1988

The Representation of symmetric patterns using the quadtree data structure

Lucia Lodolini

Rochester Institute of Technology
School of Computer Science and Technology

The Representation of Symmetric Patterns Using the Quadtree Data Structure

by
Lucia Lodolini

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Professor Peter G. Anderson

Professor Guy Johnsen

Professor Chris Comte

May 9, 1988

ABSTRACT

Hierarchical data structures for image representation have been widely explored in recent years. These data structures are based on the principle of "recursive decomposition" of an image region. The most commonly mentioned picture data structure for two-dimensional data is referred to as a "quadtree".

The purpose of this thesis is to investigate the use of a general quadtree scheme as a means of representing symmetric images. Specifically, images are generated according to the rules of selected two-dimensional plane symmetry groups.

Keywords and Phrases

Quadtrees, recursive decomposition, hierarchical data structures, plane symmetry classes

Table of Contents

2. INTRODUCTION AND BACKGROUND

The idea for this thesis stemmed from a dual interest in the use of hierarchical data structures to represent image data, and in the generation of patterns belonging to the seventeen classes of two-dimensional plane symmetry. It is an attempt to join two areas that appear to have some relationship; the recursive subdivision of two-dimensional image areas, and the representation of patterns that are characterized by a single tiling propagated across the plane. It appears that there is some benefit in using a hierarchically-based image scheme to represent symmetric patterns.

Image data represented hierarchically is based on the idea of recursively dividing a picture region into smaller and smaller regions according to some criterion until further subdivision is not possible, necessary, or desired. This concept is known as "recursive decomposition" [SAME84b]. A class of hierarchical data structure known as a "quadtree" has generally been used to apply this concept in the area of image processing.

Hierarchical representation of image data has been widely explored in recent years for several reasons.

It is a means of focusing on areas of interest in a picture, while being able to discard those regions that are not of interest. In other words, any geographical part of the image may be accessed rapidly. This ability to focus can lead to improved execution times for picture operations, as compared to some schemes [SAME84b].

The tendency for most images to contain homogeneous areas is exploited by the ability to relate these areas in a systematic way, and by

the ability to compress the storage of homogeneous areas. Other common image representations may be lacking in one or both of these capabilities; although a run-length encoding or chain code provides image compression, such local representations do not inherently contain information relating to the global structure of the image, making certain image operations difficult. The popular binary array representation provides neither global structure information or compression.

Due to its hierarchical nature, numerous operations can be performed on an image using well-defined recursive algorithms, in the form of tree traversals. It is true, however, that many operations can be performed as well or better in other schemes [SAME84b].

In general, a hierarchical approach to image representation is a conceptually clear way of thinking of an image:

> "A characteristic of the human picture analysis process is a hierarchical type of functioning. People first group data in perceptual levels; these are then further grouped into subdivisions. Experiments conducted...on the ways we describe aerial terrain photographs - or, in general, complex pictures — concluded that (1) if a picture has a central theme, we respond to it and relate other portions to it and (2) if a picture has no central theme, but is totally diffuse, we partition the picture into quadrants ("upper right,", "upper left," etc.) and either describe each region separately or the objects within the quadrants and their relationships." [ALEX and KLIN-78]

It may be useful at this point to describe some of the applications currently implemented using hierarchical image data structures.

As was previously mentioned, a characteristic of hierarchical image schemes is an ability to concentate on areas of relevance in a picture. This trait makes hierarchical image techniques natural for applications where set operations are desired. An application described in some detail in [SAME84b] is a cartographic data base, consisting

of a number of different types of maps represented hierarchically. Sample queries on the data base are put forth, whose solutions involve an intersection operation on two maps. A typical query might involve finding all regions where corn is grown that are within 400 and 600-foot elevation levels. An efficient solution is possible by intersecting a contour map, divided into 50-foot intervals, with a land use map classifying areas according to crop growth. The global nature of the data structure makes it possible to eliminate from consideration those areas where wheat is grown. (A general algorithm describing the intersection of two images represented as quadtrees can be found in the section 2.2.1.)

Other areas where a hierarchical approach to image representation has been used are hidden surface elimination, robotics, animation, and expert vision systems [SAME84b].

It should be noted that the idea of recursive image decomposition extends to three dimensions with a data structure known as an octree [MEAG82], used chiefly in the decomposition of volume data.

2.1. Two-dimensional Image Decomposition

It would be beneficial to precede the discussion of specific hierarchical data structures with a brief background of two-dimensional grid decomposition (refinement).

A "tiling" of a two-dimensional plane refers to a set of tiles that cover the plane without gaps or overlaps. Such patterns have been a part of human activity since prehistoric times.

In particular, tilings by regular polygons were the first to be the

subject of mathematical research (Johannes Kepler first investigated such tilings more than 350 years ago.) [GRUN and SHEP-77]

Stated more formally, if K denotes the number of sides of a cell in a partition, and V is the number of cells meeting at a vertex, KV regular tesselations describe those where the value of K(V) is the same for all cells (vertices). All cells are regular polygons [AHUJ83].

Of particular interest in the area of image processing is a subset of this class of tilings, or, those tilings formed by congruent copies of a single regular polygon, shown in Figure 2.A. There are only three possible KV regular tilings that satisfy this restriction, as stated in [GRUN and SHEP-77]: "The only possible edge-to-edge tilings of the plane by mutually congruent regular convex polygons are the three regular tilings by equilateral triangles, by squares, or by regular hexagons."

In K(V) notation, these three are known as 3(6) (regular triangle), 4(4), (square), and 6(3), (hexagonal).

[AJUH83] lists the two properties desirable for any planar decomposition scheme for image representation:

a. The pattern used to partition the plane should repeat infinitely.
b. It should be possible to decompose the partition into smaller and smaller versions of the same pattern.

Another way to describe the second property is to say that the ability must exist to partition each cell further into smaller cells such that the new tesselation is still a KV regular tesselation having the same K(V) values [AJUH83].

All three of the tesselations in Figure 2.A satisfy the first property, but only the square and triangular scheme satisfy the second property, as a hexagonal grid is not decomposable into smaller regular

hexagons [AHUJ83]. The failure of the second property for hexagonal tesselations makes it less attractive for image processing applications, in the sense that the smallest resolution for the image must be predetermined. Tilings that satisfy the second property are called "unlimited"; the hexagonal grid example is therefore a "limited" tiling [SAME84b].

The two "most appropriate" [AJUH83] configurations for image subdivision, square and triangular tilings, differ in terms of adjacency and orientation. Several definitions relating to these properties are given in [BELL83, et al.]:

Two tiles are said to be neighbors if they are adjacent either along an edge, or a vertex. A tiling is uniformly adjacent if the distances between the centroid of one tile and the centroid of all its neighbors are the same. The adjacency number of a tiling is the number of different intercentroid distances between any one tile and its neighbors. The square tiling has two adjacency distances, where there are three for the triangular tiling. Only the hexagonal tiling is said to have uniform adjacency.

Tiles with the same orientation can be mapped onto each other by a translation that does not involve rotation or reflection. A decomposition based on squares is said to have "uniform orientation", i.e. all four regions have the same orientation. This is not the case in the triangular scheme; there is one node out of four where the orientation differs from the remaining three siblings [AHUJ83].

Issues relating to two-dimensional plane decomposition as applied to image representation are discussed in greater detail in [AHUJ83] and [BELL83, et al.].

For our purposes, the main focus will be on planar decomposition

based on a square grid, which is discussed in detail in the next section. Triangular decomposition will be described to a lesser extent.

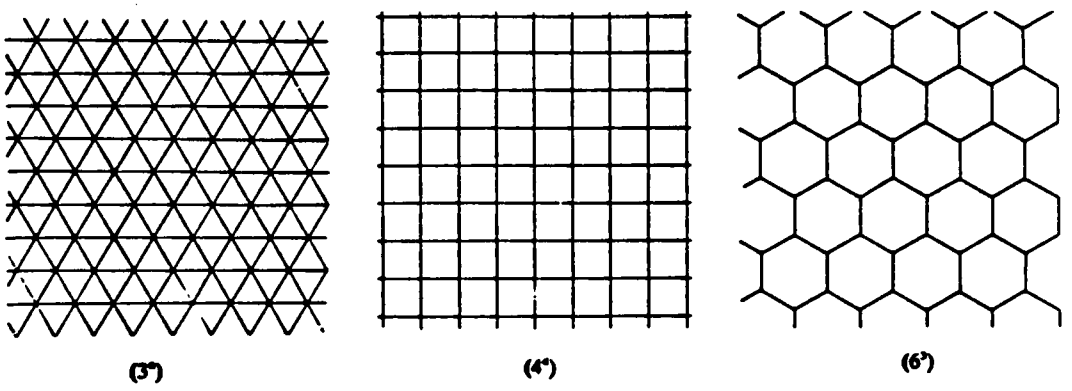


Figure 2.A. Three regular decompositions (square, triangular, hexagonal) [GRUN and SHEP, 77]

2.2. Quadtree Data Structure

The term quadtree is used to describe a class of hierarchical data structures. In image processing, the motivating principle underlying such a representation is the "recursive decomposition" of two-dimensional space, previously discussed.

A general quadtree scheme may be described in the following way:

Given an image region of size 2**n by 2**n, it is possible to recursively divide the region into areas of one-half the previous size, continuing this until some condition is reached that will signal termination of the process. The condition we will use is arrival at the level of individual BLACK or WHITE pixels, although it is possible to use other image attributes for this purpose. An attribute commonly mentioned is the average gray scale information in a picture.

In a quadtree representation, each tree node corresponds to a region of the image, generally square, with the root of the tree representing the entire picture. Any tree node will have either four descendents labelled NW, NE, SW, SE, which represent four ordered quadrants within the parent region, or no descendents (a terminal node, or leaf). A node value of GRAY is used to designate a non-homogeneous region, or, a region that requires further refinement. BLACK or WHITE leaves represent a homogeneous region of the picture that need (can) not be subdivided further. This general scheme is shown in Figure 2.B. The quadtree root will reside at level n of a quadtree used to represent a 2**n by 2**n region, its children at level n-1, and so on until level 0,

which designates the level of individual pixels. A node represents a region of size 2**l if it is found at level l in the tree, i.e. the region is of side length 2**l.

A brief overview of this method as applied to triangular areas is now given, as presented in [AHUJ83].

To recursively decompose an image into equilateral triangles, each triangle is divided into four others. The method of division is shown in Figure 2.C. The data structure used to express this can be a referred to as a quadtree, as each node corresponding to a triangular region has four children. Each triangular node has three neighbors, and each triangle has one of two orientations, differing by 60 degrees. In this scheme, the triangle in the center differs in orientation from the parent node. Its three siblings have the same orientation as the parent. The basic structure of the algorithms using square or triangular tilings may be the same since both involve quadtrees. The complexity of image operations in both types of quadtree also appears to be the same, as shown in [AHUJ83]. In general, square images will be most compactly described as square quadtrees, while triangular images are most compactly described as triangular quadtrees.

In addition to the equilateral triangle-based subdivision, [ELCO86, et al.] have described other triangular decompositions refered to as scalene (ternary), scalene (hexary), foveal (triangular) and foveal (square).

An application common for triangular decompositions is in the field of cartography; such triangular structures have been useful for fitting irregular terrains in a map. The most-used approximations to the earth's

spherical shape are the dodecahedron and icosahedron, which both require triangular decomposition [ELCO86, et al.]. In [YAMA84, et al.], a triangular subdivision is used to generate an isometric view of a three-dimensional object represented as an octtree. Such an application is frequently useful in the field of engineering.

The quadtree data structure described thus far unfortunately requires much overhead. Given a quadtree that contains B + W (BLACK + WHITE) tree nodes, an extra (B + W - 1) / 3 nodes are required to store its internal GRAY nodes [SAME84b]. Furthermore, a substantial majority of the storage is used to accommodate the pointers to a node's children. A great deal of attention has therefore been focused on schemes that will improve storage requirements, mostly by reducing or eliminating the need for pointers. Several of these schemes are described in Section 6.

From this point on, we will assume a square decompositon method as the basis for discussion.

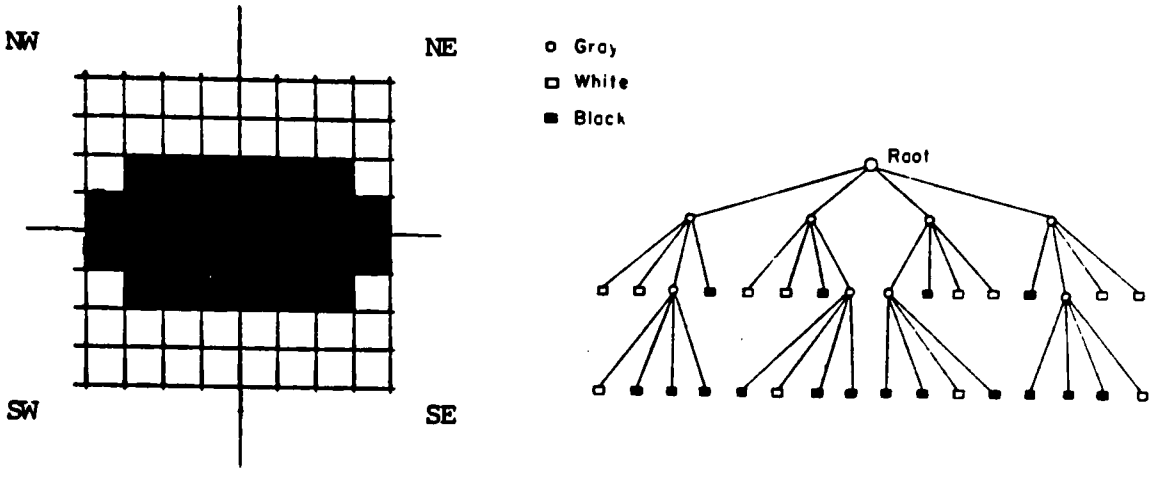


Figure 2.B. An 8x8 pixel image and its quadtree representation [CHIE and AGGA, 84]

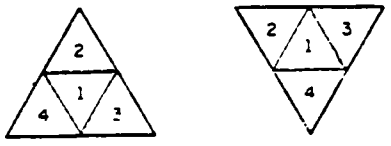
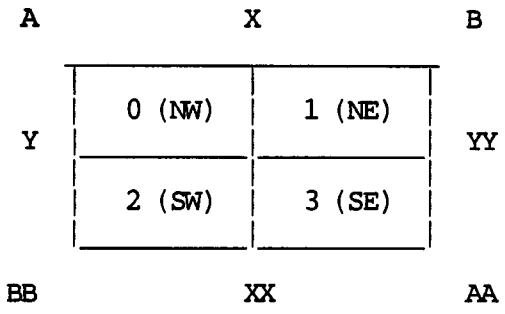


Figure 2.C. Example of regions in a triangular grid decomposition [AHUJ83]

2.1.1. Quadtree Operations

A natural by-product of the quadtree data structure is the ability to perform numerous image operations in the form of tree traversals. The computation performed at the node is generally the distinguishing factor between operations, although one operation may be modified by changing the traversal order of the nodes. The raster output of a quadtree-encoded image is an example of this. (Pseudocode for a quadtree to raster algorithm appears in Section 2.2.3.3.) The drawing and table below show the possible transformations of an image achieved by changing only the order in which quadtree nodes are traversed before output [STEW86]. (This will be the basis for the generation of patterns according to selected plane symmetry classes in Section 3.)



OPERATION PERFORMED (No operation)	QUADRANT ORDER 0	1	2	3
90 degree rotation clockwise	2	0	3	1
180 degree rotation	3	2	1	0
270 degree rotation clockwise	1	3	0	2
Reflection about X XX	1	0	3	2
Reflection about Y YY	2	3	0	1
Reflection about A AA	0	2	1	3
Reflection about B BB	3	1	2	0

Several quadtree algorithms are now described to provide examples of the type of recursive operations typical to this data structure.

Set Operations for Quadtrees [SAME84b]

As was mentioned in Section 2, the quadtree is particularly suited to perform set operations on images. The UNION operation on two images represented as quadtrees can be described algorithmically as follows [SAME84b].

Input consists of two quadtrees representing images of the same size, A and B.
Output will be a third quadtree U, containing the union of trees A and B.
Both input trees are traversed in parallel.
If node A or node B is BLACK, the resulting node in U is BLACK.
If node A and node B is GRAY, the resulting node in U is GRAY, and recurse to the children of the nodes. In this case, it is necessary to check for a merge situation in the output tree, i.e. four BLACK sibling nodes that have been created from the union of the two subtrees.
If one node is GRAY and the other WHITE, the result is the GRAY node and its subtree.

The INTERSECTION algorithm is analogous to that of UNION, where the attribute WHITE is substituted for each occurrence of attribute BLACK.

Both algorithms have an execution time proportional to the minimum number of nodes at corresponding levels of the two input trees. If construction of the output tree is included in the algorithm, the time becomes proportional to the total number of nodes in the two trees [SAME84b].

Superposition of Two Quadtrees [HUNT and STEI-79a]

The purpose of this algorithm is to output an image represented by TREEUPPER over that of TREELOWER.

Two quadtrees representing images of the same size (TREE_UPPER, TREE_LOWER) are input. Output is a tree consisting of the superposition of TREE_UPPER over TREE_LOWER.
Traverse the two input trees in parallel.
When a leaf is visited in either tree, do one of three things:

CASE 1: BOTH NODES ARE LEAVES.
If leaf of TREE_UPPER is transparent, do nothing.
If leaf of TREE_UPPER is opaque, replace the leaf TREE_LOWER with that of TREE_UPPER

CASE 2: LEAF IN TREE_UPPER, PARENT IN TREE_LOWER.
If leaf of TREE_UPPER is transparent, do nothing.
If leaf of TREE_UPPER is opaque, replace TREE_LOWER node (and therefore all descendents) with TREE_UPPER. Continue as though the descendents of TREE_LOWER have already been processed.

CASE 3: PARENT IN TREE_UPPER, LEAF IN TREE_LOWER.
Replace leaf of TREE_LOWER with parent of TREE_UPPER.
Replace subtree in TREE_UPPER with "transparent leaves".
After traversing the subtree, continue parallel traversal of two trees.

Image Approximation Using Quadtrees [SAME84b]

The "inner" or "outer" approximation of an image can be displayed as follows. Beginning at a selected level of the tree, output all GRAY nodes as WHITE (inner approximation) or BLACK (outer approximation). This will result in a very rough picture approximation.

A method that will yield a more subtle approximation of the image is referred to as quadtree truncation [SAME84b]. Beginning at a certain level in the tree, treat GRAY nodes as BLACK or WHITE, depending on the type of the majority of the node's descendents.

2.2.2. Space Considerations

Dyer [DYER82] has analyzed the space efficiency of quadtrees. By moving a square of size 2**m by 2**m to all possible positions within a 2**n by 2**n grid, he has produced the following table of results:

NODE TYPE	BEST CASE	WORST CASE	AVERAGE CASE
BLACK	1	3(2**m+1 - m) - 5	O(2**m+2 - m)
WHITE	3(n - m)	3(2**m+1 + 4n - 3m - 5)	O(2**m+2 + n - m)
GRAY	n - m	2**m+2 + 4(n - m) - 7	O(2**m+2 + n - m)
TOTAL	4(n - m) + 1	2**m+4 + 16(n - m) - 27	O(2**m+2 + n - m)

O(2**m+2 + n - m) represents space efficiency on the order of the region's perimeter plus the log of the image diameter.

It should be noted that square images are particularly well-suited to a square quadtree representation. While this is not true of all images, the results of the experimentation nonetheless imply that any image would display similar behavior in terms of average space efficiency [DYER82,p.347].

In terms of space efficiency, a worst-case situation occurs when the image consists of a checkerboard pattern of pixels. Such a tree would require the maximum number of nodes (4 x [N**2 - 1] / 3, where N is a power of two) since there are no homogeneous areas to compress above the level of individual pixels. See Figure 2.D. A best case situation would be the trivial case of a solid BLACK or WHITE square, which would be represented by a single node. Space requirements for the quadtree can be said to be a function of the resolution (number of recursion levels) of

the image. As the resolution is doubled, the quadtree grows linearly in the number of nodes. The same doubling of the resolution in a binary array representation will quadruple the number of pixels [SAME84b].

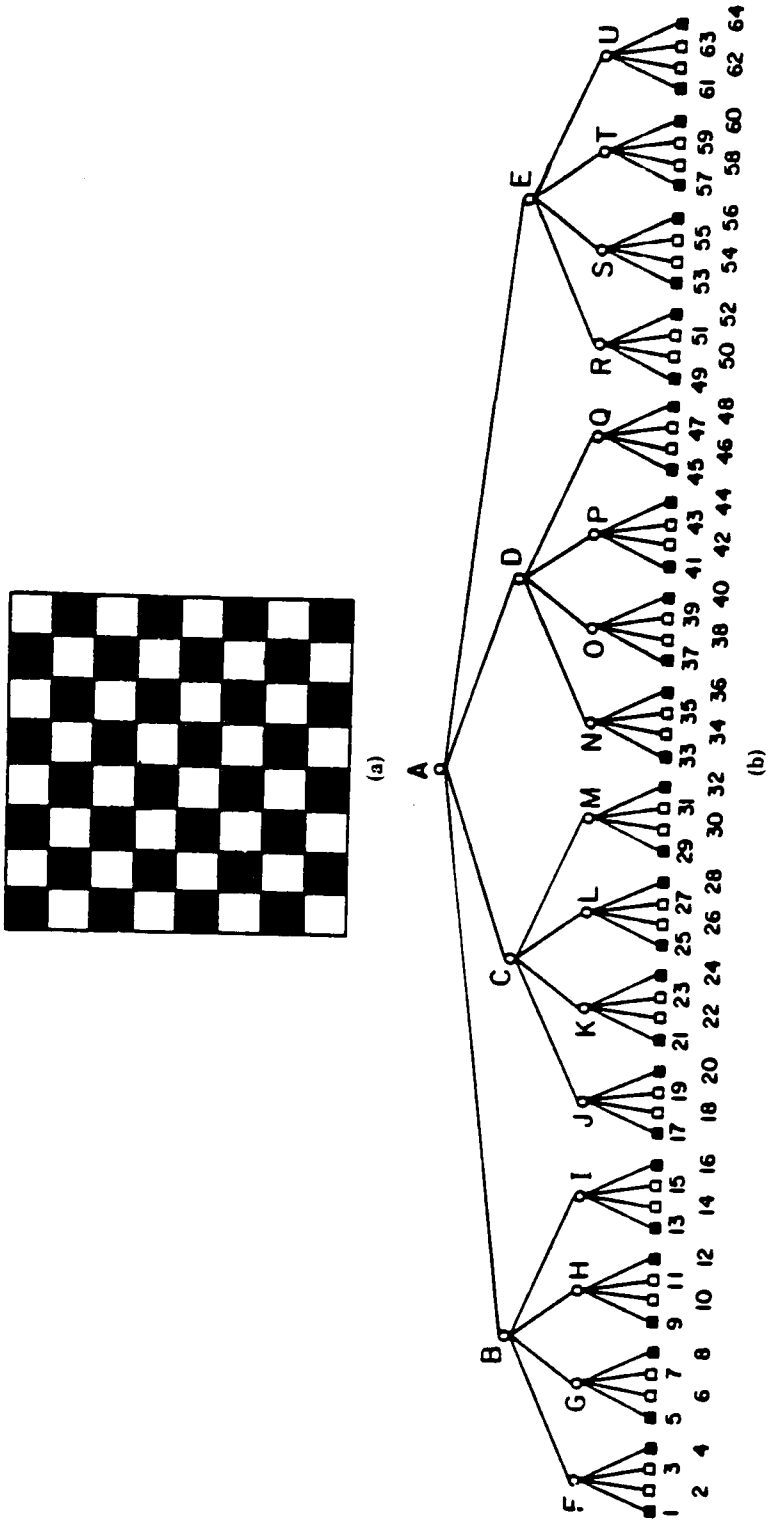


Figure 2.D. A checkerboard pattern and its quadtree, representing worst-case quadtree space efficiency [SAME84b]

2.2.3. Conversion Algorithms

Most digital images are traditionally represented as binary arrays, rasters (run-lengths), boundaries (chain codes), or polygons (vectors), with the binary array as probably the most common representation [SAME84b]. Techniques are needed to switch between the various representations and quadtrees easily. Below, algorithms describing conversion from binary arrays and raster representations to quadtrees are given. A chain code to quadtree conversion can be found in [SAME80a], and issues relating to the representation of vector images as quadtrees are discussed in [HUNT78], [HUNT and STEI-79a], and [HUNT and STEI-79b].

