N against each node in QU that it covers (i.e., B, E, G, H, I, and J). Figure 3b shows the resulting sub-blocks.

Note that sub-blocks 6, 7, 8, and 9 in Fig. 3b will form a single WHITE block in the output tree, and likewise for sub-blocks 10, 11, 12, and 13. We say that a block of the output quadtree is *active* if it is possible for it to merge with its siblings to form a larger block. Since our goal is to minimize the number of OUTPUT operations, we do not want to output the active blocks. Instead, we take advantage of the fact that the blocks of the output quadtree are generated in Morton order and

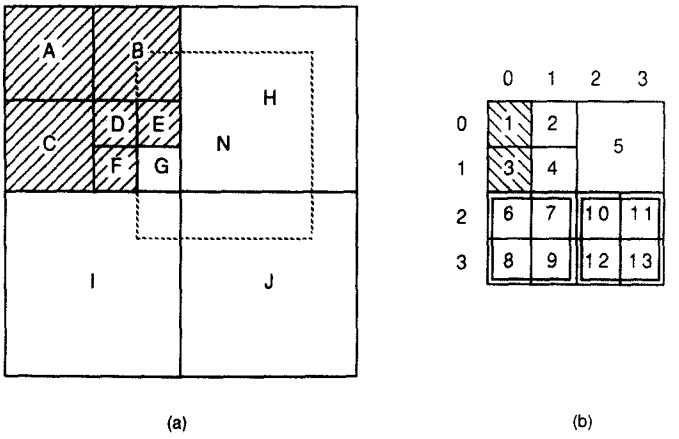

FIG. 3. An illustration of unaligned intersection: (a) the image to be intersected (QU), with the location of the first block (N) of QA superimposed; (b) the block decomposition for the first block of QA after intersection with the image in (a).

facilitate the grouping of identically colored blocks by maintaining a table named *outtab* of the active output tree blocks. Since the blocks in the output quadtree are generated in Morton order (matching the progress made in processing QA), four active blocks that make up a larger block must be adjacent in the list of all currently active output tree blocks created by the intersection algorithm. Four siblings of the same color are immediately replaced by their parent block when the fourth sibling is created. Once it is certain that active output tree blocks may not combine to form a larger block, then the contents of *outtab* can be converted to nodes of the linear quadtree and output by use of the OUTPUT function. This situation is quite easy to detect as it arises whenever the color of an active block is different from the color of the immediately preceding active block. The greatest possible number of active blocks occurs when the NW, NE, and SW quadrants of the image are all of color C, the first three subquadrants of the SE quadrant are also of color C, etc. This situation can continue for at most n levels for a $2^n \times 2^n$ image. Thus, *outtab* can contain a maximum of $3 \cdot n$ entries. Procedure ORDER_INSERT of Algorithm 2 encodes this method. Use of this procedure limits the number of calls to OUTPUT to be the number of blocks in the output tree.

The number of FIND operations is minimized by keeping track of which blocks in QU can possibly cover pixels in QA that have yet to be processed. We say that a block in QU is *active* if some, but not all, of its constituent pixels have already been used to cover processed blocks in QA. The blocks of QA are processed in Morton order. This means that the border of the region consisting of the blocks that have already been processed in QA (termed the *active border*) has the shape of a staircase. For example, consider Fig. 4, where the blocks have been assigned labels matching their Morton order. The heavy line represents the state of the active border after processing block 6. The broken line along the southern and eastern border of block 7 shows the change in the active border after processing block 7. The active border of a $2^n \times 2^n$ image consists of sets of horizontal and vertical segments such that the total length of each of these sets is 2^n pixel widths. At any

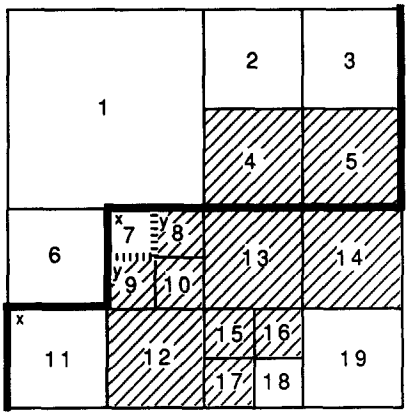

FIG. 4. The active border after processing node 6 where quadtree *QA* is the image of Fig. 1: "*x*" marks indicate possible locations for the next node to be processed; "*y*" marks indicate positions for table updating after processing node 7.

given instant, the active blocks of *QU* are those blocks that straddle (i.e., overlap) the active border of *QA*.

A description of the active border is maintained in two arrays, named *x_edge* and *y_edge*, each 2^n records long. Each record contains the descriptor for a quadtree node. Pixel-sized blocks are defined to be at level 0; a block equal in size to the entire image is at level n. The active border implementation described here is related to the implementation used by Samet and Tamminen [20]. The difference is that Samet and Tamminen's goal was to keep track only of the colors of the processed blocks that are adjacent to the border whereas in the present application we want to keep track of the blocks in *QU* that straddle the active border.

The unaligned intersection algorithm, named INTERSECT and presented as Algorithm 2 in the Appendix, works as follows. For each node N of *QA*, the function DOSET performs the following steps. Start with the pixel in the upper left corner of N (i.e., at address (cy, cx) with respect to an origin at the extreme NW corner of the image), find the block, say N', in *QU* that covers the pixel. This is achieved by examining *y_edge*[*cy*] and *x_edge*[*cx*]. There are three possible outcomes:

(1) Both *y_edge*[*cy*] and *x_edge*[*cx*] contain N'. Nothing else needs to be done.

(2) One of *y_edge*[*cy*] and *x_edge*[*cx*], but not both, contains N'. The record for N' is copied from one array to the other.

(3) Neither of *y_edge*[*cy*] and *x_edge*[*cx*] contain N' and thus we must search (i.e., use the FIND operation) *QU* for N'. In such a case, once N' has been found, it is inserted into both *y_edge*[*cy*] and *x_edge*[*cx*].

