An Optimal Boundary to Quadtree Conversion Algorithm

Mark Lattanzi and Clifford A. Shaffer

TR 89-16

AN OPTIMAL BOUNDARY TO QUADTREE CONVERSION ALGORITHM

Mark Lattanzi
Clifford A. Shaffer

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

ABSTRACT

An algorithm is presented for converting a boundary representation for an image to its region quadtree representation. Our algorithm is designed for operation on the *linear* quadtree representation, although it can easily be modified for the traditional *pointer-based* quadtree representation. The algorithm is a two phase process that first creates linear quadtree node records for each of the border pixels. This list of pixels is then sorted by locational code. The second processing phase fills in the nodes interior to the polygons by simulating a traversal of the corresponding pointer-based quadtree. Three previous algorithms [Same80, Mark85a, Atki86] have described similar conversion routines requiring time complexity of $O(n \cdot B)$ for at least one of the two phases, where B is the number of boundary pixels and n is the depth of the final tree for a $2^n \times 2^n$ image. A fourth algorithm [Webb84] can perform the border construction of this conversion in time $O(n + B)$ with the restriction that the polygon must be positioned at constrained locations in the image space. Our algorithm requires time $O(n + B)$ for the second phase, which is optimal. The first phase can be performed using the algorithm of [Webb84] for total conversion time of $O(n + B)$ with constrained location, or in time $O(B \log B)$ using a simple sort to order the border pixels with no restriction in polygon location.

Keywords and phrases: quadtrees, linear quadtrees, hierarchical data structures, chaincodes, polygon filling, traversal, image processing.

May 10, 1989

1. INTRODUCTION

The region quadtree [Same89a, Same89b] is a hierarchical data structure used for efficient representation of raster images. The region quadtree divides a $2^n \times 2^n$ image array into four equal quadrants if all pixels within that array are not the same color. The image is further subdivided into subquadrants, sub-subquadrants, ... until each block is homogeneous (see Figure 1). The resulting block decomposition may then be represented either as a tree structure describing the decomposition process, or as a list of block descriptions. Recent advances in the use of quadtrees for computer cartography and computer graphics have made efficient algorithms for conversion between the region quadtree and other image representations more important than ever. This paper describes a chaincode to quadtree conversion algorithm for linear quadtrees [Garg82], an implementation variant of the quadtree. Converting a boundary representation to a region representation requires *polygon filling*. Figures 2 and 3 provide an example of the conversion process.

Four previous algorithms for chaincode to quadtree conversion have been presented (related work may also be found in [Hunt79a]). The first [Same80] converts a chaincode to a *pointer-based* quadtree representation (i.e., a quadtree representation in which an explicit tree structure representing the block decomposition is stored, complete with pointers linking a parent node to its children as shown in Figure 1d). This algorithm works by first building a quadtree in which each border pixel derived from the chaincode is inserted as a BLACK node. The tree is constructed by progressing along the chain of boundary pixels, using neighbor finding operations as the chaincode is processed. A second phase then traverses this quadtree in order to fill the polygon bordered by the chaincode. The order of complexity for this algorithm is $O(n \cdot B)$ where B is the number of border pixels on the chaincode and n is the depth the tree for a $2^n \times 2^n image$

The second algorithm [Garg84, Atki86] converts a chaincode to a *linear* quadtree. The linear quadtree [Garg82] replaces the explicit pointer-based tree structure with a list containing only the leaf nodes from the original tree. These leaf nodes correspond to the blocks illustrated

by the image decomposition in Figure 1c. Traditionally, this list is sorted using a key derived by interleaving the bits of the row and column coordinates of the upper left pixel for the corresponding block in the image. This bit interleaved address will be referred to as the *Morton code* for the node. The resulting records (when sorted by Morton code) appear in the list in the same order as the corresponding blocks would be visited by a traversal of the pointer-based quadtree for the same image. The linear quadtree as used in [Garg82, Atki86] stores only the BLACK leaf nodes (i.e., those nodes representing blocks within the regions of interest of a binary image). We prefer to define the linear quadtree such that the node list contains all leaf nodes from the image. Note that the algorithm presented here can easily be implemented using either definition.

The algorithm of [Atki86] begins with a list of linear quadtree records corresponding to the border pixels derived from the chaincode. This list of nodes is not sorted by Morton code. Each border pixel contains information about which edges of the pixel are adjacent to WHITE pixels (i.e., those edges of the border pixels that are adjacent to the chaincode border). The initial node list is processed by a recursive procedure that splits the list into four bins depending on which quadrant of the image each node is in. The procedure is then re-invoked for each of the resulting sub-lists, which are in turn subdivided based on which subquadrant of the image the corresponding pixel appears in. After n subdivisions for a $2^n \times 2^n$ image, each sublist now contains at most four nodes, corresponding to siblings within a 2×2 sub-block of the image. This process can be viewed as a recursive radix sort on the border node list. The values of these sibling nodes and their border states determine the values of any missing siblings, and also determine if the four siblings may be merged to form a single node. The resulting node sublist is then passed back as the recursion unwinds, allowing larger and larger sized missing nodes to be added and, if possible, allowing larger siblings to be merged. The final result is an unsorted list of nodes corresponding to the linear quadtree for the image with the interior nodes of each polygon set to BLACK. The order of complexity for this algorithm is $O(n \cdot B)$ since each border pixel participates in n constant

time operations. A final sorting phase is required to generate a node list ordered by Morton code.

