The leaves of a quadtree can be specified by their colour, size, and position in one 32-bit word of data. Considerable advantages accrue for the manipulation of images if the restriction that the images be tiled by their quadtree leaves is relaxed so that any non-overlapping square areas are permissible; the code for such a set of squares of arbitrary size and registration we term squarecode. Algorithms are described for intercode conversion and for some basic geometrical manipulations of images.
For simplicity we assume that the image number space is a square with side some power of two. The leaves of the quadtree correspond to square areas of the image and have side some power of two; in addition these square areas must be positioned in the image number space so that they are permissible quadtree leaves.

2.2 Leafcode

As we have remarked, the leaves of a quadtree correspond to square areas of the image. In the treecode representation of an image the size and position of these areas are determined by their place in the treecode structure. Another way of specifying these areas (the leaves) is to give their colour, size, and position explicitly in a 32-bit word. We call the unordered set of leaves specified in this way leafcode. In our system we have assigned the 32 bits of each word as shown in Fig. 1, with four bits for the colour, eight bits for the side less one, and ten bits for each co-ordinate value.

There are several remarks to be made about this choice and arrangement. First, it allows a maximum imagesize of 1024 x 1024 pixels provided that we treat the trivial case of a uniformly coloured screen with care: care is necessary because there are not enough bits for a side greater than 512 pixels (four bits have been assigned to the colour). The necessity for the choice of side rather than the depth of recursion, which would require only four bits, will become clear when we consider squarecode. The x and y co-ordinates are for the bottom left pixel in the square; we shall refer to this pixel as the origin of the leaf.

2.3 Squarecode

Consider a two dimensional image space and the (finite) set of all possible squares that it contains which are aligned with the image space; for simplicity the image space is square. The set of all possible leaves is a subset of the set of all possible squares. If a square is operated on in such a way that it is transformed into a different shape, e.g. translation part way across the picture boundary, it must be broken up into a set of squares. As with a leaf in leafcode, in squarecode each square is specified by its colour, size, and origin; these parameters fit onto a 32-bit word as described in the previous subsection. The necessity for side as one of the parameters is now manifest: this parameter can vary from unity to a square with side some power of two. The leaves of the quadtree correspond to square areas of the image and have side some power of two; in addition these square areas must be positioned in the image number space so that they are permissible quadtree leaves.

3. INTERCODE CONVERSION

The image is stored externally as treecode whereas all operations inside the system are performed on square-code. Accordingly, when an image is read into the system the treecode must be transformed into squarecode and vice versa when the image in the system is put into the external store.

3.1 Treecode into leafcode

In order to read the tree data into the system as leafcode the basic walk over the quadtree is dressed up as follows:

```c
let treecodeleaf(x, y, scale) be
$\{$
  let nodevalue = ?
  for k = 0 to 3
  if nodevalue >= read() 
  test nodevalue < 16 then
  unless nodevalue = 0
  leafptr := leafptr + 1
  curpic := leafptr := (nodevalue << 28) | ((scale - 1) << 20) | (x << 12) | y

  switch (case)
  case 0: y := y + scale
  endcase
  case 1: x := x + scale
  y := y - scale
  endcase
  case 2: y := y + scale

$\}$
```

The `switch` statement steps the co-ordinates over the origins of the quadtree nodes as the walk proceeds. The procedure is invoked with the static variable `leafptr` at the root of the quadtree. In order to save space leaves of background colour (assumed here to be zero though any value could be chosen) are not recorded; there is no loss of information of course.

3.2 Laced-co-ordinates

The algorithm for conversion of squarecode into treecode depends on the use of what we call laced-co-ordinates. Woodward7 has used a related scheme to address quadtree leaves; his quadrant numbering convention is different from ours and consequently lacks the key ordering property that we exploit in the sorting operations used in the algorithms described below. Consider the following algorithm for the specification of a pixel laced-co-ordinate:

1. Recursively divide the image into quadrants until the pixel is reached.
2. Write the sequence of quadrant numbers, for those quadrants that are divided, in binary using 00, 01, 10, 11 to identify the bottom left, top left, bottom right, and top right quadrants, respectively.
3. Follow this sequence by the quadrant number of the pixel; these are the lowest significant bits in the laced-co-ordinate.
4. Concatenate the sequence to give the laced-co-ordinate.

