A Generic Plane-Sweep for Intersecting Line Segments *

Wolfgang Freiseisen
(wfreisei@risc.uni-linz.ac.at)
Petru Pau
(ppau@risc.uni-linz.ac.at)
RISC – Linz

November 30, 1998

Abstract

An algorithm for reporting all intersections of a set of line segments is presented. Based on the Bentley-Ottmann sweep-line algorithm, it handles all degenerate cases and takes $O((n + n_{dip})\log(n) + k)$ time, where n is the number of segments, n_{dip} is the number of distinct intersection points and k is the size of the output.

Unlike other algorithms solving the line segment intersection problem, we view the degenerate cases as parts of the problem. Therefore, they are solved in the frame of a common approach, instead of being handled separately. Some implementation details are also given using the generic programming mechanism of *C++* in the frame of CGAL (Computational Geometry Algorithm Library).

1 Introduction

Intersecting a set of line segments is one of the fundamental tasks in Computational Geometry. Important applications are the overlay of planar maps and calculating the intersection (or union) of simple polygons (see e.g. [Fre98]). There are many other applications, especially in CAD/CAM, like clipping operations or hidden-surface removal ([FvDF+93]).

The problem of determining whether any pair of a set of line segments do intersect, given and solved by Shamos and Hoey [SH76] in $O(n\log(n)+k\log(n))$ time and $O(n + k)$ space, was extended by Bentley and Ottmann [BO79] to detect how many pairs do intersect and to report them. They introduced the now familiar *line sweep paradigm*: A vertical line is moved over all segments from the left to the right and all intersections are maintained in correct order. Other optimal algorithms are given by Nievergelt and Preparata [NP82]. They showed that merging two planar convex subdivisions can be computed in $O(n\log(n)+k)$

*This research is supported by the ESPRIT IV LTR Project No. 21957 (CGAL) of the European Union.

time. Chazelle and Edelsbrunner [CE92] gave another optimal algorithm, where they also tried to reduce the amount of space.

We present in this paper a new version of the Bentley-Ottmann algorithm that takes into account all possible cases (besides the segments in general position, our algorithm handles overlapping segments, multiple intersections, vertical segments and so on); moreover, instead of handling degeneracies as special cases, we treat them in a general concept.

Whereas the Bentley-Ottmann algorithm takes $O(n\log(n) + k\log(n))$ time and $O(n+k)$ space, we achieve with suitable data structures (using the same amount of space) a time complexity of $O(n\log(n) + n_{dip}\log(n) + k)$, where n denotes the number of segments, n_{dip} is the number of distinct intersection points, and k the number of reported intersections – points or segments.

Section 2 contains the overall description of the algorithm. In Section 3 all details, including the definition of the used data structures (event-point-schedule, sweep-line status) are stated. In the Appendix we give some details of the generic implementation in the programming language *C++*. This implementation is based on the CGAL-library[1] (*Computational Geometry Algorithms Library*) [Ove96] [FGK+96].

2 The algorithm

Basically, the algorithm has been described in [BO79]. Its input consists in a set S of line segments, given by the coordinates of their endpoints. The output is the set of all distinct intersections of elements in S, either points or segments.

The algorithm makes use of a well known paradigm: *the line-sweep*. A vertical line sweeps the plane from $-\infty$ to $+\infty$ and intersects all segments in S. The sequence of segments intersected by the line determines its *status*. This status changes at certain points: It changes at the first endpoint of a segment (this endpoint will be called later *source vertex*) at the second endpoint of a segment (the *target vertex*), and when an intersection between two segments is reached.

The algorithm maintains two data structures. First the points where the line status changes: the *event points*. They are stored in a special data structure, the *event queue*. Its implementation must provide queue-specific fast operations (access/deletion of the first element, random insertion, etc.).

The *line status* is stored in another structure, suitable for fast insertion and deletion. Here the segments are kept ordered, according to the y-coordinate of their intersections with the sweeping line.

The sketch of the algorithm is presented in the following:

```
insert the vertices of the initial segments in the event_queue;
line_status := empty;
while event_queue is non-empty
    next_event := extract_first(event_queue);
    if next_event is
        initial vertex:
            treat the source vertex case;
```

[1] http://www.cs.ruu.nl/cgal

final vertex:
7. treat the `target vertex` case;
intersection point:
8. treat the `intersection point` case;

The output is reported in the `target vertex` sub-algorithm (the collinear segments are reported here) and in the `intersection point` sub-algorithm (the punctual intersections).

3 Detailed Description of the Algorithm

A robust implementation of this algorithm must work in all possible situations, including the degeneracies. The input segments can be in any position, e.g. collinear, or with multiple intersections.

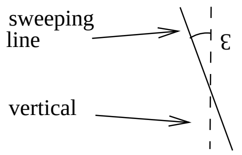
These cases can be grouped into classes and treated similarly. We relate the partitioning of the cases of degeneracy to the order in which the points are processed.

We call the endpoints of the segments in S *vertices*. The segments are *directed* according to the order in which the sweep line reaches their vertices. The *source* is the first touched vertex, and the *target* is the other one.

Three kinds of points are important for the algorithm: the *vertices* (source and target) of the initial segments, and the *intersection points*. Each segment is handled at least twice: once for each vertex, and eventually once for each of its intersections with other segments.

