Shapefile Overlay Using a Doubly-Connected Edge List

Phil Katz and Stephen St.Vincent
Swarthmore College
500 College Ave.
Swarthmore, PA 19081
[pkatz1, sstvinc2]@swarthmore.edu

Abstract

We present a method for finding the overlays of a set of polygons that uses the doubly-connected edge list structure. We first detect all intersections between polygons using a brute-force method. We then build the doubly-connected edge list, maintaining information about the original polygons, from which we can easily perform shapefile overlay operations: intersection, difference, and union.

Our algorithm runs in $O(n^2)$ time. Our doubly-connected edge list construction algorithm runs in $O(n \log(n))$ time, with the bottleneck being the brute-force $O(n^2)$ line segment intersection. Once that list is built, any given overlay operation is $O(n)$.

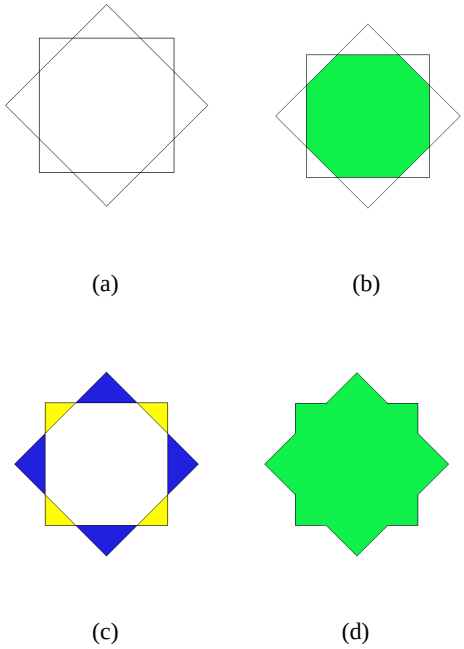

Figure 1: Examples of shapefile overlays. (a) The original polygons in set *S*. Here, we have two overlapping squares at different orientations. (b) The intersection of the two squares, represented by the green region. (c) The difference of the two polygons, shown by the blue and yellow regions. (d) The union of the two polygons. Note that the interior segments are now gone.

1 Introduction

Natural disasters such as floods often occur swiftly and without warning. It is imperative, therefore, that political entities such as counties and states be adequately prepared to deal with such disasters. Often, this level of preparation varies directly with the amount of funding recieved, which is in and of itself a function of the percieved threat in that region. Determining the extent to which region is in danger of flooding is difficult to assess anecdotally. However, by combining geographical data such as watershed layouts with political boundary data, we can use shapefile[1] overlays to assign an unbiased value to the level of danger to any region. This information

[1] A shapefile is a common file format for exchanging polygon data that does not maintain topology

can then assist in appropriate resource allocation, as well as computation of flood insurance rates.

Given a set of polygons *S*, how can one efficiently determine the new set of polygons *P* that is defined by the overlay of the polygons in *S*? Polygons in *P* can include the intersection, union, or difference of members of *S*. Figure 1 shows examples of these types of overlays.

To solve the problem of shapefile overlay, we will use methods from (de Berg et al., 1998) to build a data structure called a *doubly-connected edge list* that will allow us to calculate overlays efficiently. The more general problem of shapefile overlay has special cases which would be unusual in settings such as the one described above, such as polygons with holes in them. Still, we consider these special cases to make our algorithm as general as possible.

In section 3, we present our methods for building the doubly-connected edge list and calculating shapefile overlays. In section 4, we discuss the runtime analysis of our algorithms. In section 5, we present results applying our algorithms to simple test-case polygons as well as to real geographical data. Finally, in section 6, we discuss the implications of our results.

2 Related Work

When building topological representations, it is usually the case that we wish to restrict the topologies to follow a specific set of constraints. (Hoel et al., 1994) describe such a system. Their topologies have certain consistency requirements, and as such must follow an explicit set of rules, such as the following:

- Interiors of polygons in a feature class must not overlap
- Polygons must not have voids within themselves
- Polygons of one feature class must share all of their area with polygons in another feature class

Shapefile overlay is necessary to enforce these rules on large sets of polygons. Without an effective shapefile overlay algorithm, the work of (Hoel et al., 1994) could not be implemented in an efficient, robust manner.

