Operations on Quadtree Leaves and Related Image Areas

M. A. Oliver
The Computing Laboratory, The University, Canterbury, Kent, CT2 7NF, UK
N. E. Wiseman
Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge, CB2 3QG, UK

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.

1. INTRODUCTION

A quadtree encoding represents an image by recursively dividing the image into four quadrants until no further subdivision is needed because either there is no more image detail or the required resolution has been reached. The quadrants are ordered bottom left, top left, bottom right, top right; the reason for this choice will become manifest when squarecode has been described. In a previous paper we showed that a wide range of operations on computer images can be done efficiently, both in space and time, by the direct manipulation of the image quadtree stored depth first as a linear array of bytes. For some operations such as translate, scale, and 90° rotation, this approach did not yield simple and efficient algorithms. For some background and examples of a rather different treatment of quadtrees see Refs 2-5; hardware for quadtree display is described by Willis and Milford.

In this paper we describe a technique for image manipulation that complements the method based on the direct manipulation of the treecode that we described previously. With this technique translation, scale, and rotate this approach did not yield simple and efficient algorithms. For some background and examples of a rather different treatment of quadtrees see Refs 2-5; hardware for quadtree display is described by Willis and Milford.

In this paper we describe a technique for image manipulation that complements the method based on the direct manipulation of the treecode that we described previously. With this technique translation, scale, and rotate this approach did not yield simple and efficient algorithms. For some background and examples of a rather different treatment of quadtrees see Refs 2-5; hardware for quadtree display is described by Willis and Milford.

A quadtree leaf must have the length of its side some power of two. Given its size the leaf can only have a restricted number of positions in the image space; these are the positions of the squares when the whole image space is tiled with squares of this size. Clearly the image can be represented by a specification of the colour, size, and position of its quadtree leaves: we call such a specification leafcode. The space requirement for leafcode can compare favourably with that for the corresponding treecode. Some examples of space requirements are given in the assessment.

A more general way to encode the image is to break it up into arbitrary square areas each one of uniform colour; obviously this is not a unique prescription. The set of specifications of the colour, size, and position of image squares is called squarecode. The relation between squarecode and leafcode is described in the next section and reasons for using squarecode rather than leafcode are discussed in section 5.2.

The structure of the system is simple. Data, stored as treecode, is converted to leafcode as it is read in and is then ready to be operated upon. When the image has been manipulated it can be displayed or converted back to treecode for further manipulations that are best done directly on the treecode. In this way a flexible system, with a wide range of efficiently performed image operations, results from the amalgamation of the system described here with that described in our previous paper. Some data on the efficiency of the system are given in section 5.1.

The program fragments are written in BCPL. Variables that appear without a visible declaration may be taken to be static variables. Much of the program deals with the manipulation of the x and y co-ordinates, the colour code, and the length of the side of the square, which are all fitted onto a 32-bit word of memory. Necessarily these manipulations involve a good deal of bit manipulation with the aid of the logical bit operators & (and), | (or), »n (shift right n bits), «n (shift left n bits). Readers who want to see the detailed program should write for a copy.

2. TREECODE, LEAFCODE, AND SQUARECODE

2.1 Treecode

Treecode specifies a quadtree in depth first order. Each node has a value given in four bits; one additional bit indicates whether it is an interior node or a leaf. These five bit quantities are stored in byte fields in memory. A sequence of bytes is read as follows:

1. Take a square area.
2. Get the next byte.
3. If the byte indicates a leaf, colour the square with the value in the four bit field.
4. If an interior node, subdivide the square into its four equal quadrants and return to step 2 four times for the bottom-left, top-left, bottom-right, and top-right quadrants.

