Pattern Recognition, Vol. 20, No. 3, pp. 309–319, 1987.
Printed in Great Britain.

0031-3203/87 $3.00 + .00
Pergamon Journals Ltd.
Pattern Recognition Society

COMPARISON OF A REGULAR AND AN IRREGULAR DECOMPOSITION OF REGIONS AND VOLUMES

JOHN C. FIALA*
Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, U.S.A.

and

ROBERT M. HARALICK†
Machine Vision International, Burlington Center, Ann Arbor, Michigan 48104, U.S.A.

(*Received* 11 *February* 1986, *in revised form* 11 *August* 1986)

Abstract — A comparison is made between *K*-trees, (i.e. quadtrees and octrees,) which are regular decompositions of volumes, and new, irregular decompositions called *R*-trees. This comparison is made within the context of the representation problems that might be associated with a robotic system. The results show that the irregular decomposition is independent of object position and can provide a more efficient encoding for certain shapes. However, detecting intersections between *R*-trees requires an algorithm of greater complexity than that of interference detection for octrees due to the irregularity of the *R*-tree decomposition.

Object representation Image encoding Quadtree Octree
Hierarchical data structures Recursive Decomposition Interference detection

1. INTRODUCTION

Regular decompositions of regions and volumes have received much attention in the recent literature.[1–17] These representations have proven useful in a number of applications including graphics, data compression and robotic solid modeling. Among the most popular of these techniques are quadtrees[2–11] and octrees,[10–16] and binary trees.[1,10]

The basic idea behind this type of decomposition is that a universe entity, [square, cube, or hype-cube[11,17]] is a power-of-two in diameter and can be recursively divided in half in each of the coordinate directions down to the resolution of the power-of-two grid. For example, an octree is constructed as follows. Begin with a large cube that contains the entire work space (or universe) of all objects, including the object to be represented by this tree. This cube is the root node of all octrees of all objects. The edges of this cube define the directions of the world coordinate axes. Now, divide the cube in half in each of the three coordinate directions generating eight "octants", Fig. 1. These octants are the children of the root node. If an octant contains none of the volume of the object it is labeled "empty". If an octant is completely filled by part of the object's volume, it is labeled "full". Octants which are "full" or "empty" are not subdivided further, but "mixed" octants are recursively subdivided in eight parts until either no "mixed" octants are generated or the desired resolution is reached.

From this description it can be seen that an octree is a *regular* decomposition of a volume, in the sense that divisions are made at predetermined, regular intervals. Any level of the tree represents a known, evenly spaced subdivision of the original cube. Klinger and Dyer[4]

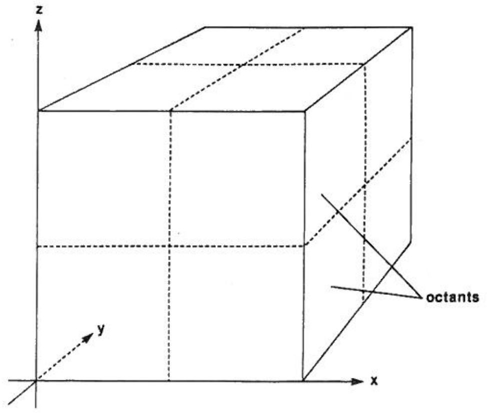

Fig. 1. Octree construction.

*Currently at the National Bureau of Standards, Gaithersburg, MD U.S.A.

†Address correspondence to R. Haralick, Dept. of Electrical Engineering, University of Washington, Seattle, Washington 98195, U.S.A.

emphasize that regular decompositions of this form have many advantages, one of which is that any geographical part of the image may be rapidly accessed. Also, since the form of the decomposition is predetermined, the tree can be encoded in a linear array format to save space as proposed by Gargantini[3,14] and Tamminen.[10] Unfortunately, as pointed out by Ahuja and Nash,[12] octree representations are not constructed with respect to the object itself but with respect to a world coordinate system. For this reason, the representation is not independent of the object's position. This presents two problems. One, if an object is translated a new octree must be constructed.[12,15,16] And two, the space required for an octree representation varies greatly with the object's position relative to the universe cube.[5]

The decomposition primitives of the octree are boxes, (specifically cubes,) whose edges are parallel to world coordinate axes. This type of primitive has some nice features including a simple mathematical description and a straight-forward algorithm for determining intersections between primitives. The reader may have wondered whether it is possible to retain this type of primitive and yet achieve position independence. The desired representation to meet this criterion would correspond to an *irregular* decomposition based on the object's shape. Also, Rubin and Whitted[23] recognized that an irregular decomposition of this form could improve both time and space efficiency in computer graphics applications, but the idea has not been pursued further.

2. BACKGROUND

Three- and k-dimensional decomposition techniques have analogous techniques in two dimensions. For instance, the two-dimensional counterpart of the octree is the quadtree. These two-dimensional versions are easier to visualize and explain; therefore, the discussion of irregular tree decompositions best begins in the two-dimensional realm.

The idea of an irregular decomposition of a two-dimensional region is not new. Pfaltz and Rosenfeld[22] presented the idea of representing a region as a set of maximal neighborhoods in 1967. This idea is a variation on Blum's Medial Axis Transformation.[20] In this representation, only the skeletal points and their respective radii need be stored to fully represent the region. However, since the areas of the blocks centered on the skeleton overlap, there is a redundancy of information resulting in less than optimal storage requirements. Also, though this skeletal representation is better suited to the problem of determining if a point belongs to a region than the boundary representation, the skeletal blocks are organized only as a simple list, which is not the most efficient for queries. The irregular decomposition of interest to this discussion should provide both query speed and space efficiency.

Ferrari, Sankar and Sklansky[21] have developed an algorithm that finds the minimal rectangular partition of a digitized region. Their rectangular partition representation does not have the redundancy of the skeletal representation and may therefore be more space efficient. Still, the partition is not organized in a manner that will result in particularly fast region queries.

The quadtree, on the other hand, provides for efficient queries through its hierarchical structure. Taking advantage of this basic structure, Samet[22] presents a tree version of the Medial Axis Transform which he calls the Quadtree Medial Axis Transform (QMAT). This representation provides space efficiency beyond that of the quadtree and eliminates some of the variability of quadtree size when the region is translated. Unfortunately, since the representation is based on a regular decomposition associated with fixed coordinate axes, the QMAT must still be reconstructed when a translation occurs. This dependence of the representation on position is something that an irregular decomposition should avoid.