The third algorithm [Mark85a, Mark85b] parallels Samet's algorithm [Same80] except it allows for more generalized input. Instead of a list of border pixels, Mark and Abel's algorithm works for any vector representation of a polygon. Thus, the first step of their algorithm is to compute the list of pixels lying on the boundary of the polygon of interest. This step takes $O(B)$ time. However, the input to step two of the algorithm, i.e., the polygon fill phase, is a sorted list of pixels by Morton code. So, the pixel list must be sorted before being passed to step two. Phase two determines the colors of all the absent pixels (nodes). The main theorem underlying this step of the algorithm states that given a sorted list of border pixels, the absent nodes between any two border pixels must be either all the same color, or change colors only one time. In this second case, the intermediary nodes will be in two contiguous sets. The first set will be one color, and the second set will be the other color. The importance of this theorem is that a limited amount of information is needed to determine the colors of all the absent nodes. Mark and Abel show that at most one ancestor find and between two and four neighbor finds are performed per node, an $O(n)$ operation. A detailed analysis of this process was presented by Mark and Abel in an earlier paper [Mark85c]. As with the algorithm of [Same80], the overall time complexity of this algorithm is dominated by their polygon fill operation: $O(n \cdot B)$ where B is the number of border pixels and n is the depth of the final tree.

Webber [Webb84] improved the conversion algorithm of Samet [Same80] by using a method called *path length balancing*. The main idea is to treat the boundary representation as two one-dimensional objects: a horizontal one and a vertical one, and strategically position the polygon in the quadtree to minimize the total cost of the neighbor find operations. Thus, the border pixels can be inserted into the quadtree, and the interior of the region can be filled in time proportional to the number of nodes in the quadtree (equivalently, the number of pixels obtained from the chaincode). Although Webber's result is linear, it is constrained by the fact that the region representation of

the polygon must be placed at a certain optimal position within the tree to achieve a linear time complexity. Sometimes this is not desirable, in which case path length balancing cannot be used. Webber [Webb84] does not address the polygon fill phase of the algorithm.

2. A NEW CONVERSION ALGORITHM

Our new algorithm converts a boundary representation to a linear quadtree in Morton code order. The boundary representation may be either a chaincode or a vector representation of the polygon's perimeter, but it must be convertible to a list of the boundary pixels. We assume for simplicity of exposition that all images are binary, although our algorithm would work equally well for multicolor images. Nodes interior to the polygon boundary are labeled BLACK after the conversion process is complete; nodes exterior to the polygon are labeled WHITE.

Our algorithm works in the spirit of Mark's algorithm [Mark85a] for producing a pointer-based quadtree. Both algorithms take as input a boundary representation of a polygon. Both algorithms operate in two phases. The first phase requires a traversal of the boundary to produce a list of the border pixels. This list is then sorted by Morton code.

The polygon filling phase of our algorithm is similar to Mark's second phase, but this portion of our algorithm is performed in time $O(n + B)$ instead of $O(n \cdot B)$. This linear time bound is achieved by using a simple table to record the expected values of neighboring nodes yet to be visited during the simulated traversal of the tree. Since a tree traversal visits every node one time, and we do constant processing at each node, our polygon filling step is $O(N)$ where N is the number of nodes in the tree. By a theorem presented in [Hunt79a], the number of nodes in the resulting quadtree is directly proportional to the number of border pixels for the polygon plus the resolution of the image, so this step takes $O(n+B)$ time overall. The total cost of our algorithm will therefore be $O(B \log B)$, dominated by the cost of the initial sorting step. Alternatively, we could use the initial construction phase of [Webb84] yielding an overall time complexity of $O(n + B)$ with the

constrained positioning of the polygon within the tree.

2.1. Detailed Description of our Conversion Algorithm

As with the algorithm of [Mark85a], the first step in our chaincode to quadtree conversion process is to generate a sorted list of linear quadtree node records representing the border pixels. Border pixels are defined to be those pixels with at least one edge on the chaincode boundary. The initial border pixel list is created by following the directional codes of the chaincode from pixel to pixel. The second (filling) step of our algorithm takes as input a set of sorted border pixels. The border pixels may be four or eight connected. Polygon boundary descriptions are restricted as follows.

> 1) Each polygon represented must be closed (i.e., it must begin and end on the same pixel).
>
> 2) Polygons may not be self-intersecting, nor may two polygons in the image intersect.

As the chaincode is processed, a linear quadtree record for each border pixel is created that contains the Morton code, the pixel's x and y coordinates, and a border code telling which sides of the pixel are adjacent to WHITE pixels (i.e., which borders of the pixel actually touch the chaincode boundary). The Morton code duplicates the x and y coordinates for the pixel; however, presentation of the algorithm is simplified if all operations are done in terms of x and y coordinates. Final processing of the node list can remove this redundant information if desired. After all border pixels have been generated, they must be sorted by Morton code, and passed to the filling phase of the algorithm.

