INDEPENDENT QUADTREES

DISSERTATION

Presented to the Graduate Council of the
North Texas State University in Partial
Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

By

Larry D. Atwood, B.S., M.S.
Denton, Texas
December, 1986

TCI

Atwood, Larry D., *Independent Quadtrees.* Doctor of Philosophy (Computer Science), December, 1986, 66 pp., 1 table, 18 titles.

This dissertation deals with the problem of manipulating and storing an image using quadtrees. A quadtree is a tree in which each node has four ordered children or is a leaf. It can be used to represent an image via hierarchical decomposition. The image is broken into four regions. A region can be a solid color (homogeneous) or a mixture of colors (heterogeneous). If a region is heterogeneous it is broken into four subregions, and the process continues recursively until all subregions are homogeneous.

The traditional quadtree suffers from dependence on the underlying grid. The grid coordinate system is implicit, and therefore fixed. The fixed coordinate system implies a rigid tree. A rigid tree cannot be translated, scaled, or rotated. Instead, a new tree must be built which is the result of one of these transformations.

This dissertation introduces the independent quadtree. The independent quadtree is free of any underlying coordinate system. The tree is no longer rigid and can be easily translated, scaled, or rotated. Algorithms to perform these operations are presented. The translation and rotation algorithms take constant time. The scaling algorithm has linear time in the number nodes in the tree. The disadvantage of independent quadtrees is the longer generation and display time.

This dissertation also introduces an alternate method of hierarchical decomposition. This new method finds the largest homogeneous block with respect to the corners of the image. This block defines the division point for the decomposition. If the size of the block is below some cutoff point, it is deemed to be to small to make the overhead worthwhile and the traditional method is used instead. This

new method is compared to the traditional method on randomly generated rectangles, triangles, and circles. The new method is shown to use significantly less space for all three test sets. The generation and display times are ambiguous. More time is taken for each node, but there are, on average, fewer nodes. The worst case is significantly worse.

TABLE OF CONTENTS

Page

LIST OF TABLES iv

LIST OF ILLUSTRATIONS v

Chapter

I. INTRODUCTION 1

Quadtrees
Problems with Quadtrees
Previous Attempts at Solving Problems
Chapter Summary

II. INDEPENDENT QUADTREES 9

The Idea
A Rigorous Explanation
Chapter Summary

III. GEOMETRIC TRANSFORMATIONS OF INDEPENDENT QUADTREES 21

Translation
Scaling
Rotation
Chapter Summary

IV. COMPARISON OF DEPENDENT QUADTREES AND INDEPENDENT QUADTREES 30

Theoretical Comparison
Statistical Comparison
Worst Case Comparison
Chapter Summary

V. CONCLUSION 37

APPENDIX 40

BIBLIOGRAPHY 65

LIST OF TABLES

Table Page

I. Dependent vs. Independent Quadtrees 32

LIST OF ILLUSTRATIONS

Figure		Page
1.	A Quadtree	2
2.	A Quadtree Representing an Image	3
3.	A Quadtree is Dependent on the Underlying grid	4
4.	Constructing an Independent Quadtree	10
5.	A Treecode Representation of a Quadtree	12
6.	The Homogeneous Blocks With Respect to the Corners of the Image	15
7.	Pseudocode for BuildTree	17
8.	Pseudocode for DrawTree	18
9.	Example of Translation	23
10.	Example of Scaling	25
11.	Example of Rotation	28

CHAPTER I

INTRODUCTION

This dissertation will deal with the problem of storing and manipulating an image using quadtrees. An image is a matrix of colors. The matrix can be of any size and the colors are black and white.

Storing an N x M image would require NM bits if stored naively. There have been several attempts to reduce the amount of storage needed below NM.

Run Length Encoding (2, pp. 498-499) was one of the first attempts at storage reduction. Run Length Encoding considers the matrix row by row. Each row is broken into a series of ordered pairs. The elements of an ordered pair indicate the color and how many times that color repeats before the color changes in that row. Run Length Encoding can achieve large storage savings, however, the image is difficult to manipulate.

Another early attempt at storage reduction was the medial axis transform (10). The medial axis transform finds the set of points within a region that is equidistant from the boundary. This set of points form a skeleton (or medial axis) of the region. The medial axis transform requires extensive calculation. It is not widely used.

Quadtrees

A *quadtree* is a generalization of a binary tree. Each node of a quadtree has four children. The children are ordered. (see Figure 1.) A quadtree is used to represent an image. An image is either homogeneous (solid black or solid white) or

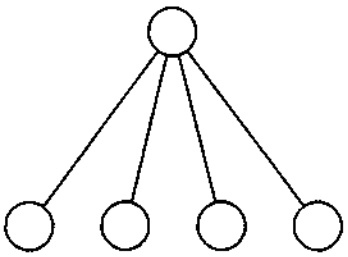


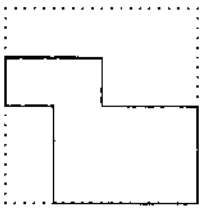
Fig. 1 A quadtree

heterogeneous (a mixture of of black and white). If the image is homogeneous then mark (B or W) the root of the quadtree as such and stop. If the image is heterogeneous then mark the root of the quadtree as such, divide the image into four subimages, and create four children of the root, corresponding to the subimages. If any of the subimages are homogeneous then mark that node as such and stop. If any of the subimages are heterogeneous then mark that node as such, divide the subimage into four subimages, and create four children of that node. Recursively continue to break up the image and create the quadtree until all nodes are homogeneous or until a desired resolution is reached. (see Figure 2.)

Extensive research has been done on the quadtree representation of an image over the last fifteen years. Samet (11) gives an excellent overview of this work and other hierarchical data structures. This work has concentrated on representations of quadtrees (3,14,6), translating other representations (e.g. polygons and binary arrays) to and from quadtrees (12,13,1,5,7), and manipulating quadtrees through some form of transformation (9,4,8,15).

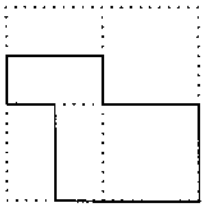
Problems with Quadtrees

There are two traditional methods of representing quadtrees; a pointer representation and a linear representation. These methods, while superficially dif-

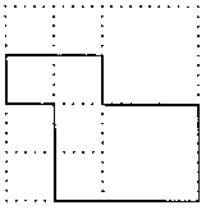


(note: interior of solid is 'Black')

The image



The image divided into four equal subregions



The heterogeneous subregions divided into equal four subregions

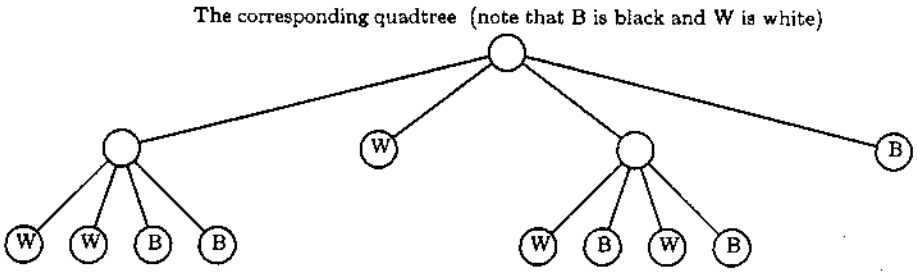


Fig. 2 A quadtree representing an image

ferent, have several assumptions in common.

First, the image is square. The quadtree is always built on a square image. Second, the length of the sides of the square is always a power of two. The square can be as large or as small as desired, as long as it is a power of two. Common values are 256, 512, and 1024.

The subregions are always of equal size. Thus we see the reason for the power of two size. Successive divisions by two will always produce equal subregions when the original size is a power of two.

The underlying grid is fixed. The lines of division that create the quadtree are fixed on the grid. The quadtree is thus dependent on the grid, and is itself fixed. (see Figure 3.)

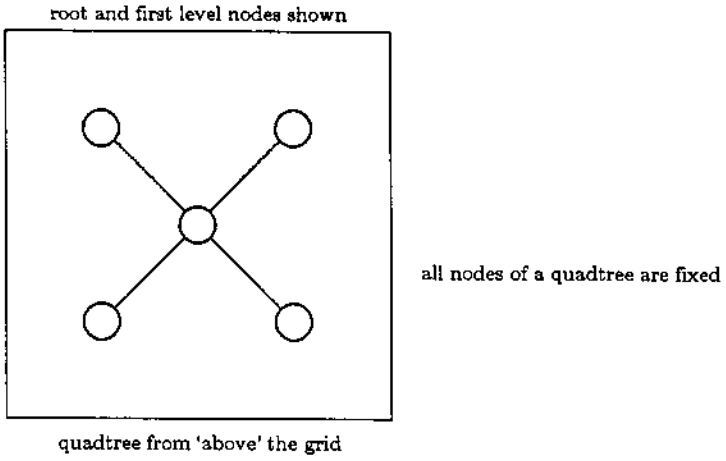


Fig. 3 A quadtree is dependent on the underlying grid

Intuitively, the quadtree is a rigid structure attached to a grid. The quadtree cannot move. Thus the three standard transformations in computer graphics, translation, scaling, and rotation, are difficult, if not impossible to implement.

Previous Attempts at Solving Problems

It is well known that the special cases of scaling by powers of two and rotations by 90 or 180 degrees is easy. The quadtree's dependence on the underlying grid makes general translation, scaling, and rotation extremely difficult. The grid's square size and limitation to powers of two make the use of quadtrees on any rectangular image impractical. All previous attempts at solving these problems have stayed within the conceptual framework of the dependent quadtree. Below is a summary of the most important work in this area.

Oliver and Wiseman (9) use a linear (non-pointer) representation of a quadtree. They develop several algorithms capable of manipulating their representation; including translation and rotation. Their translation algorithm is only briefly described and they admit another type of representation is probably better. Their rotation algorithms are limited to 90 and 180 degrees. They don't mention scaling.

Hunter and Steiglitz (4) describe a complicated algorithm for building a transformed quadtree after a general linear operator has been applied to the original quadtree. They show that the algorithm has time complexity O(n+sp+mq) where n is the number of nodes in the original tree, s is a scale factor, p is the perimeter of the original image, m is the number of regions, and q is a resolution parameter. This algorithm requires a special quadtree representation in which the leaves are directly joined to their neighbor. This representation is called a netted quadtree.

