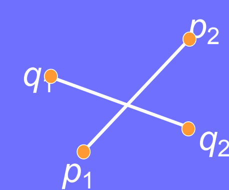
Next Topic: Line-Sweep Algorithm

- In this section, we will discuss the problem of computing all intersection between segments in a given set of segments
- The solution to this problem is based on the line-sweep paradigm. This is a **method** that is used in numerous problems, in many different domains.
- The original problem usually appear when we need to sympathize (fuze together) geometric data from different sources. Maps merging is a good example of such application.

Computing Intersection between segments

$$p(t) = p_1 + (p_2 - p_1)t \qquad 0 \le t \le 1$$

$$q(s) = q_1 + (q_2 - q_1)s \qquad 0 \le s \le 1$$



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

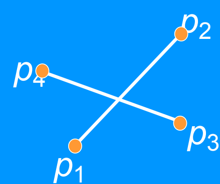
Solve (2D) linear vector equation for *t* and *s*:

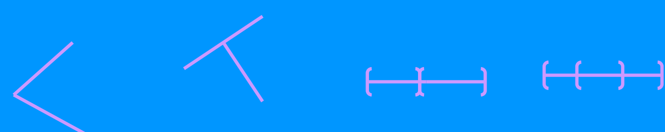
$$p(t) = q(s)$$

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

Here we actually need to use division.

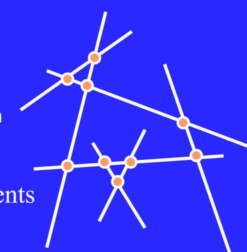
- The question of **"whether** a pair of segments intersect" could be answered more **robustly** than **"Where** do they interact"
- **Theorem:** Segments (p_1,p_2) and (p_3,p_4) intersect in their interior iff p_1 and p_2 are on different sides of the line p_3p_4 and p_3 and p_4 are on different sides of the line p_1p_2.
- This can be checked by computing the orientations of *four* triangles. Which ?
- Special cases:





Line Sweep (warning - sometimes called `plane sweep')

- **Problem**: Given *n* segments in the plane, compute all their intersections.
- Assume (**general position**):
  - No line segment is vertical.
  - No two segments are collinear.
  - No three segments intersect at a common point.



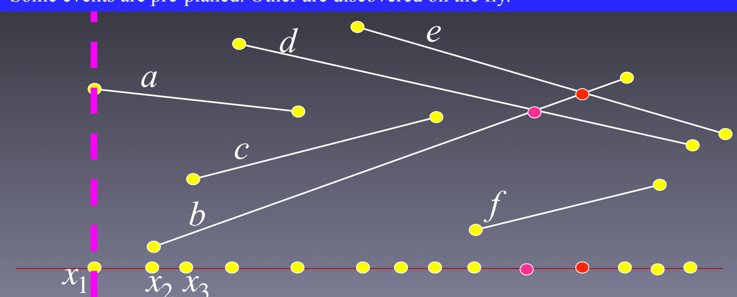
- Naive algorithm: Check each two segments for intersection. Complexity: $\Omega(n^2)$.

A description that does not assume general position can be found in MMMC.

A simpler version of this algorithm solves a simpler problem, which is
"is there any pair of cross each other"
You could find it in CLRS.

Sweep-line algorithm - **evens** driven alg.

- Sweep a vertical line from left to right, move from one event to the next event (from left to right)
- Three types of **events: left endpoint (birth), right endpoint (death) and intersection events**
- The line "knows" which segment it intersect and at which order (conceptually replacing *x*-coordinate with time).
- Each time two segments become neighbors on the line-sweep, the algorithm checks if they intersect in the future (creating an **intersection event**).
- Some events are pre-planed. Other are discovered on the fly.



Planned events (left/right endpoints) →

We will use two data structures:

1. A priority Queue (heap) - always knows what is the next event. (if you don't know about priorities queues, it could be replaced by a balanced search tree, or a SkipList)
2. A balanced tree - stores the **status (will define shortly)**

Sweep-line algorithm.

•Input: A set $P = \{s_1 \ldots s_n\}$ of segments in $\mathbb{R}^2$

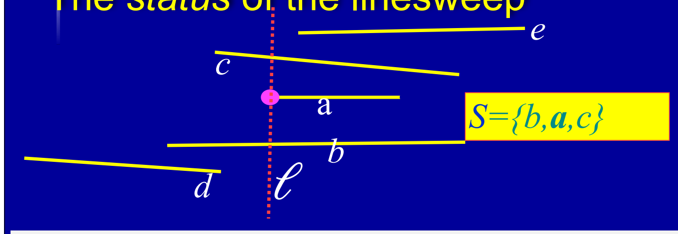
- Sweep a vertical line ℓ from left to right (conceptually replacing x-coordinate with time)
- Maintain dynamic the set $S \subseteq P$ of segments that intersect ℓ

ordered by the y-coordinate of intersection point with ℓ.

