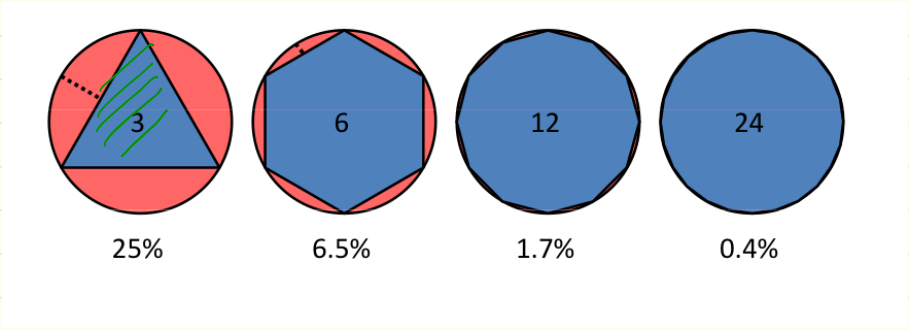
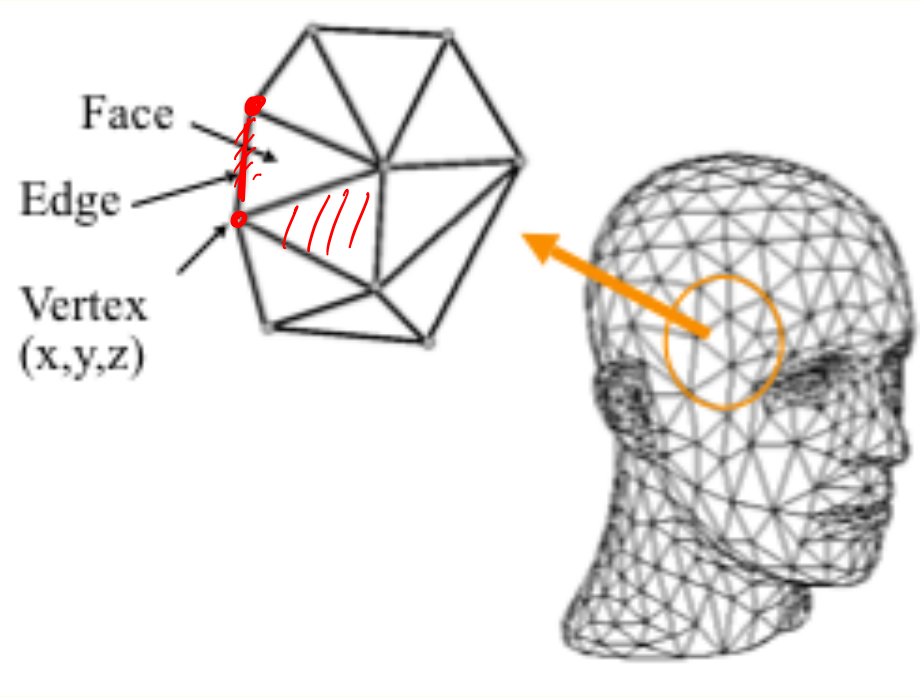
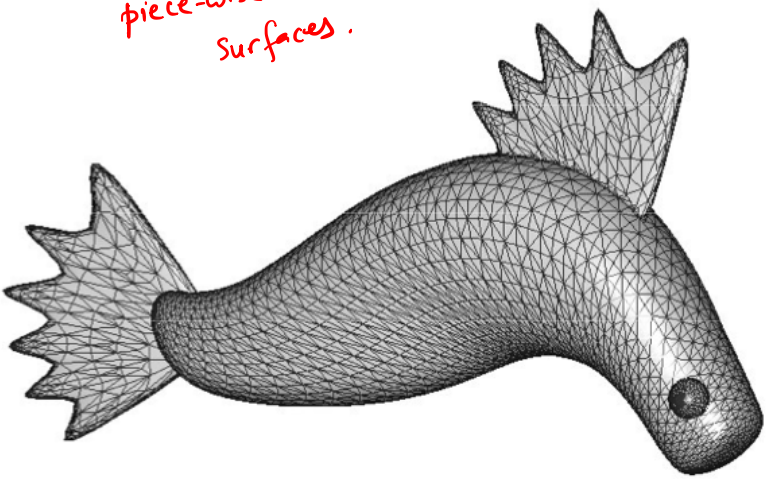
LECTURE 2: MESHES



Approximate smooth surfaces by triangles, quads....


