Intersecting Red and Blue Line Segments in Optimal Time and Precision

Andrea Mantler and Jack Snoeyink

UNC-CH Computer Science
CB 3175 Sitterson Hall
Chapel Hill, NC 27599–3175 USA
{mantler,snoeyink}@cs.unc.edu

Abstract. A common geometric problem in computer graphics and geographic information systems is to compute the arrangement of a set of n segments that can be colored red and blue so that there are no red/red or blue/blue crossings. We give a sweep algorithm that uses the minimum arithmetic precision and runs in optimal $O(n \log n + k)$ time and $O(n)$ space to output an arrangement with k vertices, or $O(n \log n)$ time to determine k. Our initial implementation in Java can be found at `http:\\www.cs.unc.edu\~snoeyink\demos\rbseg`.

1 Introduction

Important cases of the problems of polygon clipping in computer graphics [4], boolean operations in 2D computer-aided design (CAD/CAM) [19, 23], and map overlay in geographic information systems (GIS) [7] can all be abstracted as the problem of building an arrangement of n red and blue line segments in which there are no red/red or blue/blue crossings.

This problem is often attacked by first finding all k intersections of line segments, then building the arrangement, which require sorting intersections along each line if that was not done in the first step. Several algorithms have been developed that approach or achieve the optimal, output-sensitive running time of $O(n \log n + k)$ in the general (uncolored) case for finding intersections [1, 2, 5, 9] and for computing arrangements [3, 6, 7, 10] and in the red/blue case for finding intersections [8, 11, 12, 21].

It is surprisingly difficult to guarantee correct implementations of these algorithms. One reason is a large number of degenerate cases, in which an endpoint of one line segment lies inside another or the intersection of two line segments is a segment rather than a point. A second reason is that classical segment intersection algorithms use primitives with relatively high algebraic degree [6]. In general, if line segments are specified by the coordinates of their endpoints, then computing an arrangement requires four times the input precision, and computing a trapezoidation of the arrangement, as in the Bentley-Ottman sweep [3], requires five times the input precision. Floating point implementations of these algorithms will encounter roundoff error, occasionally resulting in inaccurate comparisons and incorrect results.

In developing a new algorithm, therefore, our primary aims were to minimize arithmetic precision and the effort to handle degeneracies. Note that double precision is the lower limit finding intersections by any means, since testing a pair of segments requires the evaluation of an irreducible quadratic polynomial [6]. In the red/blue case, Chan's trapezoid sweep [8] can compute intersections with only three times input precision. Boissonnat and Snoeyink [5] sketch one way to reduce the precision requirements of Chan's algorithm. In this paper, we describe a more symmetric variant of that algorithm that produces the arrangment, not just the intersections. It handles degeneracies by breaking segments exactly.

Most segment intersection algorithms, whether for the general case or the red/blue case, aim for an output-sensitive running time of $O(n \log n + k)$ to compute k intersections of n segments. The hereditary segment tree techniques [11, 21] can count intersection in $O(n \log n)$ time but do not compute an arrangement. Mairson and Stolfi [18] count and compute a bundled representation of an arrangment, but run in suboptimal $O(n(\log n + \sqrt{k}))$ time. Our algorithm can count intersections and compute a bundled form of the arrangement in optimal $O(n \log n)$ time.

2 Preliminaries

Our algorithm uses a plane sweep, which turns the static 2D problem of line segment intersection into a dynamic 1D problem of maintaining the list of segments that intersect a vertical sweepline as the sweepline moves from left to right across the plane.

We first recall the invariants maintained by Bentley and Ottmann's [3] sweep algorithm, then define witness points which play an key role in the invariants for our algorithm.

2.1 The Bentley-Ottmann sweep

Bentley-Ottmann's classic sweep algorthm [3] maintains two invariants as it moves a sweepline from left to right across the plane: that the y coordinate order of the segments intersecting the sweepline is known, and that all intersections to the left of the sweepline have been reported. Intersections between segments reveal themselves as changes in order, so these invariants must be updated when the sweepline encounters a segment endpoint or intersection point.

