Practical Segment Intersection with Finite Precision Output

John D. Hobby

Bell Laboratories
700 Mountain Ave.
Murray Hill, NJ 07974

The fundamental problem of finding all intersections among a set of line segments in the plane has numerous important applications. Reliable implementations need to cope with degenerate input and limited precision. Representing intersection points with fixed precision can introduce extraneous intersections. This paper presents simple solutions to these problems and shows that they impose only a very modest performance penalty. Test data came from a data compression problem involving a map database.

Key Words: line segment intersection; degeneracy; data compression

1 Introduction

The problem of finding all intersections among a set of line segments in the plane is fundamental to computational geometry and essential in various applications such as hidden line elimination [9], clipping and windowing [7], and physical simulations [11]. Other possible applications include computer vision [12], circuit design [5], constructive solid geometry [24], and various computational geometry problems [1, 21, 27].

Unfortunately, the standard algorithms do not always work properly in practice because they were designed for exact real arithmetic. It is possible to run the standard algorithms in exact arithmetic, but the resulting intersection points require additional precision and rational arithmetic. Since this is very unappealing for applications that require signficant computations involving the intersection points, practitioners are likely to insist on using approximate values for these points. A major difficulty is that this can introduce "extraneous intersections" as shown in Figure 1. (Since the intersection process is scale-invariant, it is convenient to use an integer grid.)

The original sweep line algorithm by Bentley and Ottmann [2] finds intersections among n segments in time $O((n+k)\log n)$ time, where k is the number of intersections. Brown's modification [3] reduces the space requirement to $O(n+k)$. The saving is fairly modest because Pach and Sharir show that the original algorithm uses $O(\min(n+k, n\log^2 n))$ space [22]. Chazelle and Edelsbrunner's optimal $O(n\log n + k)$ algorithm [4] is complicated enough to be unattractive in practice. The randomized algorithms by Clarkson [6] and Mulmuley [19, 20] are simpler, but [2] is still the algorithm to beat in practice.

Practical segment intersection algorithms need to handle degeneracies such as segments parallel to the sweep line or three segments crossing at the same point. Care must be taken that the basic operations are accurate enough to avoid topological inconsistencies due to rounding error. A

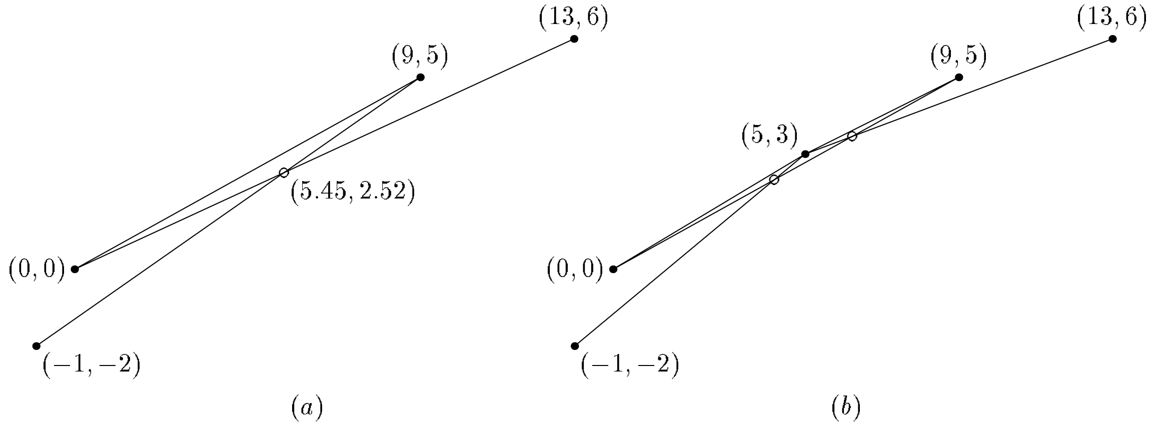

Figure 1: (a) Original input; (b) the result of rounding the intersection to integer coordinates. The extraneous intersections in (b) are marked by circles.

careful implementation of the Bentley-Ottmann algorithm can surmount these problems, but they are harder to deal with for a complicated algorithm such as Chazelle and Edelsbrunner's.

Perhaps the most interesting difficulty is the danger of extraneous intersections as illustrated in Figure 1. A naive approach is to run the basic segment intersection algorithm on its own output and repeat until no new intersection points are found. One iteration on the input in Figure 1a produces Figure 1b; the next iteration produces Figure 2; and the third iteration leaves Figure 2 unchanged. The approach works reasonably well in common cases where the iteration count is very small, but there is no guarantee it will be small.

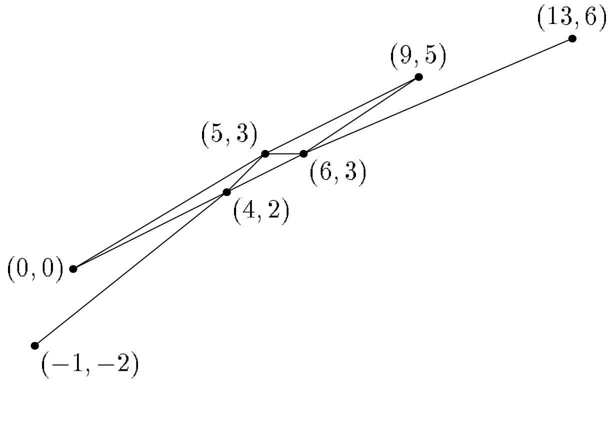

Figure 2: The result of finding the extraneous intersection points in Figure 1b and rounding to integer coordinates.

More reliable solutions can be found in the literature. None of them keep all of the segments perfectly straight, and Milenkovic and Nackman have shown that it would be impractical to do so [18]. Greene and Yao propose using short line segments called "hooks" to keep track of what segments need to be bent [10]. They insert enough extra vertices to ensure that no grid points lie strictly between a segment and its adjusted version. The number of extra vertices needed depends on the length of the segment.