Once N' has been located, its size and position are compared with that of N to generate the largest block, say B, with an upper left corner at (cy, cx) that is contained in both N and N'. B is then passed to ORDER_INSERT. The process continues at the next unprocessed pixel in Morton order until reaching the pixel in the lower right corner of N. Note that no node in *QU* will ever be the subject of

more than one FIND operation. This follows from the fact that the pixels of each block in QA are processed in Morton order. Consider the side of the active border of QA that corresponds to the unprocessed image. At any instant, the next pixel to be processed can only be one of the pixels in the vertex of one of the convex angles formed by the segments on this side (e.g., the upper-leftmost corners of blocks 7 and 11 in Fig. 4). Assume that after processing pixel (cy, cx) both $y\_edge[cy]$ and $x\_edge[cx]$ contain a pointer to node N' in QU and that the block that has been generated is of width w. The new active border of QA will have at most two new convex angles—i.e., one at $(cy + w, cx)$ and one at $(cy, cx + w)$. When pixel $(cy, cx + w)$ is processed, $y\_edge[cy]$ still points at N' since no other pixel with y coordinate value cy was processed in the meantime. Similar reasoning applies to pixel $(cy + w, cx)$ and $x\_edge[cx]$.

Algorithm 2 encodes the unaligned intersection algorithm described above. To be precise, this algorithm produces an image whose value is WHITE at those pixels where either QU's corresponding value is WHITE or the pixel is beyond the borders of one of the images; the value is that of the corresponding position in QA for all other pixels. As an example of how the unaligned intersection algorithm operates, Table 1 illustrates the workings of this algorithm for Fig. 3. Recall that in this case the image in Fig. 3a is intersected with the BLACK region corresponding to the square within the broken lines. The image in Fig. 3a corresponds to QU while the BLACK square is the first node from QA. Table 1 shows the contents of the active border arrays after processing the upper left corner of each sub-block formed from node N in Fig. 3b.

Column 1 of the table indicates the coordinates of the pixel in QA that is currently being processed. Columns 2 and 3 correspond to the contents of *y_edge* and *x_edge* after processing the pixel specified in column 1. Column 4 shows the sub-block generated for that pixel. The entries in columns 2 and 3 are of the form $p : b$, where b is the block in QU that is stored in position p in the edge array. An asterisk (*) next to a block means that it has been located by use of a FIND operation. For example, when processing pixel (0, 0) the array is empty, and thus the record for block B (the block containing pixel (0, 0)) is loaded into $y\_edge[0]$ and $x\_edge[0]$. A minus (−) next to a block means that the corresponding *y_edge* (or *x_edge*) entry did not contain a block that covers the current pixel at (cy, cx), and that the block was copied from *x_edge* (or *y_edge*). A plus (+) appears next to the copied entry. For example, after processing pixel (2, 0) relative to the origin of QA (i.e., block 6 in Fig. 3b), $y\_edge[2]$ and $x\_edge[1]$ contain blocks G and H, respectively. Pixel (2, 1) is contained in block H and thus when it is processed, we simply copy H from $x\_edge[1]$ to *y-edge*[2]. Thus for pixel (2, 1) $y\_edge[2]$ and $x\_edge[1]$ are marked with (−) and (+), respectively. Note that upon processing pixel (3, 0), block G is still contained in $x\_edge[0]$ and thus it is replaced by block I. If G had been a large block covering pixel (3, 0), then G would have still been available and thus no change would have taken place.

Entries in column 4 marked with an asterisk (*) indicate those nodes which initiate calls to OUTPUT to clear table *outtab*. Such nodes are of a different color than their predecessor. Entries marked with a plus (+) indicate a SW sibling which merges with its like-colored siblings already in *outtab*.

Earlier we mentioned that procedure ORDER_INSERT only outputs blocks that cannot be aggregated to form larger blocks. As described above, OUTPUT is

TABLE 1
Trace Table for Intersection Active Nodes

Pixel processed	Active node tables		Output
	y_edge	*x_edge*	
(0, 0)	0: B*	0: B*	1
(0, 1)	0: H*	0: B	2*
		1: H*	
(1, 0)	0: H	0: E*	3*
	1: E*	1: H	
(1, 1)	0: H	0: E	4*
	1: H −	1: H +	
(0, 2)	0: H +	0: E	5
	1: H	1: H	
		2: H −	
(2, 0)	0: H	0: G*	6
	1: H	1: H	
	2: G*	2: H	
(2, 1)	0: H	0: G	7
	1: H	1: H +	
	2: H −	2: H	
(3, 0)	0: H	0: I*	8
	1: H	1: H	
	2: H	2: H	
	3: I*		
(3, 1)	0: H	0: I	9 +
	1: H	1: J*	
	2: H	2: H	
	3: J*		
(2, 2)	0: H	0: I	10
	1: H	1: J	
	2: H	2: H	
	3: J		
(2, 3)	0: H	0: I	11
	1: H	1: J	
	2: H +	2: H	
	3: J	3: H −	
(3, 2)	0: H	0: I	12
	1: H	1: J	
	2: H	2: J −	
	3: J +	3: H	
(3, 3)	0: H	0: I	13 + *
	1: H	1: J	
	2: H	2: J	
	3: J +	3: J −	

invoked as many times as there are blocks in the final quadtree. It is possible to design an algorithm that makes even fewer calls to OUTPUT. Such an algorithm would use a *largest block insertion* policy. This policy always inserts the largest block for which the current pixel is the upper-left corner, and keeps track of those nodes that have been inserted that are still active. Thus future pixels of the same color need not cause the insertion of more nodes. See [23] for an algorithm that makes use of this approach. While ORDER_INSERT makes more calls to OUTPUT than a largest node insertion policy, ORDER_INSERT is more appropriate when the underlying implementation can use the Morton sequence order of the calls to OUTPUT to optimize the construction process. For further discussions on the virtues of largest node insertion policies, see [22, 23].

