The Explicit Quad Tree as a Structure for Computer Graphics

J. R. Woodwark
School of Engineering, University of Bath, Claverton Down, Bath BA2 7AY, UK

A quad tree, stored without links and with a location for every possible node, is proposed as a structure for holding an image under construction. In this form, picture coherence is not exploited to reduce storage requirements, but to improve the speed of interrogation and modification. Basic operations on this structure are outlined, and an efficient addressing scheme presented.

INTRODUCTION

The technique developed by Warnock[1] for hidden-surface elimination has proved to be one of the most fruitful approaches to the problem. He implemented a divide-and-conquer approach, not on the structure of the scene to be processed, but on the picture area. This is divided into successively smaller rectangles until either an area contains a piece of picture sufficiently simple to be output directly, or the resolution of the graphics device, for which the picture is being prepared, is reached. In this case an approximation may be made from the colours of the still unresolved picture components.

This approach to processing pictures has a counterpart as a storage scheme. It is called the quad tree (Fig. 1). A square picture is divided into four sub-squares. If possible these are characterized directly as wholly black or white for a binary image, or as a single shade for a coloured one. Otherwise these sub-squares are further decomposed, until a smaller sub-square, or quad, can be classified. As with the Warnock algorithm, a limiting resolution must exist to prevent unchecked growth of the structure. This may be imposed by storage considerations or by the desire to limit times for processing. Alternatively, it may be a function of the application, such as the resolution of a device.

Two main reasons exist for the application of quad trees. First, because they exploit the coherence of many pictures, whether acquired or synthesized, they are an efficient method for picture storage. Klinger and Dyer[2] present examples of the extent of data compaction to be expected on some small scenes. The author and his colleagues[3] are developing the use of the quad tree in graphics hardware design, where the quad tree also constitutes a compact data transmission format. The second property of quad trees is that they express, as well as exploit, picture structure. In picture processing, rather than generation, this allows algorithms to obtain 'impressions' of the picture at varying resolutions, and to focus attention quickly on areas of interest.

Because of the wish in both types of application to exploit the coherence of pictures to reduce storage requirements, quad trees are usually stored in the form of a linked tree structure. This has links from each father quad to its four sons, and possibly additional links from son to father for backtracking, although this may often be more economically implemented using a stack. Some picture processing applications of the quad tree have also utilized links to horizontally and vertically

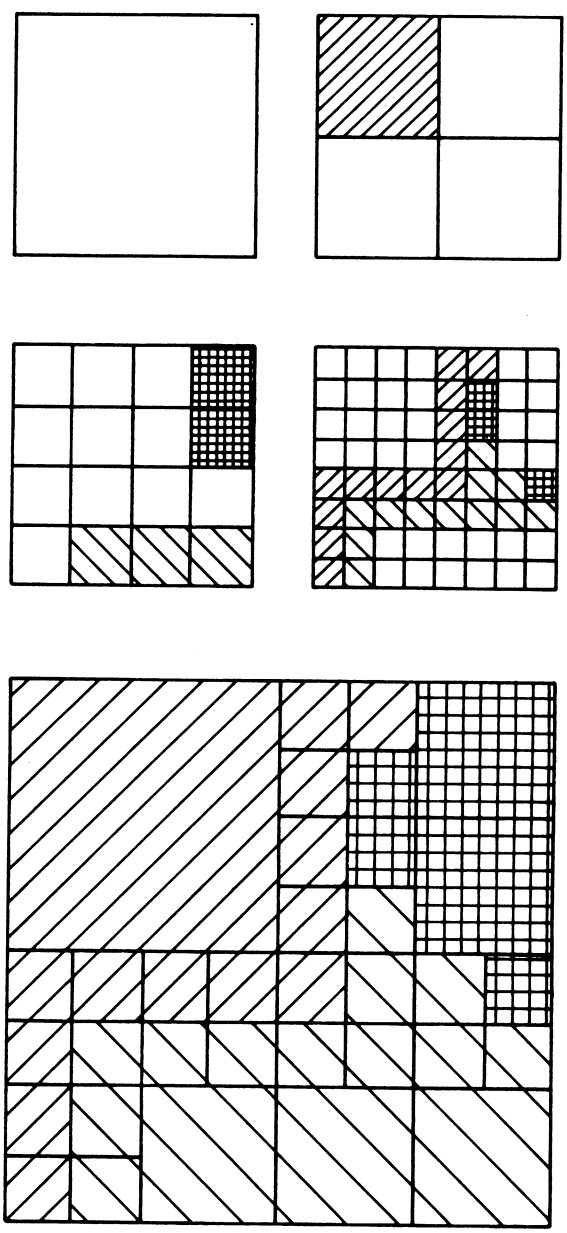

Figure 1. A quad tree representation of an 8 × 8 pixel three colour 'picture'.