Li, Grosky, and Jain (8) recognize the rigid nature of the quadtree. They describe a method to build a normalized quadtree. Normalized here means a minimum representation, in terms of space. They find the minimum tree by moving the image on the grid. Their algorithm takes O(s^2*log(s)) where s is the

length of the grid.

van Lierop (15) describes an algorithm for applying general geometric transformations to a linear representation of a quadtree. He does this by applying the transformation to the leaves of the quadtree and generating the leaves of the transformed quadtree. The algorithm takes O(M*(n+log(N))) time, where M is the number of nodes in the output tree, N is the number of leaves input, and n is the resolution of the grid.

Chapter Summary

The quadtree is introduced. It suffers from dependence on the underlying grid. Previous attempts at manipulating the quadtree do not fully recognize this dependence, and are thus complex and time consuming.

CHAPTER BIBLIOGRAPHY

1. Dyer, Charles R., Rosenfeld, Azriel, and Samet, Hanan, "Region Representation: Boundary Codes from Quadtrees," *Communications of the ACM*, vol. 23, pp. 171-179, March 1980.

2. Foley, J. D. and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA., 1982.

3. Gargantini, Irene, "An Effective Way to Represent Quadtrees," *Communications of the ACM*, vol. 25, pp. 905-910, December 1982.

4. Hunter, G. M. and Steiglitz, K., "Linear Transformations of Pictures Represented by Quad Trees," *Computer Graphics and Image Processing*, vol. 10, pp. 289-296, 1979.

5. Hunter, Gregory M., and Steiglitz, Kenneth, "Operations on Images Using Quad Trees," *IEEE Transactions of Pattern Analysis and Machine Intelligence*, vol. 1, pp. 145-153, April 1979.

6. Jackins, C. L. and Tanimoto, S.L., "Decompostion of Euclidean Space," *IEEE Transactions of Pattern Analysis and Machine Intelligence*, vol. 5, pp. 533-539, Sept. 1983.

7. Kawaguchi, Eiji and Endo, Tsutomu, "On a Method of Binary-Picture Representation and Its Application to Data Compression," *IEEE Transactions of Pattern Analysis and Machine Intelligence*, vol. 2, pp. 27-35, January 1980.

8. Li, Ming, William I. Grosky, and Ramesh Jain, "Normalized Quadtrees with Respect to Translations," *Computer Graphics and Image Processing*, vol. 20, pp. 72-81, 1982.

9. Oliver, M. A. and N. E. Wiseman, "Operations on Quadtree Encoded Images," *The Computer Journal*, vol. 26, pp. 83-91, 1983.

10. Pavlidis, Theo, *Algorithms for Graphics and Image Processing*, Computer Science Press, Rockville, MD, 1982.

11. Samet, Hanan, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, vol. 16, pp. 187-260, June 1984.

12. Samet, Hanan, "An Algorithm for Converting Rasters to Quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 3, pp. 93-95, January 1981.

13. Samet, Hanan, "Region Representation: Quadtrees from Boundary Codes," *Communications of the ACM*, vol. 23, pp. 163-170, March 1980.

14. Tamminen, Markku, "Comment on Quad- and Octtrees," *Communications of the ACM*, vol. 27, pp. 248-249, March 1984.

15. van Lierop, Marloes L. P., "Geometrical Transformations on Pictures Represented by Leafcodes," *Computer Vision, Graphics, and Image Processing*, vol. 33, pp. 81-98, 1986.

CHAPTER II

INDEPENDENT QUADTREES

This dissertation proposes a new type of quadtree, the *independent quadtree.* An independent quadtree is a quadtree whose position and orientation is independent of the implicit coordinate system of the represented image. The independent quadtree is free of any underlying grid and can represent an image of any size, not just a square power of two. Later we shall see that the independent quadtree makes geometric transformations (translation, scaling, and rotation) quite easy.

The Idea

In order to make a quadtree independent, it has to be "freed" from dependence on the underlying grid. The dependence manifests itself in the implicit coordinate system of the grid. Traditionally, this coordinate system has been used in the quadtree. The implicit coordinate system has the advantage of not having to explicitly store any coordinates, and the disadvantage of "binding" the quadtree to the grid. This dissertation proposes that the coordinate system used by the quadtree be made explicit. The independent quadtree has coordinate information as part of it's data structure. The coordinate information will consist of the location and size of the image. Supplying location and size information removes the "square" and "powers of two" limitations.

The independent quadtree is constructed in a manner quite similar to the dependent quadtree (2). (see Figure 4.)

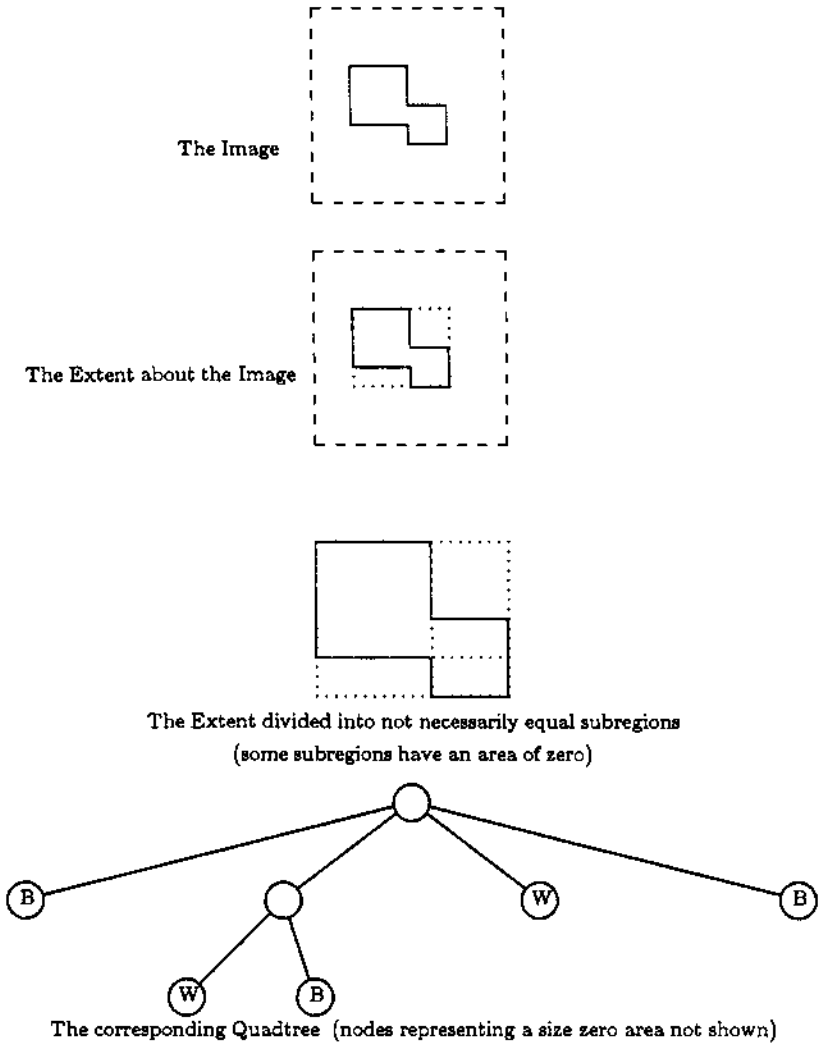


Fig. 4 Constructing an independent quadtree

First, the location and size of the image is found. This is done by constructing a bounding box around the image. Note that this is already implicitly present for the dependent quadtree. Next, the homogeneity test is applied. If the image (within the bounding box) is homogeneous then the algorithm terminates. If the image is heterogeneous the image is decomposed into four regions and continues the process recursively. For the dependent quadtree these regions would always be equal. However, for the independent quadtree the regions will be chosen according to a

different criteria.

Decomposing a region into four subregions is a difficult problem in both the dependent and independent cases. Equal size regions is the easiest to do, but this dissertation will propose something different. The largest homogeneous block with respect to the four corners will be found. This block will form the basis for the decomposition. The information about each decomposition will have to stored since it is no longer implicit.

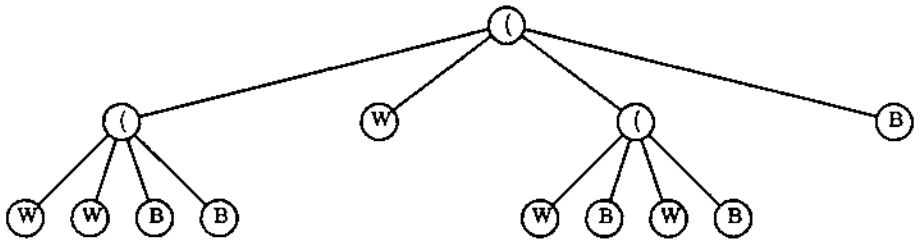
Converting the tree back into a binary grid is done in a manner quite similar to the dependent case. A Depth First Search (1; pp. 268-269) is done until a homogeneous block is found. At each node in the tree the appropriate offsets are added to the location. For dependent quadtrees it is always known exactly what those offsets would be; for independent quadtrees the information will have been stored when the tree was constructed.

If the image lacks coherence, (e.g. a matrix of randomly distributed points) then storing these offsets can take an excessive amount of storage. The storage of specific offsets can be suspended when the size of a region falls below a certain cut-off point. The tree below that cutoff point returns to an equal division, which is implicit, and does not have to store offsets. This cutoff point can be selected from within the program by setting the variables RowCutoff and ColCutoff.

A Rigorous Explanation

A linear representation known as a treecode will be used for demonstration purposes. This is for convenience only, all results will also apply to pointer representations. A treecode uses three symbols to form a sequence that will represent the tree. The three symbols are; 'B' to indicate a Black node, 'W' to

indicate a White node, and '(' to indicate that the node is heterogeneous and will be decomposed. (see Figure 5.)



would become: ((WWBBW(WBWBB

Fig. 5 A Treecode representation of a quadtree

An independent quadtree is a quadtree whose position and orientation is independent of the implicit coordinate system of the represented image.

Assume that the image is represented by a binary array of arbitrary size. Later we will make comparison to the dependent tree, so the image used here will be 256 x 256. This square power of two is used ONLY because the dependent tree requires it. It will become clear that the independent tree can be built out of arbitrary size arrays.