An example will make this clear. Take a square image with side 16; the pixel at x = 5, y = 9 has laced-co-ordinate 0110011 because the sequence of divided quadrants in the recursion is 01, 10, 00 (by 2 above) and the pixel is the 11 quadrant of the last divided quadrant (by 3 above). This is shown in Fig. 2.
A little consideration shows that the laced-co-ordinates of the leaves of a quadtree are in strictly increasing numerical order when the leaves are ordered according to a depth first walk over the quadtree. Thus sorting leafcode, initially with the words that represent the leaves in arbitrary order, into the order in which the laced-co-ordinates of the leaves are in ascending order is the key to its conversion into treecode. The reason for the choice of numbering convention for the quadrants now manifests itself: it is to ensure that the depth first order of the leaves of a quadtree is the same as the numerical order of their laced-co-ordinates.

We remark that the laced-co-ordinate of a pixel in binary has the same number of bits as do the x and y co-ordinates; indeed it is easy to see that the laced-co-ordinate can be obtained from the x and y co-ordinates by interlacing their bits as shown in Fig. 3.

Figure 2. The pixel with laced-co-ordinate 0110011 is shaded.

3.3 Squarecode into treecode

The algorithm for this operation is in four parts:

1. Convert the squarecode, square by square, into leafcode in a buffer; as this is done replace the x, y co-ordinates of each leaf by the equivalent laced-co-ordinate.
2. Sort the leaves into numerical order, i.e. into the correct order for the leaves of treecode.
3. Compress the leaves: if four leaves have the same colour and size and together are equivalent to a leaf of this colour one level up in the tree then they can be replaced by the larger leaf.
4. The treecode is constructed from the ordered leaves and written to a file; this step requires a second buffer. These steps are described in the next four subsections.

3.3.1 Make ready for sorting. Examine the squarecode square by square. If the square is not a quadtree leaf then procedure makeleaves is called: this recursively divides the imagespace into quadrants (leaves) each of which is tested to see if it is:

1. Outside the square in which case it is thrown away.
2. Inside the square, in which case it is put out as leafcode with the x, y co-ordinates replaced by the equivalent laced-co-ordinate, cf. Fig. 3.
3. Neither, i.e. either the leaf totally encloses the square or it overlaps the boundary of the square. In this case the leaf is (recursively) subdivided again unless the pixel level has been reached in which case the process stops.

The criterion for a square to be a leaf is that the square has a side which is both a power of two and is not too large for the registration of the square, i.e. its position. Let x, y, and sidelesssone be the geometric parameters of the square in binary bit patterns then if

\[(x \le y) \& sidelesssone = 0\]

the registration is all right. For the square to be a leaf sidelesssone + 1 must be a power of two.

The procedure makeleaves has the following structure:

\[
\text{let makeleaves}(x, y, \text{side}) \text{ be } \begin{cases} 
\text{testresult} = \text{testinout}(x, y, \text{side}) & \\
\text{unless testresult = outside do} & \\
\text{testresult = inside then} & \\
\text{if (buffer! outpointer = colourbits) \& ((side - 1) < 20)} & \\
\text{(lacedcoorfrom}(x, y) & \\
\text{outpointer += outpointer + 1) & \\
\text{) & \\
\text{or} & \\
\text{unless side = 0) & \\
\text{side := side/2} & \\
\text{makeleaves}(x, y, \text{side}) & \\
\text{makeleaves}(x, \text{y + side, side}) & \\
\text{makeleaves}(x + \text{y, side, side}) & \\
\text{makeleaves}(x + \text{side, y + side, side}) & \\
\text{makeleaves}(x + \text{side, y + side, side}) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \\
\text{) & \

Figure 3. Arrangement of co-ordinate digits in laced-co-ordinate.