3. THE RECTANGLE-TREE

The rectangle-tree provides a representation of a region that is efficient both in space and query speed The tree is based on an irregular decomposition that i independent of a region's position with respect tc translation. Since the rectangle-tree is composed o disjoint rectangles whose sides are parallel to fixec coordinate axes, the representation is not independen of a region's orientation.

The decomposition used by the rectangle-tree i illustrated in Fig. 2. The region is surrounded first b an enclosing box. This box is the smallest rectangl that contains all of the region and has sides parallel t the world coordinate axes. For digital images, the side will be parallel to the image boundary. After thi *outside* rectangle has been determined, the *insid* rectangle must be found. The inside rectangle is an

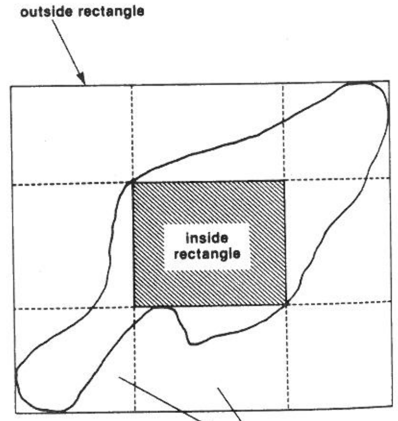

Fig. 2. Rectangle-tree decomposition.

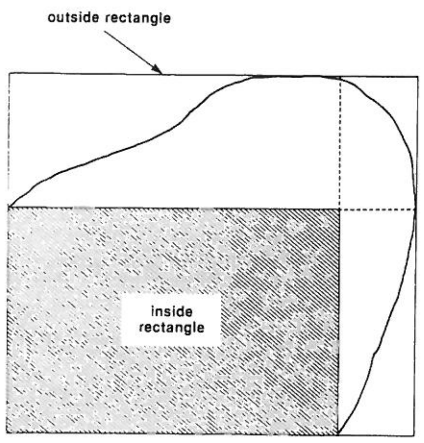

Fig. 3. Children at level 2 node.

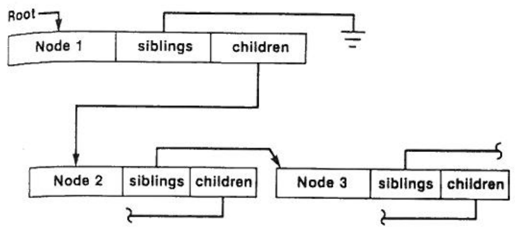

Fig. 4. Generalized tree structure.

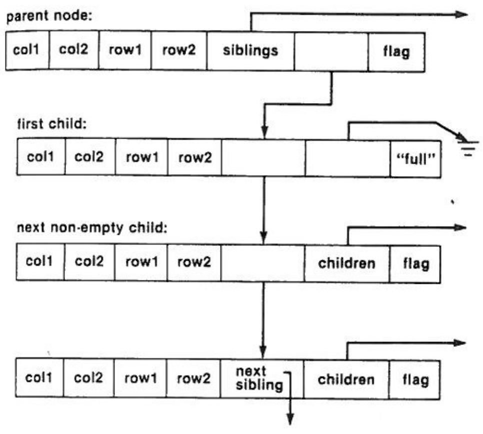

Fig. 5. Rectangle-tree structure.

largest rectangle that is completely within the region and which has sides parallel to the outside rectangle. The extension of the sides of the inside rectangle divides the outside rectangle into nine, ordered children rectangles. The first child is the inside rectangle, which is known to be completely inside the region. The other eight children are the rectangles derived by extending the sides of the inside rectangle as indicated by the dotted lines in the figure. These children rectangles become the outside rectangles for recursive decomposition. They are ordered according to the size of their inside rectangles, those with the largest inside rectangles being first. Any derived children that contain no part of the region are not included in the tree.

Any rectangle (node) in the tree which is completely contained within the region is labeled "full". Inside rectangles are always "full". "Full" nodes require no further subdivision and are therefore leaves of the tree. Nodes not "full" are "mixed" and are recusively subdivided until either the desired accuracy is achieved or no "mixed" rectangles remain. Figure 3 illustrates the children obtained from the second child of the outside rectangle of Fig. 2.

4. EXPERIMENTAL COMPARISON TO QUADTREE STRUCTURES

The experimental analysis of various tree decompositions was accomplished through GIPSY, the Spatial Data Analysis Lab's General Image Processing System.[19] For the purposes of comparison, rectangle-trees and quadtree-type structures were all stored in a similar format. The trees were stored using a generalized tree structure in a random access file on disk. Each record in the file was a node in the tree.

A generalized tree structure is one in which each node in the tree may have both children and siblings. The children of a node are nodes at the next level of the tree. These nodes represent a decomposition of the parent rectangle for trees such as quadtrees and rectangle-trees. The siblings of a node are nodes at the same level of the tree that have the same parent node. Figure 4 shows the generalized tree structure.

The rectangle-tree record format is depicted in Fig. 5. Each record has seven fields. The first four fields contain the first row, last row, first column, last column of the rectangle which constitutes the tree node. Next is the pointer to the remaining non-empty siblings of the node and a pointer to the node's children. If this node is not "full" then the first child will be the node's inside rectangle. The final field of the record is the flag to indicate whether a node is "full" or "mixed".

Similarly, the quadtree data structure has a record format as shown in Fig. 6. The principal difference from the rectangle-tree structure is that each node of a quadtree generally has either four children or none. Also, the flag of field seven may take on the values "full", "mixed", or "empty", for a quadtree.