Milenkovic has proposed algorithms that avoid this dependence on segment length. One involves rounding the intersection points and replacing the segments with polygonal lines as determined by a shortest-path condition [15, 14]. The intermediate points come from the other segment endpoints and rounded intersection points. There is also a generalized version that uses more points [16]. Milenkovic has extended his ideas to solve the difficult problem of performing a sequence of geometric operations on polygonal regions in the plane reliably and with limited precision [17].

Sugihara has a completely different segment intersection algorithm that starts with the Delaunay triangulation and uses incremental updates [23]. It copes with degeneracies and extraneous intersections, but its running time has a large term that depends on how close segments can get without intersecting.

Existing techniques for avoiding extraneous intersections are often not used in practice because of a perception that they involve a lot of complicated machinery. Milenkovic's shortest path technique is probably quite practical, but it is underutilized because his papers give very little detail and they contain a lot of other material that may discourage practitioners. Section 2 presents a conceptually simple alternative that performs well in practice and has not appeared in print. We also discuss handling degeneracies in the Bentley-Ottmann sweep algorithm since this is an essential part of practical segment intersection.

Section 3 shows how the algorithm performs on test data derived from a data compression application. The proof that the algorithm avoids extraneous intersections is delayed until Section 4. Finally, Section 5 gives some concluding remarks.

2 The algorithm

The purpose of the algorithm is take a set of line segments, find all intersection points, and insert them into the appropriate segments. The intersection points are to be rounded to some fixed grid and everything has been scaled so that grid points have integer coordinates. It is also convenient to assume that segment endpoints have been rounded to grid points. The set of points that rounds to a grid point (i, j) is $(i, j) + R$, where

$$R = \left\{ (x, y) \mid -\frac{1}{2} \le x < \frac{1}{2},\ -\frac{1}{2} \le y < \frac{1}{2} \right\},$$

and $+$ denotes the Minkowski sum.[1] Region $(i, j) + R$ is the *tolerance square* for point (i, j).

The first step is to use the Bentley-Ottmann sweep line algorithm to find the intersection points. Let T be the set of all segment endpoints and intersection points, and compute the set $\lceil T \rfloor$ by rounding each point in T to the nearest grid point; i.e., $\lceil T \rfloor$ contains the points (i, j) such that $(i, j) + R$ intersects T. Each time a segment s intersects tolerance square $P + R$ for some $P \in \lceil T \rfloor$, alter s by bending it so it passes through P. Call this operation *inserting P into segment s*. This

[1] $S_1 + S_s$ is the set of sums $P_1 + P_2$ for $P_1 \in S_1$ and $P_2 \in S_2$, and $(i, j) + R$ is short for $\{(i, j)\} + R$.

avoids extraneous intersections by taking segments that pass dangerously close to $\lceil T \rfloor$ points and bending them so they meet the points in $\lceil T \rfloor$ as shown in Figure 3. The proof that this works will be given in Section 4.

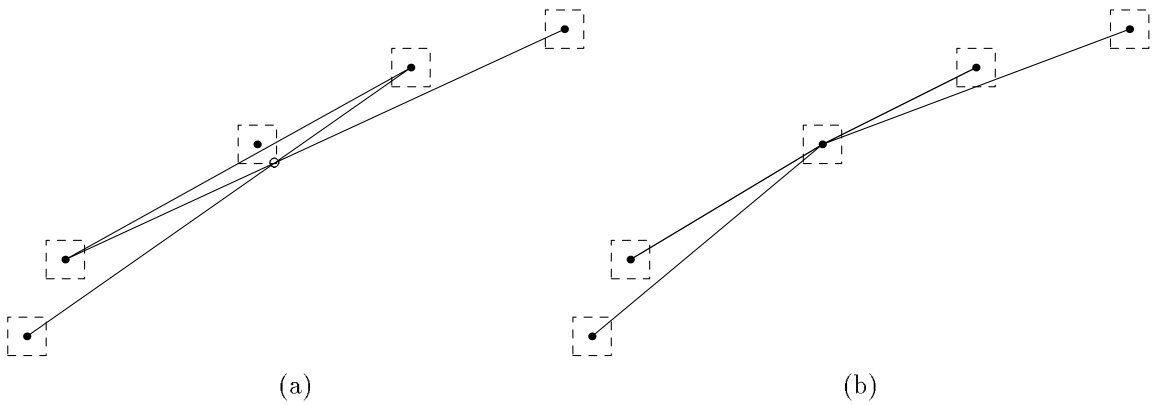

Figure 3: (a) Sample input with points in $\lceil T \rfloor$ marked by solid dots and the tolerance squares outlined by dashed lines; (b) the result of inserting $\lceil T \rfloor$ points to avoid extraneous intersections.

2.1 Intersecting Segments with Tolerance Squares

Let $\lceil x \rfloor = \lfloor x + \frac{1}{2} \rfloor$ and $\lceil (x, y) \rfloor = (\lceil x \rfloor, \lceil y \rfloor)$ for all x, y. It is tempting to think that intersections with a tolerance square $\lceil P \rfloor + R$ could be found by examining the data structures for the Bentley-Ottmann sweep algorithm when it encounters P. Say the sweep line is vertical and moves left-to-right. The problem is that the segment in question could end at $x = \lceil P_x \rfloor$, while the x coordinate P_x of P could be almost $\frac{1}{2}$ unit larger than this. Thus the Bentley-Ottmann sweep needs to stay at least $\frac{1}{2}$ units ahead. Call the Bentley-Ottmann sweep "Pass 1" and whatever follows "Pass 2".

One way to proceed would be to treat Pass 2 as a separate problem. Add the four segments that make up the boundary of each tolerance square and use a segment intersection algorithm to find all intersections between the original segments and these new segments. Pass 2 could be done with the Bentley-Ottmann algorithm or with an S–T intersection algorithm [1, 13].

