Computer Graphics
- **Meshes and Manfolds**

Junjie Cao @ DLUT

Spring 2017

http://jjcao.github.io/ComputerGraphics/

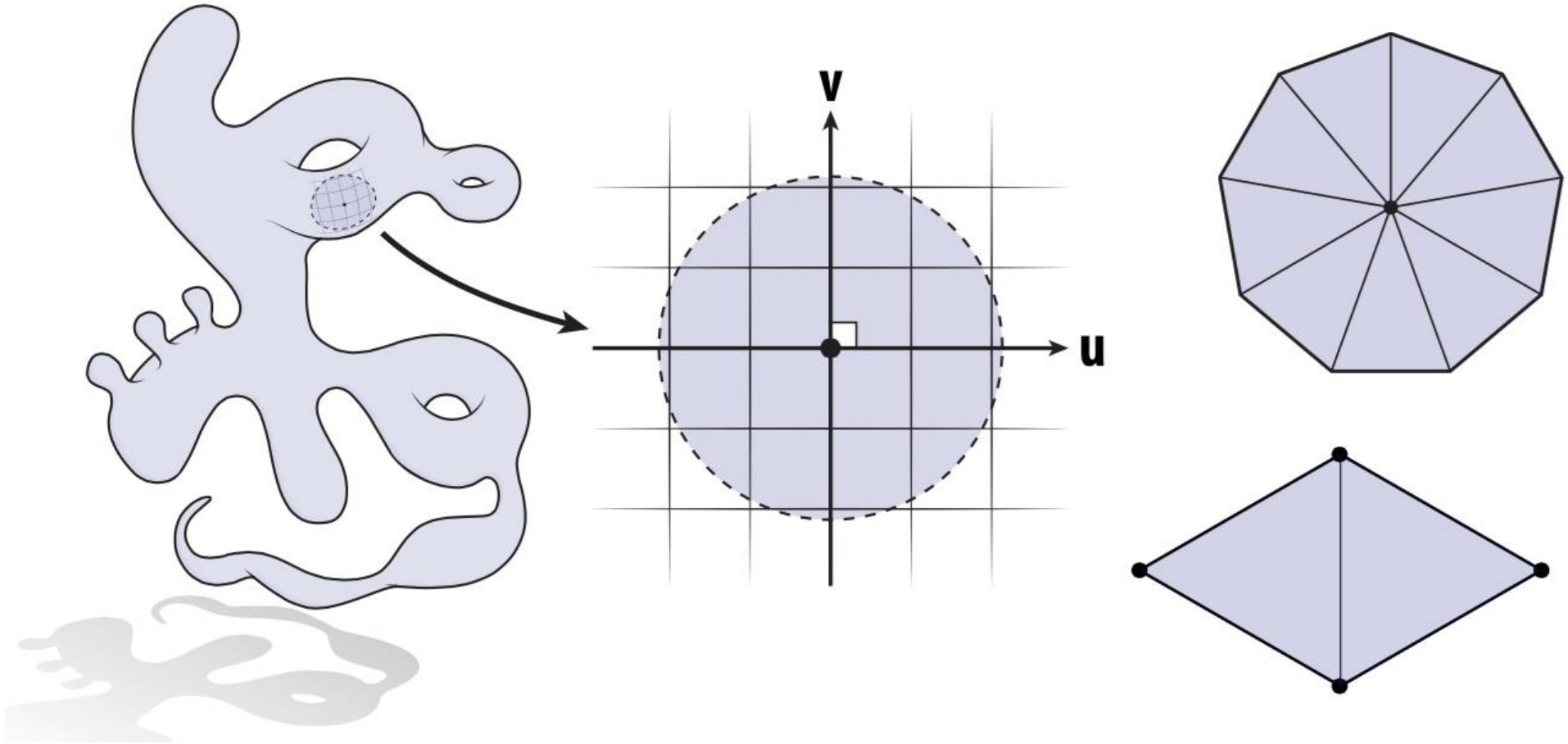
Music is dynamic, while score is static;
Movement is dynamic, while law is static.

Last time: overview of geom

- Many types of geometry in nature
- Demand sophisticated representations
- Two major categories:
  - IMPLICIT - “tests” if a point is in shape
  - EXPLICIT - directly “lists” points
- Lots of representations for both

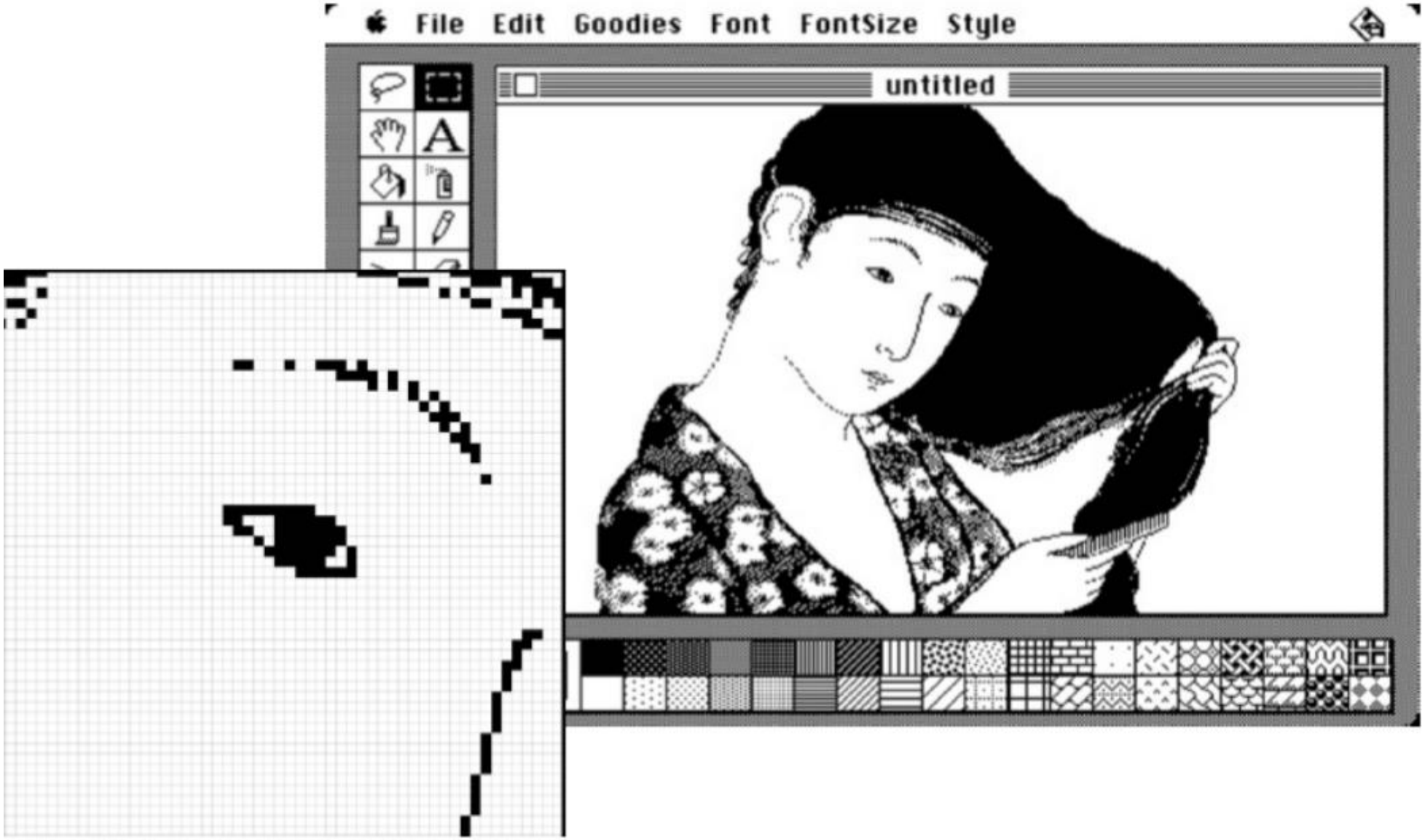
Manifold assumption

- **Today we're going to introduce the idea of *manifold* geometry**
- **Can be hard to understand motivation at first!**
- **So first, let's revisit a more familiar example...**



Bitmap Images, Revisited

- **To encode images, we used a *regular grid* of pixels:**



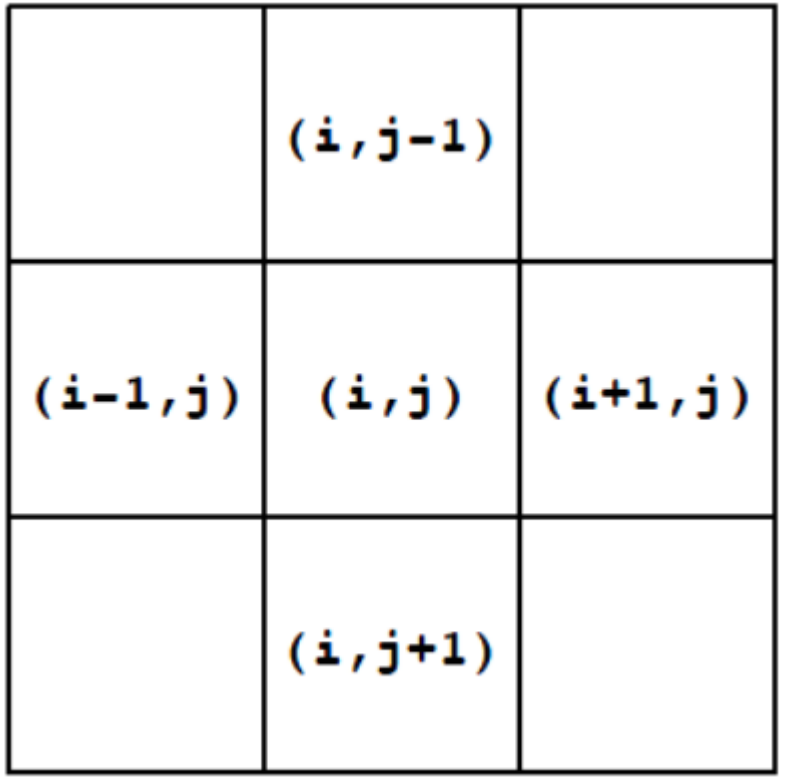
But images are not fundamentally made of little squares:

Goyō Hashiguchi, *Kamisuki* (ca 1920)

So why did we choose a square grid?

…rather than dozens of alternatives?

Regular grids make life easy



- One reason: SIMPLICITY / EFFICIENCY
  - E.g., always have four neighbors
  - Easy to index, easy to filter…
  - Storage is just a list of numbers
- Another reason: GENERALITY
  - Can encode basically any image
- Are regular grids *always* the best choice for bitmap images?
  - No! E.g., suffer from anisotropy, don't capture edges, ...
  - But *more often than not* are a pretty good choice
- Will see a similar story with geometry...

So, how should we encode surfaces?

Smooth Surfaces

- **Intuitively, a *surface* is the boundary or “shell” of an object**
- **(Think about the candy shell, not the chocolate.)**
- **Surfaces are *manifold*:**
  - **If you zoom in far enough (at any point) looks like a plane***
  - **E.g., the Earth from space vs. from the ground**

Parametric Approximations

Parametric Representation

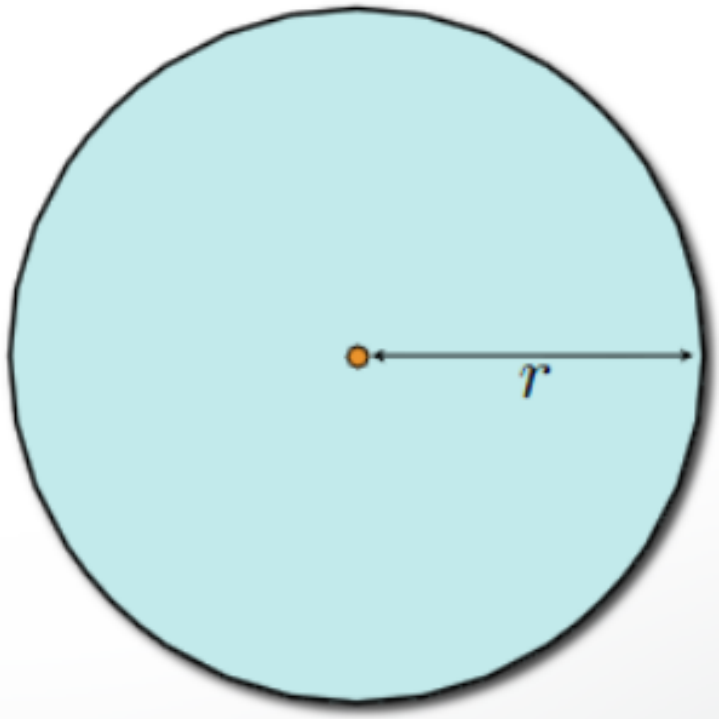
- Surface is the range of a function

$$\mathbf{f} : \Omega \subset \mathbb{R}^2 \to \mathbb{R}^3, \quad \mathcal{S}_\Omega = \mathbf{f}(\Omega)$$

2D example: A Circle

$$\mathbf{f} : [0, 2\pi] \to \mathbb{R}^2$$

$$\mathbf{f}(t) = \begin{pmatrix} r\cos(t) \\ r\sin(t) \end{pmatrix}$$



Parametric Representation

Surface is the range of a function

$$\mathbf{f} : \Omega \subset \mathbb{R}^2 \to \mathbb{R}^3, \quad \mathcal{S}_\Omega = \mathbf{f}(\Omega)$$

2D example: Island coast line

$$\mathbf{f} : [0, 2\pi] \to \mathbb{R}^2$$

$$\mathbf{f}(t) = \begin{pmatrix} ? \\ ? \end{pmatrix}$$

Parametric Representation

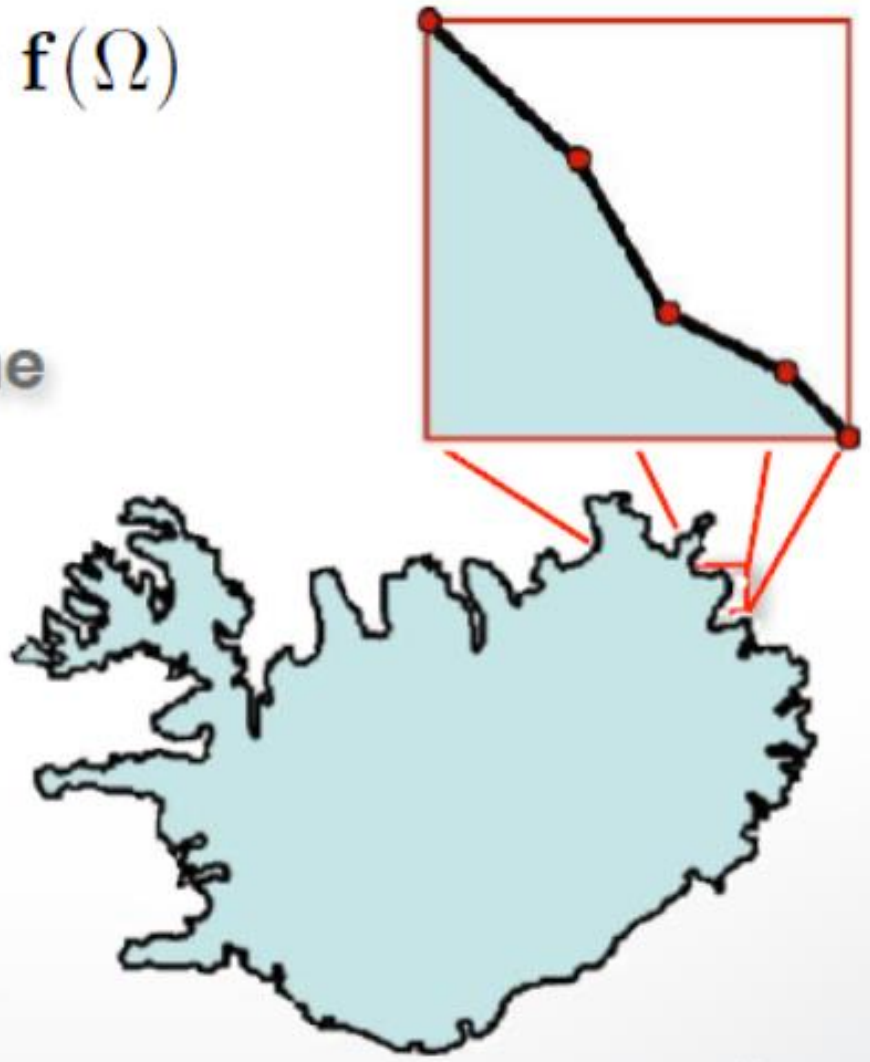
Surface is the range of a function

$$\mathbf{f} : \Omega \subset \mathbb{R}^2 \to \mathbb{R}^3, \quad \mathcal{S}_\Omega = \mathbf{f}(\Omega)$$

2D example: Island coast line

$$\mathbf{f} : [0, 2\pi] \to \mathbb{R}^2$$

$$\mathbf{f}(t) = \begin{pmatrix} ? \\ ? \end{pmatrix}$$



Parametric Representation

Polynomials are computable functions

$$f(t) = \sum_{i=0}^{p} c_i\, t^i = \sum_{i=0}^{p} \tilde{c}_i\, \phi_i(t)$$

Taylor expansion up to degree p

$$g(h) = \sum_{i=0}^{p} \frac{1}{i!}\, g^{(i)}(0)\; h^i + O\big(h^{p+1}\big)$$

Error for approximation g by polynomial f

$$f(t_i) = g(t_i)\,, \quad 0 \le t_0 < \cdots < t_p \le h$$

$$|f(t) - g(t)| \;\le\; \frac{1}{(p+1)!}\, \max f^{(p+1)} \prod_{i=0}^{p} (t - t_i) \;=\; O\Big(h^{(p+1)}\Big)$$

Parametric Representation

Approximation error is $O(h^{p+1})$

Improve approximation quality by

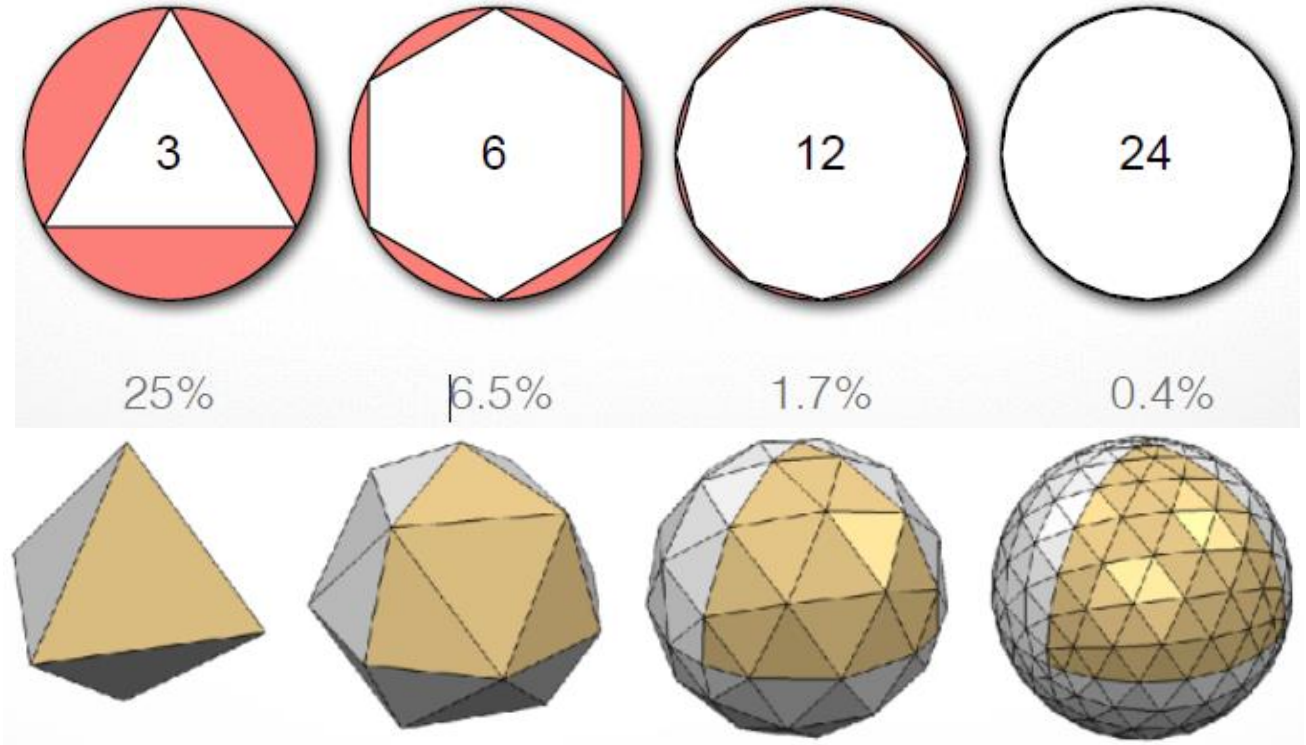
- increasing p .. higher order polynomials
- decreasing h … shorter / more segments

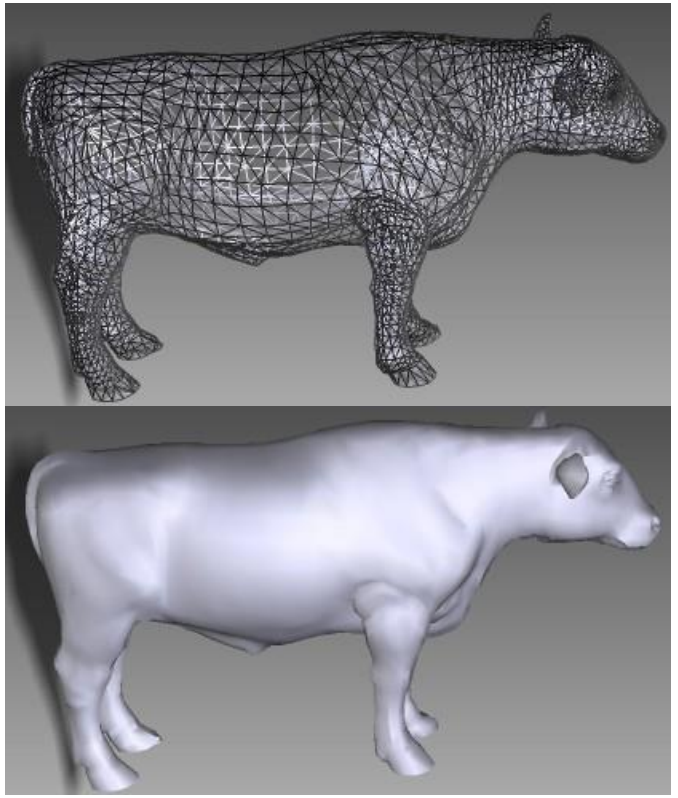
Issues

- smoothness of the target data ($\max_t f^{(p+1)}(t)$)
- smoothness condition between segments

