LARGE SCALE EDITING AND VECTOR TO RASTER CONVERSION VIA QUADTREE SPATIAL INDEXING

Clifford A. Shaffer

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106 USA

Mahesh T. Ursekar

Mapping Science Division
Intergraph, IW17A3
Huntsville, AL 35894

Abstract

We describe the use of various quadtree data structure variants for map editing and vector to raster conversion of large scale databases (e.g., millions of line segments). The *PMR quadtree* is a representation for line segments that supports fast detection of line segment intersections and gaps in polygon boundaries. These capabilities allow quick processing of the database to detect most topological inconsistencies. The data structure also supports interactive editing operations such as displaying relevant areas, modifying line segments to correct errors, and adding new line segments to separate e.g. seas from oceans. Techniques for minimizing line segment endpoint duplication are also discussed. Finally, a novel approach for converting the database from vector to raster format is presented. This technique first converts the vector database to a *line quadtree* (similar to a region quadtree with the addition of rectilinear polygon boundary information) and then uses a flood-fill algorithm to tag quadtree nodes associated with logical polygons (countries, islands and lakes). In the final stage, the line quadtree is converted to a region quadtree.

Introduction

As GIS technology progresses, users will wish to store ever larger geographic databases. Depending on the application, geographic databases today are commonly stored either in vector (i.e., boundary) or raster (i.e., areal) format. It is commonly recognized that each type of representation has advantages and disadvantages, and traditional algorithms for converting between them are well known (see, for example, Burrough, 1986; Franklin, 1979; Peuquet, 1981). Recent advances in hierarchical data structures warrants study of their application to the vector to raster conversion process. In particular, indexing data via a hierarchical representation effectively sorts the vectors, which is at the heart of traditional algorithms. So far as we know, this paper is the first report of an attempt to apply hierarchical data structures to the conversion process.

We present data structures and algorithms to support editing and rasterizing extremely large vector databases. We also discuss methods for minimizing duplication of vector endpoints, a topic relevant to any boundary representation. We have implemented our algorithms for use with the CIA World Data Bank II (WDB2) (CIA 1977), a public domain vector database of the entire world to one second resolution. We successfully rasterized that portion of the WDB2 consisting of all country, coast, and lake boundaries – approximately two million line segments.

The WDB2 is a typical "spaghetti" file in that it is simply a collection of ***chains****, where each chain is a series of connected data points (vertices) with specified position. There is no required organization or relationship between various chains. Prior to rasterization, we discovered that many errors are contained in the WDB2 such that it does not form a collection of well-formed polygons. Typical problems included small gaps along a polygon boundary (probably undershoots during digitization), and line segments that stick out past the polygon boundary – in particular for country boundaries crossing coastlines (probably overshoots during digitization). In both cases, such problems can be determined by finding vertices of degree one. We also would like to remove intersections between line segments not at a vertex point. The spatial indexing method described below supports efficient location and removal of such errors.

Before discussing the various algorithms, we first describe several quadtree representations that we used. For further details on these variants, see the books by Samet (1990a, 1990b). The *region quadtree* is the simplest and best known form of quadtree, and is most appropriate for storing raster images containing homogeneous regions. The region quadtree represents a homogeneous array of $2^n \times 2^n$ pixels as a single block, corresponding to a leaf node in the quadtree. Arrays whose pixel values are not all the same are subdivided into four equal-sized sub-arrays represented by an internal node with four children (labeled NW, NE, SW, and SE as in Figure 1d), each representing a corresponding sub-array. These sub-arrays are recursively divided into smaller sub-arrays until each such block is homogeneous. Figure 1 illustrates the quadtree representation for a raster image.

The *line quadtree* (Samet and Webber, 1984) is similar to the region quadtree, except that each leaf node of the tree is augmented by a field of 4 bits, one bit for each edge of the block represented by that node. A bit is *on* if the corresponding edge of the block is part of the region boundary. Otherwise, the bit is *off*. Note that the entire block edge must be part of the boundary, or else the block must be decomposed. Figure 2 illustrates the line quadtree representation for the region quadtree in Figure 1c. In order to save space and processing time, we have modified the line quadtree to minimize boundary duplicates. Most line segment boundaries fall between two nodes. We store such boundaries only with the eastern and southern nodes of such pairs.