How expensive is this? If there are n original segments and k intersections among them, there are up to $2n + k$ tolerance squares and up to $8n + 4k$ new segments are needed. The number could be less if we know in advance that some segments share endpoints, but even $4n + 4k$ new segments would make the second intersection-finding step much slower than the first. Of course this is a very naive estimate, but anything like a 400% overhead for Pass 2 is unacceptable in practice.

The best way to reduce the overhead is to use as much information as possible from Pass 1 and take advantage of the special properties of the tolerance squares. Although it may be possible to do this for almost any intersection algorithm, the following discussion assumes that Pass 1 is a Bentley-Ottmann sweep. The vertical edges of the tolerance squares need not be represented explicitly, and the horizontal edges come in batches that start and end together. Thus we define *Batch* i to contain all tolerance square edges that begin at $x = i - \frac{1}{2}$ and end at $x = i + \frac{1}{2}$.

Figure 4a illustrates how segment starting, ending, and crossing events give rise to tolerance squares. Pass 2 uses a sweep line algorithm where tolerance square edges (horizontal dashed lines in Figure 4b) are kept in a separate list, the *tolerance edge list*. Thus there are two lists of current segments: the *main active list* contains original segments exactly as in Pass 1; and the tolerance edge list contains only tolerance square edges. Some segments on the main list might start or end at $x = i$, while all edges in the tolerance edge list start at $x = i - \frac{1}{2}$ and end at $x = i + \frac{1}{2}$. No segments start or end on $i - \frac{1}{2} < x < i$ or on $i < x < i + \frac{1}{2}$, so the vertical slabs corresponding to these x values are called *hammocks*.

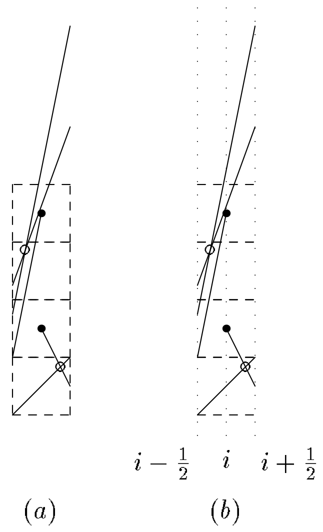

Figure 4: (a) A batch of tolerance squares and the segments whose starting, ending, and crossing events are responsible; (b) the "hammocks" between special x coordinates (marked by dotted lines). Tolerance squares are outlined by dashed lines and crossing events are marked by open circles.

Pass 2 processes batches one-at-a-time in left-to-right order so that the algorithm for Batch i can assume Batch $i - 1$ has been processed successfully. It operates by resynchronizing the main active list and the tolerance edge list when the sweep line reaches the end of a hammock. This happens in Steps 4 and 6 below. Here is the complete algorithm for processing Batch i, starting with the main active list set up for $x = i - \frac{1}{2} - \epsilon$ for some infinitesimal positive ϵ:

Algorithm 1 is essentially the same as applying the Bentley-Ottmann algorithm to the original segments and the tolerance square edges, except that the new edges are stored in a separate list and crossings between original segments and tolerance square edges are delayed until the end of a hammock. The delay avoids the need to insert into the event queue, and the separating the tolerance edge list makes the main active list easier to maintain. Key information such as where in the main active list a new segment should be inserted can be saved from Pass 1 so that the main active list can be a doubly linked list and insertions, deletions, and interchanges can be performed in constant time.

Step 4 requires maintaining a relative ordering between the tolerance edge list and the main active list so that we can compare each tolerance square edge with the segments above and below

Algorithm 1 The Pass 2 algorithm

1. Collect events E from Pass 1 where the x coordinate of the starting, stopping, or intersection point rounds to i.
2. Create horizontal edges for the top and bottom of the tolerance squares for events E and sort them by y values. Then locate the y values in the main active list. For each j where there are top and bottom edges at $y = j \pm \frac{1}{2}$ and the main active list has segments between $y = j - \frac{1}{2}$ and $y = j + \frac{1}{2}$, insert (i, j) into each such segment.
3. Update the main active list so it is valid for $x = i - \epsilon$.
4. Relocate the y values for the horizontal tolerance edges in the main active list using $x = i$, and deduce which segments must have crossed through or into tolerance squares. For each such crossing, insert the point at the center of the tolerance square into the segment.
5. Update the main active list so it is valid for $x = i + \frac{1}{2} - \epsilon$. When encountering a vertical segment, immediately walk through the tolerance edge list and insert vertices on the vertical segment for the squares it passed though.
6. Repeat Step 4 with $x = i + \frac{1}{2}$.

it and do as many interchanges as necessary to achieve a consistent ordering. A new vertex gets inserted each time a segment crosses below the top edge of a tolerance square or above the bottom edge.

Steps 3 and 5 involve inserting, deleting, and interchanging segments on the main active list to reflect the events from Pass 1. What if a pair of segments to be interchanged have tolerance square edges between them? If Segment a is about to cross below Segment b as shown in Figure 5, we can scan the tolerance edge list and find which tolerance squares should be below a at the end of the current hammock, i.e., at $x = i$ if we are in Step 3 and at $x = i + \frac{1}{2}$ if we are in Step 5. Move these tolerance square edges below b and move the others above a. Edges of the former type are e_1 and e_2 in Figure 5; b has crossed above these edges so those that are bottom edges of tolerance squares cause vertices to be inserted in b. Similarly, edges such as e_3 and e_4 that a has crossed below cause insertions in a if they are top edges of tolerance squares.

A few other implementation details are worth mentioning:

- Each active segment should have a pointer to the tolerance square edge immediately above and such edges should have pointers back to the segment below. The pointers should be null when a segment's upper neighbor is another segment or a tolerance square edge has another edge for a lower neighbor. It simplifies the program to zero out the segment-to-edge pointers at the start of Step 4 and regenerate them afterward.
- When a segment start event is encountered in Step 2, it is necessary to store a pointer to one edge of the corresponding tolerance square so that Step 5 can locate the new segment in the tolerance edge list.