- Order changes when
  - new segment is encountered (**left** endpoints - birth event),
  - existing segment finishes (**right** endpoint - death event)
  - two segments meet at an intersection point.

For simplicity assume that a segment contains its left endpoint but does not contain its right endpoint (will be clear in the next slides)

The *status* of the linesweep



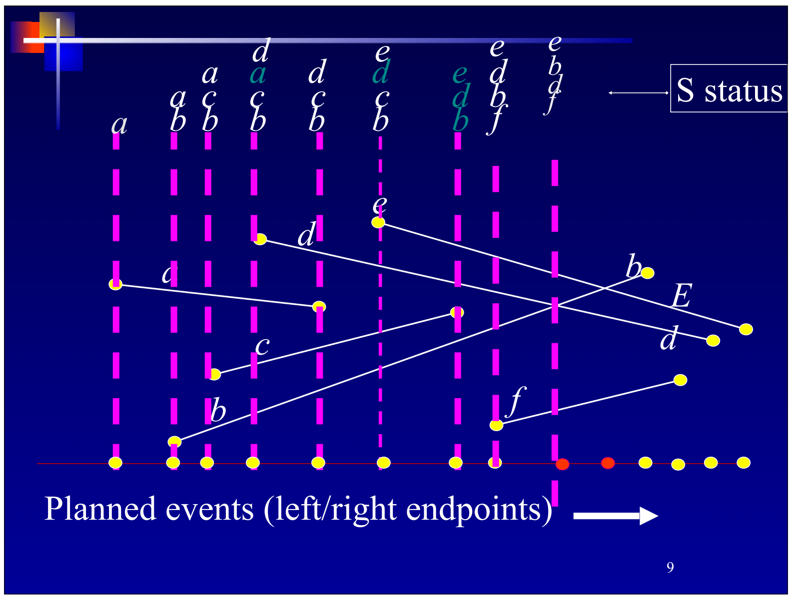
Def: The ***status*** S is the list of segments that the linesweep l intersects (in the order from bottom to top).

Def: an *event* happens when the status changes.
A birth event, a death event, and other types - to be discussed later.

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

When we reach the right endpoint of a segment, we remove this segment from the status

We will show the status right after the event.



Algorithm - overview

- Sweep with vertical line *l* from left to right
  (I.e. scan the endpoints in increasing ordered of *x*-coordinate)
- Each time that ℓ meets an **endpoint** –
  1. Update the status
  2. Check segments intersection as described in the next slides.

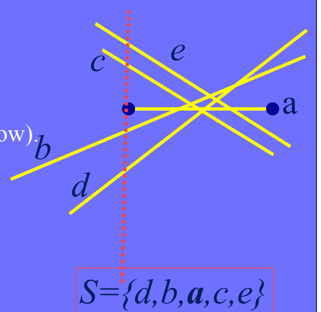
- Each time that ℓ meets an **intersection** –
  1. Report it
  2. Update the status (two segments switch places)
  3. Check for future intersections (there are ≤ 2 neighboring pairs)

Left endpoint event (birth event):

Process event points in order by from left to right.

For a **left** endpoint of segment *a*:

- Add segment *a* to the Status *S*.
- Check intersection between *a* and its immediate **neighbors** in status *S*. (≤ one nbr above, and ≤ one nbr below).
- If find any, insert as a future events.



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

(but not with e)

Right endpoint event

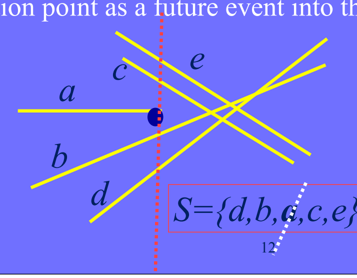
Process event points in order from left to right.

For a **right** endpoint of a segment *a (death event)*.

1. Delete segment *a* from the status *S*.
2. Check for future intersection between the segment about a, and the one below (if exist)

   (they became immediate nbrs of each other)
3. If exists, insert this intersection point as a future event into the the priority queue.

