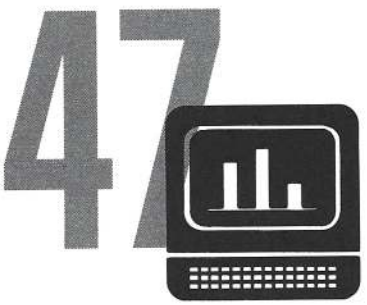

STORING IMAGES

A Cat in a Quad Tree

The arrival of quad trees in the early 1970s signaled a new stage of progress in computer graphics. Many operations, whether for the storage, manipulation, or analysis of computer images, were enhanced. Entirely new operations were made possible—all by the simple expedient of subdividing a digital image by quadrants and identifying the quadrants as nodes in a tree, the quad tree.

Consider Figure 47.1. A digital cat sits in profile. The cat may be decomposed (so to speak) into a quad tree. Armed with this structure, we may store the cat image very efficiently, carry out various geometric transformations on it, and determine whether there are any objects besides the cat.

The quad tree for our cat is generated by dividing the image matrix into quadrants, then subdividing each of these into quadrants, and so on. The last stage of subdivision is reached when each pixel, i.e., picture element, becomes a quadrant in its own right. For such subdivision to work, the size of the matrix must obviously be a power of 2. But this is usually easy to arrange.

Each quadrant in the subdivision is represented by a node in the corresponding quad tree; if the quadrant is all black or all white, the node is tagged

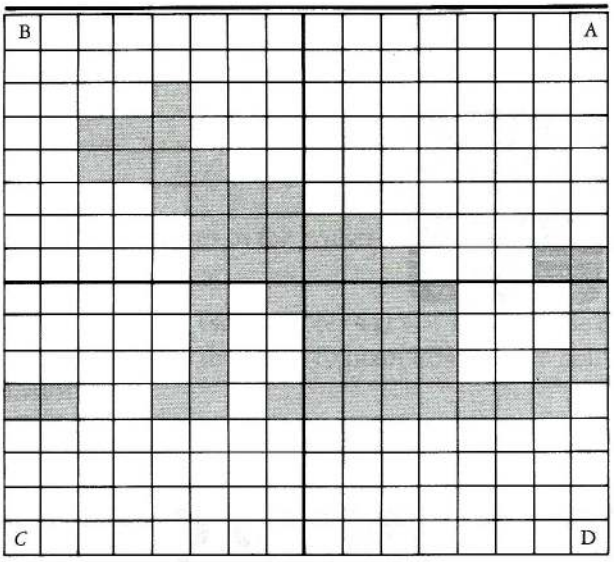

Figure 47.1 A cat and something else

accordingly and treated as a terminal node. It has no children. But if the quadrant is neither all black nor all white, the corresponding node is given four child nodes, one for each subquadrant. These are taken in the same order as the original four quadrants labeled *A*, *B*, *C*, and *D* in the illustration.

When it is decomposed by this process, the cat image is represented by a single node representing the entire matrix. The four principal quadrants are represented by four nodes appropriately labeled. The first node is the northeast quadrant, the second node is northwest, the third node is southwest, and the last node is southeast (Figure 47.2). Each of the four nodes gives rise to a subtree that is the quad tree for that quadrant; node *B*, for example, yields the subtree shown in Figure 47.3.

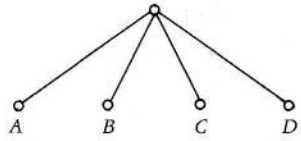

Figure 47.2 The principal nodes of a quadtree

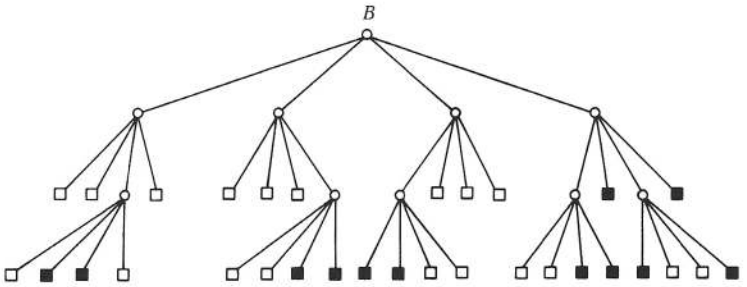

Figure 47.3 The quadtree in quadrant *B*

The quad tree above *is* the image of the cat when it is interpreted properly. A white terminal node that is k levels up from the bottom of the tree represents a $2^k \times 2^k$ quadrant that is entirely white. And so with black. The total number of nodes in the quad tree tracks the storage space it requires very closely. Thus the cat has been compressed into 41 nodes. At the same time, the cat's image in quadrant B occupies a 16×16 matrix. By conventional storage methods, 64 pixels must be stowed away. For the vast majority of images that arise in various applications, this kind of storage economy is typical.

A number of manipulations of quad trees correspond to standard manipulations of graphic images. For example, to rotate the image of the cat 90° counterclockwise, the nodes at each level must be rotated, in effect. Thus the root of the tree would display a different order (Figure 47.4). Quadrant *D* now occupies the northeast corner of the picture, *A* the northwest, and so on. The subtree pendant at *B* now takes on a different appearance as each of its nodes are rotated (Figure 47.5).