Polygonal meshes are a good compromise

- Piecewise linear approximation → error is O(h^2)
- **Theorem** Given a smooth surface S and a given error $\varepsilon > 0$, there exists a piecewise linear surface (mesh) M, such that $|M - S| < \varepsilon$ for all points of M.
- Error inversely proportional to #faces



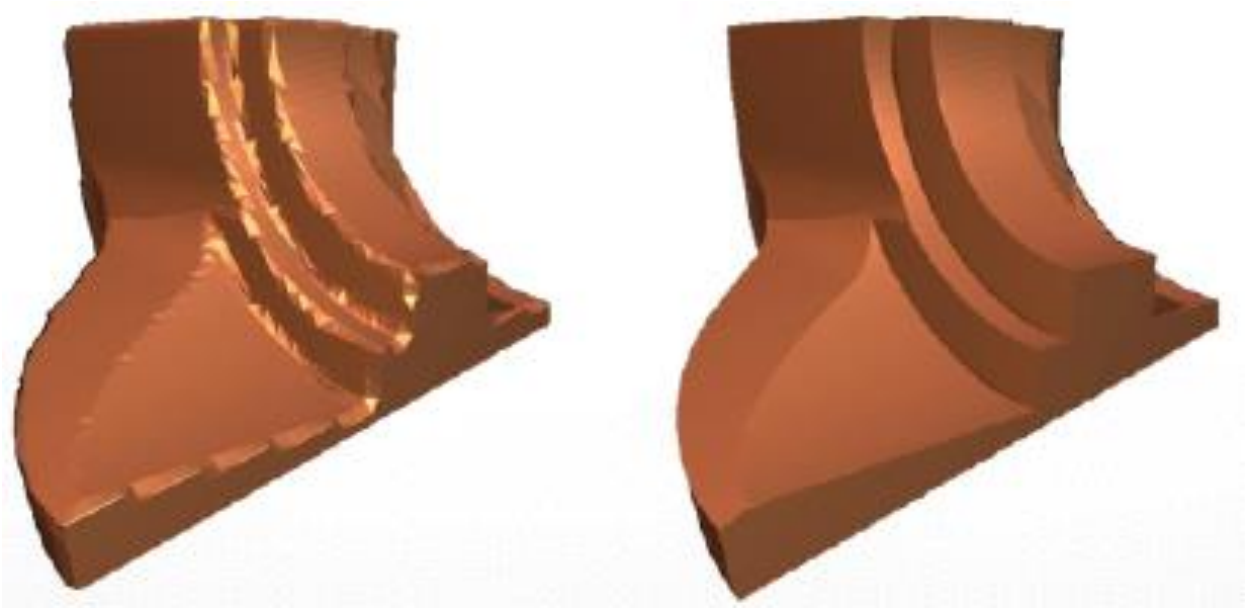


Polygonal meshes are a good compromise

- Piecewise linear approximation → error is O(h^2)
- Arbitrary topology surfaces

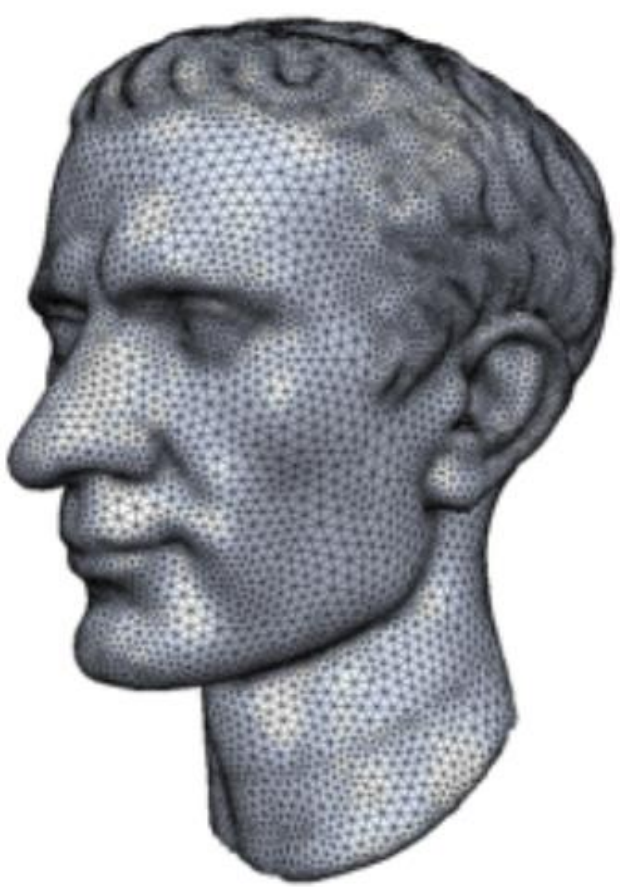
Polygonal meshes are a good compromise

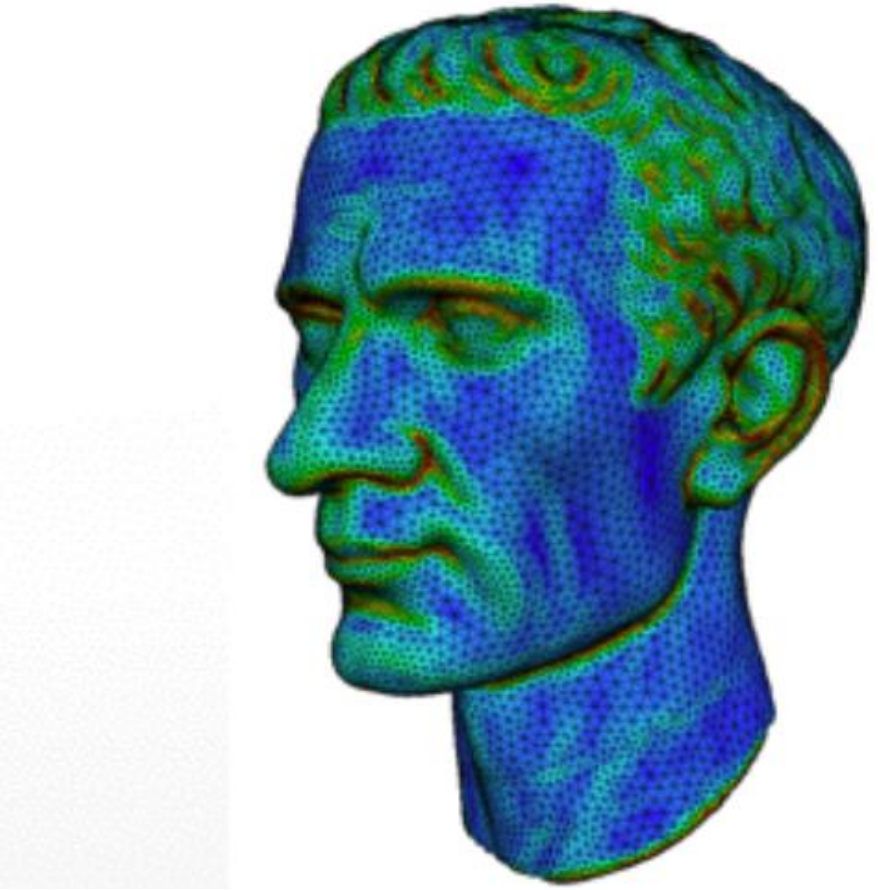
- Piecewise linear approximation → error is O(h^2)
- Arbitrary topology surfaces
- Piecewise smooth surfaces



Polygonal meshes are a good compromise

- Piecewise linear approximation → error is O(h^2)
- Arbitrary topology surfaces
- Piecewise smooth surfaces
- Adaptive sampling





Polygonal meshes are a good compromise

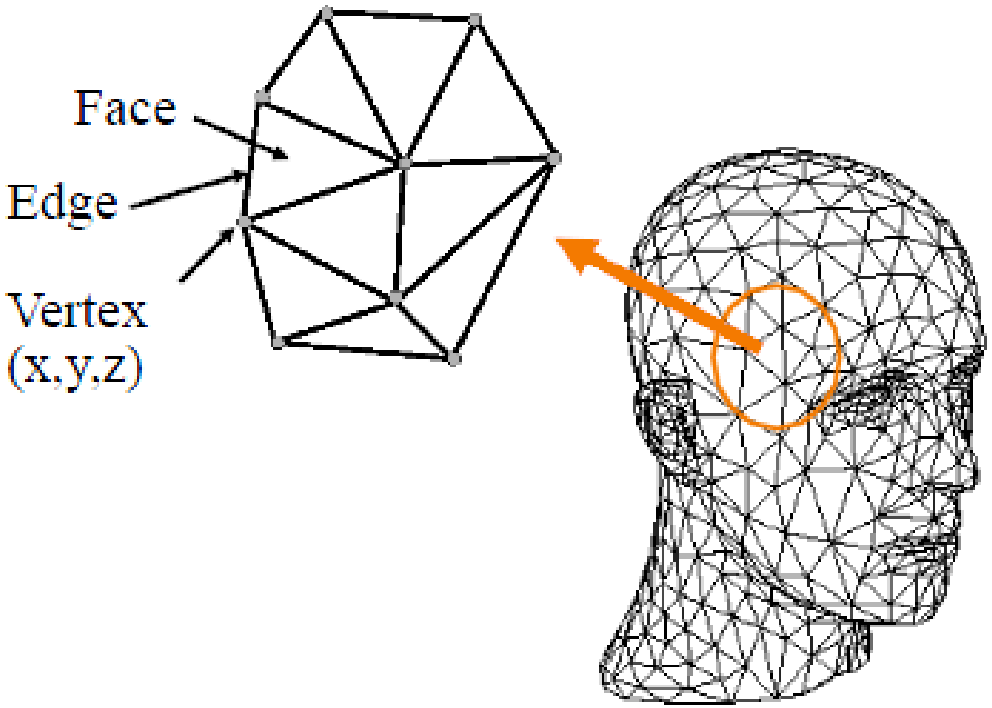
- Piecewise linear approximation → error is O(h^2)
- Arbitrary topology surfaces
- Piecewise smooth surfaces
- Adaptive sampling
- Efficient GPU-based rendering/processing

What is a Mesh?

What is a Mesh?

- A Mesh is a pair *(P,K)*, where *P* is a set of point positions $P = \{ p_i \in R^3 \mid 1 \le i \le n \}$ and *K* is an abstract simplicial complex which contains all topological information.

- *K* is a set of subsets of $\{1, \ldots, N\}$: $K = V \cup E \cup F$
  - Vertices $v = \{i\} \in V$
  - Edges $e = \{i, j\} \in E$
  - Faces $f = \{i_1, i_2, \ldots, i_{n_f}\} \in F$



- A **Graph** is a pair G=(V,E)

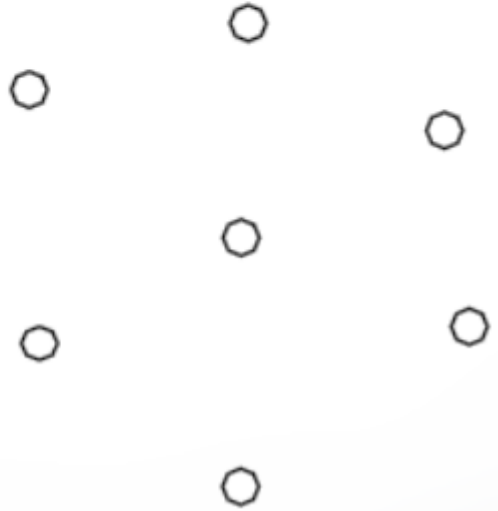
Zorin & Schroeder

Polygonal Meshes

$$\mathcal{M} = (\{\mathbf{v}_i\}, \{e_j\}, \{f_k\})$$

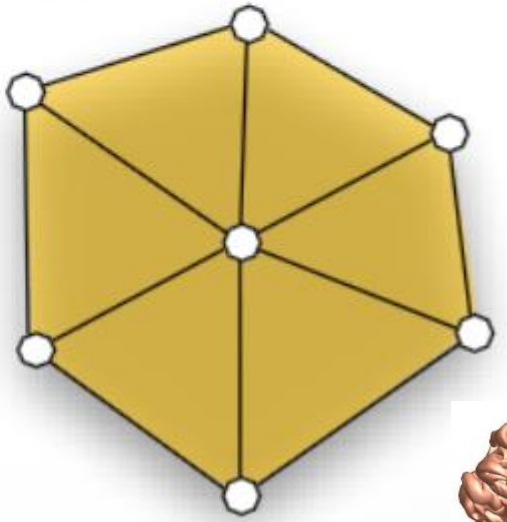
$$\mathcal{M} = (\{\mathbf{v}_i\}, \{e_j\}, \{f_k\})$$

geometry $\mathbf{v}_i \in \mathbb{R}^3$



geometry $\mathbf{v}_i \in \mathbb{R}^3$

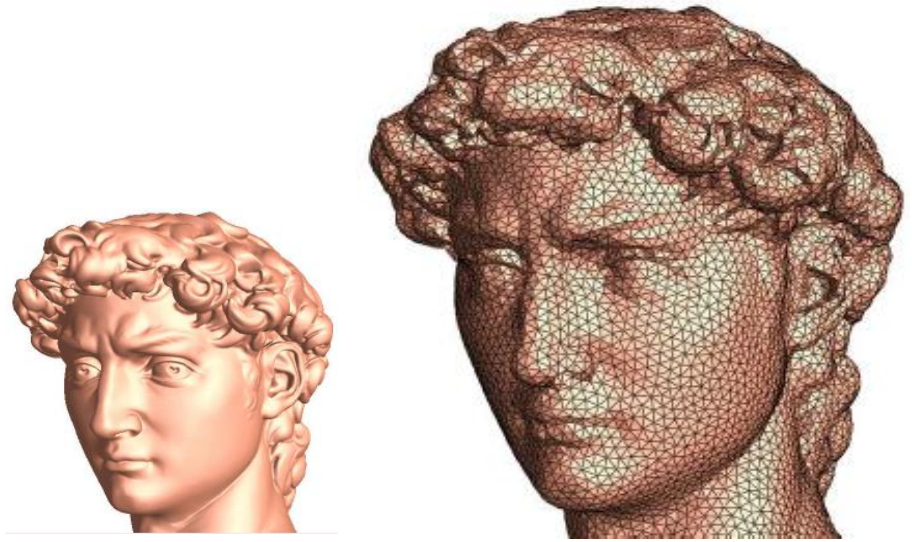
topology $e_i, f_i \subset \mathbb{R}^3$



Polygonal Meshes

- Geometry
  - Embedding – Vertex coordinates
  - Riemannian metrics – Edge lengths
  - Conformal Structure – Corner angles (and other variant definitions)

- Topology
  - Simplicial Complex, Combinatorics
  - connectivity of the vertices

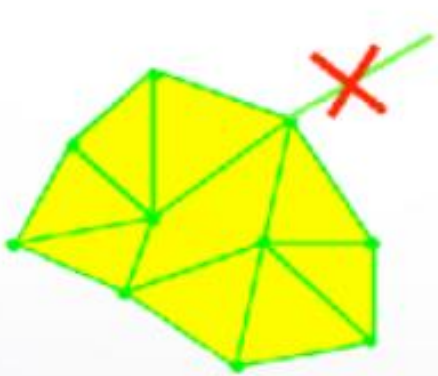


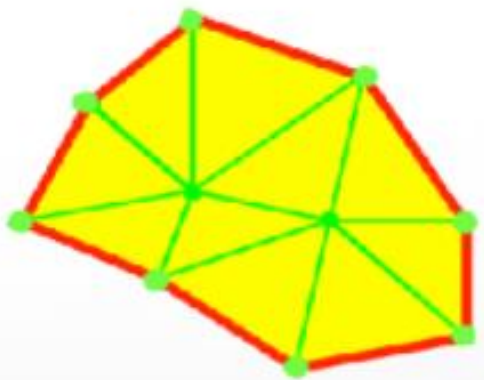
What is a Polygonal Mesh?

A set M of finite number of closed polygons Q_i if:

- Intersection of inner polygonal areas is empty
- Intersection of 2 polygons from M is either empty, a point $p \in P$ or an edge $e \in E$
- Every edge $e \in E$ belongs to at least one polygon
- The set of all edges which belong only to one polygon are called edges of the polygonal mesh and are either empty or form a single closed polygon

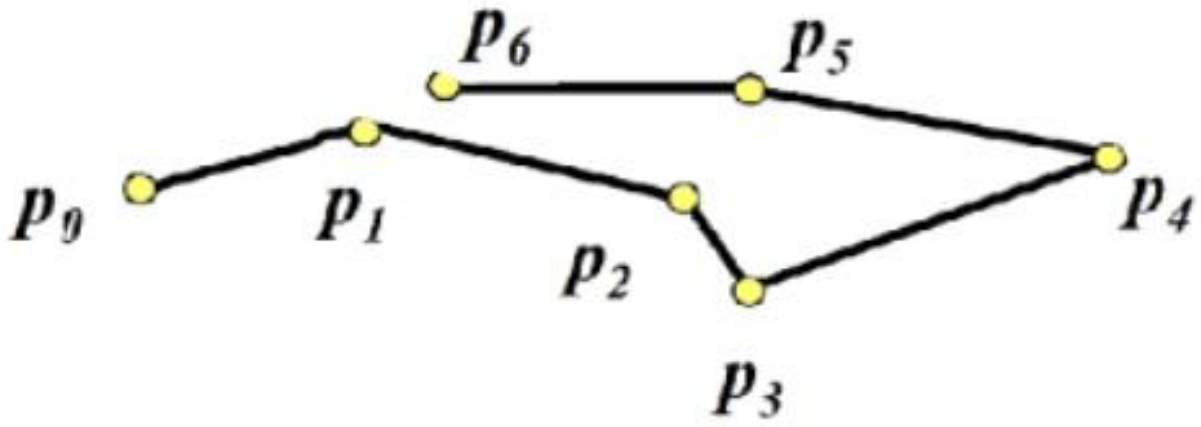
So mesh => graph
Graph may !=> mesh





Polygonal Mesh – face – **Polygon**
Triangular Mesh – face -- **Triangle**

A geometric graph $Q = (V, E)$
with $V = \{\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{n-1}\}$ in $\mathbb{R}^d$, $d \geq 2$
and $E = \{(\mathbf{p}_0, \mathbf{p}_1) \ldots (\mathbf{p}_{n-2}, \mathbf{p}_{n-1})\}$
is called a **polygon**



A polygon is called

- flat, if all edges are on a plane
- closed, if $\mathbf{p}_0 = \mathbf{p}_{n-1}$

While digital artists call it **Wireframe, …**

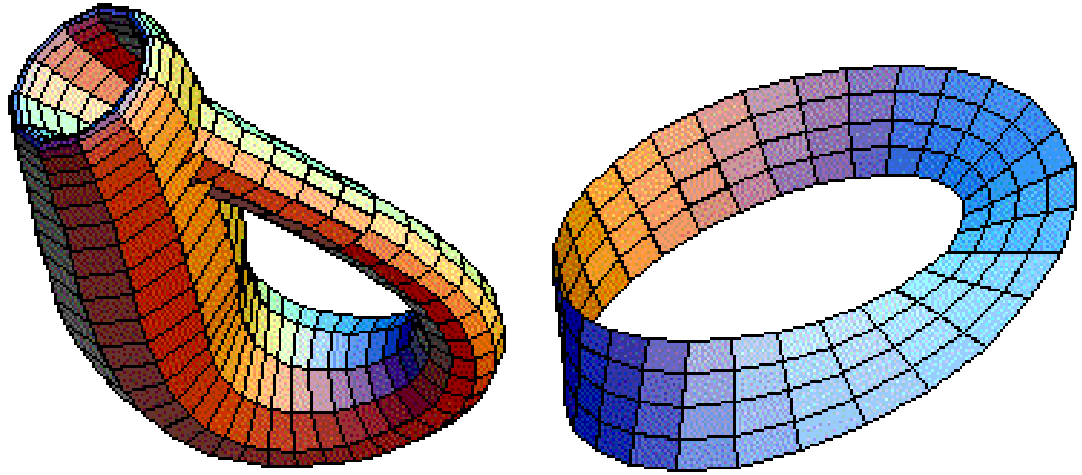
Where Meshes Come From

- Model manually
  - Write out all polygons
  - Write some code to generate them
  - Interactive editing: move vertices in space
- Acquisition from real objects
  - 3D scanners, vision systems
  - Generate set of points on the surface
  - Need to convert to polygons

Orientation of Faces

- A mesh is well oriented (orientable) if all faces can be oriented consistently (all CCW or all CW) such that each edge has two opposite orientations for its two adjacent faces

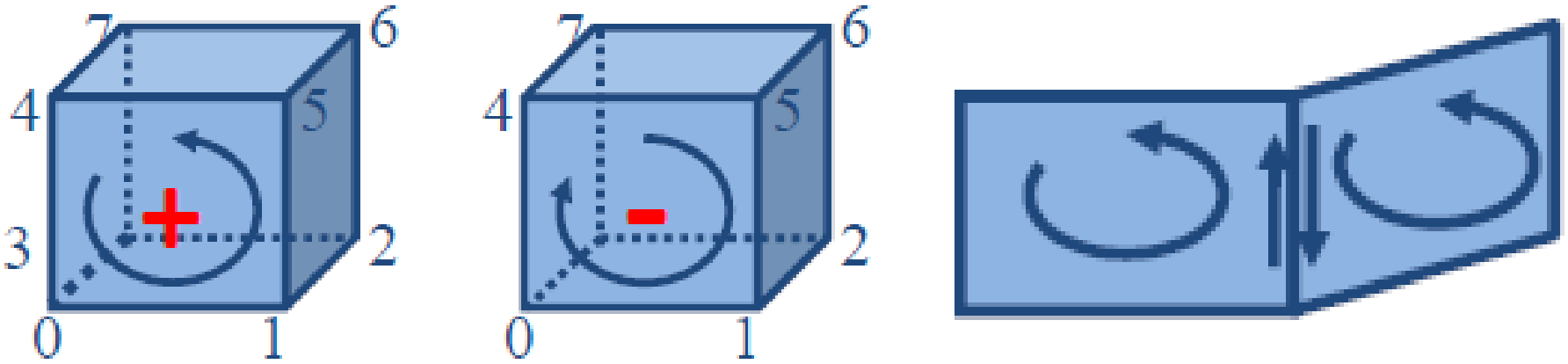
- Not every mesh can be well oriented.
  e.g. Klein bottle, Möbius strip



non-orientable surfaces

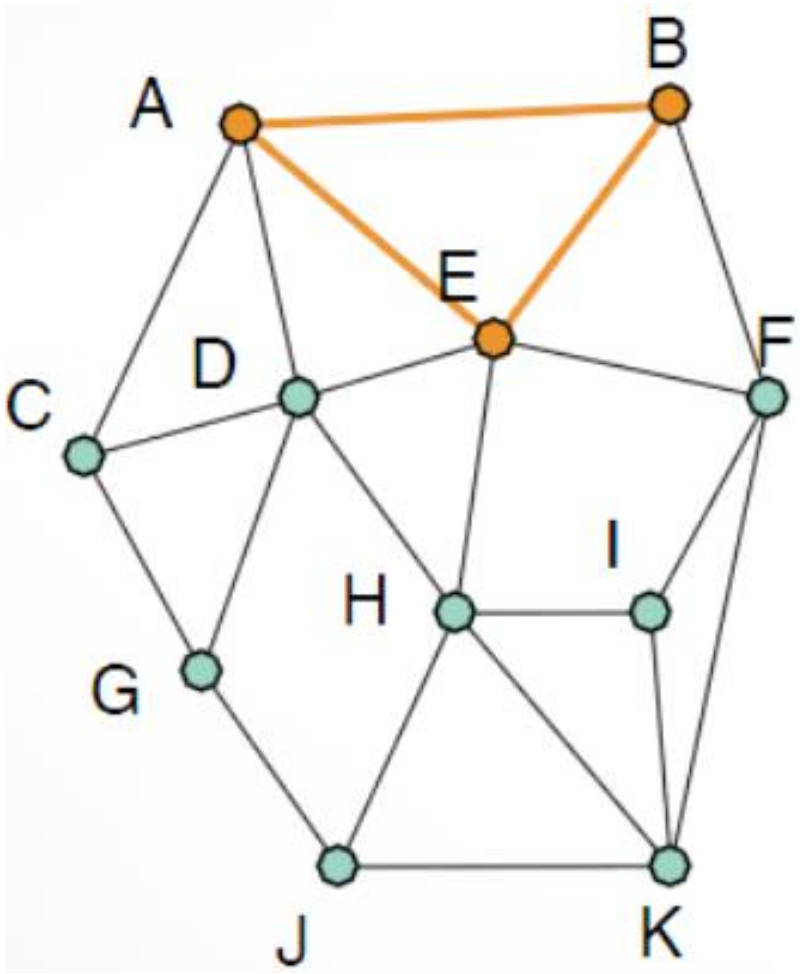
Orientation of Faces

- Each face can be assigned an orientation by defining the ordering of its vertices
- Orientation can be clockwise or counter-clockwise. The orientation determines the normal direction of face. Usually counterclockwise order is the “front” side.