Face
Edge
Vertex
(x,y,z)

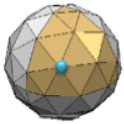
piece-wise linear surfaces.



- **Vertices:** 4,634
- **Edges:** 13,872
- **Faces:** 9,248

Triangle Meshes

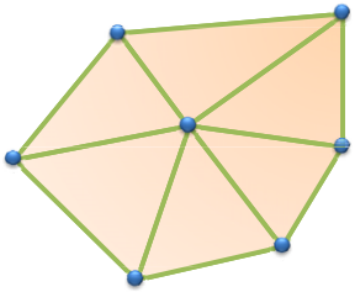
- Connectivity: vertices, edges, triangles



$$\mathcal{V} = \{v_1, \ldots, v_n\}$$

$$\mathcal{E} = \{e_1, \ldots, e_k\}\ , \quad e_i \in \mathcal{V} \times \mathcal{V}$$

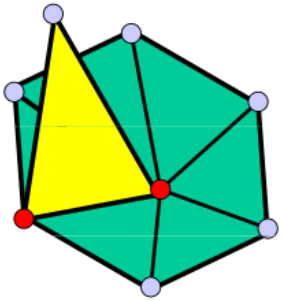
$$\mathcal{F} = \{f_1, \ldots, f_m\}\ , \quad f_i \in \mathcal{V} \times \mathcal{V} \times \mathcal{V}$$

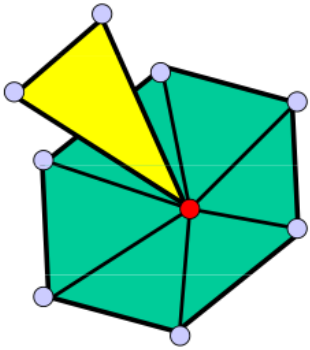


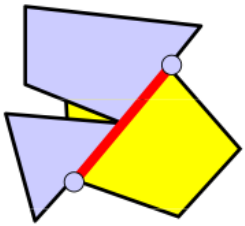
- Geometry: vertex positions

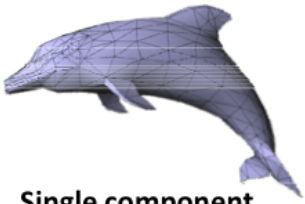
$$\mathcal{P} = \{\mathbf{p}_1, \ldots, \mathbf{p}_n\}\ , \quad \mathbf{p}_i \in \mathbb{R}^3$$

- **Manifold Conditions: (1)** Each edge is incident to only one or two faces and **(2)** the faces incident to a vertex form a closed or an open fan.
- The following examples are not manifold meshes!

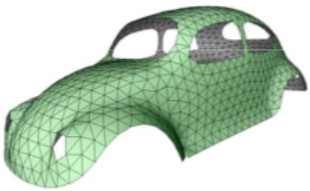




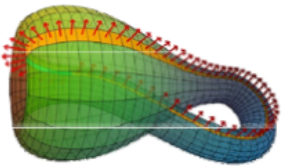




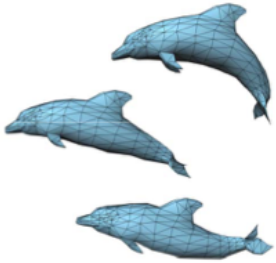
Single component, closed, triangular, orientable manifold



