Operations on Quadtree Encoded Images

M. A. Oliver* and N. E. Wiseman
Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge CB2 3QG, UK

A quadtree is a form of picture encoding which is compact and easily handled. It is based on a description of the recursive subdivision of those parts of the image where there is detail until some desired resolution is reached. This paper proposes a method of storing quadtrees and operating on images by manipulating their quadtree encodings.

1. INTRODUCTION

1.1 Description of encoding

An image represented by an $N \times N$ square array of pixels, each of P bits, would need $P \times N \times N$ bits to store in uncoded form. In practice, it is found that neighbouring pixels are often the same and that one- or two-dimensional spatial coherence can be exploited to write the image in much less than $P \times N \times N$ bits by the use of a suitable coding scheme. Run-length encoding is a popular example which makes use of one-dimensional coherence. A run-length encoded image is described in scanline order as a sequence of pairs of values a, b where a represents the number of consecutive pixels of value b in each run. Run-length encoding gives reasonably good compression in many cases (a factor of ten may be typical) but the image is not easily manipulated in its coded form. It is therefore useful for image storage and transmission but not much else.

A quadtree encoded image exploits two-dimensional coherence by recursively decomposing the image into square areas in a particular way. The code is notionally a tree structure with the root corresponding to the whole image. Unless the image is homogeneous it is subdivided into four quadrants, each being represented in the tree by a node joined by a branch to the root. These nodes are leaves when the quadrants they represent are themselves homogeneous, otherwise they carry further branches to nodes representing successively smaller subdivisions. The subdivision, and hence the tree growth, stops when either a sufficiently fine resolution is reached or all the nodes are leaves.

1.2 Why quadtree encoding could be important

At first sight the tree encoding described above appears to be just a compression trick which saves storage space. The savings are modest to good, depending on the image properties, and are generally rather better than for run-length encoding. There are, however, other aspects which are of interest and where the differences from other encoding methods are dramatic. Manipulating an image is often far easier to do by operating on its treecode than its run-length code, or even its pixel array (what one might call its uncoded form). Keeping images in a form suitable for display with only minor computations needed at display time also favours a tree encoding, the anti-aliasing values being held within the tree in a most natural way. This is explained as follows:

1. One method of calculating the anti-aliasing values for an image is based on super-sampling. The image is defined to a higher spatial resolution than the output device requires, and pixel averaging is used to reduce the resolution by the desired amount and deliver the soft edge pixel values.
2. In a quadtree encoded image a non-terminal node corresponds with an area in the image defined by the sub-tree which starts in that node. This sub-tree contains the leaves whose values represent the super-sampling of the node. Thus a node which fathers four leaves represents another leaf at half the spatial resolution of its descendants and can hold the average of the four descendant leaves as its value.

Storing the anti-aliasing values in the treecode does more than decorate the tree. Operations on treecoded images can arrange to preserve the correct averages on non-terminals, so that every image is available at all interesting resolutions. This property can be exploited so that an interactive system can always respond quickly with a low resolution answer while a better one is being loaded to the terminal. In animation work the display updates can be at low resolution while rapid movement persists, and at higher resolutions as movement slows or ceases. An image can be prepared for output on several devices of quite different resolutions and grey-scale capabilities without recomputation or compromise.

2. STORING QUADTREES

2.1 Pointer structures

Although a pointer structure may simplify any operation on the tree and speed access to its leaves, in general the memory requirements are unacceptable. In the work described here, we used one byte for each node value. A single pointer to that node would take four bytes and, if the whole tree were stored as a pointer structure, we should need around five times as much space for pointers as for leaves. Toy pictures of a few hundred leaves were not of much interest to us and all the 'real' images we worked with had a few thousand leaves (rarely more than 10 000 though). In using the system we built to manipulate quadtree encoded pictures there would typically be half

*Permanent address: The Computing Laboratory, The University, Canterbury, Kent CT2 7NF, UK.

CCC–0010–4620/83/0026–0083 $04.50

a dozen images active at any one time and memory space for pictures, though not desperately short, was not plentiful. For our work we thought it better to save space than time, and although this will not always be so, the time requirements have seemed very reasonable with a simple linear storage structure (see below).

2.2 Linear structures

If the producers and consumers of treecoded images can agree about the way each value in the code shall be interpreted there is no need to make the tree structure itself explicit in the code. We can use juxtaposition in the code to mean 'next to' in a treecoded image sense. A sequence ($A\ B\ C\ D$) of values could represent the four leaves of a single subdivided square, given in an order set by convention. If, say, B was not a leaf but was itself subdivided into ($W\ X\ Y\ Z$) then we could write $A\ B$ ($W\ X\ Y\ Z$) $C\ D$ or A (B) $C\ D\ W\ X\ Y\ Z$. In the first case every subdivided node is followed by its subdivisions—a depth first method of storing the data. In the second case each level is described in turn—a breadth first scheme. Which is best? There is no doubt that the algorithms described in this paper favour depth first because the passing down of context during recursions requires no specific action in the traversing procedures. However a breadth first method has its attractions—the point made in Section 1 about an animation application would have a stronger effect with a breadth first structure. We have not studied this issue with any thoroughness.