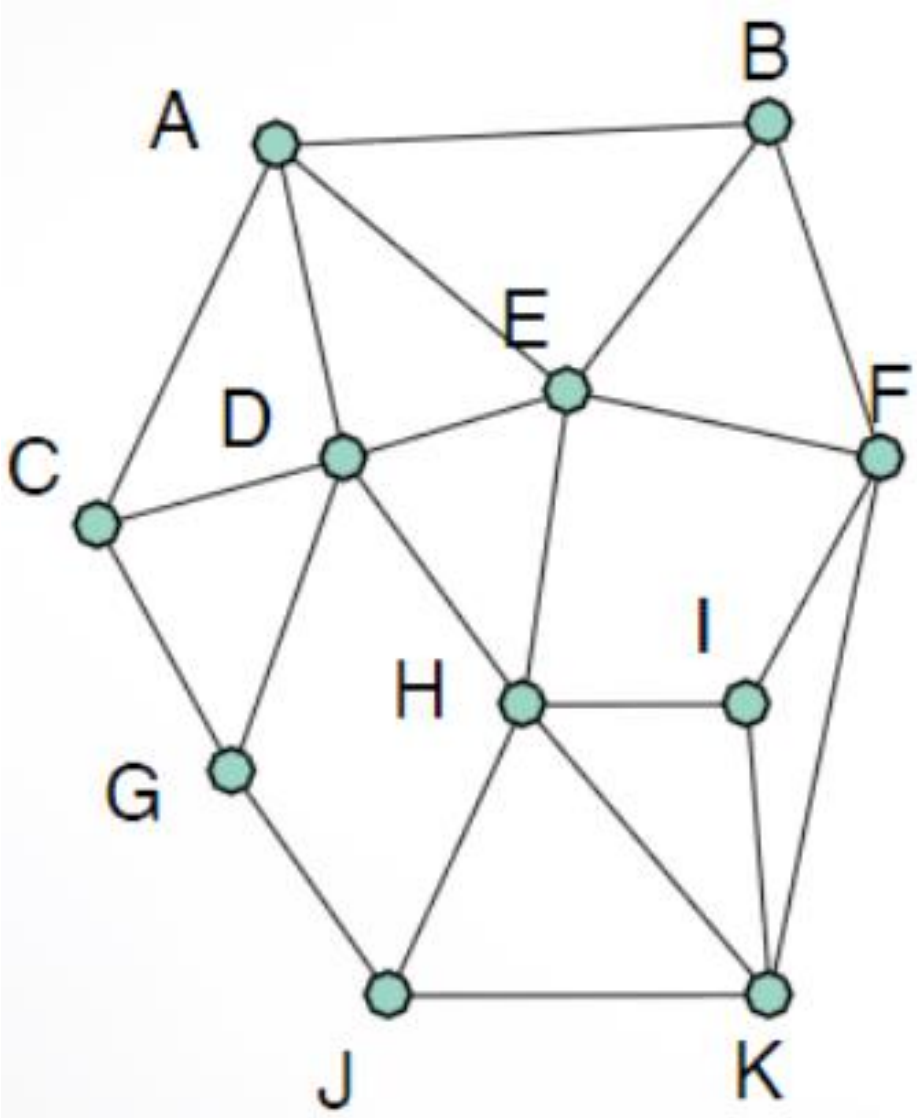
- Two neighboring facets are **equally oriented**, if the edge directions of the shared edge (induced by the face orientation) are opposing
- A polygonal mesh is **orientable**, if the incident faces to every edge can be equally oriented.

Graph Definitions



- Graph $\{V,E\}$
- Vertices $V = \{A,B,C,\ldots,K\}$
- Edges $E = \{(AB),(AE),(CD),\ldots\}$
- Faces $F = \{(ABE),(EBF),(EFIH),\ldots\}$

Graph Definitions

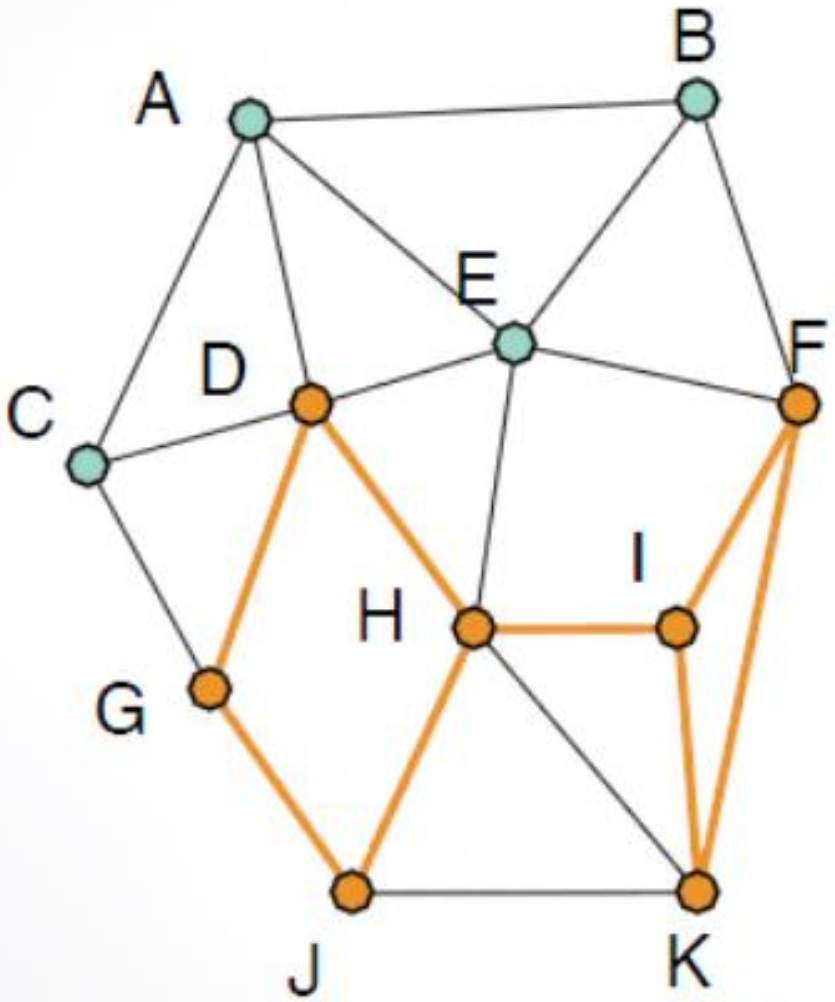


Vertex degree or valence:

number of incident edges

- deg(A) = 4
- deg(E) = 5

Connectivity



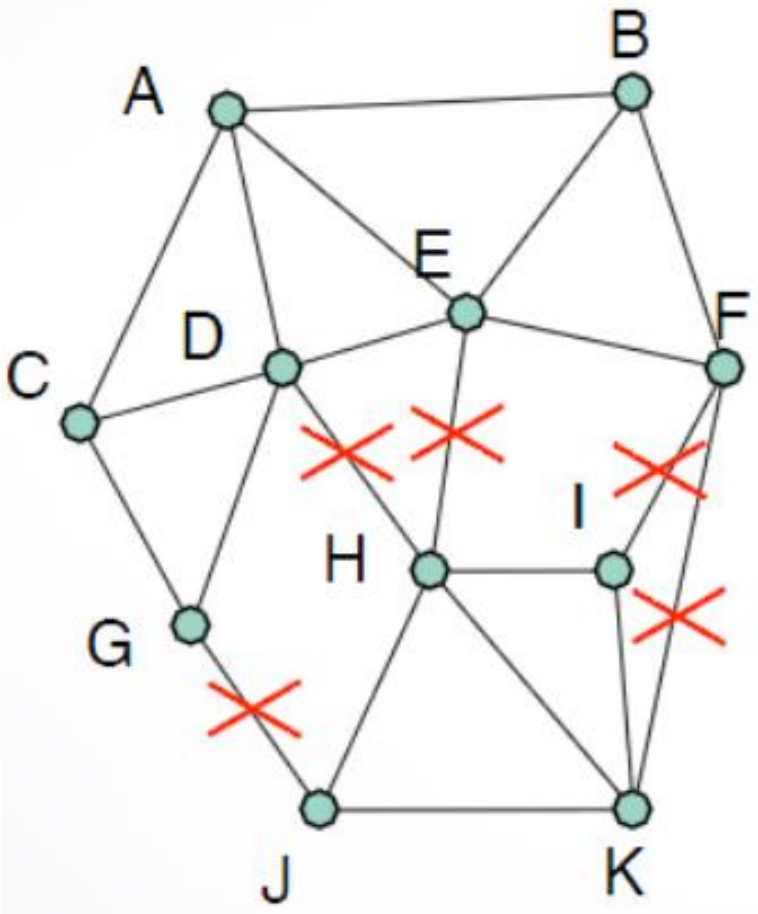
Connected:

Path of edges connecting every two vertices

Subgraph:

Graph $\{V',E'\}$ is a subgraph of graph $\{V,E\}$ if V' is a subset of V and E' is a subset of E incident on V'.

Connectivity



Connected:

Path of edges connecting every two vertices

Subgraph:

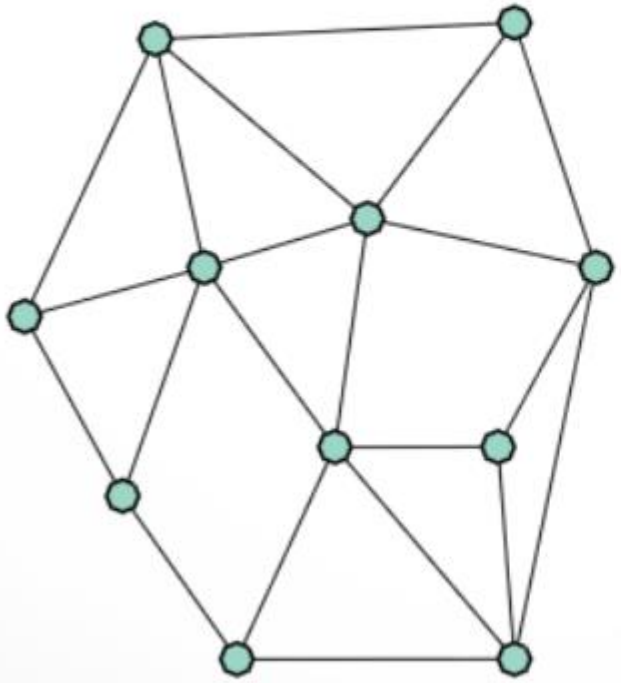
Graph $\{V',E'\}$ is a subgraph of graph $\{V,E\}$ if V' is a subset of V and E' is a subset of E incident on V'.

Connected Components:

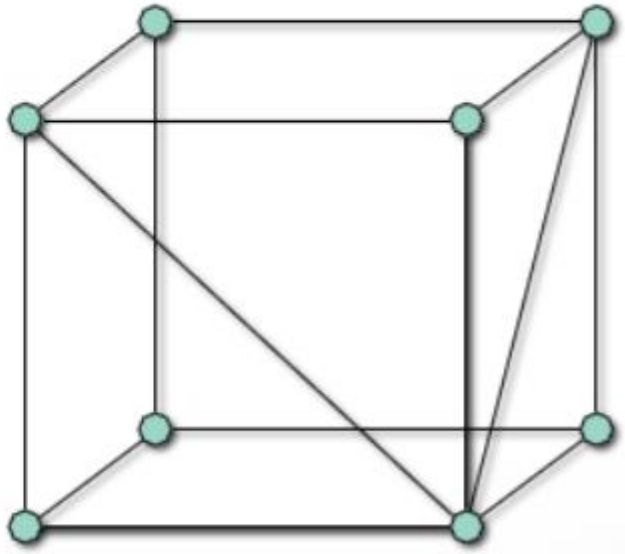
Maximally connected subgraph

Graph Embedding

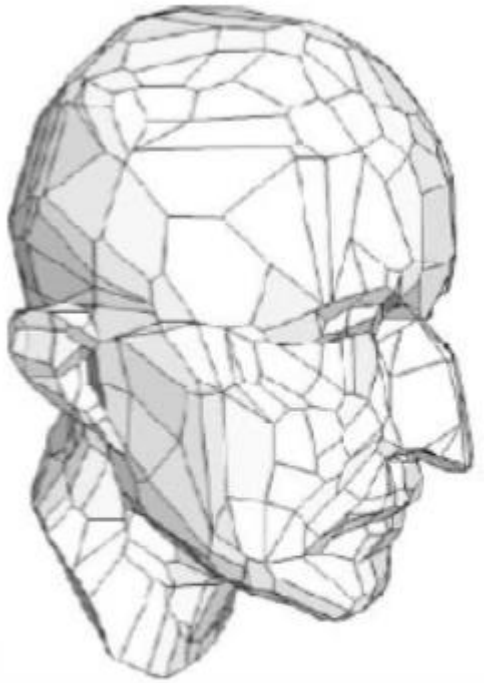
Embedding: Graph is **embedded** in $\mathbb{R}^d$, if each vertex is assigned a position in $\mathbb{R}^d$.



Embedding in $\mathbb{R}^2$



Embedding in $\mathbb{R}^3$

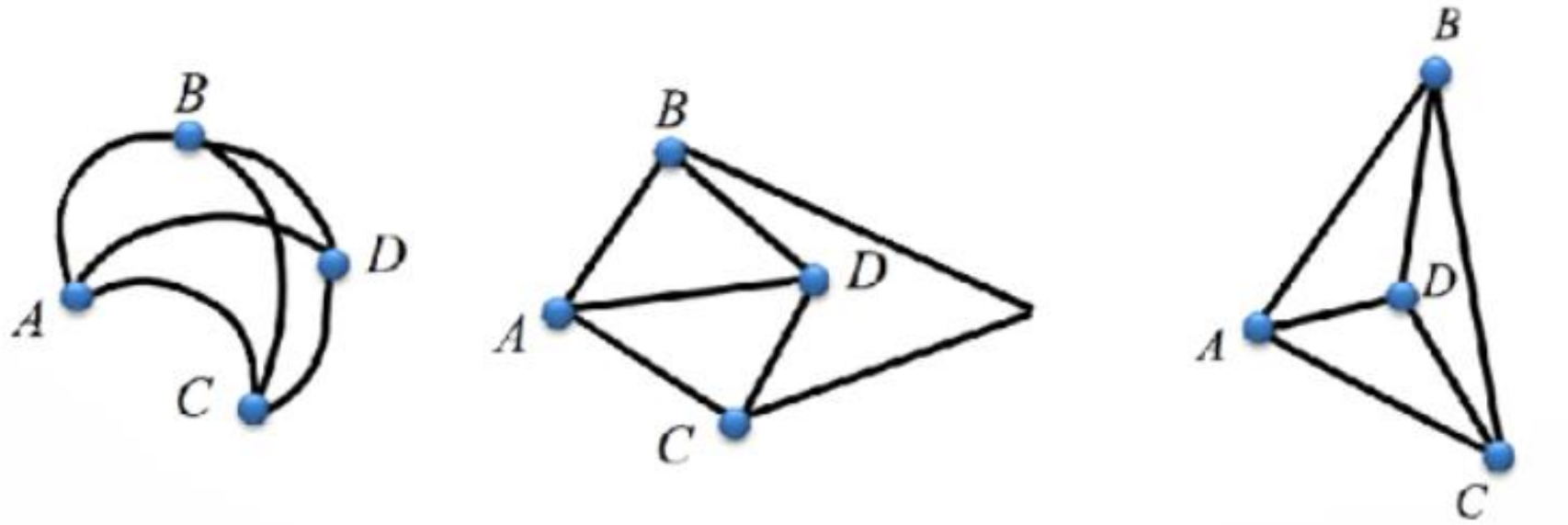


Embedding in $\mathbb{R}^3$

Planar Graph

Planar Graph

Graph whose vertices and edges **can be** embedded in $\mathbb{R}^2$ such that its edges do not intersect

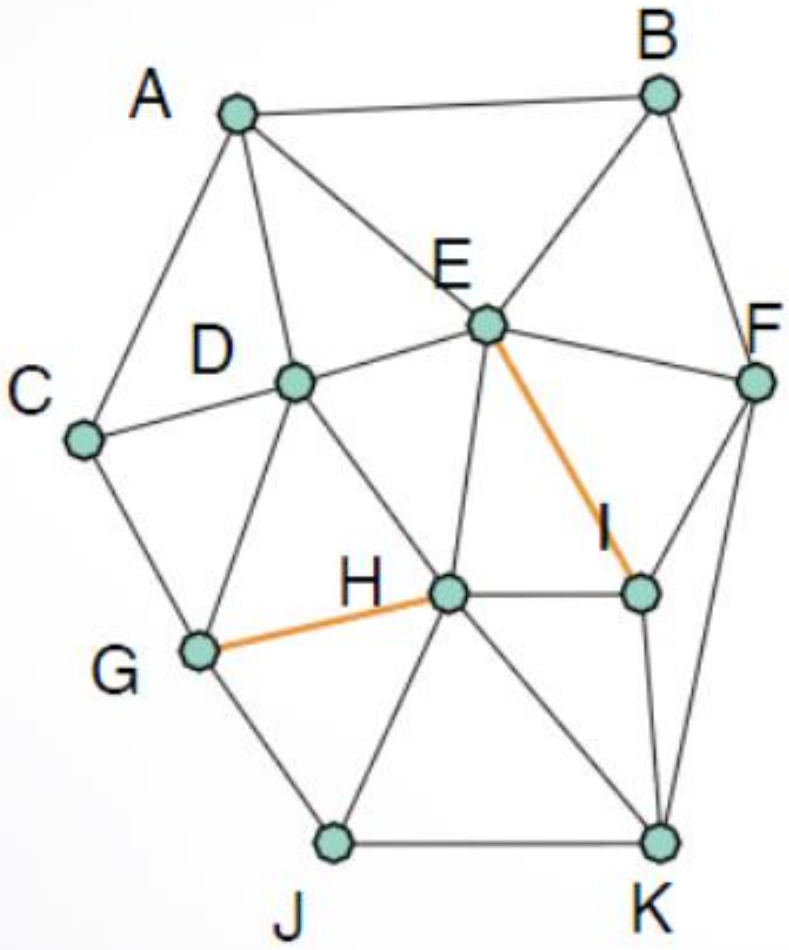


Planar Graph

Plane Graph

Straight Line Plane Graph

Triangulation



Triangulation:

Straight line plane graph where every face is a triangle

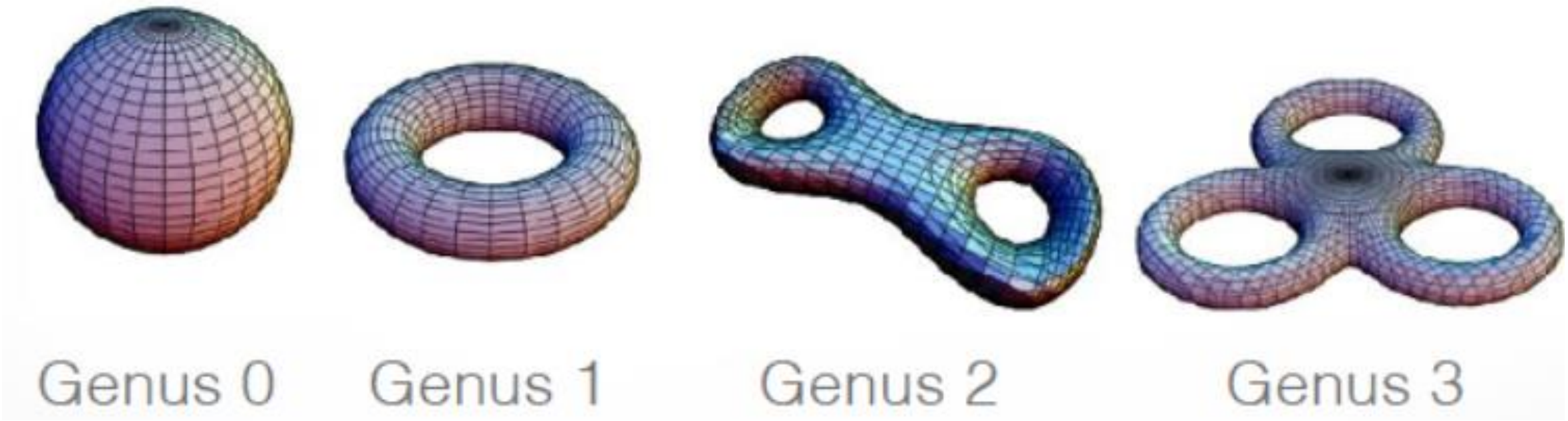
Why?