The vertical segments should also have their vertices processed in turn. We can avoid the special treatment of these degeneracies:

The sweeping line is "almost" vertical, that is, it forms a counterclockwise angle of infinitely small value ε with the vertical.



With this approach, for vertical segments the vertex of lower y-coordinate will be the source.

In the following definitions, we use a few notations:

- if p is a point in plane, its coordinates are denoted $(p.x, p.y)$;
- $seg(p)$: if p is an endpoint of a segment s, then $seg(p)$ returns s;
- $slope(s)$: if s is a non-vertical segment of source p and target q, then:

$$slope(s) := \frac{q.y - p.y}{q.x - p.x};$$

  if s is vertical, it is considered as having infinite slope ($slope(s)$ is greater than any real number).

We define now the order according to which the event points are stored in the event queue. Obviously, problems occur only when two points have the same abscissa, or when they have equal coordinates but come from different segments.

Definition 1. (Before) p is before q iff:

$p.x < q.x$ or (before 1)
$(p.x = q.x$ and $p.y < q.y)$ or (before 2)
$\big(p = q$ and
  (p is a source and q is a target) or (before 3)
  (p is a source and q is an intersection) or (before 4)
  (p is an intersection and q is a target) or (before 5)
  (both are sources and $slope(seg(p)) > slope(seg(q))$) or (before 6)
  (both are targets and $slope(seg(p)) < slope(seg(q))$) or (before 7)
  (p is the intersection $(s1, s2)$,
  q is the intersection $(s3, s4)$, and
  $slope(s2) > slope(s3)$)$\big)$. (before 8)

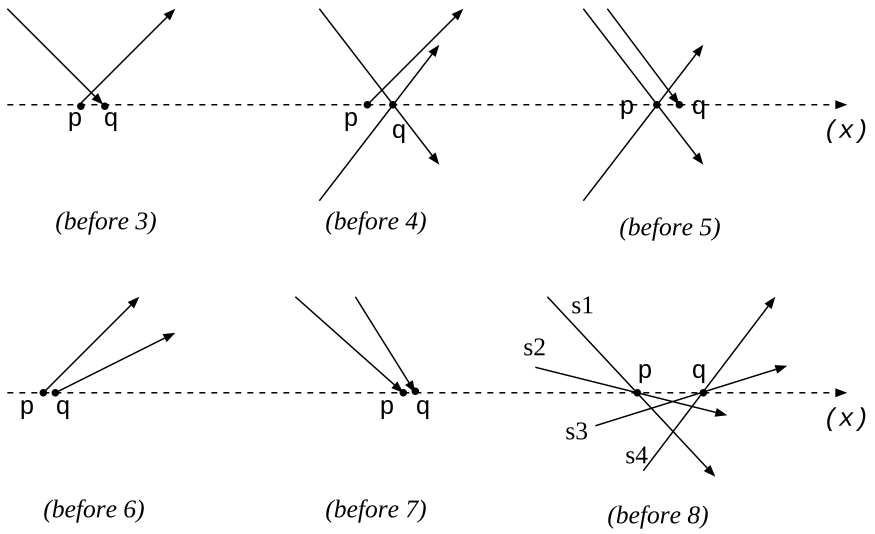


Figure 1: Special cases of *before* relation (p and q *have equal coordinates*).

Some cases in which p and q are comparable using (before 3)–(before 8) are depicted in Fig. 1. In all the cases shown here, p and q have the same coordinates, though they come from different segments, either as vertices or as intersection points.

The segments are represented as arrows pointing from source to target.

One situation is not caught by the definition of *before*: a vertex that is a common source or target for collinear segments. Such a vertex will be processed once for each segment, and the algorithm will perform exactly the same operations each time; it follows that for such cases the order is not important.

At each vertex or intersection point, the intersection between the sweeping line and the segments changes: at the right of the point, either some segments start intersecting the sweep line, or stop intersecting, or the positions of their intersections with the line change.

It is necessary to define a "vertical" order between segments – as the order in which they intersect the line at a given x-coordinate.

Definition 2. (Below) $s1$ is below $s2$ at $x0$ iff:

$$p_{x0}(s1).y < p_{x0}(s2).y \text{ or} \qquad \text{(below 1)}$$
$$(p_{x0}(s1).y = p_{x0}(s2).y \text{ and}$$
$$(p_{x0}(s1) \text{ is source or } p_{x0}(s2) \text{ is source }) \text{ and}$$
$$slope(s1) > slope(s2)). \qquad \text{(below 2).}$$

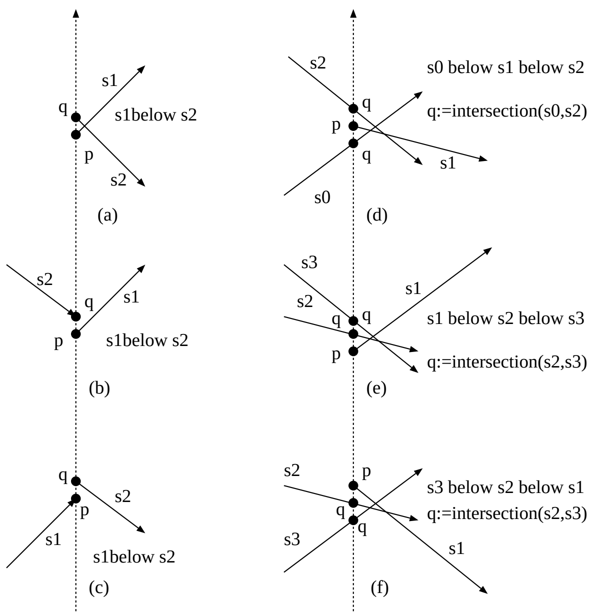


Figure 2: Special cases in which the *below* relation is defined (p and q *have equal coordinates*).