The algorithm's execution time is $O(N_A + F(N_U) + M + M')$, where N_A and N_U denote the number of nodes in QA and QU, respectively; M and M' are the number of nodes in the output quadtree before and after merging. $F(N_U)$ is the cost for the FIND operations that must be performed on QU. N_A is the number of nodes in QA that must be processed. M is the number of times nodes are inserted in *outtab*. M is also the total number of subblocks that comprise the set of nodes of QA. Clearly, $N_A \leq M$. M' is the number of calls to procedure OUTPUT. The contribution of the factor M is overshadowed by M' since M represents the number of times that *outtab* is updated, while M' represents actual calls to OUTPUT. In particular, if the quadtree resides on disk, then calls to OUTPUT require disk accesses, whereas updates to *outtab* are performed in core. Thus we get an asymptotic execution time of $O(N_A + F(N_U) + M')$. Fortunately, calls to OUTPUT are strictly in Morton order, allowing the output tree to be constructed with a minimal amount of disk I/O.

The cost of a FIND operation depends on the representation of QU. If QU is implemented as a sorted list of locational codes, then each FIND operation takes $O(\log N_U)$ time. Therefore, a more precise measure of execution time is $O(N_A + N_U \cdot \log N_U + M')$. If QU is implemented as a tree structure (i.e., using pointers, or as a linear quadtree indexed by a B-tree), then the worst-case cost for a FIND operation is $O(n)$ for a $2^n \times 2^n$ image. However, since the node that is being sought is always a neighbor of the current node in the active border array, neighbor-finding methods [16, 21] can be employed for pointer-based quadtrees. These methods have an average cost of $O(1)$. Use of ropes [9] can guarantee an $O(1)$ worst-case cost (however, such an advantage would be countered by the need to create ropes in the output tree). The cost of locating the nodes in QA can be analyzed in a similar manner. However, nodes in QA are processed in Morton order, and it can be assumed that they are also stored in Morton order. Thus the cost of accessing each node in QA is $O(1)$.

3. WINDOWING

Another function commonly provided by GIS allows the user to extract a window from an image. A *window* is simply any rectangular subsection of the image. Typically the window will be smaller than the image, but this is not necessarily the case. The window could also extend beyond the image border. Most importantly, the window's origin (or upper left corner) could potentially be anywhere in relation to the input map's origin. This means that large blocks from the quadtree representation of the image must be broken up, and possibly recombined into new blocks in

the quadtree representation for the windowed image. Details of the resulting block decomposition are discussed in [25, 28].

Shifting an image represented by a quadtree is a special case of the general windowing problem—taking a window equal to or larger than the input image but with a different origin will yield a shifted image. Shifting is important for operations such as finding the quadtree representation of an image that has the fewest nodes. It can also be used to align two images.

If windowing is viewed as a set function on two unaligned maps, an algorithm requiring at most a single call to OUTPUT per output tree node can be derived from Algorithm 2 in the following manner. Let QA correspond to a BLACK region with the same origin as the window and whose size is the least power of 2 greater than or equal to the maximum of the window's width and height. Let QU be the image from which the window is extracted. The resulting image would have QA's size and position, with the value of QU's corresponding pixel at each position that falls within the window (pixels beyond QU's border may be set to WHITE. The equivalence between windowing and unaligned set intersection should be clear. In fact, the windowing algorithm is slightly simpler, since a single BLACK node of the appropriate size takes QA's place in the algorithm, and thus only one call to procedure DOSET is needed. The calculation of the largest block contained within both the current nodes of QA and QU must be modified since the window may be smaller than the full extent of QA. The set function value calculation must be modified to return QU's value; otherwise, the windowing algorithm is identical to INTERSECT.

Algorithms have been discussed in the literature [14] for extracting a window from images represented by pointer-based quadtrees with time complexity $O(N_L + n \cdot M)$ where n is the output tree's depth, N_L is the number of leaves in the input tree, and M is the number of leaves in the output tree. Van Lierop [27] presents an $O(M \cdot (\log_2 N_L + n))$ algorithm for a linear quadtree. Samet *et al.* [19] have presented a windowing algorithm with time complexity $O(N_L + M)$. Algorithms for the translation of 3D objects represented by octrees have also been presented [7, 13, 2, 10]. Unfortunately, for each of these algorithms, M represents the number of nodes in the output tree *before* merging together nodes split by the windowing process (e.g., the block decomposition produced in Fig. 3b if N of Fig. 3a is the desired window). M can therefore be significantly greater than the number of nodes in the output tree once merging has been performed (again in Fig. 3b, the SW and SE quadrants should each merge to form a single WHITE node; M is the count of all pre-merge fragments). When such algorithms are translated to a linear quadtree implementation, this increase in node insertions will greatly affect the algorithm's performance.

Our windowing algorithm locates (only once) those input tree nodes that cover a portion of the window and performs at most one insert operation for each output node. The number of calls to ORDER_INSERT will be M, i.e., the number of nodes before merging. However, ORDER_INSERT handles merging before calling OUTPUT, resulting in great savings in the time required to manipulate the disk-based node list.

Walsh [28] attempts to minimize the work required to merge node fragments by producing the fragments associated with input quadtree node N in sorted order. The fragments of each node are combined with the remainder of the node list by

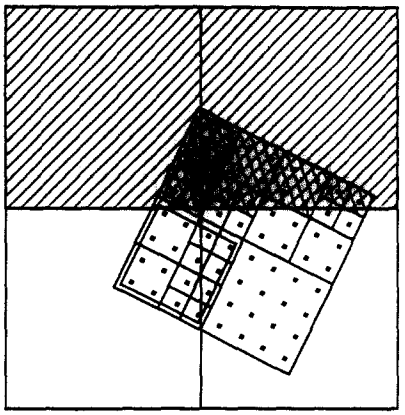

FIG. 5. An example of windowing with rotation showing the block decomposition of the resulting quadtree. Pixel centerpoints in window space are indicated by dots; a pixel P in window space corresponds to the cell in original space containing P's centerpoint.

merging the (sorted) lists. Condensation of the fragments occurs during this process. The major difference between Walsh's algorithm (as well as the others referenced above) and the algorithm presented in this paper is that our algorithm processes the output tree in Morton order, whereas the other algorithms process the input tree in Morton order. Thus, condensation of the fragments is a trivial matter for our algorithm. Instead, we must pay a cost in that nodes of the input tree may be visited more than once. Our active border arrays make this an efficient operation.