Among the other manipulations that can also be carried out on images represented by quad trees are changes of scales (by a factor of 2) and translations. To change the scale of an image, it is only necessary to remove all the terminal nodes at the bottom level of the tree and to reinterpret the root node as representing one principal quadrant. This shrinks the image by a factor of 2. To blow

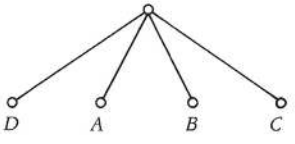

Figure 47.4 Rotating the principal nodes

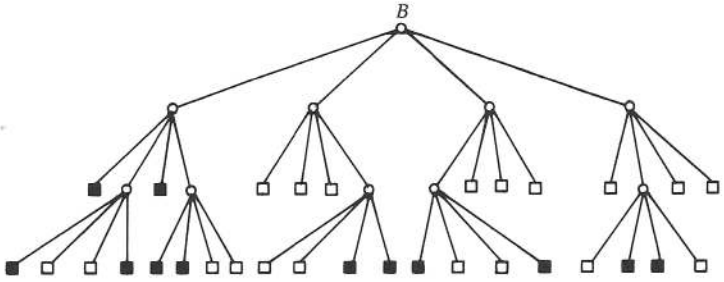

Figure 47.5 Rotating the quadtree at *B*

it up, the image must initially occupy just one principal quadrant; remove the root node and the three nodes corresponding to the quadrants not occupied by the image.

The shrinking operation is closely related to a useful approximation technique. If one or more levels are removed from the bottom of the quad tree, the resulting image loses a factor of 2 or more in resolution. It becomes approximate. For some applications such as visual reference and pattern recognition, it is sufficient to work with low-resolution images, especially ones that are so easily produced.

The final operation reviewed here identifies the components of an image. In the picture presented at the beginning of this chapter, there are two components, the cat and two adjacent pixels in front of it. It is difficult to display a convincing mouse at this resolution, but that is what the pair of pixels represents. The image, in other words, has two components. Finding the components of an image is important in numerous applications, such as the automatic scanning and counting of separate features in medical images.

The component-finding algorithm locates a black pixel and searches for black neighbors to the north, east, south, and west. It searches continually outward, assembling a list of all black pixels in the same component while continually adjoining new neighbors to the list. To identify a component in this manner, a neighbor-finding algorithm is therefore crucial. Such an algorithm operates on the quad tree, of course, so it is reasonable to speak of nodes in the tree and certain quadrants of the image matrix interchangeably. The neighbor of a given quadrant in a given direction is the smallest quadrant of at least that size sharing its border on the given side. This description is readily converted to an algorithm for traversing the quad tree. Suppose, for example, that we are searching for the western neighbor of a given node. In the algorithm that follows, the *A*, *B*, *C*, *D* quadrant-labeling convention introduced earlier is used. Thus, a *B* quadrant is a western neighbor of an *A* quadrant, and a *C* quadrant is a

western neighbor of a *D* quadrant. As the algorithm ascends the tree, it pushes the labels it encounters onto a stack. As soon as it reaches a node via a link marked *B* or *C*, the algorithm descends, always taking a link that is horizontally opposite to the one ascended on the same level of the tree.

1. **repeat**
   1. ascend one link
   2. push its label onto stack
2. **until** link label is *B* or *C*
3. **repeat**
   1. pop label from stack
   2. descend link with opposite label
4. **until** node is terminal or stack is empty

This algorithm can be generalized to take any direction as input and to find the neighbor of a given quadrant in that direction.

To use the quad tree to identify components, the tree must be traversed in some order (see Chapter 11). As each new black terminal node is visited, it is given a new label (if it is still unlabeled), and its four neighbors are examined by using the neighbor-finding algorithm above. A black neighbor (which must be a terminal node by the definition of the neighbor) receives the same label as the node being traversed:

1. *label* ← 1
2. **for each** terminal node in traversal
   1. **if** node black and unlabeled
      1. **then** label node
         1.2. **for each** neighbor
            if neighbor black and unlabeled
            then label neighbor
            else add pair of labels to list
         1.3. *label* ← *label* + 1

This is not the whole component-finding algorithm, but it is the major part. The list of equivalent pairs it produces is actually a graph which the next part of the component-finding algorithm must investigate. The graph can be explored in depth-first fashion. As long as new pairs can be found for which one label is in an equivalence class and the other is not, the algorithm continues. Finally, there

will be no pairs left to process, and each equivalence class will now receive a new label. At this point, the algorithm could output the number of such new labels (two in the cat-and-mouse example).

The third stage of the component-finding algorithm simply traverses the quad tree one more time, assigning the new label that is equivalent to each of the old labels encountered during the traversal.

Problems

1. Write an algorithm that converts a binary image matrix to the corresponding quad tree.

2. Although most images requires less storage in quad tree form than in matrix form, there is one definite exception. Find the worst case in this respect.

3. Construct a new quad tree for the cat after it has pounced on the mouse, i.e., moved two pixels west. Design a general translation algorithm for horizontal and vertical motion.

References

Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, Mass., 1989.

Hanan Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, Mass., 1989.