The *PM quadtree* (Samet and Webber, 1985) is used to store a collection of line segments.* If all line segments in the database adhere to some decomposition criteria, then the line segments are simply stored in a leaf node corresponding to the entire image. If the line segments fail to meet the criteria, then the region is subdivided into quadrants, and each quadrant recursively tests the line segments contained within it against the decomposition criteria. Figure 3 illustrates a variant of the PM quadtree (as described by Nelson and Samet, 1986) which during the insertion process splits a node once whenever a new line segment added to the node brings the total number of line segments within that node above some threshold (in Figure 4, the threshold is 2).

* These *chains* are called *segments* in the literature. We use the term chain so as not to confuse it with a *line segment*, a single line connecting two endpoints.

Vector Indexing and Editing

Our first task was to create a spatial index for the vector database that would support efficient detection of inconsistencies and allow their correction. We decided to use a variant of the PM quadtree for this purpose. Since the database is so large, the index (and the database) must be maintained on disk, with sections brought into memory when required. To simplify disk management, we determined that our tree representation should have fixed sized nodes. We also decided to use a pointer based representation (i.e., we explicitly store pointers from a node to its four children) rather than representing the tree via a *linear quadtree* (Gargantini, 1982). The various merits of pointer vs. linear quadtrees are beyond the scope of this paper; we chose the pointer version for simplicity and because a related research project will create file management software for disk-based pointer quadtrees that is more efficient than disk-based linear quadtrees (Brown, et al, 1991). As a result of these decisions, our tree is represented on disk as a series of fixed-size nodes, with the nodes maintained roughly in order of a preorder traversal of the quadtree. Once the initial index for the database is built, all other processing does little to disrupt this order (although the order is disrupted somewhat during editing), so our simple disk buffering scheme did not negatively affect performance.

Our requirements that quadtree nodes be of fixed size fits well with the PM quadtree paradigm of decomposing any set of line segments that is "too complicated". Line segment endpoint coordinates (at one second resolution) are stored as 48 bit quantities, which includes the x and y coordinates as well as some bits available for various flags. Our initial node representation simply stored up to five line segments (10 endpoints) in each node. This approach requires that no more than five line segments meet at a single vertex. This is an undesirable limitation, but one not exceeded in our test data (rarely do five polygons meet at a single point in real maps). The major objection to simply storing the line segments within the node is that endpoints are frequently duplicated – in particular, all endpoints are duplicated for a polygonal chain. Note that not all endpoints stored with a node actually fall within that node. We store in each node both endpoints for any line segment passing through that node.

Our second node representation attempts to minimize the amount of endpoint

* (PM stands for *Polygonal Map*, although to date nobody has reported using it to represent polygons as distinct from a collection of line segments).

duplication by reserving one bit from each 48 bit endpoint descriptor to be used as a continuation flag. If the flag is set, then the current endpoint is connected to the next endpoint in the node. Thus, a chain of n line segments requires only $n+1$ endpoint descriptors. Using this method, an endpoint of degree five requires only eight endpoints (described by two chains of two line segments requiring three endpoints each, and another line segment requiring two endpoints, as illustrated by Figure 4b). Thus, the node representation that we implemented stores a maximum of eight endpoints and their flags. Figure 4 illustrates our representation. This representation does a good job of minimizing repeated storage of endpoints, and reduces the total storage required by about 50the naive method of storing both endpoints of all line segments. This method still requires some duplication of endpoints within a node, and we are studying other methods to achieve further compression.

The PM quadtree for our database, using the second node represenation described, required approximately 50 Mbytes of disk space. Of this amount, about 38 Mbytes can be attributed to overhead of the indexing scheme (as opposed to approximately 12 Mbytes required to store the line segments).

