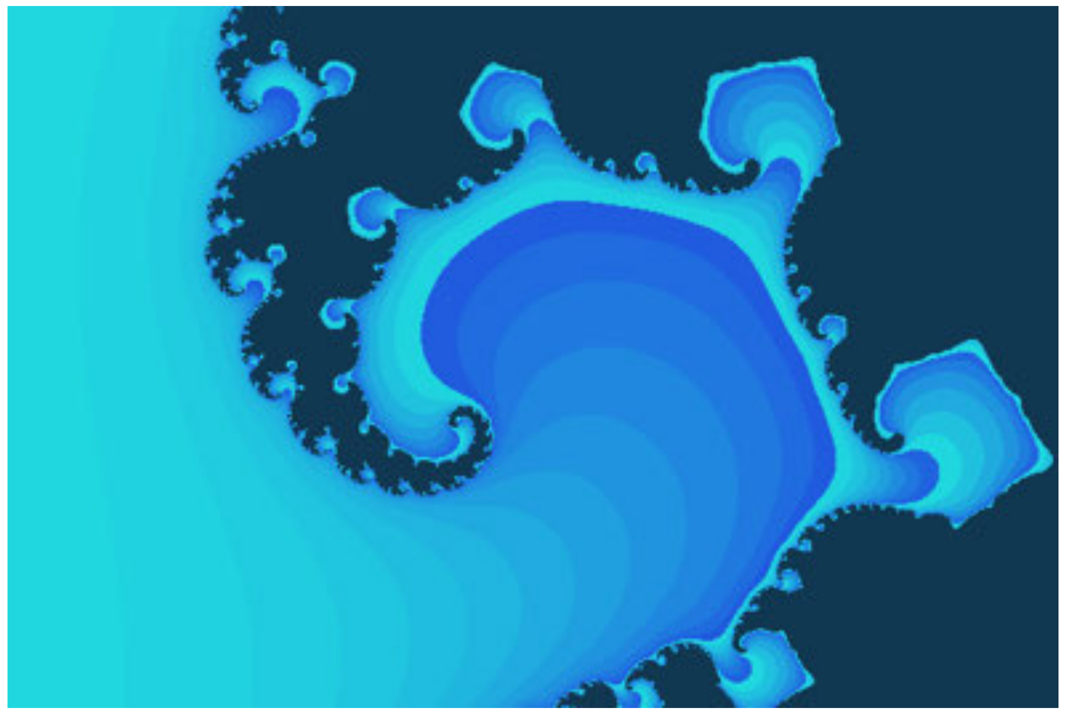
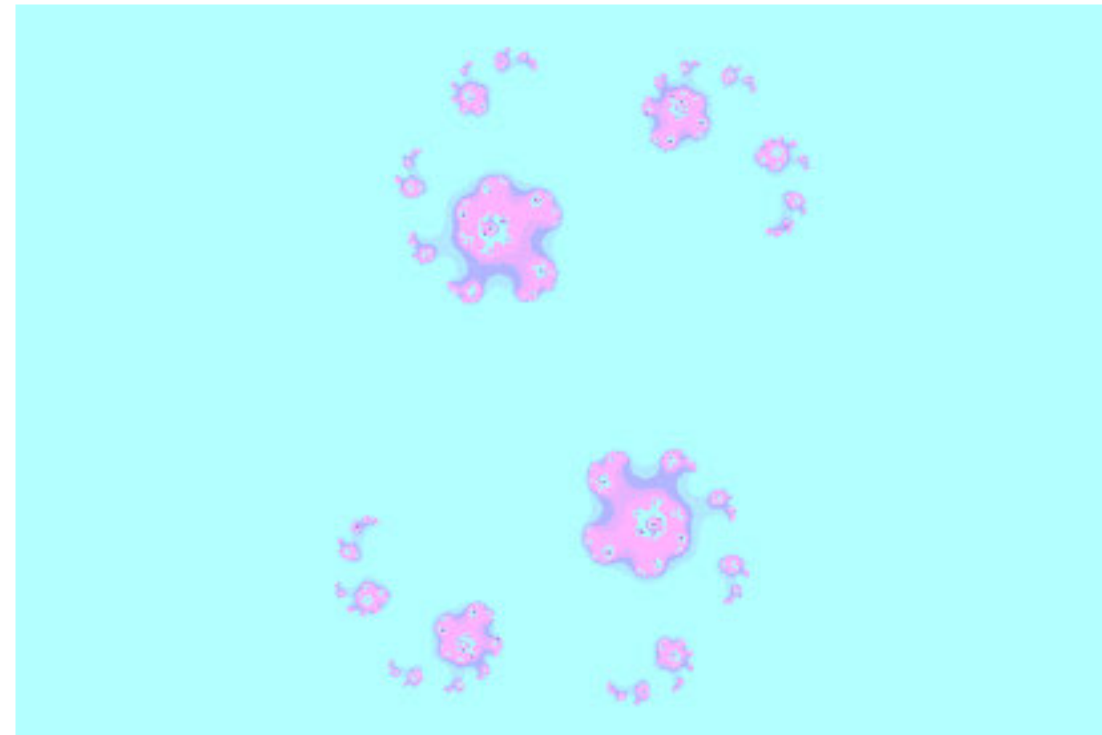
