OVERPROP — Calculating Map Overlay Polygons' Areas Without Explicitly Calculating the Polygons

Wm. Randolph Franklin

Venkateshkumar Sivaswami

Electrical, Computer, and Systems Engineering Dept. 6026 JEC,
Rensselaer Polytechnic Institute,
Troy, NY, 12180, USA

Internet: wrf@ecse.rpi.edu, Bitnet: wrfrankl@rpitsmts.bitnet
Telephone: (518) 276–6077
Telex: 6716050 RPI TROU
Fax: (518) 276–6261

ABSTRACT

We present OVERPROP, an algorithm for taking two or more input maps, and calculating the areas and perimeters of all the polygons in the resulting overlaid map. The input maps could be in any reasonable edge-based format, such as the Harvard Odyssey one. This algorithm works from a reduced representation of each map as a set of "half-edges" with no global topology. It does not first calculate the overlaid map. In fact, finding the areas of all the output polygons is simpler than finding the polygons themselves. This method is exact within the arithmetic precision of the machine, and does not use raster, Monte Carlo, or other sampling techniques. It is well suited to a parallel machine, and could be extended to overlaying more than two maps simultaneously. It is useful when the sole purpose of performing an overlay is to find some mass property of the resulting polygons.

In *Second National Conference on Geographic Information Systems*, pages 1646–1654, Ottawa, 5-8 March 1990.

1. INTRODUCTION

In the map overlay problem, two maps, or planar graphs, are combined to make a third, each of whose polygons represents the intersection of one polygon from the first input map with one from the second. For instance, if the first map's polygons are provinces and the second watersheds, then one output polygon would be that part of Ontario that drains into Hudson Bay. The overlay problem is one of the more difficult computational issues in GIS since the algorithm is complex, the data are large, and the numerical inaccuracy of computers can confound even correct algorithms [8, 9].

Our algorithm, OVERPROP, presents a simpler method, more efficient for large databases, that should be less sensitive to numerical errors. It is based on our simpler, local topological data structures for representing polygons and polyhedra. Its speed comes from the uniform grid data structure. These concepts have been described in the context of computer aided design (CAD) most recently in [3]. In the rest of this paper, we will see, first, the logical data structures, then the theoretical basis of the algorithm. We will see the algorithm described first briefly ignoring efficiency considerations, and then in more detail. Control of numerical errors, design tradeoffs, and possible extensions complete of the paper.

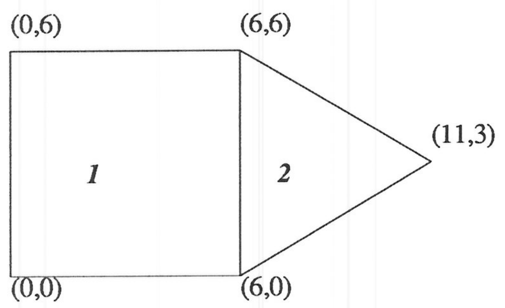
2. MAJOR DATA STRUCTURES

Conceptually, each input map is a planar graph composed of vertices and edges partitioning the E^2 plane into polygons. The graph may be disconnected with the components either nested or disjoint. Each polygon has an identification number, which may be non-unique when one logical region is composed of several polygons, such as British Columbia if the regions are provinces. By convention, the outside is polygon #0. In our map data structure, the vertices and polygons are not represented explicitly, but could be recreated if desired.

We assume that each input map is a set of tagged edges:

$$\{(x_1, y_1, x_2, y_2, p_l, p_r)\}$$

where (x_1, y_1) are the Cartesian coordinates of the edge's first endpoint, and (x_2, y_2) are the second. p_l and p_r are the numbers of the polygons to the left and right. For example, consider figure 1, which has 2 polygons, 5 vertices, and 6 edges. The data structure is as follows.