Our next step was to locate all illegal intersections (intersections between two line segments other than at their endpoints) and all "orphan" line segments (a line segment with a vertex of degree 1). Removing orphans does not guarentee that the resulting collection is well formed, (as illustrated by Figure 5), but for our database it probably resolves all inconsistencies. If not, the rasterization process will still generate "reasonable" polygons, even for the example of Figure 5. Locating orphans/intersections was combined with the building process.

Building the tree and locating the intersections/orphans for the two million line segment database required approximately 37 hours on a Mac II running A/UX (Apple's UNIX). If this appears excessive, we urge the reader to consider the following points. 1) The machine used was an original Mac II (Motorola 63030 processor at 16 Mhertz). 2) The algorithm is inefficiently coded since we created it to run only once. Code tuning can greatly improve the running time; compacting the representation (for example by storing the endpoints in 40 bits instead of 48, or reducing the duplicate endpoints further) will also improve performance by minimizing disk fetches. 3) The tree construction needs to be done (once) anyway to provide an spatial index for many other operations. 4) Given the tremendous number of vectors and an $O(n\log n)$ algorithm, this is really not so slow. 2000 line segments should require only about one minute with our current program.

We also created a display and editing program that read the file of intersections/orphans. Approximately 5000 intersections/orphans were found. For each such inconsistency point in turn, the area around that point was displayed. We then hand modified the database to correct the problem by removing, adding, or changing line segments. While automatic programs exist to resolve such inconsistencies, it is a very complicated, heuristic process. Our approach required approximately one week of editing, and allowed for human judgement in the process. We note that the PM quadtree performed so well as a data index that the bottleneck for displaying portions of the database was not time to fetch data from disk, but rather

time for the Mac's relatively poor graphics capabilities to display the lines on the screen.

Rasterization

After the database had been indexed and all inconsistencies removed, we performed rasterization. Franklin (1979) and Peuquet (1981) each present a variety of algorithms for vector to raster conversion. Most of these algorithms process either the vector set or the raster in strips if insufficient memory is available for the entire process. Our approach is to process the vector database by traversing the PM quadtree and rasterizing each line segment in turn. In order to save space and to minimize disk fetches, we rasterize to a line quadtree rather than to an array. In effect, our algorithm assumes random access to both the input PM quadtree and the output line quadtree. While we use our own buffer pool to manage quadtree node pages, this is similar to relying on virtual memory to handle main memory limitations. By processing the PM quadtree in preorder traversal, we build the line quadtree in approximately preorder traversal as well. Thus, our algorithm requires little swapping of pages. The line quadtree allows for simply rasterizing each vector and marking borders in the appropriate nodes of the line quadtree.

One advantage of this method is that we can rasterize the vector database at any scale (given sufficient disk space). Storing a complete array at one second resolution would require nearly 1000 gigabytes of disk space if only one byte of information were required at each pixel. The region quadtree can reduce this requirement by perhaps two or three orders of magnitude, but this still requires more disk space than we had available. For our tests, we scaled by a factor of 150 in each dimension. This lead to a problem of mismatched resolution between the input line segments and the output rasterization which showed itself in the form of spurious small polygons as illustrated by Figure 6. Proper generalization and filtering of the line segments to match the scaling factor should eliminate this problem. Note that these spurious polygons inflate both the time required for all remaining steps and the space required.

The next step is to assign a polygon label to each block of the line quadtree. We simply traverse the line quadtree in preorder, and for each leaf node encountered with no polygon value, we perform a standard floodfill algorithm (modified for quadtrees) using that node as the seed point. The final step is to convert the line quadtree to a region quadtree, in order to save space. In our test, the line quadtree required 4.8 Mbytes of space, while the corresponding region quadtree required 3.6 Mbytes. Building the line quadtree required approximately 81 minutes of CPU time.

For further details on all aspects of this project, see Ursekar (1991).