CCC-0010-4620/82/0025-0235 $02.00

adjacent quads. Hunter and Steiglitz[4] introduce 'ropes', or sideways links from leaf nodes, into their quad tree structure for image processing. However, links of this sort nullify some of the simplicity inherent in the quad tree concept. The quad tree representation may also be converted to other commonly used digital picture forms, such as the raster, and Samet and others[5–10] present a number of algorithms for such transformations.

In picture generation, the quad tree, as a structure rather than a strategy, is not widely used. The author of this paper has been concerned with the derivation of graphics from volume models.[11,12] In this work it has been necessary to maintain and to update a synthesized image as model evaluation progresses. The quad tree structure developed for this process is presented here in the belief that it will find application in other similar areas.

As stated above, most quad tree applications have either focused on, or at least utilized, the data compression obtainable from the quad tree, and this has implied a linked tree structure. An explicit structure, in which a storage location is assigned to every location in a quad tree down to a defined pixel level of leaf quads, does not at first seem to be useful. The number of storage locations required is approximately 1.3 times that needed for a pixel plane representation of the picture, which in any case is exactly what the bottom layer of the tree comprises. However, because the quad tree exploits area coherence, it is possible to write to and to read from the quad tree much faster than to the corresponding pixel plane. Comparing the explicit quad tree with the linked quad tree, the absence of links in the former makes it more storage efficient than might be expected. This is particularly the case when the number of bits required to store the data at each quad is small. As the number of bits assigned to the links depends on the maximum number of quads allowed in the tree, linked quad trees for data with small values consist mainly of links, not data. This sort of situation occurs when a picture is being prepared for a relatively low colour resolution graphics device.

A further problem occurs with the linked quad tree when the amount of storage available is insufficient to allow the picture to be handled in one piece. If data is being acquired from a low resolution device, such as a diode camera, this does not arise, but it may occur when a picture is being prepared for a high spatial resolution raster scan device. In this case, the picture must be constructed as a series of sub-pictures, and the coherence of the whole picture cannot be fully exploited. This is because, if a linked quad tree is used to hold each sub-picture in turn, the amount of storage allocated would have to be sufficient for the sub-picture with the lowest expected coherence. If a less coherent sub-picture is discovered, the program fails. An explicit quad tree takes advantage of sub-picture coherence when it occurs, but will accommodate a sub-picture which has no coherence at all.

Finally, an explicit quad tree allows an increase in the speed of data input into, and retrieval from, the quad tree. Links do not have to be processed, and it is possible to travel up or down the tree with equal facility. An efficient addressing technique to realize these advantages is presented as part of this paper.

AN EXPLICIT QUAD TREE STRUCTURE

The creation of a usable explicit quad tree does not initially seem to present any problems. It is necessary first to decide the number of bits of data for each quad. This determines the number of data values or colours a quad may have, less two values which must be given special meanings, and are therefore not available as colours. One of these reserved values is a code for neutral, indicating that no colour has yet been assigned to a quad during the construction of the image. The second is a code for transparent. This indicates that a quad is not a single colour, and that the quads below that one must be consulted.

When the data width has been decided, storage locations of the appropriate size can be reserved. The number required is determined by the resolution required from the tree, which in graphics applications corresponds to the pixel size. One location is taken by the root quad, four for its sons, sixteen for theirs and so on. The size of the pixel layer will be limited by the amount of storage that is available and, where the whole picture can not be stored at once, will determine the size of sub-picture that can be handled.

A simple scheme to address such a structure is to determine the offset of each pixel from the first storage location in its layer using the x and y indices of each quad, as one might address any two-dimensional array. Figure 2 shows an example of the resulting quad

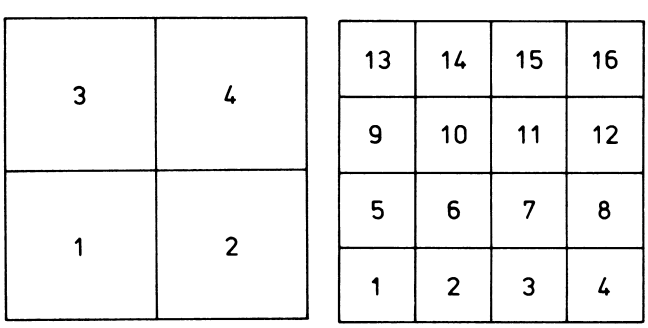

Figure 2. The second and third layers of a quad tree, showing standard rectangular array addressing.

numbering. However, such an arrangement involves a certain amount of computation to move up or down the tree, as this must be performed using the x and y indices. Additional operations are then required to generate the quads' addresses from the indices. An alternative scheme used by the author allows the tree to be traversed by single operations on the quads' addresses directly, using the numbering shown in Fig. 3.