We will initially need a way to encode an image as a quadtree, and to output this image after it is in quadtree form. In both processes, the data structure of a tree node consists of six data fields. There is a pointer to the parent of a node, four ordered pointers to its descendents (labelled NW, NE, SW, SE), and a NODETYPE field with possible values of BLACK, WHITE or GRAY. The parent of the root node is set to NULL.

2.2.3.1. Binary Array to Quadtree Conversion [SAME80]

This algorithm assumes the presence of the entire image in a binary image array. Each pixel is visited only once by the algorithm, in a manner analogous to a postorder traversal. The order in which the pixels are visited is shown in Figure 2.E.

Although a simple way of accomplishing this conversion would be to create every node in the tree, and perform the merging of redundant

nodes in a second pass, space requirements could easily become excessive for the amount of available memory. Therefore, a quadtree node is created only in a situation where the corresponding region of the image is maximal; in other words, the algorithm will never create four nodes for children of the same NODETYPE; only the parent node representing the area will be created in this situation.

The driver routine is invoked with the highest level of the tree (level n for a 2**n by 2**n image region), and a global image array. If the entire image is BLACK or WHITE, the procedure creates a single node tree and finishes. If this is not the case, control is passed to a procedure designed to construct the tree. The procedure examines every pixel and creates nodes whenever all four children are not of the same type.

Pseudocode for the binary array to quadtree algorithm is now given, as presented in [SAME80]. Some variable names have been changed for clarity's sake.

```
node procedure QUADTREE (LEVEL);
/* Find the quadtree corresponding to a 2**LEVEL by 2**LEVEL
   binary array B_ARRAY */
begin
  integer value LEVEL;
  /* B_ARRAY is global */
  global Boolean array B_ARRAY[1:2**LEVEL, 1:2**LEVEL];
  quadrant QUAD1;
  pair PAIR;
  node NODE1;
  PAIR <- CONSTRUCT (LEVEL, 2**LEVEL, 2**LEVEL);
  if TYPE(PAIR) = GRAY then
    begin /* Image is GRAY */
      FATHER(POINTER(PAIR)) <- NULL;
      return (POINTER(PAIR));
    end
  else
    begin  /* The entire image is BLACK or WHITE */
      NODE1 <- CREATENODE();
      NODETYPE(NODE1) <- TYPE(PAIR);
      for QUAD1 in {NW,NE,SW,SE} do SON(NODE1,QUAD1) <- NULL;
```

```
      FATHER(NODE1) <- NULL;
      return (NODE1);
    end;
end;


pair procedure CONSTRUCT (LEVEL,X,Y);
/* Construct the portion of a quadtree of size 2**LEVEL by
2**LEVEL having its southeasternmost pixel corresponding to
entry B_ARRAY[X,Y] of the image array. B_ARRAY is a global
variable. */
begin
  integer value LEVEL,X,Y;
  /* P_ARRAY has entries corresponding to NW, NE, SW, SE */
  pair array P_ARRAY[NW...SE];
  quadrant QUAD1,QUAD2;
  node NODE1,NODE2;
  if LEVEL = 0 then  /* process the pixel */
    /* < , > creates a (POINTER,TYPE) pair */
    return (<COLOR (B_ARRAY[X,Y]), NULL>)
  else
    begin
      LEVEL <- LEVEL - 1;
      P_ARRAY[NW] <- CONSTRUCT (LEVEL, X-2**LEVEL, Y-2**LEVEL);
      P_ARRAY[NE] <- CONSTRUCT (LEVEL, X, Y-2**LEVEL);
      P_ARRAY[SW] <- CONSTRUCT (LEVEL, X-2**LEVEL,Y);
      P_ARRAY[SE] <- CONSTRUCT (LEVEL, X, Y);
      if TYPE (P_ARRAY[NW]) <> GRAY and
         TYPE (P_ARRAY[NW]) = TYPE (P_ARRAY[NE]) =
         TYPE (P_ARRAY[SW]) = TYPE (P_ARRAY[SE]) then
         return (P_ARRAY[NW]) /* All siblings are of the same type */
      else
      begin  /* Create a non-terminal GRAY node */
        NODE1 <- CREATENODE();
        for QUAD1 in {NW,NE,SW,SE} do
          begin
            if TYPE(P_ARRAY[QUAD1]) = GRAY then
              /* link P_ARRAY[QUAD1] to its father node */
              begin
                SON(NODE1,QUAD1) <- POINTER(P_ARRAY[QUAD1]);
                FATHER (POINTER(P_ARRAY[QUAD1])) <- NODE1;
              end
            else  /* Create a maximal node for P_ARRAY[QUAD1] */
              begin
                NODE2 <- CREATENODE();
                NODETYPE(NODE2) <- TYPE (P_ARRAY[QUAD1]);
                for QUAD2 in {NW,NE,SW,SE} do SON(NODE2,QUAD2) <- NULL;
                SON(NODE1,QUAD1) <- NODE2;
                FATHER(NODE2) <- NODE1;
              end;
          end;
```

```
            NODETYPE(NODE1) <- GRAY;
            return (<GRAY,NODE1>);
          end;
        end;
    end;
```

The running time of this algorithm is proportional to the number of pixels in the image. The maximum depth of recursion is equal to the log of the image diameter [SAME80].

2.2.3.2. Raster to Quadtree Conversion [SAME81]

In the event that an incoming image is too large to be completely present in core as is required by the above algorithm, a different approach must be used to build the quadtree. Raster to quadtree conversion views the incoming picture in a row by row fashion, starting with the first row. Pixels are visited by the algorithm in the order shown in Figure 2.F. Underlying this algorithm is the assumption is that at any instant of time, a valid quadtree structure will exist, with any unprocessed pixels assigned the value of WHITE. Nodes are merged into maximal blocks as the quadtree is built, making it unnecessary to consolidate nodes at a later time in a separate merging pass.

Processing an odd-numbered row requires less work than processing even-numbered row, since no merging of nodes is possible in an odd row. A node is created for each pixel visited in an odd-numbered row. As the tree is constructed, non-terminal nodes must also be added with the appropriate number of WHITE siblings to satisfy the requirement of a valid tree existing at all times. The processing of even-numbered rows is more

complicated, since node merging may also take place. A check for a possible node merge must be performed at every even-numbered pixel in an even-numbered row. (Once one merge occurs, it is necessary to check for merges farther up the tree.)

This algorithm is suitable for use with a run-length representation of a picture. The time complexity is proportional to the number of pixels in the image, as proven in [SAME81].

2.2.3.3. Quadtree to Raster Conversion [SAME84a]

The following algorithm with pseudocode will output the quadtree representation of an image to a raster display device.

This algorithm operates as a bottom-up inorder tree traversal. Each row of pixels is visited in sequence. For each row in the raster display, every BLACK or WHITE tree node corresponding to a region block which intersects the row is visited from left to right. Each BLACK and WHITE node at level L in the tree is therefore visited 2**L times (its height in pixels). The result of each visit is a BLACK or WHITE pixel run of length 2**L output to the raster display.

The only node visit that begins at the root of the quadtree is the initial visit to the northwesternmost block in the image. The algorithm proceeds to the next row by using the tree structure to locate the southern neighbor block, rather than beginning the search for the next row at the root of the tree each time.

Pseudocode for a quadtree to raster conversion follows, as given in [SAME84a] with some variable names changed for clarity.

```
procedure QUAD_TO_RASTER (ROOT,LEVEL)
/* Output a raster representation of the 2**LEVEL by 2**LEVEL
image corresponding to the quadtree rooted at node ROOT. For
each row, the leftmost block is located and then the blocks
comprising the row are visited in sequence by ascending and
descending the appropriate links in the tree. Successive blocks
in the vertical direction are located in the same manner */

begin
  value node ROOT;
  value integer LEVEL;
  node NODE1,NODE2;
  integer ROW,WIDTH,Y;
  NODE1 <- ROOT;
  WIDTH <- 2**LEVEL;


  /* Find the leftmost block containing row 0 */
  FINDBLOCK(NODE1,0,WIDTH,0,WIDTH,WIDTH);
  Y <- 0;
  do
    begin
      for ROW <- Y step 1 until Y + WIDTH - 1 do
          OUTROW(NODE1,ROW,LOG2(WIDTH));
          /* LOG2 returns the log of WIDTH to base 2 */
      Y <- Y + WIDTH;
      /* Find the leftmost block containing row Y */
      GTEQUAL_ADJ_NEIGHBOR(NODE1,'S',NODE2,LEVEL);
      WIDTH <- 2**LEVEL;
      if GRAY(NODE2) then FINDBLOCK(NODE2,0,WIDTH,Y,Y+WIDTH,WIDTH);
      NODE1 <- NODE2;
    end
    until NULL(NODE1);
end;

procedure FINDBLOCK (NODE,X,XFAR,Y,YFAR,WDTH);
/* NODE is the root of a block of width WDTH having its lower right
corner at (XFAR-1,YFAR-1). If NODE is BLACK or WHITE, then return
the values of NODE and WDTH; otherwise, repeat the procedure for
the son of NODE that has its upper left corner at (X,Y)  */
begin
  reference node NODE;
  value integer X,XFAR,Y,YFAR;
  reference integer WDTH;
  quadrant QUAD;
  if GRAY(NODE) then
  begin
    WDTH <- WDTH / 2;
    QUAD <- GETQUAD(X,XFAR-WDTH,Y,YFAR-WDTH);
    NODE <- SON(NODE,QUAD);
```

```
    case QUAD of
       'NW': FINDBLOCK(NODE,X,XFAR - WDTH,Y,YFAR - WDTH,WDTH);
       'NE': FINDBLOCK(NODE,X,XFAR,Y,YFAR - WDTH,WDTH);
       'SW': FINDBLOCK(NODE,X,XFAR - WDTH,Y,YFAR,WDTH);
       'SE': FINDBLOCK(NODE,X,XFAR,Y,YFAR,WDTH);
    end;
  end;
end;

quadrant procedure GETQUAD(X,XCENTER,Y,YCENTER);
/* Find the quadrant of the block rooted at (XCENTER, YCENTER)
   that contains (X,Y) */
begin
  value integer X,XCENTER,Y,YCENTER;
  return (if X < XCENTER then
             if Y < YCENTER then 'NW'
             else 'SW'
          else if Y < YCENTER then 'NE'
             else 'SE');
end;

procedure GTEQUAL_ADJ_NEIGHBOR(NODE1,DIR,RETNODE,LEV);
/* Return in RETNODE the neighbor of node NODE1 in horizontal or
vertical direction DIR. LEV denotes the level of the tree at which
node NODE1 is initially found and the level of the tree at which node
RETNODE is ultimately found. If such a node does not exist, then
return NULL.
Procedure ADJ(B,I) returns TRUE iff quadrant I is adjacent
to boundary B of the node's block.
Procedure FATHER(P) returns a pointer to node P's father.
Procedure SONTYPE(P) returns the specific quadrant in which
node P lies relative to its father.
Procedure REFLECT(B,I) returns the SONTYPE value of the block of
equal size that is adjacent to side B of a block having SONTYPE
value I. */
begin
  value node NODE1;
  value direction DIR;
  reference node RETNODE;
  reference integer LEV;
  LEV <- LEV + 1;
  if not NULL (FATHER(NODE1)) and ADJ(NODE1,SONTYPE(NODE1)) then
    /* Find a common ancestor */
    GTEQUAL_ADJ_NEIGHBOR (FATHER(NODE1),DIR,RETNODE,LEV)
  else RETNODE <- FATHER(NODE1);
  /* Follow the reflected path to locate the neighbor */
  if not NULL(RETNODE) and GRAY(RETNODE) then
     begin
     RETNODE <- SON(RETNODE,REFLECT(DIR,SONTYPE(NODE1)));
     LEV <- LEV - 1;
     end;
```

```
end;


procedure OUTROW (NODE,ROW,LEV);
/* Output a raster corresponding to all of the blocks that
have segments in row ROW starting with node NODE at level
LEV.
Procedure OUTPUTRUN writes a pixel run of the appropriate
color and of width WIDTH to the screen  */
begin
  value node NODE;
  value integer LEV,ROW;
  node TMPNODE;
  integer WIDTH;
  WIDTH <- 2**LEV;
  do
    begin
    OUTPUTRUN(NODETYPE(NODE),WIDTH);
    /* Find the leftmost adjacent block containing row ROW */
    GTEQUAL_ADJ_NEIGHBOR (NODE,'E',TMPNODE,LEV);
    WIDTH <- 2**LEV;
    if GRAY(TMPNODE) then
      FINDBLOCK(TMPNODE,0,WIDTH,ROW,ROW + WIDTH - (ROW MOD WIDTH),
                WIDTH);
    NODE <- TMPNODE;
    end
    until NULL(NODE);
end;
```

As stated in [SAME84a], the algorithm's running time is a measure of the number of nodes visited, and is proportional to the sum of the heights of the blocks comprising the image. The work required is therefore dependent on the image complexity. Storage requirements consist of enough space for the quadtree, plus space for one row of pixels.

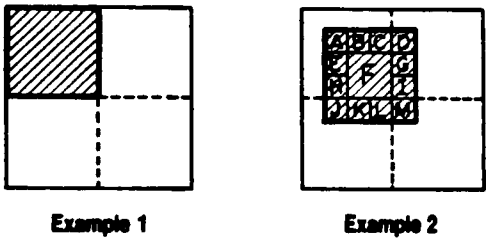
1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

2.E.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

2.F.

Figure 2.E. Traversal order of pixels in Binary Array to Quadtree Conversion
2.F. Traversal order of pixels in Raster to Quadtree Conversion [SAME84b]



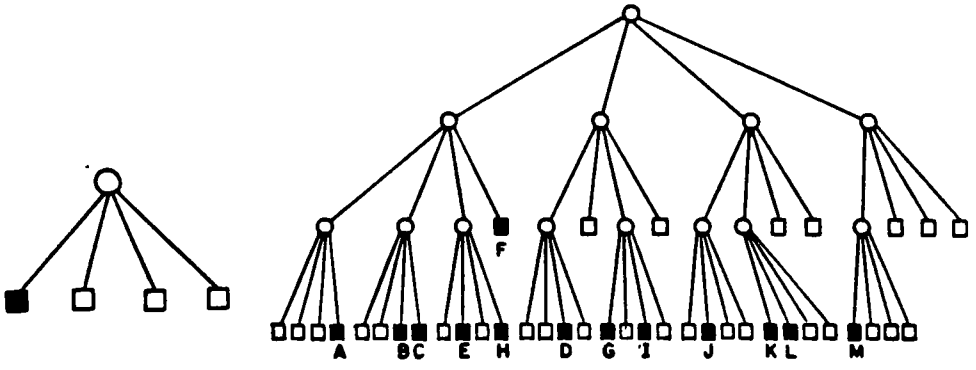


Figure 2.G. Effect of small image shift on quadtree data structure
[SCOT and IYEN-86]

2.3. Quadtree Normalization

A basic difficulty with the standard quadtree representation is its extreme sensitivity to the placement of an image on the grid. Small differences in image location, orientation, or size parameters will result in different quadtrees. Figure 2.G illustrates the effect on the quadtree structure of merely moving an image one pixel to the right and one pixel down. To remedy the effect of possible changes in one or more of these parameters, various "normalization" schemes have been proposed, i.e., methods of producing unique quadtree representations regardless of the positioning of the original image.

Such normalized quadtrees are then suitable for applications involving pattern matching. As an example of this, [CHIE and AGGA-84] store the silouettes of eight different airplanes represented by normalized quadtrees in an image library. They then generate normalized quadtrees for "noisy" images of airplanes (done by applying translations, rotations, scalings, and adding random noise to the airplane silouette before quadtree conversion takes place). By defining a measure of dissimilarity between two objects (i.e. assigning a dissimilarity weight to each tree node based on its level in the tree), they are able to calculate the dissimilarity between two images by summing up these measures as the two trees are traversed in parallel. Based on their experimentation, they assert that most of the test planes were correctly matched with the correct plane in the library, even with the addition of 90% noise.

Two normalization schemes were implemented for this application; one to normalize an image with respect to translation and rotation, and

another to normalize the image with respect to translation only.

2.3.1. Normalization with Respect to Translation, Rotation (and Size)

The first approach [CHIE and AGGA-84] is motivated by the concepts of centroid and principal axes from mechanics. These are geometric properties of an object that are invariant to the location and orientation of the object. Prior to generation of the quadtree, each object in an image is normalized to a coordinate system with the object's centroid as the origin, and with the principal axes becoming the new coordinate axes. After the centroid and principal angle for an object are calculated, the object is rotated around its centroid by the degree of the principal angle. (See Figure 2.H.) A quadtree constructed from this normalized object is no longer sensitive to translation, or rotation.

It is also a feature of this algorithm to scale the image to a standard grid size (preferably a power of two), resulting in a representation that is insensitive to size. This feature is being noted, although it was not actually implemented in this application.

In order to restore an original object from its normalized quadtree, it is necessary for extra information to be carried along as part of the data structure. Specifically, a linked list of each quadtree comprising the image is suggested. For each quadtree, information is stored pertaining to the coordinates of the object centroid, the angle of rotation of the object (principal angle), and, if normalization by size is included, fields representing the new standard size, and the amount that the object has been scaled to reach the standard size.

This approach will not necessarily yield a minimal quadtree, but the quadtree will be unique for the object.

2.3.2. Normalization with Respect to Translation Only

This approach [LI82, et al.] results in a "minimal-cost" quadtree for an image, that is unique over the class of all translations of the image. Minimal-cost refers to a quadtree with the fewest possible leaves (non-terminal nodes) necessary to store the image. Since the total number of nodes (terminal and non-terminal) in a tree is proportional to the number of leaves in the tree, the algorithm in effect minimizes the total number of quadtree nodes. The benefits of such a representation are obvious; there will be fewer nodes to store, and fewer nodes involved in subsequent tree traversal operations. This technique is also of use for applications involving shape-matching.

A simple, intuitive approach to this problem would be to translate the image in all possible ways within the array, count the number of leaves required for the quadtree at each translation, and ultimately choose the tree representation with the fewest leaves. This is expensive (O(s**4), where s is the length of the grid) and unnecessary, as will be explained below.

The algorithm assumes a starting image embedded in the northwest quadrant of a 2**n by 2**n array. A theorem in [LI82, et al.], states that in this case, the optimal grid size will be either 2**(n-1), or 2**n. An "enclosing rectangle" for the image is defined as the smallest rectangle that encloses the image and whose sides are parallel to the coordinate axes. The image's enclosing rectangle will initially abut the northwest corner of the grid. A variable LEAFCOUNTER is initialized

to S**2, where S is the original grid length.

The algorithm proceeds by recursively reducing the image array by powers of two while calculating the number of leaves needed for the image at different positions in the array. It is possible to narrow the number of translations processed to within 2**(n-1) - 1 pixels in the eastern direction, and 2**(n-1) - 1 pixels to the south. A representation with the fewest possible tree nodes will always be found within this subset of the total grid area. (Proof of this can be found in [LI82, et al.].) For any image, there may be more than one translation resulting in a quadtree representation with the smallest number of nodes. It is therefore necessary to methodically choose one of the representations as the unique one. If an ambiguous situation occurs, the algorithm will tentatively choose the translation whose northernmost BLACK pixel is a minimum Y value. If more than one choice for a minimal tree is still possible, the translation with the smallest X value is chosen.

The algorithm "translates" the image four times: by 0 pixels, by 1 pixel to the east, by 1 pixel to the south, and by 1 pixel to the southeast. After each translation, the array is compacted to 1/4 of its former size by replacing every 2 x 2 pixel region with a temporary "supernode" representing the contents of the four nodes. A BLACK or WHITE "supernode" denotes four pixels being all BLACK or all WHITE; a GRAY "supernode" specifies a non-homogeneous region. In the case of all four pixels being all BLACK or all WHITE, LEAFCOUNTER is reduced by 3, meaning that one leaf is sufficient to represent this region, rather than four. The process is repeated recursively on the new, compacted arrays until all combinations are considered.

It should be noted that the algorithm does not actually perform the four image translations. A "dummy" row and column indexed by zero and initialized to WHITE is added to the array before beginning, in order to simulate translations by one pixel by changing pixel coordinates only.

In this algorithm, it is possible to detect the situation where the smaller of the two grid sizes (2**[n-1]) can accommodate the image (i.e. when the complete, compacted image lies completely within the northwest quadrant of the original grid size).

Space requirements are on the order of 2 x (2**n) pixels, and execution time is on the order of (n x 2**2n) [SCOT and IYEN-86].

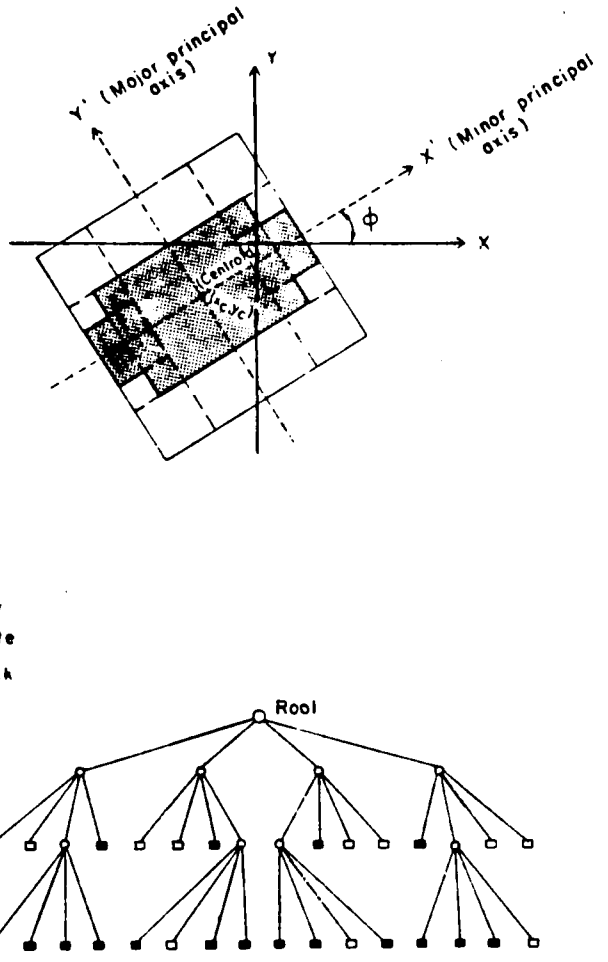


Figure 2.H. A normalized image and its quadtree representation [CHIE and AGGA-84]

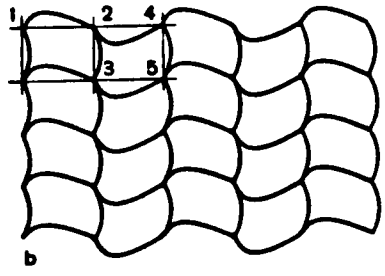


Figure 2.I. Example of lattice points, unit cell of a pattern [SHUB and KOPT-74]

2.4. Symmetry Overview [SCHA78]

The second half of this paper introduces what are referred to as the two-dimensional plane symmetry groups, and relates them to a hierarchical image representation. Plane symmetry groups are well-known in the fields of mathematics and crystallography. Perhaps one of the most fascinating non-scientific applications of these groups can be found in the drawings of Dutch artist M. C. Escher. (Seven Ecsher drawings are found at the back of this paper.)

Plane symmetry groups are also known as "wallpaper groups", as explained by Schattschneider in [SCHA78]: "Patterns which are invariant under linear combinations of two linearly independent translations repeat at regular intervals in two directions, and hence their groups are often termed "wallpaper groups". Seventeen distinct plane symmetry groups are defined.

2.4.1. Plane Symmetry Groups

The following six definitions helpful to the understanding of two-dimensional plane symmetry groups (classes) are taken from [SCHA78]:

1. A "PERIODIC" OR "REPEATING" PATTERN is a design having the following property: There exist

    a) a finite region and
    b) two linearly independent translations

such that the set of all images of the region when acted on by the group generated by these translations produces the original design.