$(x_1, y_1, x_2, y_2, p_l, p_r)$:

(0,0,6,0,1,0)

(6,0,6,6,1,2)

(6,6,0,6,1,0)

(0,6,0,0,1,0)

(6,0,11,3,2,0)

(11,3,6,6,2,0)

Figure 1: Example of Map Data Structure

Each edge of each polygon has two endpoints, each of which is a "half-edge". The vertex at (0,0) can be considered to be half on edge (0,0,6,0) (where those four numbers are respectively (x_1, y_1, x_2, y_2)) and half on edge (0,6,0,0). (6,0) is considered to be two vertices, one for each polygon. The edge (6,6,0,6) is on 2 polygons, and for each of these polygons the edge has two half vertices. Although the half-edges are not explicit data structure elements, they can be derived easily, and are important in the following algorithm. A half-edge contains the following information.

$$(v_x, v_y, t_x, t_y, n_x, n_y)$$

where (v_x, v_y) is the location of the point, (t_x, t_y) is a unit tangent vector from the vertex along the edge, and (n_x, n_y) is a unit normal vector perpendicular to the tangent, and pointing into the polygon. For example, the two half vertices of the first edge above are

(0,0,1,0,0,1)
(6,0,-1,0,0,1)

A polygon number of the output map is an ordered pair (p_1, p_2) of the two input polygons whose intersection forms this particular output polygon. The output polygons' areas are accumulated piecemeal in a set of the form $(p_1, p_2, \textit{partial-area})$. There may be at times many set elements with the same first two components.

3. THEORETICAL BASIS OF THE ALGORITHM

The algorithm is built on the following principle. Many properties of a polygon, such as area, center of mass, and moments of inertia, may be calculated separately for each half-edge of the polygon and then summed. These are "extensive" properties in a thermodynamic sense, in that the total value for an object may be found by integrating over its area or volume. This concept is also related to Greene's theorem in calculus, where an integral over an area is transformed into an equivalent integral over the boundary of the area. Formally, let polygon P have the set of half-edges V. Let $F(P)$ be some desired function of the polygon, such as area. Then for these F, there exists a functional that returns a new function f_F such that

$$F(P) = \sum_V f_F(V)$$

For example, if F=*area* then

$$f_{area} = \frac{1}{2}(v_x t_x + v_y t_y)(v_x n_x + v_y n_y)$$

If F=*perimeter* then

$$f_{perimeter} = -(v_x t_x + v_y t_y)$$

So if we can find the half-edges of the output polygons we can find the areas. The output half-edges needed by the formulae are of two types:

1. derived from an endpoint of an original edge of one of the input maps, or
2. derived from an intersection of two input map edges.

For each original endpoint we also need to know which polygon of the other map this point is in. For each intersection of two edges we already know the numbers of the relevant polygons.

4. BRIEF ALGORITHM

A sketch of the algorithm, ignoring efficiency considerations, now appears as follows.

1. Initialize a data structure to contain a list of triples, $(p_1, p_2, \textit{partial-area})$, to accumulate the parts of the areas of the output polygons. p_1 and p_2 are polygons from the first and second input maps respectively.
2. Find all intersections of any edge of map 1 with any edge of map 2. Each intersection will generate 8 half-edges, as shown in figure 2 where the tangents and normals of three are labeled.