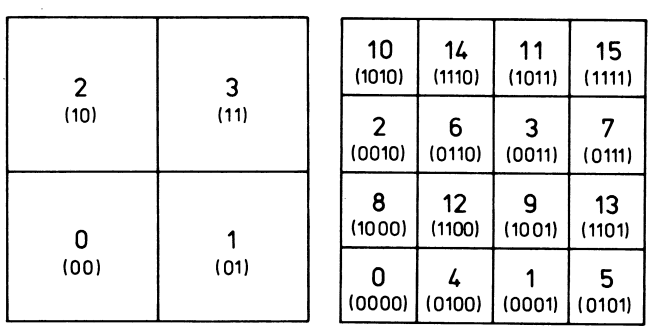

Figure 3. The second and third layers of a quad tree, showing the improved addressing scheme.

In this scheme, the lowest bits of any quad's address are determined by the address of its father. The next highest two bits are determined by its own position within its father. The low bit is set if it is at high x, the high bit if it is at high y. This continues down the tree, the two bit increase in address length at every level corresponding to the fourfold increase in the number of quads. To move up this structure, it is merely necessary to remove the highest two bits of the current address. To move down, the address is logically combined with the four possible permutations of the two bits higher than the current address width. To visit each son in turn, the addresses may be generated as follows:

son 1 := father
son 2 := father OR . . . 01 . . .
son 3 := father OR . . . 10 . . .
son 4 := father OR . . . 11 . . .

Alternatively, the following scheme:

son 2 := father OR . . . 01 . . .
son 4 := son 2 OR . . . 10 . . .
son 3 := son 4 XOR . . . 01 . . .
son 1 := son 3 XOR . . . 10 . . .

allows the same variable to be used for all the addresses. The father's address remains unchanged at the end of the operation, as it is the same as that of son 1, the last to be accessed.

A quad's brothers may be generated directly in a similar manner:

brother 2 := brother 1 XOR . . . 01 . . .
brother 3 := brother 1 XOR . . . 10 . . .
brother 4 := brother 1 XOR . . . 11 . . .

This structure may be entered directly from a quad's geometric position using a lookup table. This yields an address both for the x and for the y component of the position of the quad's bottom left hand pixel. These are ORed together, and the result is the address of the quad. One x and one y table serve for all levels in the tree, and the length of each corresponds to the width of the pixel layer.

OPERATIONS ON QUAD TREES FOR PICTURE GENERATION

Using this addressing scheme, the author has identified and implemented four primitive actions on a quad tree, as a basis for picture generation.

Ancestor check

Before entering the quad tree from the geometric position of a quad, it is necessary to check the fathers of that quad to discover whether the quad is actually below a leaf quad. This test is performed by generating the addresses of the ancestors, starting with the root, by stripping leading bits off the quad's address. If any quad is found not to be transparent, this test terminates.

Division to a quad

If a quad is to be set rather than read, then the discovery, in the test above, of an ancestor that is not transparent causes that ancestor to be set transparent, and its property to be transferred to its sons. This process is repeated on all the sons down to and including the quad to be set.

Tree traversal

If it is desired to search the whole quad tree, or the tree below a particular quad, then that tree must be traversed. A depth first traversal is performed by generating the sons of the quad from which the process is to start, and repeating the process on each of these in turn. This may be implemented in systems that do not support recursion by a stack or, because in most cases the possible depth of division is small, by coding each division level separately. The author has found this technique to be extremely fast running within an implementation in extended FORTRAN.

Reassembly

After a quad is set, it may be that it has the same data value as its three brother quads. In this case, the data value is transferred to the father quad to maintain the quad tree's structure at a minimum level of division. This process is repeated on the father quad and so on until a quad with different brothers, or the root, is encountered. It has already been stated how a quad's brothers' and father's addresses may be generated.

With these four primitives available, more complex operations on pictures or part-pictures may be performed; five of these, sufficient to allow pictures to be prepared and output, are mentioned below.

Zeroing. To zero an explicit quad tree, it is only necessary to insert the code for neutral into the root quad.

Superimpose. One method of hidden surface elimination[13] is to generate all parts of the scene, in reverse order of distance from the viewer. These are loaded into a picture buffer, and the final picture consists of the surfaces nearest the viewer, which are those he would see if looking at a real scene. To superimpose a quad on an existing picture, an ancestor check is performed down to the target quad. If an ancestor is other-coloured or neutral, the division process is performed down to the quad to be set. After setting the quad, reassembly is used to reform the quad tree if necessary.

