## Line Segment Intersection

The intersection problem:

Given 2 object (in 2/3D), find out whether they intersect, and if so, what is the point of intersection.

The objects may be complex, like circles. The complexity will be a function of a dimension.

It is common to break down complex shapes, like polygons, into simpler shapes (like into sets of edges / vertices).

Given $n$ line segments in the plance, report all points where a pair of line segments intersect.

### Naïve Algorithm

This can be done naively in $\binom{n}{2}$ time – check each pair in the set (provided we can check intersection of 2 lines in constant time). This is $O(n^2)$.

We would like an output-sensitive algorithm, that is an algorithm with complexity based on the number of intersection points.

### Lower bound

The lower bound is $O(n \lg n + k)$, where $k$ is the number of intersection output.

See lecture notes for reduction, proving it cannot be $o(n \lg n)$.

### Primitive operation:

Given segments $ab, cd$ we can detect whether they intersect in constant time:

$p(s) = (1 - s)a + sb, s \in [0,1]$

$q(t) = (1 - t)c + td, t \in [0,1]$

An intersection occurs if $p(s) = q(t)$ for some $s, t$ in the range above (if they are not in $[0,1]$ it means there is an intersection of the lines on which $ab, cd$ reside, but not on the segments themselves).

This function is given from this point on as a known constant operation.

### Plane Sweep Algorithm

Note: in class the direction of movement of the sweep line is presented as horizontal, not vertical as in the book.

Let $S = \{s_1, s_2, \dots, s_n\}$ be a set of line segments. The main elements of *plane sweep*:

- **Sweep line $l$**: simulating the sweeping along the y-axis from $+\infty$ to $-\infty$.
- Line segments are stored in order they intersect $l$.
- **Events**: event points are those in which $l$ has a content change (with respect to what segments intersect it and in what order).

Event points:

- Endpoint events: known in advance, the points in which $l$ intersects a new segment / stops intersecting it.
- Intersection events: where $l$ encounters points of intersection of 2 segments.

When 2 segments are adjacent on $l$, there is a chance they intersect. We hold a binary search tree to hold the order of segments currently intersecting $l$, and this data structure will be updated along the algorithm.

When a new segment is encountered: we need to check if it intersect with its predecessor and successor in the binary tree.

Assumptions:

- No segments are vertical. This can be implemented by giving a minor $\epsilon$-slope with respect to all other segments.
- No two segments intersect in more than 1 point.
- There are no more than 2 segments that intersect at any intersection point.

Detecting intersections:

An intersection can happen iff at some point of time they become adjacent with respect to $l$. We need to show this condition is sufficient:

**Lemma**:

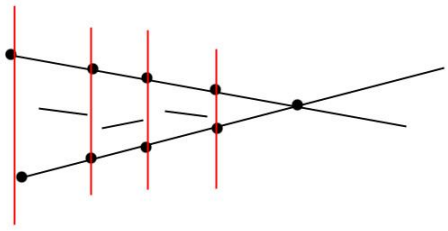
Given two segments $s_i$ and $s_j$ , which intersect in a single point $p$ (and assuming no other line segment passes through this point) there is a placement of the sweep line prior to this event, such that $s_i$ and $s_j$ are adjacent along the sweep line (and hence will be tested for intersection).

1) Event queue:

Will hold the set of future events sorted according to increasing x-coordinate. Each event will be identified whether it is a start/end endpoint or an intersection point. Should support:

- Insertion of new events
- Extracting the minimum

We will use a priority queue. When we insert to the priority queue we need to make sure the same event doesn't get inserted more than once (hence the need of both x and y coordinates of each event point inserted). An example where this might happen:



At each of the red lines the two long black lines become adjacent w.r.t. $l$ again and again – and their intersection point is checked every time, resulting with the same event.

Storing events in order: since two event points may have the same x coordinate, we need to sort all points lexicographically, by $(x, y)$ order (sort by x, and if it equals, sort by y).

