

[Segment Intersection & Point Inclusion]
Computational Geometry (CSc 635)

Jagdish Bhatta

Central Department of Computer Science & Information Technology
Tribhuvan University

Line Segment Intersection

Geometric Intersections:

One of the most basic problems in computational geometry is that of computing intersections. Instructions computation in 2D & 3D is basic to many application areas.

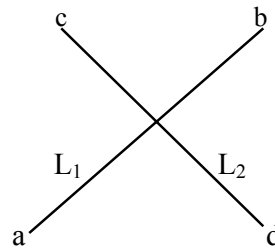
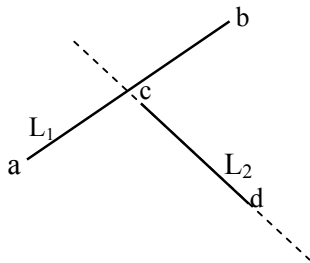
- In robotics & motion planning, it is important to know when two objects intersect for collision detection.
- In GIS, it is often useful to overlay two from a collection of line segments. So arising issue of intersection.
- In solid modeling, to build complex objects, one can apply various Boolean operations (intersection, union) on simple objects. In order to perform these operations, most basic step is determining the points where boundaries of two objects intersect.

Line Segment Intersection:

The problem is that given n -line segments in the plane, report all points where a pair of line segment intersect.

Segment Intersection Detection

1st Approach:



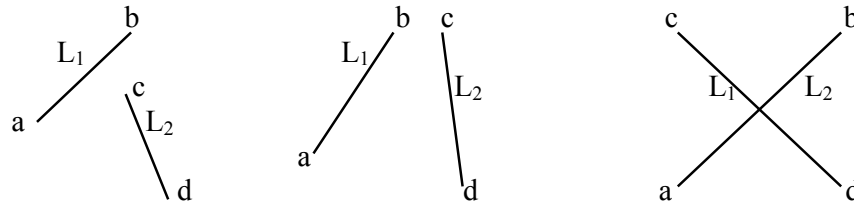
- Let L_1 (a, b) & L_2 (c, d) be any two line segments.
- Compute the equation of line through the segments L_1 & L_2 . Let these equations be l_1 & l_2 .
- Solve the equations l_1 & l_2 to get point of intersection of L_1 & L_2 .

Let $P(x, y)$ be the point. Now, check if P lies on both the segments L_1 & L_2 . If so, clearly L_1 & L_2 intersect otherwise they are not intersecting.

What if slope is 0 or ∞ ? (For vertical or horizontal lines)

Next approach (Using idea of left turn & right turn)

- The method of left turn and right turn is an efficient method for detecting the segment intersection.
- In this approach no any division operation is involved so fast to detect the intersection.

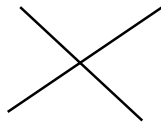


- Let $L_1(a, b)$ & $L_2(c, d)$ are two line segments. Then, L_1 & L_2 do not intersect; if abc & abd are of same type of turn.
 - if cda & cdb are of same type of turn.
 So, we can compute it in terms of area as;

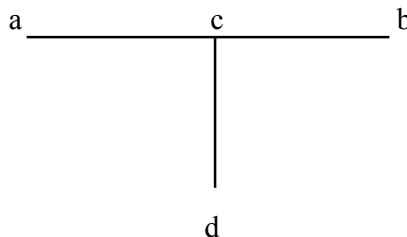
$$\left. \begin{array}{l} A(a, b, c) * A(a, b, d) > 0 \\ A(c, d, a) * A(c, d, b) > 0 \end{array} \right\} \text{ Same type of turn}$$
- The two segment intersect each other – if abc & abd are of opposite turn
 - If cda & cdb are of opposite turn.
 i.e.; $A(a, b, c) * A(a, b, d) < 0$
 $A(c, d, a) * A(c, d, b) < 0$

Proper Intersection

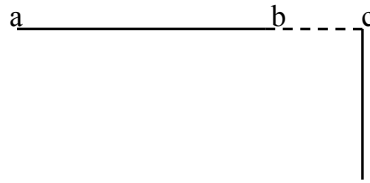
When two segments intersect at a point interior to both, i.e.; no three of the four end point are collinear.