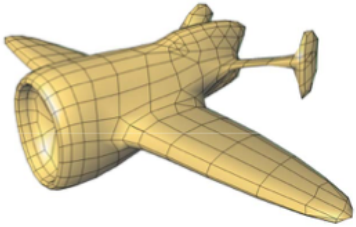
With boundaries



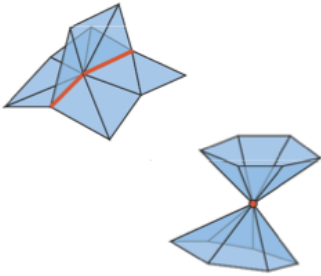
Not orientable



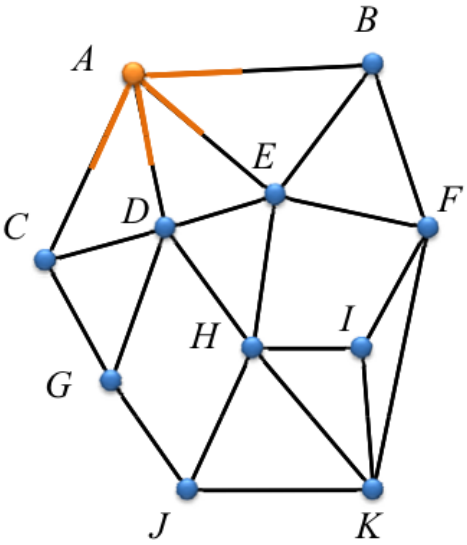
Multiple components



Not only triangles



Non manifold



Vertex degree or **valence** = number of incident edges

deg(A) = 4

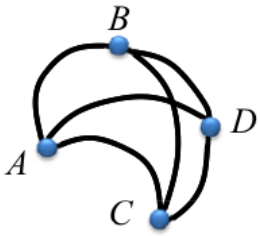
deg(E) = 5

Regular mesh =

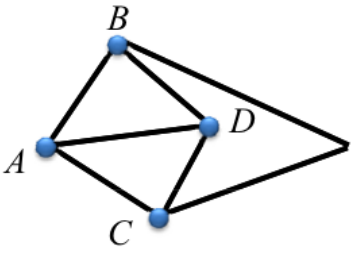
all vertex degrees are equal

Planar Graph =
Graph whose vertices and edges
can be embedded in R^2 such that
its edges do not intersect