The following data structures will describe the independent quadtree.

```
treetype   = packed array [ 0..treemax ] of char;
offsettype = packed array [ 1..nodemax, 0..1 ] of range;
extenttype = record
               Row, RowSize,
               Col, ColSize  : integer;
               DX, DY        : real;
             end;
```

The type treetype is of char for purposes of clarity only. The only values that this array can take on are 'B', 'W', and '('. In a real application 2 bits would be sufficient. Since Pascal arrays are static the constant treemax is some large number. The type offsettype will contain an (X,Y) value for each node in the tree. The constant nodemax is some large number. The subrange range is 0..255, therefore 16 bits per node will be used beyond what the dependent tree uses. The type extenttype defines the location (Row, Col), the size (RowSize, ColSize), and the orientation (DX, DY) of the quadtree initially. It is extenttype that achieves the independence from the underlying image grid. Offsets will be computed with respect to the Extent as defined in extenttype. Note that the orientation will always be perpendicular to the sides of the image initially.

There are two operations of interest. The first is constructing the quadtree from a binary array. The second is displaying the image represented in the quadtree, i.e. reconstructing the binary array. The first operation is implemented in a Pascal procedure called GenerateIndependentTree. The second operation is implemented in a Pascal procedure called DisplayIndependentTree. These procedures will be explained in detail here. Complete listings of both procedures can be found in the Appendix.

GenerateIndependentTree begins by determining the extent of the image. This is done by a call to the procedure Bound. Bound determines the upper, lower, left, and right bounds of the image in it's grid. Bound first determines the lower and upper bounds. Then it uses these new bounds to determine the left and right bounds. These bounds are then used to determine the location and size of the image. Location will be assumed to be the upper left corner, and size will be a pair of numbers; row size and column size. GenerateIndependentTree then calls

BuildTree, which is the heart of GenerateIndependentTree.

BuildTree is the recursive part of GenerateIndependentTree. BuildTree is invoked four times at each heterogeneous region, i.e. once for each subregion. BuildTree begins by checking the size of the region that it is checking for homogeneity. It is possible that the region is nonexistent (size zero). If the region is nonexistent then no recursion occurs and BuildTree is done.

BuildTree then checks if the region size is below the cutoff point. If it is, then the subdivision will be based on equal size subregions. However, the region size at this point is not necessarily a power of two, so the size of a region will not always be divisible by two. There will be a remainder of one at times. BuildTree arbitrarily chooses to add the one, if it exists, to the upper left subregion. The recursive decomposition continues on this basis.

If the region size is greater than the cutoff point the subdivision will be done on the basis of the largest homogeneous block with respect to a corner. (see Figure 6.) The size of that block is used to compute the subdivision for this region. The subdivision is stored as offsets. The color of the homogeneous block with respect to each corner is stored in an array called BlackOrWhite. BuildTree finds the homogeneous blocks by calling the procedure HomogeneousBlock four times, once for each corner of the region.

HomogeneousBlock has seven parameters. The first two indicate which corner of the region is to be operated on. The third and fourth parameters indicate which direction from the corner is to be considered. The fifth parameter is the size of the block for that corner as found by the procedure. The last two parameters are the length of the sides of the block. HomogeneousBlock tries to construct, a row at a time, a block which is the same color as the corner value. The area represented by

the addition of a new row is compared to the previous largest value. If the new area is larger, it becomes the area value. This is done for all rows in the region, so we are guaranteed that the area returned will be the largest possible, with respect to that corner.

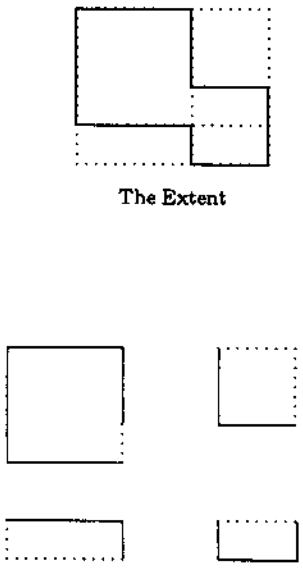


Fig. 6 The homogeneous blocks with respect to the corners of the image

After obtaining the size of the four homogeneous blocks BuildTree finds the largest one, which will be the largest in that region with respect to any corner. BuildTree also finds the color of the block.

BuildTree will then set the offsets for this node to the row size and column size for the largest block.

Next, BuildTree invokes itself three times, once for each of the smaller blocks. The largest block is homogeneous, by definition, and does not require further processing.

After the last quadrant has been processed for this region BuildTree adds the temporary element ')' to the tree. This is used by the later condensation procedure and is NOT a permanent part of the tree. After the last quadrant has been processed it may be possible to condense the block to a smaller block. BuildTree invokes the procedure Condense to perform the condense operation. Condensation is possible in the following cases:

(B) or (W) will condense to B or W respectively

(BB) or (WW) will condense to B or W respectively

(BBBB) or (WWWW) will condense to B or W respectively

Condense is implemented as a Finite State Automata.

After BuildTree returns from it's recursive decomposition of the image, control returns to GenerateIndependentTree. GenerateIndependentTree removes the ')' from the tree, collects some statistics and ends. Figure 7 gives the pseudo code for the BuildTree.

DisplayIndependentTree begins by setting maximum bounds for the image. These will later be used for any necessary clipping. DisplayIndependentTree then invokes DrawTree, the heart of the algorithm.

DrawTree first checks the size of the region it is to display. If the region is nonexistent (size zero) then no recursion occurs and DrawTree is done. DrawTree then checks to see if the region is less than the cutoff point.

If the region size is less or equal to the cutoff then the display of the tree will proceed in the traditional manner. The region will either be colored if it is homogeneous, or it will be divided into equal parts along both axes. Since the region is not necessarily a square power of two in size, the division by two will not always be

```
begin BuildTree
        if region_size = 0 then stop

        if region_size < cutoff then

                output ( '(' )

                divide region into equal size subregions

                if pixel then output ( color_of_pixel )
                else
                        BuildTree ( upper_left_corner )
                        BuildTree ( upper_right_corner )
                        BuildTree ( lower_left_corner )
                        Buildtree ( lower_right_corner )
        else
                output ( '(' )

                find largest homogeneous blocks w.r.t. the four corners

                if largest is upper_left then output (upper_left_color)
                else
                        BuildTree ( upper_left_block )
                if largest is upper_right then output ( upper_left_color )
                else
                        BuildTree ( upper_right_block )
                if largest is lower_left then output ( lower_left_color )
                else
                        BuildTree ( lower_left_block )
                if largest is lower_right then output ( lower_right_color )
                else
                        Buildtree ( lower_right_block )
        CondenseTree
end BuildTree
```

Fig. 7 Pseudocode for BuildTree

exact. There may be a remainder of one. The remainder of one will be added to the upper left region to correspond to what was done in the construction of the tree.

If the region size is greater than the cutoff then the region will either be colored if it is homogeneous, or the location and size of the four subregions must be computed on the basis of the offsets for the current node. Once the location and size are determined DrawTree can invoke itself recursively. Figure 8 gives the

pseudo code for DrawTree.

```
begin DrawTree
        if region_size = 0 then stop

        if region_size < cutoff then

                if next ( Tree ) in [ 'B', 'W' ] then DrawBlock
                else
                        DrawTree ( upper_left )
                        DrawTree ( upper_right )
                        DrawTree ( lower_left )
                        DrawTree ( lower_right )

        else
                if next ( Tree ) in [ 'B', 'W' ] then DrawBlock
                else
                        compute size and location of the four subregions

                        DrawTree ( upper_left )
                        DrawTree ( upper_right )
                        DrawTree ( lower_left )
                        DrawTree ( lower_right )

end DrawTree
```

Fig. 7 Pseudocode for DrawTree

The procedure that actually draws a homogeneous block into the binary array is called DrawBlock. DrawBlock computes the four corners of the homogeneous block. If the four corners represent a region whose sides are parallel to the implicit axes of the grid, then a simple row by row fill is performed. If the four corners form a rectangle that is not parallel to the implicit axes then a simplified version of the standard polygon fill algorithm is performed. It is a simplified fill algorithm because it is known that there will always be exactly four vertices. The simplified polygon fill algorithm determines the upper and lower bounds of the polygon. Then for each row in the range between upper and lower a left and right bound is found. The binary array is filled between left and right inclusive. Clipping will be performed when necessary.

Chapter Summary

The independent quadtree is introduced. It is independent of the underlying grid. This independence is achieved by making the coordinate system of the quadtree explicit. Algorithms for constructing and displaying the independent quadtree are presented.

Within the construction algorithm a new method of decomposition is presented. This method divides the image by finding the largest homogeneous block with respect to a corner and using that block to indicate the division points of the image.

CHAPTER BIBLIOGRAPHY

1. Horowitz, Ellis and Sahni, Sartaj, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD., 1978

2. Samet, Hanan, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, vol. 16, pp. 187-260, June 1984.

CHAPTER III

GEOMETRIC TRANSFORMATIONS OF INDEPENDENT QUADTREES

The data structures that form the independent quadtree make geometric transformations simple and elegant. Freeing the quadtree from the image grid makes the tree itself coordinate free. All of the location, size, and orientation information are contained in the record variables Extent and Offset.

Translation

Translation is the movement of an object. If the object is represented by polygons then the movement is accomplished by adding the desired offset to the vertices of the polygons (1; pp. 245-246). Translation of a dependent quadtree is far more complex. An entirely new tree has to be constructed since the old quadtree is fixed with respect to the underlying grid.

However, translation of the independent quadtree is easy. Location information for the whole image is stored in Extent. To translate an independent quadtree requires only adding the desired offset to the location of the tree. The tree structure itself, being independent of any coordinate system, is untouched.

Here is the entire Translate procedure:

```
procedure Translate ( var Extent : extenttype;
                      DeltaX,
                      DeltaY : integer );
```

```
begin { Translate }

    Extent.Row := Extent.Row + DeltaY;
    Extent.Col := Extent.Col + DeltaX;

end;  { Translate }
```

This is, obviously, a constant time algorithm. In fact, there are only two integer additions to the whole operation. Figure 9 shows an example of a circle translated by (100,100). The image printing software does not correct for distortion caused by the aspect ratio of the printer, so the circle looks flattened. The origin is the upper left corner.

Scaling

Scaling is the enlargement or reduction of an object. If the object is represented by polygons then the scaling is accomplished by multiplying the vertices by the desired scaling factors in X and Y (1; p. 247).

Scaling by a power of two is well known (2). This is done by increasing (or reducing) the size of the tree by one level. If the scaling is up by a factor of two then one of the four child nodes of the root is chosen as the new root. The display size is the same so the image of the chosen node is scaled up by a factor of two in X and Y. If the scaling is down by a factor of two then three blank nodes are added to the current root and a new root is created above them. The display size is the same so the old image is scaled down by a factor of two.

Scaling by an arbitrary integer, or even continuous scaling using reals is unknown in the literature.