***Improper Intersection***

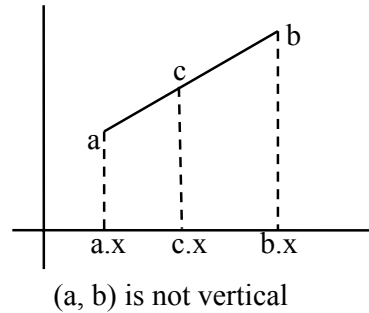
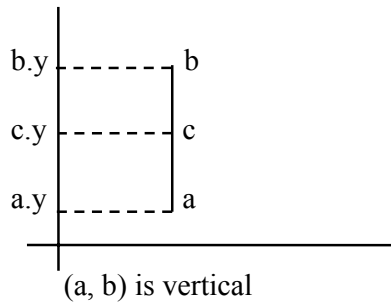
- This is a special case of intersection between two segments. Improper intersection occurs when one end point of one segment lies on the other closed segment.
- This is possible only when three of the end points are collinear.



- But only co-linearity is not sufficient to argue the improper intersection.



Betweenness: -



For line segments L1 (a, b) & L2 (c, d), we can check betweenness of one end point on another segment; say whether c lies on segment (a, b).

For this;

- If (a, b) is not vertical, then c lies on (a, b) iff x-coordinate of c lies/falls in the interval determined by the x-coordinate of a & b.

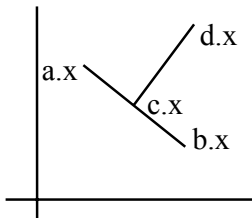
i.e. $a.x \leq c.x \leq b.x$ or

$a.x \geq c.x \geq b.x$

- If (a, b) is vertical, similar check on y-coordinates determines betweenness of c.

i.e.; $a.y \leq c.y \leq b.y$ or

$a.y \geq c.y \geq b.y$



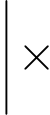
Plane Sweep Algorithm for Segment Intersection

Let $S = \{S_1, S_2, S_3, \dots, S_n\}$ denote the line segments whose intersection we wish to compute. Now with plane sweep method, it involves following main elements;

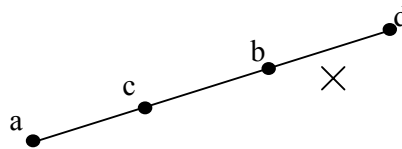
- Sweep Line: - A vertical line l_1 called the sweep line is used to sweep over the line segments from left to right. (One may use horizontal sweep line from top to bottom).
- Events: - Event is any end point or intersection point.
 - Endpoint events: - these events, sweep line encounters an end point of a line segment. There can be at most $2n$ endpoint events. These can be presorted before sweep runs.
 - Intersection events: - at these events, sweep line encounters an intersection point of two line segments. These events will be discovered as the sweep executes.
- Event Updates: - when an event is encountered, the data structure is updated accordingly that is associated with the event.
 - E.g. beginning end point, insert segment into segment list
 - At end of segment, delete segment from segment list
 - At intersection, interchange order of line segments.
- Detecting Intersection: - whenever two line segments become adjacent along the sweep line, we will check whether they have an intersection occurring to the right of the sweep line. If so, add this new event into event list assuming that it has not already been added. (We are able to identify all intersections by looking only at adjacent segments in the sweep line status during the sweep).
- Data Structures
 - Event Queue: - This holds set of future events, sorted according as x-coordinate. Each event point contains the auxiliary information of what type of event is (segment endpoint or intersection) & which segments are involved.
 - Operations supported
 - ⇒ Insert an event
 - ⇒ Extract minimum event
 - Heap or balanced binary tree
 - $O(\log n)$
 - Sweep line status: - Holds pointers to the line segments, stored in increasing order of y-coordinate along the current sweep line.
 - Operations Supported,
 - ⇒ Insert a line segment

- ⇒ Delete a line segment
- ⇒ Swap position of two line segments.
- ⇒ Determine predecessor & successor of the item.
- ⇒ $O(\log n)$

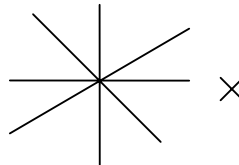
▪ Assumptions: -

- No line segment is vertical 

- If two segments intersect, they intersect in a single point (i.e. non-collinear)



- No three line segments intersect in a common point.



