ALGORITHMS FOR CONSTRUCTING QUADTREE SURFACE MAPS[1]

Ron Sivan
Hanan Samet
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland at College Park
College Park, Maryland 20742
E-mail: sivan@umiacs.umd.edu, hjs@umiacs.umd.edu

Abstract

Two algorithms for constructing surface quadtrees from grid data, using the design proposed in [Von Herzen, 1989] are presented. Both run in $O(n)$ time, where n is the size of the input. They differ in where they spend most of their processing effort. Depending on the size of the input and the storage medium it is on, either algorithm could outperform the other.

1 Introduction

The advent of low-price, high-speed computers and high quality graphic monitors has made the need for efficient methods for storing and manipulating image data for use in solid modeling more immediate. One common technique of modeling multidimensional data is to use a polyhedral approximation: the smooth, complex reality is approximated by a set of planar polygons connected in three-dimensional (3d) space. Based on the x and y coordinates of a point within the a polygon (relative to its vertices) and the 3d coordinates of the vertices themselves, the approximate elevation (i.e., z value) of the point can be calculated, and need not be stored explicitly. The space and time requirements are thereby reduced. By adjusting the density (i.e., number and size) of the polygons, many applications can be adequately served.

To further improve performance, hierarchical multi-resolution structures may be employed. Such structures can assist in adapting the density of polygonal facets to the variability in the object being modeled, and serve as an index for quick polygon location.

[1]The support of the National Science Foundation under Grant IRI-9017393 is gratefully acknowledged.

This paper deals with representing surface data, also known as $2\frac{1}{2}$-dimensional ($2\frac{1}{2}$d) data, using quadtrees. Unlike multidimensional *discrete* data, for which quadtrees are readily adaptable, using a quadtree to represent a surface suffers from the formation of cracks, as discussed in Section 2. There are several remedies suggested in the literature for this problem [Barrera and Hinojosa, 1987], [Von Herzen and Barr, 1987]. The algorithms that we present use one of the solutions given in [Von Herzen, 1989], termed a *restricted quadtree*, which is described in Section 3. Two algorithms for the construction of a restricted quadtree for a surface given by a regular sample of its elevations (also known as a Digital Terrain Map) are given in Section 4. Section 5 contains concluding remarks.

2 The Problem of "Cracks"

In modeling $2\frac{1}{2}$d surfaces, use of a traditional quadtree (i.e., a version of the region quadtree, [Samet, 1990a, Samet, 1990b]) are prone to the formation of problematic discontinuities. For example, consider Figure 1. On the boundary between two nodes of different size, at least one vertex, say A, of the smaller node is *not* a vertex of its larger neighbor. Since elevation data is stored only in vertices, the elevation of A in the larger node must be interpolated from the elevations of the other vertices in that node. On the other hand, A is a vertex in the smaller node, where elevation *is* stored. These two elevations, associated with the same location, need not coincide, thereby causing a "crack" to form, as shown in the figure.

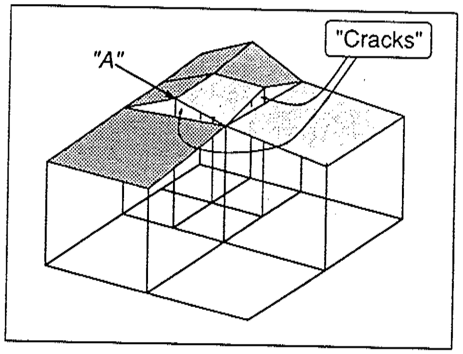

Figure 1: Example of crack formation in surface quadtree representation.

3 Restricted Quadtree Scheme for Surface Representation

A possible remedy, presented in [Von Herzen, 1989], attacks the problem from two aspects. First, the quadtree is restricted so that the occurrence of such "dangling" vertices is controlled. Second, the node itself is enhanced to cope with the eventuality that they do occur. In this solution, adjacent nodes are allowed to differ in size by at most a factor of two. In such a quadtree, called a *restricted quadtree*, there can be no more than one discontinuity per edge.

In addition, the traditionally square quadtree node is triangulated so that discontinuities are avoided. Every square is split into four right triangles by means of its two diagonals, as in Figure 2. Any of the resulting triangular faces which border smaller nodes are split again along their right angle bisector. This last step makes the potential discontinuity point a vertex of the larger node as well. Thus we can use a quadtree-like representation at the price of increasing the number of the nodes required to model a given surface.