To test the efficiency of the rectangle-tree with respect to queries, several experiments were conducted. For these experiments polygonal regions were randomly generated in square images in the following manner. The origin was assumed to be located at the center of the image. A length between zero and one-half the image size was randomly generated. Then an angle between 0 and 90° was randomly selected. This length and angle determined the location of the first vertex of the polygon in the image. Lengths less than half the image sized and angles between 0 and 121° were then randomly generated until the total angle exceeded 360°. The generation of a polygon is

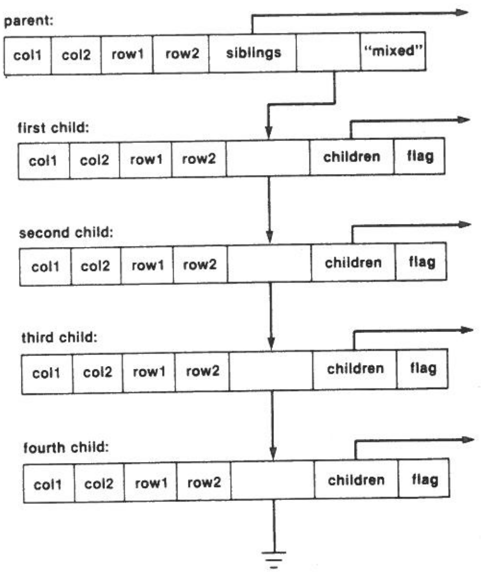

Fig. 6. Quadtree structure.

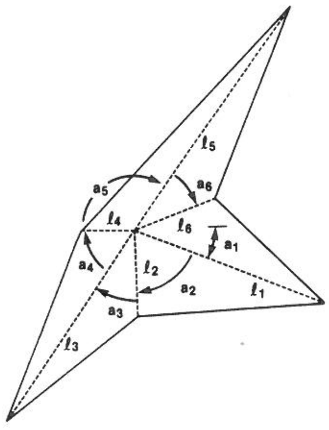

Fig. 7. Random generation of polygonal regions.

```
procedure QTREE_INT( X, RECTANGLE )
  // intersect RECTANGLE with quadtree pointed to by X //

  if ( X ≠ 0 ) then
   [ call READ( X, RECTANGLE_X, CHILDREN_X,
                               SIBLINGS_X, STATUS )
     if INTERSECTS( RECTANGLE_X, RECTANGLE ) then
       if ( STATUS = full ) then
         QTREE_INT = true
       else
         QTREE_INT = QTREE_INT( CHILDREN_X,RECTANGLE )
     if ( QTREE_INT ≠ true ) then
         QTREE_INT = QTREE_INT( SIBLINGS_X, RECTANGLE ) ]
  else
    QTREE_INT = false

  return

end QTREE_INT
```

Fig. 8. Rectangle intersection algorithm for quadtree.

```
procedure RTREE_INT( X, RECTANGLE )
    // intersect RECTANGLE with rec-tree pointed to by X //
    if ( X ≠ 0 ) then
     [ call READ( X, RECTANGLE_X, CHILDREN_X,
                                  SIBLINGS_X, FULL )
       if INTERSECTS( RECTANGLE_X, RECTANGLE ) then
         if FULL then
           RTREE_INT = true
         else
          [ call READ( CHILDREN_X, IN_RECTANGLE,
                         DUMMY, SIBLINGS_CHILD, FULL )
            if INTERSECTS( IN_RECTANGLE, RECTANGLE ) then
              RTREE_INT = true
            else
              RTREE_INT = RTREE_INT( SIBLINGS_CHILD,
                                        RECTANGLE ) ]
       if ( RTREE_INT ≠ true ) then
         RTREE_INT = RTREE_INT( SIBLINGS_X, RECTANGLE )
     ]
    else
      RTREE_INT = false
    return
  end RTREE_INT
```

Fig. 9. Rectangle intersection algorithm for rectangle-tree.

illustrated in Fig. 7, where the l's and a's are th
generated lengths and angles. The vertices ar
connected to form the polygon as shown. Once th
polygon has been created the region can be colored-i
to form the test image.

One query of interest might be to determine if
randomly generated test rectangle intersects th
randomly generated polygonal region. The algorith
for this query when the region is represented as
quadtree is given in Fig. 8. The algorithm is slightl
modified for rectangle-tree representations, as show
in Fig. 9. A simple routine INTERSECTS is assume
in these algorithms to determine whether or not tw
rectangles intersect. Also, a routine READ is assume
that obtains the tree node pointed to by X. In th
experiments, the number of accesses to the dat
structure, i.e. the number of calls to READ, was th
quantity of interest, since this corresponds to th
amount of processing done by each algorithm.

Rectangles for queries were randomly generated b
first generating a column value in the image. This wa
the last column of the rectangle. Then the first colum
was generated by randomly selecting a column in th
image less than the last column. The first and last row
of the test rectangle were generated in a simila
manner.

Table 1 shows the results of two comparison runs fo
256 × 256 images of random polygonal regions. Not
that the quadtree representation used in these runs is
complete quadtree where empty nodes are explicit i
the tree. These nodes provide no help in determinin
test rectangle intersections. Therefore, if empty nod
are left out of the quadtree structure, giving what sha
be called a black quadtree, the results of Table 2 a
obtained.

Note also that the rectangle-tree encodes only th

Table 1. Comparison of quadtrees and rectangle-trees

Test Image Size	256 × 256	
Number of Images	31	
Number of Data Points	1550	
Total Number of Intersections	733	
Total Number of Non-intersections	817	
Comparison Results	Mod Quadtree	Rectangle-tree
Avg. No. of Records in Tree	898	328
Avg. No. of Tree Accesses	6.8	3.4
Standard Dev. of Accesses	7.4	4.9

Table 2. Black quadtree comparison

Test Image Size	256 × 256	
Number of Images	50	
Number of Data Points	2499	
Total Number of Intersections	1181	
Total Number of Non-intersections	1318	
Comparison Results	Quadtree	Rectangle-tree
Avg. No. of Records in Tree	1560	335
Avg. No. of Tree Accesses	17.6	3.5
Standard Dev. of Accesses	11.3	5.1

Table 3. Modified black quadtree comparison

Test Image Size	256 × 256	
Number of Images	50	
Number of Data Points	2500	
Total Number of Intersections	1163	
Total Number of Non-intersections	1337	
Comparison Results	Blk Quadtree	Rectangle-tree
Avg. No. of Records in Tree	906	310
Avg. No. of Tree Accesses	11.1	3.4
Standard Dev. of Accesses	6.7	5.2

area of the image enclosed by the rectangle which bounds the polygonal region. A similar technique could be employed for quadtrees, i.e. first surround the region with a minimal rectangle, then make regular divisions of this rectangle to form the tree. This should lead to faster determination of null intersections for black quadtrees. The results for this modified black quadtree structure are given in Table 3, and are, in fact, an improvement over the black quadtree. However, since the minimal rectangle enclosing the region is not $2^n \times 2^n$, divisions can not be guaranteed to be exactly equal. Thus some of the encoding techniques for quadtrees are not applicable to this modified structure.