2) Sweep line status:

A data structure that will hold the segments that intersect $l$ in order in which they intersect it. We use a binary search tree. The relative order between segments stays the same until an intersection point of 2 points, in which their order will change and the status will need to be updated.

Let a segment have the line equation $y(x) = mx + b$, this is actually the **key** for the status binary tree.

As $l$ is moving, we can plug in its $x$ value into any of the equations stored in the status tree, to know the actual coordinate of that particular segment at that location of $l$. This has to be done at every event that is encountered.

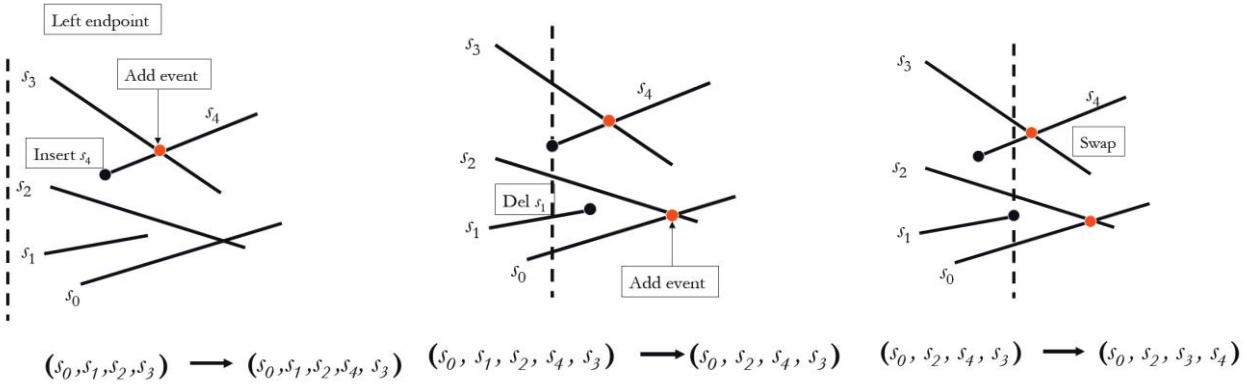
Operations that are needed to be supported:

- Delete a line segment
- Insert a line segment
- Swap positions between two line segments – right after $l$ passes their intersection point.
- Determining the predecessor and successor of any segment in the tree.

Using a balanced binary search tree will derive a $O(\lg n)$ time for any such operation.

The algorithm:

- Insert all endpoints of all segments in $S$ as events in the queue. The status is initially empty.
- While the Q is not empty, extract-min:
  - **The event is a segment left endpoint**: insert the segment into the status, based on the y coordinate at the current position of $l$ (in order with all segments currently in the status and their value at the current y position).
    Then check intersections of that segment with its pred and succ in the status.
  - **The event is a segment right endpoint**: delete the segment from the status, and for its previous pred and succ apply intersection tests, each with its pred and succ.
  - **The event is an intersection point**: swap the two line segments in order along the sweep line, and for the new upper and lower apply intersection tests, each with its new pred and succ.



Running time:

The running time is dominated by the time it takes updating the event priority queue and status binary tree data structures.

There are at most $n$ elements intersecting $l$, thus any operation will take $O(\lg n)$ in the status tree.

Since all duplicate events are disregarded, the size of the queue will be at most $2n + k - 2$ endpoint events for each segment + $k$ intersection events, so each operation: $O(\lg(2n + k))$, and since $k \leq n^2$ this is $O(\lg n^2) = O(\lg n)$.

For every event there are constant accesses to any of the data structures. Therefore the total running time is:

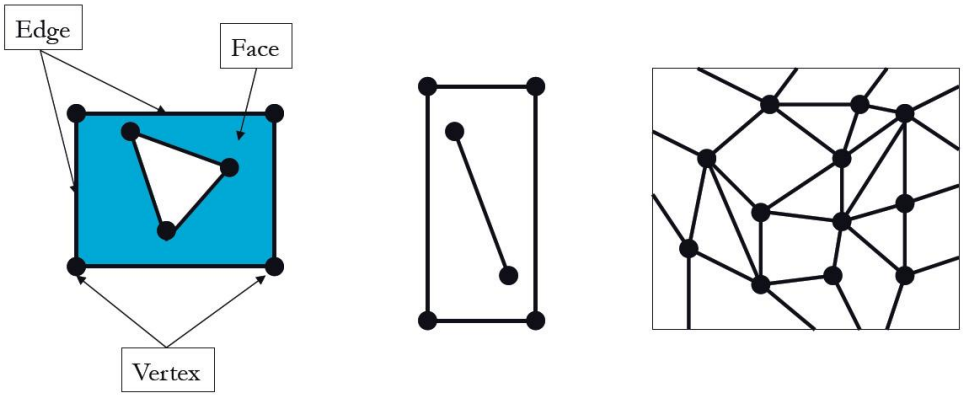
$$O\big((2n + k)\lg n\big) = O(n \lg n + k \lg n)$$

## Planar Graphs, Polygons and Art Galleries

We are usually interested in intersections of more complex shapes than line segments.

Planar Straight Line Graph (PSLG): (or planar subdivision):

- A graph in the plane with straight line edges that can intersect only at endpoints.
- The graph subdivides the plane into regions. Components:
  - 1-D vertices
  - 2-D edges
  - 3-D faces
- There will always be an unbound face.




Euler's Condition:

In the 2-D space and a given subdivision with $V$ vertices $E$ edges and $F$ faces, the formula states:

$$V - E + F = 2$$

For any subdivision, also referred as Euler's characteristic.

In general for an arbitrary genus with $g$ handle: $V - E + F = 2 - 2g$ (in $\mathbb{R}^2$ $g = 0$)

If the graph is disconnected with $C$ connected components: $V - E + F - C = 1$

**Theorem**

A planar graph with $V$ vertices has at most $3(V - 2)$ edges and at most $2(V - 2)$ faces – asymptotically, $3n$ edges and $2n$ faces.

Proof:

We can assume any face is a triangle, otherwise we can add more edges to triangulate any face. Assume we have $F$ faces and $E$ edges, and we triangulate to achieve $F' \geq F, E' \geq E$ faces and edges, respectively.

Observations:

- Every edge is adjacent to 2 triangles (faces)
- Every face is adjacent to exactly 3 edges

Therefore $2E' = 3F' \Rightarrow E' = \frac{3}{2}F'$

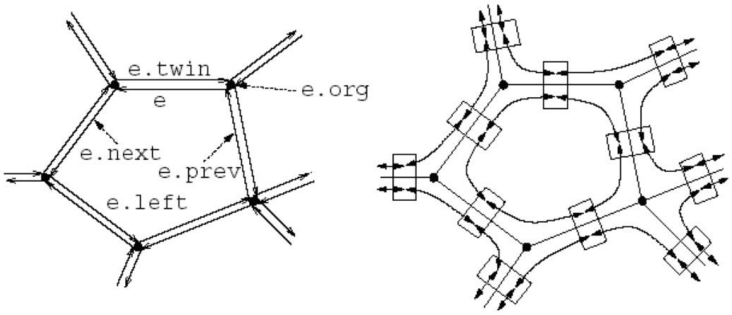
Plugging this into Euler's formula: $V - E' + F' = 2 \Rightarrow V - \frac{3}{2}F' + F' = 2 \Rightarrow F' = 2(V-2) \Rightarrow F \leq 2(V-2)$

And in a similar process we will get $E \leq 3(V-2)$.

Doubly-connected edge list:

A PSLG will be represented in a DCEL:

- A vertex stores coordinates, pointer to one of its incident edges for which it is its origin.
- Each edge is split into two oppositely directed edges (twins) each pointing to the origin of the other.
- Each edge also points to its **left** face
- Each edge holds a pointer to **next** and **prev** edges.