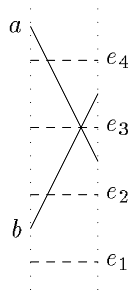

Figure 5: Segments a and b have tolerance square edges e_1 and e_2 caught between cross when they cross. The vertical dotted lines mark the limits of a hammock.

- When Step 2 locates new tolerance square edges in the main active list, it should scan up or down from a *reference segment* derived from the responsible event. Any segment that crosses $x = i - \frac{1}{2}$ will work, but efficiency dictates a careful choice. Possibilities include the segment involved in an ending event, the segment above which the new segment is to be inserted for a starting event, the segments involved in a crossing event, or (if both of these start at $x = i$) the segments above which they are to be inserted. Another option is to start wherever the next lower tolerance square edge wound up.
- Positively sloped segments should receive new vertices in order of ascending y, and the order for downward segments should be descending y. This means Step 4 should go up the tolerance edge list looking for segments that cross below a tolerance square edge, and then it should go down the list looking for crossings of the opposite type. A similar discipline is needed when removing tolerance square edges from between segments that are about to cross.

Theorem 2.1 *Algorithm 1 can be made to run in time $O(N_i \log N_i + k_i')$, where N_i is the number of segment starting, ending, and crossing events collected in Step 1, and k_i' is the number of vertices inserted. Without the sorting in Step 2, the time would be $O(N_i + k_i')$.*

Proof. Each event collected in Step 1 is used only two other times: once to create a pair of tolerance square edges in Step 2; and once to update the main active list in Step 3 or 5. Each update takes constant time except for crossing events where there are tolerance square edges to be moved. This movement and the similar processing in Steps 4 and 6 can be charged against the vertices inserted.

All that remains is to show that locating the tolerance square edges in the main active list in Step 2 takes $O(N_i)$ time. This depends on the choice of reference segments. Always choosing the final position of the next lower tolerance square edge would mean scanning the main active list without backtracking. This is not quite good enough because the main active list could have many segments that do not start, stop, or cross anything on $i - \frac{1}{2} \leq x < i + \frac{1}{2}$.

Suppose the last reference segment was r_j, the corresponding final position was s_j, and the next higher edge belongs just below s_{j+1} and came from event e. The next reference segment r_{j+1} should be s_j or one of the segments involved in event e. If such a segment crosses $x = i - \frac{1}{2}$ above r_j and s_j then that segment should be r_{j+1}, otherwise set $r_{j+1} = s_j$. This guarantees no segment is passed more than twice when Step 2 scans the main active list to locate new tolerance square edges. Furthermore, segments not involved in events on $i - \frac{1}{2} \le x < i + \frac{1}{2}$ are not passed at all, since they cannot be between r_{j+1} and s_{j+1}. Thus the total time for this part of Step 2 is $O(N_i)$ as required. □

2.2 Handling Degeneracies

A practical implementation of the Bentley-Ottmann algorithm and Algorithm 1 must work when segments can be parallel to the sweep line, three or more segments can share a common intersection, and segments can intersect at their endpoints. Most of the basic ideas needed to solve these problems are known, but they are not treated adequately in the literature. Edelsbrunner and Mücke's simulation of simplicity is relevant but does not immediately yield an attractive solution [8].

Degeneracies have little effect on Algorithm 1 as long as it inherits a valid sequence of events from the Bentley-Ottmann sweep. Since tolerance squares contain their bottom edges but not their top edges, a segment that hits the sweepline at some $y = y_0$ should be treated as above any tolerance edge whose y coordinate is y_0. The algorithm needs no other changes in order to handle borderline cases of segments intersecting tolerance squares.

The Bentley-Ottmann sweep can handle degeneracies by judicious use of three key ideas: add an infinitesimal tilt to the sweepline; shorten each segment a doubly infinitesimal amount by trimming off both ends; and ignore confusion about the relative order of crossing events. The infinitesimal shortening rule prevents Pass 1 from reporting intersections at segment endpoints. This is harmless because Pass 2 finds tolerance square intersections in such cases.

What about confusion about the relative order among crossing events? This is only a problem for coincident crossings, but the key idea is best illustrated with a non-degenerate situation such as Figure 6. If crossings should be in the order *a*-*b*, *a*-*c*, *b*-*c*, but *b*-*c* is erroneously scheduled before *a*-*c*, the algorithm can just ignore the erroneous crossing when it detects that *b* and *c* are not adjacent in the sweepline data structure. After processing the *a*-*c* crossing, *b* and *c* become adjacent and the crossing is rescheduled.

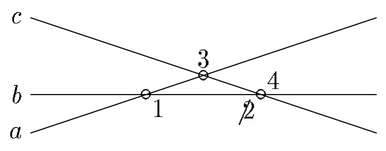

Figure 6: Segments a, b, and c with their crossings labeled according to the order in which they might be processed.

Theorem 2.2 *If the Bentley-Ottmann algorithm is modified to ignore crossing events where the segments involved are not adjacent, the event queue need not maintain the relative order of crossing events. The algorithm still finds all of the intersections and runs in time* $O((n+k)\log n)$.

Proof. Assume the sweepline moves left to right. Whenever a segment a is immediately below a segment b in the sweepline structure and a has slope greater than b, the algorithm ensures there is an event in the queue for the crossing of a and b. When the sweepline hits a starting or ending event at some x value x_0, the queue contains no crossings with x values less than x_0. Thus the sweepline structure must be in order at such times and all the correct crossings must occur.

Since crossings are scheduled only when segments actually start, end, or cross, the total number scheduled is at most $2(2n+k)$, for n segments and k actual crossings. Thus scheduling some crossings that ultimately get ignored does not increase the asymptotic running time. $\square$

What does all this mean in terms of the primitive operations that support the Bentley-Ottmann sweep? Suppose the sweepline is almost vertical and moves almost left-to-right; i.e., it has slope $-1/\epsilon$ and moves in the $(1, \epsilon)$ direction for some infinitesimal positive ϵ. The following geometrical primitives suffice:

1. Find the point where two segments intersect.
2. Decide if an intersection point is to the right of the sweepline and to the left of the endpoints of the segments involved.
3. Decide which of two events the sweepline hits first.
4. Decide whether a segment starting point is above or below an existing segment.