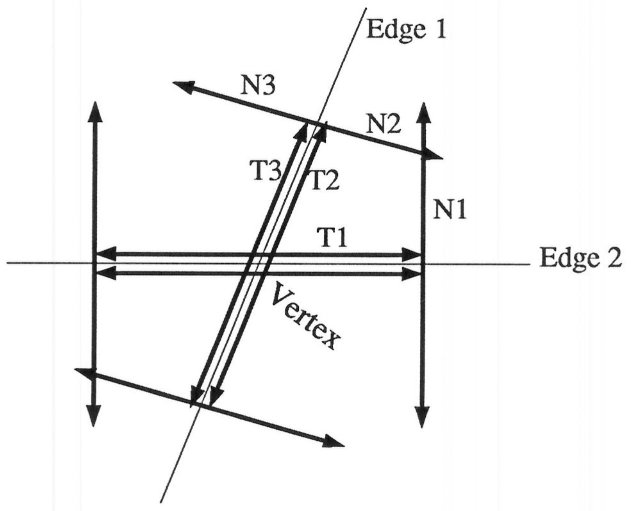


Figure 2: The 8 Half-Edges Derived from One Intersection

3. For each half-edge resulting from an intersection, find the tangent and normal vectors and the relevant polygons from each input map. Note that in figure 2 there are two polygons from map 1, one on each side of edge 1, and also two from map 2, one on each side of edge 2. Each of the 4 possible pairs of the input polygons will apply to 2 of the 8 half-edges. For example, half-edges 1 and 2 are part of the output polygon that is the intersection of the polygon on map 1 to the right of edge 1 with the polygon on map 2 above edge 2.
4. Apply the formula to each half-edge to calculate its contribution to the relevant output polygon's area.
5. Now we must calculate and process the half-edges not resulting from an intersection. For each endpoint of each edge of map 1, determine which polygon of map 2 contains it. Repeat for the edges of map 2.
6. Split each edge of each input map into four half-edges.
7. Apply the formula to these half-edges and store their partial areas.
8. Extract the final area of each non-empty output polygon, which was the goal.

5. DETAILED EFFICIENT ALGORITHM

The algorithm presented in the last section is of value only if it is efficient enough. For instance, if finding all intersections of input edges takes quadratic time in their number, then maps with 10,000 edges, and thus 50,000,000 possible intersection, will be difficult to process. This section tells how to make the algorithm practical.

1. *Finding intersections.* Franklin's uniform grid method [3, 5] can determine these sufficiently quickly, in expected time proportional to the number of input edges, N, plus the number of output intersections, K. To protect against worst-case input, Chazelle and Edelsbrunner's algorithm [2] can be used to find the intersections in time proportional to $(N \log N + K)$. However, this algorithm is much more complicated, although, unlike many computational geometry theoretically optimal algorithms, it has been implemented.
2. *Locating the polygon containing each edge endpoint.* The naive method compares each point against each edge of the other map. However, using a uniform grid again, this can be done in expected time proportional to $(N+M)$ if there are N points to test against a map with M edges. An implementation of a one-dimensional grid method by V. Sivaswami has the following performance on a Sun 3/60 workstation.

   a. The first case tested 10,000 points against a random polygon with 100,000 edges. Preprocessing took 190 μsec/edge, or 19 seconds CPU time, and classification took 7.6 msec/point, or 76 seconds.
   b. In the second test, we took a map of the USA with 915 edges in 166 chains and 49 polygons, and located 930 random points. Preprocessing took 440 μsec per edge and location took 3.8 msec per point, for a total preprocessing and query time of 3.9 seconds.

In these tests, the total time varied less than 25% as the linear grid resolution varied over a range of 25:1. The sample data are shown in figure 3.