2. The TRANSLATION GROUP of a periodic pattern consists of the set of all translations that map the pattern onto itself.

3. A UNIT of the pattern is a smallest region of the plane having the property that the set of its images under this translation group covers the plane. All units have the same area, but their outlines can have infinite variation.

4. Every periodic pattern has naturally associated to it a LATTICE of points; choosing any point in the pattern, this lattice is the set of all images of that point when acted on by the translation group of the pattern.

5. A UNIT CELL is a unit which is a parallelogram whose vertices are lattice points. The vectors which form the sides of a unit cell generate the translation group of a pattern. (See Figure 2.I.) In addition to translations, a periodic pattern may also be mapped onto itself by any of the other plane isometries: rotations, reflections, or glide reflections.

6. The GENERATING REGION of a periodic pattern is the smallest region of the plane whose images under the full symmetry group of the pattern cover the plane. Figure 2.M shows the generating regions for each plane symmetry group.

In order to summarize the seventeen groups of plane symmetry, the following table is presented as defined by those in the field of crystallography. There are four basic plane transformations involved in the classification of such tilings; they are translation, rotation, reflection, and glide reflection. Combinations of these yield the seventeen distinct symmetry classes. In "full international notation", each symmetry class is described as a concatenation of up to four symbols.

Interpretation of full international symbols (left to right):
[SHUB and KOPT-74]:

1. letter p or c denotes primitive or centered cell
2. integer n denotes highest order of rotation
3. symbol denotes a symmetry axis normal to the x-axis (the x-axis is the left edge of the cell):

m (mirror) - reflection axis
g - no reflection, but a glide-reflection axis
1 - no symmetry axis

4. symbol denotes a symmetry axis at angle & to x-axis, with & dependent on n:

& = 180 for n=1 or 2
45 for n=4
60 for n=3 or 6

symbols m,g,1 are interpreted as in 3.

No symbols in the third and fourth position indicate that the group contains no reflections or glide-reflections.

Class	Int'l (short)	Int'l (full)	Class	Int'l (short)	Int'l (full)
1	p1	(p1)	9	cmm	(c2mm)
2	p2	(p211)	10	p4	(p4)
3	pm	(p1m1)	11	p4m	(p4mm)
4	pg	(p1g1)	12	p4g	(p4gm)
5	pmm	(p2mm)	13	p3	(p3)
6	pmg	(p2mg)	14	p31m	(p31m)
7	pgg	(p2gg)	15	p3m1	(p3m1)
8	cm	(c1m1)	16	p6	(p6)
			17	p6m	(p6mm)

All symbols, with the exception of p31m and p3m1, can be referred to in the shortened form, with no loss of identification [SHUB and KOPT-74].

The above table is expressed visually in Figure 2.L. Figure 2.N shows the effect of applying the seventeen symmetry groups on a single triangle.

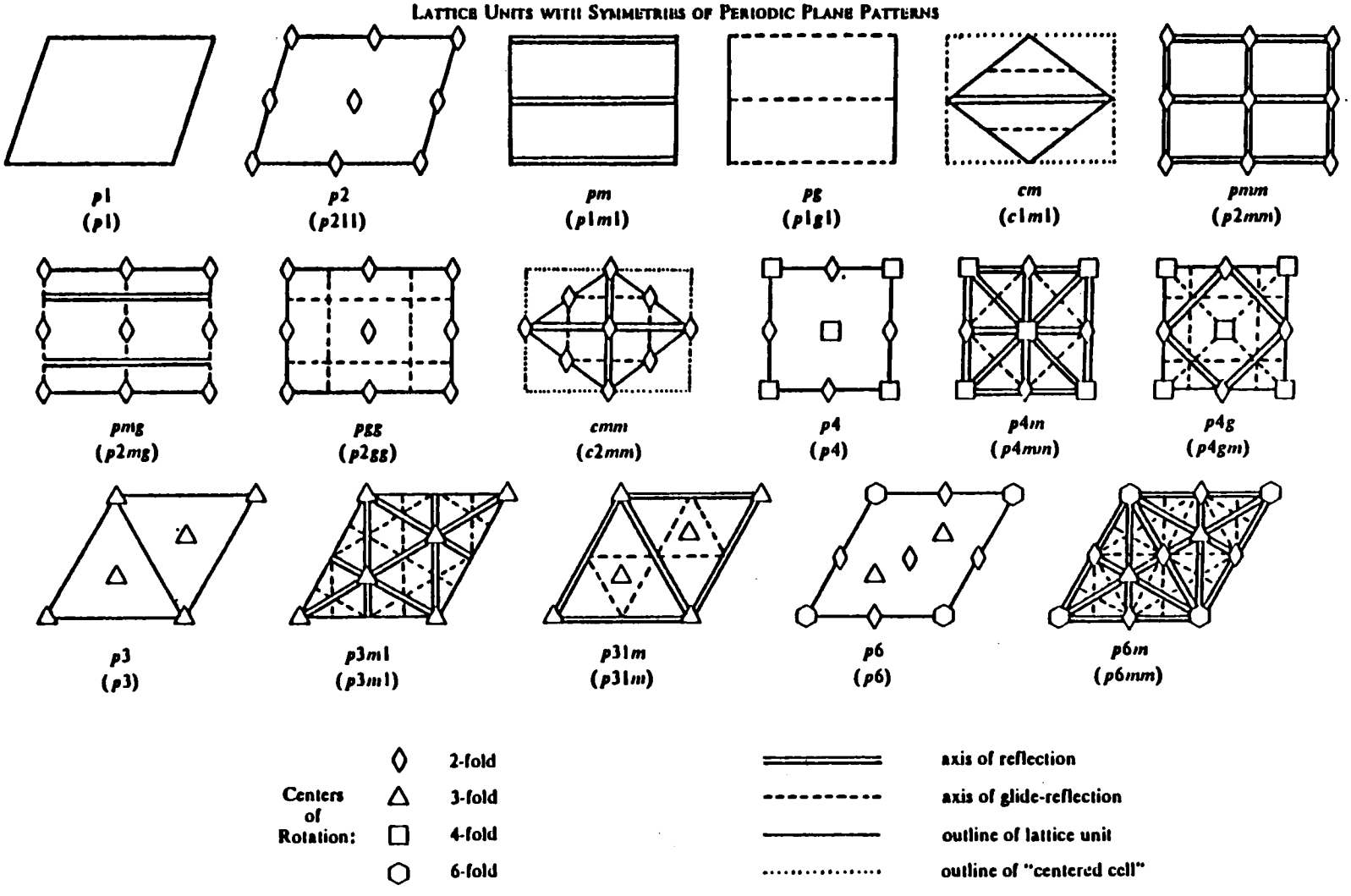
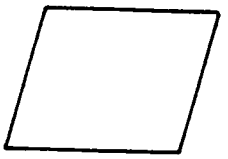


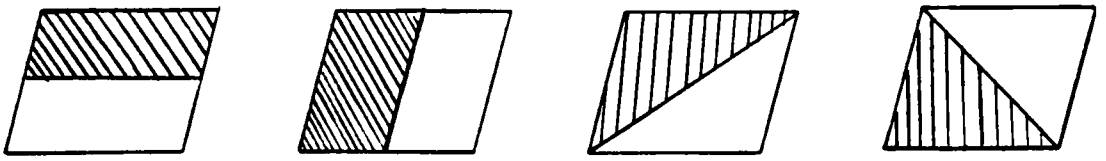
CHART 2. International notation identifies the seventeen two-dimensional crystallographic groups. The short form is given first, with the full notation in parentheses.

Figure 2.L. Reprinted from [SCHA78]

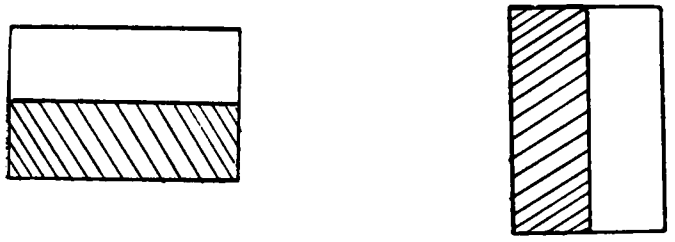
Figure 2.M. Generating regions for the 17 plane symmetry groups [TANIS]



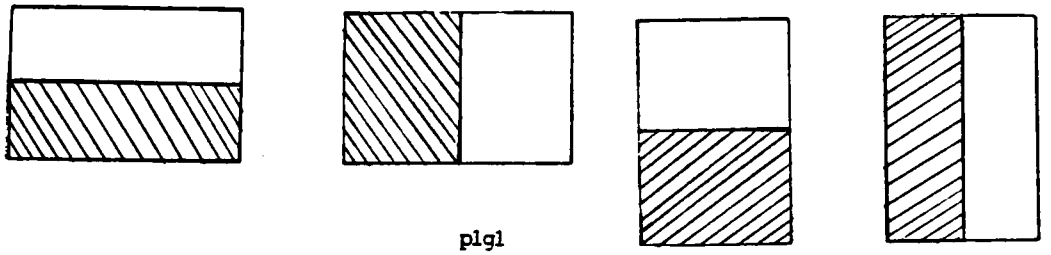
p1



p211

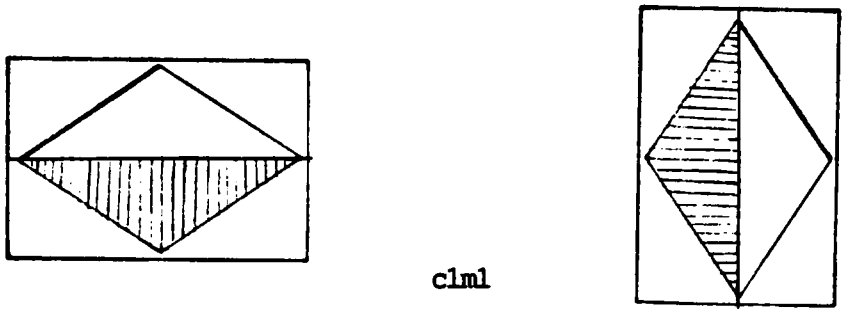


p1m1

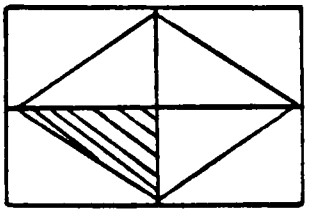


p1g1

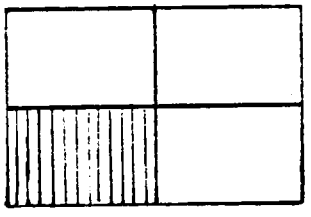
Figure 2.M. (continued) Generating regions for the 17 plane symmetry groups [TANIS]



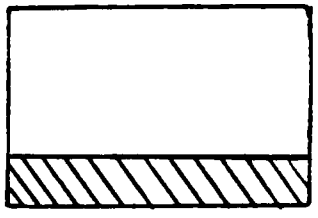
c1m1

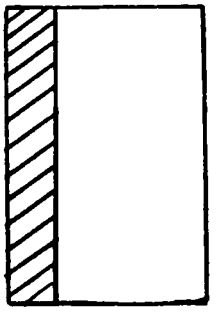


c2mm



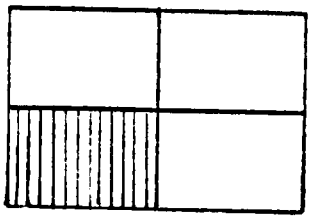
p2mm



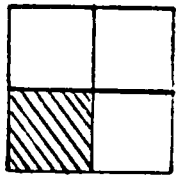


p2mg

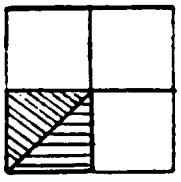
Figure 2.M. (continued) Generating regions for the 17 plane symmetry groups [TANIS]



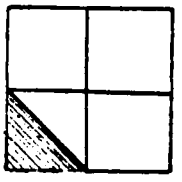
p2gg



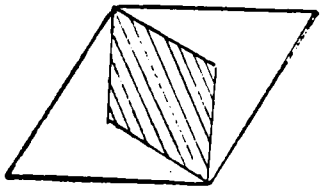
p4



p4mm

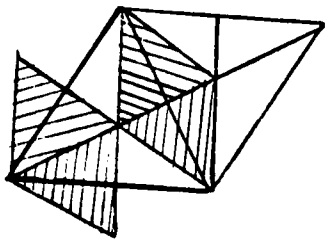


p4gm

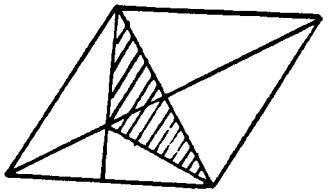


p3

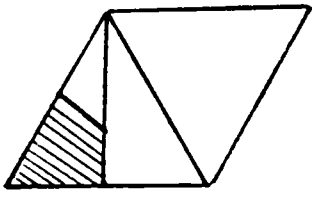
Figure 2.M. (continued) Generating regions for the 17 plane symmetry groups [TANIS]



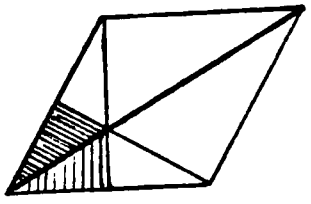
p3m1



p31m



p6



p6mm

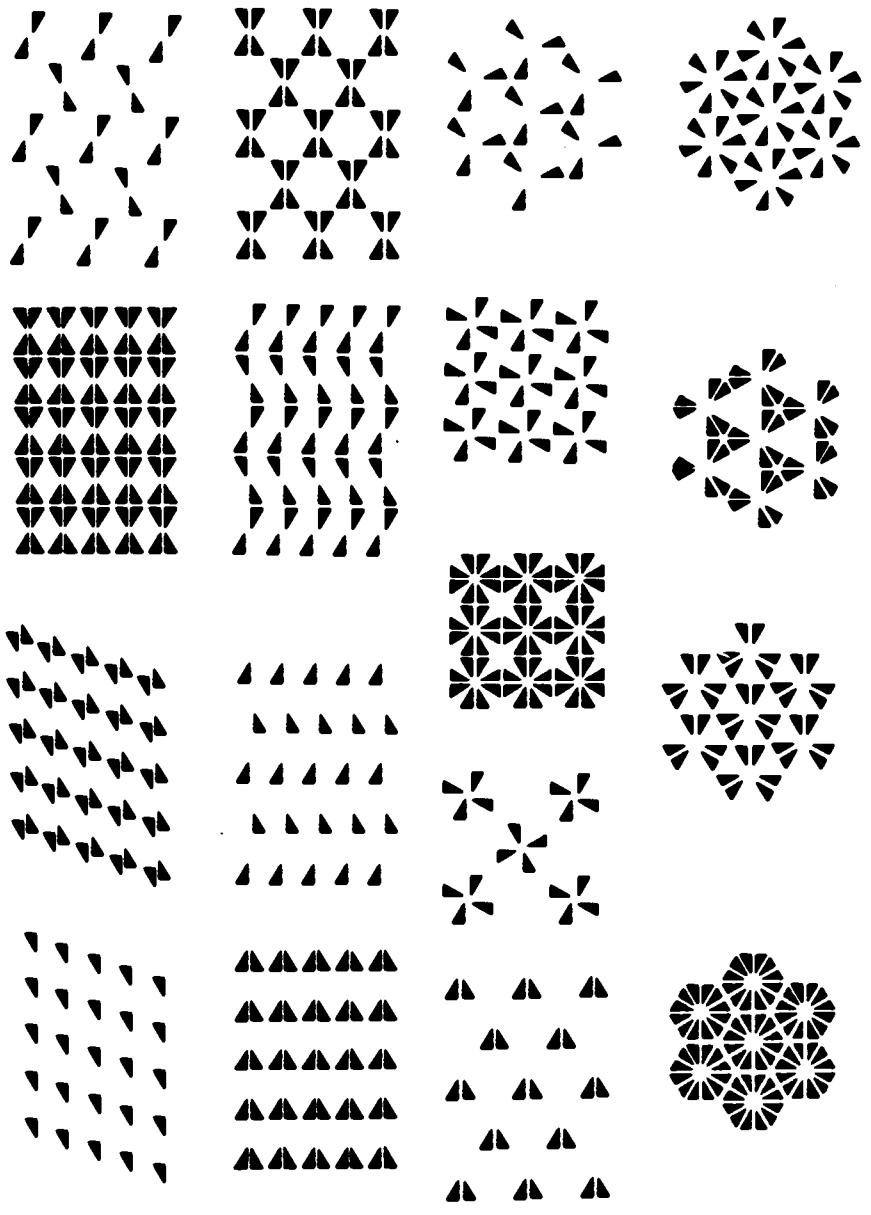


Figure 2.N. The effect of the seventeen plane symmetry groups on a single triangle. [SHUB and KOPT-74]

2.4.2. Symmetry and Quadtree Representation

The ability to organize global picture information within the quadtree structure makes it possible to use the structure to detect the existence of basic symmetries in an image.

In the analysis of a difficult problem, it is often the case that the discovery of symmetries can lead to a simplification of the problem. In the field of image analysis, symmetry is one parameter that may be used in the typical problem of classifying of an unknown image pattern according to some criterion [ALEX and KLIN-78].

In [ALEX and KLIN-78], it is proposed that node patterns exist in a quadtree representation that correspond to directional symmetries. (Directional symmetries are only one example of structural information present in an image; other such information may include shape, perimeter, straight lines, etc.) The hierarchical nature of the quadtree data structure lends itself to an efficient means of identifying the symmetry characteristics of a picture. Using a recursive search method to detect the presence of a particular symmetry, it is possible to stop searching as soon as it is determined that two node pairs at a certain level do not belong to the desired class of directional symmetry.

Relationships between picture symmetries and tree nodes have been derived with respect to the horizontal, vertical, right-diagonal and left-diagonal directions. (In a general sense, it is possible to define more directions than these.) Two pixels, lying symmetrically opposite and equidistant with respect to a given line (axis of symmetry) are said to

constitute a "class". The classes defined correspond to the following four principal directions [ALEX and KLIN-78]:

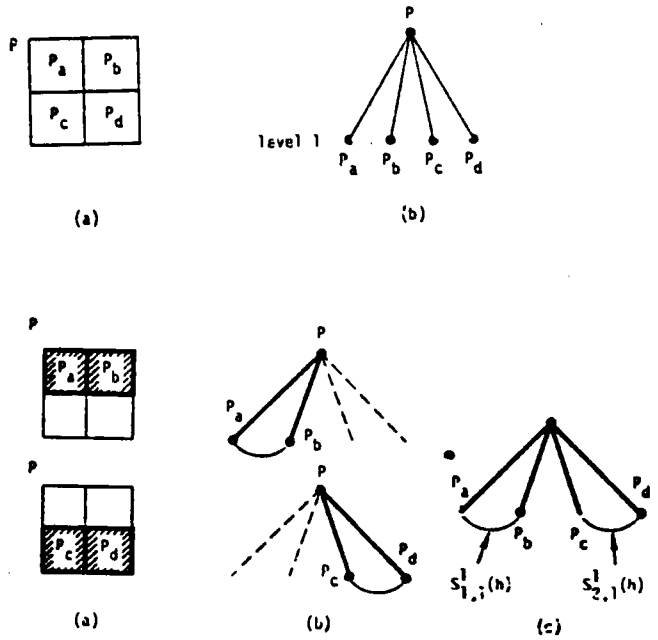
HORIZONTAL (0 or 180 degrees): pixels symmetrically opposite a vertical line
VERTICAL (90 or 270 degrees): pixels symmetrically opposite a horizontal line
RIGHT-DIAGONAL (45 or 225 degrees): pixels symmetrically opposite a left-diagonal line
LEFT-DIAGONAL (135 or 315 degrees): pixels symmetrically opposite a right-diagonal line

Figure 2.0 shows the methodology used to begin derivation of node classes for horizontal and right-diagonal symmetries at the lowest levels of an image (i.e., on a 2 x 2 pixel grid, and a 4 x 4 pixel grid). Vertical and left-diagonal symmetries are derived in a similar manner. In general, all possible patterns of nodes that satisfy a given directional symmetry are found at successive levels of a picture, and then generalized. The class notation seen in Figure 2.0 is defined below [ALEX and KLIN-78]:

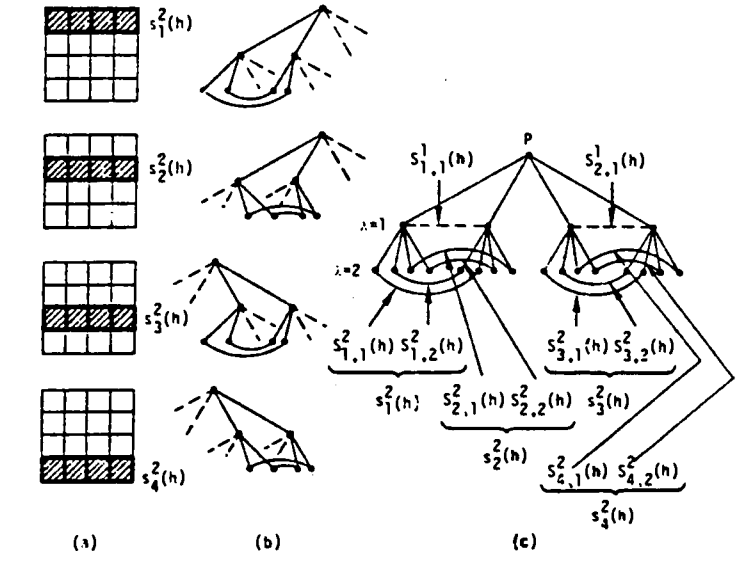
$S^{\lambda}(i,j)(u)$ represents a directional symmetry class for two grid points, where

λ is the tree level at which these grid points are found.
i is the ith "set" to which all clases containing grid-points that lie on the same straight line in the picture belong.
j is the jth class.
u is the direction along which the grid-points lie:
u = h : HORIZONTAL
u = v : VERTICAL
u = r : RIGHT-DIAGONAL
u = l : LEFT-DIAGONAL

A rigorous derivation of the general case for all four classes can be found in [ALEX and KLIN-78].

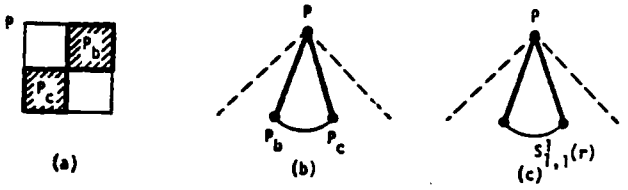


Horizontal symmetry classes at level 1. (a) The paired grid-points (shaded squares); (b) the corresponding paired nodes of the tree (solid lines); (c) the h-classes.

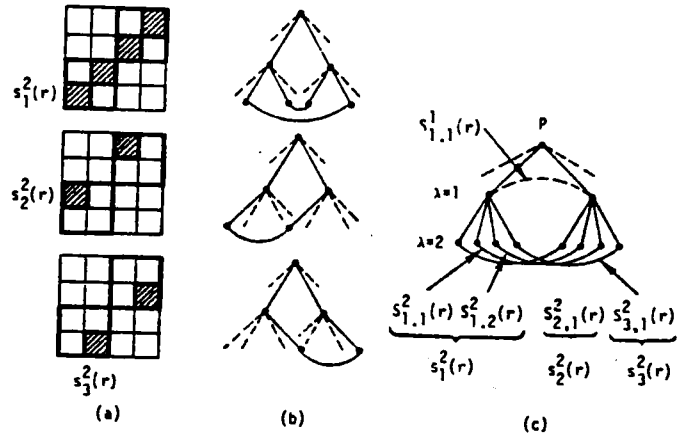


Horizontal symmetry classes at level 2. (a) The paired grid-points (shaded squares) which define sets; (b) the corresponding paired nodes of the trees (solid lines); (c) the h-classes.

