Computational Geometry

Chapter 2

Basic Techniques

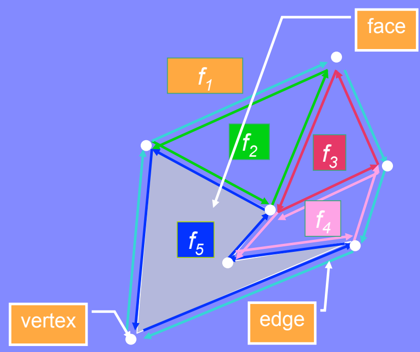
12.

On the Agenda

- The DCEL Data Structure
- Line Segment Intersection
- Plane Sweep
- Euler's Formula

22.

Doubly Connected Edge List - DCEL



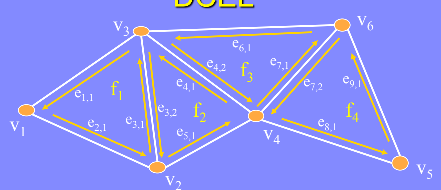
32.

DCEL

- Record for each face, edge and vertex:
  - Geometric information
  - Topological information
  - Attribute information
- aka Half-Edge Structure
  - 3 arrays - Vertices, Edges, Faces, (V,E,F)
  - Coordinates are stored only at V,
  - Avoid data duplication
  - Common operation need to support: Traversing along a line, and find its intersections with all edges and faces)

42.

DCEL

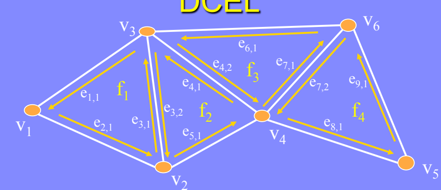


Vertex	coordinate	IncidentEdge
v_1	(x_1,y_1,z_1)	$e_{2,1}$
v_2	(x_2,y_2,z_2)	$e_{5,1}$
v_3	(x_3,y_3,z_3)	$e_{1,1}$
v_4	(x_4,y_4,z_4)	$e_{7,1}$
v_5	(x_5,y_5,z_5)	$e_{9,1}$
v_6	(x_6,y_6,z_6)	$e_{7,2}$

face	edge
f_1	$e_{1,1}$
f_2	$e_{5,1}$
f_3	$e_{4,2}$
f_4	$e_{8,1}$

52.

DCEL

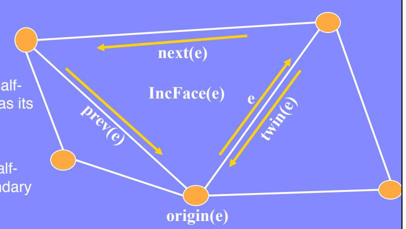


Half-edge	origin	twin	IncidentFace	next	prev
$e_{3,1}$	v_2	$e_{3,2}$	f_1	$e_{1,1}$	$e_{2,1}$
$e_{3,2}$	v_3	$e_{3,1}$	f_2	$e_{5,1}$	$e_{4,1}$
$e_{4,1}$	v_4	$e_{4,2}$	f_2	$e_{3,2}$	$e_{5,1}$
$e_{4,2}$	v_3	$e_{4,1}$	f_3	$e_{7,1}$	$e_{6,1}$

62.

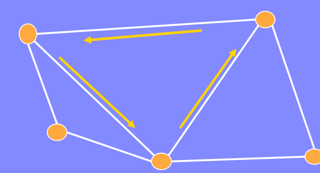
DCEL

- Vertex record:
  - Coordinates
  - Pointer to one half-edge that has v as its origin
- Face record:
  - Pointer to one half-edge on its boundary
- Half-edge record:
  - Pointer to its origin: origin(e)
  - Pointer to its twin half-edge: twin(e)
  - Pointer to the face it bounds: IncidentFace(e) (face lies to left of e when traversed from origin to destination)
  - Next and previous edge on boundary of IncidentFace(e): next(e), prev(e)



DCEL

- Support for:
  - Walk around boundary of given face
  - Visit all edges incident to vertex v (how ?)
- Queries:
  - Most queries are O(1)