3.3.2 Sorting. The 32-bit words, each representing one leaf, are sorted into the ascending numerical order of their laced-co-ordinates. As remarked in the discussion of Fig. 2 this puts the leaves into the correct order for their quadtree.

For the operations that are available there seems to be no reason to expect any systematic mixing of the leaves to occur. We used quicksort for lack of any rational argument against doing so.
3.3.3 Compression of the leaves. The fourth leaf in the
recursive subdivision of a quadrant is called a fourth-

quadrant. Let \( \text{maxdepth} \) be the maximum depth for the
chosen image size, e.g. 9 for image size 512. Consider a

quadrant at depth \( \text{maxdepth} - n \): it divides into

four quadrants at \( \text{maxdepth} - n \) and their laced-co-
ordinates differ only in their least significant bits, namely

\( x_n \) and \( y_n \) (see Fig. 3). A fourth-quadrant at depth

\( \text{maxdepth} - n \) has both \( x_n = 1 \) and \( y_n = 1 \).

The basic structure of a simple algorithm for compress-

ing four leaves into one, where possible, is given by

1. Find the next fourth-quadrant; if end of list finish.
2. If the previous three leaves and the fourth-quadrant

have the same word structure apart from the lowest

significant bits of their laced-co-ordinates replace

them by the single leaf that covers them.
3. Return to 1 unless end of list.

As it stands this does not take care of the cases where

there are not enough leaves for it to be possible to

examine three previous leaves in 2: the modification to

take care of this is simple.

3.3.4 Construction of the treecode. Consider two coloured

leaves and the problem of filling in the quadtree structure

of background leaves and recursion symbols between

them. Examination of a typical case, such as shown in

Fig. 4, shows that the problem divides into two parts:

first a number of incomplete recursive steps are com-

pleted; secondly a (different) number of recursions are

started until the next leaf is reached. An example will
demonstrate this. The quadrant shown in Fig. 4 is the

smallest quadrant that encloses the two leaves (shaded);

the leaves have laced-co-ordinates 0110010000 and

1101001000 relative to the quadrant and sizes of 2 and 1.

The sequence of background leaves that have to be

inserted is

\[
\begin{array}{cccccc}
\text{laced-co-ordinate} & \text{size} & \text{current leaf} \\
01 & 01 & 01 & 01 & 2 & \text{..} \\
01 & 01 & 01 & 01 & 2 & \text{..} \\
01 & 01 & 01 & 10 & 2 & \text{..} \\
01 & 01 & 10 & 01 & 2 & \text{..} \\
01 & 10 & 01 & 10 & 4 & \text{..} \\
01 & 10 & 11 & 11 & 4 & \text{..} \\
01 & 11 & 11 & 11 & 8 & \text{..} \\
10 & 10 & 10 & 16 & \text{..} \\
\text{recurse} & 11 & 00 & 8 & \text{..} \\
\text{recurse} & 11 & 01 & 00 & 2 & \text{..} \\
\text{recurse} & 11 & 01 & 00 & 2 & \text{..} \\
\text{recurse} & 11 & 01 & 00 & 1 & \text{next leaf}
\end{array}
\]

The right-hand binary digits denoted by \( \text{..} \) are all zeros

and the left-hand left-hand \( \text{..} \) represent the binary digit
pairs that are common to all the co-ordinates.

The procedure for the first part is

\[
\text{let openout()} \text{ be}
\]

\[
\text{let bottomshift = twicelevelup(cursidelessone)}
\]

\[
\text{let bitpair = (curcoord \& bottomshift) & \# X3}
\]

\[
\text{while bitpair < \# X3}
\]

\[
\text{let bottomshift = bottomshift + 2}
\]

\[
\text{let number = (nextcoord \& topshift) - (curcoord \& topshift)}
\]

\[
\text{while number > 1}
\]

\[
\text{let bottomshift = bottomshift + 2}
\]

\[
\text{let bitpair = bitpair + 1}
\]

\[
\text{let openout = openout + 1}
\]

The second part is done with the procedure

\[
\text{let recursedown()} \text{ be}
\]

\[
\text{let bottomshift = twicelevelup(nextsidelessone)}
\]

\[
\text{let bitpair = (nextcoord \& bottomshift) \& \# X3}
\]

\[
\text{while bitpair < \# X3}
\]

\[
\text{let number = (nextcoord \& topshift) - (curcoord \& topshift)}
\]

\[
\text{while number > 1}
\]

\[
\text{let bitpair = bitpair + 1}
\]

\[
\text{let openout = openout + 1}
\]

The tree is constructed by filling in the background

and points of recursion between the leaves in sequence.