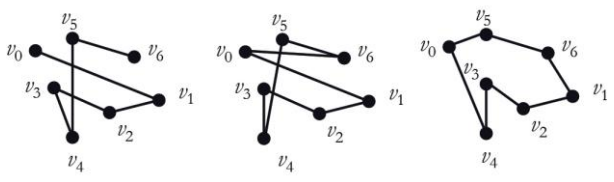

DCEL Alternative view

- Face list, for each face we store one edge (from that we can get the boundary of the face).

Simple polygons:

A *polygonal curve* is a finite sequence of consecutive line segments (edges) joined end to end.

It is closed if $v_0 = v_n$ (its vertices), and it is simple if it does not self-intersect.




Polygonal curve Closed Curve Simple polygon

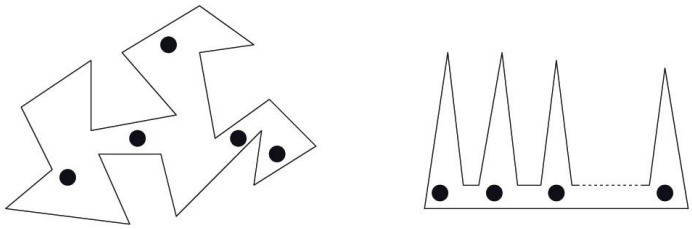
**The art gallery problem**

For two points $x, y$ in a polygon $P$, we say they see each other if $xy$ lies entirely within the interior of $P$.

The problem: find the number of points we can place inside the polygon $P$ (represented with $n$ vertices) such that all edges are visible from the inside by at least one of the points (analogy: need to place guards within an art gallery).

If the polygon is convex, the answer is simply 1. It gets more interesting if the polygon is not convex.

The right value is: $\left\lfloor \frac{n}{3} \right\rfloor$, and it is based on polygon triangulation, dual graphs and graph coloring.



**Art-Gallery Theorem**

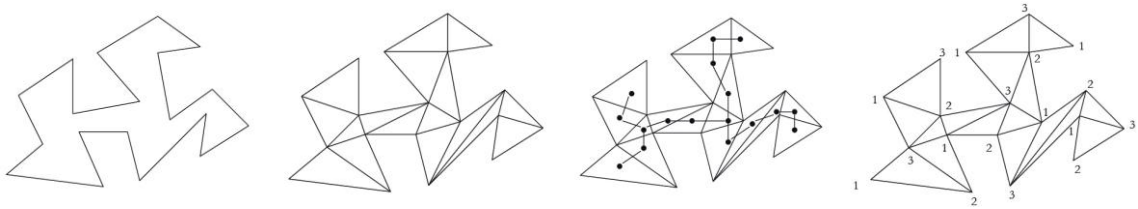
Given a polygon with $n$ vertices, there exists a guarding set with at most $\left\lfloor \frac{n}{3} \right\rfloor$ guards.

Lemma 1: every simple polygon with $n$ vertices has a triangulation with $n-3$ diagonals and $n-2$ triangles (in our case – faces)

Proof: any $n$-vertex polygon ($n \geq 4$) has a diagonal, and it can break it into 2 polygons with $m_1, m_2$ vertices that share 2 vertices, therefore $m_1 + m_2 = n + 2$, thus there are $(m_1 - 2) + (m_2 - 2) = n - 2$ triangles.

Lemma 2: given a triangulation graph $T$ of a simple polygon, it is 3-colorable.

Proof: for every planar graph G there is another planar graph called its **dual**. In the dual graph, the vertices are the faces of G and an edge exists between faces that have a shared edge in G. Triangulation, which is a planar graph, will create a tree in the dual. A cycle will happen only if G has a hole in it (but we started with a simple polygon, therefore there will not be a hole). Note that any triangle removed still leaves a tree. We recursively remove triangles until we reach 1 triangle, and then it is 3-colorable. We then add triangles back, and each addition we color the added vertex, getting a 3-coloring.



Theorem proof:

When we 3-color the graph, each color will be used at most $\frac{n}{3}$ times. Therefore we can pick a color, and pick its $\frac{n}{3}$ nodes and there place our guards.

Running time:

Linear: The triangulation is linear ($n-3$ triangles). Forming the dual graph is linear and everything is linear.