2.3 Treecodes

Following the first of the suggestions in Section 2.2 above we now describe a linear code which specifies a quadtree in depth first order. It makes no pretensions of generality, but has found use for several different projects. Each node, whether non-terminal or leaf, has a value given in 4 bits. One additional bit indicates which sort of node it is and these 5 bit quantities are stored in byte fields in memory. A sequence of bytes is read as follows:

1. Think of a square area.
2. Get next byte.
3. If leaf, colour the square with value in 4 bit field.
4. If non-terminal, subdivide the square and return to step 2 four times for the bottomleft, topleft, bottomright, and topright squares.

Thus the code 20, 4, 18, 2, 0, 5, 1, 7, 3 is understood to represent the quadtree of Fig. 1. The non-terminals are 20 and 18 in this case, representing the average value of the whole image (16 + 4) and the top left quadrant (16 + 2) respectively.

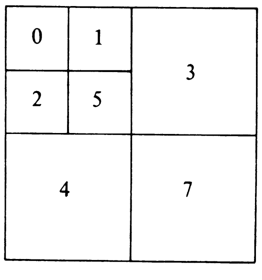


Figure 1. Quadtree image from treecode 20 4 18 2 0 5 1 7 3.

2.4 Leafcodes

The basic idea in leafcodes is to treat the square image areas represented by the leaves in quadtrees as entities separate from the structure of the treecode while retaining the relation to quadtrees by requiring that only the square areas corresponding to possible quadtree leaves are permitted; in general arbitrary square areas must be broken up into the leaves that cover them. The recursive structure of the treecode determines both the size and the position of a particular leaf. In leafcodes this information is carried directly in the code for each leaf together with its colour. The size of the leaf can be given by the number of pixels along the edge of the square (which will be a power of two) or implicitly by the recursive depth in the quadtree. The position of the leaf is determined by the x, y co-ordinates of the pixel in the lower left hand corner of the leaf which will be called the origin of the leaf. Thus a 512 × 512 picture has a number space as shown in Fig. 2.

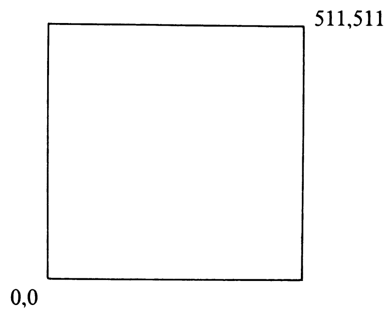


Figure 2. Number space for 512 × 512 picture.

Consider the following algorithm for the specification of the position of a pixel:

1. Divide the image into quadrants.
2. Choose the quadrant containing the pixel and repeat steps 1, 2 until no further subdivision is possible.
3. Write the sequence of quadrant numbers out in base four using 0, 1, 2, 3 to identify the bottomleft, topleft, bottomright and topright quadrants, respectively.

When the leaves of a treecode are represented in this way the leaf co-ordinates are seen to be in strictly increasing order. In the example shown in Fig. 3 the shaded square has leaf co-ordinates 031 (001101 in binary).

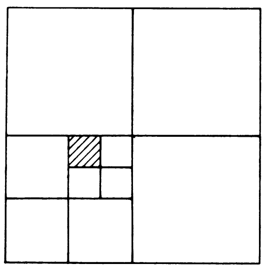


Figure 3. Leaf with leafcode 001101 is shaded.

The space requirement for leafcode can compare favourably with that for the corresponding treecode owing to two factors. For quadtrees of a few hundred nodes or more

$$\text{number of leaves} = (3/4) \times \text{number of nodes}$$

very closely. Also leaves of the background colour can simply be omitted from the list of leaves and inferred when required. For line images on a uniform background the space requirement is then only about 15% greater than that for the corresponding treecode.

3. OPERATIONS ON TREECODES

3.1 Walking a treecode

Many operations on quadtrees can be done by procedures whose basis is the same: walking over the tree with a simple recursion every time that a non-terminal node is encountered. Such a walk is achieved by

```
let walk() be
$( for i = 0 to 3
   $( pointer := pointer + 1
      if picture % pointer > 15 walk()
   $)
$)
```

The walk is started by setting the static variable pointer to the address of the first value in the tree, and then going:

```
if picture % pointer > 15 walk()
```

Note that the walk terminates when the tree it is started on expires. It can therefore be used to traverse any subtree in the code just by starting off in the right place. On exit the value of pointer is just prior to the next node that follows the tree which has been walked (i.e. on the last leaf of the tree walked). In the following sections the basic walk is embellished by statements which make use of the values read from the input tree or inferred from the execution path.

3.2 Display an image by recovering square coloured areas from the treecode

Suppose that a picture represented by its quadtree is to be drawn on a simple graphics display. The display is driven by a procedure, not given here, which draws a single coloured square of specified size and position on the screen. It is called in the form *square(x, y, side, colour)*, where *x* and *y* specify the lower left corner of the square. All that is needed is a version of walk, like that shown above, but with something to keep track of the position of each node and the size of the square area which it represents. This procedure is

```
let putout (x, y, scale) be
$( for i = 0 to 3
   $( pointer := pointer + 1
      if (scale = minsize) ∨
            ((scale > minsize) ∧ (picture % pointer < 16))
      square (x + (i > 1 → scale. 0), y + ((i ∧ 1) = 0 → scale, 0),
                                               scale, picture % pointer)
      if picture % pointer > 15
      putout (x + (i > 1 → scale, 0), y + ((i ∧ 1) = 0 → scale, 0), scale/2)
   $)
$)
```

The static variable *minsize* allows the resolution of the image to be controlled. If the picture is defined to a resolution of, say, *Z* and *minsize* is set to *Z* or smaller, then the output will be a picture using all the leaf values in the tree. If *minsize* is larger than *Z* then the values in non-terminals at resolution *minsize* are used in place of the smaller leaves which make up those nodes. Since these contain the averages of the leaf values beneath them, the form of anti-aliasing described in Section 1 is obtained very simply.