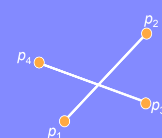


DCEL

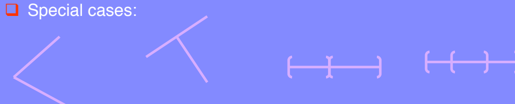
- Want to
  - Walk around the boundary of a given face of a polygon
  - Access a face from an adjacent one
  - Visit all the edges around a given vertex
- DCEL
  - Geometric structures combined by polygonal faces, edges and vertices
  - Linear space representation
  - Allow easy retrieval of data

Line Segment Intersection

- **Theorem:** Segments (p_1,p_2) and (p_3,p_4) intersect in their interior iff p_1 and p_2 are on different sides of the line p_3p_4 and p_3 and p_4 are on different sides of the line p_1p_2.
- This can be checked by computing the orientations of *four* triangles. Which ?
- Computational robust, but computationally costly.



- Special cases:



Computing the Intersection

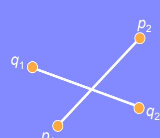
$$p(t) = p_1 + (p_2 - p_1)t \qquad 0 \le t \le 1$$
$$q(s) = q_1 + (q_2 - q_1)s \qquad 0 \le s \le 1$$

Question: What is the meaning of other values of *s* and *t* ?

Solve (2D) linear vector equation for *t* and *s*:

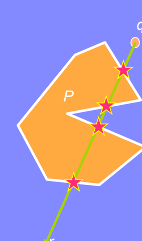
$$p(t) = q(s)$$

check that $t \in [0,1]$ and $s \in [0,1]$



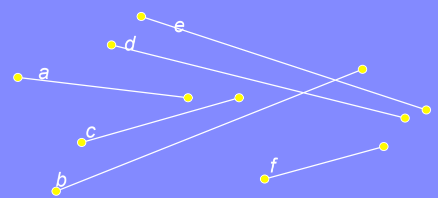
Point in Polygon

- Given a polygon *P* with *n* sides, and a point *q*, decide whether $q \in P$.
- Solution A: Count how many times a ray *r* originating at *q* intersects *P*. Then $q \in P$ iff this number is odd.
- Complexity: O(*n*)
- **Question:** Are there special cases ?



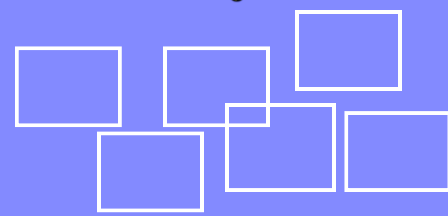
Line-segment intersection

Given n line segments, two questions arise:
- does any pair intersect? (studied in this class)
- report all intersections



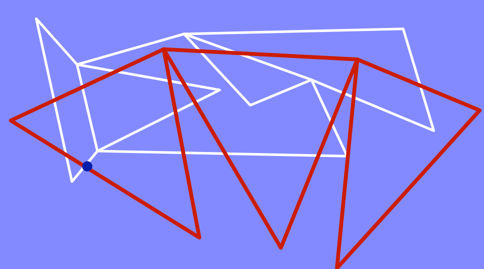
Obvious algorithm: $O(n^2)$ – checking all pairs.
Line-sweep – $O(n \log n)$ time determine if there is an intersection.

Applications 1: Checking VLSI design correctness



Need to decide if two components intersect (bug in design)

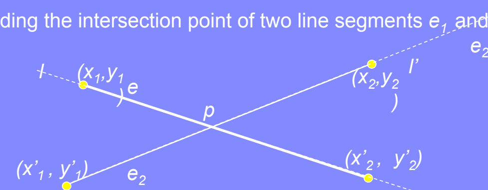
Applications 2: Finding all junctions and/or intersection points



Applications: Map overlaying:
compute all points at which a road crosses a river.

Computing intersection of two segments

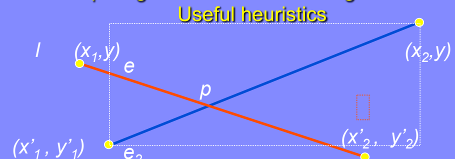
Finding the intersection point of two line segments e_1 and e_2



Let l_1 be the line containing the segment e_1.
Let l_2 be the line containing the segment e_2.
$l=\{(x,y) | y = a x + b\}$, where $a=(y_2-y_1)/(x_2-x_1)$
$l'=\{(x,y) | y = a' x + b'\}$

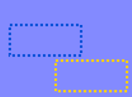
1. Find the intersection point p of the **lines** (solving a linear system with two equations).
2. Find if p lies on both segments

Computing intersection of two segments- cont
Useful heuristics



First check if their boounding boxes of the segments intersects.