- simple homogenous data structure
- efficient rendering
- simplifies algorithms
- by definition, triangle is planar
- any polygon can be triangulated

Global Topology: **Genus**

- Genus: Maximal number of closed simple cutting curves that do not disconnect the graph into multiple components.

- Informally, the number of holes or handles



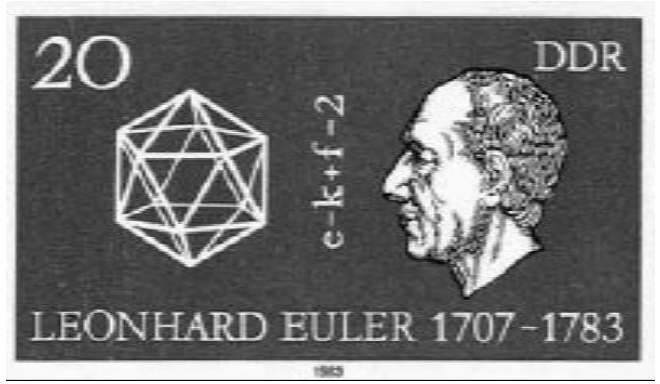
A disc (plane with boundary) / planar graph has genus zero

Euler-Poincaré Formula

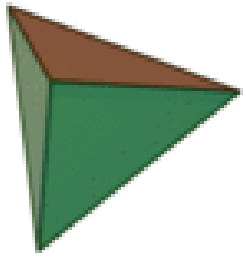
- The relation between the number of vertices, edges, and faces.

$$V - E + F = 2$$

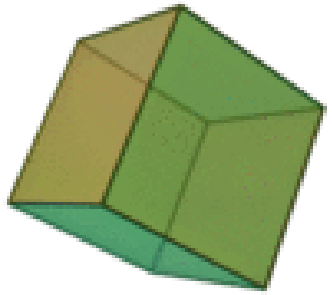
- where
  - V : number of vertices
  - E : number of edges
  - F : number of faces



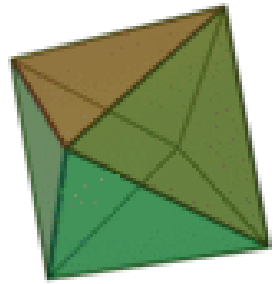
Euler Formula



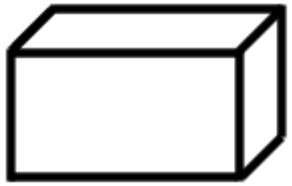
- Tetrahedron
  - V = 4
  - E = 6
  - F = 4
  - 4 - 6 + 4 = 2



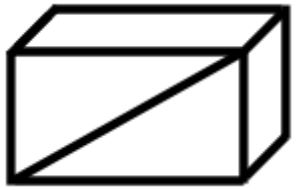
- Cube
  - V = 8
  - E = 12
  - F = 6
  - 8 -12 + 6 = 2



- Octahedron
  - V = 6
  - E = 12
  - F = 8
  - 6 -12 + 8 = 2



V = 8
E = 12
F = 6
8 - 12 + 6 = 2

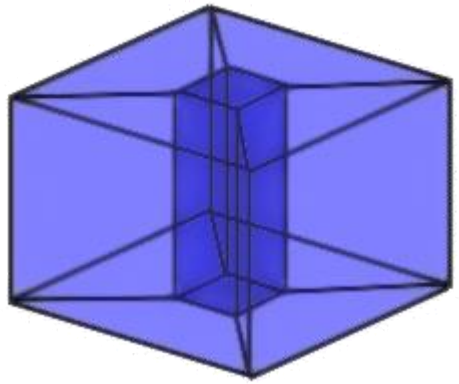


V = 8
E = 12 + 1 = 13
F = 6 + 1 = 7
8 - 13 + 7 = 2

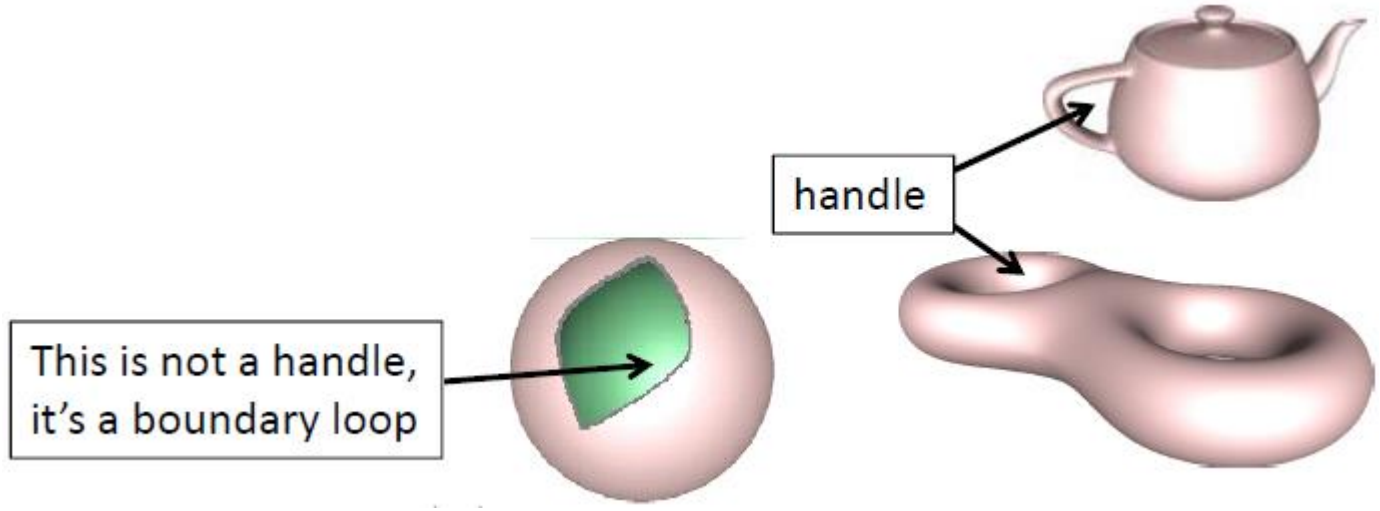
Euler-Poincaré Formula

- More general rule **Euler characteristic χ= V-E+F=2(C-G)-B**

- where
  - V : number of vertices
  - E : number of edges
  - F : number of faces
  - C : number of connected components
  - G : number of genus (holes, handles)
  - B : number of boundaries



V = 16
E = 32
F = 16
C = 1
G = 1
B = 0
16 – 32 + 16 = 2 (1 - 1) - 0



Average Valence of Closed Triangle Mesh

- **Theorem:** Average vertex degree in a closed manifold triangle mesh is ~6

- **Proof:** Each face has 3 edges and each edge is counted twice: $3F = 2E$

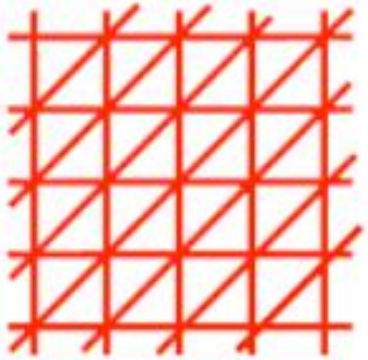
- by Euler's formula: $V+F-E = V+2E/3-E = 2-2g$
- Thus $E = 3(V-2+2g)$

- So average degree $= 2E/V = 6(V-2+2g)/V \sim 6$ for large V

Euler Consequences

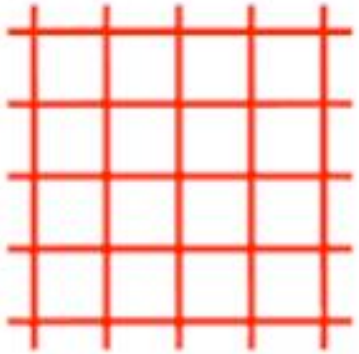
Triangle mesh statistics

- $F \approx 2V$
- $E \approx 3V$
- Average valence = 6



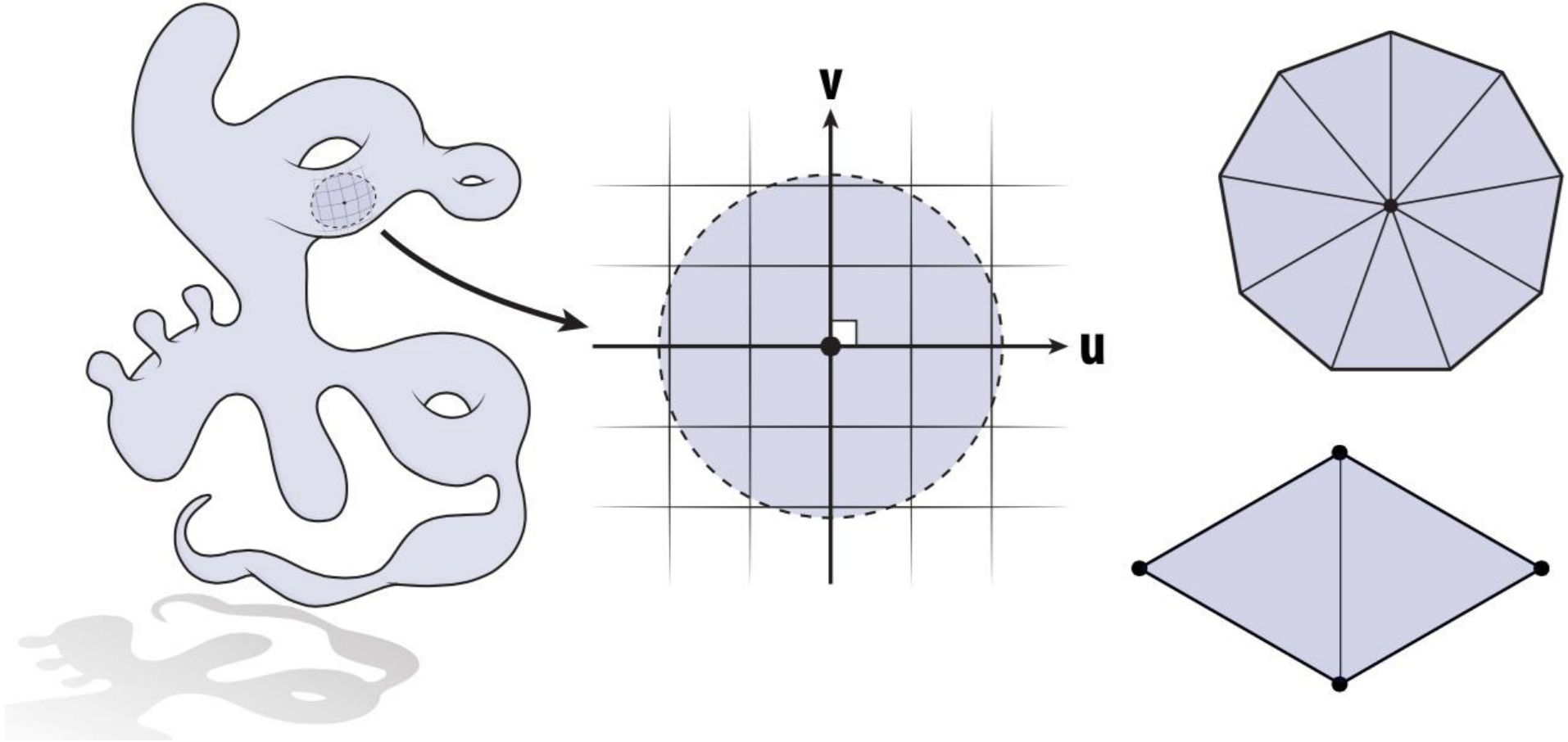
Quad mesh statistics

- $F \approx V$
- $E \approx 2V$
- Average valence = 4



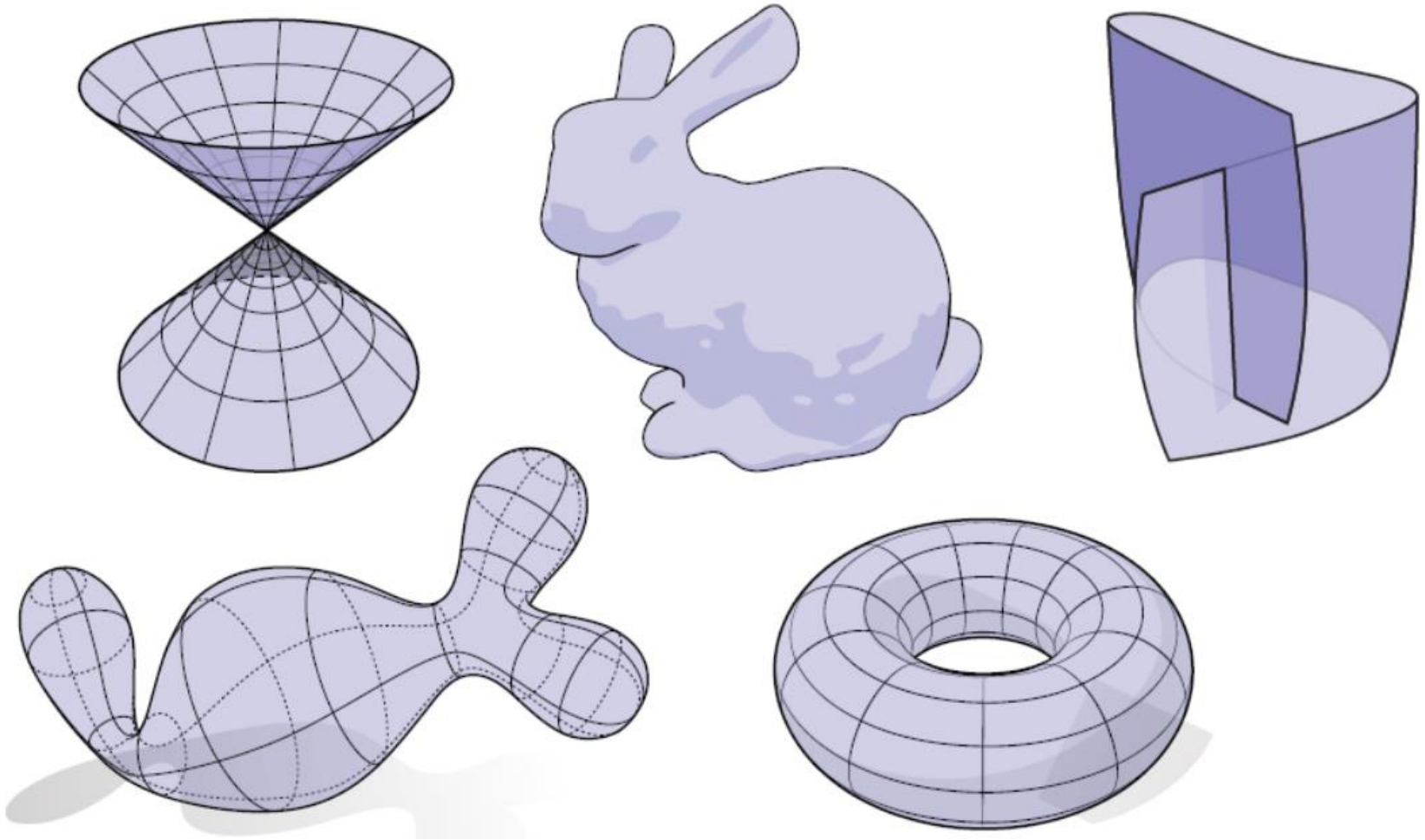
Manifold assumption

- **Today we're going to introduce the idea of *manifold* geometry**
- **Can be hard to understand motivation at first!**
- **So first, let's revisit a more familiar example...**



Isn’t every shape manifold?

- **Which of these shapes are manifold?**

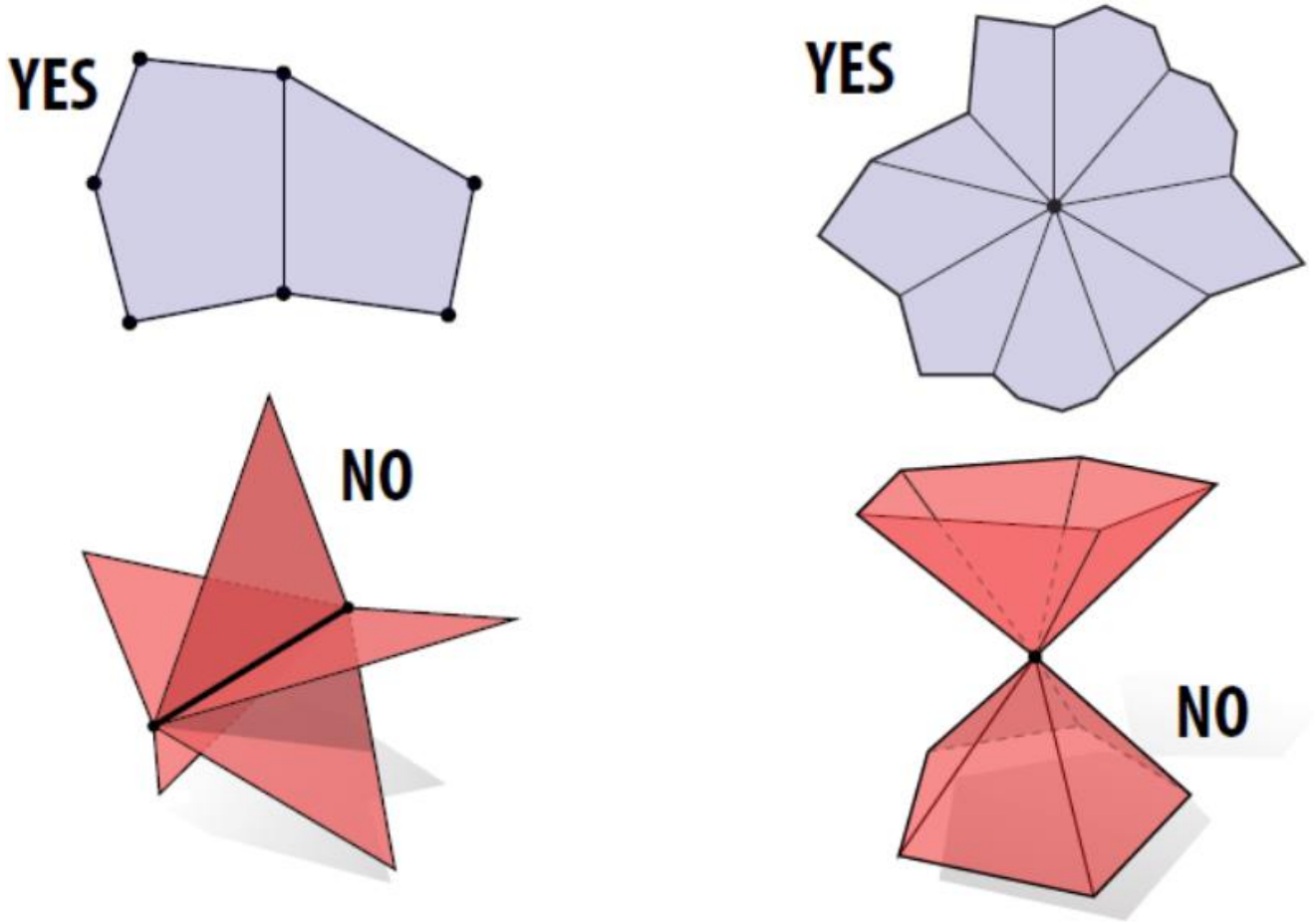


Center point never looks like the plane, no matter how close we get.

A manifold polygon mesh has fans, not fins

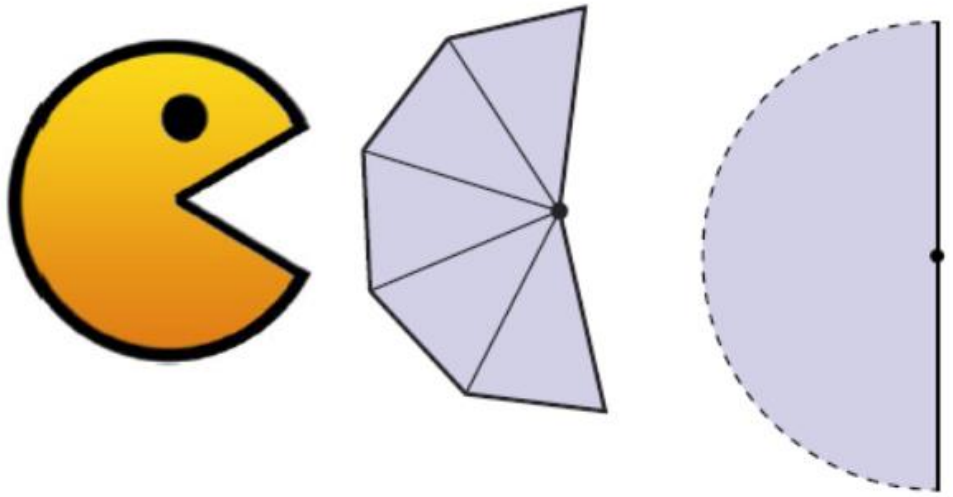
For polygonal surfaces just two easy conditions to check:

1. **Every edge is contained in only two polygons (no “fins”)**
2. **The polygons containing each vertex make a single “fan”**



What about boundary?