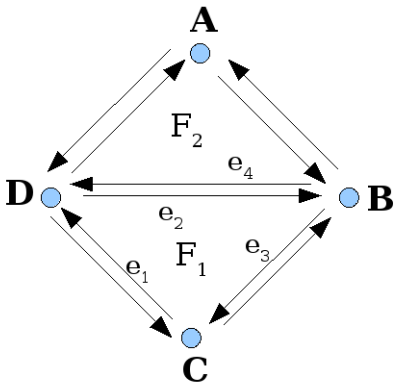

Figure 2: Doubly-connected edge list for two polygons. Half-edge e_1 has a `next` pointer to e_2 and a `previous` pointer to e_3. The current face of all three of these is F_1. The twin of e_2 is e_4, whose current face is F_2. The twins of e_1 and e_3 have `Null` as their current face.

3 Methods

To find the overlays of multiple polygons, we must construct the doubly-connected edge list. Before we can do that, we need to know where the segments of each polygon intersect the segments of the other polygons in the set.

3.1 Line intersection calculations

This is by far the most straightforward step in calculating shapefile overlays. Because the runtime of brute-force algorithms is not significantly slower than more complex algorithms for the data sets we are considering (Andrews et al., 1994), it does not cost us significantly to implement a brute-force algorithm. For each segment of a polygon, we check explicitly whether it intersects any segment of any other polygon, and keep a list of all intersection points that occur on that segment. This allows us to easily create all of the subsegments for the doubly-connected edge list, as well as to keep track of which of the original polygons a segment was associated with.

3.2 The doubly-connected edge list

A doubly-connected edge list (de Berg et al., 1998) stores all of the information regarding the set of polygons that is necessary to calculate the shapefile overlays. The basic setup of a doubly-connected

edge list begins with the edges. The edges of each polygon are stored as directed *half-edges* that go around the polygon in clockwise order such that the face that is bound by the half-edges is always to the right of each half-edge (see figure 2).

Each half-edge stores the following information:

- Starting point
- Ending point
- The ID of the face from which the half-edge originated
- A pointer to the next half-edge on the current face
- A pointer to the previous half-edge on the current face
- A pointer to its twin half-edge

As we build the half-edges that are on the interiors of the original polygons, we can build their *twin edges*. Twin edges are the same as the original half-edge, but with its orientation reversed and its face set to `null`. So, for half-edge *e*, `twin(twin(`*e*`))`=*e*.

For each new half-edge that we add, we update a dictionary, `vDict`, that contains all of the necessary edges. The `vDict` is a hash table keyed on vertices and contains a list of the half-edges that start at that vertex. This is critical for building the next- and previous-edge pointers.

3.2.1 Creating next- and previous-pointers

Determining the next and previous pointers for each half-edge is non-trivial. For the next pointer of each half-edge *e*, we must find the half-edges whose starting point is the same as the ending point of *e*. Then, we must determine which of these half-edges makes the largest clockwise angle with *e*, and set the next-pointer of *e* to that half-edge.

Now that the `vDict` has been updated for each half-edge, this task becomes much easier. Using figure 3 as an example, let us attempt to find the next-pointer of the half-edge $\overline{AX}$. By treating each half-edge as a vector originating at *X* , we can find the angle between the half-edge $\overline{AX}$ and all the other half-edges by using the cross product and the dot product:

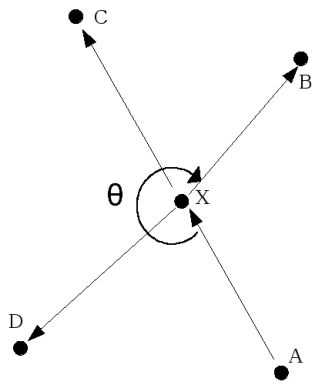

Figure 3: Finding the next-pointer. Here, we are trying to find the appropriate half-edge for the next-pointer of half-edge $\overline{AX}$. θ measures the angles between $\overline{AX}$ and the other half-edges in the image. We choose the half-edge with the largest value of θ, which in this case is $\overline{XB}$.