3.3 Rotate and reflect a treecode

Rotating a treecode by 180° is a permutation of the following sort:

$$\begin{array}{cc} 1 & 3 \\ 0 & 2 \end{array} \longrightarrow \begin{array}{cc} 2 & 0 \\ 3 & 1 \end{array}$$

If the input tree is scanned forwards while the output tree is constructed backwards, most of the work will be done. However, each cell subdivision is preceded in the input scan by the subdivided node value whereas the output tree wants it afterwards. The recursive call is used to introduce the required postponement in issuing the node value and the entire algorithm is very simple:

```
let reverse() be
$( let nodevalue = ?
   for i = 0 to 3
   $( inpointer := inpointer + 1
      nodevalue := inpicture % inpointer
      if nodevalue > 15 reverse()
      outpicture % outpointer := nodevalue
      outpointer := outpointer − 1
   $)
$)
```

Inpointer and *outpointer* are initialized to the start and end of the input and output trees, respectively (since the output tree will be equal in length to the input tree we know where to set outpointer so that it is at the end of the output tree).

Rotation of a quadtree by 90° is a permutation of the following sort:

$$\begin{array}{cc} 1 & 3 \\ 0 & 2 \end{array} \longrightarrow \begin{array}{cc} 0 & 1 \\ 2 & 3 \end{array}$$

and is considerably more difficult to do. The method presented here processes the tree one level at a time and makes multiple scans of the input. Each scan is to skip over the subtree which separates one node from the next one which is at the same level in the input data. Using it, the four nodes at one level are discovered together with the lengths of the intervening subtrees. The permutation at that level is then written out and the next levels inspected recursively. The procedure is:

```
let rotate (pointer) be
$( let res = vec 3
   let pbefore = vec 3
   let pafter = vec 3

   for i = 0 to 3
   $( pbefore!i := inpointer + 1
      res!i := inpicture % (inpointer + 1)
      scan()
      pafter!i := inpointer − pbefore!i
   $)
```

```
                                          // now start the permutation
    outpicture % pointer := res ! 2      // quadrant 2 to quadrant 0
    if pafter!2 ne 0
    $( inpointer := pbefore!2
       rotate (pointer + 1)
    $)
    pointer := pointer + pafter ! 2 + 1

    outpicture % pointer := res ! 0      // 0 goes to 1
    if pafter ! 0 ne 0
    $( inpointer := pbefore ! 0
       rotate (pointer + 1)
    $)
    pointer := pointer + pafter ! 0 + 1

    outpicture % pointer := res ! 3      // 3 goes to 2
    if pafter ! 3 ne 0
    $( inpointer := pbefore ! 3
       rotate (pointer + 1)
    $)
    pointer := pointer + pafter ! 3 + 1

    outpicture % pointer := res ! 1      // 1 goes to 3
    if pafter ! 1 ne 0
    $( inpointer := pbefore ! 1
       rotate (pointer + 1)
    $)
$)
```

Of course other permutations can be used in the same manner to rotate the other way, reflect or flip the image. The permutation for left to right reflection is, for example

$$\begin{matrix} 1 & 3 \\ 0 & 2 \end{matrix} \longrightarrow \begin{matrix} 3 & 1 \\ 2 & 0 \end{matrix}$$

and the changes to the procedure given above to achieve this are obvious.

The procedure given above is space efficient and sufficiently quick to be useful on real pictures (see Section 4). Quicker methods are possible, although at a cost in memory requirements. One approach is to rebuild the tree as a pointer structure and then permute the nodes while compressing the tree back to its linear form. For a tree of depth D the speedup should be of the order of $D/2$, but workspace is increased fourfold (if wordlength is four bytes). We did not think the trade favourable in the environment we were in.

3.4 Translate a treecode

In general, moving the image through a displacement x, y will involve a wholesale reorganization of the tree. However some observations can be made:

1. Large homogeneous areas stay large and homogeneous so the compaction which treecodes permit must still be available.
2. A leaf which is translated by some multiple of its side remains a leaf.
3. Growth of tree detail is along the edges of incursions so it is clear where to start looking for new nodes.

Consider for example Fig. 4 showing an incursion into the left edge of a leaf by a feature being moved right by a distance x. We have to replace the leaf by a sub-tree having subdivisions concentrated on its left side in which to write the new data extending at most by x into the original leaf area. We do this by keeping a leftstate vector of the values to displace into the newly constructed sub-tree and making the total translation by a succession of leaf sized moves of diminishing size.

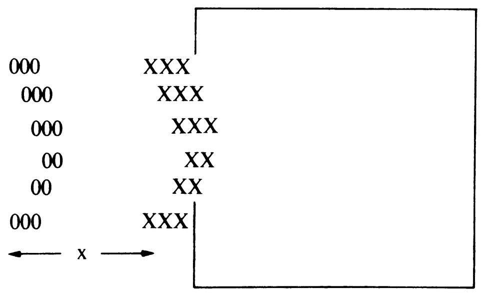


Figure 4. Incursion of an image feature into a leaf.