- The boundary is where the surface “ends.”
- E.g., waist & ankles on a pair of pants.
- Locally, looks like a *half* disk
- Globally, each boundary forms a loop

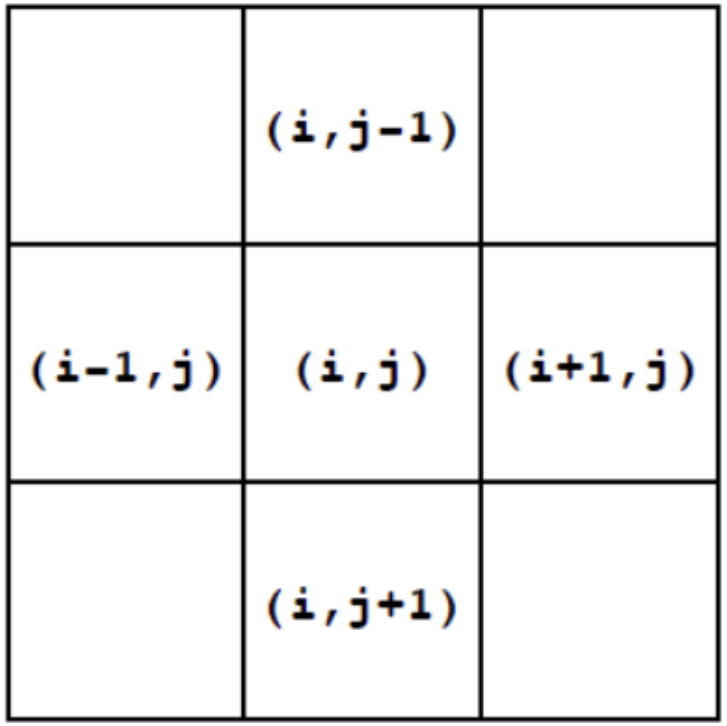


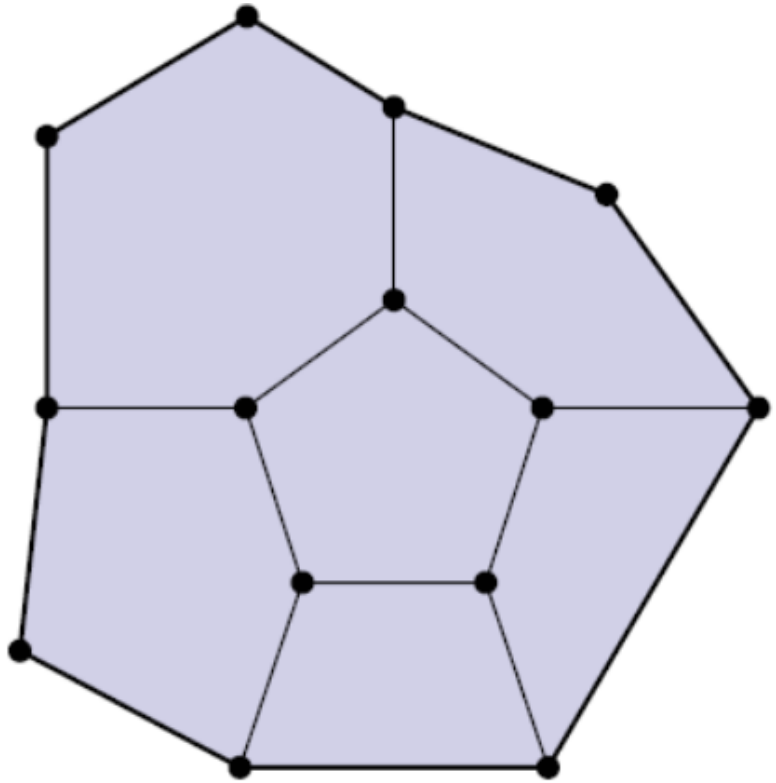
- Polygon mesh:
- - one polygon per boundary edge
- - boundary vertex looks like “pacman”

Ok, but why is the manifold assumption *useful?*

Keep it Simple!

- **Same motivation as for images:**
  - **make some assumptions about our geometry to keep data structures/algorithms simple and efficient**
  - **in *many common cases*, doesn't fundamentally limit what we can do with geometry**

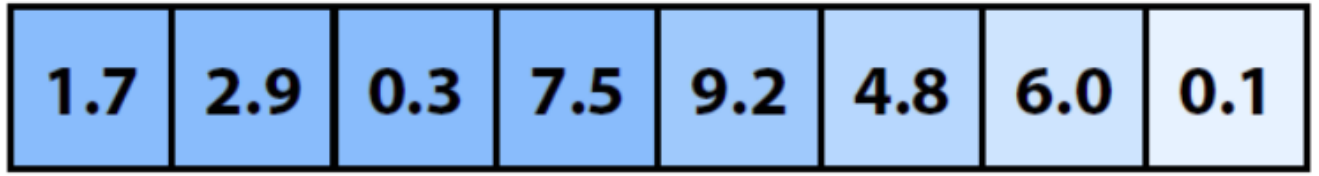




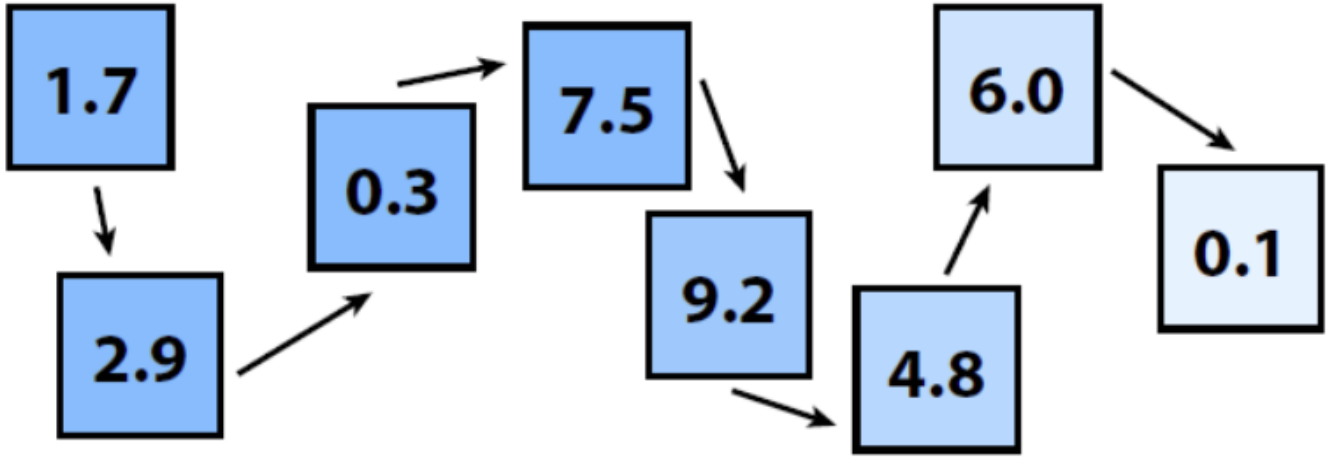
How do we actually encode all this data?

Warm up: storing numbers

- Q: What data structures can we use to store a list of numbers?
- One idea: use an *array* (constant time lookup, coherent access)



- Alternative: use a linked list (linear lookup, incoherent access)



- Q: Why bother with the linked list?
- A: For one, we can easily insert numbers wherever we like...

Polygon Soup (Array-like)

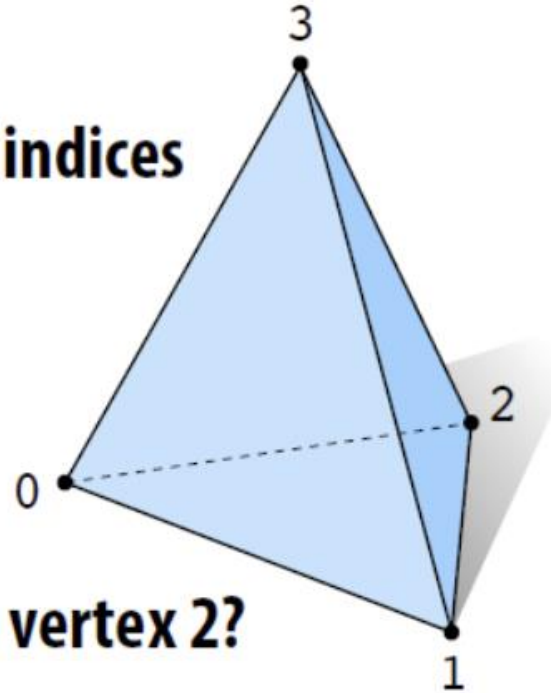
- Store triples of coordinates (x,y,z), tuples of indices
- E.g., tetrahedron:

VERTICES

	x	y	z
0:	−1	−1	−1
1:	1	−1	1
2:	1	1	−1
3:	−1	1	1

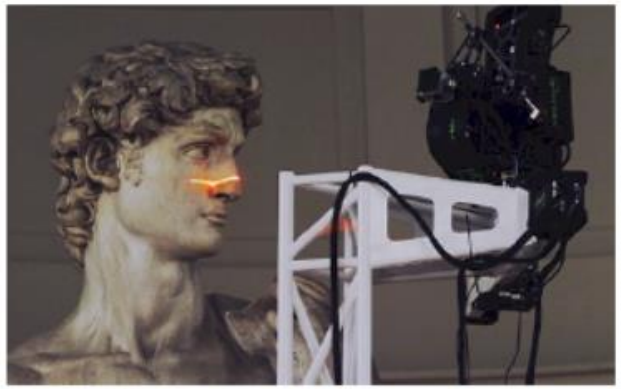
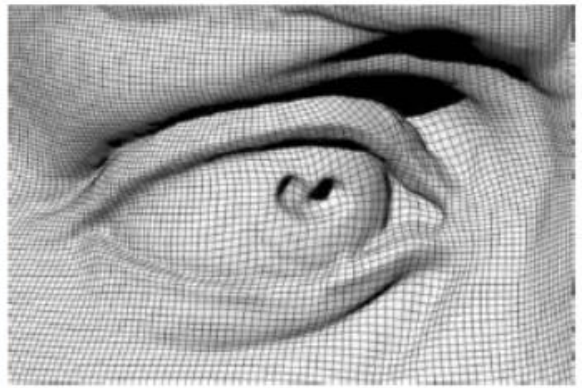
POLYGONS

	i	j	k
	0	2	1
	0	3	2
	3	0	1
	3	1	2



- Q: How do we find all the polygons touching vertex 2?
- Ok, now consider a more complicated mesh:

~1 *billion* polygons

Very expensive to find the neighboring triangles! (What's the cost?)

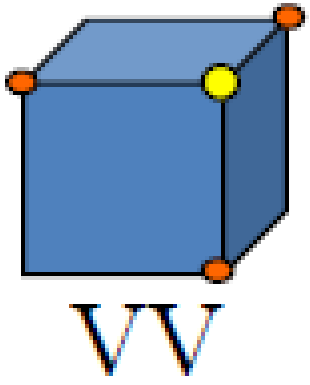
Adjacency Relationships

Definition 3 (Vertex 1-ring). *The vertex 1-ring of a vertex $i \in V$ is*

$$V_i \overset{\text{def.}}{=} \{j \in V \setminus (i,j) \in E\} \subset V.$$

The s-ring is defined by induction as

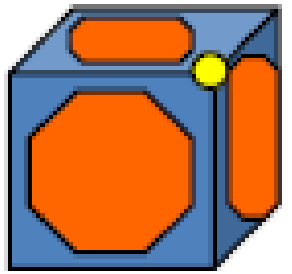
$$\forall\, s > 1, \quad V_i^{(s)} = \left\{ j \in V \setminus (k,j) \in E \quad \textit{and} \quad k \in V_i^{(s-1)} \right\}.$$



VV

Definition 4 (Face 1-ring). *The face 1-ring of a vertex $i \in V$ is*

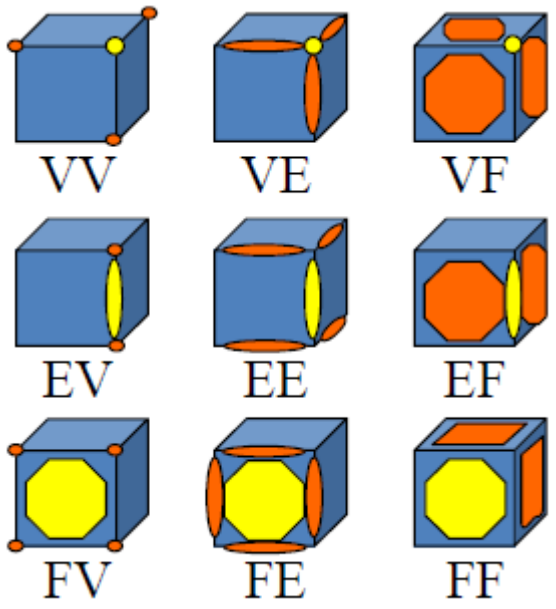
$$F_i \overset{\text{def.}}{=} \{(i,j,k) \in F \setminus i,j \in V\} \subset F.$$

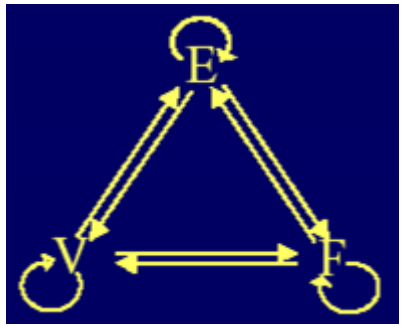


VF

Neighborhood relations [Weiler 1985]

1. Vertex – Vertex VV
2. Vertex – Edge VE
3. Vertex – Face VF
4. Edge – Vertex EV
5. Edge – Edge EE
6. Edge – Face EF
7. Face – Vertex FV
8. Face – Edge FE
9. Face – Face FF





Knowing some types of relation, we can discover other (but not necessary all) topological information
e.g. if in addition to VV, VE and VF, we know neighboring vertices of a face, we can discover all neighboring edges of the face

Incidence Matrices

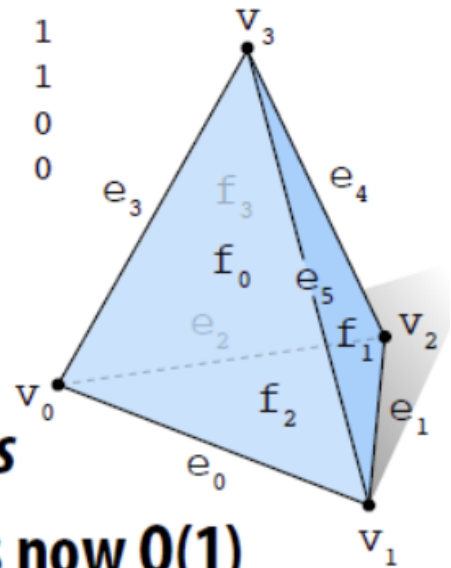
- If we want to answer neighborhood queries, why not simply store a list of neighbors?
- Can encode all neighbor information via *incidence matrices*
- E.g., tetrahedron:

VERTEX➡EDGE

	v0	v1	v2	v3
e0	1	1	0	0
e1	0	1	1	0
e2	1	0	1	0
e3	1	0	0	1
e4	0	0	1	1
e5	0	1	0	1

EDGE➡FACE

	e0	e1	e2	e3	e4	e5
f0	1	0	0	1	0	1
f1	0	1	0	0	1	1
f2	1	1	1	0	0	0
f3	0	0	1	1	1	0



- 1 means “touches”; 0 means “does not touch”
- Instead of storing lots of 0’s, use *sparse matrices*
- Still large storage cost, but finding neighbors is now O(1)
- Hard to change connectivity, since we used fixed indices
- Bonus feature: mesh does not have to be manifold

Mesh Representations

- Representations
  - Face-vertex meshes
    - Problem: different topological structure for triangles and quadrangles
  - Winged-edge meshes
    - Problem: traveling the neighborhood requires one case distinction
  - Half-edge meshes
  - Quad-edge meshes, Corner-tables, Vertex-vertex meshes, …
  - LR (*Laced Ring*): more compact than halfedge [siggraph2011: compact connectivity representation for triangle meshes]
    - Suited for processing meshes with fixed connectivity

Mesh Representations

- Choice
  - Each of the representations above have particular advantages & drawbacks
  - Choice is governed by
    - Application,
    - Performance required,
    - Size of the data,
    - and Operations to be performed.

- Example
  - it is easier to deal with triangles than general polygons, especially in computational geometry.
  - For certain operations it is necessary to have a fast access to topological information such as edges or neighboring faces; this requires more complex structures such as half-edge representation.
  - For hardware rendering, compact, simple structures are needed; thus the corner-table (triangle fan) is commonly incorporated into low-level rendering APIs such as DirectX and OpenGL.

Mesh Data Structures

- How to store geometry & connectivity?
- compact storage and file formats
- Efficient algorithms on meshes
  - Time-critical operations
  - **All vertices/edges of a face**
  - **All incident vertices/edges/faces of a vertex**

Data Structures

- **What should be stored?**
- Geometry: 3D vertex coordinates
- Connectivity: Vertex adjacency
- Attributes:
  - normals, color, texture coordinates, etc.
  - Per Vertex, per face, per edge

Mesh Data Structures

- **What should it support?**
  - Rendering
  - Queries
    - What are the vertices of face #3?
    - Is vertex #6 adjacent to vertex #12?
    - Which faces are adjacent to face #7?
  - Modifications
    - Remove/add a vertex/face
    - Vertex split, edge collapse

Different Data Structures

- Time to construct (preprocessing)
- Time to answer a query
  - Random access to vertices/edges/faces
  - Fast mesh traversal
  - Fast Neighborhood query
- Time to perform an operation
  - split/merge
- Space complexity
- Redundancy
- Most important ones are **face and edge-based (since they encode connectivity)**

Face Set (STL)

- **Face:**
  - 3 vertex positions

Triangles		
x_{11} y_{11} z_{11}	x_{12} y_{12} z_{12}	x_{13} y_{13} z_{13}
x_{21} y_{21} z_{21}	x_{22} y_{22} z_{22}	x_{23} y_{23} z_{23}
...	...	...
x_{F1} y_{F1} z_{F1}	x_{F2} y_{F2} z_{F2}	x_{F3} y_{F3} z_{F3}

9*4 = 36 B/f (single precision)
72 B/v (Euler Poincaré)

No explicit connectivity

Shared Vertex (OBJ,OFF)

- **Indexed Face List:**
  - Vertex: position
  - Face: Vertex Indices

Vertices
x_1 y_1 z_1
...
x_V y_V z_V

Triangles
i_{11} i_{12} i_{13}
...
...
...
...
i_{F1} i_{F2} i_{F3}

12 B/v + 12 B/f = 36B/v

No explicit adjacency info

Face-vertex meshes

1. a set of faces and a set of vertices.
2. **most widely used, being the input typically accepted by modern graphics hardware.**
3. **One-to-one correspondence with OBJ**

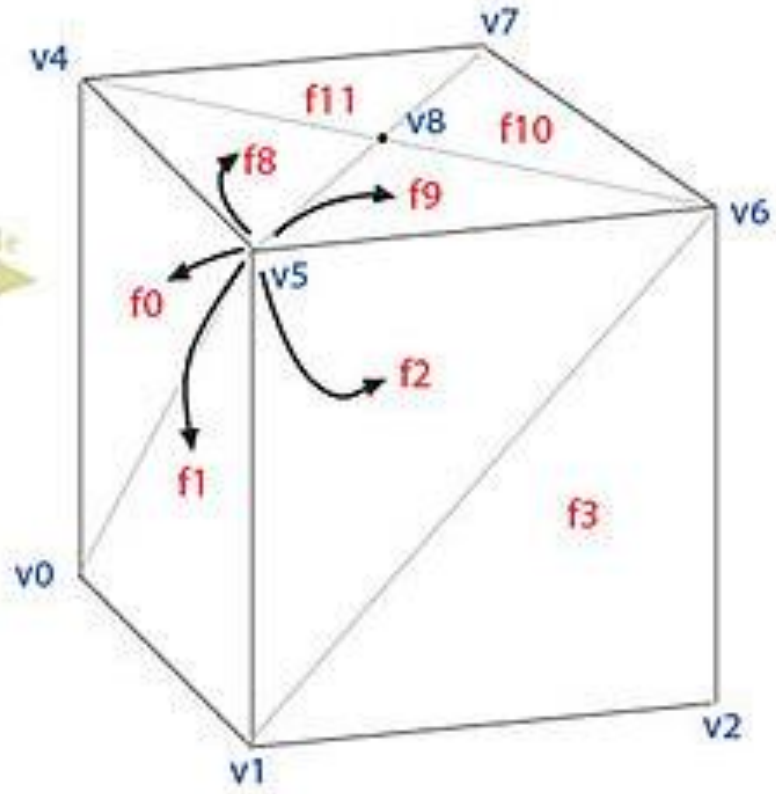
Face List

f0	v0 v4 v5
f1	v0 v5 v1
f2	v1 v5 v6
f3	v1 v6 v2
f4	v2 v6 v7
f5	v2 v7 v3
f6	v3 v7 v4
f7	v3 v4 v0
f8	v8 v5 v4
f9	v8 v6 v5
f10	v8 v7 v6
f11	v8 v4 v7
f12	v9 v5 v4
f13	v9 v6 v5
f14	v9 v7 v6
f15	v9 v4 v7

Vertex List

v0	0,0,0	f0 f1 f12 f15 f7
v1	1,0,0	f2 f3 f13 f12 f1
v2	1,1,0	f4 f5 f14 f13 f3
v3	0,1,0	f6 f7 f15 f14 f5
v4	0,0,1	f6 f7 f0 f8 f11
v5	1,0,1	f0 f1 f2 f9 f8
v6	1,1,1	f2 f3 f4 f10 f9
v7	0,1,1	f4 f5 f6 f11 f10
v8	.5,.5,0	f8 f9 f10 f11
v9	.5,.5,1	f12 13 14 15

example



Face-vertex meshes

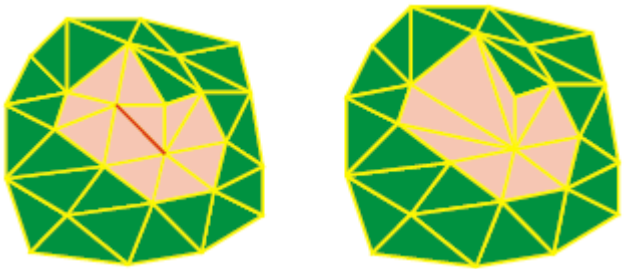
1. locating neighboring faces and vertices is constant time
2. a search is still needed to find all the faces surrounding a given face.
3. Other dynamic operations, such as splitting or merging a face, are also difficult with face-vertex meshes.