To test the space efficiency of certain tree structures Tamminen[10] uses a $2^{10} \times 2^{10}$ encoding of a black disk. Following his lead, the 1024 × 1024 black disk was used to compare quadtrees and rectangle-trees. The quadtree of this region was found to have 10,341 nodes, 3932 of which were black leaves. The rectangle-tree, on the other hand, had 2354 nodes, 1177 of which were black leaves. At first glance rectangle-trees would appear to offer a significant advantage, however the regularity of the quadtree decomposition provides certain possibilities for encoding.

Gargantini[3] has shown that it is only necessary to store the black leaves of the quadtree and each leaf node can be encoded in $3(n1) + 2$ bits for a $2^n \times 2^n$ image. So that for this example, only 29*3932 = 113, 028 bits are required for the entire quadtree representation.

Consider now the rectangle-tree. To represent rectangles in a 1024 × 1024 image, 40 bits are needed, (10 bits for each integer). Also, 12 bits are needed for each pointer and one additional bit for the flag field. This means that 65 bits are required to encode each node. Recall, however, that the sibling nodes are derived from the outside and inside rectangles of the parent node. Thus if the outside and inside rectangles are completely specified, only a 3-bit code which indicates how to derive the child from these two rectangles needs to be stored in all the nodes which are siblings to the inside rectangle node. For rectangle-trees the number of siblings will be one less than the total number of black leaves, since the root node is not a sibling and for all leaf rectangles besides its inside one there are corresponding rectangles which are siblings. So, of the 2354 nodes of the example rectangle-tree, 1176 will be derived rectangles that only require the 3-bit code. Therefore the rectangle-tree can be encoded in 28*1176 + 65*1178 = 109, 498 bits, giving a space efficiency comparable to Gargantini's quadtree encoding.

It should be mentioned here that neither of these tree encodings approaches the data compression of the linear encoding of a quadtree's preorder traversal as presented by Kawaguchi and Endo.[2] However, certain operations are applicable to the tree structures that are not available for the linearly encoded form, as pointed out in Ref.[10]

The reader may have noted that the rectangle-tree should perform well for regions where significant inside rectangles can be found. Consider the images depicted in Fig. 10 as cases where the rectangle-tree should do poorly. For the rotated square of Fig. 10(a) the quadtree consisted of 1365 nodes, 348 of which were black leaves, while the rectangle-tree had 489 nodes and 245 black leaves. For the circle encoded as in Fig. 10(b), the quadtree had 15,189 nodes, 3244

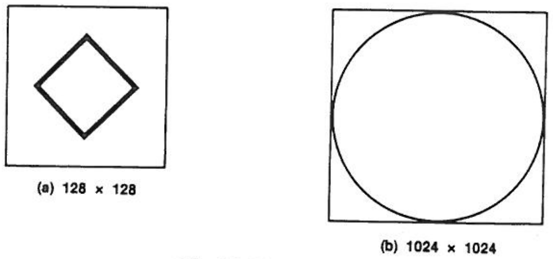

Fig. 10. Line images.

black leaves, and the rectangle-tree had 2391 nodes with 1307 black leaves.

The performance of both types of tree structures is highly data dependent. Obviously, the rectangle-tree will perform miserably on a pixel-sized checker board. But, since the desired use for this decomposition is for representing real objects (that generally have some substantial area) and not just arbitrary images, the rectangle-tree's poor performance on the checker board is not of particular concern. The practicality of this decomposition in three dimensions should be examined in some detail.

5. THE RECTANGULAR PARALLELEPIPED-TREE

The extension of the rectangle-tree to three-dimensional shape representation yields a tree of rectangular parallelepipeds (RPPs). Thus this decomposition can be termed a "rectangular parallelepiped-tree" or "RRP-tree". Sometimes the term "R-tree" will be used to refer to rectangle- and RPP-trees.

Figure 11 depicts the three-dimensional decomposition of RPP-trees. This decomposition is completely analogous to the rectangle-tree structure of two dimensions. All of the RPPs in the RPP-tree have edges parallel to the coordinate axes of a world coordinate system. The RPP-tree has an *inside* RPP and an *outside* RPP at each node which is "mixed". The planes of the boundary of the inside RPP partition the outside RPP into 27 smaller RPPs. These 27 RPPs, which include the inside RPP, form the children of the outside RPP. Any "mixed" children are recursively subdivided in a similar manner to form the entire tree.

Since the increase in dimensionality has lead to a significant increase in the number of possible children per node, the performance of the RPP-tree should be examined closely.

6. EXPERIMENTAL COMPARISON TO OCTREE STRUCTURES

In GIPSY, voxel arrays are stored as multi-band images. These multi-band images were used to conduct experiments on the properties of RPP-tree and octree encodings of three-dimensional objects.

The RPP-tree was stored in the same manner as the rectangle-tree of Fig. 5. Two fields were added to each record to indicate the first and last bands of a record's RPP. The other fields, including the pointer and flag fields, retained their original definitions. Likewise, octrees were stored as in Fig. 6 with the addition of fields indicating the first and last bands of each RPP.

For the four-sided polyhedron of Fig. 12, positioned in a 64 × 64 × 64 voxel array as shown, the following results were obtained. The octree contained 12,089 nodes, 4682 of which were black leaves. The RPP-tree contained 2852 nodes, 1434 of which were black leaves.

The intersection algorithm of Figs. 8 and 9 can be transfered directly to the three-dimensional domain by simply replacing all references to rectangles with rectangular parallelepipeds. Because the algorithms are based on a generalized tree structure, no other changes are required. These algorithms were used to compare the efficiencies of RPP-trees and octrees with respect to queries for RPP intersections. For 50 randomly generated RPPs within the array of Fig. 12, the RPP-tree had an average data structure access count of 22.2 accesses per RPP, while the octree had an average access count of 34.5. However, for the black octree, i.e. the tree containing only "mixed" and "full" nodes, the speed of the octree query was much improved. For 50 random RPPs, the RPP-tree averaged 24.5 accesses and the black octree, only 22.7 accesses.