Moving up is accomplished in the same manner. However, moving down or to the left is impossible. We can manage without such moves by reversing the image before a down-left move, making the move (which will then of course be up-right), and finally reversing the image again. Reverse is pretty quick.

Another, probably better, way to do translation (and indeed some other transformations, such as zoom, i.e. scaling) is to do it on the leafcode for the image. The displacement of a picture represented by its leafcode is evidently achieved by displacing each leaf independently. Adding an x, y displacement to a leaf co-ordinate will in general not result in a valid leafcode of course, owing to the need for leaves to register with the image space in a particular way. What has to be done is to break the translated squares back into leaves and then sort them to restore their relative ordering in the tree and compress the result by amalgamating identical quadrants. This approach is quite appealing and we expect to report in a later paper on a comparison of treecode and leafcode transformations.

3.5 Truncate a treecode

Below a prescribed depth the branches and leaves of the quadtree are removed and all nodes at the lowest remaining level are turned into leaves. This is accomplished quite simply by a walk which keeps a record of the depth reached. At and above the specified maximum depth the node values of the input picture are passed to the output picture and pointers in each are stepped on (note that the input picture can also be the output picture because the reading position is never prior to the writing position). When a non-terminal node at the maximum depth is encountered, walk is called, thus stepping the input pointer to the point just before the next node at the same depth. In this case the non-terminal is converted to a leaf before being written to the output. As the truncation proceeds, the situation may arise where four leaves in a non-terminal have the same value, one or more of the leaves having just been created. In that case compression should occur with the non-terminal becoming a leaf in place of the four equal valued leaves. The algorithm below takes care of this case with the test that follows the main for-loop in the walk:

```
let truncate (depth, average) = valof
$( let pointer = ?
   let nodevalues = vec 3

   for i = 0 to 3
   $(  inpointer := inpointer + 1
       outpointer := outpointer + 1
```

```
    test inpicture%inpointer < 16 then
    $(     nodevalues! i := inpicture%inpointer
           outpicture%outpointer := nodevalues! i
    $)
    or
    test depth < maxdepth then
    $(     pointer := outpointer
           nodevalues! i := truncate (depth + 1, inpicture%inpointer)
           outpicture%pointer := nodevalues! i
    $)
    or
    $(     nodevalues! i := inpicture%inpointer ∧ 15
           outpicture%outpicture := nodevalues! i
           walk()
    $)
  $)
  test nodevalues! 0 = nodevalues! 1 = nodevalues! 2 =
                                  nodevalues! 3 ∧ nodevalues! 0 < 16
  then
  $( outpointer := outpointer − 4
     resultis nodevalues! 0
  $)
  or
  resultis average
$)
```

Truncate is called with *inpointer* at the input picture root, *depth* set to one, and *average* set to the value of the root node.

3.6 Merge, mask and invert a treecode

Combination of quadtrees is surprisingly simple. Merging two trees is an operation similar to OR in which the output value in any pixel is the larger of the two input values. It is done by examining the nodes of each input tree once, stepping forwards in each tree with a stride chosen to keep the two scans synchronized. Thus if at a certain stage the next node in each tree is a leaf node then the leaves are combined to form an output node (this may or may not be a leaf because adjacent nodes of the same value can later be amalgamated to form bigger leaves). If the first tree contains a non-terminal node and the second a leaf, then the first scan recurses while the second waits for the first to 'catch up'. Similarly if the second contains a non-terminal while the first contains a leaf, then the second scan recurses. If both trees contain non-terminals, then both scans recurse. The algorithm is plainly seen in:

```
  let quadmerge() = valof
  $( let pointer = outpointer + 1
     let res = vec 3                   //park four subdivisions here
     let a = inpica%inpointera         //value from first input tree
     let b = inpicb%inpointerb            //and one from the other

     outpointer := outpointer + 1         //step output pointer

     if (a < 16) ∧ (b < 16)            //both leaves?
**   $( curpic%outpointer := a > b − > a, b     //merge them if so
        resultis curpic%outpointer              //and exit
     $)
     test a < 16 then              //must subdivide b if a is leaf
     for i = 0 to 3                //so do four subdivisions of b
     $( inpointerb := inpointerb + 1
        b := inpicb%inpointerb
        res! i := quadmerge()
     $)
     or
     test b < 16 then              //well then a must be subdivided
     for i = 0 to 3                //so do its four subdivisions
     $( inpointera := inpointera + 1
        a := inpica%inpointera
        res! i := quadmerge()
     $)
     or                                  //or else both subdivided
     for i = 0 to 3                      //so do them both
     $( inpointera := inpointera + 1
        inpointerb := inpointerb + 1
        a := inpica%inpointera
        b := inpicb%inpointerb
        res! i := quadmerge()
     $)
     if (res! 0 = res! 1 = res! 2 = res! 3) ∧ (res! 0 < 16) //see results
     $( curpic%pointer := res! 0          //if all the same we can
        outpointer := pointer               //amalgamate the cells
        resultis res! 0                     //& pass up the result
     $)
     for i = 0 to 3 res! i := res! i ∧ 15           //force to leaves
     curpic%pointer := 16 ∨ (res! 0 + res! 1 + res! 2
                                    + res! 3 + 2)/4  //average
     resultis curpic%pointer              //and pass up the result
  $)
```

The procedure is called with picture vectors and initial pointer values held in static variables. At the statement where leaves are actually combined alternative connectives can be substituted to combine trees in different ways. Thus masking two trees, an operation similar to AND, we could do by replacing the statement at ** by:

$$curpic\%outpointer := a < b \rightarrow a, b$$

The output tree has its non-terminals properly averaged so giving the anti-aliasing effect referred to earlier and the whole operation is an agreeably cheap computation.