The second phase processes the sorted list, simulating a traversal on the corresponding pointer-based quadtree. The purpose of this traversal is to fill in the interior of the polygons whose borders are described by the boundary pixel list. To aid filling of the polygons during this traversal, a table is maintained that indicates the expected color of adjacent nodes yet to be processed. This color table is updated as the node list is processed to reflect the current state of the traversal.

The polygon filling algorithm begins by finding the first *chunk* in the image. A chunk is defined to be all pixels whose Morton codes lie between two successive border pixels (that is, successive in terms of Morton code). Each chunk in the image must eventually be broken into quadtree nodes. A chunk's constituent nodes are determined during the traversal phase and, based on the color of each such node, our table is updated. As nodes are created during the traversal, they are passed to a routine that compares the node with its siblings to determine if all four siblings are the same color and thus may be merged into a single node. This routine simply keeps a list of consecutive nodes of the same color. When a node of a second color is processed, it may not merge with any nodes preceding it; thus, stored nodes may be output. Four consecutive siblings of the same color are merged to form a single node. At most $3n$ nodes need to be stored at one time. For further details on the output node process, see [Shaf86]. The traversal continues until all of the chunks have been processed and the resulting nodes output.

If the first border pixel of the image is not at position (0, 0), then maximal WHITE nodes are output until the first border pixel is reached. Similarly, after the last border pixel has been processed, WHITE nodes are output until the lower right corner of the image has been reached.*

2.2. The Simulated Traversal

This section describes in detail the table driven simulated traversal that gives the filling phase linear time complexity. When the initial border pixel list is created, a border code is associated with each pixel (similar to the border code used in [Same80, Atki86]). This border code describes which of the pixel's edges may be adjacent to WHITE pixels. The code can be viewed as a 4 bit map with bits 0, 1, 2, and 3 corresponding to directions N, E, S, and W, respectively. Alternatively, the code can be viewed as a four bit value generated by summing values of N = 1, E = 2, S = 4, and W = 8 for each edge on a border. For example, a pixel in the SE corner of a

* In this paper, all quadtrees are assumed to be traversed in the order NW, NE, SW, SE; the origin of each image is assumed to be (0, 0) appearing at its upper left corner; and pixel coordinates are referred to by (ROW, COL).

region has a border code of 2+4 = 6. Note that the color for all border pixels is BLACK, since by definition border pixels are within the polygon defined by the chaincode.

After sorting the border pixel list into Morton code order, the complete linear quadtree is generated by processing the pixel list in Morton order while simulating a traversal of the corresponding pointer-based quadtree. To aid this traversal, information is stored in the *color table.* This table is a two dimensional array with n rows for a $2^n \times 2^n$ image, i.e., one row for each level of the corresponding tree (except for the root at level n). Each row contains four columns, representing quadrants NW, NE, SW, and SE. Thus, each entry in the table corresponds to a quadrant at a particular level of the tree. The value stored at each entry is the expected color for the next node of that level and quadrant in the tree. Node levels are labeled from 0 (corresponding to nodes of size $2^0 \times 2^0$, i.e., pixels) to $n-1$ (corresponding to nodes of size $2^{n-1} \times 2^{n-1}$, i.e., children of the root node). Before processing begins, the color table is initialized so that all entries contain the value WHITE.

The traversal of the node list begins by setting the current level to be n, corresponding to visiting the root of the corresponding pointer-based tree. The initial position in the traversal is (0, 0), which is the upper left corner of the image (as well as the root node). At each stage of the traversal, we maintain the current level in the tree, the current (x, y) position in the tree, and three values that describe the three corner pixels of the current node (all but the SE corner). Each corner pixel in the current node may also be the corner pixel of some larger node. The *maximum corner value* for each corner is the level of the largest possible node for which that pixel is the corresponding corner. For example, pixel (0, 0) is the NW corner of a node corresponding to the entire image, while pixel (4, 0) is the NW corner of a 4×4 pixel node, regardless of the actual level for the current node. We maintain this corner information in order to allow updates to the color table in constant time. During the traversal process, corner information for the current node can be derived in constant time from the corner information associated with the pixel's parent.

Processing of the pixel list is simply a matter of processing the collection of pixels that lay between each pair of border pixels (i.e., a chunk). As the nodes within a chunk are generated and output, the color table is updated for each node. The first node in a chunk is the border pixel itself, since this border pixel (which must be the NW corner of the chunk) may have a chaincode border along its E or S edge. When appropriate, this border pixel will later be merged with its siblings. The current pixel position is then incremented to the next pixel in Morton order. The next node in the chunk is the largest quadtree block with the current pixel position as its NW corner that does not include the next border pixel. After this node is output, the next pixel position is calculated in turn – this will be the next pixel in Morton order following the node just processed. When the location of the current pixel becomes the same as the location of the next border pixel on the border node list, then the current chunk is complete and processing of the next chunk may begin. This continues until the entire list of border pixels has been processed. The purpose of the simulated tree traversal is to allow the calculation in linear time of the nodes making up the current chunk. The final chunk after the last border pixel contains the remainder of the image. All the nodes in the last chunk are WHITE.