For objects that have a little more regularity with respect to the coordinate axes, the properties of the RPP-tree become a distinct advantage. Consider the quadrilateral cylinder positioned in a 64 × 64 × 64 voxel array as shown in Fig. 13. The octree for this object had 22,569 nodes, while the RPP-tree required only 137 nodes, a significant difference to say the least. The black octree for this example contained 12,148 nodes. The average number of accesses for the black octree was 22.2 compared with 4.8 for the RPP-tree,

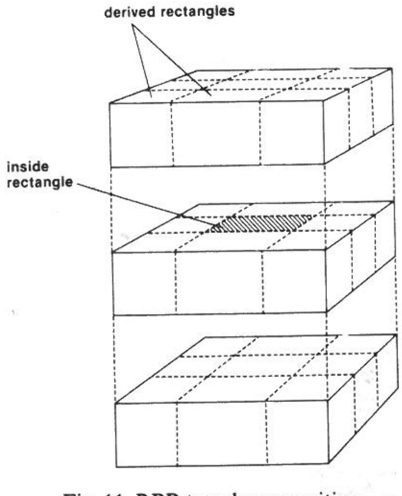

Fig. 11. RPP-tree decomposition.

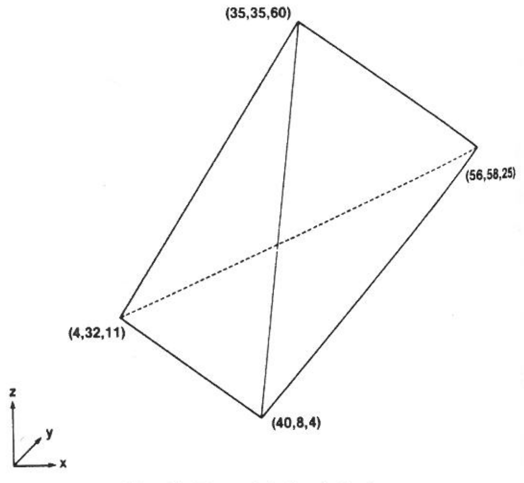

Fig. 12. Four-sided polyhedron.

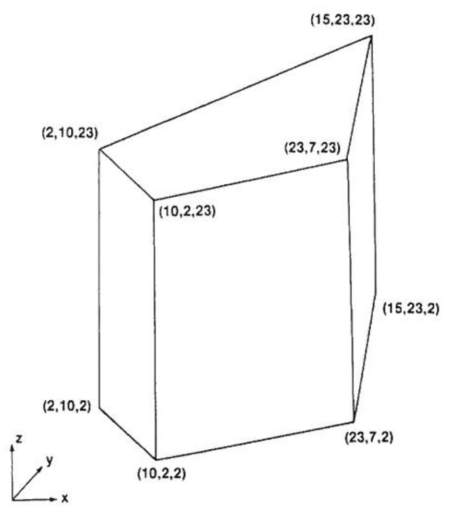

Fig. 13. Quadrilateral cylinder.

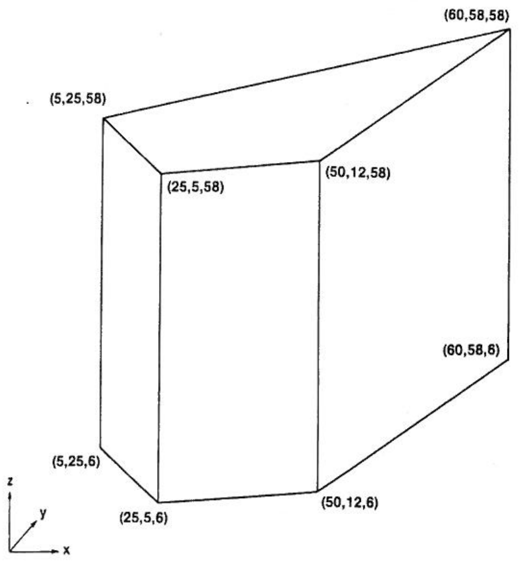

Fig. 14. Quadrilateral cylinder at corser resolution.

when 50 random intersection RPP's were tested.

Due to the irregular nature of the RPP-tree decomposition, its performance is not affected by a change in image resolution, whereas the octree's performance degrades as the resolution is increased. For example, for quadrilateral cylinders, the octree does much better against the RPP-tree at a coarser resolution. In Fig. 14, a quadrilateral cylinder similar to that of Fig. 13 is positioned in a 25 × 25 × 25 voxel array. The octree-like decomposition of this volume yielded a tree of 1607 nodes, while the RPP-tree had 52 nodes. Also, for determining the intersection of 20 randomly generated RPPs with the object, the octree structure required 11.8 accesses and the RPP-tree, 3.9 accesses, on the average.

Since the size of the octree is highly dependent on the position of the world coordinate origin, it may be possible to obtain a significant improvement by selecting a more appropriate origin when constructing the tree. To test this idea, the origin in Fig. 14 was moved to the point where the inside RPP of the root of the RPP-tree would correspond to an entire octant of the octree. This experiment lead to an octree of 943 nodes, a significant improvement over the 1607 nodes of the previous tree. However, the RPP-tree still only contained 52 nodes. It is not affected by translation!

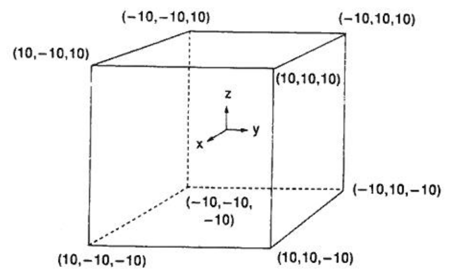

Fig. 15. Cubic shell.

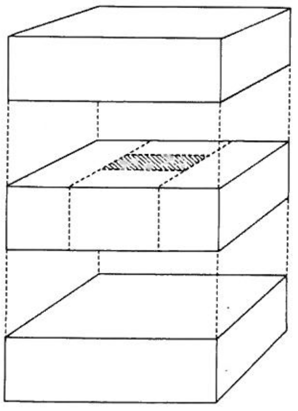

Fig. 16. Alternate irregular decomposition.