3.7 Building trees for polygons and circles

Consider first the construction of a quadtree to represent a filled convex polygon. The tree is built by a recursive procedure which compares probe cells with the polygon boundary, returning for each cell an *outside*, *inside*, or *neither* result. If inside, the cell is coloured with the inside colour. If outside, it is coloured with the outside colour, and if neither it is subdivided and the process repeated.

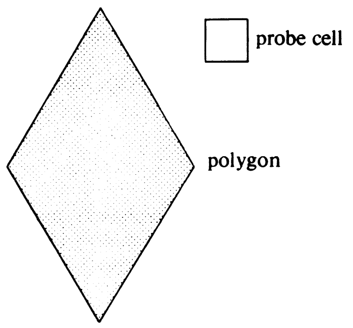


Figure 5. Polygon and probe cell.

The polygon (Fig. 5) is defined by a chain of edges tracing the boundary in, say, an anticlockwise direction and the in/out/neither condition is evaluated as follows:

For each edge:

1. Test if all cell vertices are on the 'inside' side of the edge. If so the cell is inside that edge.
2. Test if cell is outside the bounding box for that edge. If so the cell is outside that edge.
3. Test if all cell vertices are on the 'outside' side of the edge. If so the cell is outside that edge.
4. The result is *neither*.

If for all edges the test returns inside, the cell is inside the polygon. If for any edge the test returns *neither*, then the cell intersects the polygon and has to be subdivided. Otherwise the cell is outside the polygon.

Test 1 causes the condition to be evaluated incorrectly if the polygon is concave, because some probe cells actually inside the polygon are not on the 'inside' side of every edge. An example is shown in Fig. 6.

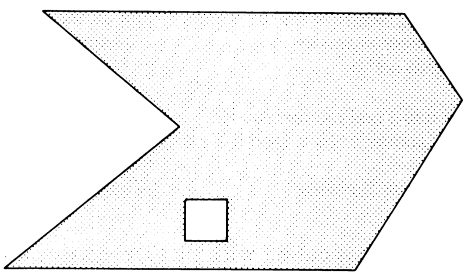


Figure 6. Probe cell not inside all edges.

The equation for the line on which an edge of the polygon falls is given by

$$ax + b - cy = 0$$

where

$$a = y2 - y1$$
$$b = y1 \cdot x2 - y2 \cdot x1$$
$$c = x2 - x1$$

and the edge joins vertices ($x1$, $y1$), ($x2$, $y2$). For an arbitrary point X, Y we have

$$aX + b - cY = m$$

and m then indicates the relative disposition of the point and the line. If $m = 0$ the point is on the line, if $m < 0$ the point is on the 'inside' side of the line and if $m > 0$ it is on the 'outside' side.

A procedure *testpolygon* which works in this way is then used in the tree constructing algorithm as follows:

```
let buildtree (polygon, probecellx, probecelly, scale) = valof
$( let pointer = outpointer + 1
   let testresult = ?
   let nodevalue = ?

   outpointer := outpointer + 1
   testresult := testpolygon (polygon, probecellx, probecelly, scale)
   picture%outpointer := testresult = outside → 15, 0
   unless testresult = neither resultis testresult
   scale := scale/2
   nodevalue := scale = minsize → 7,
      (buildtree (polygon, probecellx,
                        probecelly, scale) +
       buildtree (polygon, probecellx,
                        probecelly + scale, scale) +
       buildtree (polygon, probecellx + scale,
                        probecelly, scale) +
       buildtree (polygon, probecellx + scale,
                        probecelly + scale, scale) + 2)/4
   picture%pointer := scale = minsize → nodevalue, 16 ∨ nodevalue
   resultis nodevalue
$)
```

The procedure is called with *outpointer* at its initial value, *scale* at its maximum value, and *probecellx* and *probecelly* both zero (lower left corner is the quadtree origin). In the process of building the tree the anti-aliasing values are produced as a by-product and recorded in non-terminal tree nodes.

If only the boundary is to be converted to quadtree form the procedure above can be simplified slightly. Of course it then works for concave polygons perfectly well, or for that matter unclosed or disconnected linechains.

The procedure *testpolygon* can be replaced by others which test for a differently shaped object against the given probecells so that *buildtree* can cope with images other than of polygons. For example, a circular disc could use a procedure called *testcircle*, say; after translating the image space so that the disc is centred on the origin, it goes:

1. If all probecell vertices are closer to the origin than the radius, R, then the cell is inside the disc and a result of *inside* is returned.
2. If any vertices are closer to the origin than R then, since test 1 failed, the result is *neither*.
3. If the probe cell straddles both axes, the result is *neither*.
4. If the probe cell straddles the x (y) axis and the y-co-ordinate (x-co-ordinate) of any vertix of the cell is less than R, the result is *neither*.
5. The result is *outside*.