Bounding box-the axis-parallel that has the endpoints of the segment as a pair of diagonal vertices.



A quick operation that does not involve **division.**

Leftover from Data-structures

Given a set S of numbers, a standard balanced search tree supports

insert(x,S) , delete(x,S) , find(x,S)

Each in $O(\log n)$ time (where $n=|S|$)

It can also support the operation succ(x,S), defined as finding the smallest element of S which is strictly larger than S.

Examples $S=\{10,20,30\}$;

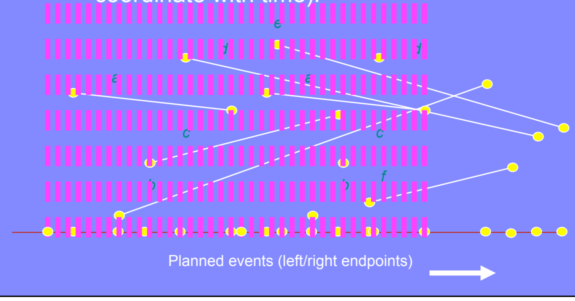
succ$(-30,S)$=10,
succ$(10,S)$=succ$(12,S)$=20,
succ$(30,S)$=succ$(40,S)$= UNDEFINED.

Leftover from Data-structures

```
Succ(pNODE p, float x){
        p = root;
        x_tmp= INFINITY ;  /* x_tmp – temporally value */
        while( p ≠ NULL ) {
                if ( p->key ≤ x )  p=p->right;
                else {
                        x_tmp = min(x_tmp, p->key) ;
                        p = p->left ;
                }
        }
        return x_tmp;
}
```

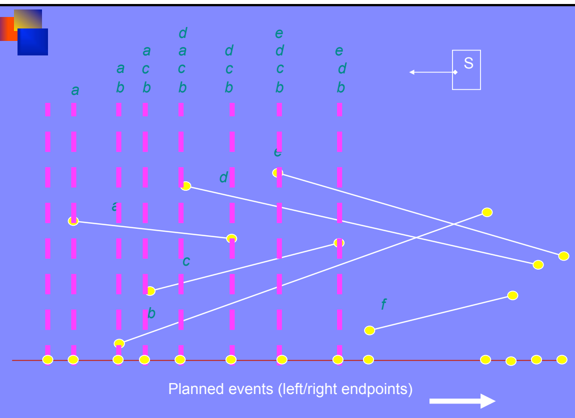
Sweep-line algorithm

Sweep a vertical line from left to right
The line “knows” which segment it intersect and at which order (conceptually replacing *x*-coordinate with time).

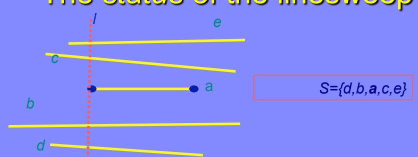


Sweep-line algorithm

- Sweep a vertical line from left to right (conceptually replacing *x*-coordinate with time).
- Maintain the **status** - a dynamic set *S* of the segments that intersect the sweep line, ordered (tentatively) by *y*-coordinate of intersection.
  - (so the lowest segment appears first one the list)
- The status is changed only when
  - new segment is encountered (**left** endpoints),
  - existing segment finishes (**right** endpoint)
  - ***Event points*** are therefore segment endpoints.



The *status* of the linesweep



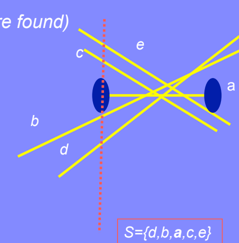
The *status* *S* is the list of segments that linesweep *l* intersects (in the order from bottom to top).

Definition: an ***event*** happens when *l* start/stop intersecting a segment.

Note: the status is not changed between events, so *l* can jump from an event to the next event.

Left endpoint event:

- *For a **left** endpoint of segment s:*
  - *Add segment s to the status S.*
  - *Check for intersection between s and its **neighbors** in S.*
- *(Will later explain how the neighbors are found)*



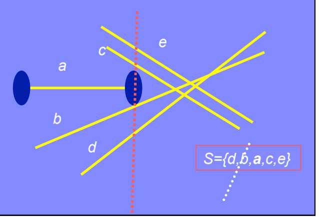
Example: ***a*** is checked for intersection with ***c*** and intersection with ***b***.