Consider the primitives in order. Intersection points must be computed accurately enough to ensure correct results when comparing x or y coordinates with segment endpoints. Suppose the coordinates of segment endpoints are integers of magnitude at most some constant L and segments span at most some other constant L_Δ in x and y. Then intersection points have rational coordinates with denominators less than $2L_\Delta^2$, and floating point with a relative accuracy of one part in

$$2LL_\Delta^2$$

is sufficient to produce results that compare correctly with integers.[2] In fact, Algorithm 1 needs to correct comparisons with numbers of the form $i + \frac{1}{2}$, so the true requirement is one part in $4LL_\Delta^2$. Intersection points need not compare correctly with each other because of Theorem 2.2.

An important tool for implementing the other primitives is slope comparison. If the direction vectors for segments a and b are $(A_{\Delta x}, A_{\Delta y})$ and $(B_{\Delta x}, B_{\Delta y})$, it suffices to test the sign of

$$A_{\Delta y} B_{\Delta x} - A_{\Delta x} B_{\Delta y}. \tag{1}$$

This requires numbers of size $2L_\Delta^2$ and works even if a and/or b is vertical. Vertical segments have $\Delta y > 0$ and other segments have $\Delta x > 0$ so that (1) treats vertical segments as having slope $+\infty$.

The second primitive is for deciding whether to schedule a crossing event when two segments become adjacent on the sweep line. The idea is to reject the crossing as behind the sweepline if the lower segment does not have a greater slope. Otherwise, we can safely compute an intersection point and then make sure it is behind the endpoints of both segments; i.e., we compare the intersection

[2]Precision on the order of L^2 suffices if the input segments are suitably adjusted and the endpoints are not required to remain on the integer grid [14].

$(\bar{x}, \bar{y})$ lexicographically with each segment endpoint (x_j, y_j) and make sure $\bar{x} < x_j$ or $\bar{x} = x_j$ and $\bar{y} < y_j$.

The third primitive is a simple lexicographic comparison between two points. For segment starting or ending events, the point involved is a segment endpoint; otherwise, it is an intersection point where two segments cross. In case of a tie, ending events come first, then crossing events, then starting events. (This is a consequence of the infinitesimal shortening rule.)

The last primitive involves comparing an integer y value with the y intercept of an active segment on the sweepline. When the segment is not vertical, the difference in y values is a rational number of magnitude at most $2L$ and a denominator at most L_Δ. Thus a relative accuracy of one part in $2LL_\Delta$ suffices for evaluating the sign of the numerator. If the segment is vertical, the comparison should just use the y value of the upper endpoint, since a segment whose starting point lies on a vertical segment is considered to be below due to the infinitesimal shortening rule and the infinitesimal tilt of the sweepline. Because the former rule forces the sweepline to advance an infinitesimal amount, slope comparison should be used to break ties between the y values of the existing segment and the new segment's starting point.

2.3 Putting It All Together

The basic idea is very simple: the Bentley-Ottmann sweep collects starting, ending, and crossing events in the order they are actually performed; then they are passed to Algorithm 1, one batch at a time. Any segment intersection algorithm could be substituted for Bentley-Ottmann, but then it would be less clear how to handle degeneracies and how to find tolerance square intersections efficiently.

Since Algorithm 1 uses the events to maintain the segment order on the sweepline, the Bentley-Ottmann sweep should pass along crossings events only when it actually swaps segments. Then Algorithm 1 finds tolerance square intersections and inserts the corresponding vertices. All the tricky geometric primitives and tie-breaking rules are part of the initial Bentley-Ottmann sweep.

The Bentley-Ottmann sweep also has the monopoly on complicated data structures. The time bound $O((n+k)\log n)$ requires the sweepline to be a balanced tree as suggested in [2], although simpler data structures are likely to be more attractive in applications where the average number of simultaneously active segments is less than about 100.

Does Theorem 2.2 allow the event queue to be simplified? It would, except that Algorithm 1 needs the crossing events to be sorted by rounded x coordinate. Without this restriction, the priority queue could be replaced by a fixed array of starting and ending events with unordered lists of crossing events interspersed; i.e., each event in the array would point to the list of crossing events that belong immediately afterward.

The overall time bound depends on the number of segments n, the intersection count k and the number of tolerance square intersections k'. Since

$$\sum_i N_i = n + k \quad \text{and} \quad \sum_i k'_i = k'$$

in Theorem 2.1, a total of

$$O((n+k)\log n + k') \tag{2}$$

is spent in Algorithm 1. In practice, N_i is very small and k' is close to k so that the total time for Algorithm 1 is essentially $O(n+k)$, which is dominated by the $O((n+k)\log n)$ for Bentley-Ottmann. In theory, the k' in (2) could dominate because k' could approach the trivial upper bound $n(n+k)$ if all the segments are almost collinear.

3 Results

The algorithm was implemented in C++ and tested on nine small but highly-degenerate input sets involving vertical segments and coincident intersections. Then larger input sets were derived from a data compression problem involving a U.S. government map database [25].

The map database specifies roads, rivers, and other features as polygonal lines defined by sequences of latitude, longitude pairs given in multiples of 10^{-6} degrees. Intersections are indicated by having the same latitude, longitude pair appear in the representation of each road. This forces some straight roads to have many vertices in their polygonal representation as shown in Figure 7. An essential step in compressing this database is to eliminate the unnecessary vertices and depend on a line segment intersection algorithm to recover the intersections during decompression. Thus the input to the segment intersection algorithm is a set of polygonal approximations to map features as shown in Figure 7b. For testing purposes, the polygonal approximations were done with the Wall and Danielsson algorithm with the maximum average error set at 10^{-4} degrees latitude [26].