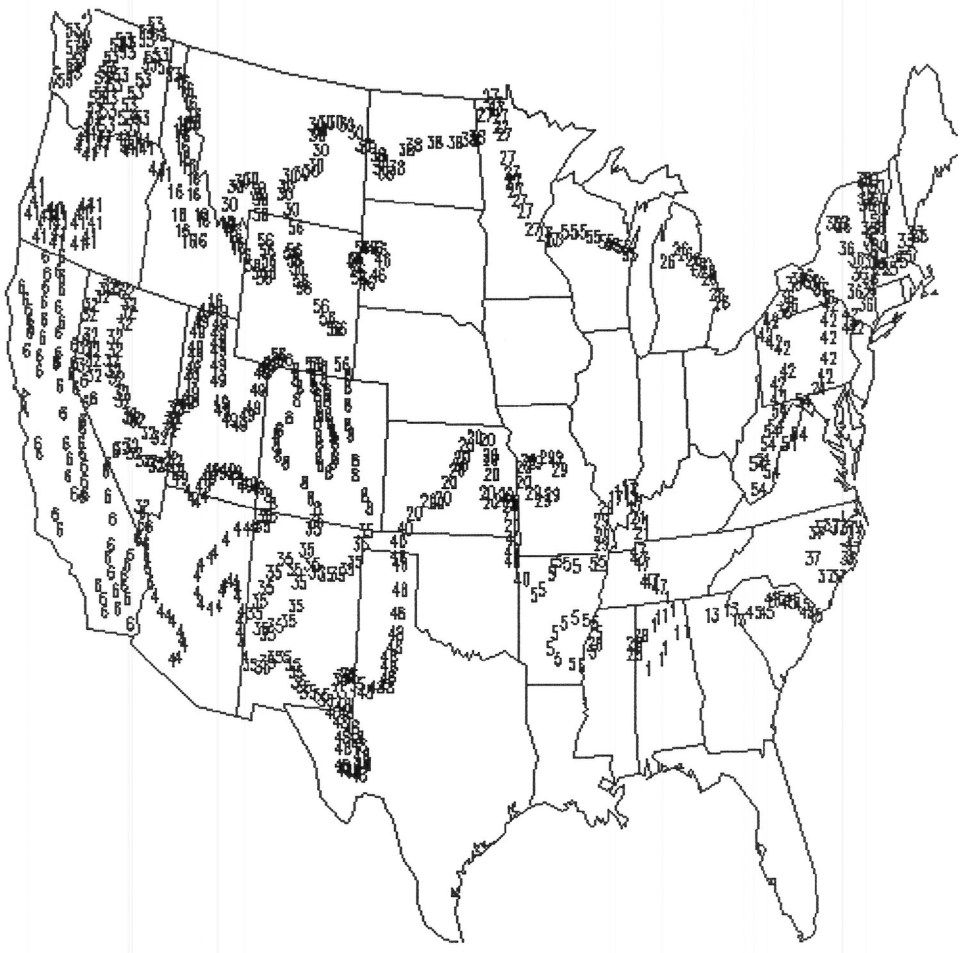


Figure 3: Point Classification in a Planar Graph. Each Point is Labeled with its Calculated Polygon Number.

If the user is feeling pessimistic about the data, and worried that the uniform grid might fail, then there do exist more complicated but worst-case optimal methods [6]. A typical time for these might be *((N+M) log M).*

3. *Implementing the list of triples.* The obvious methods are either to sort the list, or to use a hash table keyed on the pair of polygon numbers. Sorting is simpler but hashing is faster. It would be wrong to have a square array of pointers to lists, one list for each pair of input edges, since most edges of map 1 do not intersect any given edge of map 2.

We expect to have a working implementation of he full OVERPROP tested on a large database (>10,000 input polygons) in a few months. As described above, the point location program is implemented and working well.

6. THEORETICAL ANALYSIS OF POINT LOCATION

Let the average edge length of the input data be L, number of edges in the input be N and the number of grid cells be G. The width of each grid cell is therefore 1/G.

To calculate the number of cells that an average edge projects onto, we note that $L\cos\theta$ is the projected length of the edge where θ is the angle that the edge makes with the projection line. Taking an average value, we get,
Number of cells an average edge projects onto = $1 + \frac{2}{\pi} GL$
Number of (cell, edge) pairs = $N\left(1 + \frac{2}{\pi}GL\right)$
Average number of edges per cell = $\frac{N}{G}\left(1 + \frac{2}{\pi}GL\right)$

The preprocessing time consists of two parts:

a. Time to create the grid structure $T_1 = \alpha_1 G$

b. Time to classify edges into cells $T_2 = \alpha_2 N \left(1 + \frac{2}{\pi} GL\right)$

The classification time T_3, which depends on the average number of edges per cell, is given by, $T_3 = \alpha_3 \frac{N}{G} \left(1 + \frac{2}{\pi} GL\right)$

The total time is given by,

$$T = T_1 + T_2 + T_3$$
$$= \alpha_1 G + \alpha_2 N \left(1 + \frac{2}{\pi} GL\right) + \alpha_3 \frac{N}{G} \left(1 + \frac{2}{\pi} GL\right)$$

Minimizing T wrt G, we get,

$\alpha_1 + \frac{2}{\pi}\alpha_2 N L - \alpha_3 \frac{N}{G^2} = 0$ from which we have,

$G = \sqrt{\frac{N}{k_1 + k_2 N L}}$ for minimum T, where k_1 and k_2 are some constants.

We will consider the storage requirement for an implementation in which the vertices of the polygon are stored in an array and the slabs are represented as an array with each element (slab) pointing to a list of edges. The space complexity is made up of three parts : $\Theta(N)$ for the array of vertices, $\Theta(M)$ for the array of slabs, where M is the maximum number of slabs needed, and $\Theta(N \left(1 + \frac{2}{\pi} GL\right))$ for the (cell,edge) pairs. The overall space complexity is determined by the relative values of N, M and the value of G.