Scaling an independent quadtree is not only easy but it can be done using arbitrary real numbers. Scaling is performed by multiplying the size of the Extent

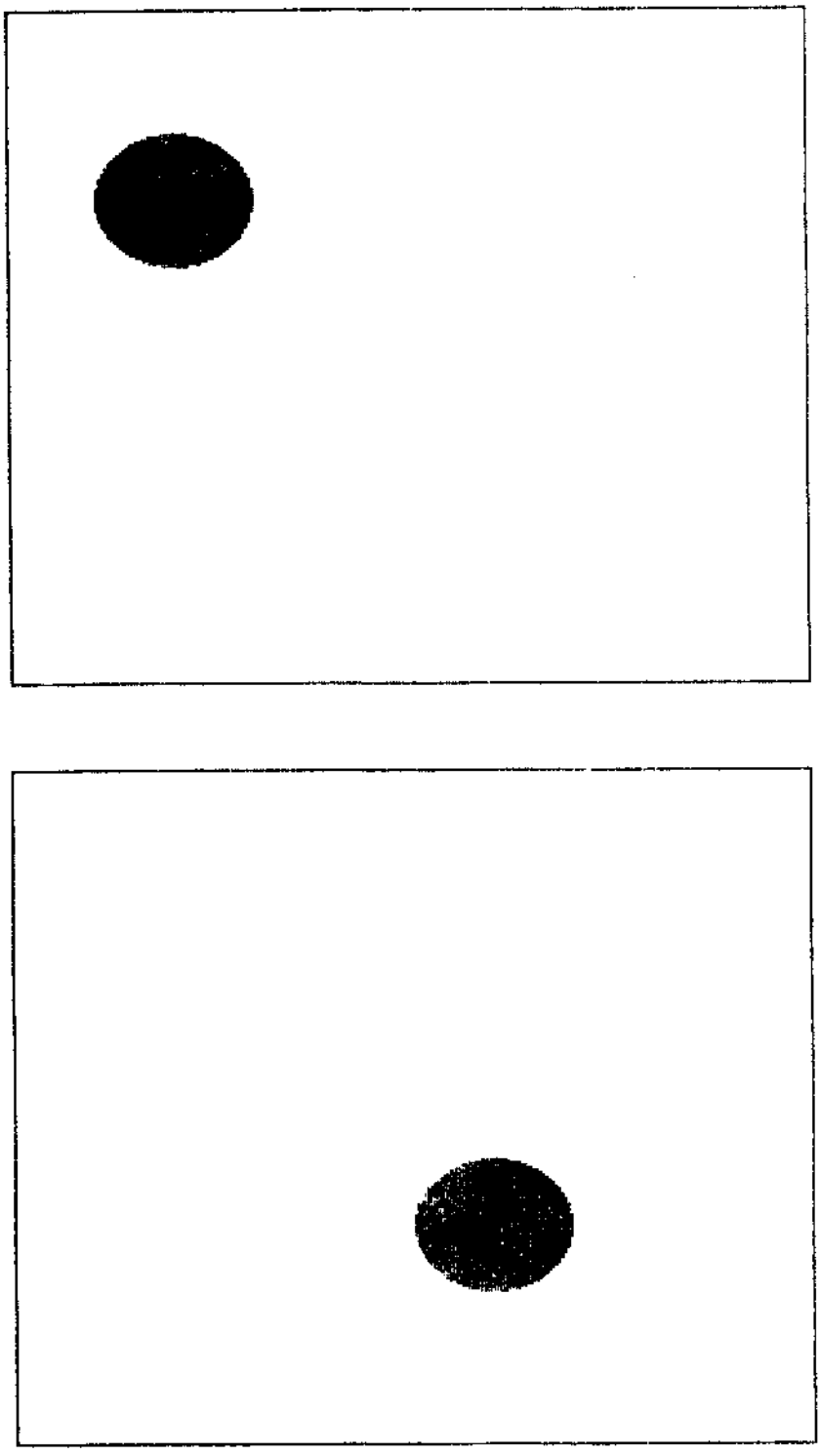


Fig. 9 Example of Translation

by the desired values, and multiplying all of the offsets by the desired values. The tree structure itself, being independent of any coordinate system, is untouched.

Here is the entire scaling procedure:

```
procedure Scale ( var Extent : extenttype;
                  var Offset : offsettype;
                      SX,
                      SY     : real        );

var

    i : integer;

begin { Scale }

    Extent.RowSize := round (Extent.RowSize * SY);
    Extent.ColSize := round (Extent.ColSize * SX);

    i := 1;
    while Offset[i,0] <> -1 do begin

        Offset[i,0] := round ( Offset[i,0] * SY );
        Offset[i,1] := round ( Offset[i,1] * SX );

        i := i + 1;

    end;


end;  { Scale }
```

This algorithm is obviously linear in the number of nodes in the tree. Figure 10 shows an example of a triangle scaled by a factor 3.0 in X and 5.0 in Y. The scaling is done with respect to the upper left corner of the Extent. The origin is the upper left corner.

In this implementation scaling only works if the cutoff values are set to zero. This is a software problem, not a theoretical problem. It has no bearing on the validity of independent quadtrees.

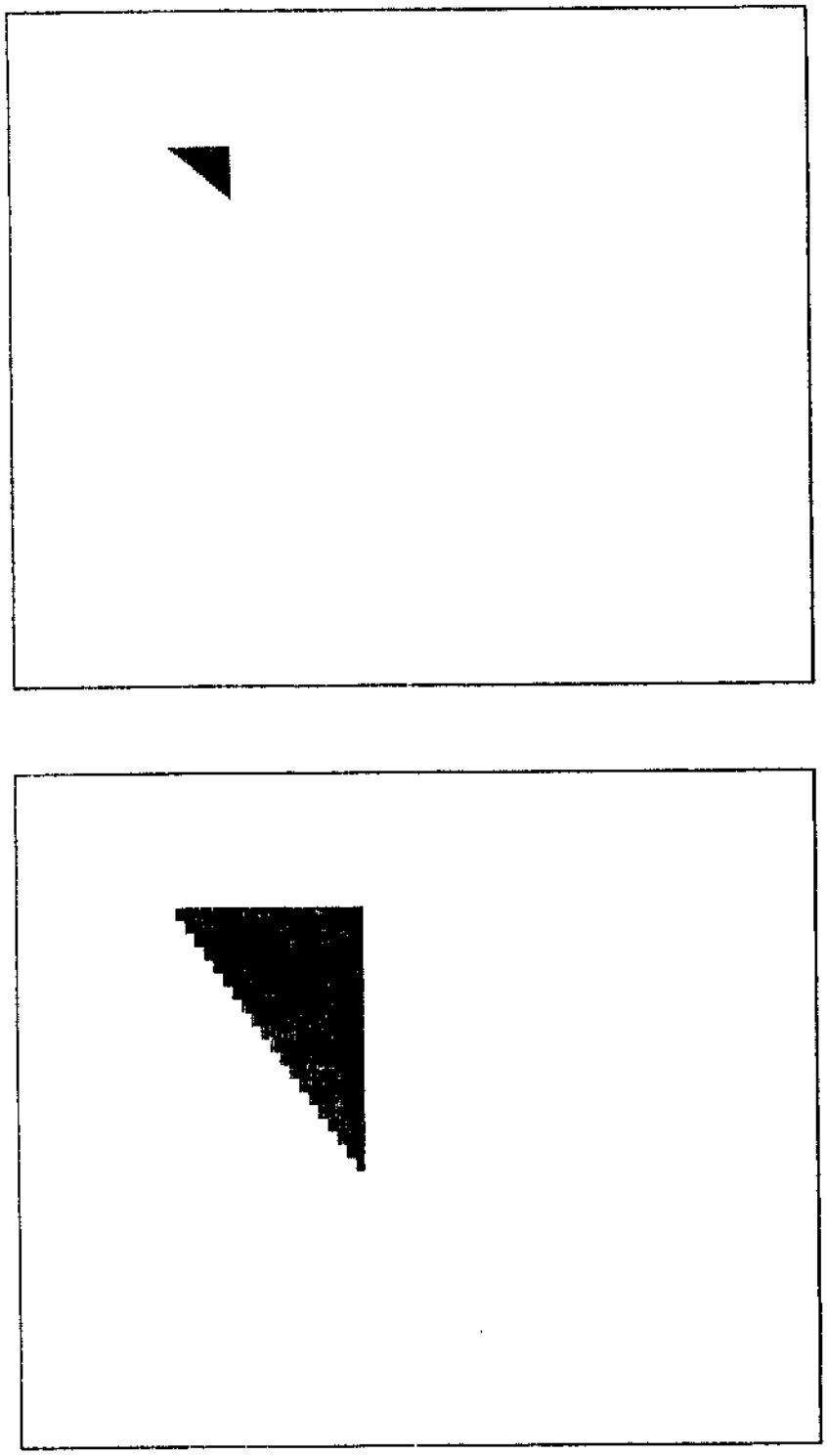


Fig. 10 Example of Scaling

Rotation

Rotation is the movement through an angle about a point. If an object is represented as polygons then the operation is performed by applying the following trigonometric transformations to each vertex:

```
X := X * cos(theta) - Y * sin(theta)
Y := Y * sin(theta) + Y * cos(theta)
```

where theta is the angle of rotation (1; p. 248).

Rotation of a dependent quadtree is unknown in the literature. As discussed in chapter II there are some general linear transformation algorithms. These algorithms construct the new tree that results from a transformation.

Rotation of an independent quadtree is quite easy. It is only necessary to apply the transformations mentioned above to the orientation parameters (DX & DY) in Extent. The tree structure itself, being independent of any coordinate system, is untouched.

Here is the entire rotation procedure:

```
procedure Rotate ( var Extent : extenttype;
                   Angle  : real        );

{ Angle should be in radians }

var
    T1,  T2,
    CosA, SinA : real;

begin { Rotate }

    CosA := cos(Angle);
    SinA := sin(Angle);

    T1 := Extent.DX * CosA - Extent.DY * SinA;
    T2 := Extent.DX * SinA + Extent.DY * CosA;

    Extent.DX := T1;
    Extent.DY := T2;
```

```
end;  { Rotate }
```

This algorithm is obviously a constant time algorithm. Figure 11 shows an example of a rectangle being rotated by approximately 45 degrees. The image printing software does not correct distortion caused by the aspect ratio of the printer, so the rectangle looks flattened. The origin is the upper left corner.