$$\sin(\theta) = \frac{\left|\left|\overline{AX} \times \overline{BX}\right|\right|}{\left|\left|\overline{AX}\right|\right| \left|\left|\overline{BX}\right|\right|}$$

$$\cos(\theta) = \frac{\overline{AX} \cdot \overline{BX}}{\left|\left|\overline{AX}\right|\right| \left|\left|\overline{BX}\right|\right|}$$

After computing $\cos(\theta)$ and $\sin(\theta)$, we can calculate the true angle θ (where $0 \leq \theta \leq 2\pi$) between $\overline{AX}$ and each of the other three half-edges in the figure. The candidate half-edge with the largest value of θ can now be set as the next-pointer. To complete the example, the next-pointer of $\overline{AX}$ would be $\overline{XB}$, and the previous-pointer of $\overline{XB}$ can be set to $\overline{AX}$. After walking completely around a face in this fashion, all of the next- *and* previous-pointers for the half-edges that bound that face will have been set, so we do not need to calculate the previous pointers explicitly.

3.3 Computing the overlays

Now that we have walked along the half-edges of every face, we have a list for each new face of all of the original faces from which it was derived. From here, we can easily specify an overlay by selecting the new faces that meet the criteria of the overlay in question.

3.4 Non-intersecting overlays

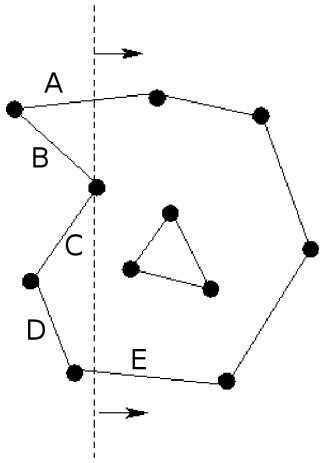

Figure 4: Line-Sweep Algorithm. The dotted line is sweeping rightward. Right before reaching the current point, there should be four edges (A,B,C,E) in the data structure, and upon reaching this point, two (B,C) should be removed.

Not every overlay of two polygons will involve the inteserction of their segments. Consider for instance the overlay of the Sahara Desert with Cumberland County, Pennsylvania. The intersection of these two polygons should be null, given that the two polygons are clearly not spatially coincident and have no intersections.

But we cannot simply say that a lack of segment intersections implies a lack of spatial coincidence. Consider now the overlay of Nebraska with the entire United States. While it is not apparent why one would choose to perform this overlay, it should be clear that this is an example of an overlay that has no segment intersections but does have some overlap.

We solve this problem by using a line-sweep algorithm (de Berg et al., 1998). Figure 4 shows two polygons, with one completely interior to the other. Once our line-intersection algorithm determines that there are no segment intersections between these two polygons, we can move into our line-sweep algorithm. We sort the vertices in order of their x-coordinate; we also store the face associated with the first vertex in the sorted list. We then step through the sorted vertices in order while maintaining a list of edges that currently intersect the sweep line. When a vertex is encountered that has an associated face that differs from the face of the first point, we can stop our line-sweep. If the new vertex is below an odd number of segments, then we know that the new face is completely interior to the other; otherwise, it must be completely exterior. This method is robust for concave polygons.

3.5 Polygons with holes

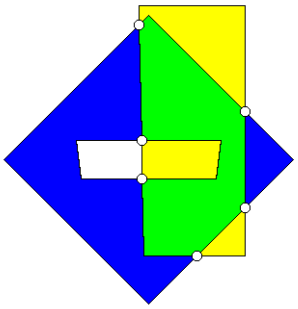

Figure 5: Polygons with holes. The blue polygon has a hole in its center, which is not filled in. Where the yellow polygon intersects with the hole, it remains unchanged.