References

1. Brown, P.R., Shaffer, C.A. and Webber, R.E. 1991. A paging scheme for pointer-based quadtrees, *Proceedings of the Nineteenth Annual ACM Computer Science Conference*, San Antonio, TX, March 1991, 687.

2. Burrough, P.A. 1986. *Principles of Geographical Information Systems for Land Resources Assessment*, Clarendon Press, Oxford.

3. Central Intelligence Agency, 1977. *World Data Bank II General Users Guide*, PB-271 869, July 1977.

4. Franklin, W.R. 1979. Evaluation of Algorithms to Display Vector Plots on Raster Devices, *Computer Graphics and Image Processing 11*, 4(December 1979), 377-397.

5. Gargantini, I. 1982. An effective way to represent quadtrees, *Communications of the ACM 25*, 12(December 1982), 905-910.

6. Nelson, R.C. and Samet, H. 1986. A consistent hierarchical representation for vector data, *Computer Graphics 20*, 4(August 1986), 197-206.

7. Peuquet, D.J. 1981. An examination of techniques for reformatting digital cartographic data/part 2: The vector to raster process, *Cartographica 18*, 3, 21-33, 1981.

8. Samet, H. and Webber, R.E. 1984. On Encoding Boundaries with Quadtrees, *Pattern Analysis and Machine Intelligence 6*, 3(May 1984), 365-369.

9. Samet, H. and Webber, R.E. 1985. Storing a collection of polygons using quadtrees, *ACM Transactions on Graphics 4*, 3(July 1985), 182-222.

10. Samet, H. 1990a. *Applications of Spatial Data Structures*, Addison Wesley, Reading MA.

11. Samet, H. 1990b. *The Design and Analysis of Spatial Data Structures*, Addison Wesley, Reading MA.

12. Ursekar, M.T. 1991. *Rasterizing the CIA World Data Bank II*, Master's Project Report, Computer Science Department, Virginia Tech, Blacksburg VA.

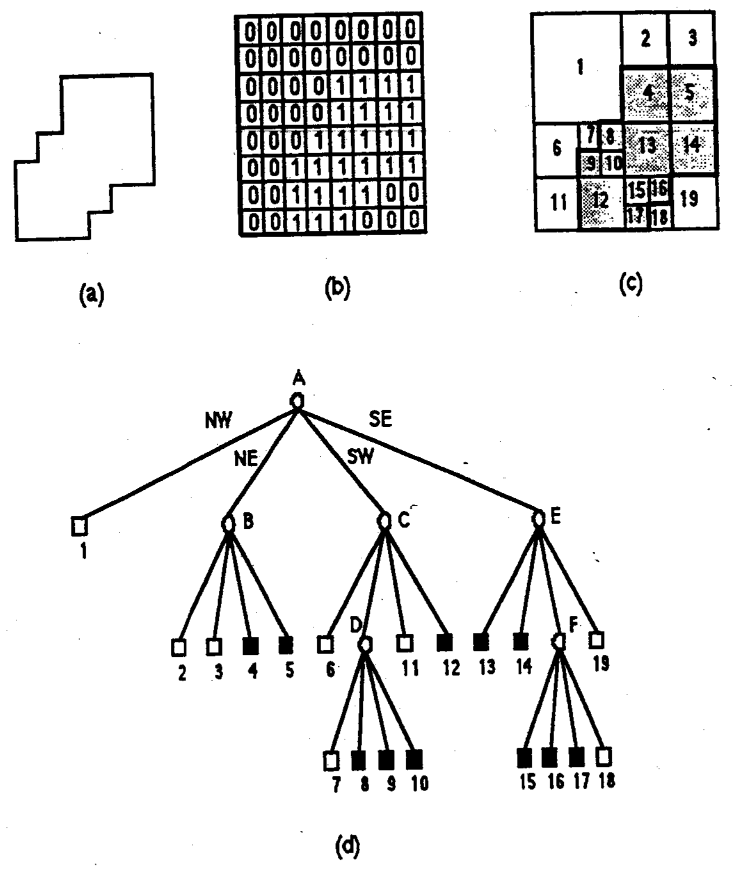

Figure 1. A region, its binary array, its maximal blocks, and the corresponding quadtree. (a) Region. (b) Binary array. (c) Block decomposition of the image in (a). Blocks within the region are shaded. (d) Quadtree representation of the blocks in (c).