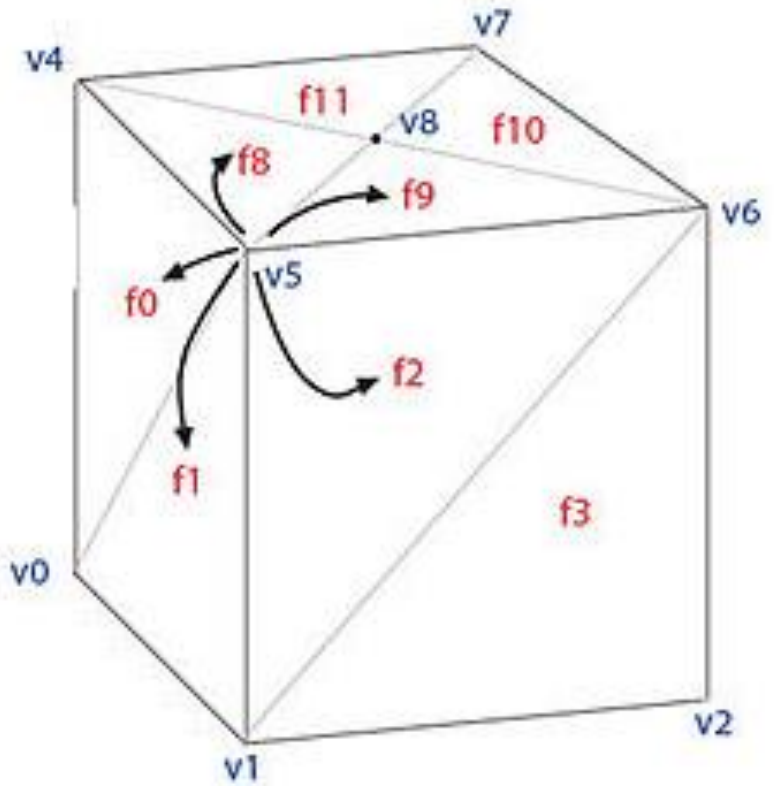
Face List

f0	v0 v4 v5
f1	v0 v5 v1
f2	v1 v5 v6
f3	v1 v6 v2
f4	v2 v6 v7
f5	v2 v7 v3
f6	v3 v7 v4
f7	v3 v4 v0
f8	v8 v5 v4
f9	v8 v6 v5
f10	v8 v7 v6
f11	v8 v4 v7
f12	v9 v5 v4
f13	v9 v6 v5
f14	v9 v7 v6
f15	v9 v4 v7

Vertex List

v0	0,0,0	f0 f1 f12 f15 f7
v1	1,0,0	f2 f3 f13 f12 f1
v2	1,1,0	f4 f5 f14 f13 f3
v3	0,1,0	f6 f7 f15 f14 f5
v4	0,0,1	f6 f7 f0 f8 f11
v5	1,0,1	f0 f1 f2 f9 f8
v6	1,1,1	f2 f3 f4 f10 f9
v7	0,1,1	f4 f5 f6 f11 f10
v8	.5,.5,0	f8 f9 f10 f11
v9	.5,.5,1	f12 13 14 15



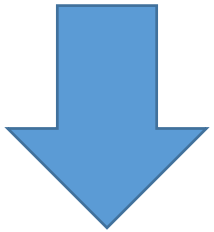


Transversal operations

- Most operations are slow for the connectivity info is not explicit.
- Need a more efficient representation

iterate over / collect adjacent	V	E	F
V	quadratic	quadratic	linear
E	quadratic	quadratic	linear
F	quadratic	quadratic	linear

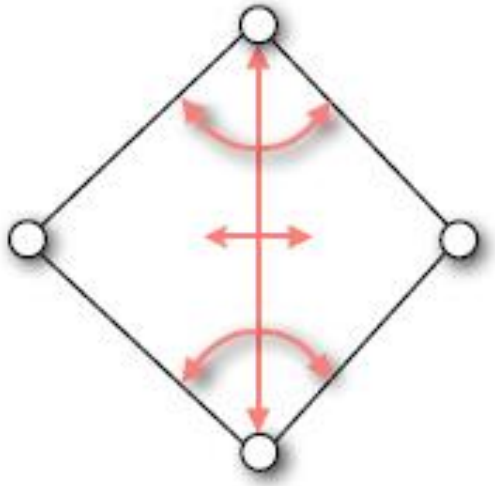
Edges always have the same topological structure



Efficient handling of polygons with variable valence

(Winged) **Edge-Based Connectivity**

- **Vertex:**
  - position
  - 1 edge
- **Edge:**
  - 2 vertices
  - 2 faces
  - 4 edges
- **Face:**
  - 1 edge



120 B/v

Edges have no orientation: special case handling for neighbors

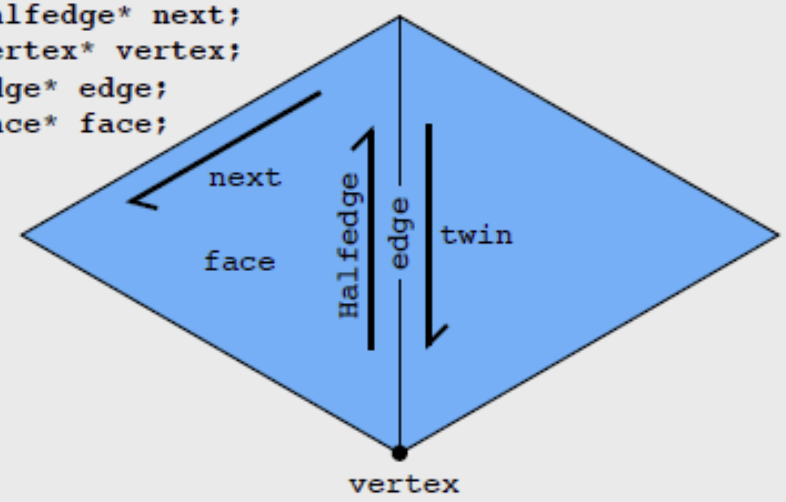
Different Data Structures

- Time to construct (preprocessing)
- Time to answer a query
  - Random access to vertices/edges/faces
  - Fast mesh traversal
  - Fast Neighborhood query
- Time to perform an operation
  - split/merge
- Space complexity
- Redundancy
- Most important ones are **face and edge-based (since they encode connectivity)**

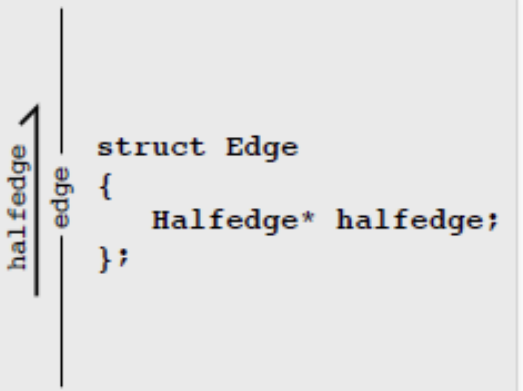
Halfedge-Based Connectivity

- Store *some* information about neighbors
- Don’t need an exhaustive list; just a few key pointers
- Key idea: two *halfedges* act as “glue” between mesh elements:

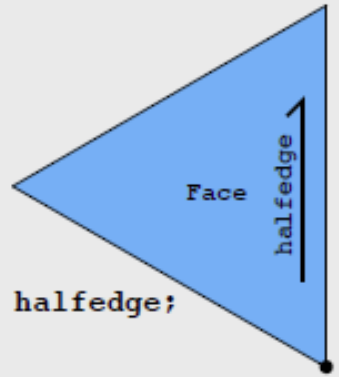
```
struct Halfedge
{
   Halfedge* twin;
   Halfedge* next;
   Vertex* vertex;
   Edge* edge;
   Face* face;
};
```



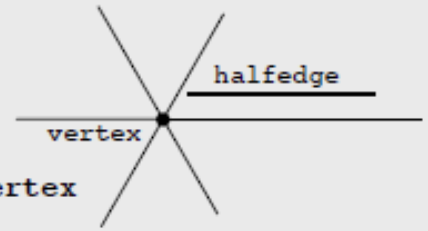

```
struct Edge
{
   Halfedge* halfedge;
};
```

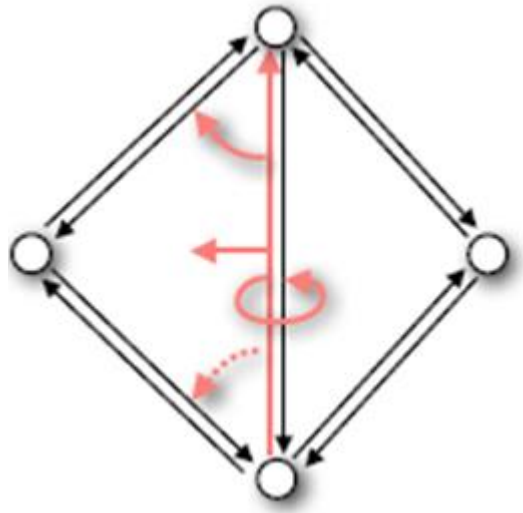



```
struct Face
{
   Halfedge* halfedge;
};
```




```
struct Vertex
{
   Halfedge* halfedge;
};
```



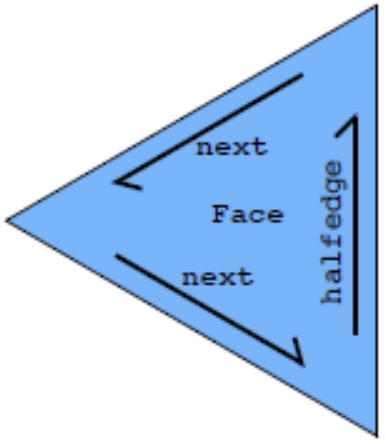


- Each vertex, edge face points to just *one* of its halfedges

Halfedge makes mesh traversal easy

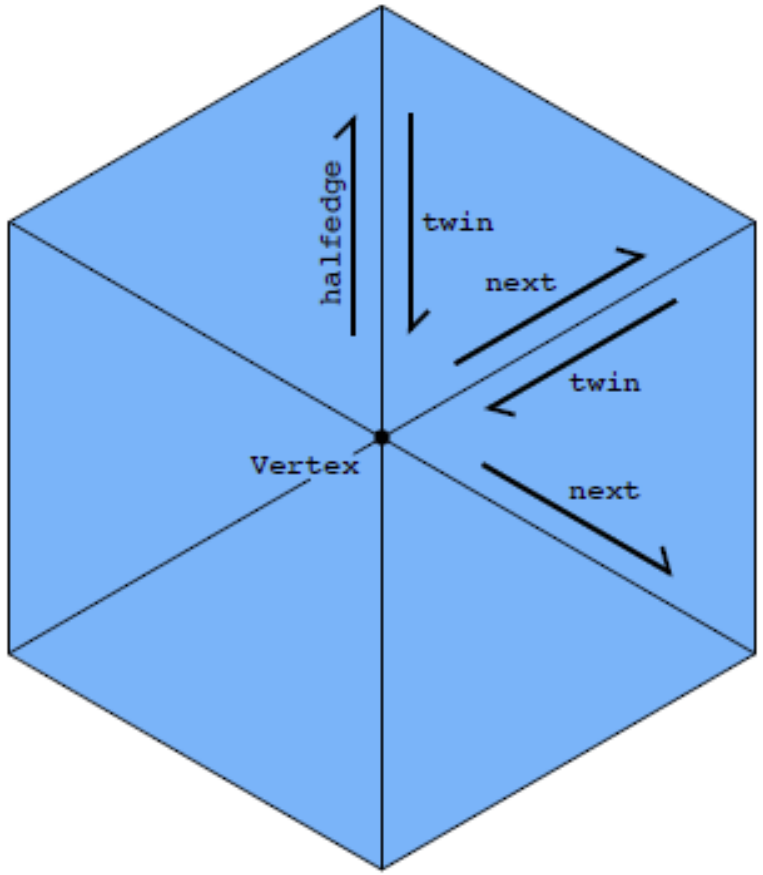
- Use “twin” and “next” pointers to move around mesh
- Use “vertex”, “edge”, and “face” pointers to grab element
- Example: visit all vertices of a face:

```
Halfedge* h = f->halfedge;
do {
   h = h->next;
}
while( h != f->halfedge );
```



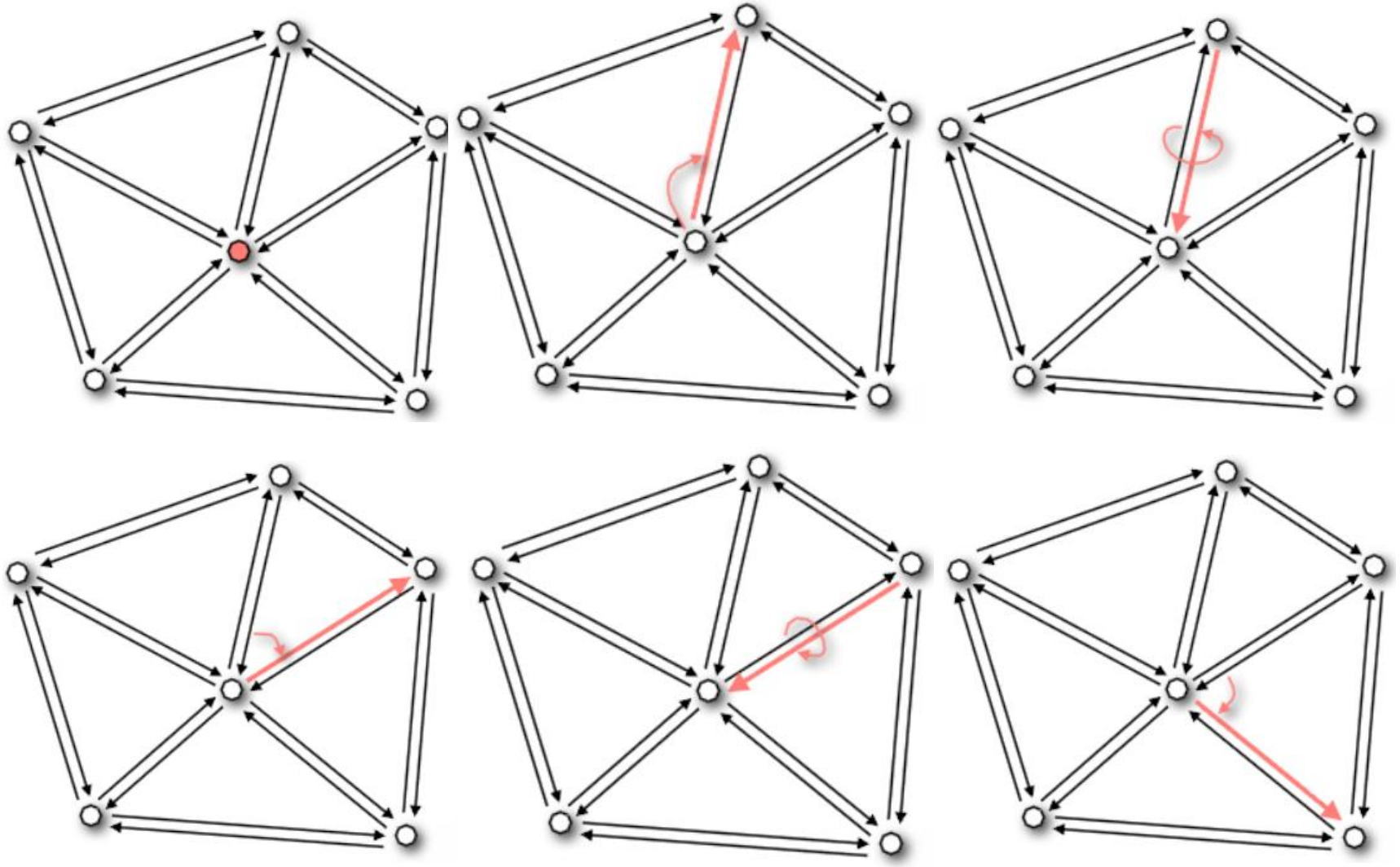
- Example: visit all neighbors of a vertex:

```
Halfedge* h = v->halfedge;
do {
   h = h->twin->next;
}
while( h != v->halfedge );
```



One-Ring Traversal

1. Start at vertex
2. Outgoing halfedge
3. Opposite halfedge
4. Next halfedge
5. Opposite
6. Next
7. …



Halfedge meshes are *always* manifold

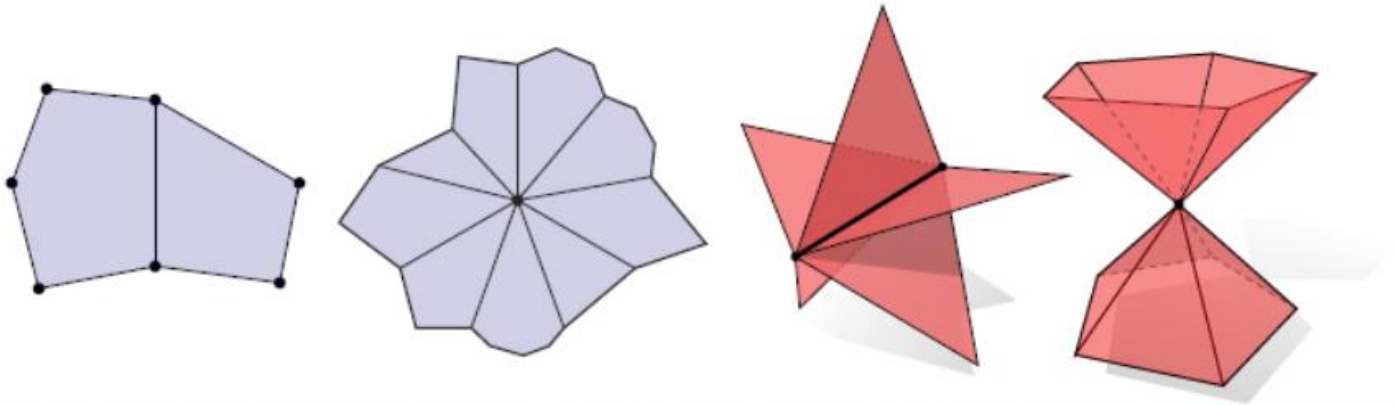
- Consider simplified halfedge data structure
- Require only "common-sense" conditions

(pointer to yourself!)

```
struct Halfedge {
   Halfedge *next, *twin;
};
```

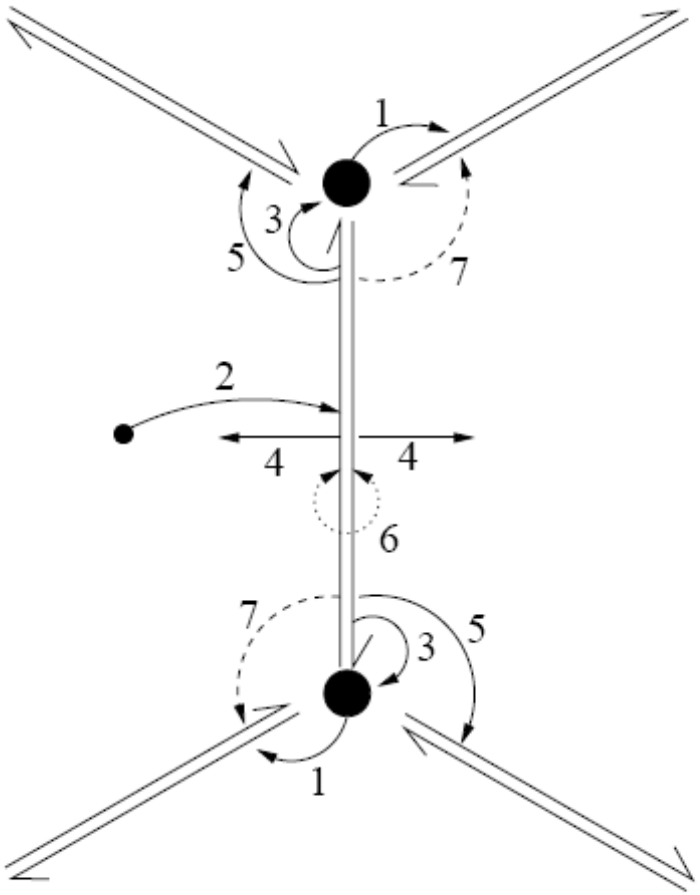
```
twin->twin == this
next != this
twin != this
```

- Keep following `next`, and you'll get faces.
- Keep following `twin` and you'll get edges.
- Keep following `next->twin` and you'll get vertices.



Q: Why, therefore, is it impossible to encode the red figures?