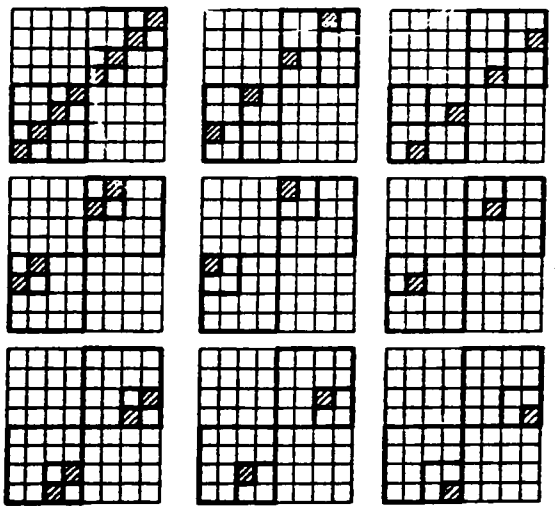
Figure 2.0. Partial derivation of horizontal symmetry class for quadtree [ALEX and KLIN-78]



Right-diagonal symmetry class at level 1. (a) The paired grid-points (shaded squares); (b) the corresponding paired nodes of the tree (solid lines); (c) the r-class.



Right-diagonal symmetry classes at level 2. (a) The paired grid-points (shaded squares); (b) the corresponding paired nodes of the tree (solid lines); (c) the r-classes.



Right-diagonal symmetry classes at level 3. The shaded squares identify the nine different sets.

Figure 2.0. (continued) Partial derivation of right-diagonal symmetry class for quadtree [ALEX and KLIN-78]

An application of this is found in [BURT84, et al.], where an algorithm in FORTRAN 77 (extended to support recursion) is presented that will identify specific directional symmetries in a hierarchically-represented image. (In addition to the four major directions mentioned previously, algorithms are included that will detect 90 and 180 degree rotational symmetries.) An algorithm for 270 degree rotational symmetry may be constructed in a similar manner. The hierarchical data structure used is not actually a quadtree, but a close relative known as an exponential pyramid [SAME84b] [BURT84, et al.]. Below, two of the FORTRAN subroutines have been expressed in a more general way, and modified to relate to the quadtree data structure relevant to this paper. It should be noted that this modified algorithm does not represent the most efficient way for a quadtree to be handled, as it was originally written for a different hierarchical data structure.

Subroutine HORZ checks for horizontal symmetries. The function looks at the image in as many resolutions (levels) as necessary to determine whether or not a horizontal symmetry exists. If a mismatch is found at any level, the algorithm terminates without considering lower levels. At each level the first major quadrant is compared with the second quadrant, and the third major quadrant with the fourth quadrant.

Assume the following:

ROOT_NODE represents the root of the quadtree being examined.
The HORZMAP array contains the quadrants numbered in the order necessary to test for horizontal symmetries.
NW=0, NE=1, SW=2, SE=3
MAX_LEVEL is the log base two of one side of the bitmap (E.g. an 8x8 pixel map will have MAX_LEVEL = log base 2 (8) = 3)

```
LOGICAL FUNCTION HORZ (ROOT_NODE,MAX_LEVEL)

ENTERING ARGUMENTS:  ROOT_NODE, MAX_LEVEL
```

```
RETURNING ARGUMENTS: NONE

INTEGER NW, NE, SW, SE
INTEGER HORZMAP(4), LEVEL, MAX_LEVEL
QUADTREE_NODE ROOT_NODE
LOGICAL TEST
DATA HORZMAP /NE,NW,SE,SW/

/* Examine two pairs of major quadrants at as many levels
   as necessary to determine lack of horizontal symmetry.
   If the loop continues to completion, horizontal symmetry
   exists. */
FOR LEVEL = 1 TO  MAX_LEVEL
    {
    IF  (SYMM (HORZMAP, NW, NE, 1, LEVEL) == .TRUE.) .AND.
        (SYMM (HORZMAP, SW, SE, 1, LEVEL) == .TRUE.)
    THEN
        HORZ = .TRUE.
     ELSE
        {
        HORZ = .FALSE.
        BREAK
        }
    }

RETURN
END
```

Subroutine SYMM tests to see if tree nodes NODE1 and NODE2, both at level LEVEL are symmetrical at level CURRENT_LEVEL. The four element array RELATION defines the type of symmetry which must exist between nodes NODE1 and NODE2. Assume the following conventions:

NODETYPE(node) is a function that returns a value of BLACK, WHITE, or GRAY.
SON(i,node) points to the ith son of a particular node
NW=0, NE=1, SW=2, SE=3
LEVEL 1 is the first level beneath the root of the tree.
LEVEL N is the last possible level (individual pixels).

```
LOGICAL FUNCTION SYMM (RELATION, NODE1, NODE2, LEVEL, CURRENT_LEVEL)

ENTERING ARGUMENTS:  RELATION, NODE1, NODE2, LEVEL, CURRENT_LEVEL
RETURNING ARGUMENTS: SYMM

INTEGER   RELATION(4), LEVEL, CURRENT_LEVEL
QUADTREE_NODE   NODE1, NODE2, SON

/*  Assume specified symmetry is not present. */
SYMM = .FALSE.
```

```
IF (LEVEL == CURRENT_LEVEL) THEN
     {
     /*  See if image contains specified symmetry at CURRENT_LEVEL. */
     IF (NODETYPE(NODE1) == NODETYPE(NODE2)) THEN
        SYMM = .TRUE.
     }
ELSE
     {
     /* Check level LEVEL+1 for the specified symmetry, with the
        proper combinations of node sons. The first failure of
        these conditions causes the routine to exit */
     IF   (SYMM (RELATION, SON(NW,NODE1), SON(RELATION(1),NODE2),
                 LEVEL+1, CURRENT_LEVEL)  == .TRUE.)   .AND.
          (SYMM (RELATION, SON(NE,NODE1), SON(RELATION(2),NODE2),
                 LEVEL+1, CURRENT_LEVEL)  == .TRUE.)   .AND.
          (SYMM (RELATION, SON(SW,NODE1), SON(RELATION(3),NODE2),
                 LEVEL+1, CURRENT_LEVEL)  == .TRUE.)   .AND.
          (SYMM (RELATION, SON(SE,NODE1), SON(RELATION(4),NODE2),
                 LEVEL+1, CURRENT_LEVEL)  == .TRUE.)
     THEN
          SYMM = .TRUE.
     }
RETURN
END
```

Two algorithms used in the detection of directional symmetry
Modified from [BURT84, et al.]

Routine HORZ is easily modifiable to apply to other symmetries. By re-ordering the quadrants in the HORZMAP array, other symmetries may be found. Routine VERT, for example, would contain VERTMAP set to /SW,SE,NW,NE/, with the comparison of major quadrant pairs modified to be NW,SW and NE,SE.

3. FUNCTIONAL SPECIFICATION

The object of this application is to use quadtree-encoded images as a basis for the generation of patterns according to the rules of selected plane symmetry groups. It is implemented on a UNIX PC, using the C language in conjunction with bitmap graphics routines that serve to write individual pixels to a bitmap region, and ultimately display them on the screen. The screen dimensions for the machine are 348 by 720 pixels, making the largest 2**n by 2**n region possible for this application 256 x 256 pixels. If normalization of the image is desired, the maximum image input size is reduced to 64 x 64 due to programming space considerations. The final image may be output to a region of maximum 256 x 256 pixels, whether or not normalization is desired. The application can be divided into six sections:

1. Introduction of Image Data

Given an initial pixel region of size 2**n by 2**n, the entire region is assumed to represent the unit cell of a symmetry pattern. A smaller internal area is designated as the generating region for the symmetry group. A starting image is introduced into this generating region, either via editing a text file, or by calling bitmap graphics routines designed to turn on specified pixel positions in a bitmap region. Use of a text editor is preferable given a "small" bitmap size (e.g., 16 x 16 or 32 x 32 pixels). A file can be opened and 1's placed in each position within the generating region where a pixel should be BLACK. (See Figures 5.A through 5.I) The program will read the text file in a row by row

fashion, and turn on those pixels in the bitmap region that correspond to the positions of the 1's. A separate text file is used to hold the original image for each symmetry class. If the starting image desired is too large to be created easily by marking pixel positions using a text editor, the image can first be plotted on graph paper and ultimately written to the bitmap region using calls to the bitmap graphics routines within the program.

At this point, prompts appear for image input size, output size, and whether or not to invoke normalization.

2. Selection of a Plane Symmetry Group

One of nine plane symmetry classes will be prompted for. Nine classes out of a possible seventeen are simulated using this quadtree technique. The classes are p1, p211, pm, p2mm, p4, p4mm, p1g1, p2gg, and p4gm as defined in Section 2.4.1. Printing offsets dependent on the symmetry class chosen are prompted for at this time. The purpose of these offsets is explained in Section 5.2.

3. Normalization of the Image (optional)

If normalization of the image is desired, the input image region will be normalized with respect to translation, using the algorithm described in Section 2.3.2. This is done prior to its conversion to a quadtree structure. Normalization is an optional step, as will be explained in Section 5.2.

4. Construction of the Quadtree

The full bitmap region containing the pattern's generating region is written to a local image array, and then converted to a quadtree using the "Binary Array to Quadtree" algorithm described in Section 2.2.3.1. Upon completion of the quadtree conversion, a node representing the quadtree's root is returned, to be used in subsequent processing.

5. Construction of a Unit Cell from the Quadtree Structure

A set of quadtree output operations are now applied to the quadtree-encoded generating region to construct the unit cell of the pattern according to the specified symmetry class. These operations can be described as a series of calls to a single quadtree-to-raster output routine (Section 2.2.3.3), invoked each time with a different traversal order for the quadtree nodes (picture quadrants). The quadrant permutations that enable various transformations of an image have been shown in the table in Section 2.1.1.

Specifically, unit cells for the nine symmetry classes used in this application are constructed using the following sequences of image transformations on the quadtree-encoded image.

SYMMETRY CLASS	IMAGE TRANSFORMATIONS TO PRODUCE UNIT CELL
p1:	NO OP
p211:	NO OP, ROT_BY_180
p1m1:	NO OP, HORIZ_REFLECT
p2mm:	NO OP, HORIZ_REFLECT, VERT_REFLECT, ROT_BY_180

p4: NO OP, ROT_BY_90, ROT_BY_180, ROT_BY_270

p4mm: NO OP, ROT_BY_90, ROT_BY_180, ROT_BY_270, RIGHT_DIAG_REFLECT, HORIZ_REFLECT, VERT_REFLECT, LEFT_DIAG_REFLECT

p1g1: NO OP, REARRANGE_QUADS: 0<->1 2<->3, VERT_REFLECT

p2gg: NO OP, ROT_BY_180, REARRANGE_QUADS: 0<->3, VERT_REFLECT, HORIZ_REFLECT

p4gm: NO OP, ROT_BY_90, ROT_BY_180, ROT_BY_270, REARRANGE_QUADS: 0<->3, VERT_REFLECT, HORIZ_REFLECT, RIGHT_DIAG_REFLECT, LEFT_DIAG_REFLECT

"NO OP" corresponds to the quadtree output algorithm invoked with the standard traversal order of 0,1,2,3, and is always necessary in order to write the original generating region to the bitmap untransformed. Each subsequent operation within a symmetry group refers to the same output algorithm invoked with a different traversal order for image quadrants. One section of the unit cell is constructed per operation. The full collection of specified quadtree operations within a symmetry group will form the entire unit cell.

The first six symmetry classes are distinguished from the final three, as the method for producing them differs slightly. The first six classes may be simulated using only the transformations available in the table in Section 2.1.1. For the last three symmetry groups, an intermediate step not in the table is necessary, involving the rearrangement of pointers to the four major quadrants below the quadtree root. This is done to simulate the glide reflection operation present in these three classes.

Specifically, the three "special" classes are produced as follows:

p1g1 - Reverse the node pointers to quadrants 0 and 1, and then to 2 and 3 in the image quadtree, after the original generating region is drawn. This serves to switch the left half and the

right half of the picture "invisibly" (without drawing it), so that the final vertical reflection operation on the quadtree will reflect the right half of the picture over the center horizontal line before it is drawn on the screen. Refer to the plg1 example in Figure 5.G.

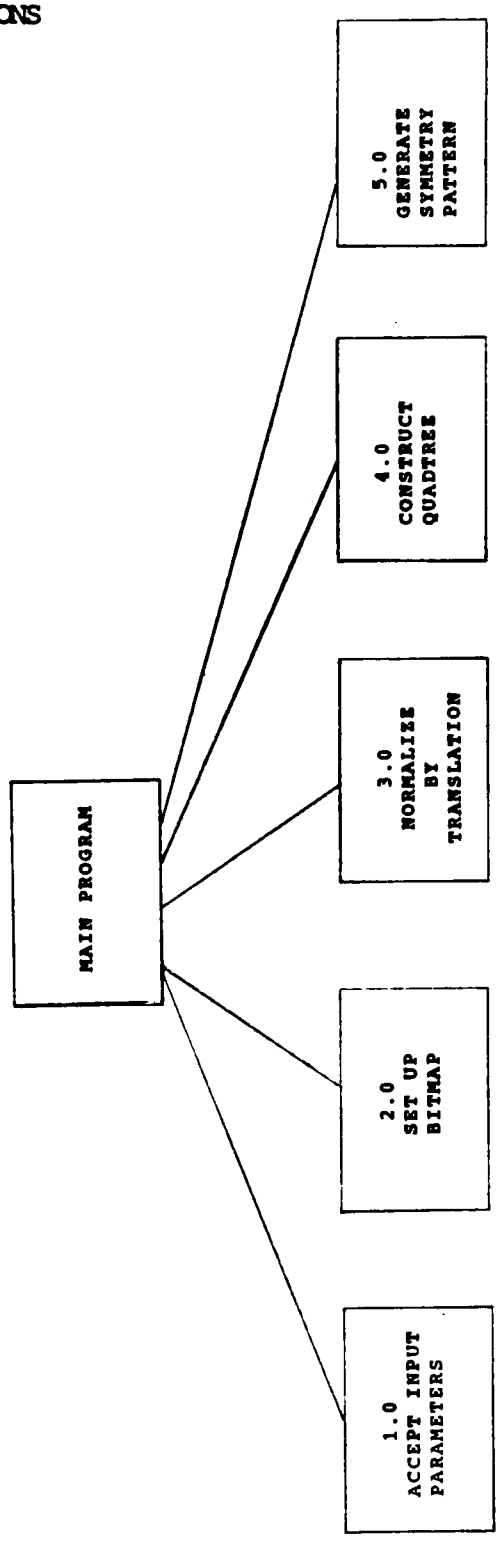
p2gg - The original image is drawn along with its 180 degree rotation. The original image is then moved from quadrant 0 to quadrant 3 (without drawing it) by reversing the node pointers to these two quadrants. The "invisible" image now in quadrant 3 is operated on by a vertical and horizontal reflection, before being drawn. Refer to the p2gg example in Figure 5.H.

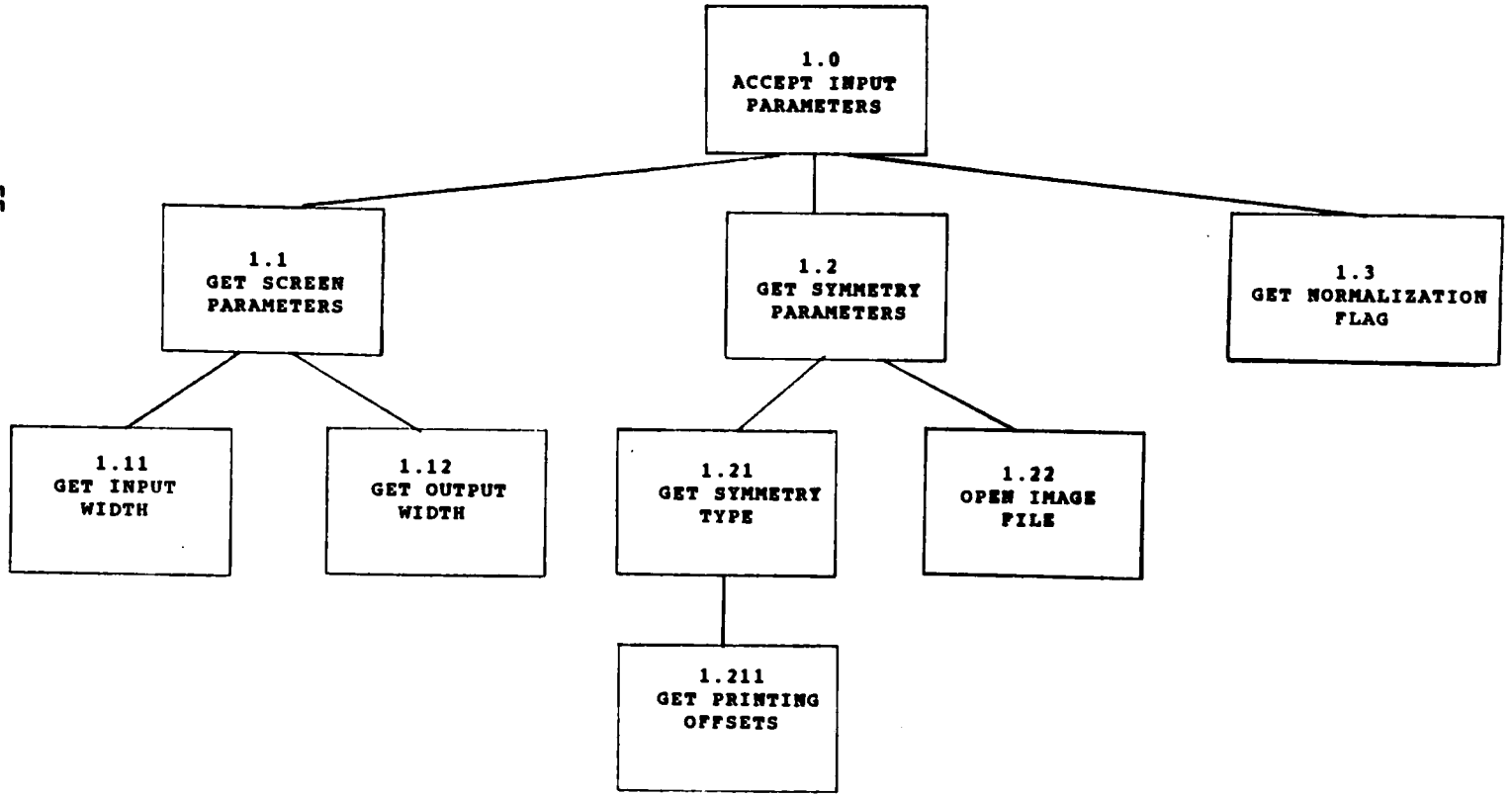
p4gm - After drawing the original image and all possible rotations, reverse the node pointers to quadrants 0 and 3. This will translate the original image in a diagonal direction, without drawing it. The final four operations (vertical, horizontal, right diagonal, and left diagonal reflections) are performed on the translated image before each portion is drawn to the screen. Refer to the p4gm example in Figure 5.I.

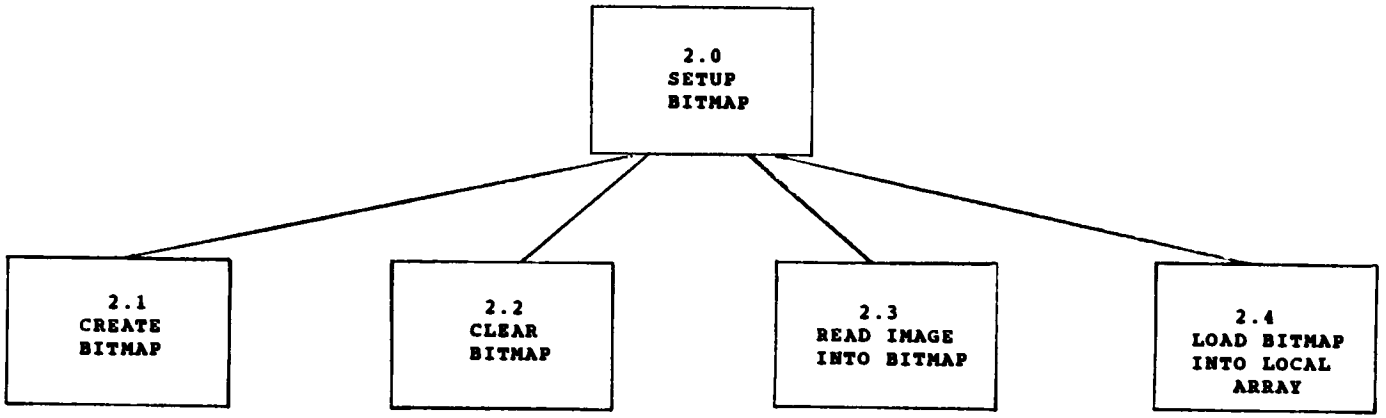
6. Propagation of the Pattern

After the full unit cell resides in the bitmap region, a bitmap graphics output routine is used to propagate copies of the unit cell across the screen in the x and y directions. (This no longer involves the quadtree structure.) Printing "offsets" representing a pixel displacement in each direction can be used in propagating each row of unit cell copies, thereby increasing the number of patterns within a symmetry class that can be displayed using this scheme. This is explained in greater detail in Section 5.2. The full pattern is now present on the screen.

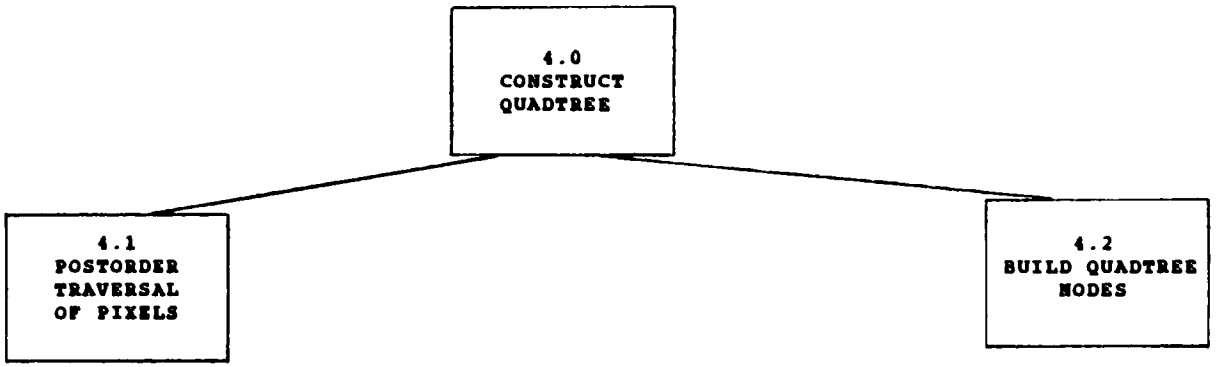
4. SOFTWARE SPECIFICATIONS

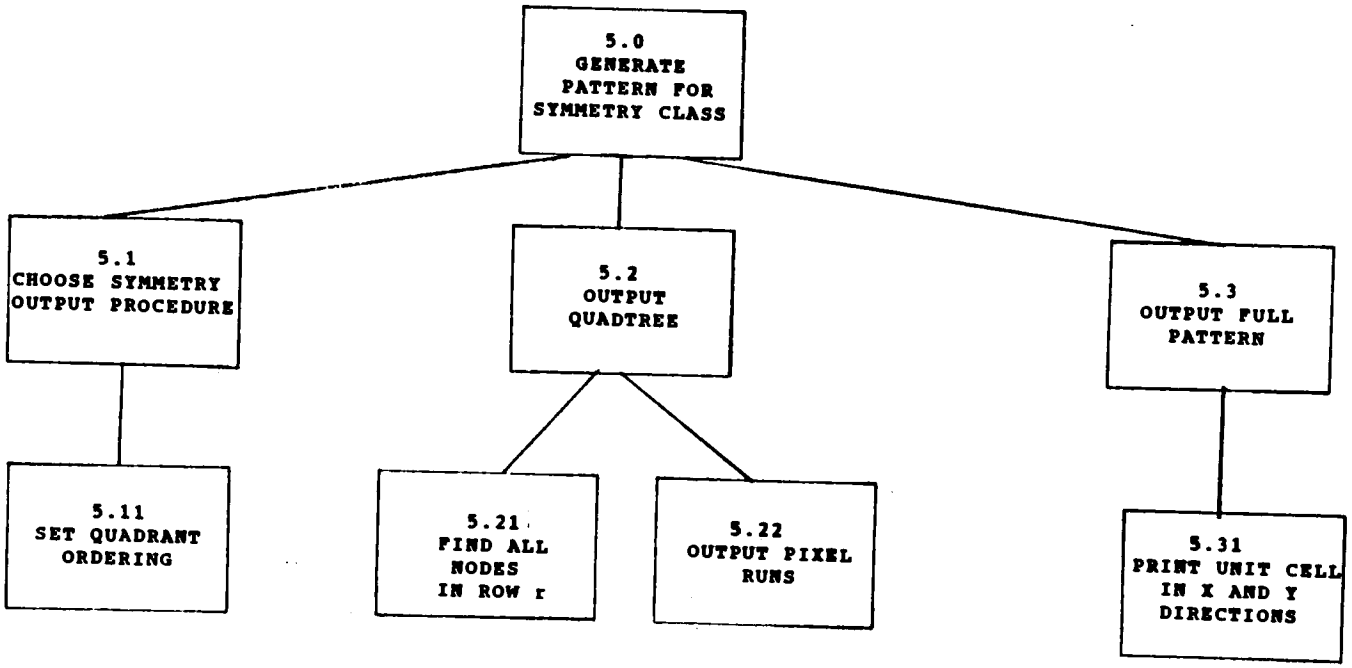



1.0
ACCEPT INPUT
PARAMETERS
1.1
GET SCREEN
PARAMETERS
1.2
GET SYMMETRY
PARAMETERS
1.3
GET NORMALIZATION
FLAG
1.11
GET INPUT
WIDTH
1.12
GET OUTPUT
WIDTH
1.21
GET SYMMETRY
TYPE
1.22
OPEN IMAGE
FILE
1.211
GET PRINTING
OFFSETS


2.0
SETUP
BITMAP
2.1
CREATE
BITMAP
2.2
CLEAR
BITMAP
2.3
READ IMAGE
INTO BITMAP
2.4
LOAD BITMAP
INTO LOCAL
ARRAY

3.0
NORMALIZE
BY
TRANSLATION
3.1
TRANSLATE
IMAGE TO (0,0)
3.2
FIND OPTIMAL
IMAGE POSITION
3.3
TRANSLATE IMAGE
TO OPTIMAL POSITION


4.0
CONSTRUCT
QUADTREE
4.1
POSTORDER
TRAVERSAL
OF PIXELS
4.2
BUILD QUADTREE
NODES


5.0
GENERATE
PATTERN FOR
SYMMETRY CLASS
5.1
CHOOSE SYMMETRY
OUTPUT PROCEDURE
5.2
OUTPUT
QUADTREE
5.3
OUTPUT FULL
PATTERN
5.11
SET QUADRANT
ORDERING
5.21
FIND ALL
NODES
IN ROW r
5.22
OUTPUT PIXEL
RUNS
5.31
PRINT UNIT CELL
IN X AND Y
DIRECTIONS

5. CONCLUSIONS

The intent of this application was to explore the feasiblity of using a hierarchical data structure in the form of a standard quadtree to generate symmetric patterns according to the rules of the plane symmetry groups. Of the seventeen distinct symmetry classes, it was possible to generate patterns belonging to nine of them (with some limitations, as described in Section 5.2). Specifically, the classes are p1, p211, pm, p2mm, p4, p4mm, p1g1, p2gg, and p4gm.