7. ERROR CONTROL

Numerical roundoff errors, and the numerous sliver polygons generated even if there is no roundoff, help make the polygon overlay problem so hard. With OVERPROP, slight errors in calculating intersections vertex coordinates will cause proportionally slight errors in the areas. However, although OVERPROP does not explicitly use the topology of the polygons, it implicitly assumes that it is consistent in the following ways.

1. There are no missing edges, and no gaps between edges of a polygon.
2. The polygon numbers stored with the edges are all correct.

In fact the sensitivity of OVERPROP to these input errors may be used to test for consistency by the following method.

1. Repeatedly randomly translate and rotate the input maps.
2. Find the areas with OVERPROP.
3. If the answers are different by more than numerical roundoff, then the input is bad. If the answers agree after several random operations, then the input is almost certainly internally consistent.

OVERPROP does not care about roundoff errors that cause one chain of edges to cross another erroneously as shown in figure 3. The will cause the output to be off by an amount proportional to the geometric error in spite of this topological error.

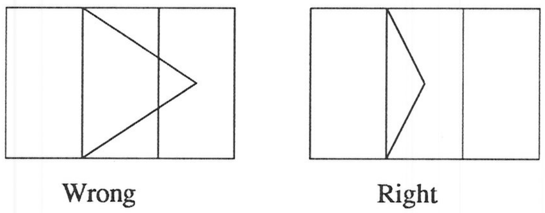


Figure 4: Topological Error Caused by Numerical Inaccuracy

8. DESIGN TRADEOFFS

The data structures and algorithms used here were selected from many possibilities. Factors considered included efficiency of execution and efficiency of implementation, i.e. simplicity. Some methods not used included the following.

1. *Storing the vertices in a separate array and having the edges contain vertex numbers instead of vertex coordinates.* This is the choice between immediate versus indirect data. Storing the vertex coordinates

immediately in the edges makes processing the edges cheaper. Depending on wordlengths, the storage will be the same or greater. If a vertex position takes 2 bytes each for x and y, and a vertex number takes 4 bytes, then the storage is identical. On the other extreme, if each coordinate takes 4 bytes, but a vertex number takes only 2, then the indirect method will be smaller. With memory prices for workstations at $100 per megabyte and falling, then memory is not so important as before. However, big programs will always execute more slowly than otherwise equivalent small programs because of bus bandwidth limitations, and because the big programs will use the cache less efficiently.

2. *Using integers instead of reals for coordinates.* This saves space and execution time. However, attention must be paid to scaling since that is no longer automatic.
3. *Splitting long chains of edges that form a common boundary into the separate edges.* This somewhat controversial choice makes the data structure bigger, but each data element is now fixed length instead of variable. This is a similar concept to a relational database satisfying Codd's third normal form [7]. Also the algorithm is simpler with fixed length items, and parallelizes better.

9. POSSIBLE EXTENSIONS

9.1 OVERLAYING MULTIPLE INPUT MAPS SIMULTANEOUSLY

These ideas extend to finding the areas of the results of overlaying more than two input maps simultaneously. As with two maps, the output half-edges are derived from input edge endpoints and intersections of edges. What is new is that the location of each half-edge must be determined in every input map. In this case, the advantage of OVERPROP compared to actually finding a sequence of more and more complicated intermediate maps would be even greater. There would also be no artifacts resulting from the order in which the several input maps were processed, since they would all be used together.

9.2 EXECUTION ON PARALLEL MACHINES

OVERPROP is designed also to be implementable on a parallel machine. The component operations are generally one of two types.

1. Map a function over a set of elements to calculate a new set, or
2. sort a set of elements, perhaps combining adjacent elements.

Mapping is easily executable even on a SIMD (single instruction, multiple data stream) machine. Sorting is a little harder, but parallel sort algorithms exist.

This ease of parallel execution contrasts to many other, more complicated, geometric algorithms that are essentially sequential. We have not implemented this complete algorithm in parallel yet, but Chandrasekhar Narayanaswami and Mohan Kankanhalli have implemented the edge intersection part in parallel, as well as other other algorithms, [1, 4]. The machines used have been a 16 processor Sequent Balance 21000 and a 32 processor hypercube. Both are MIMD machines. The Sequent has a common memory, runs Unix, and makes it easy to allocate different processors to different processes, or to different iterations of the same loop of the same process. As many as 15 of the 16 processors can be used in this case. Typical speedups when using 15 processors have been ten times faster than when using one processor.