Clearly, if two segments are not comparable at $x0$ (i.e. none of them is below the other) they are equal, therefore $x0$ corresponds to an intersection point (or to an endpoint of a common segment, in case of collinearity).

With these two definitions, we proceed now to the description of the main data structures and algorithm.

3.1 Main Data Structures

3.1.1 The Event-Point Schedule

The event-point schedule (shortly event queue) contains information associated to the points at which the sweep-line changes its status.

There are three types of events:

1. source vertex;
2. target vertex;
3. intersection point.

It would be suitable for the events that come from source and target vertices to store some information about their incident segments. The intersection points will have associated such information for a pair of segments (those segments that intersect here).

The events will be inserted in this structure according to the order defined above. Definition 1 makes sure that intersection points are inserted in the event queue *between* the source and target vertex of each intersecting segment.

In case of multiple intersections, the idea is to forbid the insertion in the event queue of several points having the same coordinates, but different pairs of intersecting segments. The only point stored will mark the intersection between the upmost and the lowest segments that intersect here.

There will be situations in which an already existent intersection point must be updated; an example is depicted in Fig. 3 (a). Here, after processing the source vertices of $s1$ and $s2$, an intersection point is detected and inserted in the event-point structure. When the algorithm processes $s3$, it also detects the intersection with $s2$; the already existent intersection point $(s1, s2)$ will be replaced by $(s1, s3)$.

In situations like that depicted in Fig. 3 (b) the event-point structure keeps only the intersections (s_1, s_k).

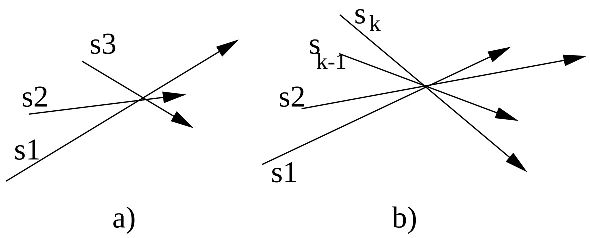


Figure 3: Special situations for multiple intersection points.

The case of multiple intersection points with collinear segments can be solved by using a similar approach. If $s1$ is non-collinear with $s2, \ldots, s_k$ and all these latter segments are collinear, then a single intersection, between $s1$ and a segment in $s2, \ldots, s_k$, is inserted in the event-queue structure. The companion of $s1$ in the pair that defines this intersection point will be defined below.

The structure should support the following operations:

- *insertion* of an event point;
- *extraction and deletion* of the first point, i.e. minimal with respect to the order *before*.

A suitable structure would be a *priority queue* or a *balanced tree*. With some care, the STL[2] *set* can be used. These structures ensure $O(\log n)$ operations for

[2] *Standard Templates Library*, extensively used in CGAL.

the insertion of a new element, and constant time for eliminating the first and for determining the neighbors of a given element.

3.1.2 The Sweep-Line Status Structure

The sweep-line status (shortly line status) contains the segments intersected by the sweeping line between two event points, sorted with respect to the y-coordinate of their intersection. This order, introduced by Hoey [SH76], has been extended at the beginning of this section to cover a situation of equality.

Each segment contained in this structure has two neighbors, one from above and one from below (except the upmost and the lowest segments). The order in which the segments occur in the structure must be a consistent representation of the position of the segments *between* two event points.

In the general case, between two consecutive events, the order of segments in the line status corresponds to the order in which the sweeping line cuts the segments at the first event. The degeneracies introduce some cases that should be treated separately. For these cases, the order mirrors the position of the segments *immediately after* the first event.

The line status should support the following operations:

Insertion of a segment when its source is reached as an event.

After insertion the segment must be tested against its neighbors in the line status and their eventual intersection points will be inserted in the event queue.

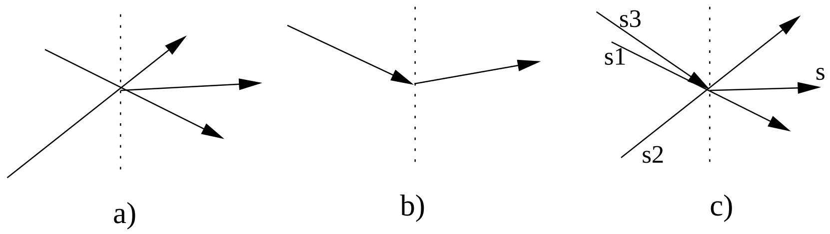


Figure 4: Special situations for multiple intersection points.

The way we have defined the orders "before" (Definition 1) and "below" (Definition 2) enables the treatment of situations of equality with another vertex/intersection point similar to the general case. Indeed:

- If the source vertex is equal to an intersection point, the intersection will be positioned after the source in the event queue. The position of the corresponding segment in the line status will be determined by the relation between its slope and the slope of the intersecting segments (Fig. 4 (a)).
- If the source vertex is equal to a target vertex, as in Fig. 4 (b) and (c), two situations are possible:
  1. the corresponding segments will be neighbors; when tested against each other, their intersection point will be found and eventually inserted in the event queue;

2. the corresponding segments will not be neighbors; this situation can happen only if the actual event is also an intersection point. Again, the intersection between the newly inserted segment and a neighbor is found and it eventually updates the event queue. Moreover, these segments will be found to belong to the pencil of segments that intersect at the same points, hence their intersection will also be reported.