Their algorithm maintains two data structures: L, a list ordered by y coordinate of the segments that intersect the sweepline, and Q, a priority queue ordered by x coordinate of all segment endpoints and the intersection points right of the sweepline that are defined by adjacent segments in L. When the sweepline encounters the left endpoint of a line segment, it can search L to determine where to add that segment, then update Q to have only the intersections between adjacent segments in L. For a right endpoint, a segment is removed from L and Q is updated. For an intersection point, the segments involved are swapped in L, and again Q is updated. If L and Q support logarithmic time search, insertion, and deletion, then the algorithm runs in $O((n+k)\log n)$ time.

To maintain the list L, it is sufficient to be able to test whether an endpoint lies above or below a line, which is a degree two polynomial when the segments are specified by their endpoint coordinates. For Q, however, one must compare x coordinates of endpoints and intersection points, which is a degree three test, and x coordinates of interseciton points, which is a degree five test.

2.2 Witnesses to intersection

By limiting our computation to double precision, we lose the ability to compare x coordinates of intersections to either intersections or endpoints. We can only compare x coordinates of endpoints, and test whether an endpoint is above or below a given segment.

In fact, we can deform the input line segments, as long as they remain monotone and do not pass over any endpoints. The tests we use cannot distinguish a difference [5, 6]. Figure 1 illustrates that deformation can move the intersection point of two segments, $a \cap b$, so our algorithms cannot rely on ordering intersections by x coordinates. Figure 1 also illustrates, however, that if we deform segments a and b to push their intersection point as far to the right as possible, this intersection point must stop before it reaches the leftmost point in a wedge between the two segments. We call this point (p in the figure) the *witness* for the intersection of a and b, because it is the first point that certifies that these segments have swapped from their initial order. Note that the right endpoints of segments will serve as witnesses if there are no earlier points in the wedge, so every intersection between a pair of segments has exactly one witness point.

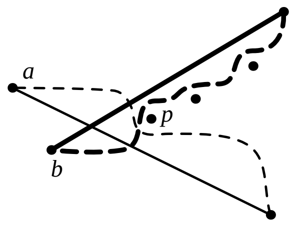

Fig. 1. Point p witnesses $a \cap b$

Under different terminology, witnesses also play an important role in Boissonnat and Preparata's lazy sweep [6]. In their paper, a pair of segments is called *prime* for p, when p is the witness of their intersection.

2.3 From the general to the red/blue case

In the general case, we cannot hope to build an arrangement in double precision. The main reason is that quadruple precision is needed to determine the orientation of a triangle formed by three segments. Also, for curve segment intersection, there is a family of lower bound examples [5] with n segments and k total intersections that require $\Theta(n\sqrt{k})$ operations to count all intersections, where the hidden constant is independent of n and k.

The special case of red/blue segment intersection, in which the segments can be colored red and blue with no red/red or blue/blue crossing pairs, we do have hope. Without red/red crossings, for example, any deformation of segments may change the locations where a single blue segment intersects the reds, but will not change the intersection order. We exploit this in the next section.

3 Optimal red/blue segment intersection

We turn now to the special case of red/blue segment intersection, where we can compute the arrangement using only double precision tests. We describe the invariants, events, data structure, and processing for an algorithm to count the intersections or report the arrangment explicitly. The analysis of running time is simple. We also comment on computing a compact representation of the arrangement, and on the handling of degeneracies.

3.1 Invariant and events

Our algorithm performs a sweep, maintaining the invariants that all intersections whose witnesses are left of the sweepline have been reported, and that the order of segments along the sweepline is consistent with pushing all intersections as far right as possible.

The events during this sweep occur only at the endpoints of segments. This gives the potential for a practical advantage over a Bentley-Ottmann sweep: we can pre-sort the events and avoid a dynamic priority queue. On the other hand, our structure containing the segments ordered along the sweepline becomes correspondingly more complex, since it must use an order consistent with a deformation, and not an order given by actual segment positions.

3.2 Data structure and processing

The data structure that stores the ordered list of segments crossing the sweepline consists of three pieces, which are illustrated schematically in Figure 2.

First, we group segments into red and blue *bundles*, which are the maximum consecutive sequences of segments of the same color. Notice that since there are no crossings between segments of the same color, segments within a bundle will remain ordered even if they are deformed. Each bundle stores the segments in a small balanced tree structure, and keeps pointers to the top-most and bottom-most. Each bundle must support insert, delete, split, and merge operations.