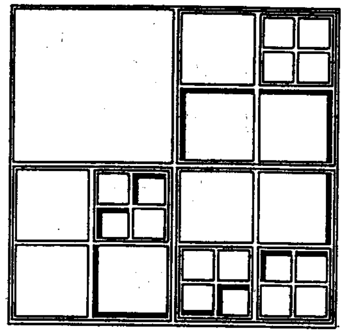

Figure 2. The line quadtree for the image of Figure 1.

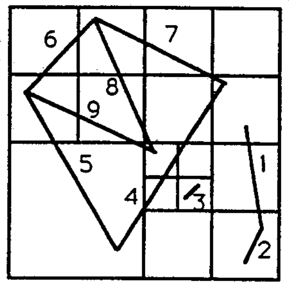

Figure 3. The PMR quadtree for a set of line segments inserted in the order of their labels.

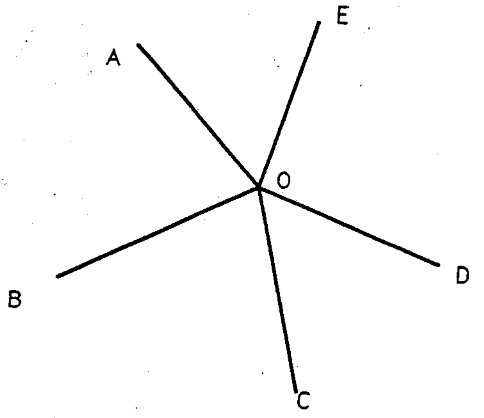

Figure 4. A set of five line segments meeing at a point. The naive method of endpoint storage would store five separate line segments, and thus ten endpoints: $\overline{AO}$, $\overline{BO}$, $\overline{CO}$, $\overline{DO}$ and $\overline{AO}$. Our revised method stores chains $\overline{AOB}$, $\overline{COD}$ and $\overline{EO}$ for a total of eight endpoints.

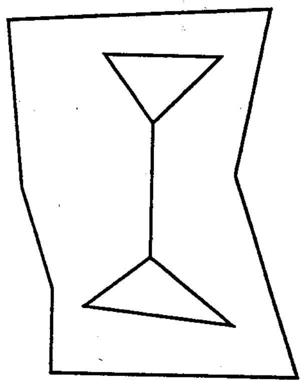

Figure 5. An example of illegal topology that won't be caught by local operations. However, the rasterization of this example would yield a reasonable set of polygons, equivalent to removing the centermost line segment.

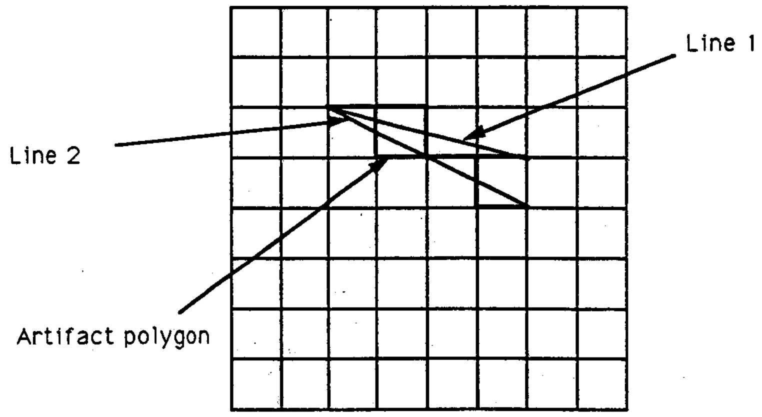

Figure 6. An example of the formation of undesirable polygons due to high resolution line segments overlayed on a low resolution grid.