As a final experiment, the "worst case" of RRP-tree space efficiency was tested. In Section 4, the "worst case" space efficiency turned out to be a thin square with no sides parallel to the coordinate axes [Fig. 10(a)]. For the three-dimensional "worst case" consider the cube of Fig. 15 rotated by 45 degrees around the x-, y-, and z-axes, and translated 32 units in the positive x-, y-, and z-directions into the center of a 64 × 64 × 64 image. The object that was encoded was the one-voxel-wide solid that comprised the surfaces of the cube. The octree for this object contained 8145 nodes, 2355 of which were black leaves and 4772 of which were white leaves. The RPP-tree had 2,281 nodes, 1169 of which were black leaves.

The reader may have noticed that the definition of children RPPs can be made in ways other than that given in Fig. 11. Once the inside RPP is determined, the children RPPs can be chosen in any manner that completely decomposes the remaining volume of the outside RPP. Obviously, the choice of children RPPs can affect the resulting tree size. For example, consider the decomposition structured as in Fig. 16. Here, the outside RPP is divided into seven children RPPs instead of 27 as in Fig. 11. For the rotated cubic shell, an *R*-tree of this form possessed 1986 nodes, 993 of which were black leaves. This is a 12.9% decrease in the total number of nodes required for object representation. For the four-sided polyhedron of Fig. 12, a 5.5% drop in the total number of tree nodes was discovered.

In any case, the overall performance of the *R*-tree is highly data dependent. However, as has been shown, the RPP-tree has the advantage of being invariant under translations, which is particularly useful in robotics applications. This possible appropriateness of the *R*-tree for robotics will be examined in greater detail in the next sections.

7. ROBOTIC OPERATIONS ON R-TREES AND OCTREES

For a robotic system, there are three operations on tree decompositions that are of particular importance. Since robots typically manipulate the objects in their environment, or objects are moved to facilitate inspection tasks, it is important to be able to translate and rotate the tree decomposition representations. Also, since collision checking is a prominent operation in planning robot movements, a system must be able to determine if two trees intersect. The following sections review the principles of these three operations with respect to *R*-trees and octrees.

8. TRANSLATION

Several authors have presented algorithms for the translation of octrees.(12,15,16) These algorithms either require that translations occur only in integer multiples of the base voxel size,(12) or that some error is incurred in the translated octree so that the original octree must always be stored for further translations.(16) In either case, translation of an octree requires rebuilding of the tree.

R-trees, however, are independent of the world coordinate system with respect to translation. The actual *R*-tree will be an integer decomposition of the discretized representation. That is, the RPPs that make-up the tree will be defined by the integers, first column, last column, first row, last row, first band, and last band, within the discretized outside RPP. This integer representation is less expensive space-wise than storing the actual real number positions of these RPPs. To locate the outside RPP of this integer *R*-tree a single real vector l is needed. If the *R*-tree is to be translated (by any amount,) the only modification is to l. Thus, translation of *R*-trees is trivial.

9. ROTATION

Arbitrary rotation of octrees is not possible, although some schemes have been presented for obtaining approximations to rotated trees.(13,16) Rotations by multiples of 90° are possible with octrees and several algorithms have been given.(15,16) These algorithms are based on the fact that a rotation by a multiple of 90° results in a reordering of the nodes of the tree. No new nodes are created and the entire operation can be carried out by a simple relabeling algorithm.

Since *R*-trees are composed of RPPs aligned with the world coordinate axes, as are octrees, they have the same properties with respect to rotations as do octrees. Arbitrary rotations of *R*-trees are not possible, though approximation schemes could be devised. Rotations by multiples of 90° are possible and require only that each node of the tree be relabeled. For example, for a rotation of 90° about the x-axis, where columns of the outside RPP image segment the x-axis of the world coordinate system, the (first row, last row) pair of each node is swapped with the (first band, last band) pair of that node. If the outside RPP did not have an edge lying in the x-axis of the world coordinates, then the position vector l must also be updated to complete the operation. This is an advantage over the octree, which must be translated to the origin, rotated, and then translated back to complete the operation.

10. INTERFERENCE DETECTION

The algorithm for determining whether two objects represented by octrees have any voxels in common is well-known.(13) This algorithm is given in Fig. 17, slightly modified to accommodate a generalized tree structure. The octrees tested by this algorithm are in the form described in Section 6. It is assumed that there is a routine READ which obtains a node of the tree from the data structure. As before, the number of accesses to the octree is an indication of the amount of processing done by the algorithm.

Recall that the octrees of two objects share the same

```
procedure OCTREES_INT( X, Y )

    // intersect the octree pointed to by X
       with the octree pointed to by Y    //


    if ( X ≠ 0 ) and ( Y ≠ 0 ) then
     [ call READ( X, RPP_X, CHILD_X, SIBLING_X, STATUS_X )
       call READ( Y, RPP_Y, CHILD_Y, SIBLING_Y, STATUS_Y )
       if ( STATUS_X = void ) or ( STATUS_Y = void ) then
          OCTREES_INT = OCTREES_INT( SIBLING_X, SIBLING_Y )
       else
        [ if ( STATUS_X = full ) or ( STATUS_Y = full )
          then OCTREES_INT = true
          else
             OCTREES_INT = OCTREES_INT( CHILD_X, CHILD_Y )

          if ( OCTREES_INT ≠ true ) then
             OCTREES_INT = OCTREES_INT( SIBLING_X,
                                        SIBLING_Y ) ]

     ]
    else
     OCTREES_INT = false
    return

end OCTREES_INT
```

Fig. 17. Octrees intersection algorithm.

universe cube. Also, because the tree decomposition is a regular one, the positions of the eight children of the universe cube are the same for either object, as are the positions of the children of any octant in the tree. Thus, at all levels of the octrees of the two objects the node RPPs exactly correspond, that is, provided both octrees have nodes down to that level. This means that no routine is needed to determine if two RPPs intersect, and that the algorithm need only compare the flag fields of corresponding octants to determine if an intersection exists. However, this also means that the octree must retain both its "empty" and "full" nodes, and therefore take-up more space.