Elimination of a segment, when its second endpoint is reached as an event.

After the elimination, its former neighbors must be checked against each other and, if they intersect at a point that lies at the right of the current event, this intersection should be inserted in the event queue.

Note: Due to the way we defined the order "before", all the intersections of a segment with other segments have been detected and processed at the moment of its removal from the line status.

Change place. When an intersection event point is reached, the segments that intersect here must *change place* in the structure. This operation is followed by testing these segments against their new neighbors and by inserting their eventual intersections that lie at the right of the current event point in the event queue.

If there exists a pencil of segments sharing the current event, they form a contiguous sequence of elements in the line status. This sequence is bounded by the two segments that are associated to the event (those whose information is assigned to the intersection point).

At the right of the intersection point, their order in the line status is reversed. Only the upper and the lower segment should be tested against their neighbors (see fig. 7, (b)).

Above/Below. In order to be able to test a segment against its neighbors in the line status, the segments from *above* and *below* are easy to find.

For this structure a *dictionary* is suitable. The insertion and elimination of an element can thus be done in $O(\log n)$ operations (here we denoted by n the size of the structure); the neighbors of an element can be reported in constant time. In addition, this structure should allow to reverse a contiguous portion for which the bounding elements are known.

The only degeneracy that has not been discussed yet is when the intersection of two segments is also a segment. This happens at an event point that corresponds to a source vertex. When the new segment is tested against its neighbors – in fact, when it is inserted in the line status structure – it is found to have the same y-coordinate and the same slope as one of its neighbors. We suggest that this new segment is inserted in the line status structure *below* the segment with which it shares a portion. Subsequently, it should be tested against the upper and lower noncollinear neighbor.

If an intersection point (found in this way) belongs to more segments, they can be safely reversed in the line status structure.

3.2 Algorithm Details

In all the subsequent discussions, x_0 denotes the x-coordinate of the event point (next_event) p; s is the segment to be inserted/deleted, and $s1$ and $s2$ its neighbors in the line status: $s1$ is considered above (or equal to) s at x_0, and $s2$ is below. When there are many segments equal to s at x_0, $s1$ is the "lowest" in the line status structure.

3.2.1 The SOURCE VERTEX case

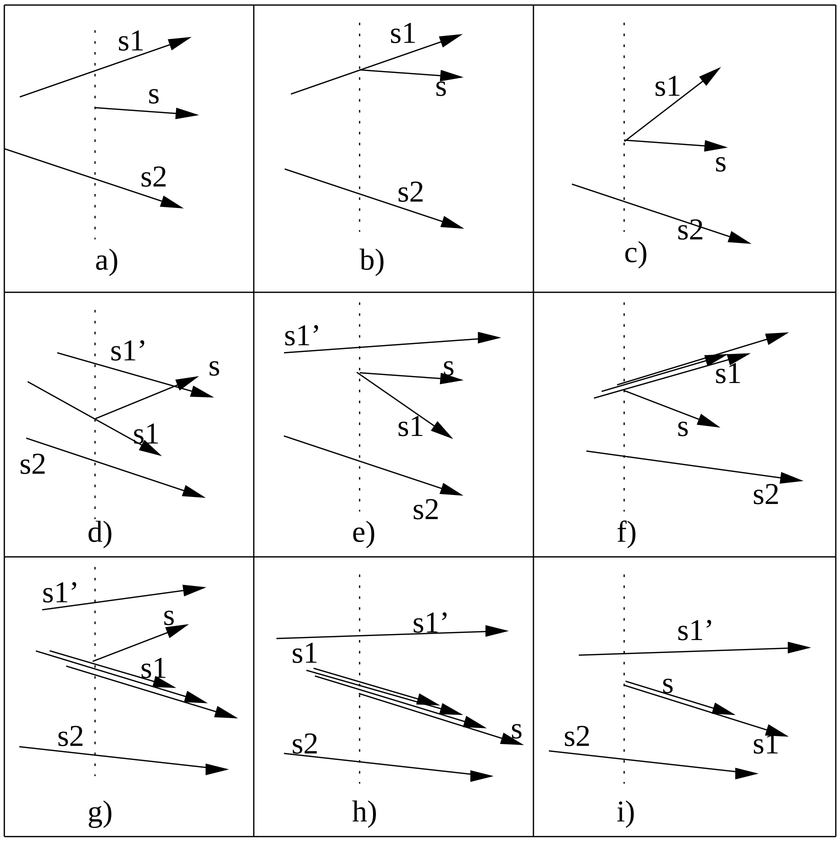


Figure 5: The general case (a) and some degenerate cases when an initial vertex is encountered.

The sketch of the algorithm for an initial vertex is the following:

```
1.  determine the segment s that contains p as a source vertex;
2.  insert s in line_status and determine its neighbors s1 and s2
3.  if s1 exists
        then
4.          if (p belongs to s1) and (s and s1 are collinear)
5.              then treat Collinearity case
                else /* s1 is a common neighbor */
6.                  test s against s1;
7.                  if s2 exists
                        then test s agains s2;
        else
8.          if s2 exists
9.              then test s against s2;
10. insert the evenual intersections in the event_queue
        or eventually update their associated information.
```

Insertion in the line status

If p belongs only to s, the position of s in `line_status` is simply determined by a search. There are two situations in which p belongs to other segments:

1. p has been previously detected as an intersection point; in this case, all the segments intersecting at p are in `line_status`, even though there are some having p as source vertex. These segments can be compared with s (using *(before2)*) and the position of s in the line status is well defined.