Translation, Scaling, and Rotation can be combined to achieve any linear transformation of an independent quadtree. The transformations are, in general, not commutative. Foley and Van Dam (1; pp. 253-254) show that translation can be commuted with translation, scaling can be commuted with scaling, rotation can be commuted with rotation, and scaling can be commuted with the other two when the X and Y scale factors are equal.

Chapter Summary

Algorithms for the translation, scaling, and rotation, of independent quadtrees are presented. Independent quadtrees can be translated by any real number offset (subject to roundoff). The translation algorithm is constant in time and space. Independent quadtrees can be scaled by any real number (subject to roundoff). A current software limitation is that the cutoff point must be set to zero. The scaling algorithm is linear in the number of nodes of the tree. Independent quadtrees can be rotated by any real angle (subject to roundoff). The rotation algorithm is constant in time and space.

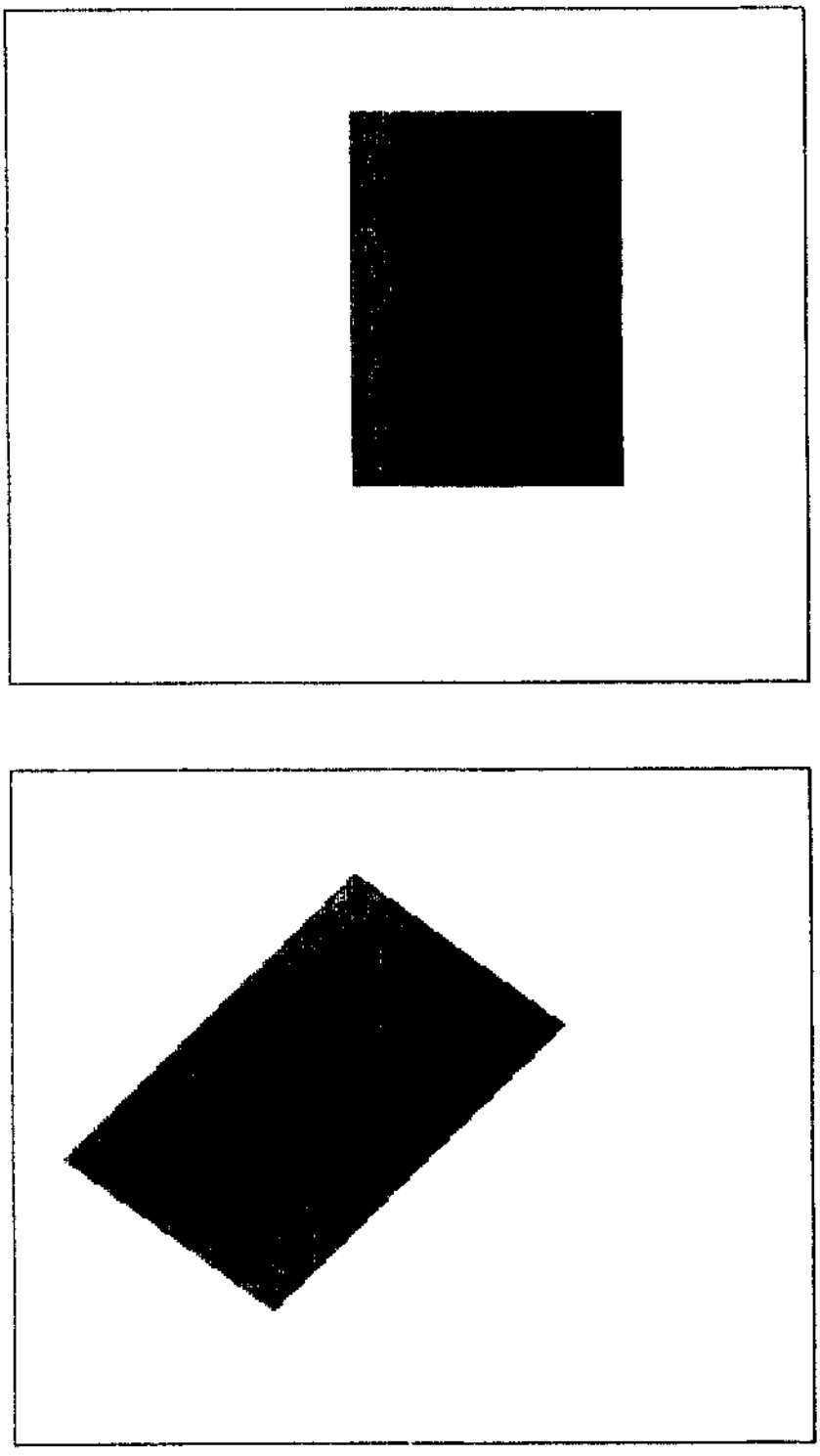


Fig. 11 Example of Rotation

CHAPTER BIBLIOGRAPHY

1. Foley, J. D. and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA., 1982.

2. Samet, Hanan, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, vol. 16, pp. 187-260, June, 1984.

CHAPTER IV

COMPARISON OF DEPENDENT QUADTREES AND INDEPENDENT QUADTREES

Independent quadtrees have a major advantage over dependent quadtrees in their ability to manipulate the image. The comparison of the generation and display of independent quadtrees with the generation and display of dependent quadtrees is the subject of this chapter. For comparison purposes two procedures called GenerateDependentTree and DisplayDependentTree were constructed. These procedures were constructed in a manner analogous to that suggested by Samet (3). Listings of GenerateDependentTree and DisplayDependentTree can be found in the Appendix.

Theoretical Comparison

First we consider the time requirements for the generation of the quadtree. GenerateDependentTree visits every element in the image array exactly once. So GenerateDependentTree is O(e) where e is the number of elements in the image. GenerateIndependentTree may visit an element more than once, but never more than the number of levels in the tree (as it searches for the homogeneous blocks). Let l be the number of levels in the tree. Then, GenerateIndependentTree is O(l*e) where e is the number of elements in the image.

Next, we consider the time requirements for the display of the quadtree. DisplayDependentTree visits each node in it's quadtree exactly once.

DisplayIndependentTree visits each node in it's quadtree exactly once. Both algorithms are therefore, O(n) where n is the number of the nodes in the tree. The algorithms are equivalent up to a constant term; however, the number of nodes in each tree may not be the same.

Next, we consider the space requirements for the quadtree. Both dependent and independent quadtrees require two bits per node (storing 'B', 'W', or '('). In addition, the independent quadtree has the overhead of the extent which is four integers and two reals. The independent quadtree may also store offset information at each node. The independent quadtree has, therefore some overhead and larger nodes than the dependent quadtree. However, the approach that the independent quadtree takes to region decomposition (largest homogeneous block) may lead to fewer nodes. The actual space determinations are data dependent.

Statistical Comparison

The theoretical comparisons are not very enlightening. To obtain a better understanding of how independent and dependent quadtrees compare with each other we need to run them and test for significant differences. The four procedures GenerateDependentTree, GenerateIndependentTree, DisplayDependentTree, and DisplayIndependent were implemented in Pascal running on a VAX 11/780 under Unix* 4.2BSD. Pascal was chosen for its algorithmic clarity. C would be a better choice for efficiency.

Three sets of test data were used, rectangles, triangles, and circles. These objects represent the three types of boundaries between object and background; horizontal and vertical, sloping, and curved. Each element in a test set was of

*Unix is a registered trademark of Bell Labs

random size and position. One object was generated per image. There were 500 random objects in each set. All of the object will be contained within the grid, i.e. no need for clipping. The average time and space usage of dependent and independent quadtrees is shown in Table I.

TABLE I
DEPENDENT VS. INDEPENDENT QUADTREES

Test Set	Dependent	Independent
Space* Usage		
Rectangles	1815	130
Triangles	2629	2241
Circles	1345	1186
Generation Time*		
Rectangles	7050	1759
Triangles	7392	3675
Circles	6975	1983
Display Time*		
Rectangles	184	96
Triangles	195	484
Circles	116	228

*space is in bits, time is in milliseconds

The table shows that space usage was less for independent quadtrees over all three test sets. Independent quadtrees also took less time to generate over all three test sets. The results for display were less straightforward. The independent quadtrees for rectangles took less time to display, but they took more time for triangles and circles.

In order to test the statistical significance of these results, 95% confidence intervals for the difference in means were computed. Since the population variance is not known the Student's t distribution was assumed (1; pp. 262-264). The null hypothesis is that the there is no significant difference between independent and dependent quadtrees for space, generation time or display time, over any of the test sets. This can be accepted or rejected by the presence or absence of zero in confi-

dence intervals for the difference of the means.

The confidence intervals for space were:

Rectangles	1592	<	D	<	1778
Triangles	281	<	D	<	494
Circles	70	<	D	<	247

where D is the difference, dependent minus independent. The numbers represent bits. Zero is not present in any of these intervals so the null hypothesis of no difference is rejected. Independent quadtrees use less space. This is easily accounted for by the adaptive nature of the independent quadtree. It tries to find large homogeneous blocks, implying less storage if the image has any coherence to it.

The confidence intervals for generation time were:

Rectangles	5199	<	D	<	5389
Triangles	3564	<	D	<	3870
Circles	4907	<	D	<	5077

where D is the difference, dependent minus independent. The numbers are in milliseconds. Zero is not present in any of these intervals so the null hypothesis of no difference is rejected. Independent quadtrees take less time to generate. This is a somewhat surprising result. The theoretical analysis seemed to indicate that the independent quadtrees should take more time. But again, the adaptive nature of the algorithm means fewer blocks, even though each block takes longer to generate. However, the difference is probably not as pronounced as it might seem from the confidence intervals. GenerateDependentTree is highly recursive. A nonrecursive version of the algorithm would probably show gains relative to GenerateIndependentTree. The amount of gain is hinted at by the results for display time.

The confidence intervals for display time were:

Rectangles	71	<	D	<	104
Triangles	-311	<	D	<	-267
Circles	-126	<	D	<	-98

where D is the difference, dependent minus independent. The numbers represent milliseconds. Zero is not present in any of the intervals so the null hypothesis of no difference is rejected. The display of an independent quadtree representing a rectangle takes less time, while triangles and circles take more. The adaptive algorithm is displaying it's bias for objects that have boundaries perpendicular to the axes of the image. The trees generated for triangles and circles were not small enough to overcome the inherently longer time per block taken by the independent tree.

Worst Case Comparison

The worst kind of image for a dependent quadtree is a checkerboard at lowest resolution (2; p. 217). A checkerboard is even worse for an independent quadtree. A test was run on a 128 x 128 checkerboard. The independent quadtree used 4.6 times more memory. The independent quadtree took 17 times longer to generate. The independent quadtree took 7 times longer to display. Clearly, in the worst case, the independent quadtree is inferior to the dependent quadtree.