Consider the nations of South Africa and Lesotho. Lesotho is an independent nation completely interior to the borders of South Africa. As such, South Africa may be treated as a polygon with a hole in it. If we worked for the South African government and needed to determine overlays, we would surely want to take Lesotho into account.

We can handle this by allowing each face to have a pointer to its *inner edges*. These inner edges will form a closed polygon. The outer half-edges of this interior polygon (which run counter-clockwise) have the same face as the original polygon, while the inner half-edges (which run clockwise) have their face set to null. From here, we can perform our normal shapefile overlay process, starting with segment-intersection.

4 Runtime analysis

Our brute-force polygon intersection calculation is $O(n^2)$. For each line segment, we test all of the line segments in the other polygon for intersection. Given that we do not check segments in the same

polygon, this upper bound of $O(n^2)$ can never be reached, but is still the appropriate theoretical upper bound. This assumes that the polygons are simple.

Our line-sweep algorithm is $O(n \log(n))$. For each of the n vertices, the operations we need to perform (search, insertion, and deletion) can be implemented (with a binary tree, for example) to require $O(\log(n))$ operations to maintain the line-sweep data structure.

Once we have the doubly-connected edge list built, we will have k edges, stemming from the n original edges. To build the overlay faces, we need only go through each of the k edges once, removing them from the list of all edges once we assign them to a face. So the overlay algorithm itself (assuming that the doubly-connected edge list has already been built) is $O(k)$. In the worst case, k is $O(n^2)$, but in most real-world polygon intersections, k will be $O(n)$.

5 Results

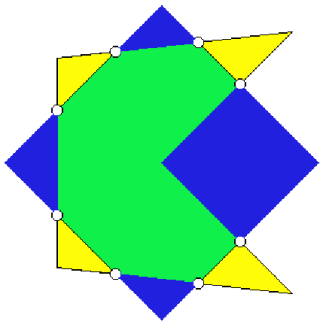

Figure 6: Example of polygon intersection. The blue area indicates areas that are only covered by the convex polygon (rotated square). The yellow areas were only covered by the concave polygon. The green area represents the intersection of the two polygons. The white circles show points of intersection between the segments of the two polygons.

Figure 6 is a sample run of our intersection algorithm on two hand-made polygons. The image was created using the Python graphics library from (Zelle, 2004). The green polygon in the center shows the intersection polygon. The blue and yellow polygons combine to define the difference of the polygons. The entire shaded area is the union of the two polygons. The small white circles show the points of intersection between the two polygons.

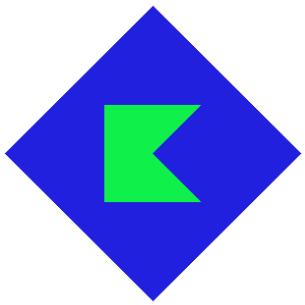

Figure 7: Interior polygons. This image shows the effects of scaling the concave polygon in figure 6 so that it fits completely inside the concave polygon. No area has maintained the yellow coloring from above.

Figure 7 shows the intersection of two polygons where one of the polygons is completely interior to the other. Despite the lack of intersection points, our algorithm has handled this flawlessly.

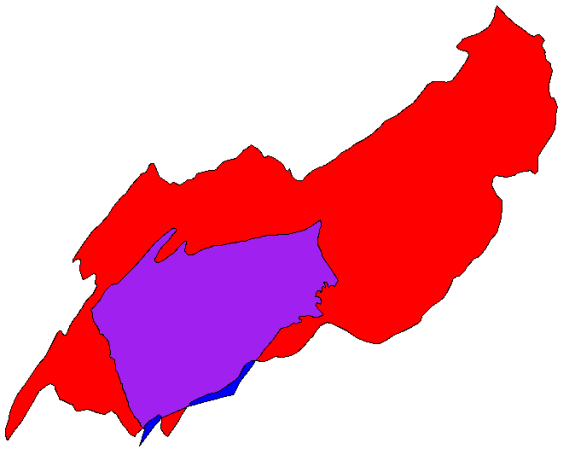

Figure 8: Intersection of Cumberland County with watershed 2050305. The red region is the watershed only, the blue region is Cumberland County only, and the purple region is the intersection of the two.