   The intersection point eventually updates, if the new segment is the upmost or the lowest in the pencil of segments that intersect here.

2. p belongs to another segment that has been previously inserted in `line_status`; again, using *(before 2)*, s is comparable with this segment and, moreover, they will be neighbors. When tested against each other, their intersection is found and inserted in `event_queue`.

Collinearity case

The case (i) differs from (h) in Fig. 5 by the fact that there exists at least one other segment collinear with s with source p. Nonetheless (h) and (i) can be treated together.

```
5.1.  find s1' the lowest segment in line_status
          that is above and non-collinear with s;
5.2.  if s1' exists
5.3.      then test s against s1';
5.4.  if s2 exists
5.5.      then test s against s2;
```

The operation in line 5.1. can be performed by searching the lowest upper segment in `line_status` at $x_0 - \varepsilon$, where ε is a value smaller than the distance on x-axis between p and the previous event.

Analysis of SOURCE VERTEX subalgorithm

Some appropriate information should be kept in order to ensure constant time for the operation in line 1 – determining the segment whose source is p. The geometric operations – testing collinearity, computing intersections, etc. – are done in constant time. The computation of the neighbors is also done in constant time.

Lines 2 and 5.1 contributes with $O(\log(\text{size}(\texttt{line\_status})))$ operations.

The insertion in line 10 is done in $O(\log(\text{size}(\texttt{event\_queue})))$ operations.

Summarizing, we count for this step $O(\log(\text{size}(\texttt{event\_queue})) + \log(\text{size}(\texttt{line\_status})))$ operations.

3.2.2 The TARGET VERTEX case

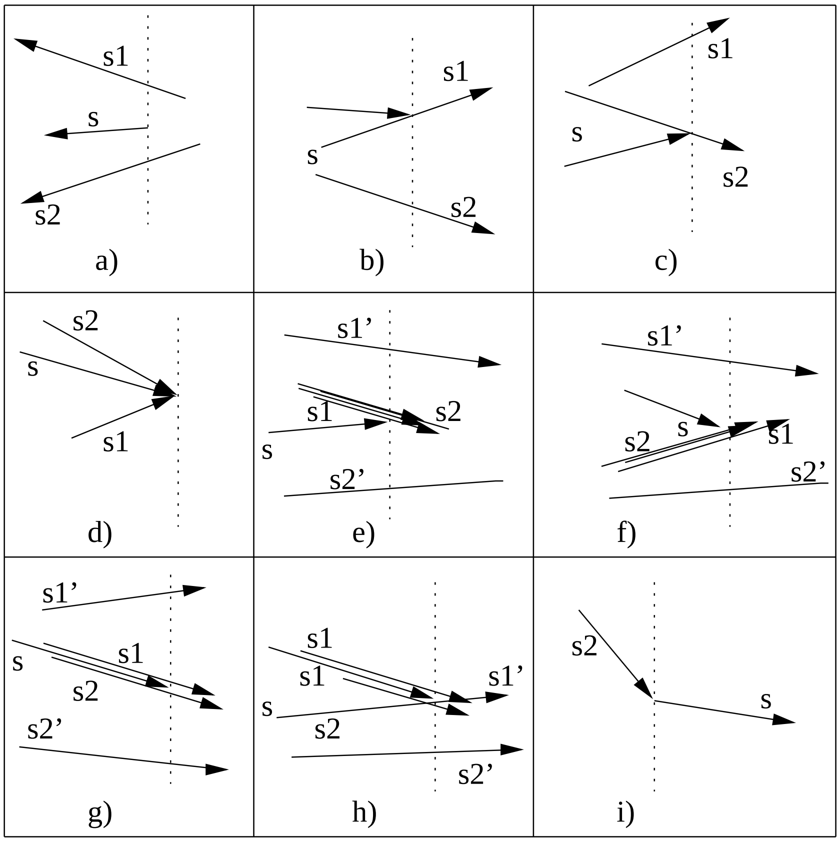


Figure 6: The general case (a) and degenerate cases (b) - i)) when a final vertex is encountered.

When a target vertex is reached, all its eventual occurences as an intersection point have already been processed for all but collinearity cases – because of the convention regarding the order of events.

Figure 6 contains a few situations that can occur when treating the TARGET VERTEX case. However, only the general situation (a) and the collinearity cases (e) - i)) must be treated separately.

In the collinearity case, all the collinear segments have been tested against their non-collinear neighbors – either in the phase of inserting a new segment or at the intersection, so this test is not necessary.

1. determine the segment s whose target vertex is p;
2. determine its neighbors $s1$ and $s2$ in `line_status`;
3. remove s from the `event_queue`;
4. if both $s1$ and $s2$ exist
    then
5.   if s is not collinear with any of $s1$ or $s2$
6.   then test $s1$ against $s2$
7.   insert or update their eventual intersection in the event queue
                if it is after p;
            else
8.   for all segments s' collinear with s
                    report their common segment with s.