The tree is started with a call of recursedown and finishes

with openout with topshift equal to twice the maximum
depth for a tree.

Finally, the interior node values which temporarily

have all been set to 22 must be computed. This is done

by a recursive walk over the tree structure: the node

values are computed and inserted as they become

available.
4. OPERATIONS ON SQUARES

In all the algorithms that follow the operations are performed square by square. Accordingly we shall give just the algorithm for the operation on a single square. Recall that the squarecode contains the x and y co-ordinates of the origin of the square, see Fig. 1.

4.1 Translation

The origin of the square is translated and the co-ordinates of the diagonally opposite corner are computed. The translated square is tested to see if it needs to be clipped to the image space; if it does the resulting rectangle is broken up into leaves by the procedure makeleaves which is described in section 3.3.1.

4.2 Scale

The operation to scale the image about a specified point has the same form as the procedure translate in the previous subsection. xO is scaled as follows:

\[ xR = (xO - xcentre) \times \text{scalefactor(imagesize)} - xcentre \]

and similarly for yO, x, and y; these are the co-ordinates of the corners of the square. The co-ordinates of the central point for the operation are xcentre and ycentre. It is assumed that integer arithmetic is used. Accordingly we multiply the desired scale factor by a suitably large number, we choose imagesize, to get scalefactor and after multiplication of the distance to be scaled we divide by imagesize to get the result. The scaled square is clipped and broken up into leaves if necessary.

4.3 Restricted reflection and rotation

The restricted reflections are in lines through the centre of the image; the lines available are horizontal, vertical, inclined up at 45 degrees from the horizontal, and inclined down 45 degrees from the horizontal. Rotations of 90, 180, and 270 degrees are achieved by combinations of two reflections. For these reflections and rotations squares are always reflected into squares because we have chosen the image centre as a symmetry point.

Consider a reflection in the vertical line. This is depicted for a square in Fig. 5.

![Figure 5. Reflection in vertical line through image centre.](image)

A little consideration will show that the x-co-ordinate of the pixel at the origin of the square changes from xO to

\[ xR = \text{imagesize} - xO - \text{side} \]

Thus one assignment does the job. Rotations take two assignments of this sort.

4.4 Rotation through an arbitrary angle about the image centre

Each square in the squarecode is rotated about the image centre and the rotated square is filled with leaves by the procedure makeleaves. The origin of the square, (x, y), becomes (xR, yR): the procedure

\[
\begin{align*}
\text{let rotate}(x, y) &= \text{valof} \\
\text{result} &= \text{halfimagesize} + ((x - \text{halfimagesize}) \times \text{cosine\angle})/\text{imagesize} \\
&\quad - ((y - \text{halfimagesize}) \times \text{sine\angle})/\text{imagesize}
\end{align*}
\]

gives xR; a similar procedure gives yR.

The squarecode is traversed just once. The function testinout used in makeleaves is quite simple since the rotated square is a simple convex polygon.