Insertion behind current picture. This process is useful if the scene is being considered in order of distance from viewer, and updates to the pictures must be behind the current contents of the quad tree. This is more complex than superimposition, but may allow computation on portions of the scene that are actually hidden to be avoided. The ancestors of the quad to be set are checked. If any are coloured, the process terminates. If a neutral ancestor is found, division down to the quad to be set takes place. In this case the quad can be set directly, and the process finishes. The process also finishes if the quad is coloured. If it is neutral, it may be coloured and any reassembly performed. If the target quad is transparent, the quads below must be searched. If any neutral quads are found, each is set to the required colour, and reassembly is invoked.

Checking the status of a quad. Before computing part of a scene, it may be appropriate to examine the quad tree to see if the corresponding portion of the picture is completely coloured, in which case the computation may be omitted. To check a quad, its ancestors are checked first. If either a neutral or a coloured ancestor is found, the state of the quad is completely determined. This is also true if the quad itself is coloured or neutral. If the target quad is transparent, then it is necessary to search the quad tree below that quad. As soon as a single neutral quad is found the process can finish as the area of the quad to be examined is clearly not all opaque. If no neutral quad is found, however, then the entire quad under examination is coloured.

Output After the quad tree has been fully computed, it must usually be output to a device as part of a picture. This process is basically a tree traversal starting from the root. However, a problem with the addressing scheme outlined here is that there is no quick transformation back to the quad origin coordinates from the quad address. Instead, the quad coordinates can be obtained by starting with the coordinates of the root quad, and updating them with each descent of the tree, so that the coordinates of the current quad are always known. This operation can also be done with logical operators alone. Finding the sons of a quad with their origin coordinates takes the following form:

son 2 := father OR . . . 01 . . .
x := x OR size

son 4 := son 2 OR . . . 10 . . .
y := y OR size

son 3 := son 4 XOR . . . 01 . . .
x := x XOR size

son 1 := son 3 XOR . . . 10 . . .
y := y XOR size

where 'size' is the width in pixels of the son quads, which can be obtained by lookup, and consists of a zero field except for a single bit which is set. Both x and y are returned to their original values after the operation, as the origin of the first son is the same as that of the father.

CONCLUSIONS

The explicit, or full, quad tree has been presented as a little used data structure which may nevertheless have important applications, particularly with raster scan graphics devices. The obvious storage penalty of the structure is less than might be imagined, and access to its contents may be extremely efficient.

Acknowledgements

The author is grateful to the Science and Engineering Research Council for supporting this work.

REFERENCES

1. J. E. Warnock, *A Hidden-surface Algorithm for Computer Generated Pictures*, University of Utah Computer Science Department Report TR4–15 (1969).
2. A. Klinger and C. R. Dyer, Experiments on Picture Representation using Regular Decomposition, *Computer Graphics and Image Processing* **5**, 68–105 (1976).
3. D. J. Milford, P. J. Willis and J. R. Woodwark, Exploiting Coherence in Raster Scan Displays, *Proceedings of the Electronic Displays 81 Conference*, London, Network, pp. 34–46 (1981).
4. G. M. Hunter and K. Steiglitz, Operations on Images using Quad Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI–1** (No. 2), 145–153 (1979).
5. C. R. Dyer, A. Rosenfeld and H. Samet, *Region Representation: Boundary Codes from Quad trees*, University of Maryland Computer Science Center Report TR–732 (1979).
6. H. Samet, *Region Representation: Quadtrees from Boundary Codes*, University of Maryland Computer Science Center Report TR–741 (1979).
7. H. Samet, *Computing Perimeters of Images Represented by Quadtrees*, University of Maryland Computer Center Report TR–755 (1979).
8. H. Samet, *Region Representation: Quadtrees from Binary Arrays*, University of Maryland Computer Science Center Report TR–767 (1979).
9. H. Samet, *Region Representation: Raster-to-Quadtree Conversion*, University of Maryland Computer Science Center Report TR–766 (1979).
10. H. Samet, *Region Representation: Quadtree-to-Raster Conversion*, University of Maryland Computer Science Center Report TR–768 (1979).
11. J. R. Woodwark and K. M. Quinlan, The Derivation of Graphics from Volume Models by Recursive Subdivision of the Object Space, *Proceedings of the Computer Graphics 80 Conference*, Brighton, Online Publications, pp 335–343 (1980).
12. J. R. Woodwark and K. M. Quinlan, Reducing the Effect of Complexity on Volume Model Evaluation, *Computer Aided Design Journal* **4** (No. 2) (1982).
13. I. E. Sutherland, R. F. Sproull and R. A. Schumacker, A Characterisation of Ten Hidden-Surface Algorithms, *ACM Computing Surveys* **6** (No. 1) 1–55 (1974).

Received May 1981