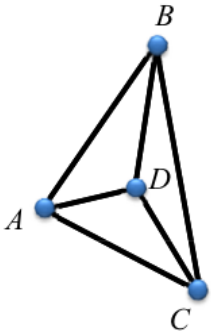
Planar Graph

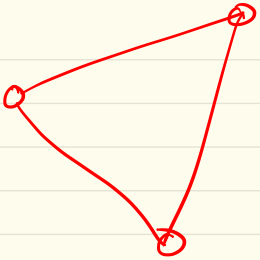


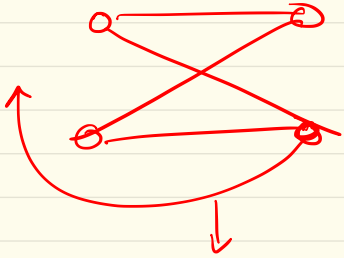
Plane Graph



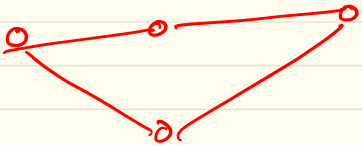
Straight Line Plane Graph

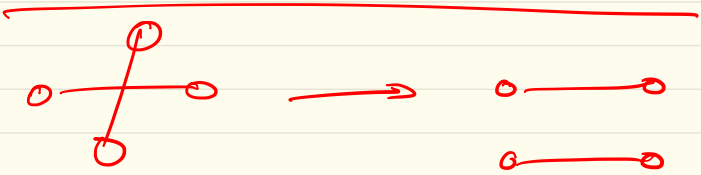


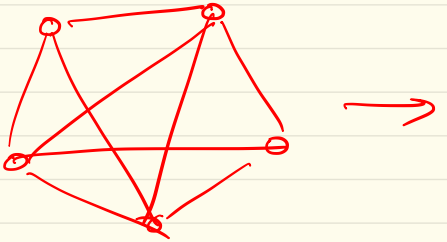




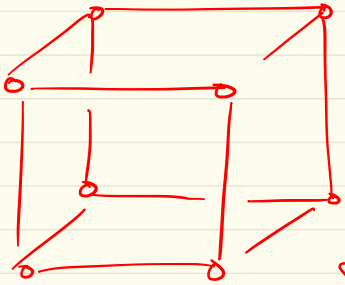
x



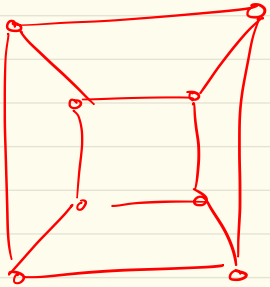




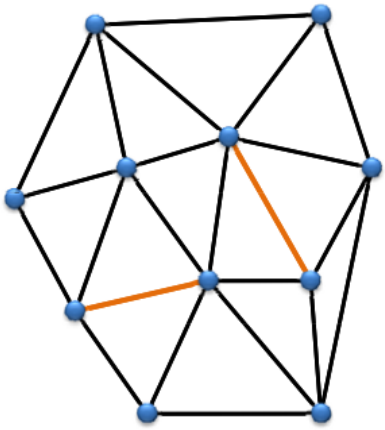
not planar.



is it planar ?



planar graph.



Triangulation:

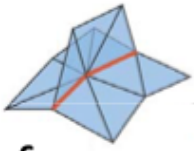
Straight line plane graph where every face is a *triangle*.

Mesh:
straight-line graph embedded in R^3

Boundary edge:

adjacent to exactly *one* face

Regular edge:
adjacent to exactly *two* faces

Singular edge:

adjacent to more than two faces

Closed mesh:

mesh with no boundary edges

How to store a mesh in the computer?

Trade-offs.

Simple storage → inefficient to find info

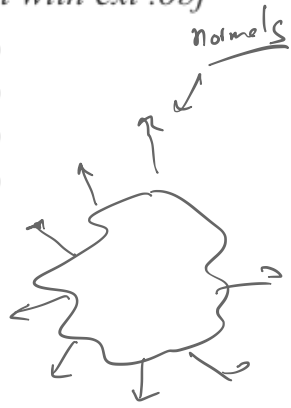
Complicated storage → inefficient to maintain info.

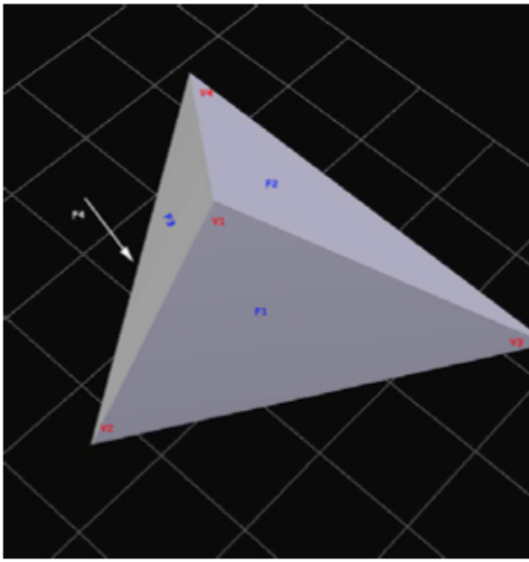
Queries:

Obj Example

- Tetrahedron

```
# Obj file format with ext .obj
1 v 1.0 1.0 1.0
2 v 2.0 0.0 0.0
3 v 0.0 0.0 0.0
4 v 1.0 1.6 0.0
  f 1 3 2
  f 1 2 4
  f 1 4 3
  f 2 3 4
```