Remark. In Step 8, the portion of the segment *between* the last event point and the current one is reported as common intersection, although for some of the initial segments this can be only a part of their intersection.

Analysis of TARGET VERTEX subalgorithm

Lines 1, 5, 6 are done in constant time. If the event point stores any information about the position of its parent segment in the `line_status`, then line 2 will require constant time.

The operations in line 2 and 7 can be performed in $O(\log|$`event_queue`$|)$ time. The operations performed in line 8 depend on the size of the output; they are bounded above by $O(n)$ (since at most $n-1$ segments can be collinear with the segment whose target vertex is being currently processed).

The total number of operations performed at this step is $O(\log|$`line_status`$|+\log|$`event_queue`$|+n_t)$, where n_t is the size of the output reported when treating a target vertex p_t.

3.2.3 The INTERSECTION POINT case

An intersection point might appear only once in the events queue, even though it is a multiple intersection (Fig. 7). Consequently, when such an event is reached, all the segments between the two that are associated with the event must reverse their order.

Some intersection tests are done afterwards. The upmost and lowest segments in the pencil are tested against their upper and respectively lower non-intersecting neighbors.

In the following description, u and v are the segments that intersect in p, so that v is below u immediately before p.

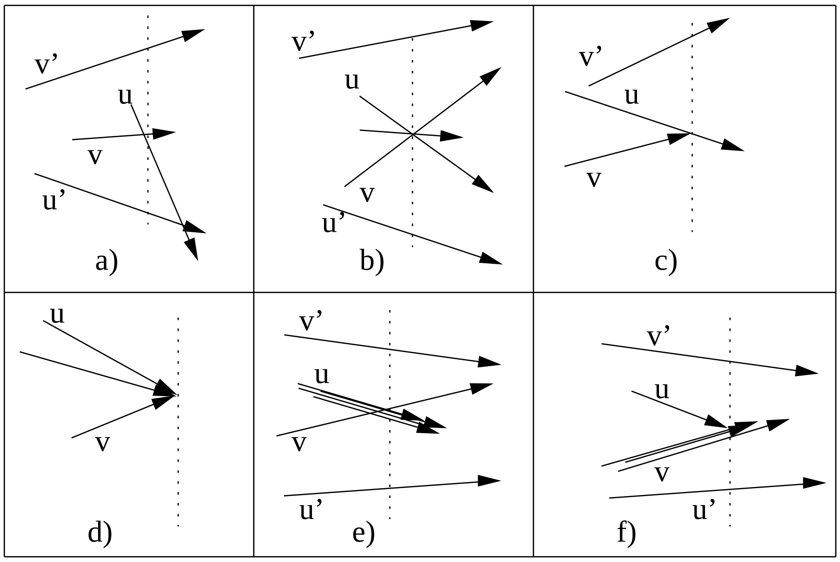


Figure 7: The general case (a) and degenerate cases when an intersection point is encountered.

1. determine the position of u and v in `line_status`;
2. determine the upper and lower non-intersecting neighbors $u1$ and $v1$;
3. report p as the intersection between the segments
   in the pencil of segments bounded by u and v;
4. reverse the position in `line_status` of all the segments in this pencil;
5. check u against $u1$ and v against $v1$;
6. insert or update the eventual intersections in the `event_queue`.

Analysis of INTERSECTION POINT Subalgorithm

Line 1 and 2 are done in constant time, with suitable information stored in the event queue as well as in the line status; line 5 is also done in constant time.

Now, if n_m segments intersect in this point, they are stored between $u1$ and $v1$; the current intersection point is reported for each pair of these segments, thus for $O(n_m^2$ such pairs.

We assume that the successor of an event in `event_queue` can be found in constant time. Thus, for reversing the n_m segments in the line status, $O(n_m$ operations have to be done (in line 4).

Finally, the insertion in the event queue of any eventual intersection of the upper and lower segments in the pencil with their corresponding neighbors takes $O(\log |$`event_queue`$|)$ operations (line 6).

The total number of operations performed at this step is $O(\log |$`event_queue`$|+ n_i)$, where n_i is the number of intersections reported when treating an intersection point p_i.

3.3 The Overall Analysis

There are n sources and n target vertices; there are two kinds of intersections: points and segments.

The collinearities are treated in the `target vertex` case, while the punctual intersections are handled in `intersection point` case.

The size of the event queue can grow up to the maximum number of intersection points, which is $O(n^2)$; thus $\log |\texttt{event\_queue}|$ is $O(\log(n))$. Clearly, $\log |\texttt{line\_status}|$ is of the same order.

Source: since there are n source vertices, and for each of them the `source vertex` subalgorithm does $O(\log n)$ operations, the overall time for source vertices is $O(n \log(n))$;

Target: the overall time for target vertices is $O(n \log(n) + k_t)$, where k_t is the total number of common segments reported; k_t ranges between 0 and n^2;

Intersection: the overall time consumed by the intersection subalgorithm is $O(n_{dip} \log n + k_i)$, where n_{dip} is the number of **d**istinct **i**ntersection **p**oints, and k_i is the total number of punctual intersections reported; both range between 0 and n^2, but n_{dip} is not greater than k_i.

If we sum all the operations, we get $O((n+n_{dip}) \cdot \log(n) + k)$, where $k = k_t + k_i$ is the size of the output.

References

[BO79] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[CE92] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39:1–54, 1992.