The Hypercube has a separate, rather small, memory for each processor, and communication between processors is somewhat slow. Since one processor has too little memory to run a reasonable sized problem, we couldn't compare a sequential and a parallel time. Therefore we measured

1. the communication time compared to the total CPU time, and
2. the CPU time of the slowest processor compared to the average of all the processors.

The communication time was 1/3 or less of the total time, and the slowest processor took no more than twice the average processor. Therefore if the load were perfectly balanced over the 32 processors and had no communication cost, it would be under three times faster, which is a respectable performance.

10. SUMMARY

OVERPROP, unlike other polygon overlay algorithms uses no explicit global topology. There is no tracing chains of edges. Polygons completely contained inside other polygons are not a problem, in fact are not even recognized; however the correct answer is produced with the area of the outer polygon excluding the inner area. Polygons may also have multiple separate components. OVERPROP is a result of an investigation into how little topology we actually need explicitly to store, and concurrently, by how much can special cases be reduced. Sometimes a more complete topology is needed, for instance to draw the output polygons, but for mass calculations it is not.

11. ACKNOWLEDGEMENTS

This work was supported by NSF Presidential Young Investigator grant CCR-8351942. Partial support for this work was provided by the Directorate for Computer and Information Science and Engineering, NSF Grant No. CDA-8805910. We also used equipment at the Computer Science Department and Rensselaer Design Research Center at RPI. Part of this work was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by and operates under contract to DARPA and the Air Force Systems Command, Rome Air Development Center (RADC), Griffiss Air Force Base, NY, under contract # F306002-88-C-0031. Part of the research reported here was made possible through the support of the New Jersey Commission on Science and Technology and the Rutgers University CAIP Center's Industrial Members.

12. BIBLIOGRAPHY

[1] Chandrasekhar, N., and Franklin, W. R. A fast practical parallel convex hull algorithm. Tech. rep., Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, 1990.

[2] Chazelle, B., and Edelsbrunner, H. An optimal algorithm for intersecting line segments in the plane. In *Foundations of Computer Science - 29th Annual Symposium* (White Plains, NY, Oct 1988).

[3] Franklin, W. R., Chandrasekhar, N., Kankanhalli, M., Akman, V., and Wu, P. Y. Efficient geometric operations for CAD. In *Geometric Modeling for Product Engineering*, M. J. Wozny, J. U. Turner, and K. Preiss, Eds. Elsevier Science Publishers B.V. (North-Holland), 1990, pp. 485–498.

[4] Franklin, W. R., and Kankanhalli, M. Parallel object-space hidden surface removal. Tech. rep., Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, 1990.

[5] Franklin, W. R., Narayanaswami, C., Kankanhalli, M., Sun, D., Zhou, M.-C., and Wu, P. Y. Uniform grids: A technique for intersection detection on serial and parallel machines. In *Proceedings of Auto Carto 9: Ninth International Symposium on Computer-Assisted Cartography* (Baltimore, Maryland, 2-7 April 1989), pp. 100–109.

[6] Preparata, F. P., and Shamos, M. I. *Computational Geometry: An Introduction.* Texts and Monographs in Computer Science Springer-Verlag, 1985.

[7] Ullman, J. D. *Principles of Database and Knowledge-base Systems.* Principles of computer science series, 14. Computer Science Press, 1988.

[8] White, D. A new method of polygon overlay. In *An Advanced Study Symposium on Topological Data Structures for Geographic Information Systems* (Cambridge, MA, USA, 02138, 16-21 October 1977), Laboratory for Computer Graphics and Spatial Analysis, Harvard University.

[9] White, D., Maizel, M., Chan, K., and Corson-Rikert, J. Polygon overlay to support point sample mapping: The national resources inventory. In *Proceedings of Auto Carto 9: Ninth International Symposium on Computer-Assisted Cartography* (Baltimore, Maryland, 2-7 April 1989), pp. 384–390.