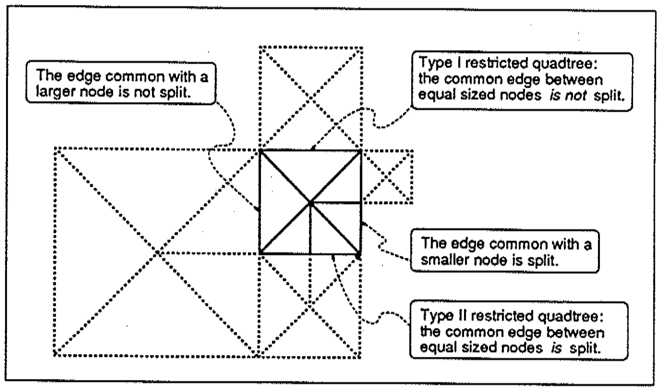
The edge common with a larger node is not split.
Type I restricted quadtree: the common edge between equal sized nodes *is not* split.
The edge common with a smaller node is split.
Type II restricted quadtree: the common edge between equal sized nodes *is* split.

Figure 2: restricted quadtree node structure.

Equal-sized nodes may have their common border either whole or split. The choice between the two makes for two variants of the restricted quadtree (see the top and bottom boundaries of the node in Figure 2). The former reduces somewhat the number of triangles in the tree, but the latter has been found empirically to provide for better shading in display. All examples in this paper assume this second variant of the restricted quadtree [Samet, 1990b].

4 Restricted Quadtree Construction Algorithms

We present two algorithms for constructing a restricted quadtree. They follow the classic bottom-up and top-down paradigms, and are therefore identified using these names. The description of these algorithms uses several terms which are defined in Section 4.1. The smallest, indivisible nodes in a restricted quadtree, called *atomic* nodes, deserve special consideration and are discussed in Section 4.2. The actual construction algorithms are described in Sections 4.3 and 4.4.

4.1 Assumptions and Terminology

The restricted quadtree describes an area whose shape is a square and whose side is a power of 2. For datasets that are not square areas, the smallest appropriate square large enough to contain the dataset is assumed. A value of zero is used for elevations of locations within the square which are nonetheless outside the dataset.

In order to form a node in a restricted quadtree, we need the elevations of the surface at its nine vertices: the center, the four corners and the midpoints of the four edges of the square. A node is *constructed* by obtaining these elevation values from the input and generating a structure appropriate for insertion into the restricted quadtree being built. For most nodes, these values are *not* consecutive in the input. However, since the input data is essentially a two-dimensional array, the offset to any datum can be calculated and its value retrieved in $O(1)$ time even if the input resides in secondary storage. This can also be done even if the actual input is embedded within a larger square, as described above.

The elevation at a node's vertex is a *stored* value, corresponding to that existing in the input. At any other point within the node, the elevation is *computed* by interpolating stored values. Every non-root node in a restricted quadtree shares some vertices with its immediate parent. We call such vertices *persistent* vertices. The vertices at the four corners of a node are persistent; the vertex at the center and those at the midpoints of its edges are not. For example, the elevation at a *non-persistent* vertex is a *stored* value at the node but a *computed* one at its parent.

4.2 The Construction of Atomic Nodes

In terms of implementation, the relationship between the smallest nodes and the input needs clarification. Each node has at least 5 vertices (4 at its corners and one at its center) and possibly as many as 9 (adding the midpoints of its edges when it is necessary to decompose them). The elevations stored at each of these vertices, in the ideal case, are the actual input values. However, it is conceivable to store interpolated values, corresponding to virtual data points, at locations that are not in the original dataset (i.e., non-grid points).

The following are three possible atomic node construction schemes:

1. Build a node around each data value, making it the center vertex and assign interpolated values to the other vertices, as in Figure 3a. Each atomic node then represents a single elevation sample.

2. Each four input data values whose positions form a unit grid square are used as the basis for an atomic node. The elevation at the center vertex is interpolated from the other four. See Figure 3b. Each atomic node uses four elevation samples.

3. Atomic nodes may be constructed from a 3×3 subgrid (see Figure 3c). Here all vertices, even at this low level, represent actual input data values. Each atomic node uses nine elevation samples.