Figures 5.A through 5.I show examples of patterns generated from this application for each of the nine symmetry classes. The top left drawing on each page shows the original input image embedded in a 2**n by 2**n area, with the internal generating region outlined. The top right drawing is the unit cell generated for the symmetry class. The pattern at the bottom of the page represents the final propagation of the unit cell across the screen. All unit cells for these figures are assumed square.

In general, this approach is based on the ability to produce a symmetric pattern without the necessity of storing image data for the entire pattern. As stated in Section 2.4.1, the generating region of a symmetric pattern contains the picture information necessary to build the finished pattern according to the rules of a specified symmetry class. In this application, a slightly larger region (the size of the pattern's unit cell, containing the generating image within) is stored. This is done to facilitate construction of the unit cell, using a traversal-based routine.

The original image may be normalized with respect to translation before its quadtree is built, resulting in the construction of a minimal-cost quadtree.

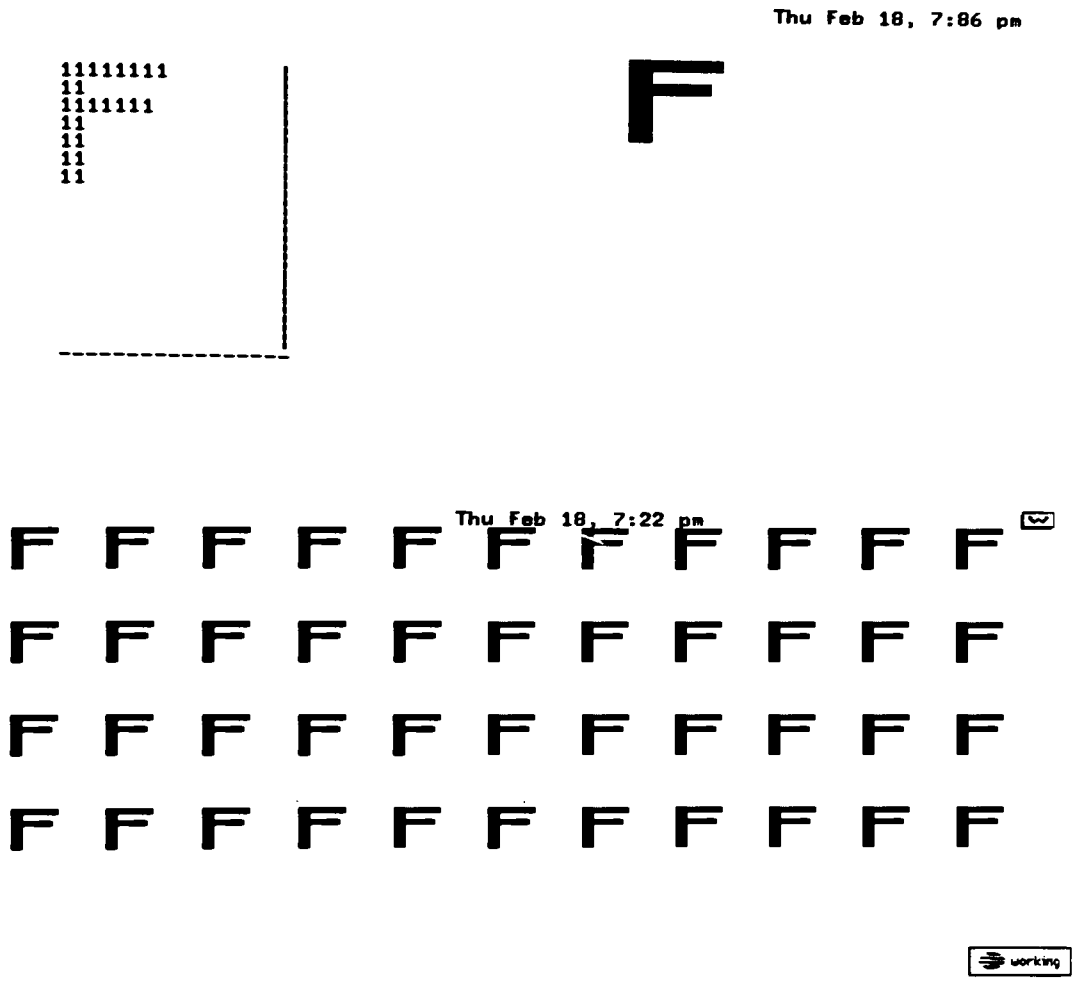


Figure 5.A. Generation of p1 symmetry class

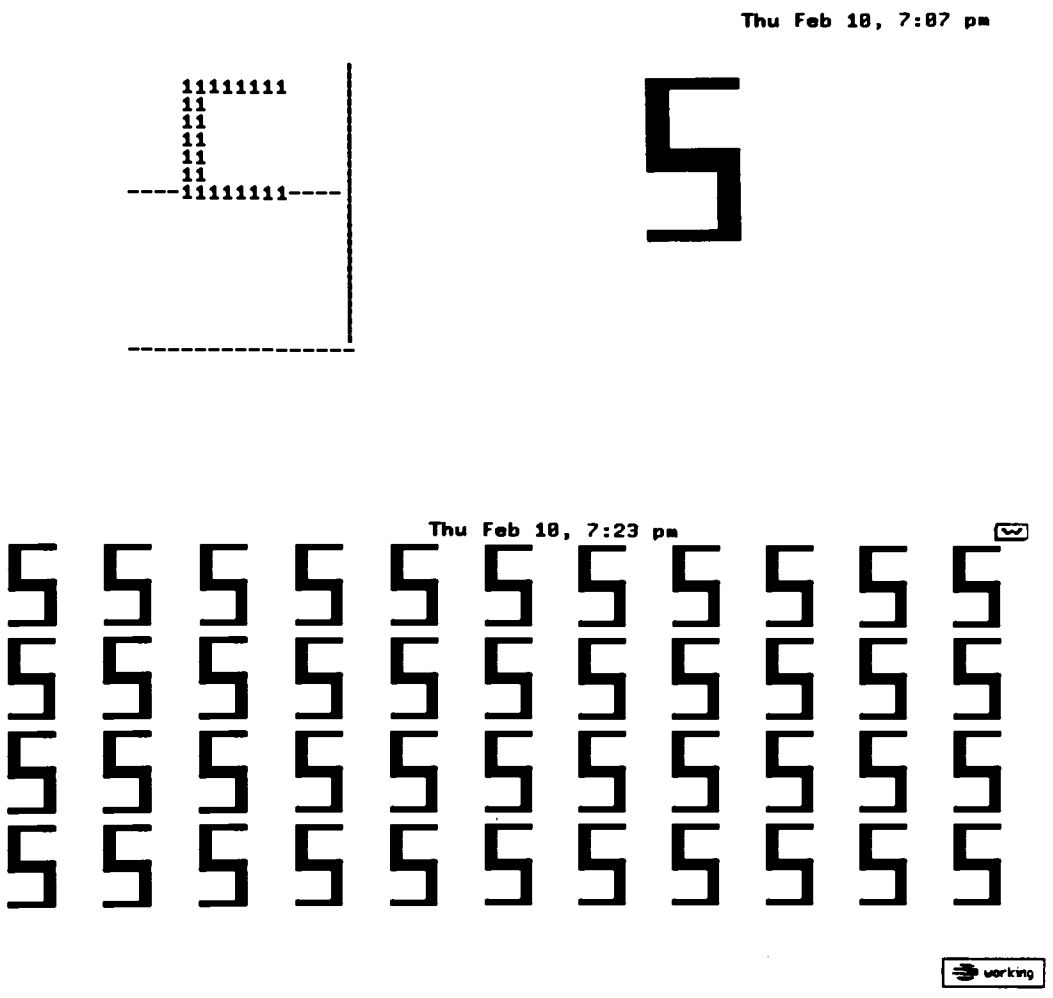


Figure 5.B. Generation of p211 symmetry class

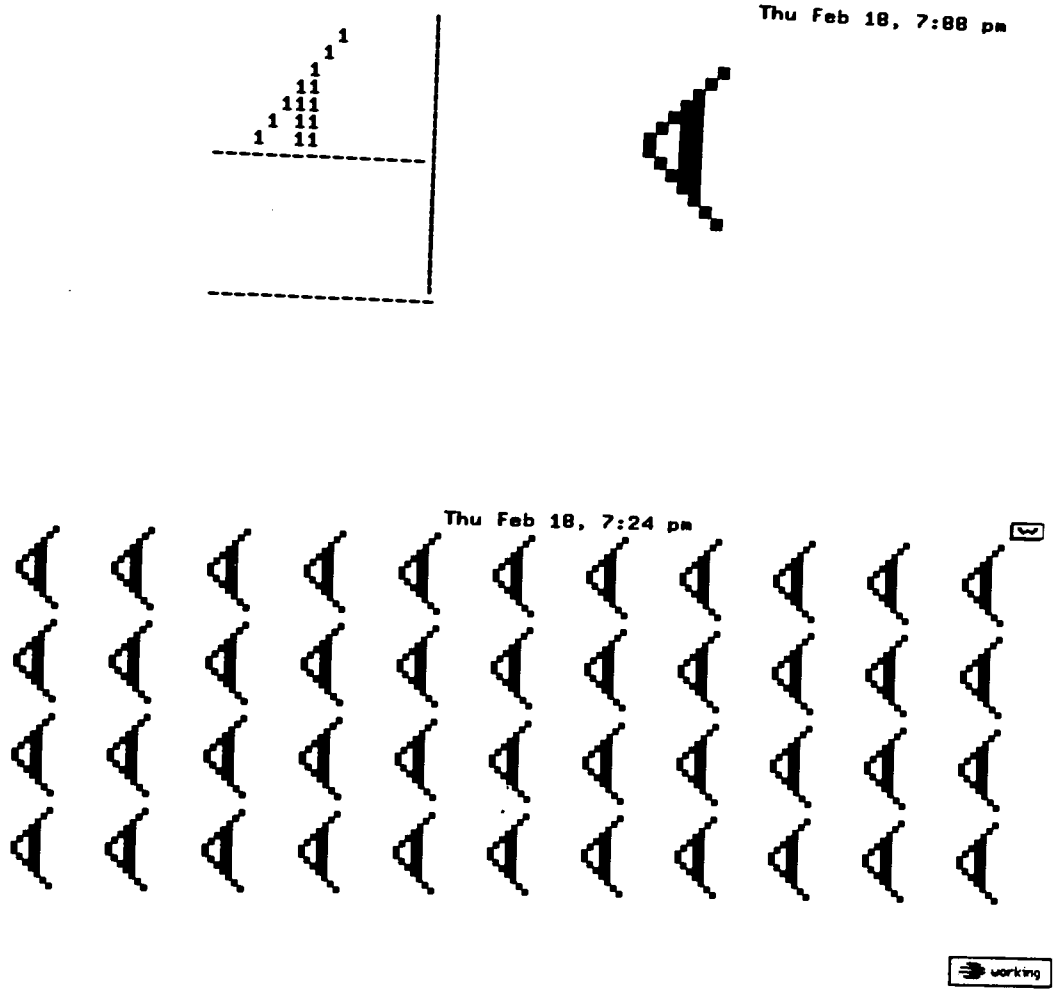


Figure 5.C. Generation of p1m1 symmetry class

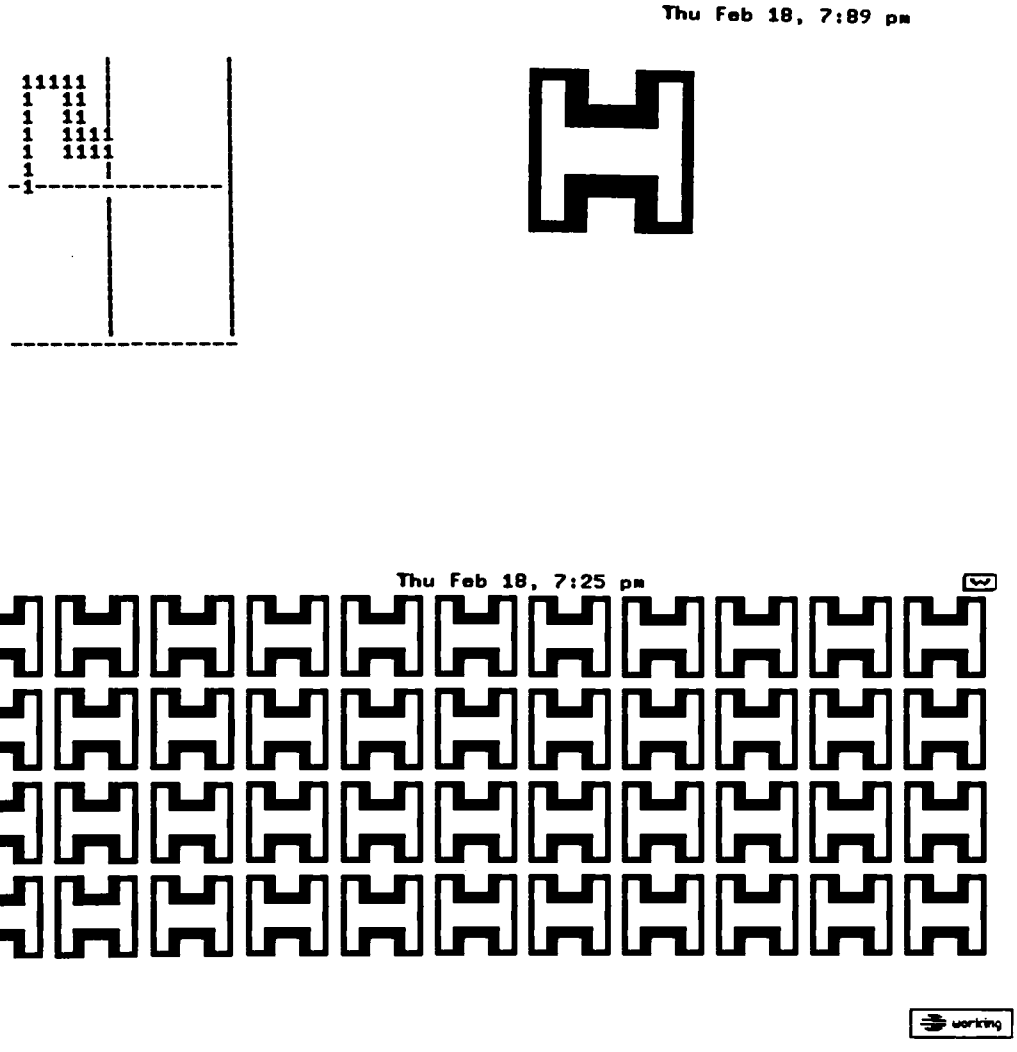


Figure 5.D. Generation of p2mm symmetry class

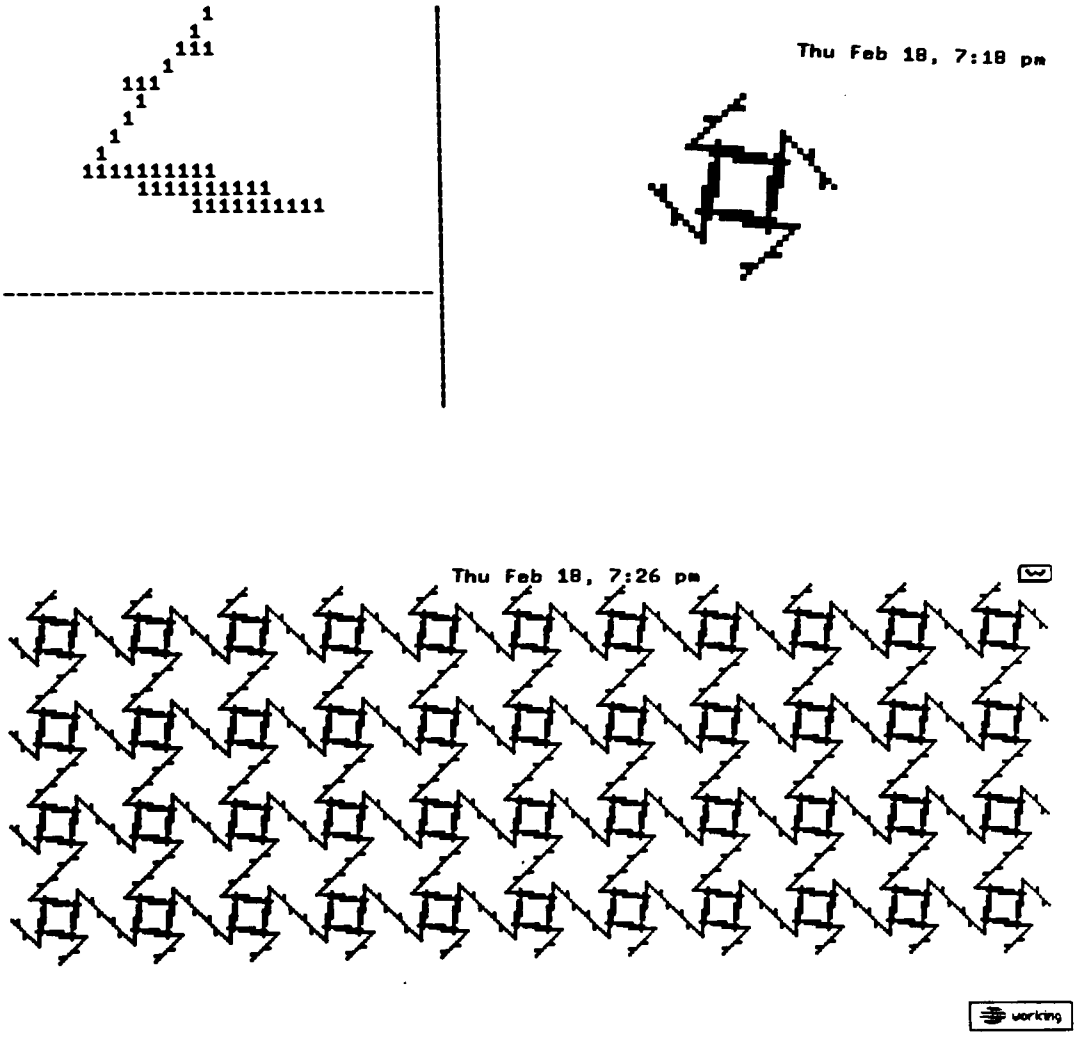


Figure 5.E. Generation of p4 symmetry class

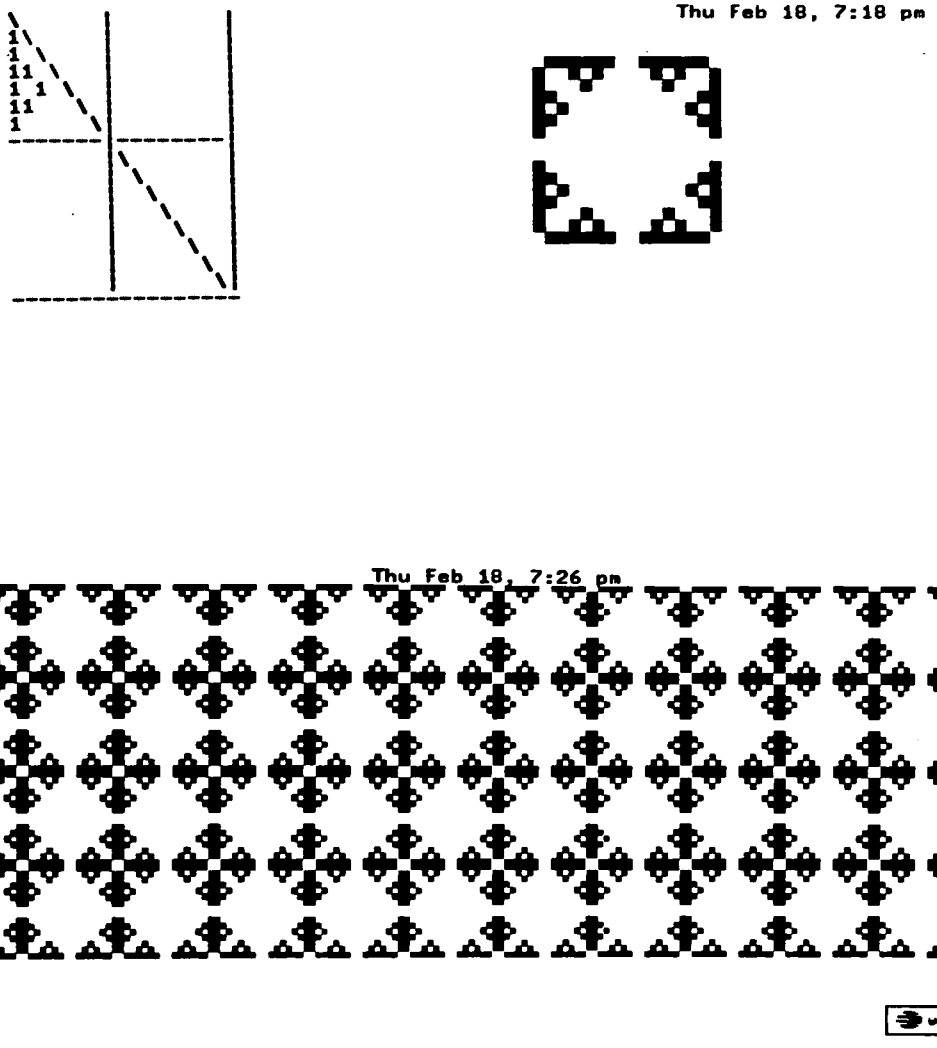


Figure 5.F. Generation of p4mm symmetry class

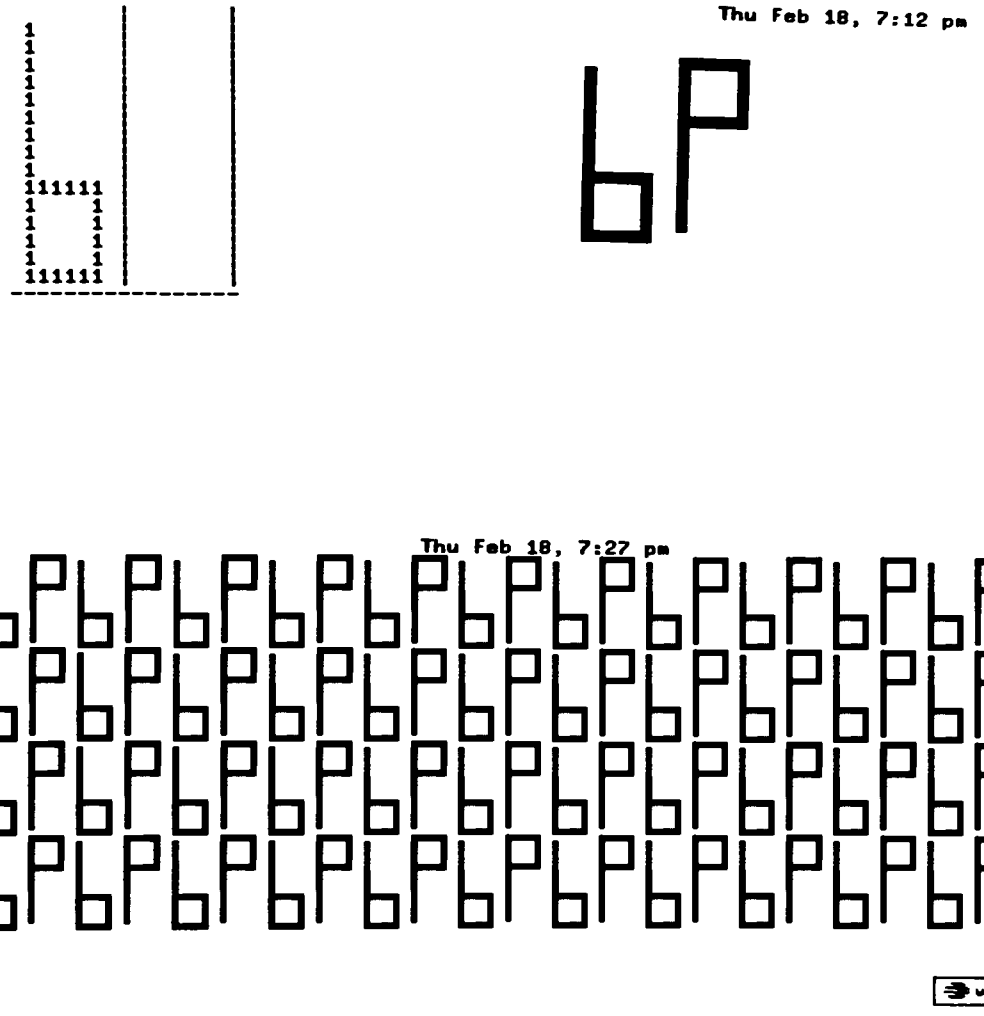


Figure 5.G. Generation of p1g1 symmetry class

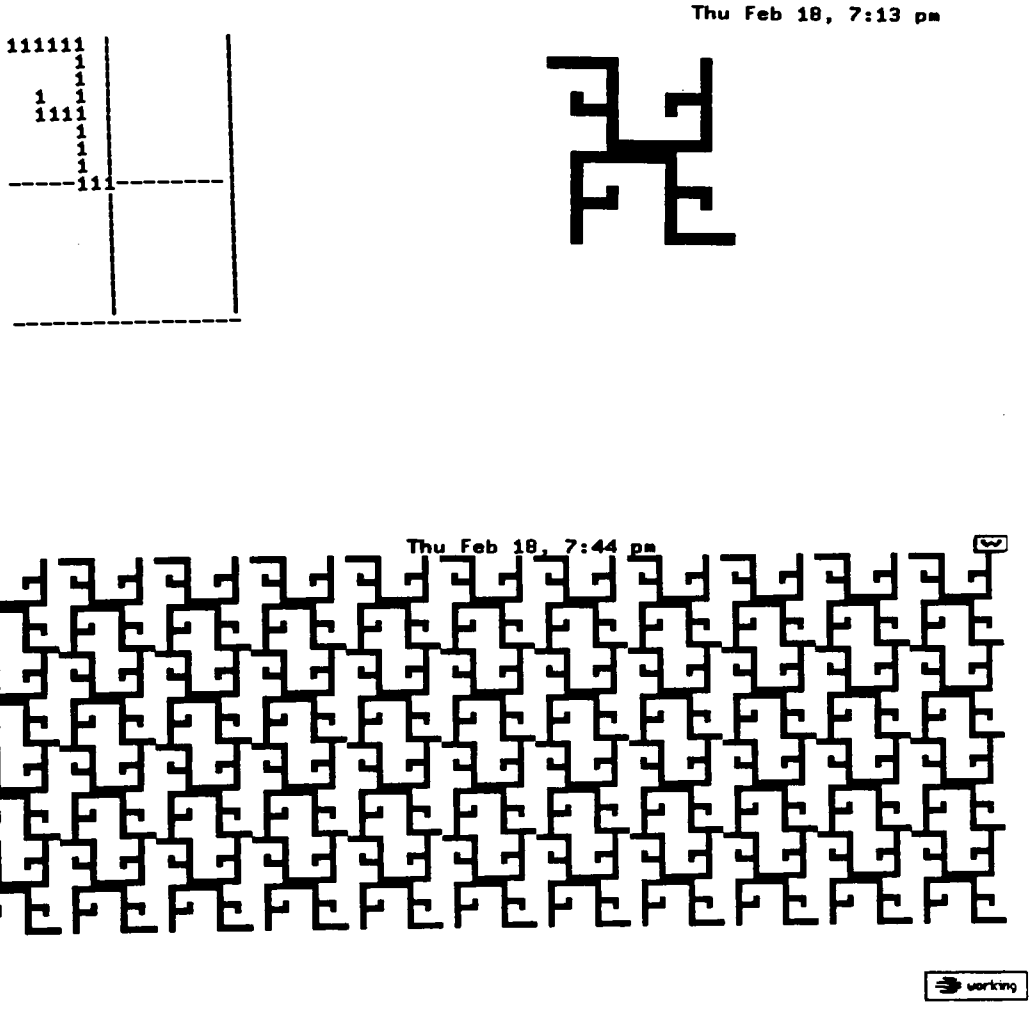
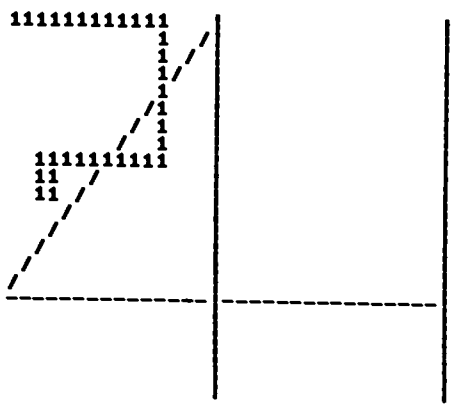
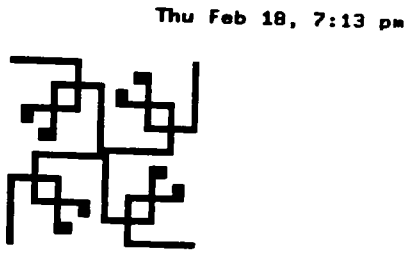


Figure 5.H. Generation of p2gg symmetry class





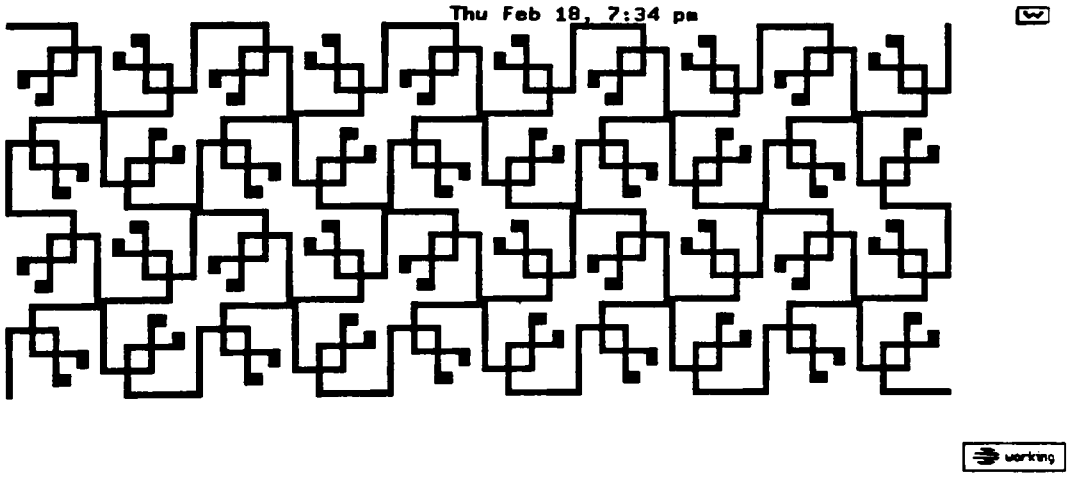


Figure 5.I. Generation of p4gm symmetry class

Initially, it should be noted that within one symmetry class, different choices for a generating region are possible. A unit cell for symmetry class p1m1, for example, can be defined (refer to Figure 2.M.) with either a horizontal or a vertical orientation, thus affecting the initial positioning of the generating region. If a generating region is desired that differs from one currently assumed in the application, it would be necessary to place the starting image in a different area of the original grid, and adjust the quadtree output operations accordingly; e.g., a change in orientation for a symmetry group may require reflection of the generating region over the vertical axis instead of reflection over the horizontal axis.

Specification of a 2**n by 2**n output size larger than the size of the initial image will produce a magnification of the unit cell, and thus of the final pattern. An original input size of 16 x 16, for example, with an output size of 64 x 64, will result in a size increase of the output image by three times the original. This is a demonstration of the scaling capability inherent to the data structure (see Figure 5.J). Starting with a small initial quadtree and drawing it to a larger scale will also reduce the storage requirements and execution time necessary to process the quadtree.

5.1. Problems Encountered and Solved

In general, modifications were necessary to resolve differences in the pixel coordinate scheme between the algorithms implemented, and the UNIX PC. On the UNIX PC, an image bitmap of size 256 x 256 begins in the

upper left corner as (0,0), proceeding across the row by (1,0), (2,0), (3,0), ... (255,0), and down the column by (0,1), (0,2), (0,3), ... (0,255). In many instances, this was not the orientation assumed in the algorithms; it was therefore necessary to modify the algorithms to conform to this scheme.

All algorithms implemented in this application assume an image coordinate system starting with (0,0) in the upper left corner, with the exception of the normalization by translation procedure, which assumes starting indices of (1,1). If normalization is desired, an extra initial translation of the image by one pixel right and one pixel down must be done prior to normalization in order to shift any image pixels out of the first row and column; the normalization procedure reserves the zero row and column for a special purpose. After the image is normalized to a position that will yield a minimal-cost quadtree, it is readjusted to the original coordinate system starting with (0,0) to satisfy the requirements of subsequent algorithms.

It is a feature of the normalization by translation algorithm to recognize the situation where the optimally placed image requires a smaller power of two grid size than the original. This occurs when the optimal image is completely contained within the smaller (2**[n-1]) quadrant. Unfortunately, choice of the next smaller grid size for an image would result in a quadtree of smaller size, which would in turn alter the effect of the symmetry output routines on the construction of the unit cell. The pattern's unit cell would not be constructed as expected; the output routines would rotate and reflect the generating region onto itself, rather than within the larger unit cell area. It is for this reason that the normalization algorithm was restricted to select the optimal transla-

tion conforming to the incoming image size only, even if this meant bypassing a slightly more efficient quadtree.

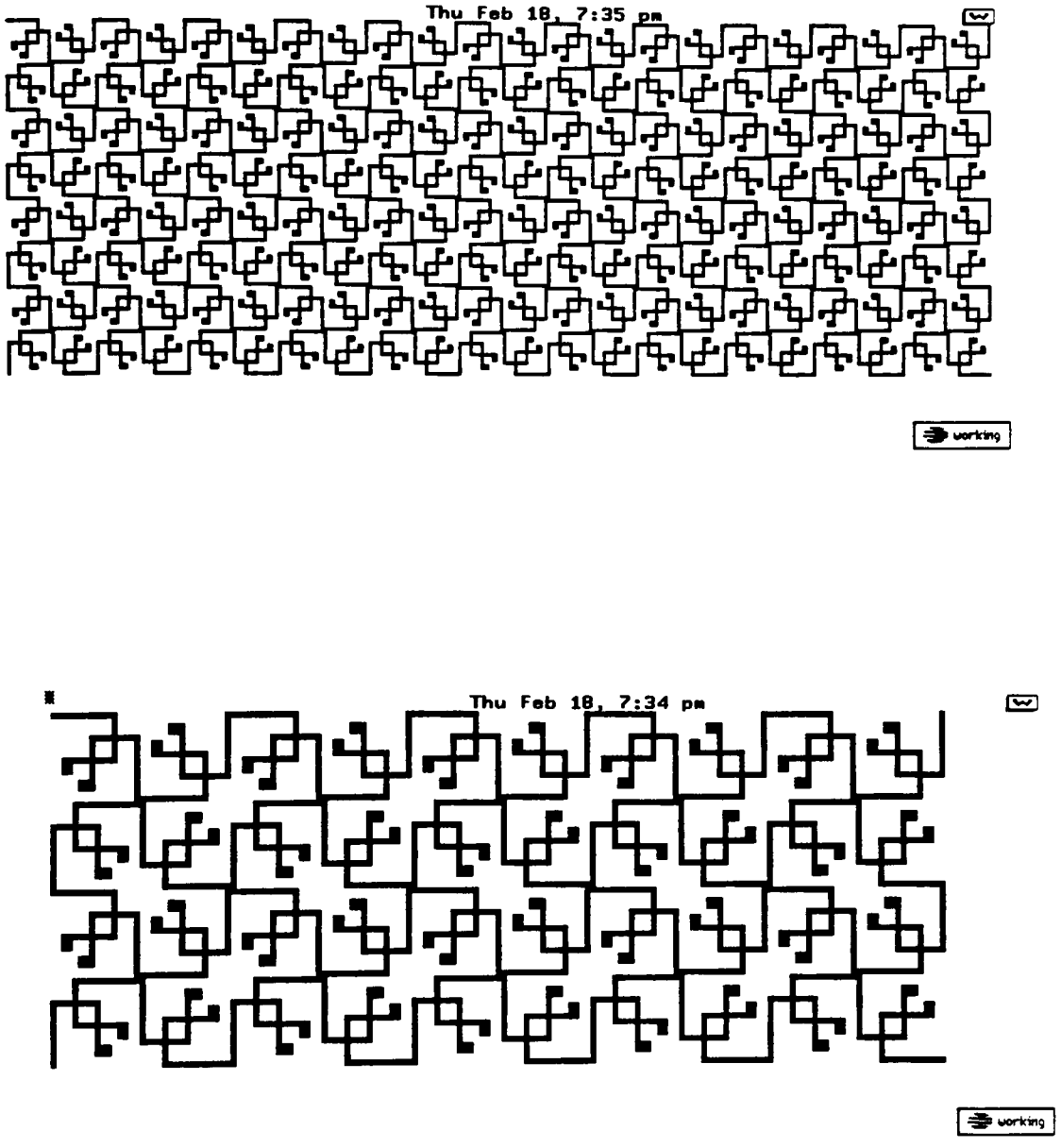