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



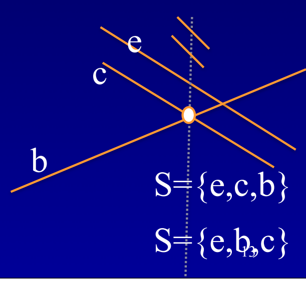
Intersection point event

Process event points (left to right)

An **intersection event**, caused by intersection of segment ***b*** with a segment ***c***:

1. Swap the order of these two segments in the status **S**.
2. Check for **future** intersection point between ***c*** and its **new** neighbor on ℓ. If exists, insert this event into the priority queue as a future event.
3. Repeat step 2 analogously with ***b***.

Example: b is checked for intersection with e.



Theorem: If segments c,b intersect at a point q, then at some event left of q, they become immediate neighbors along the sweeping line ℓ

Comment: Obviously there is some time when the thm holds, but why is there also an event ?

Proof: Assume WLOG that c is born after b. Consider the birth event of c. If c and b are neighbors on ℓ at this time, we are done.

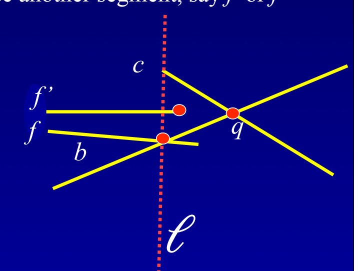
If they are not neighbors on ℓ , it is because another segment, say *f* or *f'* separating them on ℓ .

But then either

- ***f*** intersects *b (*at a point to the left of ***q****)*, *or*
- ***f*** intersects *c (*at a point to the left of ***q****)* or
- ***f '*** has a right endpoint *(*at a point to the left of ***q****)*

Each of these cases creates and event before q.

QED



Successor and predecessor

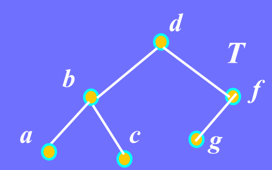
For a sorted set of keys S={4,19,52,77, 103}, the operation succ(x,S), th
successor of x in S, is the smallest key strictly larger than x.
succ(-3) returns 4,   succ(4) returns 19,   succ(5) returns 19,
succ(103)   returns 'NULL'

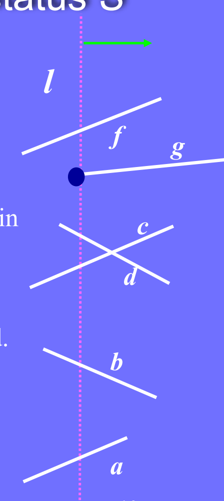
The pred(x,S) is the largest key strictly smaller than x
pred(19,S) returns 4.   pred(4) =NULL

In a balanced binary search tree, or a SkilList,

Basically, perform $find(x+\varepsilon)$, or $find(x-\varepsilon)$ for a really small ε.

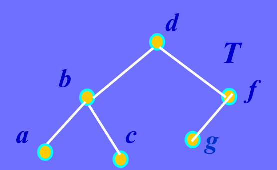
Maintaining the status *S*

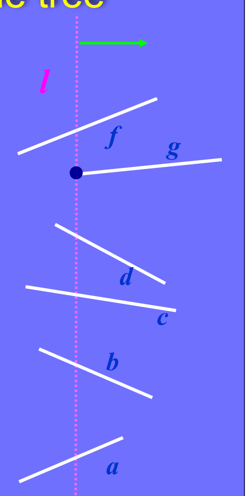




- We maintain the list of segment that intersect *l* in a sorted search tree *T,* sorted by the order they appear along *l.*
- We do not maintain exact y-values since they keep changing, but we calculate them if needed.
- When we insert a segment, we compare the *y*-coordinates of intersections of segments with *l*.
- **Note**: The *y*-coordinate change as *l* moves, but the <u>order</u> is changing only at events.

Operations on the tree





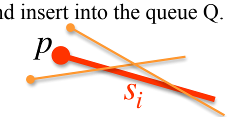
Insert(*T,s*) - Insert the segment *s* into the Tree *T*.
Delete(*T,s*)

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

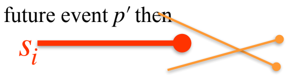
Algorithm – all together

Sort endpnts of the segments in S from left to right, and insert into the queue Q.
for each event p in the Q do

if p is the **left endpnt** of a segment s_i then

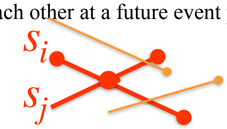


Insert(T,s_i)
if(Above(T,s_i) exists and intersects s_i at a future event p' then
Insert p' this event into the queue.
if(Below(T,s_i) exists and intersects s_i at a future event p' then
Insert p' this event into the queue.