Figure 8 shows the intersection of Cumberland County, Pennsylvania, with watershed 2050305 (USGS Hydrological Unit Code). Figure 9 shows Cumberland County intersected with watershed 2050306.

Polygons	Time (ms)	Points	Intersect time (ms)	DE list time (ms)	Overlay time (ms)
CC, WS5	8126	693	6132	238	1642
CC, WS6	7950	688	6000	252	1556

Table 1: Benchmarking on sample county and watershed data. The Cumberland County (CC) polygon had 284 points; the watershed 2050305 (WS5) polygon had 409 points; the watershed 2050306 (WS6) polygon had 404 points.

.

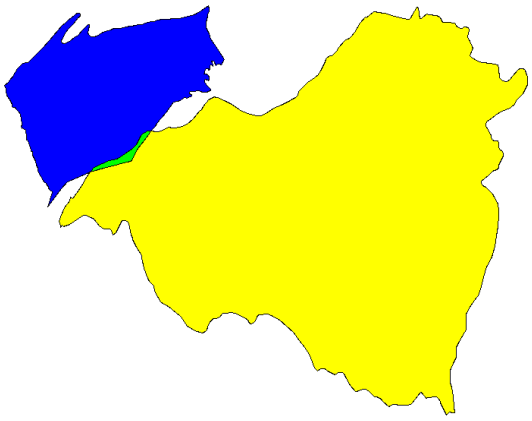

Figure 9: Intersection of Cumberland County with watershed 2050306. The yellow region is the watershed only, the blue regions are Cumberland County only, and the green region is the intersection of the two.

Table 1 shows benchmarking figures for these two runs. The tests were run on an Intel(R) Pentium(R) 4 running at 3.00 GHz with 156 MB of RAM.

6 Discussion

The goal of this project was to implement shapefile overlay in an efficient manner. In order to accomplish this goal, we implemented a doubly-connected edge list that allowed our algorithms to efficiently compute overlays. With the exception of our brute-force line segment intersection, all of our algorithms are relatively computationally inexpensive. Table 1 shows that the line segment intersection is the clear bottleneck. It is possible to implement faster line segment intersection algorithms, but that is not within the scope of this project.

Our algorithm works successfully for most complex cases, including concave polygons, polygons inside other polygons, polygons with holes, and up to three polygons, as in figure 10. Our algorithm can not handle the case of intersecting two polygons that share an edge. This case is pathological for line intersection as well as for constructing a doubly-connected edge list.

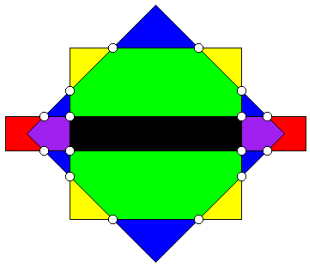

Figure 10: Intersection of Three Polygons. This image shows that our algorithm can be extended to three polygons.

7 Conclusions

Shapefile overlay is a fundamental building block of computational geometry. It is imperative to the field that shapefile overlay can be computed quickly and correctly. Although we do not introduce any revolutionary methods to accomplish this task, we show that it can be performed with fairly straightforward algorithms on simple data structures.

References

D. S. Andrews, J. Snoeyink, J. Boritz, T. Chan, G. Denham, J. Harrison, and C. Zhu. 1994. Further comparison of algorithms for geometric intersection problems. In *6th International Symposium on Spatial Data Handling*.

Mark de Berg, Mark van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 1998. *Computation Geometry: Algorithms and Applications*. Springer.

E. Hoel, S. Menon, and S. Morehouse. 1994. Building a robust relational implementation of topology. In *8th International Symposium on Spatial and Temporal Databases*.

John Zelle. 2004. *Python Programming: An Introduction to Computer Science*. Franklin, Beedle, & Associates.