The first two tests have an obvious interpretation. The third test deals with the case when the cell overlaps the disc, but all vertices are outside the circle. Test 4 is necessary because the disc can penetrate the cell along an edge even when all vertices are outside the circle.

3.8 Area filling in general

Given a quadtree for an arbitrary closed boundary the task of filling the interior of the boundary is quite different from that just described. The tree is already correctly structured and the algorithm has only to colour leaves in it. Well known methods which start by seeding an interior point and growing colour up to the boundary need modification when used with a quadtree because the adjacency tests are complicated. It is easy to find what lies above and to the right of a particular node, but not what is below or to the left. What our method does is sweep the tree colouring upwards and to the right, then turn the image upside down (using reverse as described above). This is done repeatedly until no changes in colour occur. Any number of seeds can be given to start the algorithm off—if well chosen the number of sweeps can be minimized, but even with a single poorly positioned seed most images are coloured in a few seconds. The basic idea is shown below:

```
let colourwalk (x, y, scale) be
$( let nodevalue = picture%pointer
   for k = 0 to 3
   $( pointer := pointer + 1
      test nodevalue > 15
      then colourwalk (x, y, scale/2)
      or
      if colourcontact (x, y, nodevalue, scale)
      picture%pointer := 15
      outpicture%outpointer := nodevalue
      outpointer := outpointer - 1
      switchon k into
      $( case 0: y := y + scale
            endcase
            case 1: $x := x + scale$
                    $y := y - scale$
                endcase
            case 2: $y := y + scale$
         $)
      $)
$)
```

Two static state vectors, *leftstate* and *bottomstate*, carry the information about coloured areas and boundary leaves to the left and below the node being inspected. These are updated by the function *colourcontact* each time a change is made to the colour of a leaf. There are three cases to be distinguished:

1. The leaf is already filled. The state vectors must be brought up to date and the value *true* returned.
2. The leaf is part of the boundary. Update the state and return *false*.
3. The leaf is neither filled nor part of the boundary. The state vectors are consulted to see if there is any contact with a coloured leaf; only leaves that have already been seen in the current pass can have any effect. According to the result the leaf may either be filled and *true* returned or *false* is returned. The state is updated in either case.

Colourwalk is only concerned with setting the leaf values for interior points and will leave the tree with incorrect values in non-terminals. A final step therefore is to calculate the correct averages for these non-terminals as necessary.

4. ASSESSMENT

4.1 Space and time comparisons

The efficiency of treecodes in representing arbitrary pictures is not easy to assess. For contrived pictures one can find that the code behaves well or badly, according to desire, and no very good notion of a 'canonical' picture occurred to us. Each of the examples shown in Fig. 7 has its tree size appended and the compression is seen to vary from 25 upwards. Although these are high numbers, nobody could claim that the pictures are in any useful sense representative. What is important, however, is the way the algorithms exploit shortness of code to deliver quickness of answers. It is unusual to find that an encoding that saves space will at the same time speed computations. The examples were used to compare the speeds of some of the operations described above on a Motorola M68000 microprocessor using a 32-bit implementation of BCPL as the programming language and without any serious attempt at optimizing the code. The compiler benchmark runs about 3 times faster on a VAX11/780 so we would expect a corresponding improvement in quadtree processing times on such a machine.

First experiment. For an image space 512 × 512 we built quadtrees for three circular outlines, each of radius 75 pixels, positioned respectively at (206,284), (306,284) and (256,198). The generation took 1.8 s each and gave tree sizes of 2381, 2381 and 2429 nodes. These trees are identified as A, B and C in the table below:

Operation	Resolution (pixels)	Tree size (nodes)	Time (s)
D := merge (A, B)	1	3429	1.1
E := merge (D, C)	1	6893	1.6
F := invert (E)	1	6893	0.3
G := reverse (F)	1	6893	0.5
H := fill (E, a, b, c, w)	1	6893	6.5
I := fill (E, a, b, c)	1	6893	6.5
J := fill (E, w)	1	6893	6.1
K := rotate (E)	1	6893	5.0
E2 := truncate (E)	2	3325	0.5
K2 := rotate (E2)	2	3325	2.2
H2 := fill (E2, a, b, c, w)	2	3325	3.2
E4 := truncate (E)	4	1565	0.4
K4 := rotate (E4)	4	1565	0.9
H4 := fill (E4, a, b, c, w)	4	1565	1.6
E8 := truncate (E)	8	653	0.3
K8 := rotate (E8)	8	653	0.4
H8 := fill (E8, a, b, c, w)	8	653	0.7

The seeds given to fill were *a*, *b*, *c*, *w* which were the centres of the circles A, B and C together with a point in their common region (in fact 256,256).

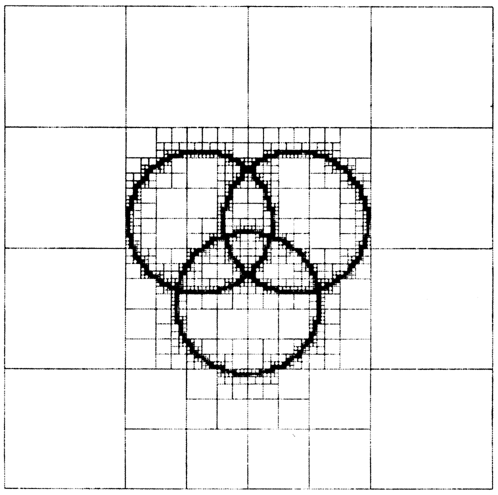
Second experiment. Using a stylus-tablet some sketches of simple line features were digitized and converted to quadtree form. One of these is the spiral shown in Fig. 7 (referred to as picture L in the experiment). We also had some polygonal outlines of some high quality alphabets designed by David Kindersley. A short text in one of these alphabets was converted to quadtree form and is named picture M.