Chapter Summary

Dependent and independent quadtrees are compared theoretically and empirically. Theoretically independent quadtrees should take longer to generate and display. The space requirements of quadtrees are data dependent.

Empirically, the independent quadtree takes less space. The empirical results indicate that the independent quadtree take less time to generate, but this may be

due to the intensely recursive nature of the dependent algorithm. The independent quadtree takes longer to display.

Overall, these results suggest a space-time trade-off. The independent quadtree will take less space, but may take longer to generate and will take longer to display, for a generalized object.

CHAPTER BIBLIOGRAPHY

1. Hoel, Paul G., *Introduction to Mathematical Statistics (Fourth Edition)*, John Wiley & Sons, Inc., New York, 1971.

2. Samet, Hanan, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, vol. 16, pp. 187-260, June 1984.

3. Samet, Hanan, "Region Representation: Quadtrees from Binary Arrays," *Computer Graphics and Image Processing*, vol. 13, pp. 88-93, 1980.

CHAPTER V

CONCLUSION

A quadtree is a tree in which each node has four ordered children or is a leaf. It can be used to represent an image via hierarchical decomposition. The image is broken into four regions. A region can be a solid color (homogeneous) or a mixture of colors (heterogeneous). If a regions is heterogeneous it is broken into four regions, and the process continues recursively until all regions are homogeneous (or until a specified limit is reached).

The traditional quadtree suffers from dependence on the underlying grid. The grid coordinate system is implicit, and therefore fixed. The fixed coordinate system implies a rigid tree. A rigid tree cannot be translated, scaled, or rotated. Instead, a new tree must be built which is the result of one of these transformations.

This dissertation has introduced the independent quadtree. The independent quadtree is free of any underlying coordinate system. The tree is no longer rigid and can be easily translated, scaled, or rotated. Algorithms to perform these operations have been presented. The translation and rotation algorithms take constant time. The scaling algorithm has linear time in the number nodes in the tree. The disadvantage of independent quadtrees is the longer display time. However, the ease with which an image may be transformed should more than compensate for the longer display time.

This dissertation has also introduced an alternate method of hierarchical decomposition. This new method finds the largest homogeneous block with respect

to the corners of the image. This block defines the division point for the decomposition. If the size of the block is below some cutoff point, it is deemed to be to small to make the overhead worthwhile and the traditional method is used instead. This new method is compared to the traditional method on randomly generated rectangles, triangles, and circles. The new method has been shown to use significantly less space for all three test sets. The generation and display times are ambiguous. More time is taken for each node, but there are, on average, fewer nodes. The worst case is significantly worse.

Future Work

The scaling algorithm should be made to work for any value of the cutoff point. It can almost certainly be improved to constant time. This will require changes to GenerateIndependentTree and DisplayIndependentTree, which may make them run even slower. Constant time algorithms for all three transformations should make the slower time worthwhile.

Obviously, the worst case performance needs to be improved. This can be accomplished by making the algorithm even more adaptive. It should be possible to dynamically determine when the algorithm should switch from a irregular decomposition (homogeneous blocks) to a regular decomposition (equal sizes). In fact, it should be possible to switch back and forth between the two types of decomposition.

There is still at least one major obstacle to quadtrees becoming an important modeling tool. Quadtrees are hard to encode. This dissertation assumed that the image was already in binary form. But the major representation in use today in computer graphics is the polygon, i.e. a set of vertices. It is easy to describe a

polygon, it is not easy to describe a quadtree. An interactive solution may be feasible.

The 3D case

This work should easily generalize to three dimensions. In three dimensions the quadtree is known as an octtree. Each dimension is divided along it's axis. This implies eight regions, thus an octtree. The traditional method, as for the quadtree, is to divide the space into equal parts. The disadvantages of this approach have been well documented by this dissertation (see chapter I) and should hold for the three dimensional case as well as the two dimensional case. The movement of objects through space, once a major challenge, should be significantly simplified by using independent octtrees. The major advantages of octtrees is the simple solution they offer to the hidden surface problem. Independent octtrees should still have that advantage and offer fast transformations as well.

Generalizing to four dimensions may be useful as well. Four dimensions would introduce a time element. The possibility of animation is intriguing. Beyond four dimensions is not interesting to the field of computer graphics.