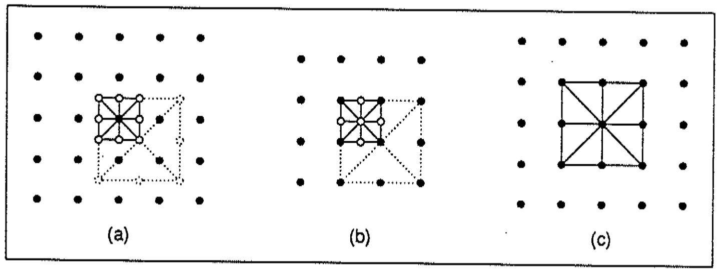

Figure 3: The construction of atomic nodes from raw input data. The black dots represent the grid; the open circles are virtual data points whose value is interpolated. The ratio of input values per atomic node are (a) 1:1; (b) 4:1; (c) 9:1. Note that when nodes merge to form larger nodes, only scheme (a) ends up with persistent interpolated values.

Approach (a) simplifies the relationship between the input data and the resulting tree, but most of the elevation values that will eventually be stored will not correspond to actual input elevation data samples, but instead are averages of two or four adjacent locations. Moreover, when the corresponding nodes merge, the true data points do not persist in the resulting node, so that the final tree may contain no elevation values that correspond to actual input values. This may cause smoothing and other undesirable effects on the resulting model. In approach (b), the number of elevation values that will eventually be stored which do not correspond to actual input values is much smaller. Moreover, when four atomic nodes are merged, all the vertices of the resulting node correspond to actual input values. Approach (c) suffers from none of the above maladies, but is somewhat complex, particularly at the periphery of the dataset.

4.3 Bottom-Up Construction Algorithm

The bottom-up construction algorithm has two operational phases:

- Input phase: construct the bottom level of atomic nodes.
- Merge phase: coalesce all mergible nodes.

In the input phase, the input is read and the atomic nodes are constructed. Depending on the implementation of atomic nodes in terms of data samples (as discussed in Section 4.2), the input is read in one, two, or three rows at a time. Note that this is done in the input's natural order. Since all the nodes have the same size, the restriction on the ratio of sizes of neighboring nodes is automatically satisfied.

In the merge phase, each level of the tree, starting with the one just above the one constructed in the input phase, is visited in turn. At each level, every leaf node is tested for mergibility. If a node is mergible, then coalesce it with its three siblings with which it shares a common parent. The four siblings are then deleted from the tree and the coalesced parent is inserted in their place.

A leaf node is mergible if the all of the following conditions hold:

- The node is the NW son of its immediate parent. (This relationship is established based on the properties of the regular decomposition rules, and is independent of whether or not such a parent node actually exists in the tree at this time.)
- The size of the node is equal to the sizes of its currently existing east, south and SE neighbors.
- Neither the node nor its above specified neighbors have any smaller-sized neighbors. Note the if such neighbors existed, then the proposed merge would create a node at least four times larger than those smaller neighbors, in violation of the restricted quadtree definition.
- The elevation values stored at the non-persistent vertices of the nodes being merged must be within tolerance of the values computed for their locations in a parent node when it is created. See Figure 4.

The algorithm terminates once all the nodes at a given level have been processed and no more merges can be performed. Termination is guaranteed since at least at the root level, if indeed the map may be merged into a single node and the process continues that far, no more merges can take place.

The input phase consists of $O(n)$ operations: there are $O(n)$ data items to be read, and each may be accessed in $O(1)$ time. $O(n)$ atomic nodes are produced, but since the construction of an atomic node is independent of all other nodes, this only requires $O(1)$ time.

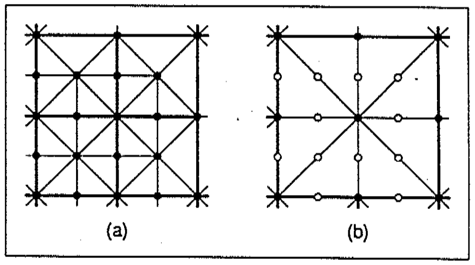

Figure 4: Merging nodes: (a) initial four leaf nodes; (b) resulting merged node. Open circles denote vertices that do not persist in the resulting merged node. Their elevations must agree, within a given tolerance, with the average of the elevations at the nodes with which they share an edge for the merge to proceed.

In the merge phase, every possible node in each tree level is tested. The total number of nodes in all levels above the bottom one is known to be $\frac{1}{3}$ of the number of the nodes in the bottom level, which is also $O(n)$. Since each node in the restricted quadtree is tested only once, the merge phase also requires $O(n)$ operations. Thus the total execution time of the two phases is $O(n)$.