- At any point of a sweep line, all event to its left are supposed to be preprocessed & it remains only to right for further process

Algorithm:

$$S = \{S_1, S_2, S_3, \dots, S_n\}$$

- (1) Insert all the end points of the line segments of S into the event queue. The sweep line status is empty. (Algorithm proceeds by adding & deleting discrete events from this queue until it is empty).
- (2) While the event queue is non-empty, extract the next event in the queue. Following three cases will occur, depending on type of event;

⇒ Segment left endpoint: -

- Insert this segment into the sweep line status.
- Test/check for intersection, to the right of the sweep line, with segments immediately above & below.
- If the intersection point is found, insert it into event queue.

⇒ Segment right endpoint: -

- Delete this line segment from sweep line status.

- Test for intersection, to the right of the sweep line, between the segments immediately above & below.
- Insert point if found into the event queue.

⇒ Intersection point:

- Report the point.
- Swap two relevant segments in the sweep line status.
- For the new upper segment, test it against its predecessor for an intersection, to the right of sweep line.
- Insert point (if found) into the event queue, such that it is not reported earlier.
- Similarly, for new lower segment, check it against its successor for intersection.

Analysis:

Total events: $(2n + k)$ where k may go up to n^2 in worst case.

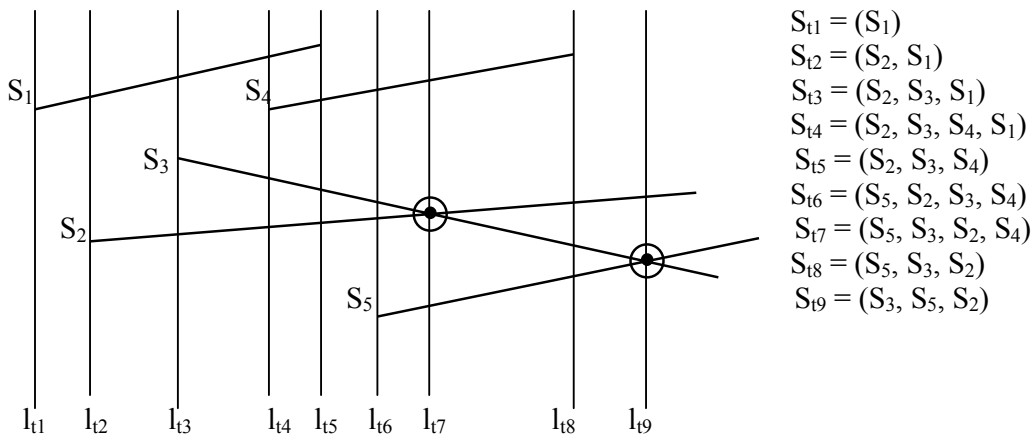
- Each event involves constant no. of access or operations to sweep line status or event queue. Since each such operation takes $O(\log n)$ time. (Since we are using binary tree).
- So, total time spent processing all the events is,

$$O((2n + k) \log n)$$

$$\approx O(n \log n)$$

⇒ Time to maintain data structures overcome the intersection computation time.

E.g.



Point Inclusion in a simple polygon

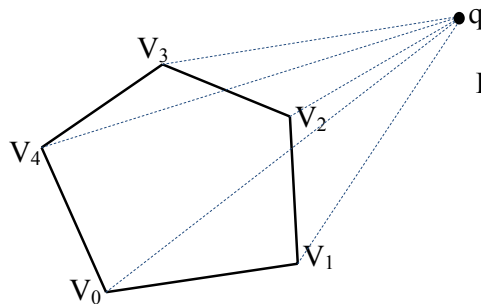
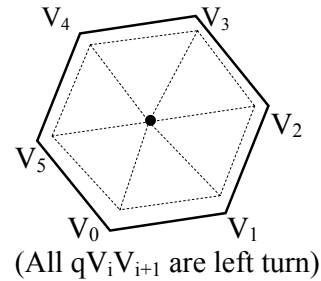
(Use: Polygon filling, determining intersection of multiple polygons)

Problem: - Given a point q and a polygon P of n -vertices, determine whether q lies inside P or no?

Convex Polygon:

If P is a convex polygon, then for each edge of P , perform the test left turn/right turn.

- If q is inside the polygon then for each edge (V_i, V_{i+1}) of P , qV_iV_{i+1} has the same type of turn. So left turn (q, V_i, V_{i+1}) test can be performed.
- If q is outside the polygon, then for each edge of polygon the test is not true for all edges. Some of them will be of opposite turn.