Figure 5.J. Example of quadtree scaling capability

5.2. Discrepancies and Shortcomings of the System

There were two basic shortcomings encountered during the propagation of patterns in this application. One involves the difficulty in representing non-square unit cell shapes with a square decomposition; the second involves applying operations that may alter the appearance of the final pattern in some way.

We now address the first problem. The current quadtree implementation expects a square (2**n by 2**n) pixel region to represent the unit cell of a pattern. Three of the nine symmetry classes used (p4, p4mm, p4gm) have a unit cell shape defined as exclusively square, leaving six classes (p1, p211, pm, p2mm, p1g1, p2gg) where square unit cells are only a subset of possible unit cell shapes. That is, the lattice shapes possible for these six classes are not restricted to the square. Classes p1 and p211 have a parallelogram defined as the unit cell region; the unit cell region for p2, p2mm, p1g1, and p2gg is defined to be rectangular. In order to compensate for the square quadtree's restriction to a square region shape, two printing "offsets" in the x and y directions were used with these six classes to enable the propagation of the resulting bitmap unit cell in various ways. Non-zero printing offsets in either direction will cause the bitmap graphics output routine to move to the right and down a specified number of pixels beyond the unit cell size before drawing each row of unit cells on the screen. The types of patterns constructed by this approach represent those where the unit cell area is assumed to be non-square, even though the image can only be defined within a square region. This implies that the portion(s) of a rectangle or

parallelogram lying outside the square can only be blank (WHITE). Figure 5.K illustrates an example of this situation. Two printing offsets of zero will propagate a square unit cell without these gaps (WHITE areas), as it is actually represented in the square quadtree-encoded generating region. Although the use of printing offsets allow the simulation of a greater variety of patterns for a symmetry class than would be possible without them, it will nonetheless be impossible to build a pattern that fills the screen without gaps between the unit cells if the unit cell is assumed to be non-square. This is demonstrated by the two patterns shown in Figures 5.L and 5.M.

Figure 5.K

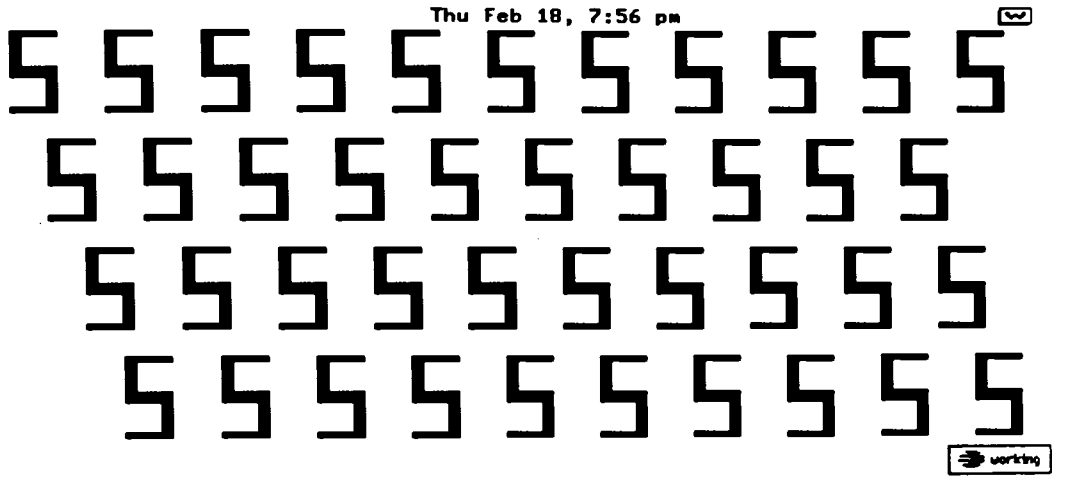
The second shortcoming involves problems encountered after the pattern's original starting image is normalized. Normalizing an image to produce a unique quadtree representation can be desirable, as in an application involving pattern matching. However, such normalization can be detrimental to the desired asthetic of a symmetric pattern. Normalization may result in an undesired loss of accuracy in the original generating region (normalization with respect to rotation and translation), or may cause the unit cell to be built in a manner that was not anticipated (normalization by translation only).

Normalization with respect to rotation and translation (Section

2.3.1.) was implemented for this application, but ultimately not used. A problem with normalizing an image using this method lies in the possible introduction of pixel-error, particularly as the image size used becomes smaller. Loss of accuracy can occur when the original image is rotated around its centroid by a principal angle that is not a multiple of 90 degrees before conversion to a quadtree. It will no longer be possible to reproduce the exact starting image from a quadtree that has been generated from a normalized image where pixel-error has been introduced. Since it may be desirable to maintain the asthetic quality of the original generating image, loss of accuracy could become a significant drawback. (Any pixel-error will be especially apparent if the image is output to a larger scale than the original.)

A further shortcoming with this method lies in the redrawing of the original image from the normalized quadtree; expensive multiplications of each pixel position by the sine and cosine of the principal angle are necessary in order to regain the original image after normalization.

Normalization by translation only (Section 2.3.2) was more successful for this application, and is the scheme actually used. It does not involve the type of image transformation (i.e. non-90 degree rotations) that may result in pixel-error. This method, however, is not without a disadvantage.



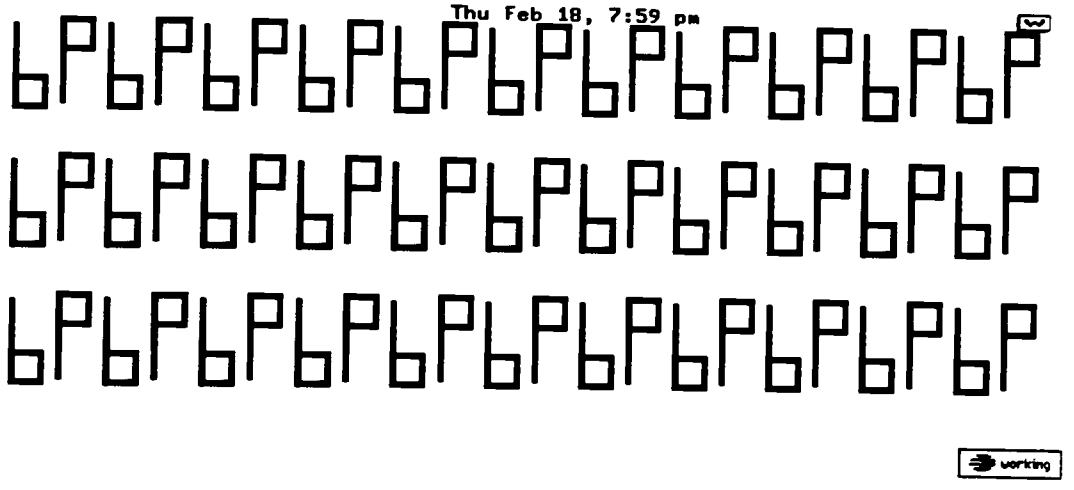
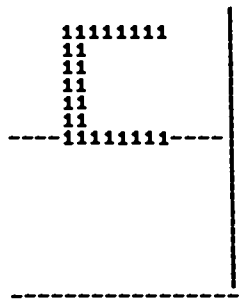


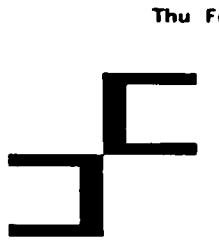
Figure 5.L Symmetry class p211 (above) simulated with parallelogram-shaped unit cell
Figure 5.M Symmetry class p1g1 simulated with rectangle-shaped unit cell

The possibility that the generating image will undergo a translation for its optimal quadtree to be found implies that the resulting output under a symmetry class may be shifted significantly from what was originally expected. This can be seen in Figures 5.N through 5.R. At the time of image input, the amount that the image will be shifted is not easy to predict, outside of the assurance that it will be moved a within an area of 2**(n-1) - 1 by 2**(n-1) - 1 pixels for an image of size 2**n by 2**n. The option of disabling normalization altogether was introduced for this reason. "Experimentation" with the effects of normalization on a particular image can be done, and it may be possible to adjust the original input image in some way, or choose to leave it unnormalized, if the resulting pattern differs greatly from what was envisioned.