APPENDIX

This appendix contains listings of the procedures described in this dissertation. Those procedures are *GenerateDependentTree, DisplayDependentTree, GenerateIndependentTree, and DisplayIndependentTree.* It also contains a listing of the global variables used by those procedures.

```
const
    min         = 0;
    max         = 255;

    treemax     = 60000;
    nodemax     = 60000;
    resolution  = 8;

    elementmax  = 30;

type
    range       = min..max;

    imagetype   = packed array [ range, range ]
                of boolean;

    treetype    = packed array [ 0..treemax ]
                of char;

    offsettype  = packed array [ 1..nodemax, 0..1 ]
                of range;

    extenttype  = record
                    Row, RowSize,
                    Col, ColSize  : integer;
                    DX, DY        : real;
                end;

var
    Image,
    OutDImage,
    OutIImage     : imagetype;

    DTree,
    ITree         : treetype;

    Extent        : extenttype;

    Offset        : offsettype;

    report        : text;

    RowCutoff,
    ColCutoff     : integer;

    obsd, obtgd,
    obtdd, obsi,
    obtgi, obtdi  : array[1..500] of integer;

    z             : integer;
```

```
procedure

GenerateDependentTree ( var image    : imagetype;
                        var Tree     : treetype  );

var
    twos      : array[1..resolution] of range;
    i,
    treecount,
    starttime : integer;



procedure BuildTree ( level,
                      quadrant,
                      row,
                      col      : integer );

procedure Condense;

var i : integer;

begin { Condense }

    i := treecount - 5;

    if ( ( Tree[i  ] = '(' ) and
         ( Tree[i+1] = 'W' ) and
         ( Tree[i+2] = 'W' ) and
         ( Tree[i+3] = 'W' ) and
         ( Tree[i+4] = 'W' ) )

       then begin
              Tree[i]   := 'W';
              treecount := treecount-4;
            end

    else

    if ( ( Tree[i  ] = '(' ) and
         ( Tree[i+1] = 'B' ) and
         ( Tree[i+2] = 'B' ) and
         ( Tree[i+3] = 'B' ) and
         ( Tree[i+4] = 'B' ) )

       then begin
              Tree[i]   := 'B';
              treecount := treecount-4;
            end;

end; { Condense }
```

```
begin { BuildTree }

    if quadrant = 0 then begin
                    Tree[treecount] := '(';
                    treecount       := treecount + 1;
                    end;

    if level < resolution then begin

        BuildTree ( level+1, 0,
            row              , col              );
        BuildTree ( level+1, 1,
            row              , col+twos[level+1] );
        BuildTree ( level+1, 2,
            row+twos[level+1], col              );
        BuildTree ( level+1, 3,
            row+twos[level+1], col+twos[level+1] );

    end; {if}

    if level = resolution then begin

        if image[row,col] then begin
                    Tree[treecount] := 'B';
                    treecount       := treecount+1;
                    end
                else begin
                    Tree[treecount] := 'W';
                    treecount       := treecount+1;
                    end;

    end; {if}

    if quadrant = 3 then Condense;

end;  { BuildTree }



begin { GenerateDependentTree }

    treecount := 0;

    twos[resolution] := 1;
    for i := resolution-1 downto 1 do
        twos[i] := twos[i+1] * 2;

    starttime := clock;

    BuildTree ( 0, 0, 0, 0 );
```

```
    Tree[treecount] := chr(3);   {ETX}

    obtgd[z] := clock - starttime;
    obsd [z] := 2*(treecount - 1);

end;  { GenerateDependentTree }
```

```
procedure

DisplayDependentTree ( var Tree     : treetype;
                       var outimage : imagetype);

var
    twos       : array [ 0..resolution ] of integer;
    i,
    treecount,
    starttime  : integer;


procedure DrawBlock ( row,
                      col,
                      size : integer );

var
    i, j             : integer;

begin { DrawBlock }

    for i := row to row+size-1 do
    for j := col to col+size-1 do

        outimage [ i,j ] := true;

end;  { DrawBlock }



procedure DrawTree ( level,
                     row,
                     col   : integer );

begin { DrawTree }

    if Tree[treecount] in ['B','W','('] then

        case Tree[treecount] of

            'B' : begin
                  DrawBlock ( row, col, twos[level] );
                  treecount := treecount+ 1;
                  end;

            'W' : treecount := treecount + 1;


            '(' : begin
                  treecount := treecount + 1;
                  DrawTree ( level+1, row,
```

```
                col             );
            DrawTree ( level+1, row,
                col+twos[level+1] );
            DrawTree ( level+1,
                row+twos[level+1],
                col             );
            DrawTree ( level+1,
                row+twos[level+1],
                col+twos[level+1] );
            end;

    end; {case}

end;  { DrawTree }


begin { DisplayDependentTree }

    twos[resolution] := 1;
    for i := resolution-1 downto 0 do
        twos[i] := twos[i+1]*2;

    starttime := clock;

    treecount := 1;

    DrawTree ( 0, 0, 0 );

    obtdd[z] := clock - starttime;

end;  { DisplayDependentTree }
```

```
procedure

GenerateIndependentTree ( var Image  : imagetype;
                          var Tree   : treetype;
                          var Extent : extenttype;
                          var Offset : offsettype );


var
    treecount,
    offsetcount,
    starttime   : integer;



procedure RemoveRP ( var Tree : treetype );

var
    T   : treetype;
    i,j : integer;

begin { RemoveRP }

    j := 1;

    for i := 1 to treecount do

        if Tree[i] <> ')' then begin
                               T[j] := Tree[i];
                               j := j + 1;
                               end;

    Tree := T;
    treecount := j-1;

end; { RemoveRP }



procedure Bound ( var Extent : extenttype );

label 100, 200, 300, 400;

var
    i, j,
    Upper, Lower, Left, Right : range;
    Flag : boolean;

begin { Bound }

    Flag  := false;
```

```
    Lower := 0;
    Upper := max;
    Left  := 0;
    Right := max;

    for i := 0 to max do
    for j := 0 to max do

        if Image[i,j] then begin
                           Lower := i;
                           Flag  := true;
                           goto 100;
                       end;

    100 : if Flag then begin

        for i := max downto 0 do
        for j := 0 to max do

            if Image[i,j] then begin
                               Upper := i+1;
                               goto 200;
                           end;

        200 : for j := 0 to max do
              for i := Lower to Upper do

                  if Image[i,j] then begin
                                     Left := j;
                                     goto 300;
                                     end;

        300 : for j := max downto 0 do
              for i := Lower to Upper do

                  if Image[i,j] then begin
                                     Right := j+1;
                                     goto 400;
                                     end;

    end;
    400 : Extent.Row     := Lower;
          Extent.Col     := Left;
          Extent.RowSize := Upper - Lower;
          Extent.ColSize := Right - Left;
          Extent.DX      := 1.0;
          Extent.DY      := 0.0;

end; { Bound }
```

```
procedure BuildTree ( Level,
                      Quadrant,
                      Row,
                      Col,
                      RowSize,
                      ColSize  : integer;
                      Adapt    : boolean );

type
    SizeType     = array [ 0..3, 0..1 ] of range;
    AreaType     = array [ 0..3 ] of integer;
    ValueType    = array [ 0..3 ] of boolean;
    QuadrantType = 0..3;

var

    Size         : SizeType;
    Area         : AreaType;
    Largest      : QuadrantType;
    BlackOrWhite : ValueType;
    i,t          : integer;
    Indicator    : char;
    RowInc,
    ColInc       : integer;



procedure HomogeneousBlock (    Row,
                                Col,
                                RowChange,
                                ColChange : integer;
                            var Area      : integer;
                            var AreaRow,
                                AreaCol   : range   );

var
    NewArea,
    CurrentRow,
    CurrentCol,
    RowLimit,
    ColLimit   : integer;

    Value      : boolean;

begin { HomogeneousBlock }

    Area     := 0;
    RowLimit := Row + (RowChange * RowSize);
    ColLimit := Col + (ColChange * ColSize);
    Value    := Image[Row,Col];
```

```
    CurrentRow := Row;
    while (CurrentRow <> RowLimit) do begin

        CurrentCol := Col;
        while (CurrentCol <> ColLimit) and
            (Value = Image[CurrentRow, CurrentCol])
            do
                CurrentCol := CurrentCol + ColChange;

        if CurrentCol <> ColLimit then
                    ColLimit := CurrentCol;

        NewArea := (abs(CurrentRow-Row)+1)
                * (abs(ColLimit-Col));

        if NewArea > Area then
                    begin
                    Area    := NewArea;
                    AreaRow := abs(CurrentRow-Row)+1;
                    AreaCol := abs(ColLimit-Col);
                    end;

        CurrentRow := CurrentRow + RowChange;

    end; { while }

end;  { HomogeneousBlock }


procedure Condense;

var
    State, i : integer;

begin { Condense }

    i := treecount - 2;

    State := 0;

    repeat

        case State of

            0 : begin
                if Tree[i] = 'W' then State := 1;
                if Tree[i] = 'B' then State := 2;
                end;

            1 : if Tree[i] = '(' then begin
```

```
        Tree[i] := 'W';
        treecount := i + 1;
        if Adapt then
          offsetcount := offsetcount - 1;
        State := 7;
        end
    else
        if Tree[i] = 'W' then State := 3
                         else State := 7;

2 : if Tree[i] = '(' then begin
        Tree[i] := 'B';
        treecount := i + 1;
        if Adapt then
          offsetcount := offsetcount - 1;
        State := 7;
        end
    else
        if Tree[i] = 'B' then State := 4
                         else State := 7;

3 : if Tree[i] = '(' then begin
        Tree[i] := 'W';
        treecount := i + 1;
        State := 7;
        end
    else
        if Tree[i] = 'W' then State := 5
                         else State := 7;

4 : if Tree[i] = '(' then begin
        Tree[i] := 'B';
        treecount := i + 1;
        State := 7;
        end
    else
        if Tree[i] = 'B' then State := 6
                         else State := 7;

5 : if Tree[i] = 'W' then begin
        Tree[i-1] := 'W';
        treecount := i;
        State := 7;
        end
    else
        State := 7;

6 : if Tree[i] = 'B' then begin
        Tree[i-1] := 'B';
        treecount := i;
        State := 7;
```

```
                end
            else
                State := 7;

        7 : null;

    end; {case}

    i := i - 1;

until State = 7;

end;  { Condense }


begin { BuildTree }

    if (RowSize <> 0) and (ColSize <> 0) then

    if (RowSize <= RowCutoff) and
       (ColSize <= ColCutoff) then begin

        Adapt := false;

        if (RowSize = 1) and (ColSize = 1) then

            if Image[Row,Col] then begin
                Tree[treecount] := 'B';
                treecount       := treecount + 1;
                end
            else begin
                Tree[treecount] := 'W';
                treecount       := treecount + 1;
            end

        else begin

            Tree[treecount] := '(';
            treecount       := treecount + 1;

            if odd(RowSize) then RowInc := 1
                            else RowInc := 0;

            if odd(ColSize) then ColInc := 1
                            else ColInc := 0;

            BuildTree ( Level+1, 0,
                        Row,
                        Col,
                        RowSize div 2 + RowInc,
```

```
                ColSize div 2 + ColInc,
                Adapt  );

        BuildTree ( Level+1, 1,
                Row,
                Col + ColSize div 2 + ColInc,
                RowSize div 2 + RowInc,
                ColSize div 2,
                Adapt  );

        BuildTree ( Level+1, 2,
                Row + RowSize div 2 + RowInc,
                Col,
                RowSize div 2,
                ColSize div 2 + ColInc,
                Adapt  );

        BuildTree ( Level+1, 3,
                Row + RowSize div 2 + RowInc,
                Col + ColSize div 2 + ColInc,
                RowSize div 2,
                ColSize div 2,
                Adapt  );

    end;

end

else begin

Tree[treecount] := '(';
treecount       := treecount + 1;

BlackOrWhite[0] :=
      Image[ Row,          Col          ];
BlackOrWhite[1] :=
      Image[ Row,          Col+ColSize-1 ];
BlackOrWhite[2] :=
      Image[ Row+RowSize-1, Col          ];
BlackOrWhite[3] :=
      Image[ Row+RowSize-1, Col+ColSize-1 ];

HomogeneousBlock ( Row,          Col,
         1,  1, Area[0], Size[0,0], Size[0,1] );

HomogeneousBlock ( Row,          Col+ColSize-1,
         1, -1, Area[1], Size[1,0], Size[1,1] );

HomogeneousBlock ( Row+RowSize-1, Col,
        -1,  1, Area[2], Size[2,0], Size[2,1] );
```

```
HomogeneousBlock ( Row+RowSize-1, Col+ColSize-1,
        -1, -1, Area[3], Size[3,0], Size[3,1] );

Largest := 0;
for i := 1 to 3 do
    if Area[Largest] < Area[i] then Largest := i;

if BlackOrWhite[Largest] then Indicator := 'B'
                         else Indicator := 'W';

case Largest of

    0 : begin  Offset[offsetcount,0] := Size[0,0];
               Offset[offsetcount,1] := Size[0,1];
         end;
    1 : begin  Offset[offsetcount,0] := Size[1,0];
               Offset[offsetcount,1] :=
                   ColSize - Size[1,1];
         end;
    2 : begin  Offset[offsetcount,0] :=
                   RowSize - Size[2,0];
               Offset[offsetcount,1] := Size[2,1];
         end;
    3 : begin  Offset[offsetcount,0] :=
                   RowSize - Size[3,0];
               Offset[offsetcount,1] :=
                   ColSize - Size[3,1];
         end;

end; {case}

offsetcount := offsetcount + 1;

t := offsetcount - 1;

if Largest = 0 then begin
                    Tree[treecount] := Indicator;
                    treecount       := treecount + 1;
                    end
else
    BuildTree ( Level+1, 0,
                Row,
                Col,
                Offset[t,0],
                Offset[t,1],
                Adapt  );

if Largest = 1 then begin
                    Tree[treecount] := Indicator;
                    treecount       := treecount + 1;
                    end
```

```
    else
        BuildTree ( Level+1, 1,
                    Row,
                    Col + Offset[t,1],
                    Offset[t,0],
                    ColSize - Offset[t,1],
                    Adapt  );

    if Largest = 2 then begin
                      Tree[treecount] := Indicator;
                      treecount       := treecount + 1;
                      end
    else
        BuildTree ( Level+1, 2,
                    Row + Offset[t,0],
                    Col,
                    RowSize - Offset[t,0],
                    Offset[t,1],
                    Adapt  );

    if Largest = 3 then begin
                      Tree[treecount] := Indicator;
                      treecount       := treecount + 1;
                      end
    else
        BuildTree ( Level+1, 3,
                    Row + Offset[t,0],
                    Col + Offset[t,1],
                    RowSize - Offset[t,0],
                    ColSize - Offset[t,1],
                    Adapt  );

    end; {if size not 0}

    if Quadrant=3 then begin
                    Tree[treecount] := ')';
                    treecount := treecount + 1;
                    Condense;
                    end;

end;  { BuildTree }


begin { GenerateIndependentTree }

    treecount   := 1;
    offsetcount := 1;

    starttime := clock;
```

```
    Bound ( Extent );

    BuildTree ( 0, 0, Extent.Row,
                  Extent.Col,
                  Extent.RowSize,
                  Extent.ColSize,
                  true  );

    Tree[treecount]       := chr(3);  {ETX}
    Offset[offsetcount,0] := -1;

    RemoveRP ( Tree );

    obtgi[z] := clock - starttime;
    obsi [z] := (2*(treecount - 1))
           + (16*(offsetcount-1)) + 128;

end;  { GenerateIndependentTree }
```

```
procedure

DisplayIndependentTree ( var Tree     : treetype;
                         var Extent   : extenttype;
                         var Offset   : offsettype;
                         var OutImage : imagetype  );

var
    XLeft, XRight, YBottom, YTop : integer;

    starttime : integer;

    treecount,
    offsetcount : integer;

procedure DrawBlock ( Row, Col,
                      RowSize, ColSize : integer;
                      DX, DY : real );

type
    numtype = array [ 0..3 ] of integer;
var
    X, Y           : numtype;

    i, j,
    Lower, Upper,
    Left, Right,
    Temp, ScanLine  : integer;

    PerDX, PerDY    : real;



procedure DetermineBounds (     Y     : numtype;
                            var Lower,
                                Upper : integer );

begin { DetermineBounds }

    if Y[0] < YBottom then Lower := YBottom
                      else Lower := Y[0];

    if Y[3] > YTop    then Upper := YTop
                      else Upper := Y[3];

end;  { DetermineBounds }


procedure Validate ( var Left, Right : integer );
```

```
var
    Temp : integer;

begin  { Validate }

    if Left > Right then begin
        Temp  := Left;
        Left  := Right;
        Right := Temp;
        end;

    if Left < XLeft then Left := XLeft;
    if Left > XRight then Left := XRight+1;

    if Right < XLeft then Left := XLeft-1;
    if Right > XRight then Right := XRight;

end;  { Validate }



procedure WhichTwoLines (     X, Y     : numtype;
                          ScanLine : integer;
                      var Left,
                          Right    : integer );

var
    M1, M2 : real;

begin { WhichTwoLines }

    M1 := (Y[0]-Y[1])/(X[0]-X[1]);
    M2 := (Y[0]-Y[2])/(X[0]-X[2]);

    if ((ScanLine>=Y[0]) and (ScanLine<=Y[2])) then
        Left := round ( ((ScanLine-Y[0])/M2)+X[0] )
    else
        Left := round ( ((ScanLine-Y[2])/M1)+X[2] );

    if ((ScanLine>=Y[0]) and (ScanLine<=Y[1])) then
        Right := round ( ((ScanLine-Y[0])/M1)+X[0] )
    else
        Right := round ( ((ScanLine-Y[3])/M2)+X[3] );

end;  { WhichTwoLines }



procedure FillSquare;

var
```

```
    i,j,
    ColLeft, ColRight,
    RowLower, RowUpper  : integer;

begin { FillSquare }

    if X[0] > X[1] then begin
                    ColLeft := X[1];
                    ColRight := X[0];
                    end
                else begin
                    ColLeft := X[0];
                    ColRight := X[1];
                    end;

    if ColLeft  < XLeft  then ColLeft  := XLeft;
    if ColRight > XRight then ColRight := XRight+1;

    if Y[0] < YBottom then RowLower := YBottom
                    else RowLower := Y[0];
    if Y[2] > YTop    then RowUpper := YTop+1
                    else RowUpper := Y[2];

    for i := RowLower to RowUpper-1 do
    for j := ColLeft  to ColRight-1 do
        OutImage[i,j] := true;

end;  { FillSquare }


function SignedUnit ( Indicator : real ) : real;

begin { SignedUnit }

    if (Indicator < 0.0) and
      (Indicator > -1.0) then SignedUnit := -1

    else
    if (Indicator > 0.0) and
      (Indicator <  1.0) then SignedUnit :=  1

    else
    { Indicator = 0.0 or -1.0 or 1.0 }
            SignedUnit :=  0;

end;  { SignedUnit }


begin { DrawBlock }
```

```
X[0] := Col;
Y[0] := Row;

X[1] := X[0]
      + round(ColSize * DX + SignedUnit(DX) );
Y[1] := Y[0]
      + round(ColSize * DY + SignedUnit(DY) );

PerDX := -DY;
PerDY :=  DX;

X[2] := X[0] +
round(RowSize * PerDX + SignedUnit(PerDX) );
Y[2] := Y[0] +
round(RowSize * PerDY + SignedUnit(PerDY) );

X[3] := X[2] +
round(ColSize * DX + SignedUnit(DX) );
Y[3] := Y[2] +
round(ColSize * DY + SignedUnit(DY) );

for i := 0 to 2 do
for j := i+1 to 3 do
    if Y[j] < Y[i] then begin

        Temp := X[j];
        X[j] := X[i];
        X[i] := Temp;
        Temp := Y[j];
        Y[j] := Y[i];
        Y[i] := Temp;
        end;

if ((X[0]=X[2]) and (X[1]=X[3])) or
   ((X[0]=X[3]) and (X[1]=X[2])) then
    FillSquare


else begin

    DetermineBounds ( Y, Lower, Upper );

    for ScanLine := Lower to Upper do begin

    WhichTwoLines ( X, Y, ScanLine,
                Left, Right );

            Validate ( Left, Right );

            for i := Left to Right-1 do
```

```
                OutImage[ScanLine,i] := true;

            end; {for}

    end; {else}

end;  { DrawBlock }


procedure DrawTree ( Row, Col,
                RowSize, ColSize : integer;
                DX, DY : real );

var
    RowOffset, ColOffset : integer;
    R0, R1, R2, R3,
    C0, C1, C2, C3       : integer;
    PerDX, PerDY          : real;
    RowInc, ColInc        : integer;

begin { DrawTree }

    if ((RowSize<>0) and (ColSize<>0)) then

    if (RowSize <= RowCutoff) and
       (ColSize <= ColCutoff) then begin

        if Tree[treecount] in ['B','W','('] then

        case Tree[treecount] of
            'B' : begin
                DrawBlock ( Row, Col,
                   RowSize, ColSize, DX, DY );
                treecount := treecount + 1;
                end;

            'W' : treecount := treecount + 1;

            '(' : begin
                treecount := treecount + 1;

                if odd(RowSize) then RowInc := 1
                            else RowInc := 0;

                if odd(ColSize) then ColInc := 1
                            else ColInc := 0;

                DrawTree ( Row,
                        Col,
                        RowSize div 2 + RowInc,
```

```
                    ColSize div 2 + ColInc,
                    DX, DY );

            DrawTree ( Row,
                    Col + ColSize div 2
                    + ColInc,
                    RowSize div 2 + RowInc,
                    ColSize div 2,
                    DX, DY );

            DrawTree ( Row + RowSize div 2
                    + RowInc,
                    Col,
                    RowSize div 2,
                    ColSize div 2 + ColInc,
                    DX, DY );

            DrawTree ( Row + RowSize div 2
                    + RowInc,
                    Col + ColSize div 2
                    + ColInc,
                    RowSize div 2,
                    ColSize div 2,
                    DX, DY );
          end;
    end; {case}
end


else begin

if Tree[treecount] in ['B', 'W', '(' ] then

    case Tree[treecount] of

      'B' : begin
          DrawBlock ( Row, Col, RowSize,
                  ColSize, DX, DY );
          treecount := treecount + 1;
          end;

      'W' : treecount := treecount + 1;

      '(' : begin
          treecount   := treecount + 1;
          offsetcount := offsetcount + 1;
          RowOffset   := Offset[offsetcount,0];
          ColOffset   := Offset[offsetcount,1];

          C0 := Col;
```

```
                R0 := Row;

                C1 := C0 + round(ColOffset * DX );
                R1 := R0 + round(ColOffset * DY );

                PerDX := -DY;
                PerDY :=  DX;

                C2 := C0 + round(RowOffset * PerDX );
                R2 := R0 + round(RowOffset * PerDY );

                C3 := C2 + round(ColOffset * DX );
                R3 := R2 + round(ColOffset * DY );

                DrawTree ( R0,     C0,
                           RowOffset, ColOffset,
                           DX, DY );

                DrawTree ( R1,     C1,
                           RowOffset, ColSize-ColOffset,
                           DX, DY );

                DrawTree ( R2,     C2,
                           RowSize-RowOffset, ColOffset,
                           DX, DY );

                DrawTree ( R3,     C3,
                           RowSize-RowOffset,
                           ColSize-ColOffset,
                           DX, DY );
                end;

            end; {case}

        end; {size check}

        if Tree[treecount] = ')' then
          treecount := treecount + 1;

end;  { DrawTree }



begin { DisplayIndependentTree }

        XLeft   := 0;  XRight := max;
        YBottom := 0;  YTop   := max;

        starttime := clock;
```

```
    treecount   := 1;
    offsetcount := 0;

    DrawTree ( Extent.Row, Extent.Col,
               Extent.RowSize, Extent.ColSize,
               Extent.DX, Extent.DY
             );

    obtdi[z] := clock - starttime;

end; { DisplayIndependentTree }
```