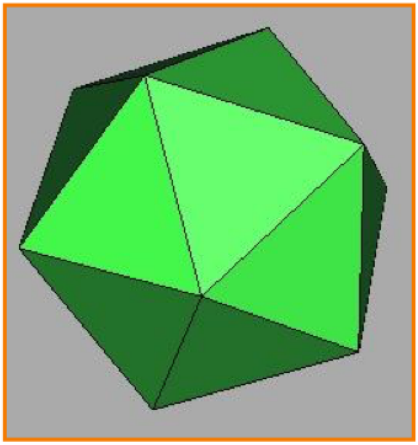


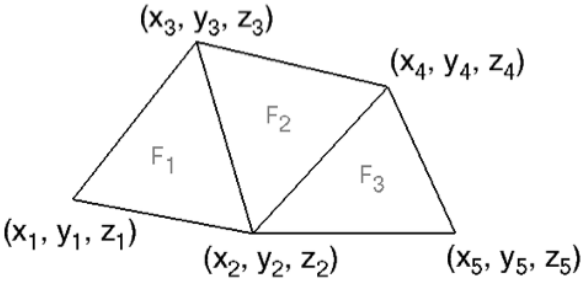


anti-clockwise ordering.

Independent Faces

- Each face lists vertex coordinates



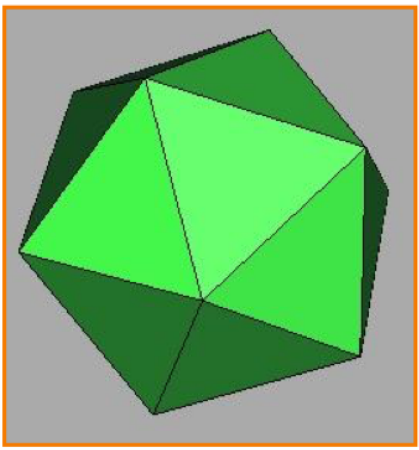


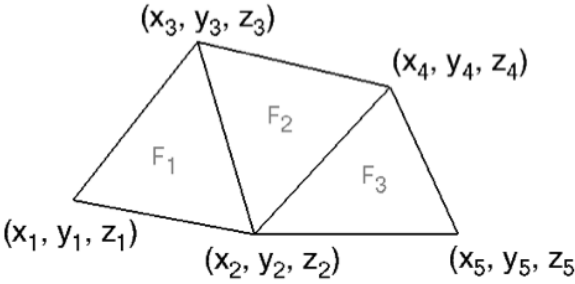
FACE TABLE	
F_1	(x_1, y_1, z_1) (x_2, y_2, z_2) (x_3, y_3, z_3)
F_2	(x_2, y_2, z_2) (x_4, y_4, z_4) (x_3, y_3, z_3)
F_3	(x_2, y_2, z_2) (x_5, y_5, z_5) (x_4, y_4, z_4)