With only minor modifications, our set function/windowing algorithms can handle rotated windows (or set operation on two images which are rotated with respect to one another) as in Fig. 5. This is done by transforming each node of the image (or of I2 in the intersection case) to the window's coordinate system (equivalently, the coordinate system of image QA). For improved efficiency, this can be done by means of pre-computed transformation tables. Computation of the largest block contained within the window (or the current nodes of QA) and QU must also be modified. The edge arrays used to store active nodes of the output image require no modification. Figure 5 shows an illustration of the results of the rotated windowing algorithm. Algorithm 3, named WINDOW, presents the rotated windowing algorithm along with code for setting up the transformation tables. The algorithm as presented works for any rotation angle in the range of 0 to 90° (measured as a clockwise angle from the horizontal). Rotation by arbitrary angles would require minor modifications to the transformation tables. Unfortunately, as with any point sampling approach to rotation our algorithm is subject to serious aliasing errors.

4. TEMPLATE MATCH MEASUREMENT FOR QUADTREES

Template matching is an important operation for applications such as character and feature recognition. A simple measure of the "goodness" of the match between two images can be defined as the number of positions where the images' corresponding pixels have equal values. In some cases, the locations or origins of the objects to be matched are known. An example of this is the validation process performed when adding new maps to a database system. Those portions of the new map that overlap

existing maps should be compatible with the old data; if not, then they may need special attention to resolve inconsistencies. In this application, the operation is primarily limited by the efficiency of the matching algorithm. In other applications, the template origin is known, but the object's location in the input image is not. A search for the position producing the best match will be necessary. Even here, some measure for the match at various positions must be computed, and thus, an efficient matching calculation algorithm is desirable.

The matching function, once a relative positioning for the two images has been selected, can be viewed as an unaligned set function. A simple match function will set to BLACK those blocks for which both images have the same value and to WHITE those blocks for which they do not (the value of blocks within the borders of one image, but beyond the borders of the other, will depend on the application). Actually, this algorithm needs only to count the matching pixels, not to create an output image. Thus, a match evaluation function can replace calls to ORDER_INSERT. In the simplest case a counter for the matched pixels is maintained and updated as indicated by the values passed to ORDER_INSERT. Where a node portion of size $2^i \times 2^i$ matches in both images, 2^{2i} is added to the counter. Other match measurements could easily be computed using this framework. One such example would be to add the differences between the corresponding node values (for multi-colored images). As each block is processed, its size would be multiplied by the difference in the corresponding node values and this total added to the counter.

A different approach to image matching can also be performed efficiently using linear quadtrees. This technique involves computing moments for two images and comparing these numeric values rather than comparing the images directly [8, 15]. The (i, j) moment for an image is defined as

$$m_{ij} = \sum_y \sum_x y^i x^j f(y, x),$$

where $f(y, x)$ is the value of the pixel at (y, x). Shneier presents an algorithm for computing the centroid of a pointer-based quadtree that visits each node exactly once during a tree traversal [26]. Other moments for images represented by linear quadtrees can be calculated in a similar manner, requiring one visit to each node during a node list traversal. One advantage of this technique is that the moment for the image with respect to a different origin can be calculated without translating the quadtree. Alternatively, moments can be used to compute a normalized origin, orientation, and scale for an image. Translation and scaling algorithms similar to the windowing procedure described above can then be used to adjust the image. Algorithm 4, named MOMENT, provides a generalized moment calculation function for linear quadtrees.

5. CONCLUSIONS

We have presented efficient algorithms for set operations such as union and intersection, as well as related algorithms for windowing and template matching. In addition, a generalized moment calculation algorithm is presented. These algorithms are optimal in terms of the worst case number of FIND and OUTPUT operations. In particular, our intersection algorithm requires that every node of the first input tree be visited in Morton order, every node of the second input tree also be visited, and the output tree be constructed in Morton order.

TABLE 2
Unaligned Quadtree Intersection Algorithm Statistics

Map	Nodes	(0, 0)		(1, 1)		(100, 100)	
		Time	Nodes	Time	Nodes	Time	Nodes
Floodplain	5206	3.3	4693	9.1	5665	4.2	4279
Landuse	28549	9.0	10243	13.3	10303	8.8	6532
Topography	24859	7.7	5659	12.5	5803	7.8	5527
Pebble	44950	11.6	6217	16.2	6304	11.6	6536
Stone	31969	9.0	4210	13.7	4150	8.8	3862
ACC	3253	3.0	670	8.7	646	3.4	679

Most tree or list implementations for quadtrees should allow the Morton code order traversal and creation operations to be implemented in $O(1)$ time for each node. Tables to support in-core caching of nodes from the second input tree guarantee that each node is read from disk only once. Thus, any quadtree representation that efficiently supports Morton order traversal, Morton order construction, and random access FIND operations would be able to support an efficient implementation of these algorithms.