[FGK+96] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The cgal kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *ACM Workshop on Applied Computational Geometry*, pages 191–202, Philadelphia, Pennsylvenia, May, 27–28 1996. Lecture Notes in Computer Science 1148.

[Fre98] Wolfgang Freiseisen. Colored dcel for boolean operations in 2d. Technical Report 98–03, RISC, University of Linz, Februar 1998.

[FvDF+93] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, and Phillips. *Introduction to Computer Graphics*. Addison-Wesley, Reading, MA, 1993.

[NP82] J. Nievergelt and F. P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM*, 25:739–747, 1982.

[Ove96] Mark H. Overmars. Designing the computational geometry algorithms library cgal. In M. C. Lin and D. Manocha, editors, *ACM Workshop on Applied Computational Geometry*, Philadelphia, Pennsylvenia, May, 27–28 1996. Lecture Notes in Computer Science 1148.

[SH76] M. I. Shamos and D. Hoey. Geometric intersection problems. In *Proc. 17th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 208–215, 1976.

4 Appendix

Program 1 (C++ Class Definitiones):

```
/*
   This structure definitions belong to the paper
   of W. Freiseisen and P. Pau:
   "A Generic Plane-Sweep for Intersecting Line Segment"
   draft version 22. July 1998
*/


struct SweepTraits {

    typedef NUMBERTYPE    NT;
    typedef POINT<NT>    Point;
    typedef SEGMENT<Point>    Segment;

    NT    x(const Point& p) const;
    NT    y(const Point& p) const;

    Point    source(const Segment& s) const;
    Point    target(const Segment& s) const;

    /* NT    slope(const segment& s) const;*/
    bool    is_slope_smaller(const segment& s1, const segment&
s2) const;
    NT    linear_function_value(const Segment& s, const &NT x)
const;


    /* Extensions */

    NT    x(Point* p) const
    {
        assert(p == NULL);
        return x(*p);
    }

    NT    y(Point* p) const
    {
        assert(p == NULL);
        return y(*p);
    }

    bool    is_slope_smaller(segment* s1, segment* s2) const
    {
        assert(s1 == NULL);
        assert(s2 == NULL);
```

```
        return is_slope_smaller(s1,s2);
    }
};


template<class I>
class Event {
public:
    typedef typename I::NT        NT;
    typedef typename I::Point    Point;
    typedef typename I::Segment    Segment;


    enum EventType {
          Intersection, Source, Target
          };


  Event( Point* pt ) :  _pt(pt) {}
  Point * point() const { return _pt; }

  virtual EventType type() const = 0;
  virtual Segment* segment(bool first = true) const = 0;
  virtual ~Event() {}

protected:
    Point *_pt;
};

template<class I>
class Source_Event :  public Event<I> {
public:
  Source_Event<I>(Point* pt, Segment* s) :  _s(s), Event(pt) {}
  EventType type() const { return Source; }
  Segment* segment() const { return _s; }
private:
    Segment *_s;
};

template<class I>
class Target_Event :  public Event<I> {
public:
  Target_Event<I>(Point* pt, Segment* s) :  _s(s), Event(pt) {}
  EventType type() const { return Target; }
  Segment* segment() const { return _s; }
private:
    Segment *_s;
};

template<class I>
```

```
class Intersection_Event : public Event<I> {
public:
  Intersection_Event<I>(Point* pt, Segment* s1, Segment* s2)
    : _s1(s1), _s2(s2), Event(pt) {}
  EventType type() const { return Intersection; }
  Segment* segment(bool first) const { return first ? _s1 : _s2;
}
private:
    Segment *_s1, *_s2;
};

template<class I>
class Event_Point_Schedule {
public:
    typedef Event<I>* EventPoint;

  bool insert(EventPoint e) { return _event_list.insert(e); }

  void remove(EventPoint e) {
    /* removes the event from the schedule and
     gives also the memory free (!!!)
   */
    if( !_event_list.remove(e) )
      return false;
    delete e;
    return true;
  }

  bool Before(EventPoint e1, EventPoint e2) const {
    /* see Definition 1 */
    if (x(e1.point()) < x(e2.point()) ) return true;
    if (x(e1.point()) > x(e2.point()) ) return false;
    if (y(e1.point()) < y(e2.point()) ) return true;
    if (y(e1.point()) > y(e2.point()) ) return false;

    if (e1.type()==SOURCE && e2.type()==TARGET)
        return true;
    if (e2.type()==SOURCE && e1.type()==TARGET)
        return false;
    if (e1.type()==SOURCE && e2.type()==INTERSECTION)
        return true;
    if (e2.type()==SOURCE && e1.type()==INTERSECTION)
        return false;
    if (e1.type()==INTERSECTION && e2.type()==TARGET)
        return true;
    if (e2.type()==INTERSECTION && e1.type()==TARGET)
        return false;

    if (e1.type()==SOURCE && e2.type()==SOURCE &&
        is_slope_smaller( e1.segment(), e2.segment() )
```

```
        )
        return true;
    if (   e1.type()==SOURCE
         && e2.type()==SOURCE
    && !is_slope_smaller( e1.segment(), e2.segment() )
        )
        return false;
    if (e1.type()==TARGET && e2.type()==TARGET &&
         !is_slope_smaller( e1.segment(), e2.segment() )
        )
        return true;
    if (e1.type()==TARGET && e2.type()==TARGET &&
         is_slope_smaller( e1.segment(), e2.segment() )
        )
        return false;

    return false;
  }

  bool operator<(EventPoint ev1, EventPoint ev2) const {
        return Before(ev1, ev2);
  }

 private:
    set<EventPoint> _event_list;
 };


 /* */
 template<class I>
 struct Line_Status_Element {

    typedef typename I::NT        NT;
    typedef typename I::Point    Point;
    typedef typename I::Segment    Segment;
      typedef Event<I>::EventType    EventType;

    Line_Status_Element<I>(const NT& x, Segment* s, Event<I>*
e)
      :  _x0(x), _seg(s), _ev(e)
    {
      _y0= I::linear_function_value(*_seg, _x0);
    }

    EventType type() const { return _ev->type(); }
    Segment* segment() const { return _seg; }

    NT y0(const NT& x) {
        if( _x0 == x ) return _y0;
        _x0= x;
```

```
        _y0= I::linear_function_value(*_seg, _x0);
        return _y0;
    }

    /* members */
    Event<I>    *_ev;
    Segment    *_seg;
    mutual NT _x0, _y0;
};


template<class I>
class Sweep_Line_Status {
public:

  bool insert( const Point& pt, Segment* seg) {
    _x= pt.x();
    return _status.insert( Line_Status_Element<I>(_x, seg) );
  }

  bool remove( Segment* seg) {
    return _status.remove( Line_Status_Element<I>(_x, seg) );
  }

  bool swap( const Point& pt, Segment* seg1, Segment* seg2) {
    return remove(seg1) && remove(seg2) &&
           insert( pt, seg1) &&  insert( pt, seg2);
  }

  bool below( const NT& x0,
              Line_Status_Element<I>& e1,
          Line_Status_Element<I>& e2) const
  {
    /* see definition 2 */
    return e1.y0(x0) < e2.y0(x0) ||
           ( e1.y0(x0) == e2.y0(x0) &&
             (e1.event() == Source || e2.event() == Source) &&
         !(is_slope_smaller( e1.segment(), e2.segment()) )
         );
  }


  bool operator<(const Line_Status_Element<I>& e1,
                 const Line_Status_Element<I>& e2) const
  {
    return below(_x, e1, e2);
  }

private:
    set< Line_Status_Element<I> > _status;
    mutual NT _x;
};
```
program end