BIBLIOGRAPHY

Books

Foley, J. D. and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA., 1982.

Hoel, Paul G., *Introduction to Mathematical Statistics (Fourth Edition)*, John Wiley & Sons, Inc., New York, 1971.

Horowitz, Ellis and Sahni, Sartaj, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD., 1978

Pavlidis, Theo, *Algorithms for Graphics and Image Processing*, Computer Science Press, Rockville, MD, 1982.

Articles

Dyer, Charles R., Rosenfeld, Azriel, and Samet, Hanan, "Region Representation: Boundary Codes from Quadtrees," *Communications of the ACM*, vol. 23, pp. 171-179, March 1980.

Gargantini, Irene, "An Effective Way to Represent Quadtrees," *Communications of the ACM*, vol. 25, pp. 905-910, December 1982.

Hunter, G. M. and Steiglitz, K., "Linear Transformations of Pictures Represented by Quad Trees," *Computer Graphics and Image Processing*, vol. 10, pp. 289-296, 1979.

Hunter, Gregory M., and Steiglitz, Kenneth, "Operations on Images Using Quad Trees," *IEEE Transactions of Pattern Analysis and Machine Intelligence*, vol. 1, pp. 145-153, April 1979.

Jackins, C. L. and Tanimoto, S.L., "Decompostion of Euclidean Space," *IEEE Transactions of Pattern Analysis and Machine Intelligence*, vol. 5, pp. 533-539, Sept. 1983.

Kawaguchi, Eiji and Endo, Tsutomu, "On a Method of Binary-Picture Representation and Its Application to Data Compression," *IEEE Transactions of Pattern Analysis and Machine Intelligence*, vol. 2, pp. 27-35, January 1980.

Li, Ming, William I. Grosky, and Ramesh Jain, "Normalized Quadtrees with Respect to Translations," *Computer Graphics and Image Processing*, vol. 20, pp. 72-81, 1982.

Oliver, M. A. and N. E. Wiseman, "Operations on Quadtree Encoded Images," *The Computer Journal*, vol. 26, pp. 83-91, 1983.

Samet, Hanan, "An Algorithm for Converting Rasters to Quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 3, pp. 93-95, January 1981.

_____, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, vol. 16, pp. 187-260, June 1984.

_____, "Region Representation: Quadtrees from Binary Arrays," *Computer Graphics and Image Processing*, vol. 13, pp. 88-93, 1980.

_____, "Region Representation: Quadtrees from Boundary Codes," *Communications of the ACM*, vol. 23, pp. 163-170, March 1980.

Tamminen, Markku, "Comment on Quad- and Octtrees," *Communications of the ACM*, vol. 27, pp. 248-249, March 1984.

van Lierop, Marloes L. P., "Geometrical Transformations on Pictures Represented by Leafcodes," *Computer Vision, Graphics, and Image Processing*, vol. 33, pp. 81-98, 1986.