As an empirical test of the efficiency of our methods, we calculated intersections between several pairs of maps. The test maps were taken primarily from a testbed used in a prototype quadtree-based GIS [24]. Maps are represented as a linear quadtree whose node list is indexed by a B^+-tree. Table 2 shows the sizes of the test maps and the time required to compute the intersections on a VAX 11/785 running BSD 4.3 UNIX, each pair computed for a series of origin offsets. *QU*'s origin is positioned at (0, 0), (1, 1), and (100, 100), respectively. Note that offset (0, 0) is equivalent to a registered intersection (and thus should require the least amount of work), offset (1, 1) is the "worst" possible offset since all nodes larger than a single pixel will be misaligned in the two trees, while offset (100, 100) falls between these extremes. All times are in CPU seconds. In each test, *QU* was the hundred year floodplain region from our testbed. This map contains 4693 leaf nodes when represented as a linear quadtree. The six maps tested against were a slightly modified version of the same floodplain map, a topography map, a landuse survey map, a map derived from the landuse class map containing a single landuse class (named ACC), and two thresholded texture images of pebbles and a stone block, respectively. Their sizes, in terms of the total number of leaf nodes, are indicated in

TABLE 3
Quadtree Rotated Windowing Algorithm Statistics

Map	256 × 256		128 × 128	
	Time	Nodes	Time	Nodes
Floodplain	3.9	1660	2.1	859
Landuse	14.5	9598	3.7	2479
Topography	18.5	12934	3.9	2560
Pebble	18.9	12613	5.0	3547
Stone	13.5	7945	2.7	1267
ACC	2.3	817	0.2	1

the table. For a given map pair and origin positioning, the size of the resulting map measured as the number of leaf nodes is also indicated.

Table 3 shows the timing results for extracting windows rotated 45° for two positions taken from each of the test images. The first column shows the result of calculating a window of size 256 × 256 with origin at (0, 256). The second column shows the result of calculating a window of size 128 × 128 with origin at (0, 128). As with the intersection test, times are measured in CPU seconds and the number of leaf nodes in the resulting maps are indicated.

APPENDIX

The following algorithms make use of a number of primitive operations not further defined in the code. Boolean functions WHITE(*node*) and BLACK(*node*) return TRUE if the value of *node* is WHITE or BLACK, respectively. SON(n, i) returns the ith son of node i. CREATE_POINTER_NODE(*val*) creates a pointer-based quadtree node with value *val*. COPYTREE(*tree*) returns a complete copy of subtree *tree*. Y_OF, and X_OF return the appropriate coordinate for the upper-left corner of a node or map. WIDTH(*node*), DEPTH(*node*), and VALUE(*node*) return the appropriate attribute for *node*. MAX_NODE_DEPTH is a global constant containing the maximum depth allowed for a quadtree. OUTPUT and FIND are the abstract node write and node locate functions discussed in the text.

ALGORITHM 1. Intersection of two aligned images represented by quadtrees.

```
function SIMPLE_INTERSECT(intree1,intree2 : ↑NODE) : ↑NODE;
  {Return the intersection of two pointer-based quadtrees by calculating the node
   value if either intree1 or intree2 is a leaf node; otherwise, recursively calculate
   the intersections of their corresponding sub-trees.}
var n : ↑NODE;
begin
  if WHITE(intree1) or WHITE(intree2) then
    return (CREATE_POINTER_NODE('WHITE'));
  if BLACK(intree1) then {return a copy of intree2}
    return (COPYTREE(intree2));
  if BLACK(intree2) then {return a copy of intree1}
    return (COPYTREE(intree1));
  n := CREATE_POINTER_NODE('GRAY');
  for i in {'SW', 'SE', 'NW', 'NE'} do
    SON(n, i) := SIMPLE_INTERSECT(SON(intree1,i), SON(intree2,i));
  if WHITE(SON(n,'NW')) and WHITE(SON(n,'NE')) and
    WHITE(SON(n,'SW')) and WHITE(SON(n,'SE')) then
    return (CREATE_POINTER_NODE('WHITE')) {merge}
  else return (n);
   end;
```