Program 2 (The definition of *Event* class):

```
template <class I>
class Event
{
public:
  typedef enum
    {
      SOURCE=0,
      TARGET=1,
      INTERSECTION=2
    } Type_of_event;
  typedef typename I::Segment Segment;
  typedef typename I::Point   Point;
  typedef typename I::NT      NT;
  typedef typename I::FT      FT;

  typedef vector&lt;Segment&gt;::iterator
        Segment_iterator;

  Event();
  /* some constructors ... */

  void set_segment1(const Segment_iterator& s);
  void set_segment2(const Segment_iterator& s);
  NT x() const { return _point.x(); }
  NT y() const { return _point.y(); }
  Point  point() const { return _point; }
  Type_of_event  type() const { return _type; }
  FT slope() const { return _slope; }
  Segment_iterator segment() const { return _segment_1; }
  Segment_iterator segment1() const { return _segment_1; }
  Segment_iterator segment2() const { return _segment_2; }

private:
  Point            _point;
  Type_of_event    _type;
  FT               _slope;
  Segment_iterator _segment_1;
  Segment_iterator _segment_2;
};
```

program end

Program 3 (Implementation of the order `Before`):

```
template <class I>
struct point_before:  binary_function<I, I, bool>{
  bool operator () (const I& p1, const I& p2) const
  {
    if (p1.point().x()<p2.point().x()) return true;
```

```
    if (p1.point().x()>p2.point().x()) return false;
    if (p1.point().y()<p2.point().y()) return true;
    if (p1.point().y()>p2.point().y()) return false;
    if (p1.type()==I::SOURCE && p2.type()==I::TARGET)
            return true;
    if (p2.type()==I::SOURCE && p1.type()==I::TARGET)
            return false;
    if (p1.type()==I::SOURCE && p2.type()==I::INTERSECTION)
            return true;
    if (p2.type()==I::SOURCE && p1.type()==I::INTERSECTION)
            return false;
    if (p1.type()==I::INTERSECTION && p2.type()==I::TARGET)
            return true;
    if (p2.type()==I::INTERSECTION && p1.type()==I::TARGET)
            return false;
    if (p1.type()==I::SOURCE && p2.type()==I::SOURCE &&
        p1.slope()>p2.slope()) return true;
    if (p1.type()==I::SOURCE && p2.type()==I::SOURCE &&
        p1.slope()<p2.slope()) return false;
    if (p1.type()==I::TARGET && p2.type()==I::TARGET &&
        p1.slope()>p2.slope()) return true;
    if (p1.type()==I::TARGET && p2.type()==I::TARGET &&
        p1.slope()<p2.slope()) return false;
    return false;
  }
};
```
program end

Program 4 (The definition of *LineElement* class):

```
template <class I>
class Line_status_element :  public I::Segment
{
public:
  typedef typename I::Segment Segment;
  typedef typename I::Point   Point;
  typedef typename I::NT      NT;
  typedef typename I::FT      FT;

  Line_status_element();
 /* some constructors... */

  bool is_horizontal();
  bool is_vertical();
  FT slope();
  FT y_at_x(const typename FT& x);
  bool is_source(const Point& p);
  bool is_target(const Point& p);
};
```
program end