Unfortunately, the *R*-trees of two objects do not share a universe RPP, nor are their nodes guaranteed to correspond in any uniform manner. This requires that a child of one *R*-tree, whose parent node intersects a node of another *R*-tree, be compared with each child node of the other *R*-tree in order to determine an intersection. If no intersection is found for this child, then its sibling must be compared with each child of the intersecting *R*-tree node. This process will continue until each child of the first *R*-tree has been tested with every child of the second *R*-tree, (or until an intersection is found). Thus, the *R*-tree algorithm has worse complexity due to the irregularity of the decomposition, but it should be remembered that the major factor determining the speed of processing is the nature of the two objects and their (non-) intersection.

The algorithm for detecting interference between two *R*-trees is given in Fig. 18. As in earlier algorithms, the subprograms READ and INTERSECTS are assumed. It is also assumed, for simplicity, that the array encodings of the *R*-trees have the same origin. If this was not the case, the RPPs of one tree would have to be translated before the use of INTERSECTS function.

```
procedure RTREES_INT( X, Y )
    // intersect two R-trees pointed to by X and Y //
    if ( X ≠ 0 ) and ( Y ≠ 0 ) then
    [ call READ( X, RPP_X, CHILD_X, SIBLING_X, FULL_X )
      call READ( Y, RPP_Y, CHILD_Y, SIBLING_X, FULL_Y )
      if INTERSECTS( RPP_X, RPP_Y ) then
       [ if ( FULL_X ) and ( FULL_Y ) then
            RTREES_INT = true
         else if ( FULL_X ) then
            RTREES_INT = RTREES_INT( X, CHILD_Y )
         else if ( FULL_Y ) then
            RTREES_INT = RTREES_INT( Y, CHILD_X )
         else
            RTREES_INT = RTREES_INT( CHILD_X, CHILD_Y )
       ]
      if ( RTREES_INT ≠ true ) then
         RTREES_INT = RTREES_INT( X, SIBLING_Y )
      if ( RTREES_INT ≠ true ) then
         RTREES_INT = RTREES_INT( SIBLING_X, Y )
    ]
    else
      RTREES_INT = false
    return
end RTREES_INT
```

Fig. 18. *R*-trees intersection algorithm.

The reader should note that if the objects are far enough apart so that the bounding RPPs at the highest level of the two *R*-trees do not intersect, this will be detected immediately; whereas the octree intersection, depending on the size of the objects with respect to the size of the universe cube, may take a considerable amount of computation. The same is generally true if two objects overlap greatly. In this case, their inside RPPs will intersect, resulting again in early detection. For example, consider the overlay of the $64 \times 64 \times 64$ images of Figs. 12 and 13. The intersection of the *R*-trees of these objects, using the RTREES-INT algorithm, resulted in an intersection being detected after only four accesses to the data structures. The octree algorithm, however, required 40 accesses to the octree data structures.

For the other combinations of objects in Section 6 the following results were obtained. The intersection of the quadrilateral cylinder of Fig. 13 and the rotated, cubic shell obtained from Fig. 15 yielded an access count of four for the *R*-tree representations, and an access count of 44 for the octree representations. The *R*-tree algorithms perform well in this case due to the compactness of the *R*-tree representation for the quadrilateral cylinder. A less compact *R*-tree representation, such as that of the four-sided polyhedron of Fig. 12, intersected with the cubic shell results in a poorer performance by the *R*-tree. Detection of interference between the polyhedron and cubic shell required 26 accesses for the *R*-trees and 82 accesses for the octrees.

From these results it would appear that *R*-tree interference detection is faster than octree interference detection. This is generally true when there is a fair amount of overlap between the objects, or when the objects have fairly space-efficient *R*-tree representations. However, consider the intersection between the polyhedron of Fig. 12 and the object of Fig. 19. Here,

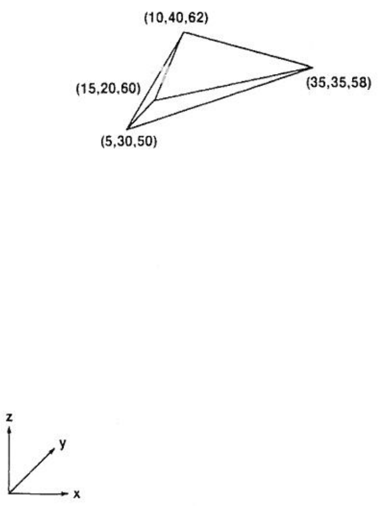

Fig. 19. Small polyhedron.

there is not much overlap between two objects with rather inefficient *R*-tree representations. Interference between these objects was detected by the *R*-tree algorithm after 124 accesses to the *R*-trees, and by the octree algorithm after 40 accesses to the octrees.

The real deficiency of the *R*-tree algorithm is the worst case performance. The octree algorithm will never require more accesses than twice the number of nodes in the smaller octree. The *R*-tree algorithm, on the other hand, can make several times as many accesses to the data structures as there are nodes in the largest *R*-tree. As an example, the object of Fig. 19 is shifted slightly to the position shown in Fig. 20. This shifted object now narrowly misses intersecting the polyhedron of Fig. 12. The *R*-tree algorithm determines that there is no intersection only after 29,630 accesses to the *R*-trees.

11. CONCLUSION

The comparison between *K*-trees and *R*-trees has shown several interesting things about hierarchic decompositions. Of particular significance is that irregular decompositions have position independence which can be quite useful when objects are non-stationary. Regular decompositions have no position independence. Also of importance is that the regularity of regular decompositions can be exploited to obtain efficient interference detection and compactly encoded forms. However, it appears that, if the tree structure is to be retained, the space required by both *K*-trees and *R*-trees is nearly the same. In fact, for certain shapes, the irregular decomposition out-performs the regular decomposition considerably with respect to space efficiency.

The usefulness of *R*-trees in applications will depend greatly on the type of data expected and the type of operations to be performed. But there are also several areas where variations on this basic representation should be explored for feasibility. Among these are the possible variations on choosing children once the inside box has been determined, and the potentiality of other primitives in a tree decomposition. For example, spherical primitives would provide rotational independence and fast primitive intersection determination.