ALGORITHM 2. Intersection of two unaligned images represented by linear quadtrees.

```
var
  in1tree, in2tree, outtree : QUADTREE; {input and output trees}
  offsety, offsetx : integer; {(offsety, offsetx) is position of in2tree w.r.t. in1tree}
```

```
  cy, cx : integer; {(cy, cx) is current location of traversal in in1tree}
  y_edge : array [0..WIDTH(in2tree)] of NODE; {Initially all zeros}
  x_edge : array [0..WIDTH(in2tree)] of NODE; {Initially all zeros}
  outtab : array [0..3*MAX_NODE_DEPTH - 1] of NODE;
  outcurr : integer; {current position in outtab}

procedure INTERSECT;
  {Calculate the intersection of two (possibly unaligned) input maps, producing an
   output map whose size and position is identical to that of the first map.}
var nd : ↑NODE;

begin
  offsety := Y_OF(in2tree) - Y_OF(in1tree); {Calc. upper left corner of in2tree}
  offsetx := X_OF(in2tree) - X_OF(in1tree); {w.r.t. upper left corner of in1tree}
  cy := 0; cx := 0; {Start at upper left corner of in1tree}
  foreach nd in in1tree do DOSET (nd);
end;

procedure DOSET(in1node : ↑NODE);
  {For in1node, a node of in1tree, break it into pieces matching the nodes of
   in2tree (active in2tree nodes are stored in y_edge and x_edge), compute the
   intersection value on the node pieces, and insert the result into outtree through
   procedure ORDER_INSERT. (stopy, stopx) is position beyond in1node; depth
   is depth of output node fragment; max is largest dimension of matching blocks
   in both trees.}
var
  stopy, stopx, depth, max : integer;
  in2node : ↑NODE; {node found in in2tree}
  val : color; {value of output fragment}
begin
  stopy := cy + WIDTH(in1node); stopx := cx + WIDTH(in1node);
  while (cy < stopy) and (cx < stopx) do begin
    if (Y_OF(x_edge[cx]) + WIDTH(x_edge[cx]) <= cy) and
      (X_OF(y-edge[cy]) + WIDTH(y_edge[cy]) <= cx) then begin
        {Locate new node in the tree. Assume FIND will return an appropriate
         'WHITE' node if (cy, cx) is not contained in in2tree}
        FIND(in2tree, in2node, cy - offsety, cx - offsetx);
        Y_OF(x_edge[cx]) := Y_OF(y_edge[cy]) := Y_OF(in2node) + offsety;
        X_OF(x_edge[cx]) := X_OF(y_edge[cy]) := X_OF(in2node) + offsetx;
        VALUE(x_edge[cx]) := VALUE(y_edge[cy]) := VALUE(in2node);
        WIDTH(x_edge[cx]) + WIDTH(y_edge[cy]) := WIDTH(in2node)
    end
    else if Y_OF(x_edge[cx]) + WIDTH(x_edge[cx]) <= cy then
        x_edge[cx] := y_edge[cy]
    else if X_OF(y_edge[cy]) + WIDTH(y_edge[cy]) <= cy then
        y_edge[cy] := x_edge[cx];
    {Compute biggest block starting at (cy, cx) contained in both input nodes}
    max := min(min(stopy - cy, stopx - cx),
```

```
                min(X_OF(x_edge[cx]) + WIDTH(x_edge[cx]) - cx,
                    Y_OF(x_edge[cx]) + WIDTH(x_edge[cx]) - cy));
      depth := 0;
      while (2^(depth+1) <= max) and (cx mod 2^(depth+1) = 0) and (cy mod 2^(depth+1) = 0)
                do depth := depth + 1;
      val := if VALUE(in1node) = 'WHITE' then 'WHITE'
          else VALUE(x_edge[cx]) {Compute intersection value}
      ORDER_INSERT(val, depth); {Insert result into output tree, update (cy, cx)}
  end {while (cy < stopy)...}
end;

procedure ORDER_INSERT(value : color; depth : integer);
  {Accept blocks in Morton sequence order, merging four siblings of the same
   color. Call OUTPUT only for maximal blocks.}
var i : integer;

begin
  if (outcurr < > 0) and (value < > VALUE(outtab[outcurr - 1])) then begin
     {Different value—flush current nodes in outtab}
     for i := 0 to outcurr - 1 do
        OUTPUT(outtree, Y_OF(outtab[i]), X_OF(outtab[i]),
                DEPTH(outtab[i]), VALUE(outtab[i]));
     outcurr := 0
  end;
  while outcurr > = 3 and depth = DEPTH(outtab[outcurr - 3]) and
    cx - 2^depth = X_OF(outtab[outcurr - 3]) and
    cy - 2^depth = Y_OF(outtab[outcurr - 3]) do begin
        {Collapse 4 siblings with same value}
        depth := depth - 1; outcurr := outcurr - 3;
        cx := cx - 2^depth; cy := cy - 2^depth {adjust upper-left corner}
  end;
  X_OF(outtab[outcurr]) := cx; Y_OF(outtab[outcurr]) := cy;
  DEPTH(outtab[outcurr]) := depth; VALUE(outtab[outcurr]) := value;
  outcurr := outcurr + 1;
  GET_NEXT_MORTON(cx, cy, depth); {compute location of next block}
  if cx = WIDTH(outtree) then {at end of tree; flush outtab}
       for i := 0 to outcurr - 1 do
          OUTPUT(outtree, X_OF(outtab[i]), Y_OF(outtab[i]),
                  DEPTH(outtab[i]), VALUE(outtab[i]))
end;
```