Here, qV_0V_1 – left turn
 qV_1V_2 – right turn
 qV_2V_3 – right turn
 qV_3V_4 – left turn
 qV_4V_0 – left turn

Here, since all turns are not of same type so the query point is outside the polygon.

Non-Convex Polygon:

For non-convex polygon, just left turn/right turn test may not be sufficient to decide whether the query point is inside or outside of the polygon. Other possible tests should be done for non-convex cases.

Winding Number Method:

Here, idea is to compute the winding number of the query point with respect to the polygon. If the winding number is non-zero, the point lies inside the polygon otherwise not.

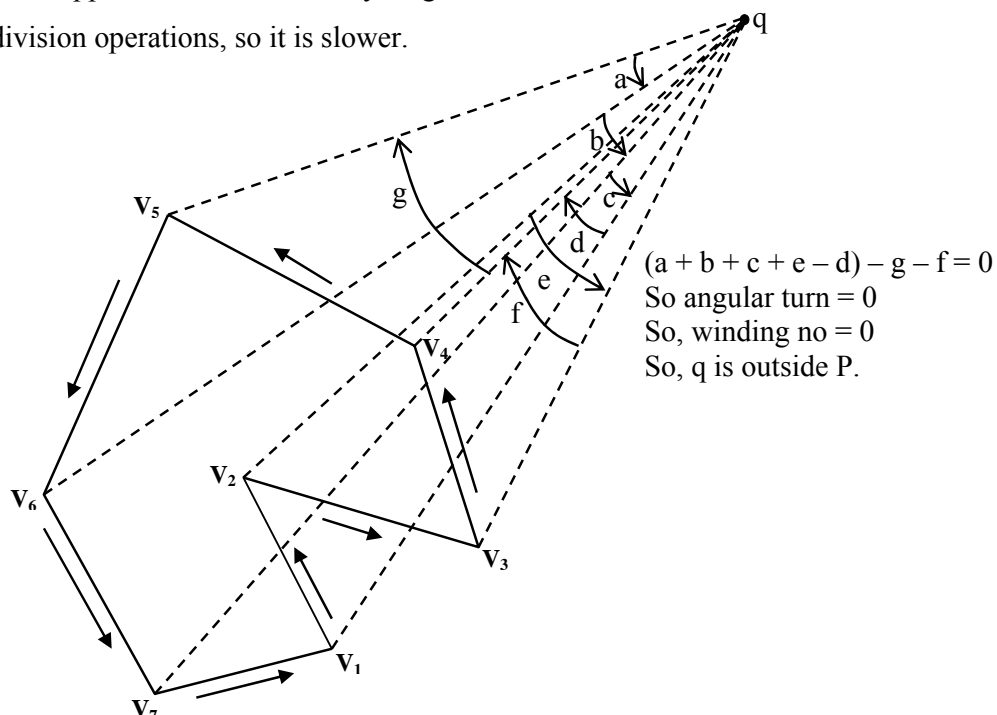
The winding number of a point q with respect to a polygon P is the number of revolutions the boundary of the polygon makes about q . i.e. total signed angular turn divided by 2π .

To compute winding number, imagine we are standing at the query point q , watching a point p , complexity traverse the boundary of P in counterclockwise always facing towards the point p , sum up the angles subtended by each side of the polygons. Clearly, the sum of the angles divided by 2π will result the winding number.

If q is inside P , we should turn full circle, 2π angle. So winding no will be non-zero.

If q is not inside P , total angular turn will be zero; i.e. winding no. = 0.

This approach involves costly trigonometric functions that are cosine as well as division operations, so it is slower.



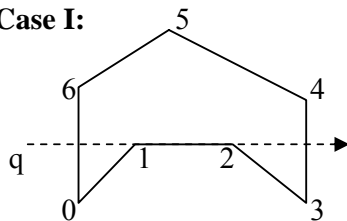
Ray-Crossing Method:

In this method, to determine whether a query point lies inside or outside of the polygon we will do following:

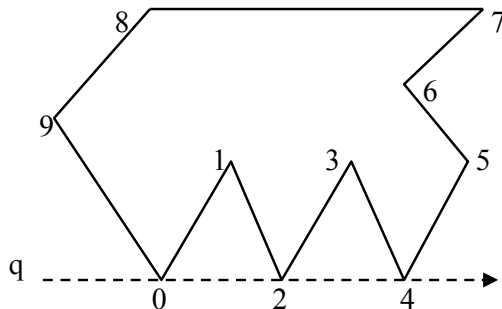
- Cast a ray from the query point q in an arbitrary direction (say positive x direction).
- Determine whether the number of crossings/intersections that the ray makes with the polygon P is even or odd.
- If the number of crossing is even, q lies outside the polygon P otherwise, if it is odd, q lies inside the polygon.