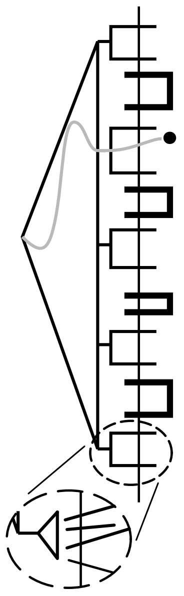
Fig. 2. Bundles

Second, we place all bundles into a doubly-linked list in order. Bundles will alternate colors.

Third, we organize red bundles into a *bundle tree*, a balanced search tree supporting split and merge operations.

We now describe the event processing when the sweepline reaches endpoint p. We include the handling of degenerate cases, such as segments sharing or containing endpoints.

First, we locate p among the red bundles using the bundle tree. We will find either that p is between two red bundles, inside one red bundle, or inside at most two red bundles with any intermediate red bundles consisting of segments that all end at p. If we search and split the at most two bundles that contain point p, then each red bundle is either 'above,' 'ending,' or 'below' at p.

Next, we locate p among the blue bundles using the linked list, again splitting at most two blue bundles so that each blue bundle is either 'above,' 'ending,' or 'below' at p. Although this is a linear search, the bundles involved will be merged in the next step, so we can charge the search cost to the merging.

Once we know all bundle positions relative to p, we can look for adjacent bundles, one red and one blue, that are in the wrong positions. These bundles have intersections witnessed by p. For example, if a red 'above' bundle R is below a blue 'ending' or 'below' bundle B, then every pair in $R \times B$ has an intersection that is witnessed by p. We can report these intersections, then move R up and past B by merging R with the red bundle immediately above, merging B with the blue bundle below, and repairing the bundle tree and linked list to reflect the actions to bundles R and B.

Finally, after all bundles are in the correct relative positions, we remove those ending at point p, and, if needed, form new bundles for segments starting at p. If p lies exactly on a segment, then we break that segment into two, so that one ends and one starts at p. This completes the processing at p.

3.3 Analysis

We claim that, if the data structures are implemented correctly, then the algorithm correctly maintains the invariants, and therefore correctly computes all intersections.

The analysis of running time is not difficult. Each endpoint causes a constant number of tree searches and splits, each of which can be carried out in $O(\log n)$ time. Because each merge joins two trees that were created by a segment insertion or a split, the total number of merges is also $O(n)$. List searches can be charged to bundle merges, as can the counting of intersections. Therefore, all operations take a total of $O(n \log n)$ time. The reporting of intersections can be charged to output complexity, giving a total of $O(n \log n + k)$ time if the arrangement is reported explicitly.

In our implementation, we use splay trees [22] for the bundles and bundle trees. It is not hard to extend an amortized analysis for splay trees [15] to the merges and splits used by our algorithm.

3.4 Compact representation of the arrangement

By using an idea from persistent data structures [13] we can record intersections in bundles using $O(n \log n)$ time and space, even if there are quadratically many intersections. Each time we are to split or merge a bundle, we copy the root-to-leaf path that is to change, and make our modifications to the copy. This preserves access to the old bundle, and the intersections can be recorded by recording at each endpoint p which bundles intersections were witnessed by p.

Mount developed a data structure for storing a planar subdivision of size n on a polyhedron of size n in $O(n \log n)$ space, while still supporting point location [20]. He decomposes the arrangement of their overlay into grids where bundles of subdivision and polyhedron edges all cross. He notes that it would be interesting to compute this bundle structure directly to avoid a quadratic worst case in applications such as computing the Voronoi diagram on the surface of a polyhedron. Although our computation does give a bundle structure directly, we cannot use it for point location as Mount does. Our structure represents a deformation of the arrangement, and if a new query point is considered, then the deformation may need to change to accomodate it.

3.5 Degeneracies

We close with one observation on how we have handled degenerate cases. In an intersection algorithm, new points are calculated along segments. The implementer is faced with a choice of whether to break line segments at these new intersection points, or to hold the original segments as sacred and use the new points primarily for display. The decision usually depends on whether the implementer trusts the floating point arithmetic, or has been burned in the past.

Our approach is to break segments, but only when the endpoints are representable in single precision. Specifically, if an endpoint lies on a segment, which we can test exactly in double precision, then we can break the segment. Thus, the only remaining degeneracy is the case of overlapping segments with same start and end points.

4 Conclusion

We have given an algorithm building an arrangement of red/blue line segments that is optimal in not only time and space but also in the algebraic degree of its predicates. This makes it possible to guarantee correct results using only double the precision of the input coordinates. Our initial Java implementation, in Figure 3, can be found at `http:\\www.cs.unc.edu\~snoeyink\demos\rbseg`.