Else if p is the **right** endpnt of s then
if Above(T,s_i) and Below(T,s_i) and intersect each other at a future event p'
Then Insert p' event into the queue.
Delete(T,s_i)

Else if p is an **intersection** point of s_i and s_j Then



Report intersection point.
switch their order in the status
Check if they intersect their new neighbors (and insert into Q if yes.)
End for

Question 1:
Can the same intersection point be reported more than once ?

Question 2:
Are the following two numbers always equal?

k_2 = number of intersection points.

k_1 = the number of pairs of segments crossing each other

How does it effect space requirement?

Naively - $\Theta(n+k)$ memory (as usual, k is the number of intersection points)

A tighter analysis shows that we need only O(n) memory

Question 2:
Are the following two numbers always equal?

k_2 = number of intersection points.
k_1 = the number of pairs of segments crossing each other

Time analysis

- There are *2n* endpoints – *O(n* log *n)* time for sorting
- Each left endpoint event requires
  - Insertion into the tree *O(*log *n)*.
  - Finding successor/predecessor *O(* log *n)*.
  - Checking intersection with Above/Below, and maybe inserting one or two events in *Q* – *O(3* log *n)*
- Each right endpoint event requires
  - Deletion from the tree *O(*log *n)*.
  - Finding successor/predecessor *O(*log *n)*.
  - Checking intersection between Above/Below – *O(1)*.
- On each intersection point, the algorithm spends *O(*log *n)* for the event itself, and *O(2*log *n)* for future events.
- Total – *O((n+k)* log *n)*. Here *k* is the total number of intersection points.

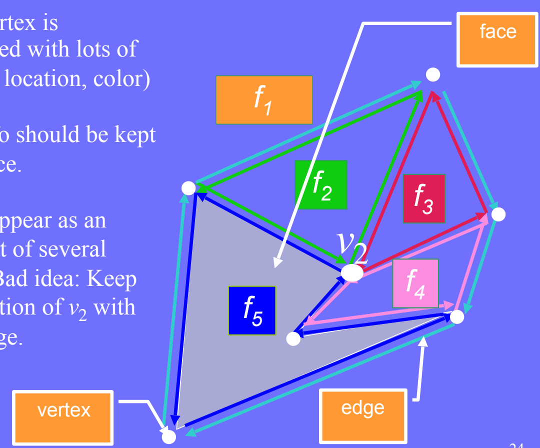
- We are done with line sweep
- Next: Representations of Planar Maps

Doubly Connected Edge List - DCEL

Each vertex is associated with lots of info (eg location, color)

This info should be kept only once.

Eg: v_2 appear as an endpoint of several edges. Bad idea: Keep the location of v_2 with each edge.



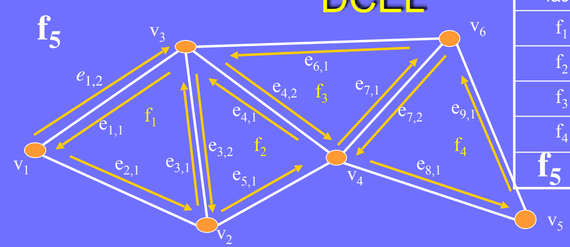
DCEL

- 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

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)

DCEL



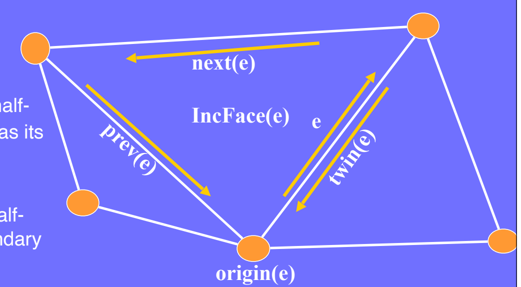
face	edge
f_1	$e_{1,1}$
f_2	$e_{5,1}$
f_3	$e_{4,2}$
f_4	$e_{8,1}$
f_5	$e_{1,2}$

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)	$e_{7,2}$

Half-edge	origin	twin	Incident face	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	$e_{7,1}$	$e_{6,1}$

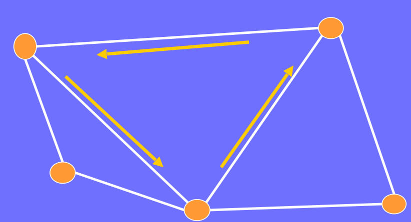
DCEL

- 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)

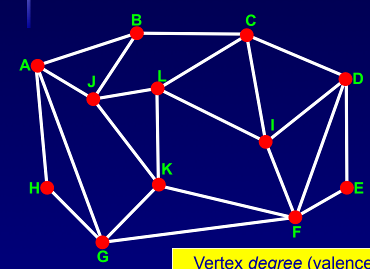


DCEL

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



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)}

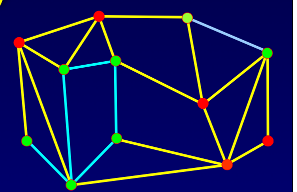
Connectivity

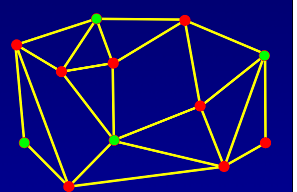
A graph is *connected* if there is a path of edges connecting every two vertices.