Procedure TRAVERSE of Section 2.3 is the heart of our algorithm. It simulates a pointer-based quadtree traversal while simultaneously traversing the list of border pixels. If the node at the current level does not fit into the current chunk (determined by function CHECK_NODE_SIZE), then the algorithm recursively processes the node's four children. For each of these children, the NW corner's coordinates and the maximum corner values are computed. A maximum corner value is the level of the largest node with that pixel in the corresponding corner. The largest eastern neighbor for a node N will therefore be at the same level as N's maximum NE corner value. Similarly, the largest southern neighbor for N is its maximum SW corner value. If the block corresponding to the current node in the quadtree traversal fits within the current chunk, then the color of the current node is determined and the node is passed to the node merging routine ORDER_INSERT.

(Further details on ORDER_INSERT may be found in [Shaf86, Shaf89].) If the current node is a border pixel, then its color is BLACK. If the current node is not a border pixel, then the color of the current node is found in the color table. The entry examined in the color table is at the level and quadrant of the maximum corner value for the NW pixel of the current node. After the node's color is determined, this color may then be used to update the color table.

If the current node is a NW, NE, or SW child of its parent, then the color table is updated. For SE children, no update is required since other nodes will later update those same color table entries. The color table is updated as follows. For the current node, the sizes for the largest adjacent eastern neighbor and the largest adjacent southern neighbor are calculated. They are obtained from the maximum corner values for the current node. If the current node is a border pixel, the pixel's border code determines what color to use when updating the table. For example, if the pixel has stored a border along the eastern edge, then the largest quadrant directly east of the current pixel is set to be WHITE. This is accomplished by setting the entry in the color table for the largest eastern neighbor (level and quadrant). If the current node is not a border pixel, then the neighbor receives the same value as the current node. The quadrant of the neighbor is determined by a simple function based on the current node's location and the level passed down from the current node's parent in the variable *maxne*. If no eastern border exists, the color table entry is set to BLACK. The southern direction is processed in a similar way, using the value *maxsw*.

When the table is used to determine a node's color, the level of the largest node that has a common northwest corner pixel with the current node is used for the table lookup. This insures that the correct color will still be in the table, since no nodes will have been processed that could have changed this value in the table due to the order in which nodes are processed. Figure 4 illustrates the node processing scheme.

2.3. Conversion Algorithm Pseudocode

This section contains a Pascal-like pseudocode description for the filling phase of our conversion algorithm. This pseudocode contains four procedures. START_TRAVERSE initializes the table values and begins the simulated pointer-based quadtree traversal by calling TRAVERSE, which performs the actual traversal. The last two procedures are CHECK_NODE_SIZE, a function that determines if the current node will fit into the current chunk, and UPDATE_TABLE, a procedure that updates the color table based on the level and quadrant passed to it.