Right endpoint event

For a **right** endpoint of segment *s*:

- Delete segment *s* from dynamic set *S*.
- Check for intersection between neighbors *s* and its **neighbors** in *S*.

Example: *c* is checked for intersection with *b*.



Theorem: If there is an intersection point, the algorithm finds it.

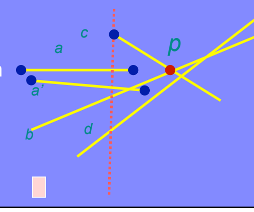
Proof: Let *p* be the leftmost intersection point.

Consider the last event before (to the left of) *p*, at which *c* or *b are born*

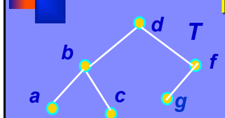
If they are not neighbors on *l*, it is because another segment, say *a* separates between them.

But then either *a* has a right endpoint to the left of *p* and then *c* and *b* become neighbors.

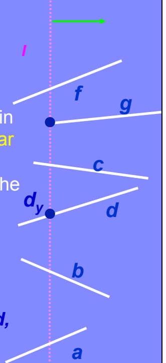
Or *a'* intersects *b* or *c* at a point to the left of *p*, contraditction to *p* be the leftmost point.



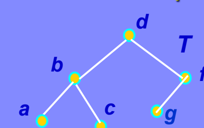
Maintaining the status *S*

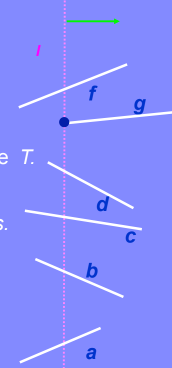


- We maintain *S* - the list of segments that intersect *l* in a sorted search tree *T*, sorted by the order they appear along *l*.
- When we insert a new segment *g*, we compare *y'*, the *y*-coordinates of intersections of segments with *l*.
- Example: Since *d* is the root of *T*, we compute the intersection point of *l* with *d.* call it d_y.
- We compare *y'* and d_y and deduce that *g* is above *d*, so it should be inserted into the right subtree.
- Continue recursively.



Operations on the tree





Insert(*T,s*) - Insert the segment *s* into the Tree *T*.
Delete(*T,s*)

Above(*T,s*) - Find the segment just below *s*.
Example **Above**(*T,b*) = *c*.
(successor oprtaion in *T*)
Below(*T,s*)– Analogous operation (predecessor)

AnySegmentsIntersect(S) - pseudocode

```
Create an empty set T
Sort endpnts of the segments in S from left to right.
FOR each endpnt p in the sorted list of end points do {
  IF p is the left endpnt of a segment s then Insert(T,s);
     IF {Above(T,s) exist and intersects s} or
        {Below(T,s) exists and intersects s} Then
   Return TRUE /*found intersection*/
  }
  ELESE {/* p is the right endpnt of s */
        IF both Above(T,s) and Below(T,s) exist then
           IFAbove(T,s) intersects Below(T,s) then return TRUE
   Delete(T,s)
  }
 Return "no two segments interesct"
```

Report All intersection (S) -pseudocode

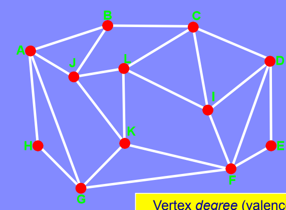
```
Create an empty set T
Sort endpnts of the segments of S and insert into events queue Q.
REPEAT { find next event p in Q
  IF p is the left endpnt of a segment s THEN{
  insert(T,s).
  If Above(T,s) intersects s THEN insert new intersection point into Q
  If Below(T,s) intersects s THEN insert new intersection point into Q
}
 ELSE IF  p is the right endpnt of s
   IF Above(T,s) intersects Below(T,s) THEN
      insert intersection into Q.

 ELSE /* p is an intersection point of s,s' */
   Check the two new nboring pairs for intersections and insert into Q
}UNTIL Q is empty /*reached rightmost endpoint */
```