Operation	Resolution (pixels)	Tree size (nodes)	Time (s)
N := fill (L, seed)	4	2205	3.6
P := fill (~M, seed)	1	9161	10.7
Q := inv (~M)	1	9161	0.4
R := merge (Q, N)	1	6881	2.4
S := rotate (R)	1	6881	4.8
T := merge (R, S)	1	10213	2.9

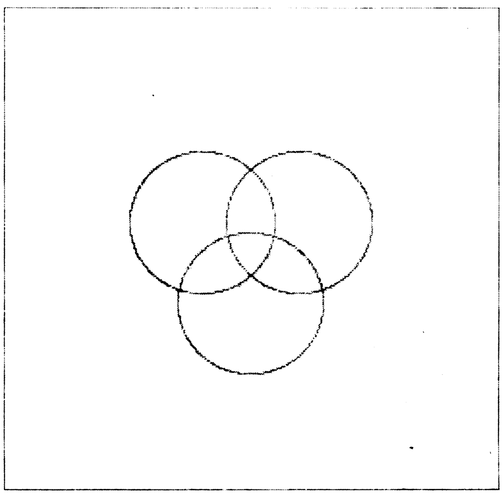
The final illustration in Fig. 7 shows an image composed from three circular discs at 256 × 256, 512 × 512, and 1024 × 1024 resolutions built as a single quadtree and displayed at 512 × 512. Although not easily visible in the reproduction of this paper the actual image quality we achieved impressively demonstrates the effectiveness of the anti-aliasing in the rendering of fine detail.

4.2 Future work

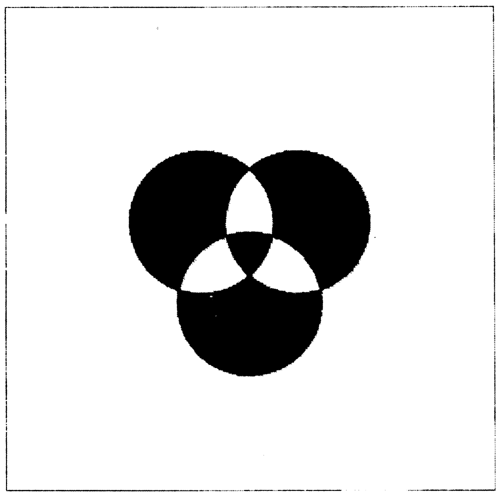
The present work describes how to operate directly on treecode. Readers who want to see the detailed programs should write for a copy. Some treecode operations have not yet been coded by us in a fully satisfactory form and there is little to report on the breadth first storage



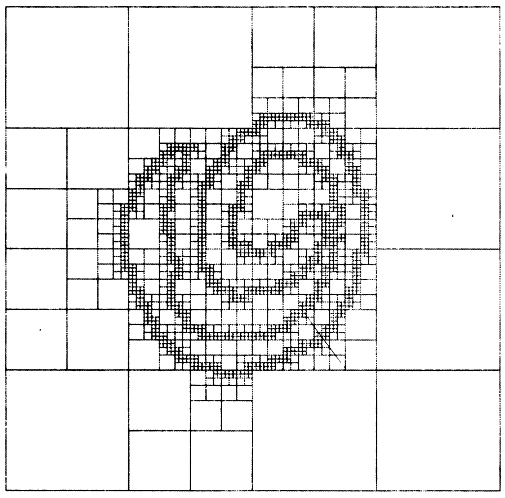
Picture E:
Quadtree outlines
Treesize 6893 nodes