The pseudocode below calls several primitive procedures that did not merit a formal description. BIT_SET(*border*, *side*) returns TRUE if the bit corresponding to *side* is set in *border*, and FALSE otherwise. The function FINDQUAD(x, y, *level*) returns the quadrant of the node containing the point (x, y) at *level* with respect to the node's parent. ORDER_INSERT(*level*, *color*, x, y) outputs a *color* node of size $2^{level} \times 2^{level}$ whose NW corner pixel is (x, y). This procedure merges siblings if they are all the same color. GET_NEXT and TEST_NEXT both deal with the input list of border pixels. GET_NEXT(x, y, *border*) takes the next border pixel off the head of the border pixel list and stores its coordinate and border information in x, y, and *border*. TEST_NEXT(x, y) checks if the next pixel on the list has the coordinates x and y. GET_X() and GET_Y() return the x and y coordinates of the next border pixel, respectively.

```
  { Declarations of types and global variables. }
type
  QUADTYPE = (NW, NE, SW, SE);
  BCODE = integer; { border code type: N = 1, E = 2, S = 3, W = 4. }
  NODE = record { record type for border pixel list }
    x, y : integer;
    border : BCODE { border code for each border pixel }
  end;
var
  colortable : array[0..MAXLEVEL, NW..SE] of integer;
  border : BCODE; { border code of node currently being processed }
  currlvl : integer; { level of node currently being processed }
```

```
  currquad : QUADTYPE; { quadrant of node currently being processed }

procedure UPDATE_TABLE(maxnw, maxne, maxsw, x, y : integer);
  { Updates the color table based on the current node just outputted. }
var
  width : integer; { Width of the current node. }
  eastquad, southquad : QUADTYPE; { Largest quadrant to E or S of current one. }
begin
  width := 2^currlvl;
  case currquad of
    NW: begin
      eastquad := NE; southquad := SW;
      if BIT_SET(border, E) then { If eastern border exists }
        colortable[currlvl][eastquad] := WHITE
      else colortable[currlvl][eastquad] := color;
      if BIT_SET(border, S) then { If southern border exists }
        colortable[currlvl][southquad] := WHITE
      else colortable[currlvl][southquad] := color
    end
    NE: begin
      eastquad := FINDQUAD(x + width, y, maxne);
      southquad := SE;
      if BIT_SET(border, E) then { If eastern border exists }
        colortable[maxne][eastquad] := WHITE
      else colortable[maxne][eastquad] := color;
      if BIT_SET(border, S) then { If southern border exists }
        colortable[currlvl][southquad] := WHITE
      else colortable[currlvl][southquad] := color
    end
    SW: begin { Eastern neighbor already set by NE sibling }
      southquad := FINDQUAD(x, y + width, maxsw);
      if BIT_SET(border, S) then { If southern border exists }
        colortable[maxsw][southquad] := WHITE
      else colortable[maxsw][southquad] := color
    end
    SE: { Do nothing, neighbors will be set later on }
  end{ case currquad of ... }
end; { UPDATE_TABLE }

function CHECK_NODE_SIZE(x, y : integer) : boolean;
  { Returns whether or not the quadrant of size (2^currlvl x 2^currlvl) whose NW corner is (x, y) is
    contained within the current chunk. }
var size : integer;
begin
  if (border <> 0) and (currlvl <> 0) then { If current node is a border pixel }
    return (FALSE);
  size := 2^currlvl - 1;
  if (x + size >= GET_X()) and (y + size >= GET_Y()) then
    return (FALSE)
```

```
    else return (TRUE)
  end; { CHECK_NODE_SIZE }

  procedure TRAVERSE(maxnw, maxne, maxsw, x, y : integer);
    { This recursive procedure imitates a pointer-based tree traversal in order to fill in the polygon's
      area. Its parameters are the levels of largest quadrants that the current node is a northwest,
      northeast, and southwest corner of. Also passed along are the current x, y coordinates. }
  var color : integer;
  begin
    if CHECK_NODE_SIZE(x, y) then
      begin { If current node fits into the current chunk, output current node }
        if border = 0 then { if current node is NOT a border pixel }
          color := colortable[maxnw][FINDQUAD(x, y, maxnw)];
        UPDATE_TABLE(maxnw, maxne, maxsw, x, y)
        ORDER_INSERT(currlvl, color, x, y);
        border := 0 { Other nodes in chunk have no border }
      end{ then clause }
    else { Recurse down a level into four children }
      begin
        currlvl := currlvl - 1;
        for currquad in NW, NE, SW, SE do
          begin
            case currquad of
              NW: begin
                TRAVERSE(maxnw, currlvl, currlvl, x, y);
                x := x + 2^currlvl
              end{ NW }
              NE: begin
                TRAVERSE(currlvl, maxne, currlvl, x, y);
                x := x - 2^currlvl;    y := y + 2^currlvl
              end{ NE }
              SW: begin
                TRAVERSE(currlvl, currlvl, maxsw, x, y);
                x := x + 2^currlvl
              end{ SW }
              SE: TRAVERSE(currlvl, currlvl, currlvl, x, y)
            end; { end case }
            if TEST_NEXT(x, y) then GET_NEXT(x, y, border)
          end; { End for loop }
        currlvl := currlvl + 1
      end{ End recursive case }
  end;

  procedure START_TRAVERSE;
  begin
    currlvl := MAXLEVEL;
    if TEST_NEXT(x, y) then
      begin
        x := 0;    y := 0;    border := 0
```

```
    end
  else
    GET_NEXT_BORDER_PIXEL(x, y, border);
  TRAVERSE(MAXLEVEL, MAXLEVEL, MAXLEVEL, x, y)
end;
```

2.4. Example of the Conversion Algorithm

Figure 2a shows an example polygon with all border pixels shaded and their borders shown with a heavy outline. Figure 2b numbers each border pixel by order of its Morton code. Figure 2b also labels with letters those other blocks of the image that will be generated by the traversal phase. Note that nodes A and B are not part of the border list; however, they are interior to the polygon. Also note that blocks 1, 2, 3, and A must eventually be merged to form a single quadtree node. Each border pixel has a border code associated with it. This code indicates which sides of the border pixel are adjacent to WHITE pixels.

Recall that a chunk is the collection of pixels whose Morton codes lie between the Morton codes of two consecutive border pixels. For example, the chunk defined by border pixels 3 and 4 of Figure 2b has two nodes: the first is the level 0 (pixel-sized) node containing pixel 3. The second is the node labeled A. When pixel 4 is reached, a new border pixel (pixel 5) is read off the list, and a new chunk is calculated. Note that every chunk begins with a level 0 node containing the border pixel at the beginning of that chunk. The final chunk defined by pixel 9 at the end of the image contains nodes 9, E, F, G, H, I, and J.

During processing, each node has three corner values associated with it, indicating the maximum level for the node containing the corresponding corner pixel. For node 1, the maximum corner values corresponding to the (NW, NE, SW) corners are (3, 0, 0). Pixel B has values (0, 0, 1). Pixel 5 has values (0, 2, 0).