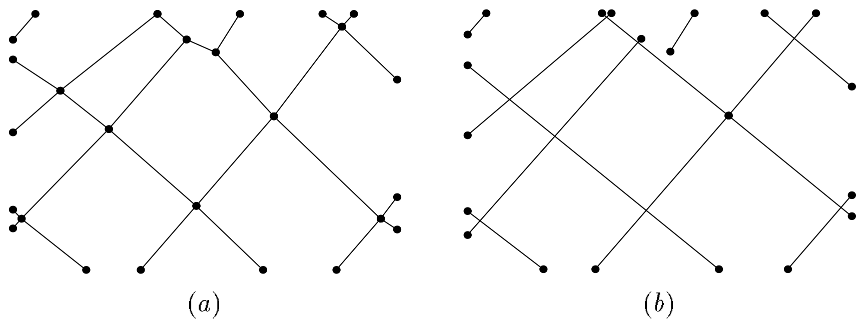

Figure 7: (a) Roads from the map database with data points marked by dots; (b) the same roads with polygonal approximations that bypass explicit intersections. The region shown is about 330 meters by 220 meters.

This application is interesting because the map data contains degeneracies, involves large numbers of line segments, and needs to be decompressed quickly. It also turns out that rounding intersections to grid points can generate extraneous intersections, and these need to be carefully controlled if the correct connectivity is to be preserved.

Table 1 summarizes some test runs for a C++ implementation on an SGI 4D/380S with 33Mhz MIPS R3000 processors. The overhead for finding tolerance square intersections ranges from a factor of $.12/.07 = 1.73$ for the smallest problem to a factor of $46.64/37.8 = 1.23$ for the largest. Thus the time for the Bentley-Ottmann sweep dominates as predicted in Section 2.3.

N. latitude	W. longitude	ϵ	n	k	k'	sweep-line	run time B-O	run time total
40.675–40.7	74.575–74.6	10^{-6}	297	84	86	14	0.07	0.12
40.65–40.7	74.55–74.6	10^{-6}	905	341	346	27	0.24	0.37
40.6–40.7	74.5–74.6	10^{-6}	2894	1902	1939	57	0.94	1.44
40.6–40.8	74.4–74.6	10^{-6}	8614	3630	3723	77	3.33	4.94
47.6–47.8	122.0–122.2	10^{-6}	10829	4153	4474	77	4.53	6.64
47.6–47.8	122.2–122.4	10^{-6}	12752	14291	15362	115	5.97	9.05
40.6–41.0	74.4–74.8	10^{-6}	52359	35694	36418	233	37.80	46.64
40.6–40.8	74.4–74.6	10^{-5}	8614	3555	3750	77	3.37	4.85
47.6–47.8	122.0–122.2	10^{-5}	10829	4005	4485	77	4.67	6.42
47.6–47.8	122.2–122.4	10^{-5}	12752	14076	15355	115	5.84	8.90
40.6–40.8	74.4–74.6	10^{-4}	8614	2740	3833	77	3.42	4.87
47.6–47.8	122.0–122.2	10^{-4}	10829	2912	4560	77	4.25	5.90
47.6–47.8	122.2–122.4	10^{-4}	12752	12046	15171	115	5.88	8.55

Table 1: Results of test runs on the indicated portions of the map database as preprocessed by the Wall and Danielsson algorithm [26]. The grid spacing is ϵ degrees latitude, and the n, k, and k' columns give the number of segments, the number of intersection points, and the number of tolerance square intersections. The "sweepline" column gives the average number of segments on the sweepline, and the last two columns give average run time in seconds for Bentley-Ottmann and Bentley-Ottmann plus Algorithm 1.

Table 1 covers a wide range of values for the grid spacing ϵ. Since coordinates are integer multiples of ϵ, large ϵ values make it more likely that nearby intersection points round to the same coordinates. Such points are collapsed together if they both lie on the same input segment. This makes the k values in the table decline with increasing ϵ because they were computed by counting intermediate vertices added to input segments.

The main effect of increasing ϵ is to increase the gap between k and k'. This is the number of extra vertices added to avoid extraneous intersections like those in Figure 1. For the 40.6–40.8N., 74.4–74.6W. data set, this number ranges from $3723 - 3630 = 93$ at $\epsilon = 10^{-6}$ to $3833 - 2740 = 1093$ at $\epsilon = 10^{-4}$. For comparison, a second iteration of naive rounding with $\epsilon = 10^{-6}$ increases the intersection count from 3630 to 3640 and a third iteration finds no more intersections. Thus iterated naive rounding adds 10 vertices instead of 93 but takes $3.75 + 5.30 + 4.97 = 14.02$ seconds instead of 4.94 seconds. At $\epsilon = 10^{-4}$, the numbers are 57 new vertices instead of 1093 and $3.42 + 4.76 + 5.55 = 13.73$ seconds instead of 4.87.

4 The Extraneous Intersection Theorem

Our original goal was to take a set of line segments, and break them up by inserting intersection points so that no nontrivial intersections remain. Standard algorithms such as Bentley-Ottmann are designed to do this. We only need to show that the discretization by rounding tolerance square intersections preserves the property defined formally as follows: A set $\mathcal{A}$ of line segments is *fully intersected* if unequal segments in $\mathcal{A}$ intersect at their endpoints or not at all.

The discretization process depends on the set T of segment endpoints and on the region

$$R^- = \left\{ (x,y) \mid -\frac{1}{2} < x \leq \frac{1}{2},\ -\frac{1}{2} < y \leq \frac{1}{2} \right\}$$

obtained by negating the coordinates of points in the region R that Section 2 used for defining tolerance squares. The discretization operator $\mathcal{D}_T$ maps any real point set $S \subseteq \mathbb{R}^2$ into the set of all segments AB such that there exists a segment $\bar{A}\bar{B} \subseteq S$ where

$$\lceil T \rfloor \cap (\bar{A}\bar{B} + R^-) = \{A, B\}$$

and $\lceil T \rfloor = (T + R^-) \cap \mathbb{Z}^2$ is result of rounding points in T to integer grid points $\mathbb{Z}^2$. (For a set of segments $\mathcal{A}$, $\mathcal{D}_T(\mathcal{A})$ is the union of all $\mathcal{D}_T(S)$ for $S \in \mathcal{A}$).