It is for the benefit of this normalization algorithm that the generating region for a pattern is required to include the NW quadrant (See Section 2.3.2). If normalization is to be bypassed, this requirement can be relaxed, i.e. different generating regions can be chosen, with appropriate changes made to the symmetry output routine calls. (See Section 5.)

With larger images, the inability to normalize the image because of possible alteration of the output pattern could be overly expensive in terms of storage, and of the performance of subsequent quadtree operations. Small quadtrees do not benefit greatly from minimizing the number of nodes. It is for this reason that an alternative quadtree representation (Section 6) may be a preferable overall approach to improve storage and efficiency.





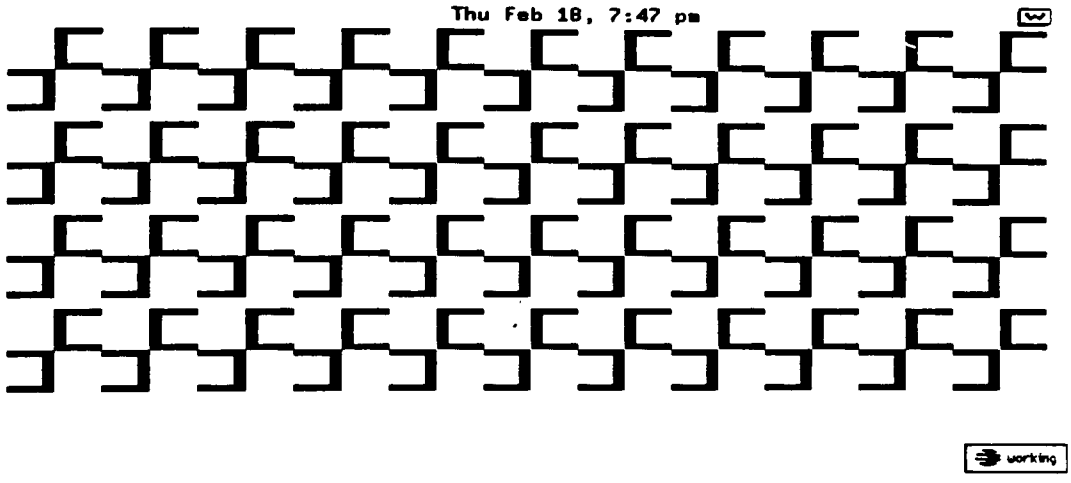


Figure 5.N. Generation of p211 symmetry class (normalized)

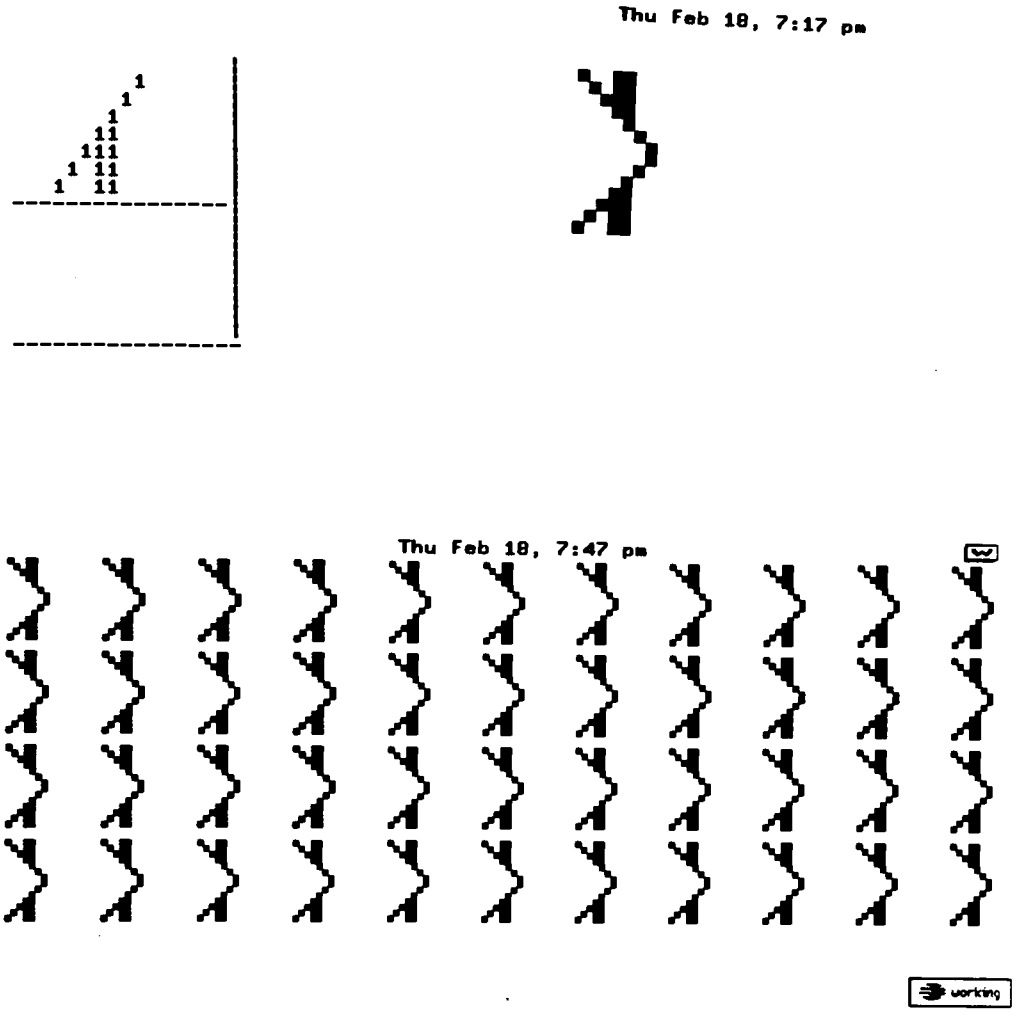


Figure 5.0. Generation of p1m1 symmetry class (normalized)

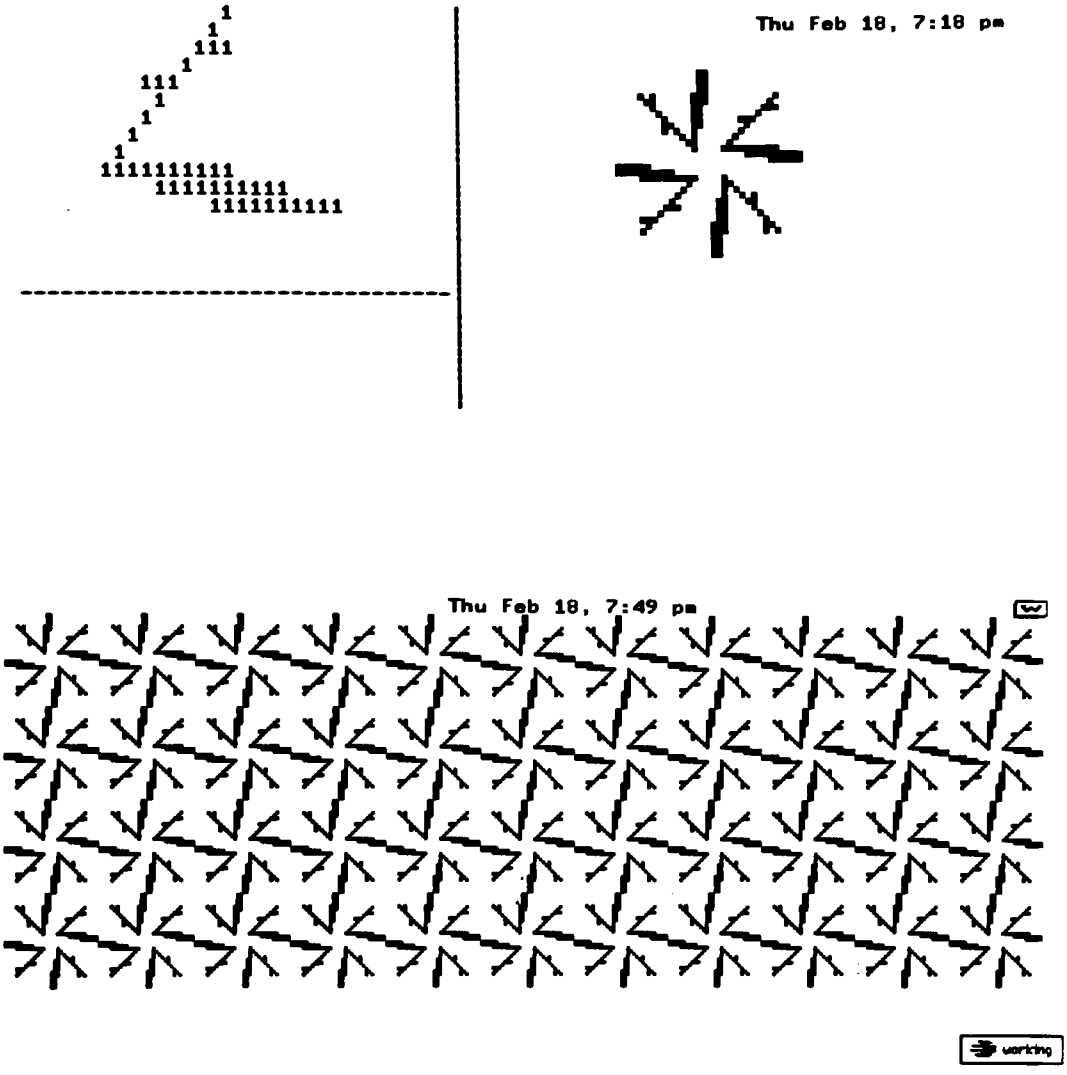
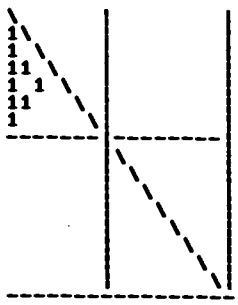
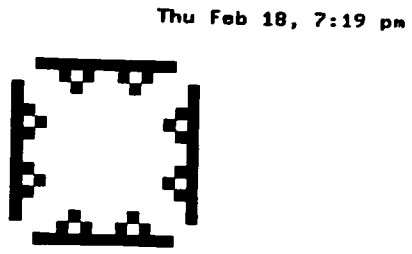


Figure 5.P. Generation of p4 symmetry class (normalized)





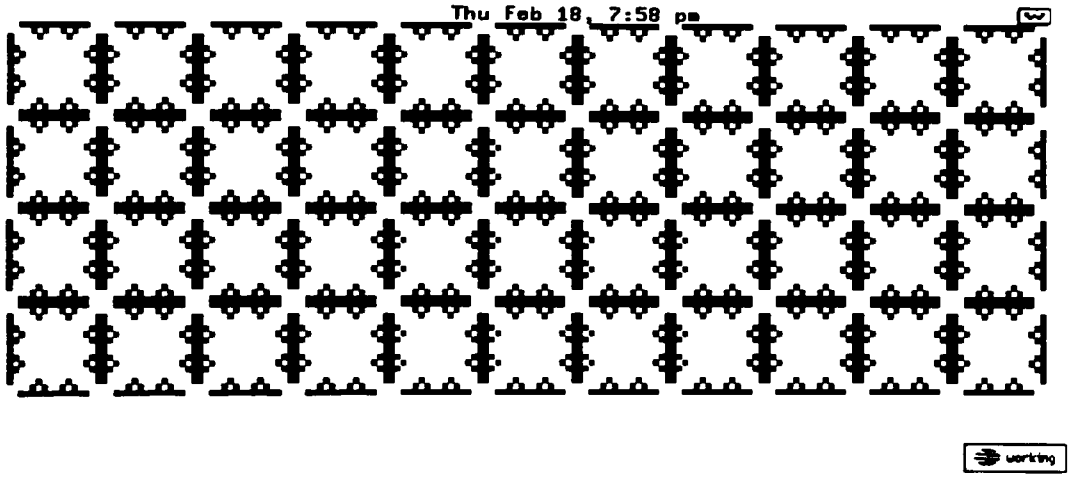
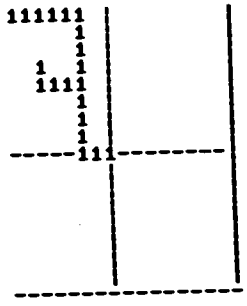
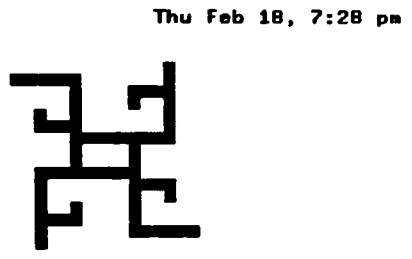


Figure 5.Q. Generation of p4mm symmetry class (normalized)





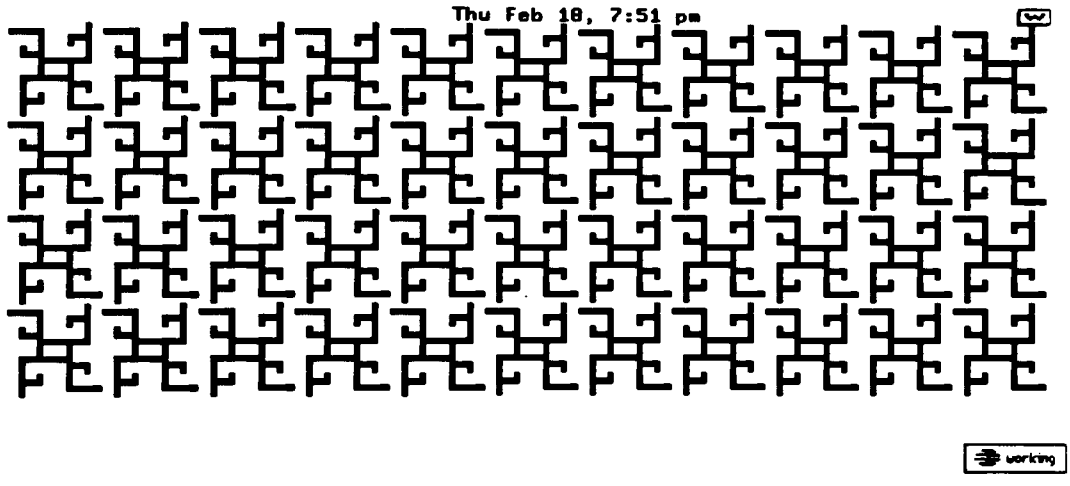


Figure 5.R. Generation of p2gg symmetry class (normalized)

5.3. Suggestions for Future Extentions

The characteristics of hierarchical decomposition of image data as applied to the propagation of plane symmetry groups seem to suggest experimentation in several areas.

5.3.1. Triangular Decomposition

Occurrences of three and six-fold rotation (120 and 60 degree rotation, respectively) are present in five of the seventeen symmetry classes. Such angles of rotation are not naturally expressed in terms of the square decomposition scheme, which is only capable of performing rotations in multiples of 90 degrees. As was mentioned in Section 2.2, recursive decomposition by equilateral triangles is considered the technique next most suited for image processing, after square decomposition. Recursive division of an image using equilateral triangles has basic parallels to square decomposition, noting exceptions in orientation and adjacency properties. It appears to follow that some of the reflection and rotation properties present in square decomposition may be present in this triangular method, and may be useful in generating some of the symmetry classes not produceable so far. It would be of interest to attempt decomposition by equilateral triangles (or perhaps one of the other triangular methods described in [GARG87]), and explore the effects of quadrant reordering on the generation of unit cells that involve non-90 degree rotations.

5.3.2. Non-Regular Decomposition

The problem of representing non-square unit cell shapes in this application, (i.e. parallelogram, rectangle) with a square decomposition method suggests that the original decomposition shape needs to be altered in some way. The regular decomposition scheme used in this application is based on the division of the picture into equal squares at each level, with the area boundaries fixed ahead of time. A decomposition that is not regular, e.g. involving rectangles of arbitrary size, is an alternative to this. Such a method may require less space to represent an image, but would be governed by the characteristics of the incoming image, meaning a search would be required to determine the optimal partition points [SAME84b].

For symmetry classes whose unit cells are not necessarily square, (i.e. parallelogram-shaped or rectangular), the partitions might be chosen according to the shape of the unit cell of the symmetry group. A limit on the resolution (above individual pixel level) would have to be determined prior to decomposition, since a rectangle or parallelogram can not be decomposed indefinitely into areas of the same shape. (See Section 2.1.) Each quadtree node would represent an area the shape of the unit cell, where each successive level would represent areas of 1/2 the size of the previous level, until the resolution limit is reached. It appears that the quadtree traversal operations used in the current application would hold for this scheme, but a reduction in the detail of the original image must be expected.

5.3.3. Symmetry Detection

Knowing that it is possible to detect simple directional symmetries within a hierarchical image structure (Section 2.4.2.), it follows that identification of plane symmetry classes might be made by applying symmetry detection algorithms to quadtree-encoded patterns in some way. Visual examination of unit cells for patterns restricted to the nine symmetry classes generated in this application indicates that in many cases, the unit cells have unique characteristics in terms of the directional symmetries previously mentioned. The table below shows combinations of directional symmetries found in unit cells for each of the nine classes.

	HORIZ	VERT	LEFT-DIAG	RIGHT-DIAG	90 DEG ROTATIONAL	180 DEG ROTATIONAL	270 DEG ROTATIONAL
p1							
p211						X	X
* p1m1	X						
p2mm	X	X				X	X
p4					X	X	X
p4mm	X	X	X	X	X	X	X
p1g1							
p2gg						X	
p4gm					X	X	X

* p1m1 unit cell may have vertical symmetry only, depending on the orientation.

Unfortunately, there are cases of ambiguity in this table; unit cells for classes p1 and p1g1 contain no directional symmetry, and classes p4 and p4gm are both identified by having 90, 180 and 270 degree rotational symmetries in their unit cells. If sample unit cells from the remaining

eight symmetry classes are examined along with these nine, more ambiguities can be found. A more detailed analysis of what makes each class unique seems necessary to solve this problem; if these attributes can be represented and detected within the quadtree structure, perhaps a general two-dimensional plane symmetry detection algorithm would be successful.

5.3.4. Pattern Modification

An extention of the current application might involve the ability to interactively modify the quadtree-based generating region (or unit cell) of a pattern, and propagate the changes throughout the full pattern based on this local modification. Ideally, this implies the ability to locate the subtree(s) representing the section of the pattern to be modified, rebuilding only those subsets of the quadtree, and propagating only the parts of the pattern that have changed. The data structure's ability to focus on "areas of interest" in an image appears to make this possible. A scenario could be described as follows, given the current scheme of the application:

1) A magnified copy of the generating region would be displayed, with the internal square partitions outlined. (The partitions should stop somewhere above the level of individual pixels, for the sake of clarity on the screen.)

2) A method for interactively choosing the partition(s) to be modified, along with a method of describing the modification(s) would have to be devised.

3) Once the changes are indicated, the program would traverse the original quadtree to locate the subtree(s) and rebuild the subtree(s) according to the new picture information.

4) It would be necessary to modify the quadtree output routine to write pixel runs to the existing bitmap corresponding only to those rows in the image that have been modified, implying that this row information must be available.

5) Once the new unit cell has been constructed, the current method of bitmap propagation would have to operate as is, i.e., overwriting the entire pattern, unless the modified area(s) were written to new (smaller) bitmaps that could be superimposed on the existing pattern at the proper positions on the screen.

6. IMPROVEMENTS TO THE QUADTREE REPRESENTATION

As was mentioned previously (Section 2.2), the standard quadtree representation employed here involves a significant amount of overhead in the form of pointer data, and in the storage of internal GRAY nodes. Some of the alternative hierarchical schemes of current interest are now described. An extensive overview of hierarchical image representations is presented in [SAME84b].

6.1. Linear Quadtree [GARG82]

The linear quadtree is a pointerless hierarchical representation that takes the form of a sorted list of base-four integers. It is characterized by the storing of BLACK tree nodes only. This is possible because the representation for a node implicitly contains the path from the root to the node, rather than relying on the storage of explicit pointers. The location of WHITE nodes can be inferred from the list of BLACK nodes.

The space and time complexity of the linear quadtree depends only on the number of BLACK nodes. This scheme produces a storage savings of at least 66% over the standard quadtree, and higher than 90% when a standard quadtree approaches its worst-case space requirements [GARG82]. Since there is no reduction in the number of BLACK nodes over the standard quadtree structure, space requirements in terms of the number of BLACK nodes will remain unchanged [SCOT and IYEN-86].

Adjacency information (i.e. the location of neighboring nodes) in

the four principle directions is naturally expressed in a node's encoding scheme, without the need of traversing explicit pointers [GARG82,p.910].

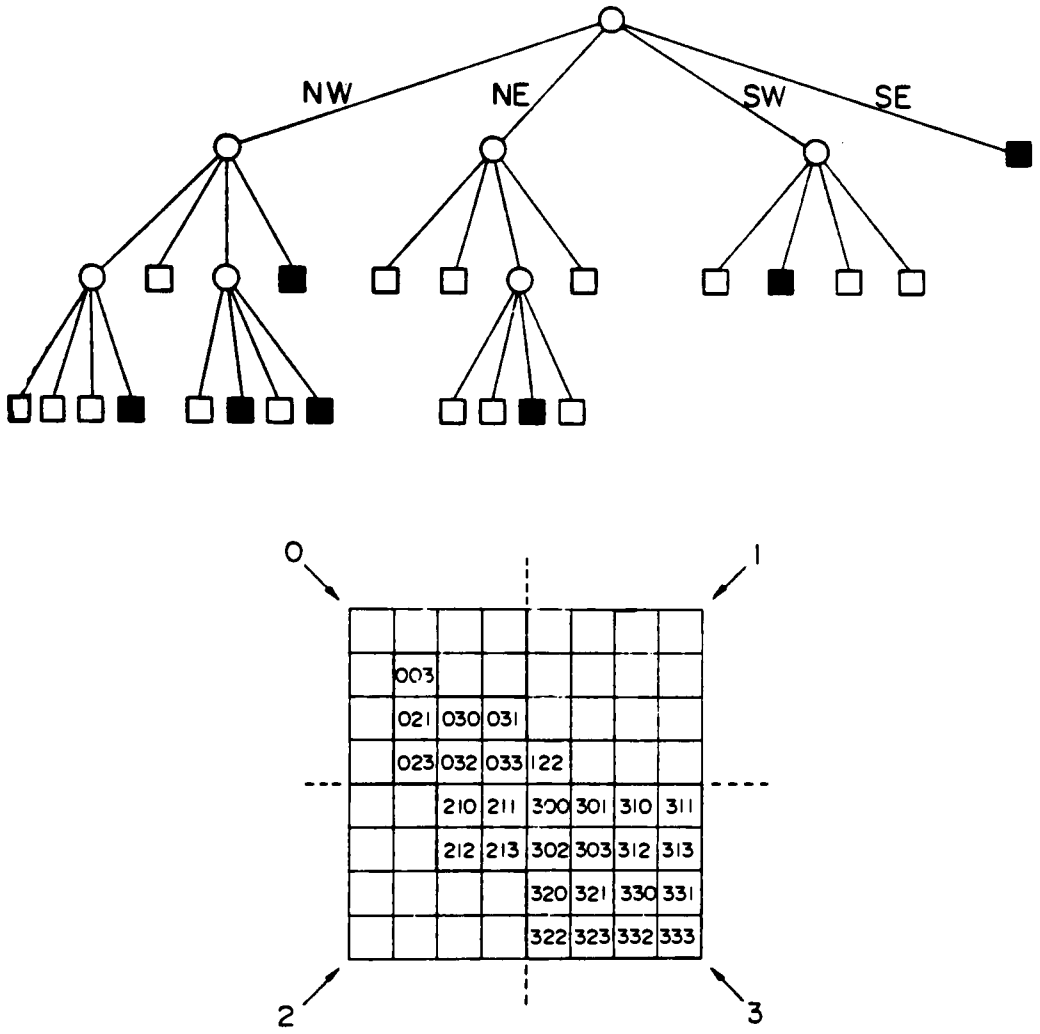


Figure 6.A. A standard quadtree (above) with its linear quadtree representation [GARG82]

In general, each image quadrant is encoded using the convention of the NW quadrant as 0, the NE quadrant as 1, the SW quadrant as 2, and the SE quadrant as 3, as with the standard quadtree. Each BLACK pixel is expressed as a weighted sequence of base four integers such that the digit of weight 4**(n - i) (1 <= i <= n) identifies the quadrant to which the pixel belongs at the ith level of subdivision. This is shown in Figure 6.A. Stated another way, each BLACK pixel (x,y) in the original bitmap is mapped to a base-four integer that will show the successive quadrants to which (x,y) belongs. A code of 123 means that the pixel is in the NE quadrant at the first level of recursion, the SW quadrant at the second, and in the SE quadrant at the last subdivision. Encoding a BLACK pixel therefore has a time complexity proportional to the image's resolution [GARG82].