5. ASSESSMENT

5.1 Space and time comparisons

To give a rough idea of how the computing times compare for treecode and squarecode we took picture H which is defined in Ref. 1 and processed it both as treecode and squarecode. Picture H has a quadtree with 6893 nodes and 3500 of these are leaves. The algorithms for the operations on treecode are described in Ref. 1. Some results are given in Table 1 with the times in seconds.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time for treecode</th>
<th>Time for squarecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>G := reverse(H)</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>K := rotate(H)</td>
<td>5.0</td>
<td>0.3</td>
</tr>
<tr>
<td>Z1 := translate(H, 256, 0)</td>
<td>1.1</td>
<td>0.6</td>
</tr>
<tr>
<td>Z2 := translate(H, 128, 0)</td>
<td>1.5</td>
<td>0.7</td>
</tr>
<tr>
<td>Z3 := translate(H, 64, 0)</td>
<td>1.5</td>
<td>0.7</td>
</tr>
<tr>
<td>Z4 := translate(H, 4, 0)</td>
<td>1.8</td>
<td>0.7</td>
</tr>
<tr>
<td>Z5 := translate(H, 2, 0)</td>
<td>2.0</td>
<td>0.7</td>
</tr>
<tr>
<td>Z6 := translate(H, 1, 0)</td>
<td>2.6</td>
<td>0.7</td>
</tr>
</tbody>
</table>

The rotation is a 90 degree rotation. Notice that the arguments given to translate are picture number, x-displacement, and y-displacement; with squarecode there is usually no significant variation in time attributable to the operation details, i.e. the values of the x and y displacements. An arbitrary translation displacement makes little difference to the squarecode times; for treecode the displacement has to be broken down into a sequence of translations each displacement a power of two.

The conversion from squarecode to treecode varies considerably because of the need to re-form leaves from squares. Some examples are given in Table 2; again times are in seconds.

<table>
<thead>
<tr>
<th>Picture</th>
<th>Squarecode to treecode</th>
<th>Number of leaves</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z1</td>
<td>3.0</td>
<td>1740</td>
</tr>
<tr>
<td>Z2</td>
<td>6.0</td>
<td>3384</td>
</tr>
<tr>
<td>Z3</td>
<td>6.3</td>
<td>3500</td>
</tr>
<tr>
<td>Z4</td>
<td>8.0</td>
<td>3488</td>
</tr>
<tr>
<td>Z5</td>
<td>11.3</td>
<td>3494</td>
</tr>
<tr>
<td>Z6</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
The conversion and compression for picture Z6 required more workspace than was allocated in the program but from other work we know that the final tree would contain 3425 leaves and the execution time would be around 14 seconds.

5.2 Remarks on the method of squarecode

Three geometrical operations have been described. They differ in their effect on the squarecode. Fundamentally translation and scale both transform squares into squares. Clipping those squares that overlap the image space boundary (putting a window on the new image) generates rectangles which are replaced by leaves; this generates a larger quantity of data to represent the image, though the net effect may be a reduction if other squares are translated out of the image space. In addition to this effect rotation generates more data due to squares always being transformed into objects that are not square, namely squares not aligned with the image space. This can lead to considerable amounts of new data if there are large squares to be rotated (recall that background colour is not explicitly in the squarecode).

This proliferation of leaves is a consequence of the choice of aligned squares as the fundamental elements in the data structure. This choice was dictated by the requirements:

1. The system should complement the quadtree based system described in our previous paper.
2. The definition of a square should fit onto one 32-bit word of memory.

If the second requirement is relaxed, several possibilities open up, for example:

(a) Choose a larger image space (for both the treecode and squarecode systems of course) and place a window on the image space for display; clipping at display time will not effect the image data structure.
(b) Allowing rectangles rather than squares as the basic data structure eliminates the generation of new data when clipping.
(c) For rotation the choice of squares of any orientation as the fundamental image data elements ensures that apart from clipping no new data has to be generated. However this does lead to other problems which are under investigation.

5.3 Future work

A unified system which uses the three dimensional analogues of both quadtrees and squarecode, namely octtrees and cubecode, is being investigated.

A generalization of the squarecode and cubecode representations to a representation based on rectangles and rectangular parallelepipeds is being investigated for use on machines with SIMD architecture. This representation, 'boxcode', is well suited to parallel manipulation and work is under way in the development of parallel algorithms for a system to run on the ICL Distributed Array Processor.

Acknowledgement

We are indebted to Mr. R. D. Eager for his assistance in transporting our partially completed program from Cambridge to Canterbury.

REFERENCES

6. P. J. Willis and D. J. Milford, A quad encoded display, School of Mathematics, University of Bath (1982), to be published.

Received March 1983