Vertex and Face Tables

same as obj file

- Each face lists vertex references

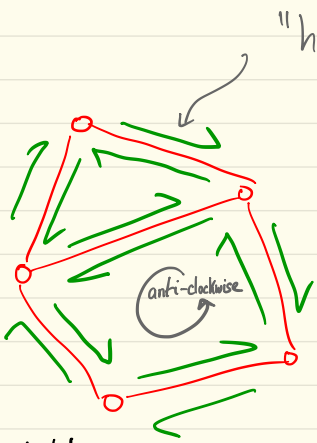




VERTEX TABLE			
V_1	X_1	Y_1	Z_1
V_2	X_2	Y_2	Z_2
V_3	X_3	Y_3	Z_3
V_4	X_4	Y_4	Z_4
V_5	X_5	Y_5	Z_5

FACE TABLE			
F_1	V_1	V_2	V_3
F_2	V_2	V_4	V_3
F_3	V_2	V_5	V_4

HALF-EDGE DATA STRUCTURE



Keep THREE tables

VERTICES

v_1
v_2
v_3
⋮

1. Pointer to any outgoing half-edge
2. Vertex position in 3D (x,y,z)

FACES

f_1
f_2
⋮

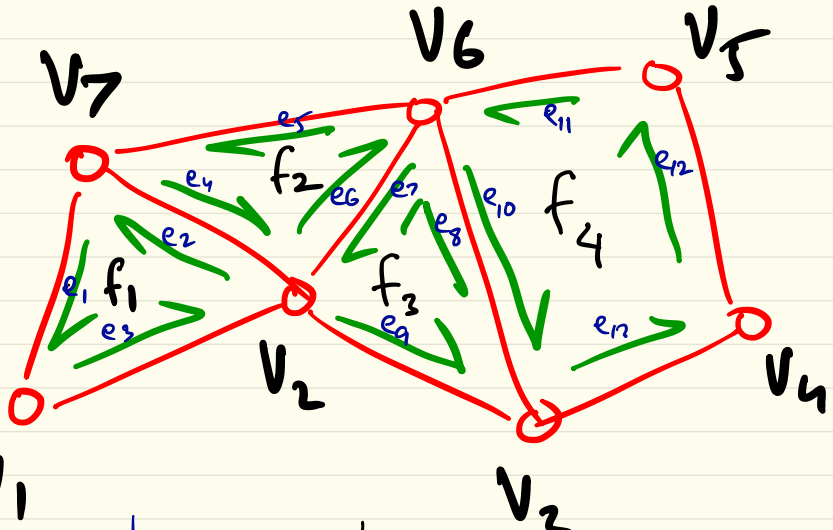
Pointer to any halfedge lying in that face

HALF EDGES

e_1
e_2
⋮

1. origin vertex
2. twin half-edge
3. pointer to face
4. pointer to next
5. pointer to prev

called the halfedge data structure



VERTEX TABLE

V_1	e_3
V_2	e_6
V_3	e_{13}
V_4	e_{12}
V_5	e_{11}
V_6	e_7
V_7	e_1

FACE TABLE

f_1	e_1
f_2	e_4
f_3	e_8
f_4	e_{10}

HALF-EDGE TABLE

	vertex	face	twin	next	prev
e_2	V_2	f_1	e_4	e_1	e_3
e_3	V_1	f_1	—	e_2	e_1
e_4					
e_5					
e_6	V_2	f_2	e_7	e_5	e_4
e_7					
e_8					

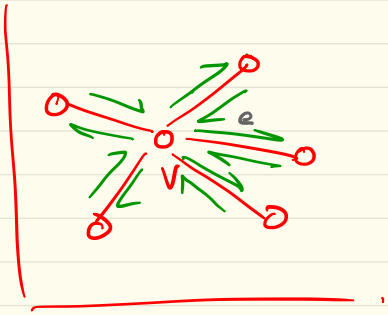
This data-structure in action.

Give: myVertex * v

Output: the list of all the vertices adjacent to v.

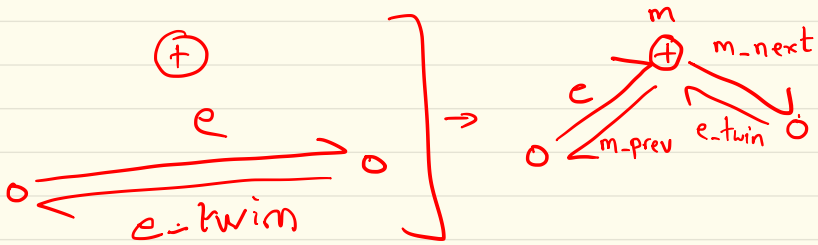
```
myHalfedge * e = v -> origin;
do {
    output  e -> next -> source

    e = e -> twin -> next;
} while ( e != v -> origin );
```



Input: myHalfedge *e, myPoint3D *m;

Output: break e into 2 by adding point m.




```
d->next = e->next
d->prev = e
d->twin = dt
d->face = e->face
d->source = m
```

```
dt->next = et
dt->prev = et->prev
dt->twin = d
dt->face = et->face
dt->source = et->sour
```

```
e->next = d
e->prev =
e->twin =
e->face =
e->sour =
```

```
et->next =
et->prev = dt
et->twin =
et->face =
et->sour = m
```

```
m->originof = d;
dt->prev->next = dt;
d->next->prev = d;
dt->source->originof = dt;
```

TRIANGULATE A FACE :

Input: myFace *f

Output: break this face into triangles.