After all BLACK pixels have been encoded in this way, their integer codes can be sorted and stored as a list. It will then be possible to condense the list, replacing any four codes that represent the same region with one code. If four pixels have representations that differ by only the last digit, the four codes can be eliminated and replaced by the first n-1 digits, concatenated with a "don't care" marker. We will call such a marker 'X' temporarily. BLACK pixels 310,311,312,313 can be replaced in the list by 31X without loss of information, as they represent the same homogeneous quadrant. Continuing in this way, if 30X, 31X, 32X, 33X exist in the list, they too may be replaced by one node represented as 3XX. The final result of this process will be a list that is still sorted, providing marker 'X' is assigned an integer value greater than 3. This final list is referred to as the linear quadtree of an image, and corres-

ponds to a postorder traversal of the BLACK nodes.

The absence of explicit pointers does not affect the ability of node adjacency information to be derived conveniently. Given any code within a linear quadtree, it is possible to locate a neighboring node in the four principle directions by simple arithmetic algorithms. [GARG82, p. 907].

The linear quadtree structure is particularly convenient if set operations are to be performed. Operations such as union and intersection are done by merging and condensing two (or more) sorted linear quadtree lists. The execution time of such procedures is proportional to number of BLACK nodes in the two (or more) linear lists. Algorithms for the union, intersection, and difference functions for linear quadtrees are presented and analyzed for correctness in [BAUE85].

This data structure involves a reduction in traversal flexibility, as the sorted integer list represents a traversal of the quadrants in a particular order. Performing image operations of the type used in this application (rotation, reflection) suggest the necessity of remapping the digits in each base-four integer code and resorting the resulting list [GARG83]. Since this application relies heavily on the ability to traverse the data structure in eight different orders easily, the linear quadtree may not be a natural alternative to the standard quadtree method.

6.2. Quadcodes

[LI and LOEW-87] present a hierarchical structure whose basis is similar to that of the linear quadtree, called the quadcode representation.

The quadcode scheme differs from that of the linear quadtree in that quadcodes are viewed as a direct description of the image, rather than using the idea of a tree structure as an interpretive medium. Image operations become "calculations" performed directly on the quadcodes, rather than by traversal of a tree representation of an image.

6.3. One-To-Four Structure [STEW86]

The attractiveness of the one-to-four structure (or just one-to-four) lies in the ability to reduce the amount of pointer information stored, while at the same time preserving the traversal flexibility that may be lost in a linear list method.

Rather than storing a pointer to each of a node's four children as in the standard quadtree, one pointer is kept to point to a block of records representing the node's subordinate quadrants (thus the name one-to-four). The basic data structure can be described as a record containing five fields. Four of the fields contain node descriptor information for the node's children in a specified order (0, 1, 2, 3, or NW, NE, SW, SE), and the fifth is the pointer to a group of records representing the node's children, also in the sequence NW, NE, SW, SE. (This pointer will always point to the first child's record.) The data structure is shown in Figure 6.B. If it is assumed that a one-to-four represents a full quadtree, the amount that a son's record is offset from the pointer to it can be expressed as

POINTER + [RECORDLENGTH * (QUADRANT 0-3)].

Unfortunately, this equation will not hold for any one-to-four structure

that does not represent a full quadtree. Described in another way, if nodes 1 and 2 are leaves, and nodes 0 and 3 are non-terminal, then node 0's son record will be the first in the list, and node 3's will be the second, rather than the fourth. There are several possible ways to compensate for this [STEW86]:

1) Inspect all preceding quadrant descriptors to find the number of son's records that actually exist in front of the desired record. The one-to-four will be at its most compact using this method, but access time is added to traversal operations.

2) Pad the one-to-four structure with dummy records as place holders for any missing records. Records in blocks of four are guaranteed. This requires more space, but traversal time will be optimal.

3) Store extra information with each pointer to keep track of the offset to each son's record. Storage requirements are increased slightly, but storage for dummy records becomes unnecessary.

For this application, the one-to-four appears to be a good candidate for an alternative to the standard quadtree data structure. The traversal flexibility necessary to perform reflections and rotations is maintained, while the number of node pointers necessary has been reduced significantly (each pointer in the one-to-four replaces up to sixteen pointers in the standard quadtree) [STEW86].

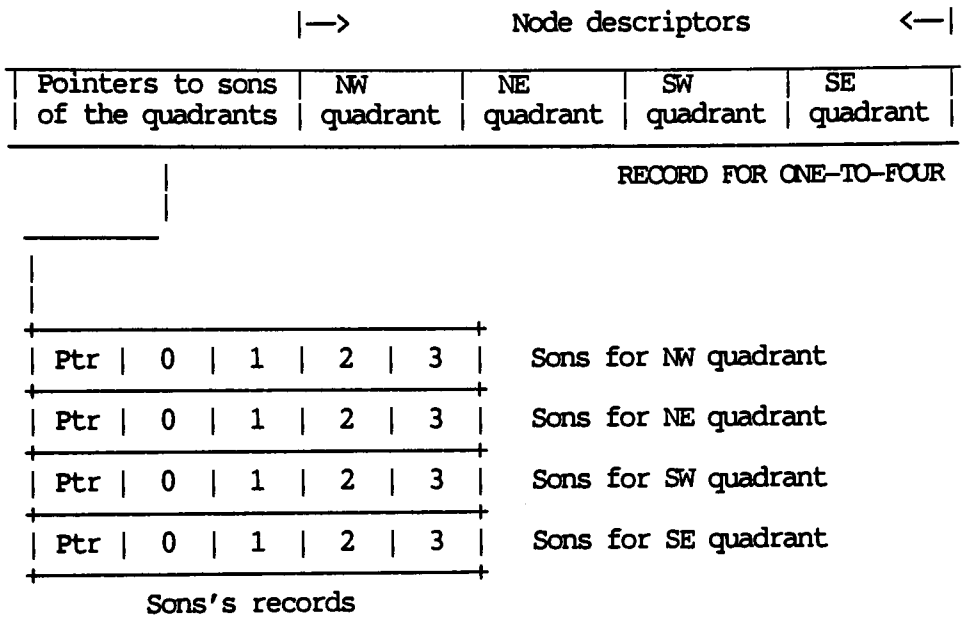
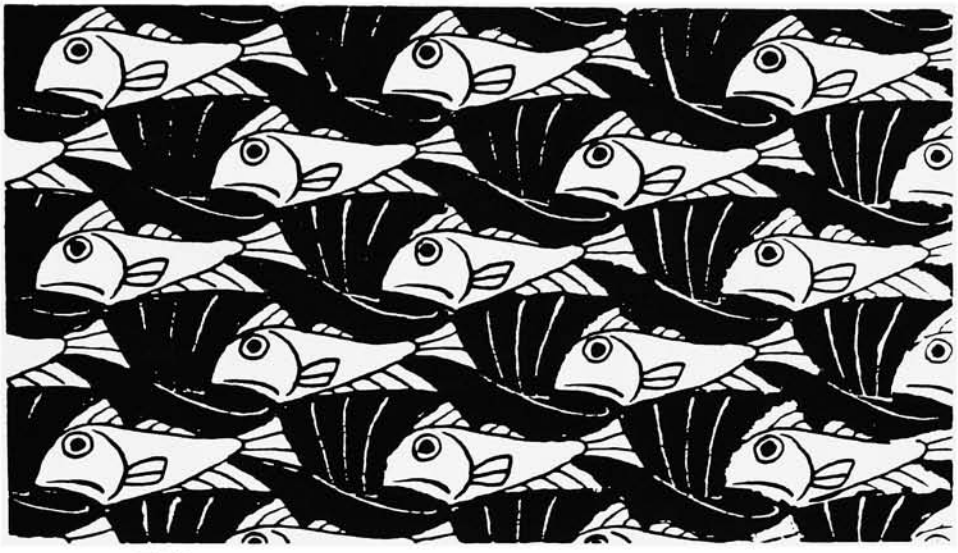
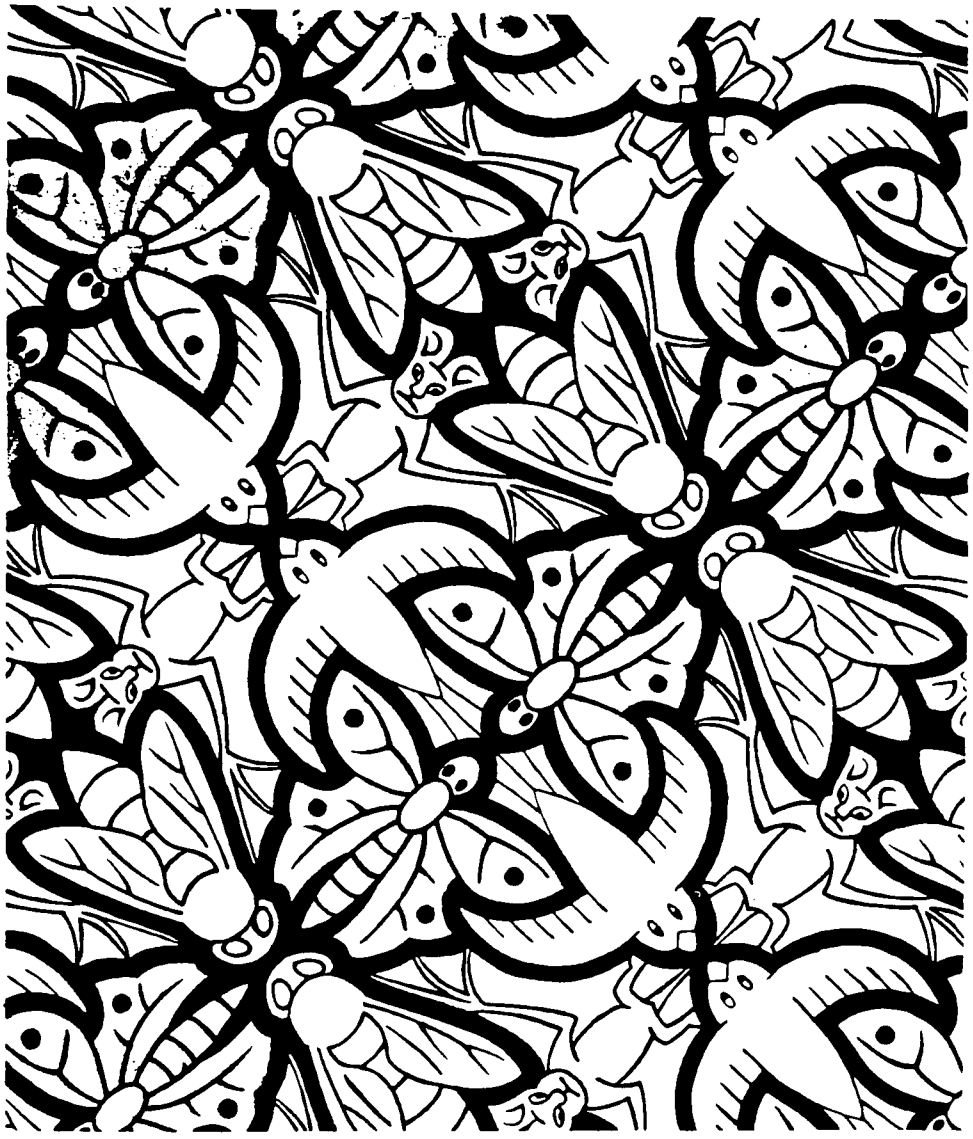


Figure 6.B. Basic construct of one-to-four structure [STEW86]

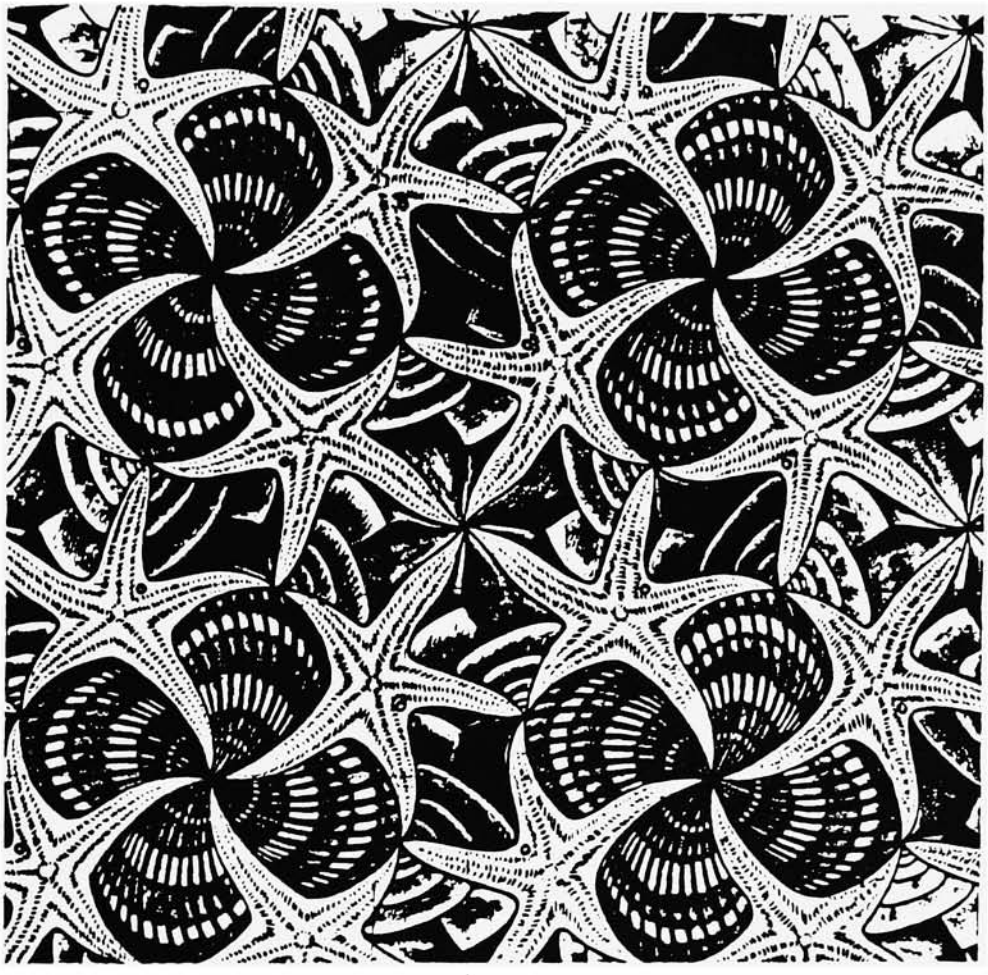


Appendix I [SCHA78]
Example of p1 symmetry class (above) — M. C. Escher
Example of p1g1 symmetry class (below) — M. C. Escher

Appendix II [SCHA78]
Example of p211 symmetry class — M. C. Escher

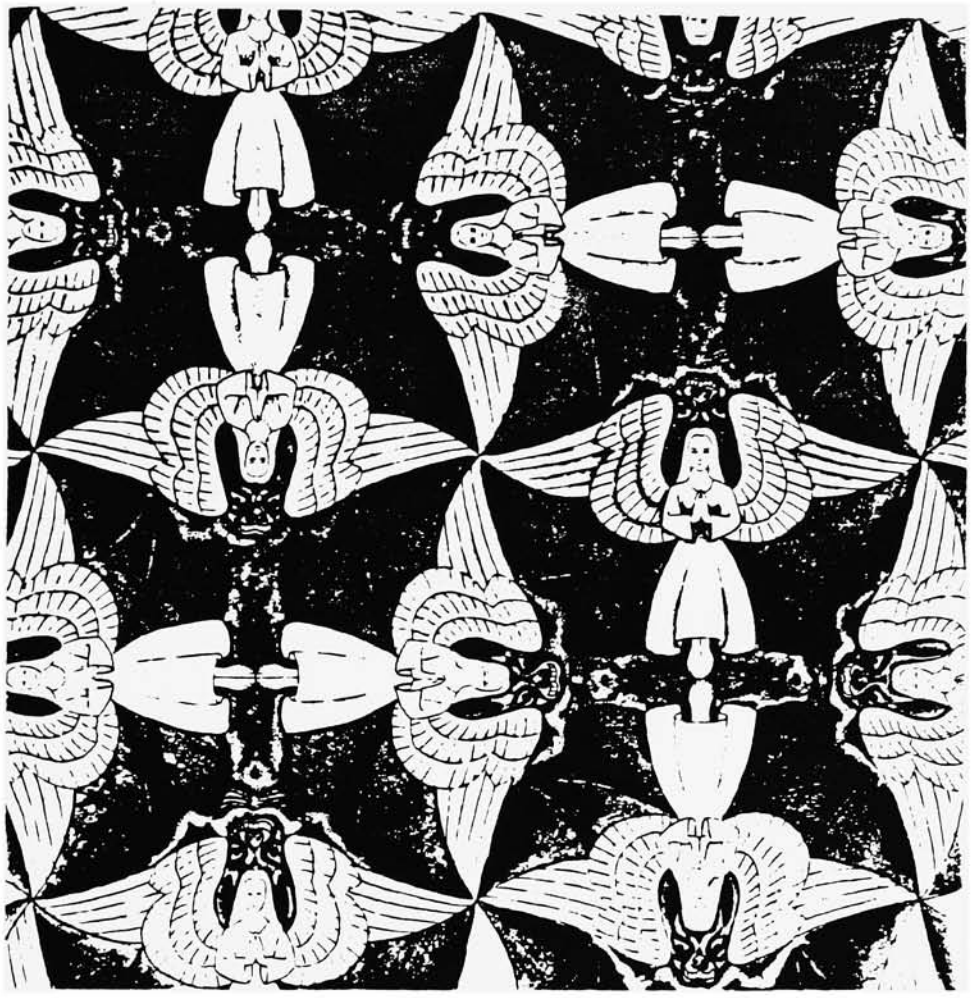


Appendix III [SCHA78]
Example of p2mm symmetry class — M. C. Escher



Appendix IV [SCHA78]
Example of p4 symmetry class — M. C. Escher

Appendix V [SCHA78]
Example of p2gg symmetry class — M. C. Escher



Appendix VI [SCHA78]
Example of p4gm symmetry class — M. C. Escher

BIBLIOGRAPHY

[AHUJ83] Ahuja, Narendra, "On Approaches to Polygonal Decomposition for Hierarchical Image Representation", Computer Vision, Graphics, and Image Processing, 24, (1983), pp 200-214.

[ALEX and KLIN-78] Alexandridis, Nikitas, and Klinger, Allen, "Picture Decomposition, Tree Data-Structures, and Identifying Directional Symmetries as Node Combinations", Computer Graphics and Image Processing, 8, (1978), pp 43-77.

[BAUE85] Bauer, Michael A., "Set Operations on Linear Quadtrees", Computer Vision, Graphics, and Image Processing, 29, (1985), pp 248-258.

[BELL83, et al.] Bell, S.B.M., Diaz, B.M., Holroyd, F., Jackson, M.J., "Spatially Referenced Methods of Processing Raster and Vector Data", Image and Vision Computing, Vol 1, No.4, November 1983, pp 211-220.

[BURT84, et al.] Burton, F. Warren, Kollias, John G., and Alexandridis, Nikitas A., "An Implementation of the Exponential Pyramid Data Structure with Application to Determination of Symmetries in Pictures", Computer Vision, Graphics, and Image Processing, 25, (1984), pp 218-225.

[CHIE and AGGA-84] Chien, C.H. and Aggarwal, J.K., "A Normalized Quadtree Representation", Computer Vision, Graphics, and Image Processing, 26, (1984), pp 331-346.

[DYER82] Dyer, Charles R., "The Space Efficiency of Quadtrees", Computer Graphics and Image Processing, 19, (1982), pp 335-348.

[ELCO86, et al.] Elcock, E.W., Gargantini, I. and Walsh, T.R., "Triangular Decomposition", Image and Vision Computing, Vol. 5, (3), August 1987, pp 225-231.

[GARG82] Gargantini, Irene, "An Effective Way to Represent Quadtrees", Communications of the ACM, Vol. 25, (12), December 1982, pp 905-910.

[GARG83] Gargantini, Irene, "Translation, Rotation and Superposition of Linear Quadtrees", International Journal of Man-Machine Studies, Vol. 18, (3), March 1983, pp 253-263.

[GRUN and SHEP-77] Grunbaum, Branko and Shephard, Geoffery, "Tilings by Regular Polygons", Mathematics Magazine, Vol. 50, No. 5, November 1977, pp 227-247.

[HUNT78] Hunter, Gregory, "Efficient Computation and Data Structures for Graphics", Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J.

[HUNT and STEI-79a] Hunter, Gregory and Steiglitz, Kenneth, "Operations on Images Using Quad Trees", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol PAMI-1, No. 2, April 1979, pp. 145-153.

[HUNT and STEI-79b] Hunter, Gregory and Steiglitz, Kenneth, "Linear Transformations of Pictures Represented by Quad Trees", Computer Graphics and Image Processing, 10, (1979), pp 289-296.

[LI82, et al.] Li, Ming, Grosky, William I., and Jain, Ramesh, "Normalized Quadtrees with Respect to Translations", Computer Graphics and Image Processing, 20, (1982), pp 72-81.

[LI and LOEW-87] Li, S.X., and Loew, M.H., "The Quadcode and Its Arithmetic", Communications of the ACM, Vol. 30, (7), July 1987, pp 621-626.

[MACG76] MacGillavry, Caroline H., "Fantasy and Symmetry: The Periodic Drawings of M.C. Escher", Harry N. Abrams, Inc., New York, 1976.

[MEAG82] Meagher, Donald, "Geometric Modeling Using Octree Encoding", Computer Graphics and Image Processing, 19, (1982), pp 129-147.

[SAME80] Samet, Hanan, "Region Representation: Quadtrees from Binary Arrays", Computer Graphics and Image Processing, 13, (1980), pp 88-93.

[SAME80a] Samet, Hanan, "Region Representation: Quadtrees from Boundary Codes", Communications of the ACM, Vol. 23, (5), March 1980, pp 163-170.

[SAME81] Samet, Hanan, "An Algorithm of Converting Rasters to Quadtrees", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-3, No.1, January 1981, pp 93-95.

[SAME84a] Samet, Hanan, "Algorithms for the Conversion of Quadtrees to Rasters", Computer Vision, Graphics and Image Processing, 26, (1984), pp 1-16.

[SAME84b] Samet, Hanan, "The Quadtree and Related Hierarchical Data Structures", Computing Surveys, Vol. 2, June 1984, pp 187-260.

[SCHA78] Schattschneider, Doris, "The Plane Symmetry Groups: Their Recognition and Notation", American Mathematical Monthly, Vol. 85, No. 6, June/July 1978, pp 439-450.

[SCOT and IYEN-86] Scott, David S. and Iyengar, Sitharama, "TID - A Translation Invariant Data Structure of Storing Images", Communications of the ACM, Vol. 29, (5), May 1986, pp 418-429.

[SHUB and KOPT-74] Shubnikov, A.V. and Koptsik, V.A., "Symmetry in Science and Art", Plenum Press, New York, 1974, pp 129-158.

[STEW86] Stewart, I.P., "Quadtrees: Storage and Scan Conversion", Computer Journal, 29,(1) 1986, pp 60-75.

[TANIS] Tanis, Elliot A., "Construction of Tesselations Using the Computer", Department of Mathematics, Hope College, Holland, Michigan, no date.

[YAMA84, et al.] Yamaguchi, K., Kunii T.L., Fujimura, K., and Toriya, H., "Octree-Related Data Structures and Algorithms", IEEE Computer Graphics and Applications, Vol. 4, No. 1, (January 1984), pp 53-59.