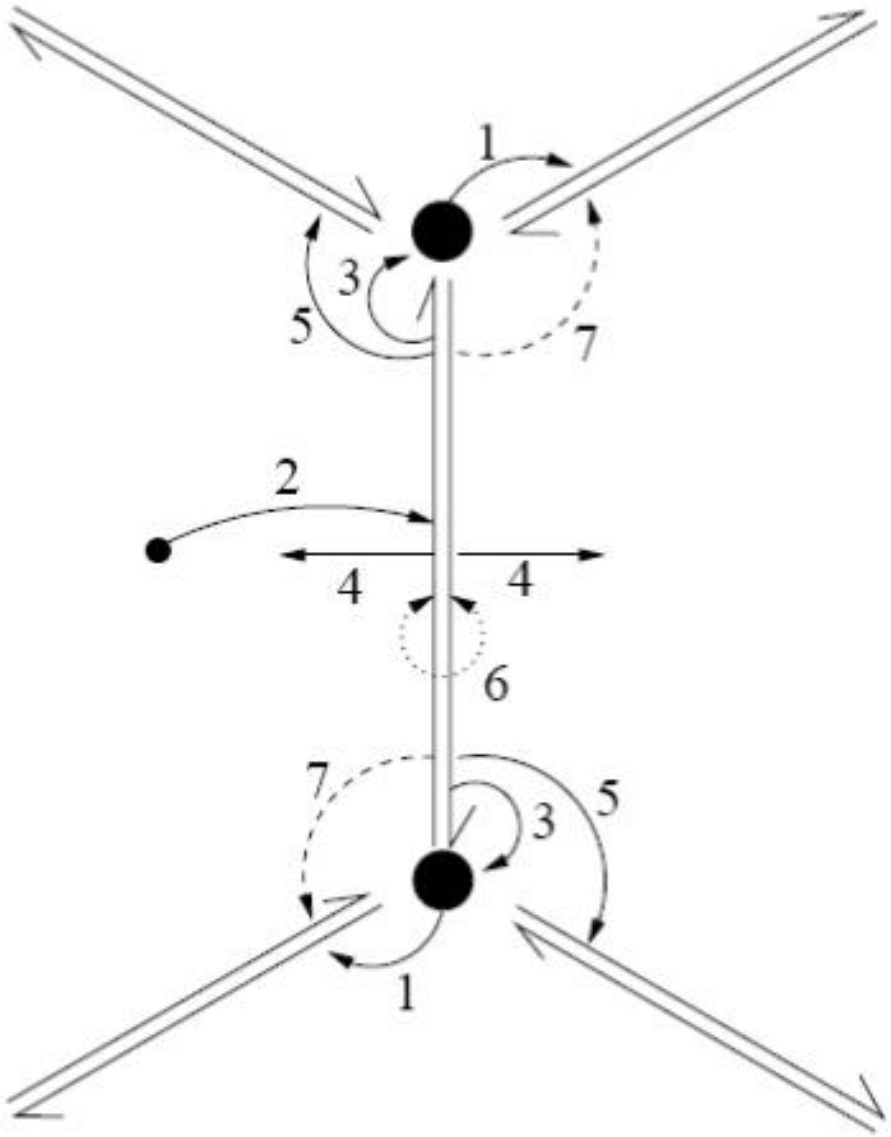
How HDS can -- OpenMesh

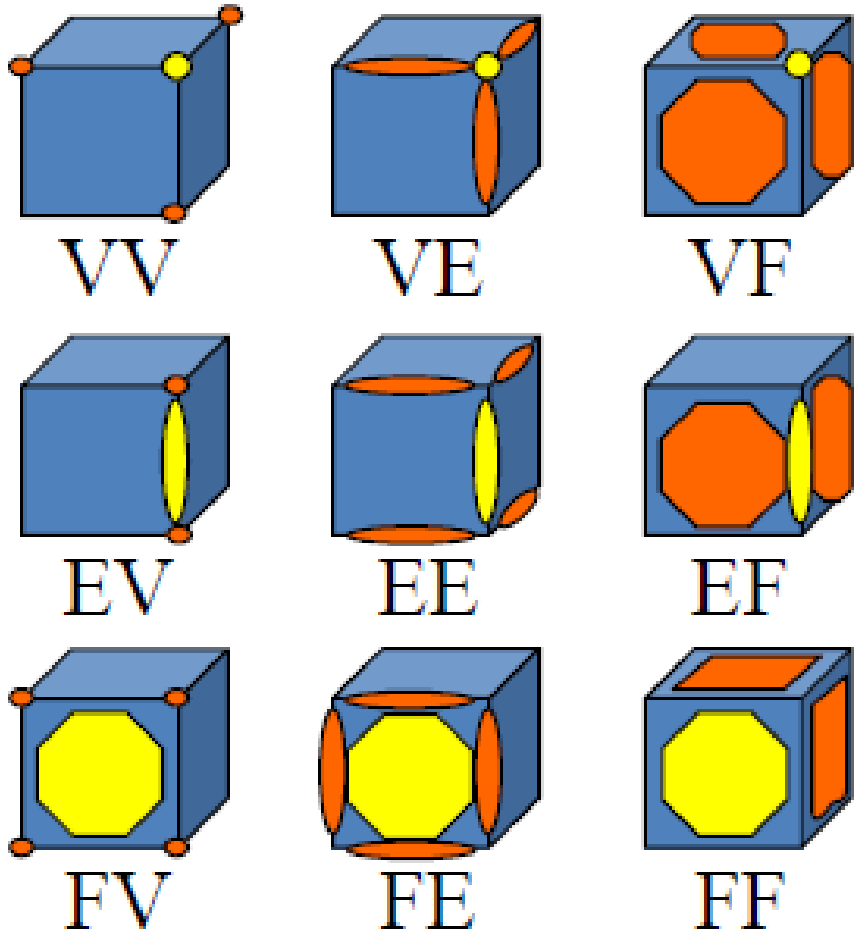


1. Vertex $\mapsto$ one outgoing halfedge,
2. Face $\mapsto$ one halfedge,
3. Halfedge $\mapsto$ target vertex,
4. Halfedge $\mapsto$ its face,
5. Halfedge $\mapsto$ next halfedge,
6. Halfedge $\mapsto$ opposite halfedge (implicit),
7. Halfedge $\mapsto$ previous halfedge (optional).

All basic queries take constant O(1) time!

How HDS can -- OpenMesh





All basic queries take constant O(1) time!

Attributes

- Each object stores attributes (traits) which defines other structures on the mesh:
  - metric structure: edge length
  - angle structure: halfedge
  - curvature : vertex
  - conformal factor: vertex
  - Laplace-Beltrami operator: edge
  - Ricci flow edge weight; edge
  - holomorphic 1-form: halfedge

Comparison of Polygon Mesh Data Structures

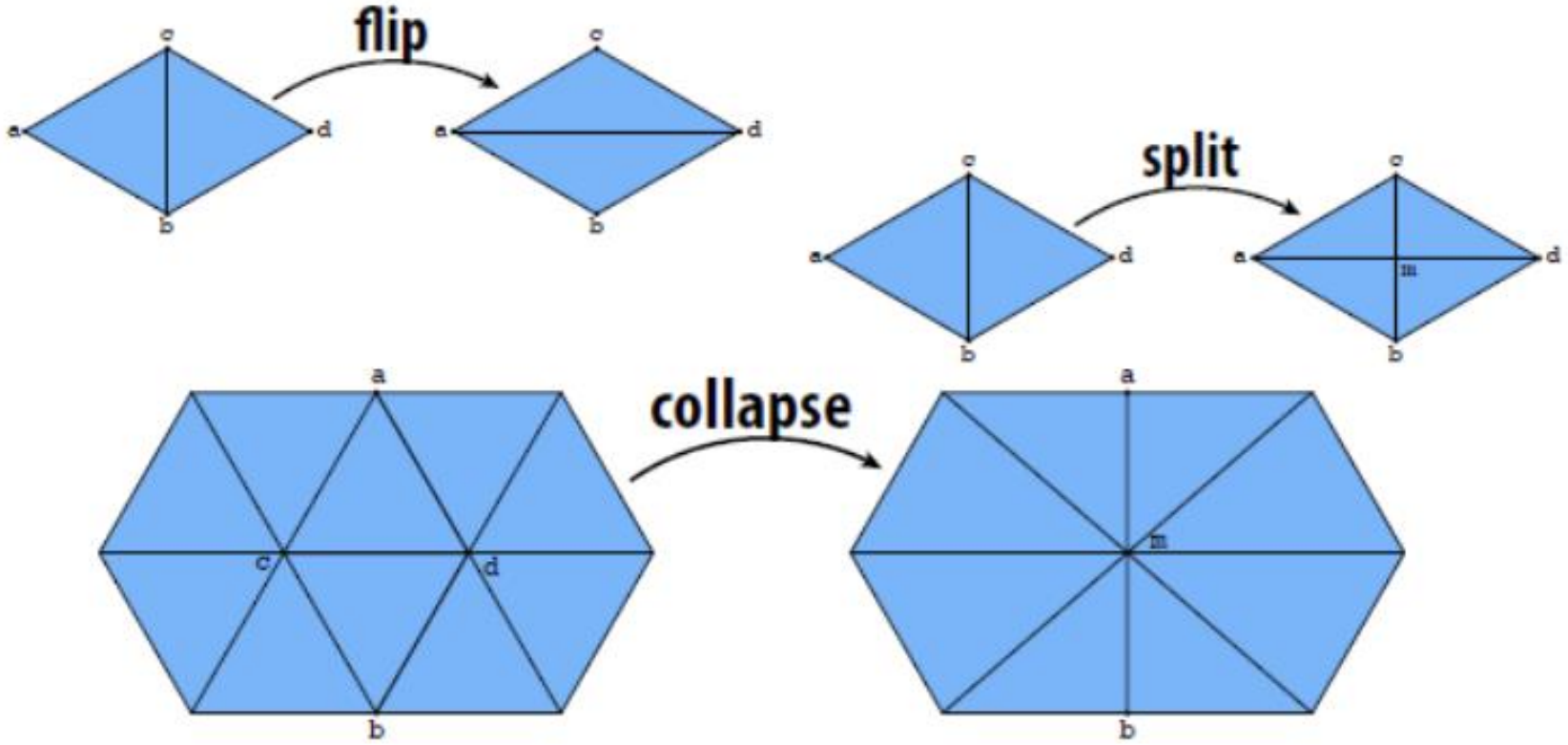
Case study: triangles.	Polygon Soup	Incidence Matrices	Halfedge Mesh
storage cost*	~3 x #vertices	~33 x #vertices	~36 x #vertices
constant-time neighborhood access?	NO	YES	YES
easy to add/remove mesh elements?	NO	NO	YES
nonmanifold geometry?	YES	YES	NO

Conclusion: pick the right data structure for the job!

*number of integer values and/or pointers required to encode *connectivity*
(all data structures require same amount of storage for vertex positions)

Halfedge meshes are easy to edit

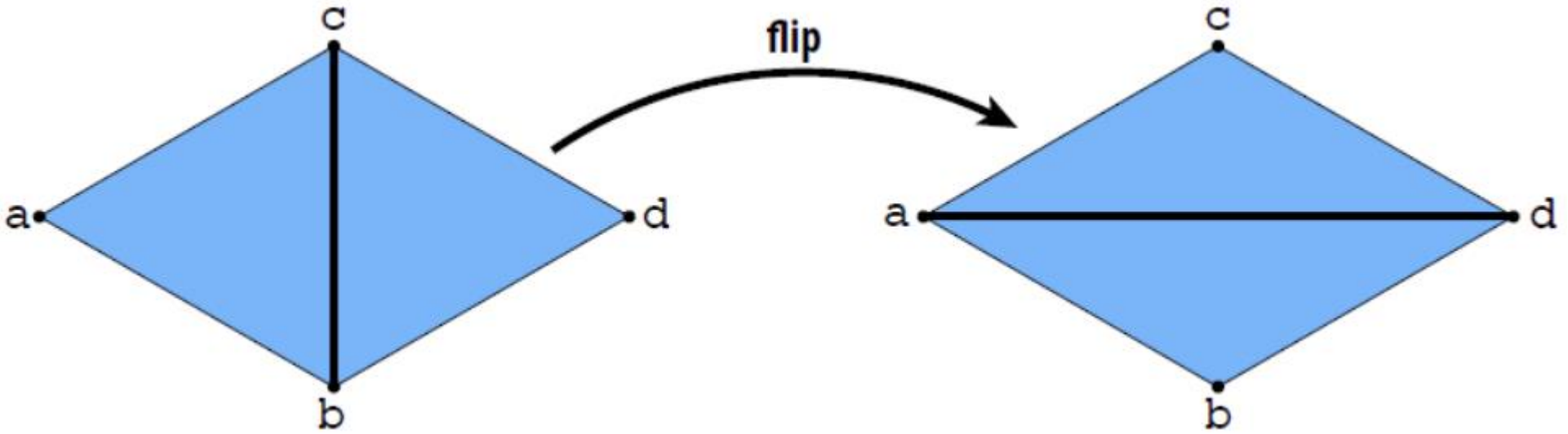
- Remember key feature of linked list: insert/delete elements
- Same story with halfedge mesh (“linked list on steroids”)
- E.g., for triangle meshes, several atomic operations:



- How? Allocate/delete elements; reassigning pointers.
- Must be careful to preserve manifoldness!

Edge Flip (Triangles)

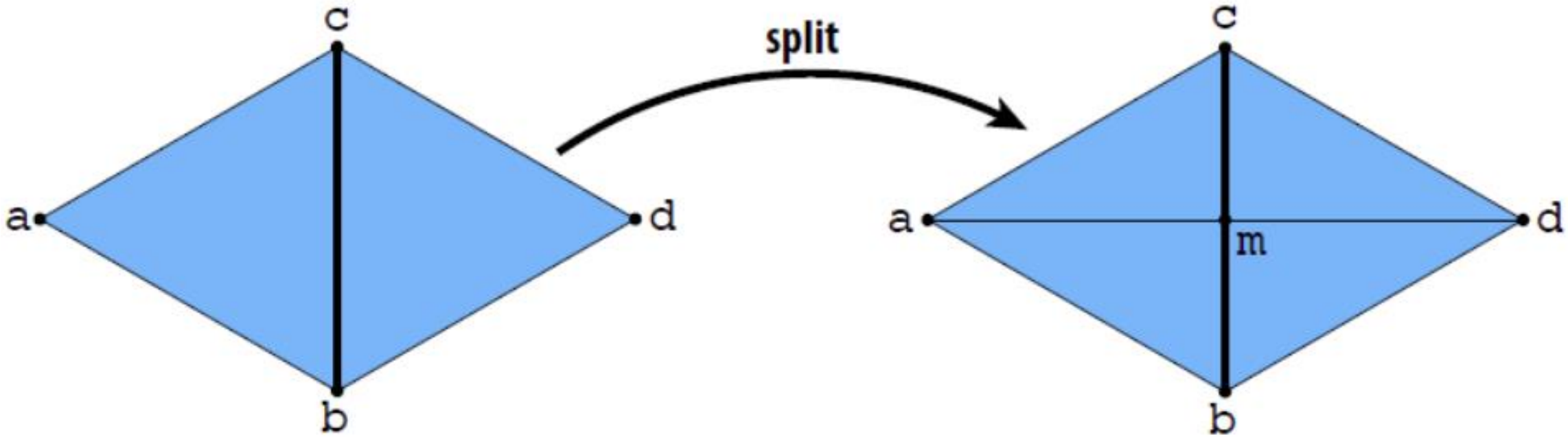
- Triangles (a,b,c), (b,d,c) become (a,d,c), (a,b,d):



- Long list of pointer reassignments (edge->halfedge = ...)
- However, no elements created/destroyed.
- Q: What happens if we flip twice?
- Challenge: can you implement edge flip such that pointers are
- *unchanged* after two flips?

Edge Split (Triangles)

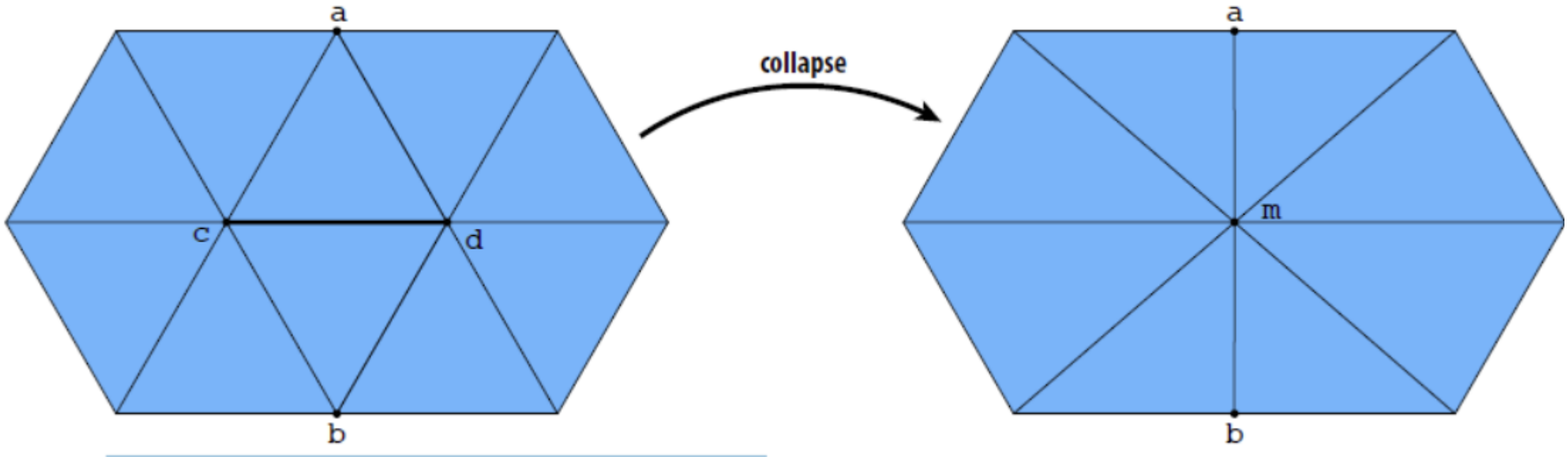
- Insert midpoint m of edge (c,b), connect to get four triangles:



- This time, have to *add* new elements.
- Lots of pointer reassignments.
- Q: Can we “reverse” this operation?

Edge Collapse (Triangles)

- Replace edge (b,c) with a single vertex m:

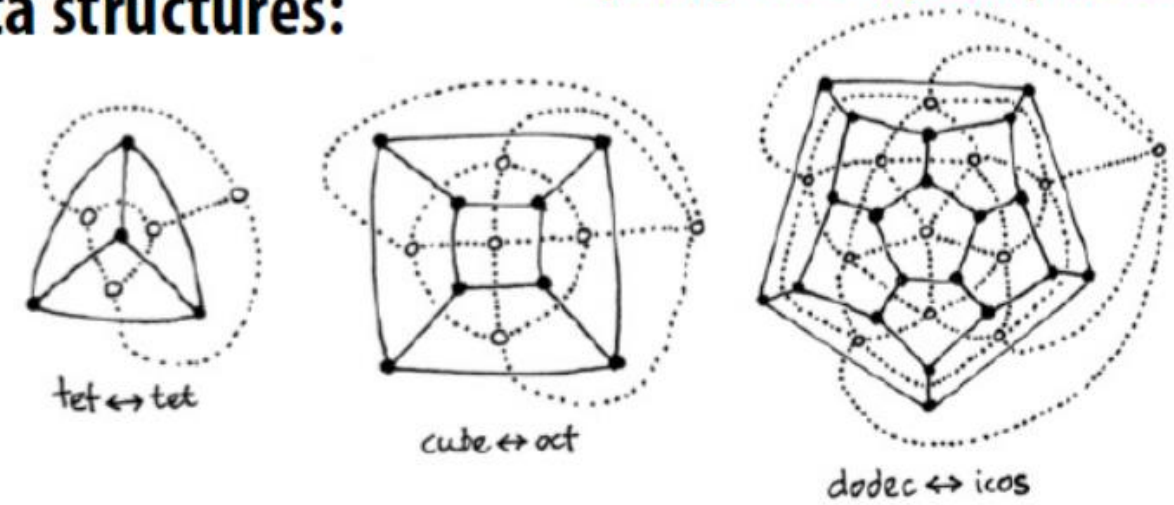


- Now have to *delete* elements.
- Still lots of pointer assignments!
- Q: How would we implement this with a polygon soup?
- Any other good way to do it? (E.g., different data structure?)

Alternatives to Halfedge

Paul Heckbert (former CMU prof.)
quadedge code - http://bit.ly/1QZLHos

- Many very similar data structures:
  - winged edge
  - corner table
  - quadedge
  - ...



- Each stores local neighborhood information
- Similar tradeoffs relative to simple polygon list:
  - CONS: additional storage, incoherent memory access
  - PROS: better access time for individual elements, intuitive traversal of local neighborhoods
- (Food for thought: can you design a halfedge-like data structure with reasonably coherent data storage?)

Half-edge data structure

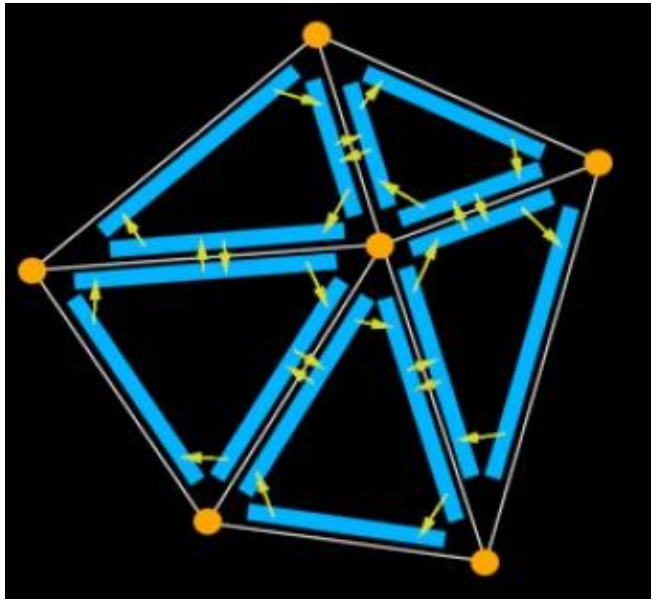
(What?)A common way to represent triangular (polyhedral) mesh. 3D analogy: half-face data structure for tetrahedral mesh

(Why?) Effective for maintaining incidence info of vertices:

- Efficient local traversal
- Low spatial cost
- Supporting dynamic local updates/manipulations (edge collapse, vertex split, etc.)

(Who?)

- **CGAL, OpenMesh (OpenSG), MCGL (for matlab)**
- A free library from Xin li.
- A free surface library from Xianfeng Gu.
- Denis Zorin uses it in implementing Subdivision.



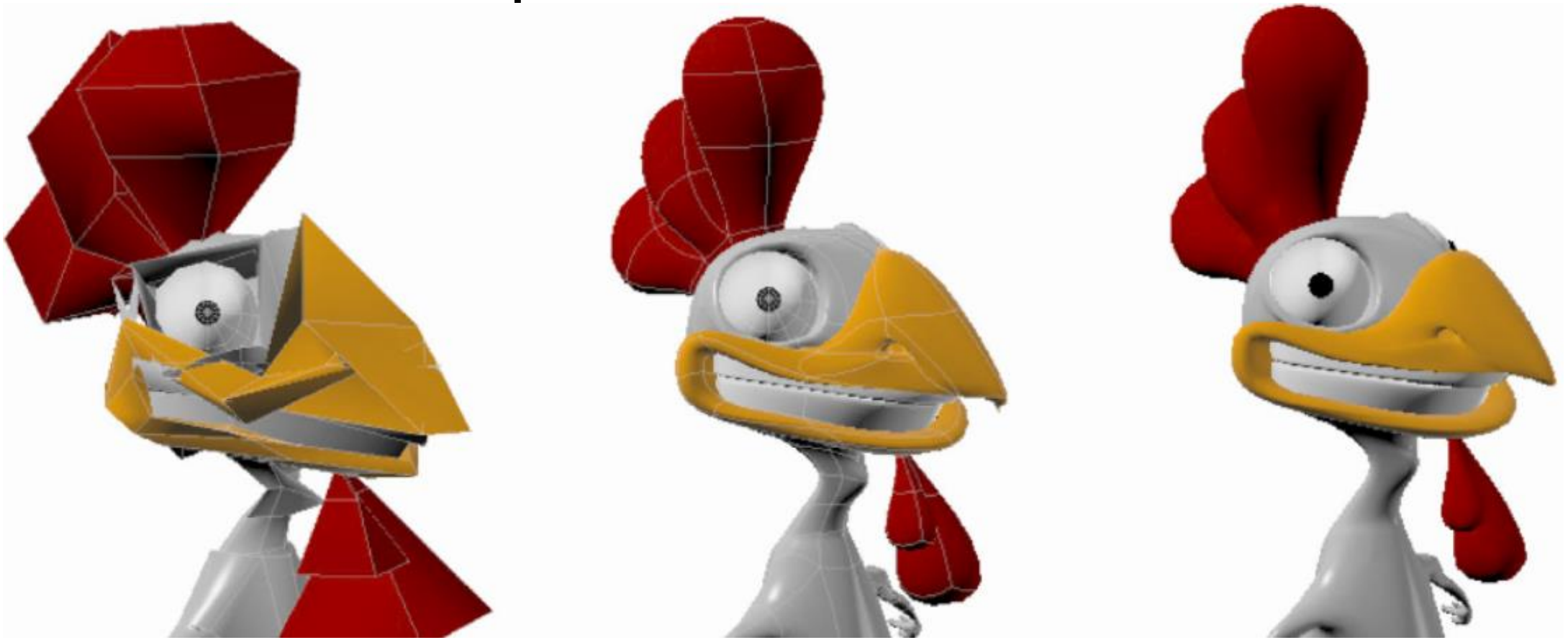
Why *Open*mesh?