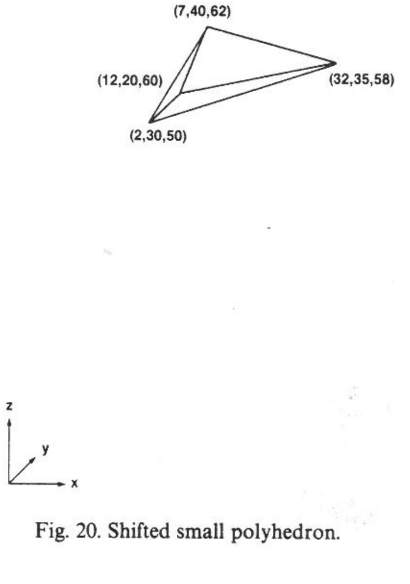

Fig. 20. Shifted small polyhedron.

REFERENCES

1. K. Knowlton, Progressive transmission of grey-scale and binary pictures by simple efficient, and lossless encoding schemes, *Proc. IEEE* **68**, 885–896 (1980).
2. E. Kawaguchi and T. Endo, On a method of binary-picture representation and its application to data compression, *IEEE Trans. Pattern Anal. Mach. Intell.* **2**, 27–34 (1980).
3. I. Gargantini, An effective way to represent quadtrees, *Commun. ACM* **25**, 905–910 (1982).
4. A. Klinger and C.R. Dyer, Experiments on picture representation using regular decomposition, *Comput. Graphics Image Process.* **5**, 68–105 (1976).
5. M. Li, W.I. Grosky and R. Jain, Normalized quadtrees with respect to translation, *Comput. Graphics Image Process.* **20**, 72–81 (1982).
6. H. Samet, Region representation: quadtrees from binary arrays, *Comput. Graphics Image Process.* **13**, 88–93 (1980).
7. H. Samet, An algorithm for converting rasters to quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **3**, 93–94 (1981).
8. H. Samet, A quadtree medial axis transform, *Commun. ACM* **26**, 680–693 (1983).
9. H. Samet, The quadtree and related hierarchical data structures, CS-TR-1329, Computer Science Dept. and Center for Automation Research, University of Maryland, November (1983).
10. M. Tamminen, Comment on quad- and octrees, *Commun. ACM* **27**, 248–249 (1984).
11. C.L. Jackins and S.L. Tanimoto, Quad-trees, oct-trees, and *K*-trees: a generalized approach to recursive decomposition of Euclidean space, *IEEE Trans. Pattern Anal. Mach. Intell.* **5**, 533–538 (1983).
12. N. Ahuja and C. Nash, Octree representations of moving objects, *Comput. Vision Graphics Image Process.* **26**, 207–216 (1984).
13. M.N. Boaz, Spatial coordination of transfer movements in a dual robot environment, Master's Thesis, Virginia Tech (1983).
14. I. Gargantini, Linear octrees for fast processing of three-dimensional objects, *Comput. Graphics Image Process.* **20**, 365–374 (1982).
15. C.L. Jackins and S.L. Tanimoto, Octrees and their use in representing three-dimensional objects, *Comput. Graphics Image Process.* **14**, 249–270 (1980).
16. D. Meagher, Geometric modeling using octree encoding, *Comput. Graphics Image Process.* **19**, 129–147 (1982).
17. M. Yau and S.N. Srihari, A hierarchical data structure for multidimensional digital images, *Commun. ACM* **26**, 504–515 (1983).
18. R.M. Haralick, J.C. Fiala and L.G. Shapiro, Volume intersection problems, Technical Report, Spatial Data Analysis Laboratory, Virginia Tech, (1983).
19. S. Krusemark and R.M. Haralick, Achieving portability in image processing software packages, *Proc. IEEE Conf. on Pattern Recognition and Image Processing*, Las Vegas, Nevada, pp. 451–457 (1982).
20. H. Blum, A transformation for extracting new descriptors of shape, *Models for the Perception of Speech and Visual Form*, W. Dunn, ed. MIT Press, Cambridge, MA (1964).

21. L. Ferrari, P.V. Sankar and J. Sklansky, Minimal rectangular partitions of digitized blobs, *Comput. Vision, Graphics Image Process.* **28**, 58–71 (1984).
22. J.L. Pfaltz and A. Rosenfeld, Computer representation of planar regions by their skeletons, *Commun. ACM* **10** (1967).
23. S. M. Rubin and T. Whitted, A 3-dimensional representation for fast rendering of complex scenes, *Comput. Graphics* **14**, (1980).

About the Author — John C. Fiala received the B.S. degree in Mathematics from Arkansas State University in 1982 and the M.S. degree in Electrical Engineering from Virginia Polytechnic Institute and State University in 1985. He is currently employed in the Robot Systems Division of the National Bureau of Standards where his research interests include task-level programming systems and sensory-interactive robotics.

About the Author — Robert M. Haralick was born in Brooklyn, New York, on 30 September 1943. He received a B.A. degree in Mathematics from the University of Kansas in 1964, a B.S. degree in Electrical Engineering in 1966 and a M.S. degree in Electrical Engineering in 1967. He has worked with Autonetics and IBM. In 1965 he worked for the Center for Research, University of Kansas, as a research engineer and in 1969, when he completed his Ph.D. at the University of Kansas, he joined the faculty of the Electrical Engineering Department there where he last served as a Professor from 1975 to 1978. In 1979 Dr. Haralick joined the Electrical Engineering Department at Virginia Polytechnic Institute and State University where he was a Professor and Director of the Spatial Data Analysis Laboratory. From 1984 to 1986 Dr. Haralick served as Vice President of Research at Machine Vision International, Ann Arbor, MI. Dr. Haralick now occupies the Clairmont Egtvedt Boeing chaired professorship in the Department of Electrical Engineering at the University of Washington.

Dr. Haralick has done research in pattern recognition, multi-image processing, remote sensing, texture analysis, data compression, clustering, artificial intelligence and general systems theory, and has published over 200 papers. He is responsible for the development of GIPSY (General Image Processing System), a multi-image processing package which runs on a minicomputer system.

He is a Fellow of the Institute of Electrical and Electronic Engineers, and a member of the Association for Computer Machinery, and the Pattern Recognition Society.