Intersections in the resulting arrangement are represented by pointers to their segments, as in [7]. Representing their coordinates requires rational numbers with numerators of degree three and denominators of degree 2. In future work, we will apply snap rounding ideas [17, 14, 16] to consistently round the output back to single precision.

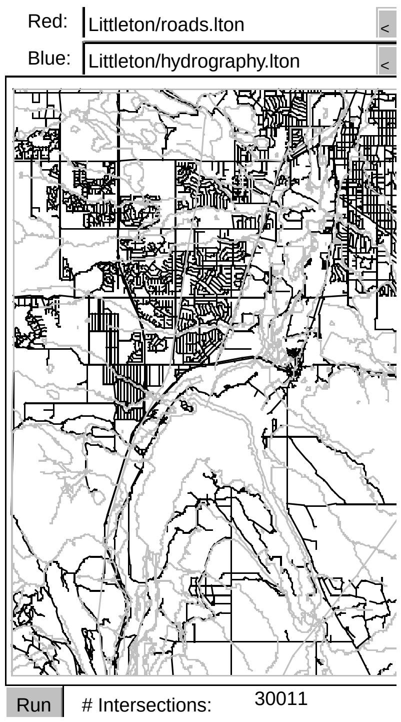

Fig. 3. Intersection of roads and hydrography data in our Java applet

Acknowledgements

We thank Emo Welzl for asking whether intersections could be counted in batches, and thank Sergei Bespamyatnikh for discussions on this algorithm. We gratefully acknowledge the support of NSF Grant 9988742.

References

1. Ivan J. Balaban. An optimal algorithm for finding segment intersections. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 211–219, 1995.
2. U. Bartuschka, K. Mehlhorn, and S. Näher. A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem. In *Proc. Workshop on Algorithm Engineering*, pages 124–135, 1997.
3. J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, September 1979.
4. J. F. Blinn. A trip down the graphics pipeline: Line clipping. *IEEE Comput. Graph. Appl.*, 11(1):98–105, 1991.
5. J.-D. Boissonnat and J. Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 370–379, 1999.

6. Jean-Daniel Boissonnat and Franco P. Preparata. Robust plane sweep for intersecting segments. *SIAM J. Comp.*, 29(5):1401–1421, 2000.
7. Andreas Brinkmann and Klaus Hinrichs. Implementing exact line segment intersection in map overlay. In *Proc. 8th Intl. Symp. Spatial Data Handling*, pages 569–579. International Geographical Union, 1998.
8. T. M. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Proc. 6th Canad. Conf. Comput. Geom.*, pages 263–268, 1994.
9. Bernard Chazelle. Reporting and counting segment intersections. *J. Comput. Syst. Sci.*, 32:156–182, 1986.
10. Bernard Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39(1):1–54, 1992.
11. Bernard Chazelle, H. Edelsbrunner, Leonidas J. Guibas, and Micha Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. *Algorithmica*, 11:116–132, 1994.
12. Olivier Devillers and Andreas Fabri. Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers. *Internat. J. Comput. Geom. Appl.*, 6:487–506, 1996.
13. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, 1989.
14. M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.
15. Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java.* John Wiley & Sons, New York, NY, 1998.
16. Leonidas Guibas and David Marimont. Rounding arrangements dynamically. *Internat. J. Comput. Geom. Appl.*, 8:157–176, 1998.
17. J. D. Hobby. Practical segment intersection with finite precision output. *Comput. Geom. Theory Appl.*, 13(4):199–214, October 1999.
18. H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume 40 of *NATO ASI Series F*, pages 307–325. Springer-Verlag, Berlin, West Germany, 1988.
19. Victor J. Milenkovic. Practical methods for set operations on polygons using exact arithmetic. In *Proc. 7th Canad. Conf. Comput. Geom.*, pages 55–60, 1995.
20. D. M. Mount. Storing the subdivision of a polyhedral surface. *Discrete Comput. Geom.*, 2:153–174, 1987.
21. L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. *CVGIP: Graph. Models Image Process.*, 56(4):304–311, 1994.
22. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–381, 1983.
23. R. B. Tilove and A. A. G. Requicha. Closure of boolean operations on geometric entities. *Comput. Aided Design*, 12:219–220, 1980.