The advantage of this algorithm is that it scans the input in sequence. This is important in applications which store their data on a medium which does not support random access (e.g., magnetic tape). On the other hand, the amount of work it performs is insensitive to the size of the output. Processing a surface which could adequately be modeled by a map with a single node will still take as long as building a map with nothing but atomic nodes. We shall see that the next algorithm, the top-down one, is different in these two respects.

4.4 Top-Down Construction Algorithm

This algorithm attempts to adapt the ideas of the *predictive* quadtree build algorithm described in [Shaffer and Samet, 1987]. In the course of constructing an area quadtree from raster data, the predictive algorithm only splits nodes — it never merges any. The algorithm is therefore optimal in the sense that the work it does is proportional to the size of the desired output. In principle, that algorithm maintains a partially-constructed minimal quadtree which is still consistent with the data read so far, by making the most convenient assumptions about the unread portion of the input. When all the input is processed, the existing tree is the desired result.

Like the predictive quadtree build algorithm, the top-down construction algorithm maintains a partially constructed restricted quadtree which is consistent with the data processed so far. However, the order in which the construction proceeds is driven by the levels in the tree rather than the order of the input.

As a first step, the root node is constructed, using the input elevation values associated with the locations indicated in Figure 5a. Once this has been done, (and throughout the construction process), an elevation value is associated with every point within the map's extents, reflecting either an input (stored) value or an interpolated (computed) value. Initially these values will probably be only a poor approximation of the input surface, since only 9 values are stored and the rest are computed. But as more nodes are inserted, the computed values approach those of the desired surface.

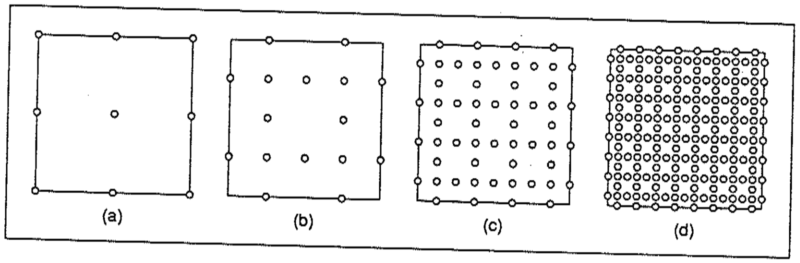

Figure 5: Locations of non-persistent vertices in the first few levels of the restricted quadtree: (a) root level; (b), (c), and (d) describe levels 2, 3, and 4, respectively. The square represents the extent of the restricted quadtree.

In each level after the first, each possible node is considered in turn. For each node, the non-persistent vertices are determined. The relative locations of these vertices for levels 2, 3, and 4 are shown in Figures 5b, 5c, and 5d, respectively. At each such non-persistent vertex, the elevation value computed from the current tree is obtained. The actual input elevation value for the same point is also read in. The two values are compared for agreement within the predefined tolerance. If a node contains a vertex whose input and computed elevations are sufficiently different, the node is inserted into the current tree.

The insertion operation is not as simple as it was in the case of the bottom-up algorithm. In particular, the insertion of a node may initiate a cascade of splits among other nodes already in the tree due to the restricted quadtree definition which constrains the sizes of adjacent nodes. For example, consider Figure 6. Figure 6a depicts a node in the tree being constructed, say at level ℓ. A pass over the next level, $\ell+1$, has not yielded any discrepancy with the input within the bounds of this node, as seen in Figure 6b. In the next step, at level $\ell+2$, an input value which differs sufficiently from the computed elevation for that location is detected (solid triangle in Figure 6c). The node at level $\ell+2$ which contains this point should now be inserted.