ALGORITHM 3. Rotated windowing algorithm.

```
{Transformation functions}
define DOXTRNS(y, x)   trunc(X_OF(xtrans[x]) + X_OF(ytrans[y]))
define DOYTRNS(y, x)   trunc(Y_OF(xtrans[x]) + Y_OF(ytrans[y]))
define PI              3.14159
```

```
type
  POINT = record
    y, x : real
  end;
  QUADRANT = {NW, NE, SW, SE};

var
  intree, outtree : QUADTREE;
  insize, outsize : integer;
  {Tables xtrans and ytrans translate from coordinates relative to the window into
   coordinates relative to the input map. Table xtrans gives the absolute coordinate
   transform for a point in the window offset by x units from the origin. It takes
   the translation as well as the rotation into account. Table ytrans gives the
   relative coordinate transform for an offset of y units; i.e., it takes only the
   rotation into account (since translation was done for x offset).}
  xtrans : array [0..outsize] of POINT; {Transformation table—x coordinate}
  ytrans : array [0..outsize] of POINT;{Transformation table—y coordinate}
  invrs : array [QUADRANT] of POINT; {Inverse trans. for corners of input map}
  cy, cx : integer; {(cy, cx) is current position in window}
  y_edge : array [0..WIDTH(intree)] of NODE; {Initially all zeros}
  x_edge : array [0..WIDTH(intree)] of NODE; {Initially all zeros}
  outtab : array [0..3*MAX_NODE_DEPTH - 1] of NODE;
  outcurr : integer; {current position in outtab}

procedure WINDOW(rotate : trigtype; fy, fx, wy, wx : integer);
  {Compute the output tree representing the window with origin (with respect to
   intree's origin) of (fy, fx), size wy × wx, and rotation rotate. Rotate should be
   in degrees or radians as required by trig functions.}
var
  thsin, thcos : real;
  di : integer;
begin
  {Initialize global variables and transformation tables}
  insize : WIDTH(intree); outsize := WIDTH(outtree);
  outcurr := 0;
  thsin := sin(rotate); thcos := cos(rotate);
  for di := 0 to outsize do begin
      xtrans[di].x := thcos*(di + 0.5) + fx;
      xtrans[di].y := thsin*(di + 0.5) + fy;
      ytrans[di].x := -thsin*(di + 0.5);
      ytrans[di].y := thcos*(di + 0.5)
  end;
  {invrs holds the inverse transformation for the input map's corner points; used
   by OVERLAP.}
  invrs[NW].x := -thsin*fy - thcos*fx;
  invrs[NW].y := -thcos*fy + thsin*fx;
  invrs[NE].x := -thsin*fy - thcos*(insize - fx);
  invrs[NE].y := -thcos*fy + thsin*(insize - fx);
  invrs[SW].x := -thsin*(insize - fy) - thcos*fx;
```

```
  invrs[SW].y := -thcos*(insize - fy) + thsin*fx;
  invrs[SE].x := -thsin*(insize - fy) - thcos*(insize - fx);
  invrs[SE].y := -thcos*(insize - fy) + thsin*(insize - fx);
  DO_WIND(fy, fx, wy, wx, outsize)
end;

procedure DO_WIND(fy, fx, wy, wx, outsize : integer);
  {For each node of the input file, break it into pieces matching the node positions
   of the output file, storing active nodes in the edge tables. Output fragments are
   passed to ORDER_INSERT. The output fragment is of size 2^maxd; (trcy, trcx) is
   (cy, cx) in rotation space.}
var
  innode : ↑NODE;
  val : color; {output fragment color}
  maxd,trcy,trcx : integer;
begin
  cy := 0; cx := 0;
  while ((cy < > outsize) and (cx < > outsize)) do begin
      {working on part of node}
      maxd := 0;
      if ((cy > = wy) or (cx > = wx)) then begin {outside window}
          color := 'WHITE'; {insert WHITE node}
          while ((cy + 2^(maxd+1) < = outsize) and (cx + 2^(maxd+1) < = outsize) and
                 (cy mod 2^(maxd+1) = 0) and (cx mod 2^(maxd+1)) do
              maxd := maxd + 1
      end {if ((cy > = wx)...}
      else begin {in window}
          while ((cy + 2^(maxd+1) < = wy) and (cx + 2^(maxd+1) < = wx) and
                 (cy mod 2^(maxd+1) = 0) and (cx mod 2^(maxd+1) = 0)) do
              maxd := maxd + 1
          trcy := DOYTRNS(cy, cx); trcx := DOXTRNS(cy, cx);
          if ((trcy < 0) or (trcx < 0) or (trcy > = insize) or (trcx > = insize)) then
          begin
              {off the input map—insert largest possible WHITE node}
              while (OVERLAP((cy, cx, 2^maxd)) do
                  maxd := maxd - 1;
              color := 'WHITE'
          end {if}
          else begin {node is on the input map—calculate node size}
              {first update tables}
              if (PTIN(x_edge[trcx], trcy, trcx) = 'FALSE') then
                  if (PTIN(y_edge[trcy], trcy, trcx) = 'FALSE') then begin
                      {locate new node in the tree}
                      innode := FIND(intree, trcy, trcx);
                      x_edge[trcx] := y_edge[trcy] := innode
                  end
                  else x_edge[trcx] := y_edge[trcy]
```

```
                else if (PTIN(y_edge[trcy], trcy, trcx) = 'FALSE') then
                    y_edge[trcy] := x_edge[trcx];
                {compute biggest block starting at (cy, cx) within input tree node}
                while (CONTAINS(x_edge[trcx], trcy, trcx, 2^maxd) = 'FALSE') do
                    maxd := maxd - 1;
                val := VALUE(x_edge[trcx])
            end {else (node is on the input map)}
                end; {else in window}
                ORDER_INSERT(val, 2^maxd) {Insert result into the output tree}
  end {while ((cy < > outsize)...}
end;

function OVERLAP(cy, cx, csize : integer) : boolean;
  {TRUE iff square (cy, cx, csize) anywhere overlaps the input map. Check by
   determining if node corner is in map, then if map corner is in node. Will work as
   long as node size is at most half of input map size.}
var trcx, trcy : integer;
begin
  csize := csize - 1; {want to compute actual corner points for actual square}
  trcx := DOXTRNS(cy, cx); trcy := DOYTRNS(cy, cx);
  if ((trcy > = 0) and (trcx > = 0) and (trcy < insize) and (trcx < insize)) then
      return ('TRUE');
  trcx := DOXTRNS(cy, cx + csize); trcy := DOYTRNS(cy, cx + csize);
  if ((trcy > = 0) and (trcx > = 0) and (trcy < insize) and (trcx < insize)) then
      return ('TRUE');
  trcx := DOXTRNS(cy + csize, cx); trcy := DOYTRNS(cy + csize, cx);
  if ((trcy > = 0) and (trcx > = 0) and (trcy < insize) and (trcx < insize)) then
      return ('TRUE');
  trcx := DOXTRNS(cy + csize, cx + csize);
  trcy := DOYTRNS(cy + csize, cx + csize);
  if ((trcy > = 0) and (trcx > = 0) and (trcy < insize) and (trcx < insize)) then
      return ('TRUE');
  {No corner of node is in input map, but still might overlap}
  if (invrs[NW].x > = cx and invrs[NW].y > = cy and
     invrs[NW].x < cx + csize and invrs[NW].y < cy + csize) then
         return ('TRUE');
  if (invrs{NE].x > = cx and invrs[NE].y > = cy and
     invrs{NE].x < cx + csize and invrs[NE].y < cy + csize) then
         return ('TRUE');
  if (invrs[SW].x > = cx and invrs[SW].y > = cy and
     invrs[SW].x < cx + csize and invrs[SW].y < cy + csize) then
         return ('TRUE');
  if (invrs[SE].x > = cx and invrs[SE].y > = cy and
     invrs[SE].x < cx + csize and invrs[SE].y < cy + csize) then
         return ('TRUE');
  return ('FALSE')
end;

function CONTAINS(trcy, trcx, csize : integer) : boolean;
```

```
  {'TRUE' iff the square (trcy, trcx, csize) in rotation space is completely con-
   tained in the node represented by p (in input space)}
begin
  csize := csize - 1; {want to compute corner points for this square}
  if (not PTIN(p, trcy, trcx)) then return ('FALSE');
  trcx := DOXTRNS (cy, cx + csize); trcy := DOYTRNS (cy, cx + csize);
  if (not PTIN(p, trcy, trcx)) then return ('FALSE');
  trcx := DOXTRNS (cy + csize, cx); trcy := DOYTRNS (cy + csize, cx);
  if (not PTIN(p, trcy, trcx)) then return ('FALSE');
  trcx := DOXTRNS (cy + csize, cx + csize);
  trcy := DOYTRNS (cy + csize, cx + csize);
  return (PTIN(p, trcy, trcx))
end;

function PTIN(p : NODE; x, y : integer) : boolean;
  {'TRUE' iff point (x, y) is in the node represented by p.}
begin
  return ((x > = X_OF(p) and (y > = Y_OF(p) and
        (x < X_OF(p) + WIDTH(p)) and (Y < Y_OF(p) + WIDTH(p))
end;
```