© Wiley Heyden Ltd, 1983

CCC-0010-4620/83/0026-0375 $03.00

© Wiley Heyden Ltd, 1983

THE COMPUTER JOURNAL, VOL. 26, NO. 4, 1983

375
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:

```plaintext
let treetoleaf(x, y, scale) be

// let nodenvalue = ?
for k = 0 to 3

// test nodenvalue < 16 then
unless nodenvalue = 0

// leafptr = leafptr + 1

curpic!leafptr = (nodenvalue < 28) | ((scale - 1) < 20) | (x < 12) | y

// or treetoleaf(x, y, scale/2)

switchon k into

case 0: y := y + scale

endcase

case 1: x := x + scale

y := y - scale

endcase

case 2: y := y + scale


The switchon 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. Woodwork 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 01100011 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.
3.3 Squarecode into treecode

Convert the squarecode, square by square, into leafcode in a buffer; as this is done replace the \((x,y)\) coordinates of each leaf by the equivalent laced-co-ordinate.

3. 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 \((x,y)\) co-ordinates 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 \texttt{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 \(x, y, \text{sidelessone}\) be the geometric parameters of the square in binary bit patterns then if

\[(x\ y) & \text{sidelessone} = 0\]

the registration is all right. For the square to be a leaf \(\text{sidelessone} + 1\) must be a power of two.

The procedure \texttt{makeleaves} has the following structure:

\[
\text{let makeleaves}(x, y, \text{side}) \text{ be} \\
\text{let testresult = testinout}(x, y, \text{side}) \\
\text{unless testresult} = \text{outside} \text{ do} \\
\text{testresult} = \text{inside then} \\
\text{let buffer}! \text{ outpointer = colourbits} ((\text{side} - 1) < 20) \\
\text{lacedcoorfrom}(x, y) \\
\text{outpointer} \text{ = outpointer + 1} \\
\}
\]

or

unless \text{side} = 0

\[
\text{let side} = \text{side}/2 \\
\text{makeleaves}(x, y, \text{side}) \\
\text{makeleaves}(x, y + \text{side}, \text{side}) \\
\text{makeleaves}(x + \text{side}, y, \text{side}) \\
\text{makeleaves}(x + \text{side}, y + \text{side}, \text{side}) \\
\}
\]

The function \texttt{testinout} returns \text{inside}, \text{outside}, or \text{neither} according to how the leaf specified by the leaf origin \(x, y\) and \text{side} is related to the square in hand at the time of the call. A little care has to be taken in order to ensure that the pixels on the common boundary of adjacent squares are neither counted twice nor omitted altogether. \text{Lacedcoorfrom} simply returns the laced-co-ordinate of the point \(x, y\).

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 so doing.
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 \((\text{maxdepth} - n - 1)\): it divides into four quadrants at \((\text{maxdepth} - n)\) and their laced-coordinates 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 compressing 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-coordinates 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 completed; 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-coordinates 01100100 and 11010010 relative to the quadrant and sizes of 2 and 1. The sequence of background leaves that have to be inserted is

\[
\begin{align*}
\text{laced-co-ordinate} & \quad \text{size} \\
\text{current leaf} & \quad 2 \\
\text{recurse} & \quad 8 \\
\text{next leaf} & \quad 1
\end{align*}
\]

The right-hand binary digits denoted by \(\cdots\) are all zeros and the left-hand left-hand \(\cdots\) represent the binary digit pairs that are common to all the co-ordinates.

The procedure for the first part is

\[
\begin{align*}
\text{let } \text{openout}( ) & \quad \text{be} \\
\text{let } \text{bottomshift} & \quad = \text{twicelevelup(} \text{cursidelessone} ) \\
\text{until } \text{bottomshift} & \quad = \text{topshift} \\
\text{let } \text{bitpair} & \quad = \text{curcoord} \gg \text{bottomshift} & \# \text{X3} \\
\text{while } \text{bitpair} & \quad < \# \text{X3} \\
\text{let } \text{outpointer} & \quad = \text{outpointer} + 1 \\
\text{buffer} & \quad ^{2}/_{4} \text{outpointer} = 0 \\
\text{bitpair} & \quad = \text{bitpair} + 1 \\
\text{bottomshift} & \quad = \text{bottomshift} + 2 \\
\text{number} & \quad = \text{(nextcoord} \gg \text{topshift}) - \text{(curcoord} \gg \text{topshift}) \\
\text{while } \text{number} & \quad > 1 \\
\text{let } \text{outpointer} & \quad = \text{outpointer} + 1 \\
\text{buffer} & \quad ^{2}/_{4} \text{outpointer} = 0 \\
\text{number} & \quad = \text{number} - 1 \\
\end{align*}
\]

The function \text{twicelevelup} returns twice the value of the difference between the maximum available depth and the depth of the leaf.

The second part is done with the procedure

\[
\begin{align*}
\text{let } \text{recurse(} ) & \quad \text{be} \\
\text{let } \text{bottomshift} & \quad = \text{twicelevelup(} \text{nextsidelessone} ) \\
\text{until } \text{bottomshift} & \quad = \text{topshift} \\
\text{let } \text{outpointer} & \quad = \text{outpointer} \\
\text{buffer} & \quad ^{2}/_{4} \text{outpointer} = 22 \\
\text{topshift} & \quad = \text{topshift} - 2 \\
\text{bitpair} & \quad = \text{(nextcoord} \gg \text{topshift}) & \# \text{X3} \\
\text{k} & \quad = 0 \\
\text{while } \text{k} & \quad < \text{bitpair} \\
\text{let } \text{outpointer} & \quad = \text{outpointer} + 1 \\
\text{buffer} & \quad ^{2}/_{4} \text{outpointer} = 0 \\
\text{k} & \quad = \text{k} + 1 \\
\end{align*}
\]

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 \text{recurse(} \text{down}) and finishes with \text{openout} with \text{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:

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

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 to 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.

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

\[
\text{let rotate}(X(x, y)) = \text{valof resultis halfimagesize + ((x-halfimagesize) \times \text{cosine}\angle - (y-halfimagesize) \times \text{sine}\angle)/\text{imagesize}
\]

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 COMPUTER JOURNAL, VOL. 26, NO. 4, 1983 379
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 affect 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 cube code, is being investigated.

A generalization of the squarecode and cube code 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.