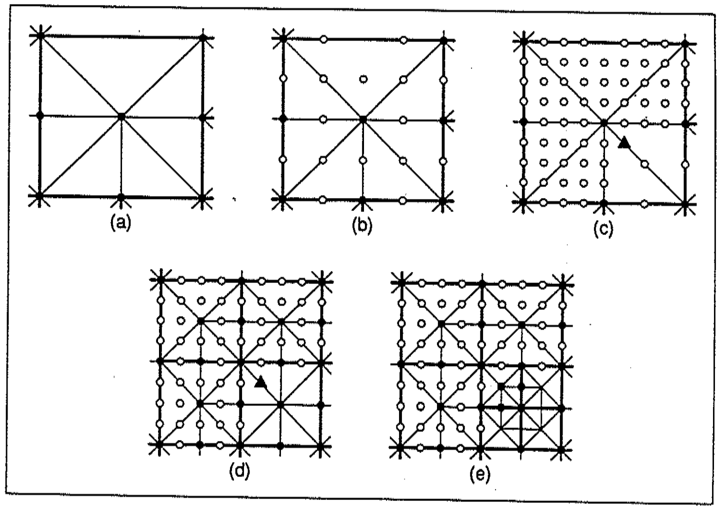

Figure 6: Example of node insertion in top-down construction: (a) Initial state of some node; (b) the situation after the construction pass over the level below the one containing the node; (c) the state when, during the pass over the second level below the node, an input value (marked by a triangle) which significantly differs from the elevation computed for the same location is encountered; (d) first split; (e) final split. Solid circles denote stored elevation values; open circles are locations where the input and computed values were found to be in agreement. For the sake of simplicity, changes required at the node boundary (which depends on *its* neighbors) are not entirely shown.

However, it is too small to be inserted directly, since its neighbors would be larger than twice its size. Its parent node in level $\ell+1$ needs to be inserted first, along with its siblings, as in Figure 6d. Only then can the small node be inserted (Figure 6e).

As we see, a node may be inserted once if it is a part of the final output, or, in the worst case, may then be split once as a result of the necessity to insert one of its descendants. No node, however, is processed more than once. Since the total number of possible nodes in the tree is bounded by $O(n)$ (as argued in Section 4.3), the algorithm's total execution time is also $O(n)$.

Unlike the bottom-up approach, in the top-down construction atomic nodes remain in the tree if and only if they are part of the final result. In that sense, the top-down algorithm is faster. However, if the input is stored lexicographically in (x, y) (as DTMs usually are), then the top-down algorithm means that the input is scanned completely out of sequence. Moreover, some input values may be read twice. As seen in the example in Figure 6, a node may be skipped when its level is processed (and its input data read), but nonetheless be inserted later due to a cascade split (when its input is read again). Clearly, none of these concerns apply if sufficient RAM is available to store the entire input dataset.

Very large datasets may still be supported by a combination of both the bottom-up and the top-down algorithms. Qualitatively, the area covered by the input may be decomposed into a square array of square blocks, each small enough to fit entirely into the available RAM. The top-down algorithm may be run on each one separately, yielding a forest of restricted quadtrees. Next, the bottom-up algorithm may be applied to the level of the roots of the existing trees, performing any merges required to form a single tree from all of them. Theoretically, the combined algorithm may now merge nodes and therefore no longer possesses the optimality the top-down algorithm claims. However, at least topographical surfaces tend to be sufficiently uniform so that most of their leaf nodes concentrate within a few levels far from the root, and under these conditions, merges are not likely even in the combined algorithm.

5 Conclusion

We have described two algorithms for constructing a restricted quadtree to model a surface given by its digital terrain map. Although one does less work, it reads the input nonsequentially, which may cause it in practice to run slower on large datasets. A combination of the two algorithms is proposed, which for many practical datasets may alleviate the restriction on size.

References

[Barrera and Hinojosa, 1987] R. Barrera and A. Hinojosa. Compression method for terrain relief. Technical report, CINVESTAV – IPN, Engineering Projects section, Department of Electrical Engineering, Polytechnic University of Mexico, Mexico City, 1987.

[Samet, 1990a] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[Samet, 1990b] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.

[Shaffer and Samet, 1987] C.A. Shaffer and H. Samet. Optimal quadtree construction algorithms. *Computer Vision, Graphics, and Image Processing*, 37(3):402–419, March 1987.

[Von Herzen, 1989] Brian Von Herzen. *Applications of Surface Networks to Sampling Problems in Computer Graphics.* PhD thesis, California Institute of Technology, Pasadena, CA, July 1989.

[Von Herzen and Barr, 1987] B. Von Herzen and A.H. Barr. Accurate triangulations of deformed, intersecting surfaces. *Computer Graphics*, 21(4), July 1987.

PROCEEDINGS

5TH

INTERNATIONAL SYMPOSIUM ON

SPATIAL DATA HANDLING

IGU Commision on GIS

August 3 - 7, 1992
Charleston, South Carolina
USA

Volume 1