The traversal process begins with the color table initially containing the value WHITE for

all entries. Figure 3 follows the color table as it is updated by the various nodes. Pixel 1 is the first node used to update the color table. After pixel 1 is processed, the color table is modified as shown in Figure 3a (the initialized WHITE values are not shown for clarity). Entries updated by pixel 1 are underlined. The largest eastern and southern neighbors of pixel 1 are at level 0, so corresponding entries at level 0 in the color table are set to BLACK. After pixel 2 is processed, the color table is as shown in Figure 3b. Since the largest eastern neighbor of pixel 2 is a 2×2 quadrant that is a NE child of its parent, row 1, column NE in the color table is set to BLACK. Figure 3c shows the color table after processing pixel 3. Since node A is not a border pixel, the color table is accessed to determine node A's color. Node A is at level 0, and is a SE child. Thus, its value as indicated by the color table is BLACK. Node A does not update the table as it is a SE child. Figures 3e and 3f show the color table after processing pixels 4 and 5, respectively. Since pixel 5 has an eastern border and its maximum NE corner value is 2, the table entry (level = 2, quad=NE) is set to WHITE. Pixel B is a SW child, so it updates the table for its largest southern neighbor, of size 2×2. This entry in the table is set to BLACK, as shown in Figure 3f. Pixel 6 is a SE quadrant so no update is performed. This process continues until the whole image has been processed or equivalently, the entire tree has been traversed. Figure 2c shows the final block decomposition for the image after all processing and merging has been performed.

3. ANALYSIS

As mentioned previously, [Same80], [Mark85a], and [Atki86] present conversion algorithms that are $O(n \cdot B)$. Samet's algorithm individually inserts B border pixels into a quadtree at level n, while Atkinson *et al*'s algorithm performs n passes through a node list of length B. Since Mark and Abel perform neighbor find operations $O(n)$ each at all $O(B)$ nodes of the quadtree, their algorithm is also $O(n \cdot B)$. Thus, all three of these earlier works have a depth factor (n) in their time complexity. The algorithm of [Webb84] is $O(n + B)$, but it imposes a restriction by limiting

the placement of the image within the image plane (the quadtree).

Our algorithm consists of two phases. First, a chaincode (or any other boundary representation) is processed, generating a list of linear quadtree nodes corresponding to the border pixels for the polygons. For a list of B border pixels, the time for this step is $O(B)$. The list is then sorted by Morton code, requiring $O(B \log B)$ time. The output of the first phase is a sorted list of the border pixel nodes. $O(B \log B)$ is asymptotically the same as $O(n \cdot B)$. However, our first step has been reduced to a simple sort while some of the other algorithms [Same80, Webb84, Atki86] have a more complex initial construction phase.

The most significant aspect of our new algorithm is that our filling phase can be performed in time $O(n+B)$. By Hunter's theorem [Hunt79a], the number of nodes in the quadtree is $O(n+B)$, since B is the length of the perimeter for the polygons represented. Our traversal process (named TRAVERSE in the pseudocode) simulates a traversal of the corresponding pointer-based quadtree, which contains $O(n+B)$ nodes with each node processed once. Each node (i.e., each part of a chunk) is processed in constant time, since all subroutines other than TRAVERSE operate in constant time. Thus, the total time complexity of our filling phase is $O(n + B)$. Very little additional space is required (our color table has $4n$ entries), and no ropes for the tree are required as in [Hunt79a, Hunt79b].

4. CONCLUSIONS

This algorithm accomplishes three goals. First, it presents a different and potentially useful approach for converting border codes to a region quadtree. Second, it is the first boundary to region conversion algorithm (for linear or pointer-based quadtrees) that has unrestricted $O(n + B)$ time complexity for sorted data (border pixels) and $O(B \log B)$ for lists of unsorted border pixels. This is preferable to $O(n \cdot B)$ for small values of B, i.e., short chaincodes. Three previous algorithms [Same80, Mark85a, Atki86] have worst case time complexity of $O(n \cdot B)$ for an image resolution of

$2^n \times 2^n$. Webber's algorithm [Webb84] has a linear time complexity for the border pixel sort phase, but it restricts the placement of the polygon in image space. Our algorithm can be combined with Webber's for a total time requirement of $O(n+B)$. Third, ours is the first linear quadtree algorithm that we are aware of that utilizes the algorithmic technique of simulating a pointer-based quadtree traversal during processing of the linear quadtree node list. This traversal technique is what allows our fill algorithm to operate in linear time. While we have engaged in discussions with researchers on the theoretical implications of such an approach, it is the first actual presentation of such an implementation. As the quadtree representation is utilized for more applications, a wide variety of algorithm techniques will prove increasingly useful to researchers and practitioners.

This algorithm could be easily modified to generate a pointer-based quadtree in $O(B \log B)$ time. However, the initial quadtree construction phase must not operate by inserting the border nodes into the quadtree as done in [Same80]. Instead, we would retain our initial phase in which we sort the border pixels by Morton code, and then modify our traversal phase to actually construct the pointer-based quadtree rather than simulate its traversal. Our algorithm can also be easily modified to operate in three dimensions. The color table must be expanded so as to store eight octants for each level, and the border codes require six bits (one for each face of the voxel). The table is still quite small, and the operation of the algorithm is essentially unchanged.