A graph **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.

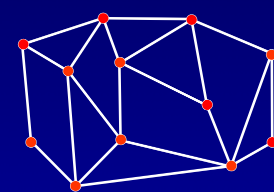
A subset **V'** of **V** is an *independent* set in **G** if the subgraph it induces does not contain any edges of **E**.



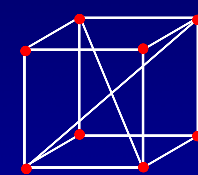


Graph Embedding

A graph is *embedded* in $\mathbb{R}^d$ (the d-dimensional space) if each vertex is assigned a position in R^d.



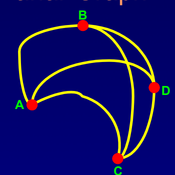
Embedding in R^2



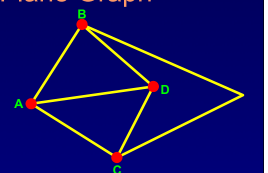
Embedding in R^3

Planar Graphs and Plane Graphs

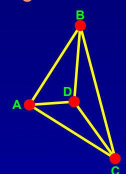
Planar Graph



Plane Graph



Straight Line Plane Graph

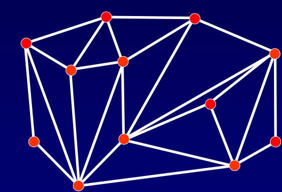


- A planar graph is a graph whose vertices and edges can be embedded in R^2 such that its edges do not cross. (meeting only at endpoints)
- Every planar graph can be drawn as a straight-line segments without crossing edges.
- The term "**plane graph**" means a planar graph together with an embedding of the graph in the plane, such that its edges are drawn using straight non-crossing edges.

Triangulation

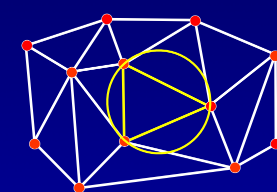
A *triangulation* is a straight line plane graph whose faces are all triangles.

(excluding, of course, the outer face)

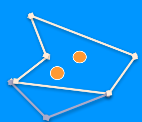


A *Delaunay triangulation* of a set of points is the unique set of triangles such that such that the circumcircle of any triangle does not contain any other point.

The Delaunay triangulation avoids long and skinny triangles.



Topology and Euler Formula

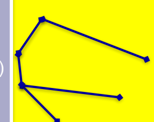


Example $n_v = 6$
$n_f = 3$ **(2+the outside face)**
$n_e = 7$

Two points p,q below to the same face if we can start walking from p and reach q without crossing any edges

Euler Formula For a connected planar graph without parallel edges:
Let
- n_v denote the number of vertices,
- n_f denote the number of faces (including one "surrounding the graph)
- n_e denote the number of edges

Then $n_f = 2 + n_e - n_v$
Proof by induction: remove edges until left with a tree.
This is the **Base Case:** n_f **but in a tree** $n_e = n_v - 1$. **Claim holds.**
Next, add any edge that was deleted.
Both #faces and #edges increased by 1. QED



Conclusion

Theorem: In a triangulation:
1. $n_v = \Theta(n_f)$ **and** $n_e = \Theta(n_f)$ (that is, all three terms are within a constant from each other)
2. The average vertex degree is ~6.

Proof: In such a triangulated mesh. Replace each edge by two half -edges. Each face (excluding the outer one) uses exactly 3. So $n_f = 1_{outter\ face} + 2n_e/3$.

$n_v - e_e + n_f = 2$ **Euler Formula**

$n_v - n_e + \left(1 + \frac{2}{3}n_e\right) = 2$

$n_v - \frac{1}{3}n_e = 1$

$n_e \approx 3n_v$

hence $n_f = 2(n_v - 2)$

Average(deg) = $2n_e/n_v = \frac{1}{n_v}\sum_{v_i \in V} \mathtt{deg}(v_i) \approx 6$.