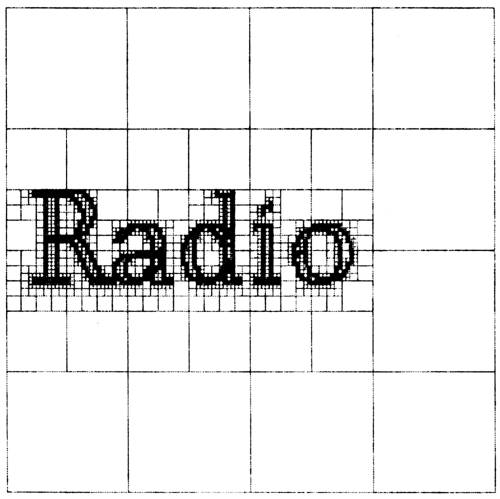
Picture E:
Treesize 6893 nodes



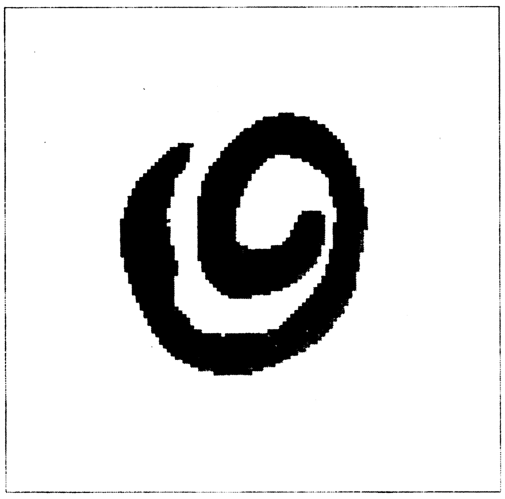
Picture H:
Treesize 6893 nodes



Picture L:
Quadtree outlines
2205 nodes

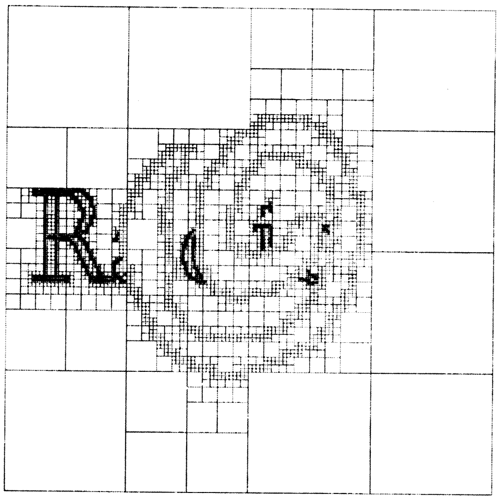


Picture M:
Quadtree outlines
9161 nodes

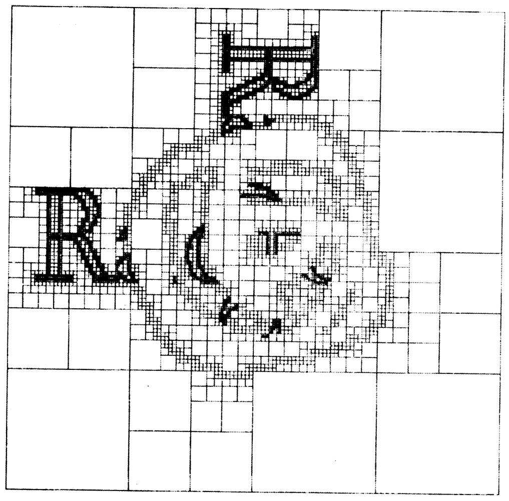


Picture N:
2205 nodes

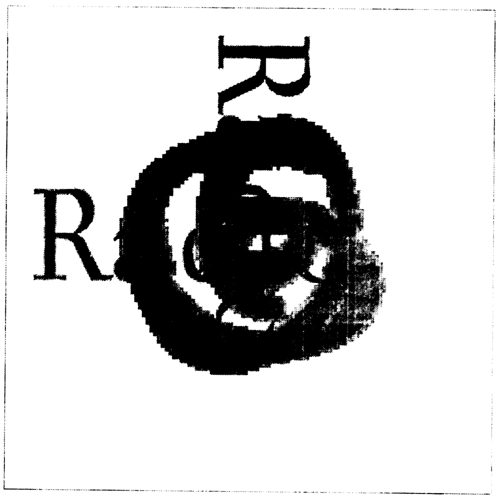
Figure 7.



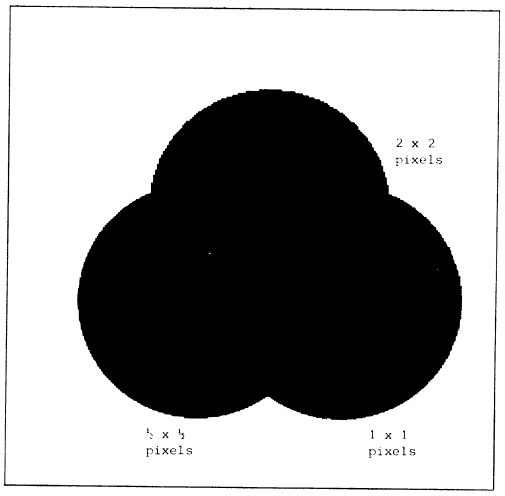
Picture R:
Quadtree outlines
6881 nodes



Picture T:
Quadtree outlines
10213 nodes



Picture T:
10213 nodes



Anti-aliasing demonstration

Figure 7.

structure referred to in Section 2.2. Work on these topics continues.

For some operations treecode is an inappropriate data structure as it does not always support simple algorithms that lead to fast computation. Leafcode is particularly appropriate for some such operations e.g. translation or scaling ('zoom'). These possibilities are in the process of being investigated and we hope to report on them in the near future.

Acknowledgement

We are grateful to Jon Fairbairn for the use of his program which was used in the preparation of the figures produced on a Versatec plotter.

REFERENCES

1. R. D. Dyer, A. Rosenfeld and H. Samet, Region representation: boundary codes from quadtrees. *Communications of the ACM* **23**, 171–179 (1980).
2. G. M. Hunter and K. Steiglitz, Linear transformation of pictures represented by Quadtrees, in *Computer Graphics & Image Processing*, Vol. **10**, 289–296 (1979).
3. G. M. Hunter and K. Steiglitz, Operations on images using Quadtrees. *IEEE Transactions on Pattern Analysis & Machine Intelligence* **PAMI-1**, 145–153 (1979).
4. H. Samet, Region representation: raster to quadtree conversion. *Computer Science TR-766*, University of Maryland, College Park, Maryland (May 1979).
5. H. Samet, Region representation: quadtrees from boundary codes. *Communications of the ACM* **23**, 163–170 (1980).

Received June 1982