Theorem 4.1 *If T is the set of segment endpoints from a fully intersected segment set $\mathcal{A}$, then $\mathcal{D}_T(\mathcal{A})$ is also fully intersected.*

The proof depends on two lemmas. The first uses the notation $\lceil x \rfloor$ to mean $\lfloor x + \frac{1}{2} \rfloor$.

Lemma 4.2 *For any line segment s, there is a direction $(\alpha_x, \alpha_y) = (1, \pm 1)$ such that no two points in $\mathbb{Z}^2 \cap (s + R^-)$ have the same $\alpha_x x + \alpha_y y$.*

Proof. Points in $\mathbb{Z}^2 \cap (s + R^-)$ are of the form $(\lceil x_i \rfloor, \lceil y_i \rfloor)$, where $(x_i, y_i) \in s$. Choosing $\alpha_y = 1$ if s has positive slope and $\alpha_y = -1$ otherwise guarantees

$$\alpha_x \lceil x_1 \rfloor + \alpha_y \lceil y_1 \rfloor \neq \alpha_x \lceil x_2 \rfloor + \alpha_y \lceil y_2 \rfloor$$

unless $(\lceil x_1 \rfloor, \lceil y_1 \rfloor) = (\lceil x_2 \rfloor, \lceil y_2 \rfloor)$. □

Lemma 4.3 *If segments s_1 and s_2 have endpoints in T and intersect at their endpoints or not at all, then any pair of unequal segments $\sigma_1 \in \mathcal{D}_T(s_1)$ and $\sigma_2 \in \mathcal{D}_T(s_2)$ intersect at their endpoints or not at all.*

Proof. Lemma 4.2 guarantees that there is a coordinate system

$$(\xi, \eta) = \left(\frac{\alpha_x x + \alpha_y y}{2}, \frac{-\alpha_y x + \alpha_x y}{2} \right),$$

where the endpoints of the segments in $\mathcal{D}_T(s_1)$ all have different ξ coordinates. If these points are $(\xi_{1,1},\eta_{1,1})$, $(\xi_{1,2},\eta_{1,2})$, $\ldots$, $(\xi_{1,m_1},\eta_{1,m_1})$, they describe a piecewise-linear function on the interval $[\xi_{1,1},\xi_{1,m}]$:

$$F_1(\xi) = \frac{\eta_{1,i}(\xi_{1,i+1}-\xi)+\eta_{1,i+1}(\xi-\xi_{1,i})}{\xi_{1,i+1}-\xi_{1,i}} \quad \text{where} \quad \xi_{1,i} \le \xi \le \xi_{1,i+1}.$$

A similar function $F_2(\xi)$ for the discretization of s_2 is based on the endpoints $(\xi_{2,j},\eta_{2,j})$ for $1 \le j \le m_2$ of the segments in $\mathcal{D}_T(s_2)$. Functions F_1 and F_2 approximate the lines

$$\eta = \beta_1 + \gamma_1\xi \quad \text{and} \quad \eta = \beta_2 + \gamma_2\xi$$

that contain s_1 and s_2.

The lemma can be thought of as a statement about the ξ values where $F_1(\xi) = F_2(\xi)$. Since s_1 and s_2 intersect only at their endpoints, we can assume without loss of generality that

$$\beta_1 + \gamma_1\xi \le \beta_2 + \gamma_2\xi \tag{3}$$

for all ξ where F_1 and F_2 are both defined. Then it suffices to show that

$$F_1(\xi) \le F_2(\xi) \quad \text{for} \quad \xi \in \{\, \xi_{j,i} \mid 1 \le j \le 2,\ 1 \le i \le m_j, 1 \le \xi_{j,i} \le \xi_{2-j,m_{2-j}} \,\}. \tag{4}$$

so that $F_1(\xi) \ne F_2(\xi)$ between $\xi_{j,i}$ values unless $F_1(\xi) = F_2(\xi)$ for an interval $\xi_{j,i} \le \xi \le \xi_{j,i+1}$. When this happens, $(\xi_{j,i},\eta_{j,i})$ and $(\xi_{j,i+1},\eta_{j,i+1})$ are the endpoints of a segment common to $\mathcal{D}_T(s_1)$ and $\mathcal{D}_T(s_2)$.

Since the segments in $\mathcal{D}_T(s_j)$ belong to $s_j + R^-$ for $j = 1,2$, the difference $F_j(\xi) - \beta_j - \gamma_j\xi$ is limited to the range of η coordinates in R^-. This ranges over an open or semi-open interval

$$(-\tfrac{1}{2},\tfrac{1}{2}) \quad \text{or} \quad [-\tfrac{1}{2},\tfrac{1}{2}).$$

Thus (3) implies $F_2(\xi_{1,i}) \ge \eta_{1,i}$ if $\beta_2 + \gamma_2\xi_{1,i} \ge \eta_{1,i} + \frac{1}{2}$. Otherwise $\beta_2 + \gamma_2\xi_{1,i} \in \eta_{1,i} + R$ and the definition of $\mathcal{D}_T$ forces $F_2(\xi_{1,i}) \ge \eta_{1,i}$. Similar reasoning shows $F_1(\xi_{2,i}) \le \eta_{2,i}$ so that (4) holds and the lemma follows. $\Box$

Theorem 4.1 follows from Lemma 4.3. The segments in $\mathcal{D}_T(\mathcal{A})$ belong to $\mathcal{D}_T(s)$ for $s \in \mathcal{A}$, and the lemma guarantees that such segments intersect only as allowed for segments in a fully intersected set.