This method is often known as even/odd test for point inclusion in a simple polygon. But with this approach, there are number of “degenerated cases” that we have to consider. These are the special cases of intersections, when ray intersects the edges at one of its end point or coincides with entire edge.

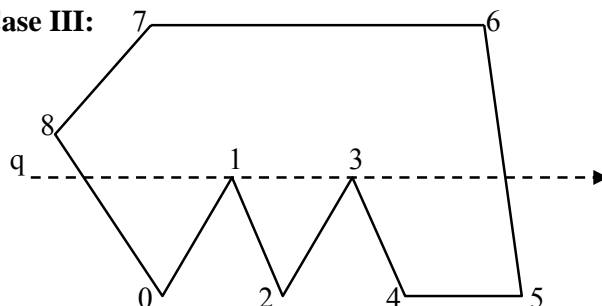
For these cases, special tests should be performed as;

Case I:

Here, for edge (1, 2), the ray and edge coincide, so the count is ignored for this case.

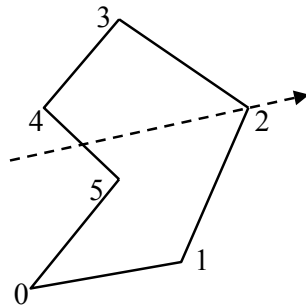
Case II:

Here, ray crosses with (9, 0), (0, 1), (1, 2), (2, 3), (3, 4), (4, 5). Count is taken for this case. (So, consider upper end points as count.)

Case III:

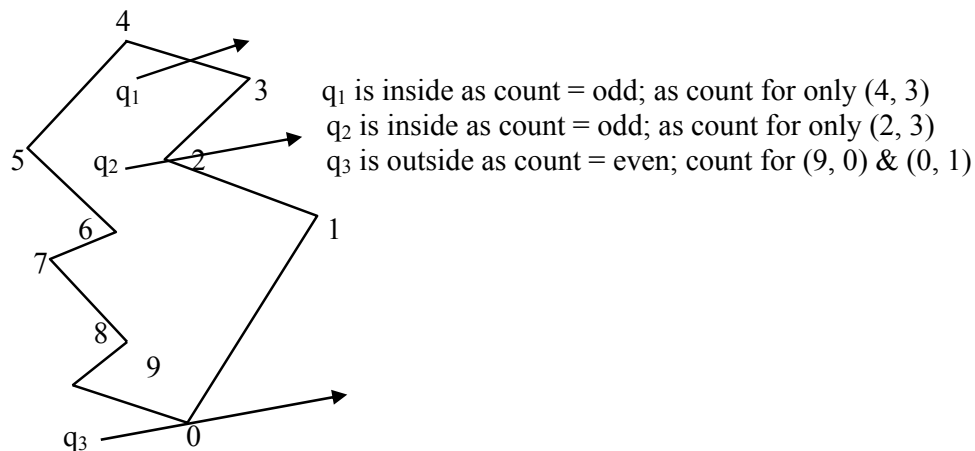
Here, ray crossing with (0, 1), (1, 2), (2, 3), (3, 4) are ignored for the count since ray crosses upper end of the edge. (So, ignore lower end points so do not count.)

Case IV:



Here, ray crosses (1, 2) in upper end and (2, 3) in lower end. So, count intersection for upper end and ignore for lower one.

Example:



General Algorithm:

Given a polygon P with n -vertices and a query point q , let R be the long horizontal ray through q in arbitrary direction (say +ve x)

Now, the pseudo code is as follows:

```

int count = 0
for i = 0 to n-1
{
    if edge[i] is not horizontal
    {
        if edge[i] intersects R at any of its point, except upper extreme
        point (i.e. lower external point)
        count ++
    }
}
  
```

```
if count is odd
    q is inside P
else
    q is outside P.
```

Analysis:

Clearly, the algorithm runs for n -times so complexity is $O(n)$.