ALGORITHM 4. Generalized moment calculation.

```
var counter : integer;

function MOMENT(intree : quadtree; i, j, shiftx, shifty : integer) : integer;
  {Calculate moment M(i, j) for quadtree intree which is shifted (shiftx, shifty)
   from the origin.}
var nd : ↑NODE;
begin
  foreach nd in intree do DOMOMENT(nd, i, j, shiftx, shifty);
  return (counter)
end;

procedure DOMOMENT(nd : ↑NODE; i, j, shiftx, shifty : integer);
var val, row, col, ct1, ct2, x, y, width : integer;
begin
  val := COLOR(nd); width := WIDTH(nd);
  x := X_OF(nd) - shiftx; y := Y_OF(nd) - shifty;
  for row := 0 to row = width do begin
      for col := 0 to col = width do ct1 := (col + x)^i;
      ct2 := ct1 * (row + y)^j
  end;
  counter := counter + (ct2 * val)
end;
```

REFERENCES

1. D. J. Abel, A B^+-tree structure for large quadtrees, *Comput. Vision Graphics Image Process.* **27**, No. 1, 1984, 19–31.
2. N. Ahuja and C. Nash, Octree representations of moving objects, *Comput. Vision Graphics Image Process.* **26**, No. 2, 1984, 207–216.

3. M. Callen, I. James, D. C. Mason, and N. Quarmby, A test-bed for experiments on hierarchical data models in integrated geographic information systems, in *Spatial Data Processing Using Tesseral Methods* (B. M. Diaz and S. B. M. Bell, Eds.), Swindon, Great Britain, 1986.
4. D. Comer, The ubiquitous B-tree, *ACM Comput. Surveys* **11**, No. 6, 1979, 121–137.
5. I. Gargantini, An effective way to represent quadtrees, *Commun. ACM* **25**, No. 12, 1982, 905–910.
6. I. Gargantini, Translation, rotation and superposition of linear quadtrees, *Int. J. Man-Mach. Stud.* **18**, No. 3, 1983, 253–263.
7. T.-H. Hong and M. O. Shneier, Rotation and translation of objects represented by octrees, in *Proceedings, 1987 Int. Conference on Robotics and Automation*, Raleigh, NC, 1987, pp. 947–952.
8. M.-K. Hu, Visual pattern recognition by moment invariants, *IRE Trans. Inform. Theory* **8**, No. 2, 1962, 179–187.
9. G. M. Hunter and K. Steiglitz, Operations on images using quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **1**, No. 2, 1979, 145–153.
10. D. Meagher, Geometric modeling using octree encoding, *Comput. Graphics Image Process.* **19**, No. 2, 1982, 129–147.
11. D. M. Mark and J. P. Lauzon, Approaches for quadtree-based geographic information systems at continental or global scales, in *Proceedings, Auto-Carto London*, London, September 1986, pp. 344–364.
12. G. M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, IBM Canada, 1966.
13. W. M. Osse and N. Ahuja, Efficient octree representation of moving objects, in *Proceedings, 7th Int. Conference on Pattern Recognition, Montreal, 1984*, pp. 821–823.
14. F. Peters, An algorithm for transformations of pictures represented by quadtrees, *Comput. Vision Graphics Image Process.* **32**, No. 3, 1985, 397–403.
15. A. P. Reeves and A. Rostampour, Shape analysis of segmented object using moments, in *Proceedings, Pattern Recognition and Image Processing 81*, August 1981, pp. 171–174.
16. H. Samet, Neighbor finding techniques for images represented by quadtrees, *Comput. Graphics Image Process.* **18**, No. 1, 1982, 37–57.
17. H. Samet, The quadtree and related hierarchical data structures, *ACM Comput. Surveys* **16**, No. 2, 1984, 187–260.
18. H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber, A geographic information system using quadtrees, *Pattern Recognit.* **17**, No. 6, 1984, 647–656.
19. H. Samet, A. Rosenfeld, C. A. Shaffer, R. C. Nelson, and Y.-G. Huang, *Application of Hierarchical Data Structures to Geographical Information Systems, Phase III*, Computer Science TR-1457, University of Maryland, College Park, MD, November 1984.
20. H. Samet and M. Tamminen, Computing geometric properties of images represented by linear quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **7**, No. 2, 1985, 229–240.
21. H. Samet and C. A. Shaffer, A model for the analysis of neighbor finding in pointer-based quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **7**, No. 6, 1985, 717–720.
22. C. A. Shaffer, *Application of Alternative Quadtree Representations*, Ph.D. dissertation, Computer Science Department, University of Maryland, College Park, MD, 1986; University of Maryland TR-1672, June 1986.
23. C. A. Shaffer and H. Samet, Optimal quadtree construction algorithms, *Comput. Vision Graphics Image Process.* **37**, No. 3, 1987, 402–419.
24. C. A. Shaffer, H. Samet, and R. C. Nelson, QUILT: A Geographic Information System Based on Quadtrees, *Int. J. Geographic Informations Systems*, to appear; University of Maryland TR 1885, July 1987.
25. C. A. Shaffer, A formula for computing the number of quadtree node fragments created by a shift, *Pattern Recognit. Lett.* **7**, No. 1, 1988, 45–49.
26. M. Shneier, Calculations of geometric properties using quadtrees, *Comput. Graphics Image Process.* **16**, No. 3, 1981, 296–302.
27. M. L. P. van Lierop, Geometrical transformations on pictures represented by leafcodes, *Comput. Vision Graphics Image Process.* **33**, No. 1, 1986, 81–98.
28. T. R. Walsh, Efficient axis-translation of binary digital pictures by blocks in linear quadtree representation, *Comput. Vision Graphics Image Process.* **41**, No. 3, 1988, 282–292.
29. R. E. Webber, *Analysis of Quadtree Algorithms*, Ph.D. dissertation, Computer Science Department, University of Maryland, College Park MD, 1983; University of Maryland TR-1376, March 1984.