5. REFERENCES

1. [Atki86] H.H. Atkinson, I. Gargantini, and T.R.S. Walsh, Filling by quadrants or octants, *Computer Vision, Graphics, and Image Processing 33*, 2(February 1986), 138-155.

2. [Garg82] I. Gargantini, An effective way to represent quadtrees, *Communications of the ACM 25*, 12(December 1982), 905-910.

3. [Garg84] I. Gargantini and H.H. Atkinson, Linear quadtrees: a blocking technique for contour filling, *Pattern Recognition 17*, 3(May 1984), 285-293.

4. [Hunt79a] G. Hunter and K. Steiglitz, Operations on images using quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 2(April 1979), 145-153.

5. [Hunt79b] G. Hunter and K. Steiglitz, Linear transformation of pictures represented by quadtrees, *Computer Graphics and Image Processing 10*, (July 1979), 289-296.

6. [Mark85a] D. Mark and D. Abel, Linear quadtrees from vector representations of polygons, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 3(May 1985), 344-349.

7. [Mark85b] D. Mark and J.P. Lauzon, Linear quadtrees for geographic information systems, *Proc. Int. Symp. Spatial Data Handling*, Zurich, Switzerland, August 1984, 412-430.

8. [Mark85c] D. Mark and D. Abel, Linear quadtrees from vector representations: Polygon to quadtree conversion, CSIRONET Tech. Rep. No. 18, Canberra, Australia.

9. [Same80] H. Samet, Region representation: quadtrees from boundary codes, *Communications of the ACM 23*, 3(March 1980), 163-170.

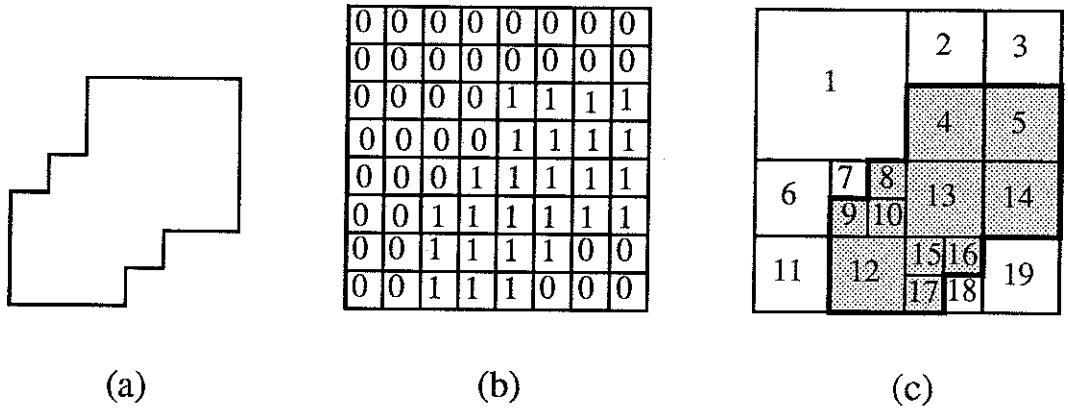
10. [Same89a] H. Samet, *Design and Analysis of Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*, Addison-Wesley, Reading, MA, 1989.

11. [Same89b] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA, 1989.

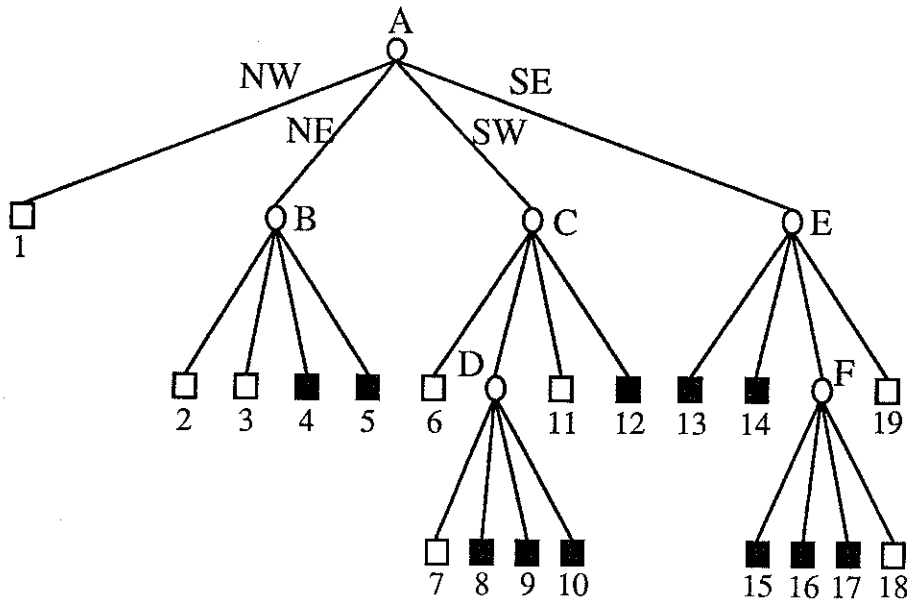
12. [Shaf86] C.A. Shaffer, *Application of Alternative Quadtree Representations*, Ph.D. dissertation, TR-1672, Computer Science Department, University of Maryland, College Park, MD, June 1986.