Running time

- There are *2n* endpoints – $O(n \log n)$ time for sorting
- Each left endpoint event involved
  - Insertion into the tree $O(\log n)$
  - Finding successor/predecessor $O(\log n)$
  - Checking intersection with Above/Below – $O(1)$
- Each right endpoint event involved
  - Deletion from the tree $O(\log n)$.
  - Finding successor/predecessor $O(\log n)$.
  - Checking intersection between Above/Below – $O(1)$
- Total – $O(n \log n)$

Graph Definitions



G = <V,E>
V = vertices = {A,B,C,D,E,F,G,H,I,J,K,L}
E = edges = {(A,B),(B,C),(C,D),(D,E),(E,F),(F,G),(G,H),(H,A),(A,J),(A,G),(B,J),(K,F),(C,L),(C,I),(D,I),(D,F),(F,I),(G,K),(J,L),(J,K),(K,L),(L,I)}

Vertex *degree* (valence) = number of edges incident on vertex.
deg(J) = 4, deg(H) = 2
k-regular graph = graph whose vertices *all* have degree *k*

A *face* of a graph is a cycle of vertices/edges which cannot be shortened.
F = faces = {(A,H,G),(A,J,K,G),(B,A,J),(B,C,L,J),(C,I,J),(C,D,I),(D,E,F),(D,I,F),(L,I,F,K),(L,J,K),(K,F,G),(A,B,C,D,E,F,G,H)}

Connectivity

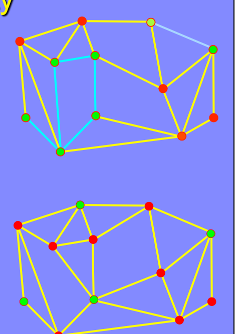
A graph is *connected* if there is a path of edges connecting every two vertices.

A graph is *k-connected* if between every two vertices there are *k* edge-disjoint paths.

A graph **G'=<V',E'>** is a *subgraph* of a graph **G=<V,E>** if **V'** is a subset of **V** and **E'** is the subset of **E** incident on **V'**.

A *connected component* of a graph is a maximal connected subgraph.

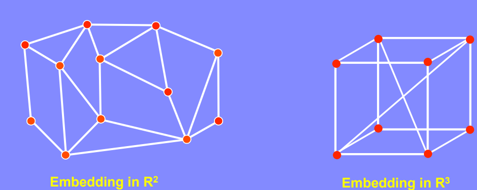
A subset **V'** of **V** is an *independent* set in **G** if the subgraph it induces does not contain any edges of **E**.



332.

Graph Embedding

A graph is *embedded* in R^d if each vertex is assigned a position in R^d.

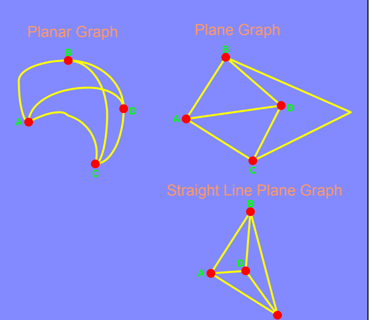


342.

Planar Graphs

A planar graph is a graph whose vertices and edges can be embedded in R^2 such that its edges do not intersect.

Every planar graph can be drawn as a straight-line plane graph.



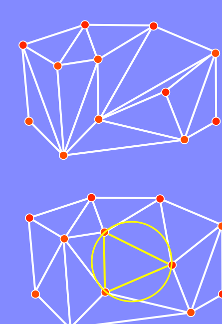
352.

Triangulation

A *triangulation* is a straight line plane graph whose faces are all triangles.

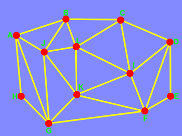
A *Delaunay triangulation* of a set of points is the unique set of triangles such that such that the circumcircle of any triangle does not contain any other point.

The Delaunay triangulation avoids long and skinny triangles.



362.

Topology



v =12
f = 14
e = 25

Euler Formula

For a connected planar graph:

v+f-e = 2

v = # vertices.
f = # faces
e = # edges

Exercises

Theorem: In a triangulation:

1. e = f = O(v)
2. The average vertex degree is ~6.

Proof: In such a mesh, f = 2e/3.
By Euler's formula: v+2e/3-e = 2
hence e = 3(v-2) and f = 2(v-2).

So Average(deg) = 2e/v = 6(v-2)/v
~ 6 for large v.