5 Conclusion

The simple idea of breaking segments where they intersect tolerance squares suffices to eliminate the extraneous intersections that can result from rounding line segment intersections. Since Theorem 4.1 does not require input segments to start and end at grid points, the idea can also be used for rounding segment endpoints to a coarser grid as can be useful in data compression applications.

Section 2 gives a very practical algorithm based on the Bentley-Ottmann sweep. It also gives a simple, efficient scheme for handling degeneracies. The Bentley-Ottmann sweep was chosen for its practical importance, but the ideas could be applied to other algorithms if desired.

The algorithm improves on the time and output size bounds of Greene and Yao [10] by settling for a weaker relationship between the input and output topologies. Greene and Yao show that the intersection between two of their redrawn line segments is a single point or a polygonal line. Lemma 4.3 allows the intersection to be a set of disjoint line segments, but the proof does show that there is no interleaving: (4) forces the η coordinates of the output for segments s_1 and s_2 to be ordered as the η coordinates for s_1 and s_2 are.

Atts.
References

References

[1] Pankaj K. Agarwal and Micha Sharir. Red-blue intersection detection algorithms with applications to motion planning and collision detection. *SIAM Journal on Computing*, 19(2):297–321, April 1990.

[2] Jon L. Bentley and Thomas A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, September 1979.

[3] K. Q. Brown. Comments on 'algorithms for reporting and counting geometric intersections'. *IEEE Transactions on Computers*, C-30(2):147–148, February 1981.

[4] Bernard Chazelle and Herbert Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, January 1992.

[5] Kuang-Wei Chiang, Surendra Nahar, and Chi-Yuan Lo. Time-efficient vlsi artwork analysis algorithms in GOALIE2. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 8(6):640–648, June 1989.

[6] Kenneth L. Clarkson. Applications of random sampling in computational geometry, II. In *Proceedings of the 4th Annual ACM Symposium on Computational Geometry*, pages 1–11, New York, June 1988. ACM.

[7] H. Edelsbrunner, M. H. Overmars, and R. Seidel. Some methods of computational geometry applied to computer graphics. *Computer Vision Graphics and Image Processing*, 28(1):92–108, October 1984.

[8] Herbert Edelsbrunner and Ernst Peter Mücke. A technique to cope with degenerate cases in geometric algorithms. In *Proceedings of the 4th Annual ACM Symposium on Computational Geometry*, pages 118–133, New York, June 1988. ACM.

[9] Michael T. Goodrich. A polygonal approach to hidden-line and hidden-surface elimination. *CVGIP: Graphical Models and Image Processing*, 54(1):1–12, January 1992.

[10] Daniel H. Greene and F. Frances Yao. Finite-resolution computational geometry. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 143–152. IEEE Computer Society, October 1986.

[11] Z. Jaeger, R. Engleman, and A. Sprecher. Statistics of structure within solid fragments studied by 2D simulation. *Applied Physics*, 59(12):4048–4056, June 1986.

[12] Jerzy W. Jaromczyk and Godfried T. Toussaint. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, 80(9):1502–1517, September 1992.

[13] Harry G. Mairson and Jorge Stolfi. Reporting and counting intersections bettween two sets of line segments. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 307–325. Springer Verlag, 1988.

[14] Victor Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 500–506. IEEE Computer Society, October 1989.

[15] Victor Milenkovic. Rounding face lattices in the plane. In *1st Canadian Conference on Computational Geometry*, 1989.

[16] Victor Milenkovic. Rounding face lattices in *d* dimensions. In Jorge Urrutia, editor, *Proceedings of the Second Canadian Conference on Computational Geometry*, pages 40–45, Ontario, August 1990. University of Ottawa.

[17] Victor Milenkovic. Robust polygon modeling. *Computer-Aided Design*, to appear.

[18] Victor Milenkovic and Lee R. Mackman. Finding compact coordinate representations for polygons and polyhedra. *IBM Journal of Research and Development*, 34(35):753–769, September 1990.

[19] Ketan Mulmuley. A fast planar partition algorithm, I. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 580–589, 1988. To appear in J. Symbolic Logic.

[20] Ketan Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, New York, 1993. To appear.

[21] J. Nievergelt and F. P. Preparata. Plane sweep algorithms for intersecting goemetric figures. *Communications of the ACM*, 25(10):739–747, October 1982.

[22] János Pach and Micha Sharir. On vertical visibility in arrangements of segments and the queue size in the bently-ottman line sweeping algorithm. *SIAM Journal on Computing*, 20(3):460–470, June 1991.

[23] Kokichi Sugihara. An intersection algoirthm based on Delaunay triangulation. *IEEE Computer Graphics and Applications*, 12(2):59–67, March 1992.

[24] Robert B. Tilove. A null-object detection algorihtm for constructive solid geometry. *Communications of the ACM*, 27(7):684–694, July 84.

[25] TIGER/LINE census files, 1990, technical documentation. Technical report, Bureau of the Census, U. S. Dept. of Commerce, Washington, D.C., 1991.

[26] Karin Wall and Per-Erik Danielsson. A fast sequential method for polygonal approximation of digitized curves. *Computer Vision Graphics and Image Processing*, 28(2):220–227, November 1984.

[27] Ying-Fung Wu, Peter Widmayer, Martine D. F. Schlag, and C. K. Wong. Rectilinear shortest paths and minimum spanning trees in the presence of rectilinear obstacles. *IEEE Transactions on Computers*, C-36(3):321–331, March 1987.

Practical Segment Intersection with Finite Precision Output

John D. Hobby

Bell Laboratories
700 Mountain Ave.
Murray Hill, NJ 07974

ABSTRACT

The fundamental problem of finding all intersections among a set of line segments in the plane has numerous important applications. Reliable implementations need to cope with degenerate input and limited precision. Representing intersection points with fixed precision can introduce extraneous intersections. This paper presents simple solutions to these problems and shows that they impose only a very modest performance penalty. Test data came from a data compression problem involving a map database.

Key Words: line segment intersection; degeneracy; data compression