- **Flexible / Lightweight**
  - Random access to vertices/edges/faces
  - Arbitrary scalar types
  - Arrays or lists as underlying kernels
- **Efficient in space and time**
  - Dynamic memory management for array-based meshes (not in CGAL)
  - Extendable to specialized kernels for non-manifold meshes (not in CGAL)
- **Easy to Use**

Ok, but what can we actually *do* with our fancy new data structure?

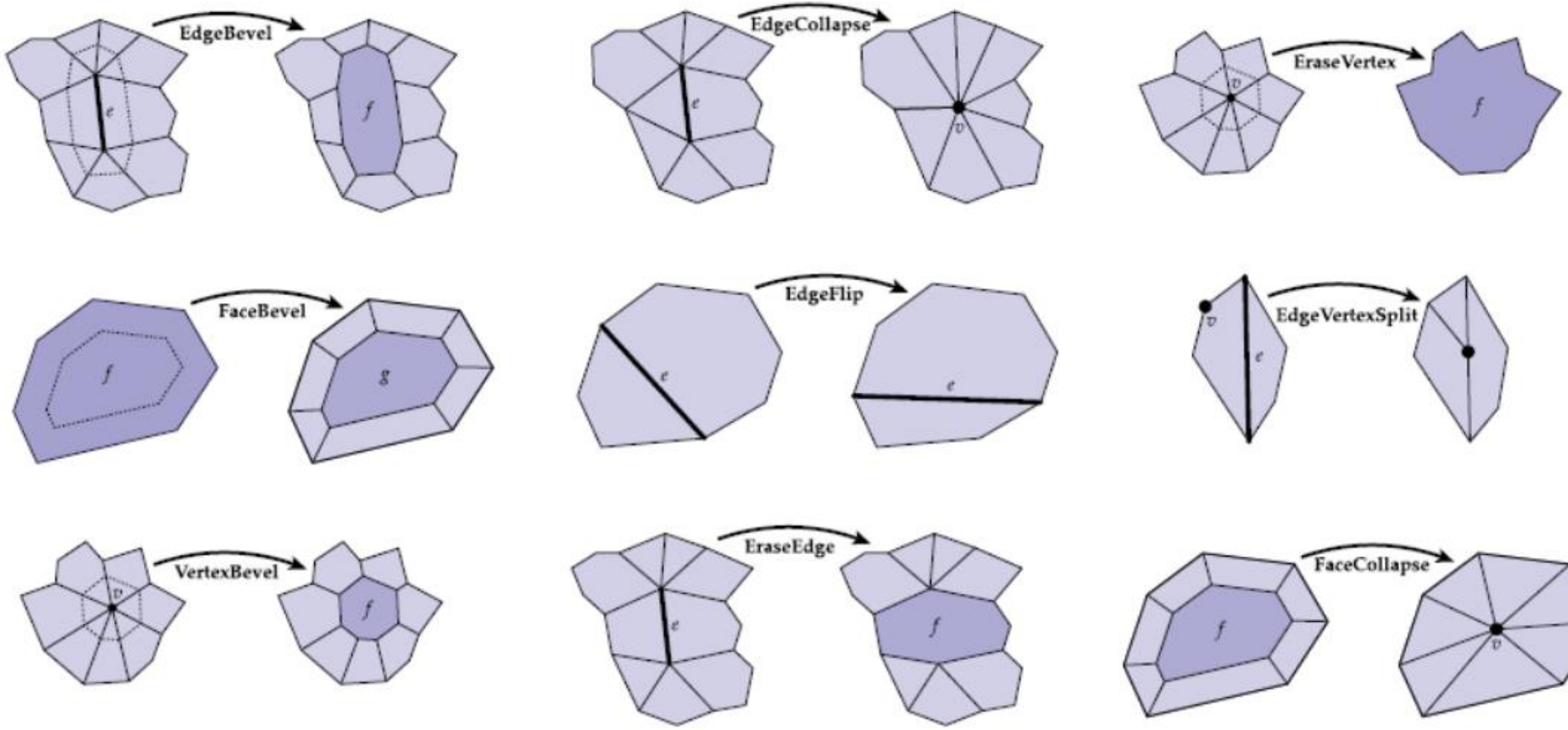
Subdivision Modeling

- Common modeling paradigm in modern 3D tools:
  - Coarse “control cage”
  - Perform local operations to control/edit shape
  - Global subdivision process determines final surface



Subdivision Modeling—Local Operations

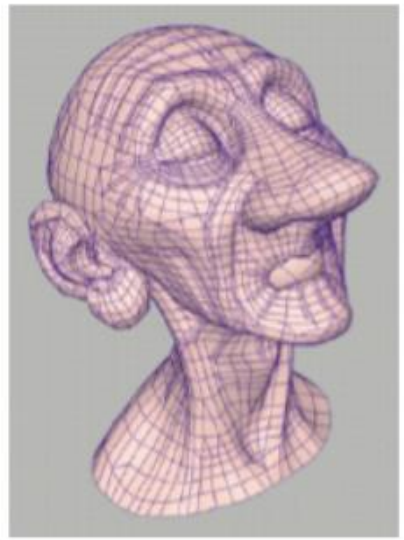
- **For general polygon meshes, we can dream up lots of local mesh operations that might be useful for modeling:**

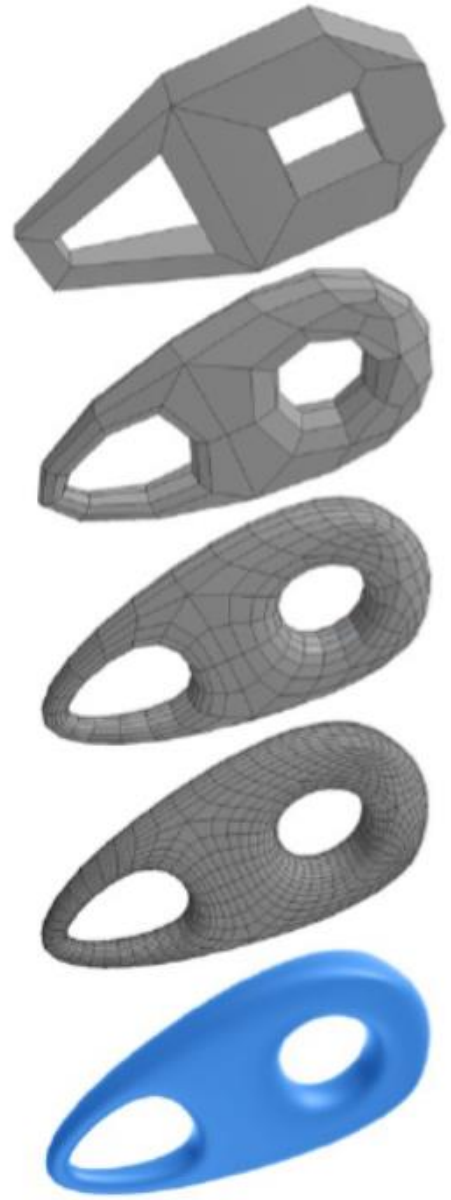


…and many, many more!

Global Subdivision

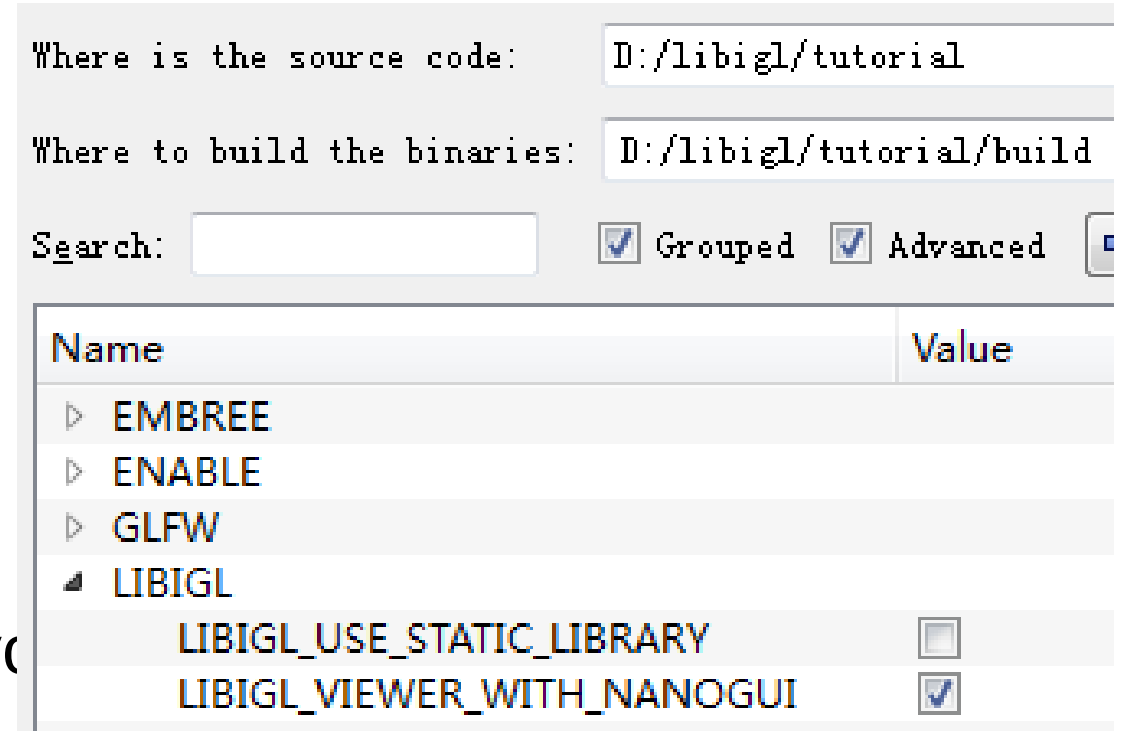
- Start with coarse polygon mesh ("control cage")
- Subdivide each element
- Update vertices via local averaging
- Many possible rule:
  - Catmull-Clark (quads)
  - Loop (triangles)
  - ...
- Common issues:
  - interpolating or approximating?
  - continuity at vertices?
- Easier than splines for modeling; harder to evaluate pointwise





Environment – c++

- Visual studio 2015 community
- CMAKE
- Python 3
- CGAL
  - Boost
  - Qt
  - libQGLViewer (cool example for picking)
- Eigen
- Libigl (use cmakegui to generate vc Win64,)
  - CoMISo
  - Nanogui (build it first, then cmake libigl)
  - Embree (for picking) (copy bin and lib to D:\libigl\external\embree, then cmake again)



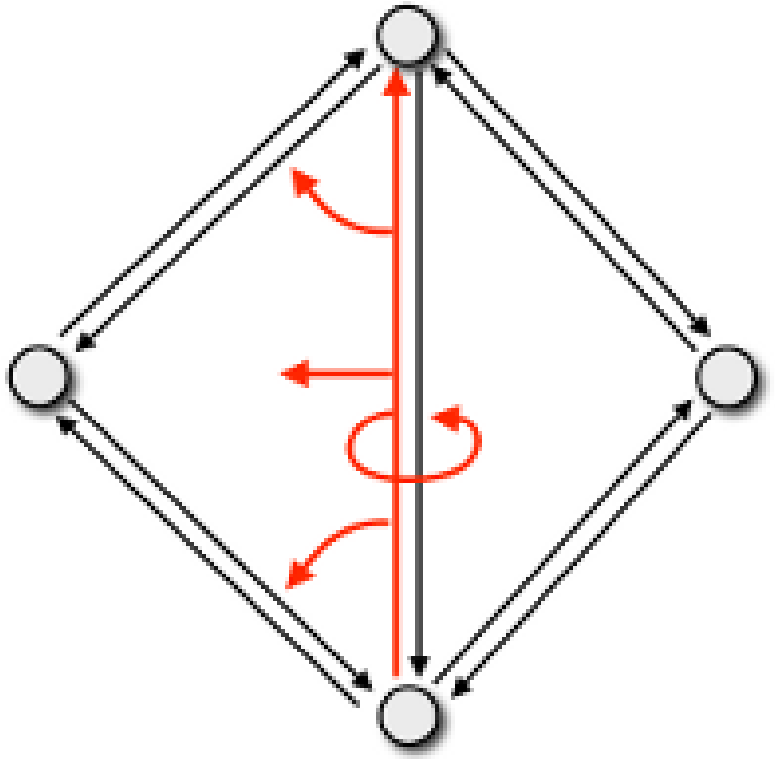
Environment - Matlab

- Matlab 2015b
- jjcao_code: https://github.com/jjcao/jjcao_code.git

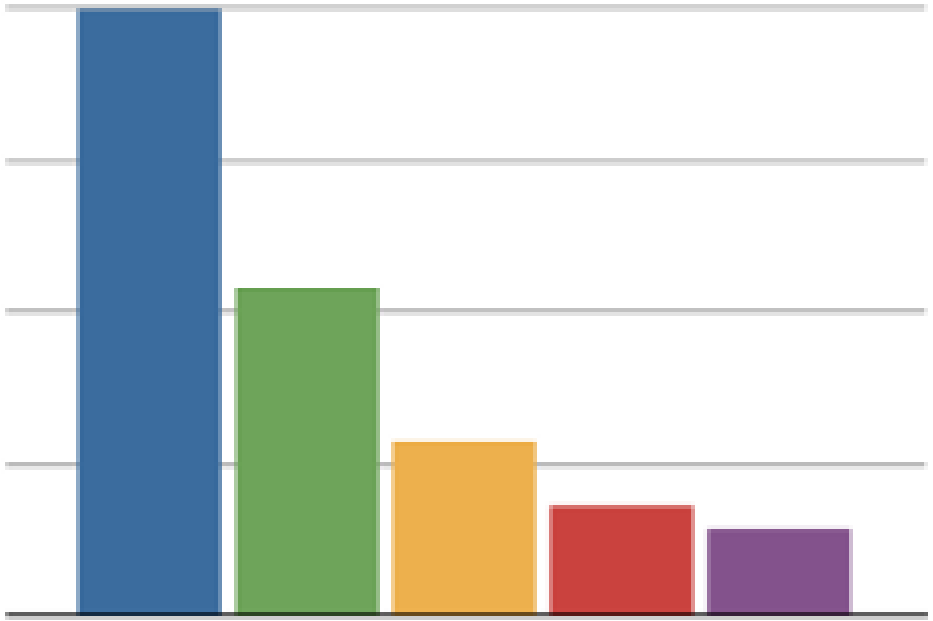
Lab

- Lab1
  - **Chapter 1 of libigl tutorial** or jjcao_code\toolbox\jjcao_plot\eg_trisurf.m
- Lab2 [optional]
  - See User manual of Halfedge Data Structures of CGAL
  - run the examples or jjcao_code\toolbox\jjcao_mesh\datastructure\test_to_halfedge.m

Design, Implementation, and Evaluation of theSurface_mesh Data Structure



```
class Vertex_iterator
{
public:

    /// Default constructor
    Vertex_iterator(Vertex v=Vertex()) : hnd_(v) {}

    /// Cast to the vertex the iterator refers to
    operator Vertex() const { return hnd_; }

    /// are two iterators equal?
    bool operator==(const Vertex_iterator& rhs) const
    {
        return hnd_==rhs.hnd_;
    }

    /// are two iterators different?
    bool operator!=(const Vertex_iterator& rhs) const
    {
        return !(operator==(rhs));
    }

    /// pre-increment iterator
    Vertex_iterator& operator++()
    {
        ++hnd_.idx_;
        return *this;
    }

    /// pre-decrement iterator
    Vertex_iterator& operator--()
    {
        --hnd_.idx_;
        return *this;
    }

private:
    Vertex hnd_;
};
```



[Design, Implementation and Evaluation of the Surface_mesh Data Structure](), *IMR 2011*. has code

Resources

- *https://github.com/jjcao/jjcao_code.git*
- SourceTree

- Gabriel Peyre's numerical tour!

- Wiki
- OFF file format specification

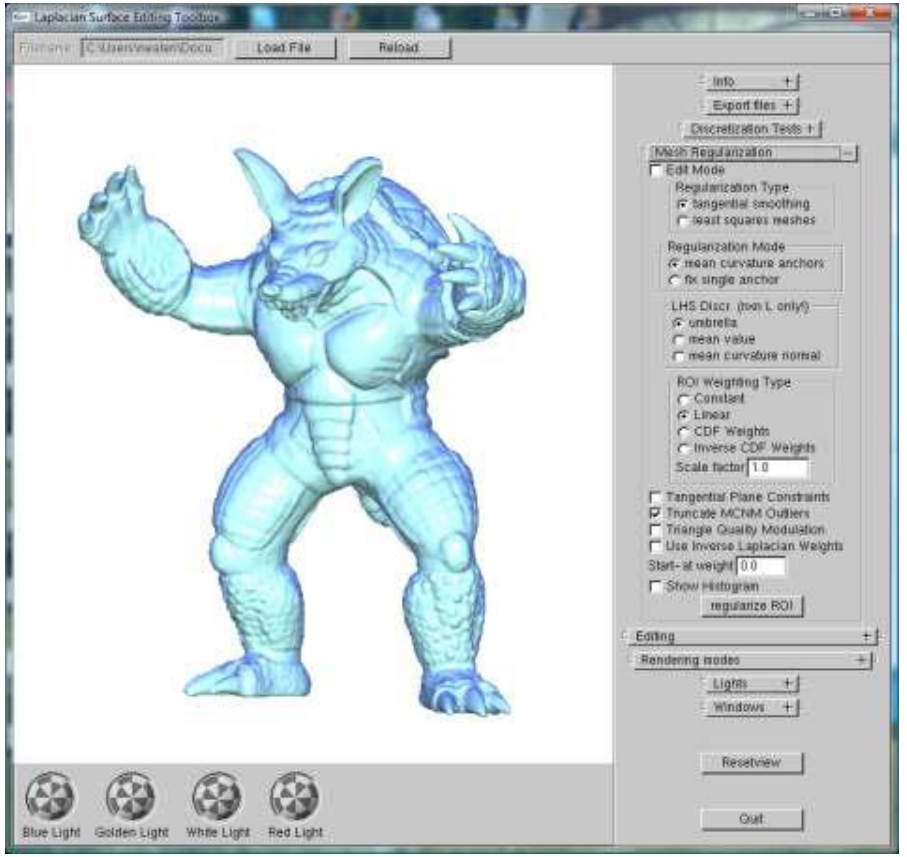
- Xianfeng Gu, lecture_8_halfedge_data_structure

Thanks!

Old assignment

Assignment 1: Mesh processing “Hello World”

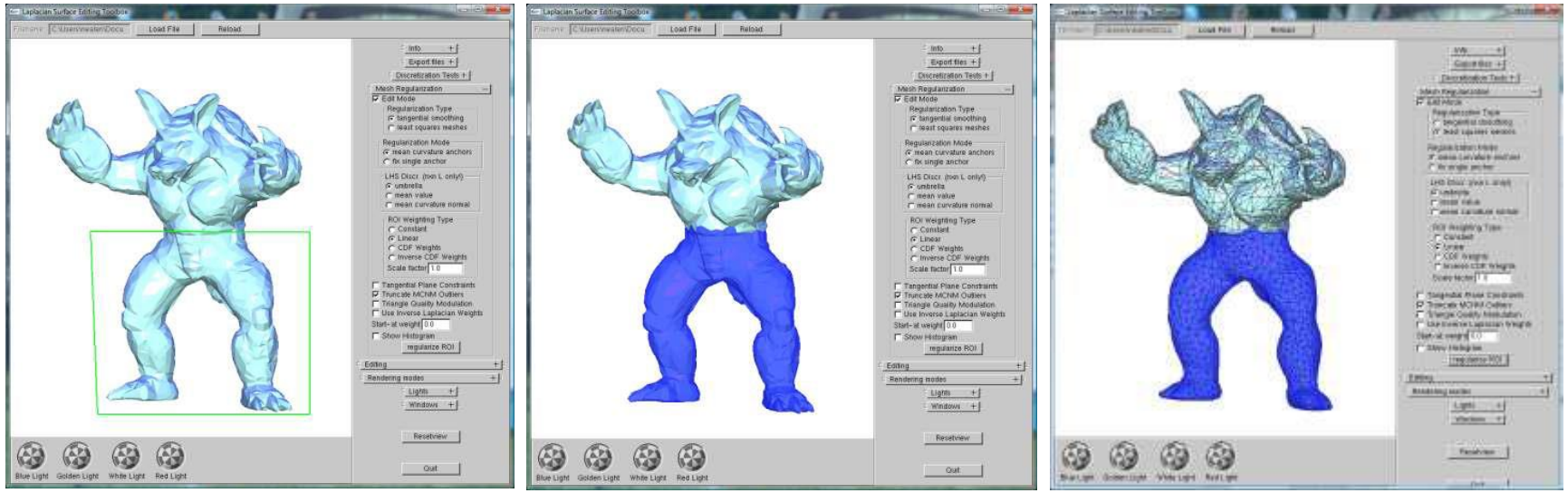
- Goals: learn basic mesh data structure programming + rendering (flat/gouraud shaded, wireframe) + basic GUI programming
- by **MATLAB** or **VC**



You can ask the help from school senior!

Assignment 2: selection + operation tools

- Goals: implement image-space selection tools and perform local operations (smoothing, etc.) on selected region
- VC



Final Project

- Implementation/extension of a space or surface based editing tool
  - makes use of assignments 1 + 2
  - Your own suggestion, with instructor approval
- Includes written project report & presentation
  - Latex style files will be provided?
  - Power Point examples will be provided?