13. [Shaf89] C.A. Shaffer and H. Samet, Set operations for unaligned linear quadtrees, to appear in *Computer Vision, Graphics, and Image Processing*. Also, Department of Computer Science TR 88-31, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, September 1988.

14. [Webb84] R.E. Webber, *Analysis of Quadtree Algorithms*, Ph.D. dissertation, TR-1376, Computer Science Department, University of Maryland, College Park, MD, March 1984.

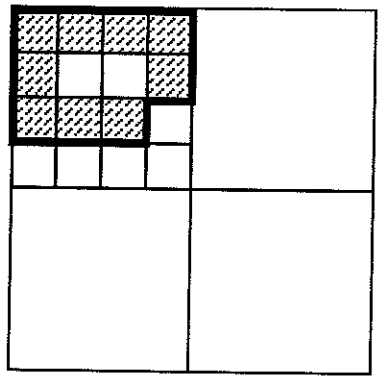


(a) (b) (c)

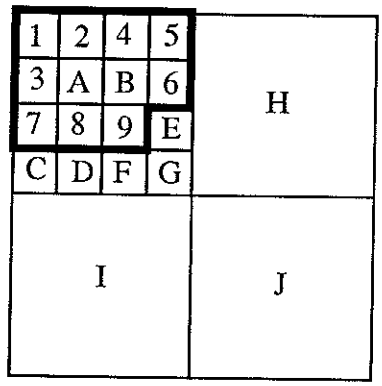


(d)

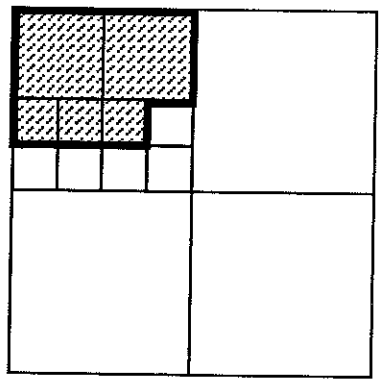
Figure 1. A region, its binary array, its maximal blocks, and the corresponding quadtree. (a) Region. (b) Binary array. (c) Block decomposition of the region in (a). Blocks in the region are shaded. (d) Quadtree representation of the blocks in (c).



(a)



(b)



(c)

Figure 2. Chaincode to Quadtree conversion process. (a) Initial list of border pixels and their boundary. (b) Boundary of region with border pixels labeled 1-9 and other nodes lettered. (c) Block decomposition of the final region quadtree.

1	NW	NE	SW	SE
0		B	B	
1				
2				

(a)

2	NW	NE	SW	SE
0		B	B	B
1		B		
2				

(b)

3	NW	NE	SW	SE
0		B	B	B
1		B	B	
2				

(c)

4	NW	NE	SW	SE
0		B	B	B
1		B	B	
2				

(d)

5	NW	NE	SW	SE
0		B	B	B
1		B	B	
2		W		

(e)

B	NW	NE	SW	SE
0		B	B	B
1		B	B	B
2		W		

(f)

Figure 3. Step by step example of color table update. (a) the color table after it has been updated by pixel 1. (b) after pixel 2. (c) after pixel 3. (d) after pixel 4.

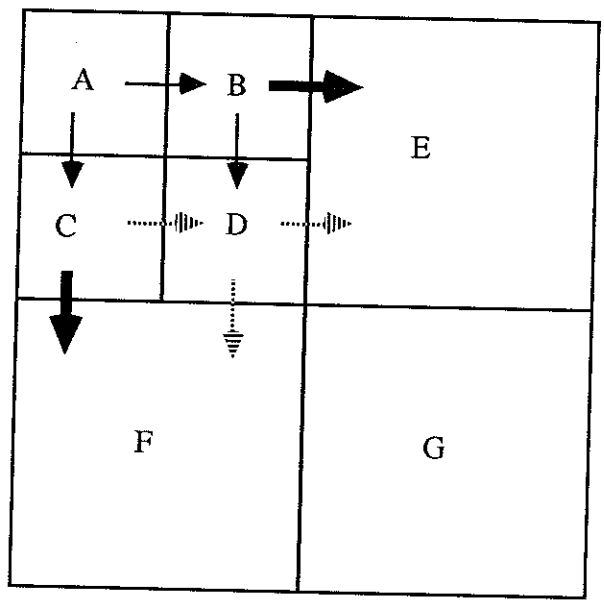


Figure 4. Node A, a NW quadrant will update entries in the color table for its siblings to the east and to the south (solid arrows). Node B will update entries in the color table for its largest eastern neighbor (Node E, bold arrow), and for its southern sibling (solid arrow). Node C will update entries in the color table for its largest southern neighbor (Node F, bold arrow). No update is necessary for the eastern sibling (broken arrow). Node D performs no updates since its neighbors are updated elsewhere.