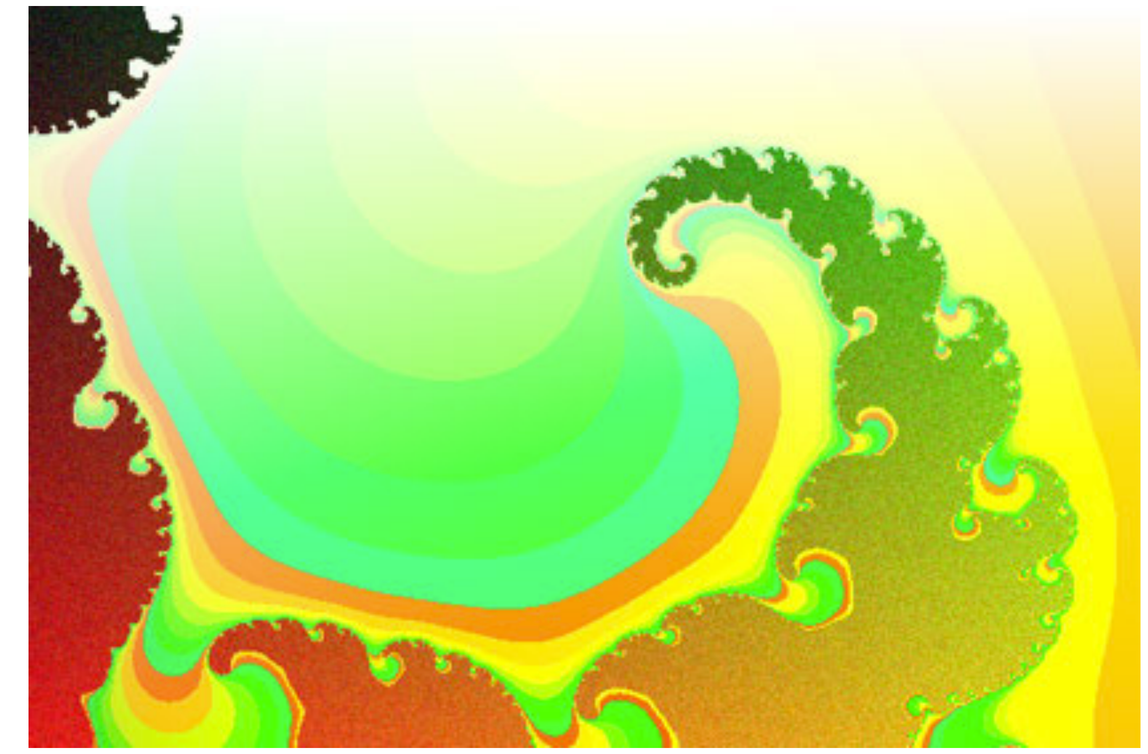
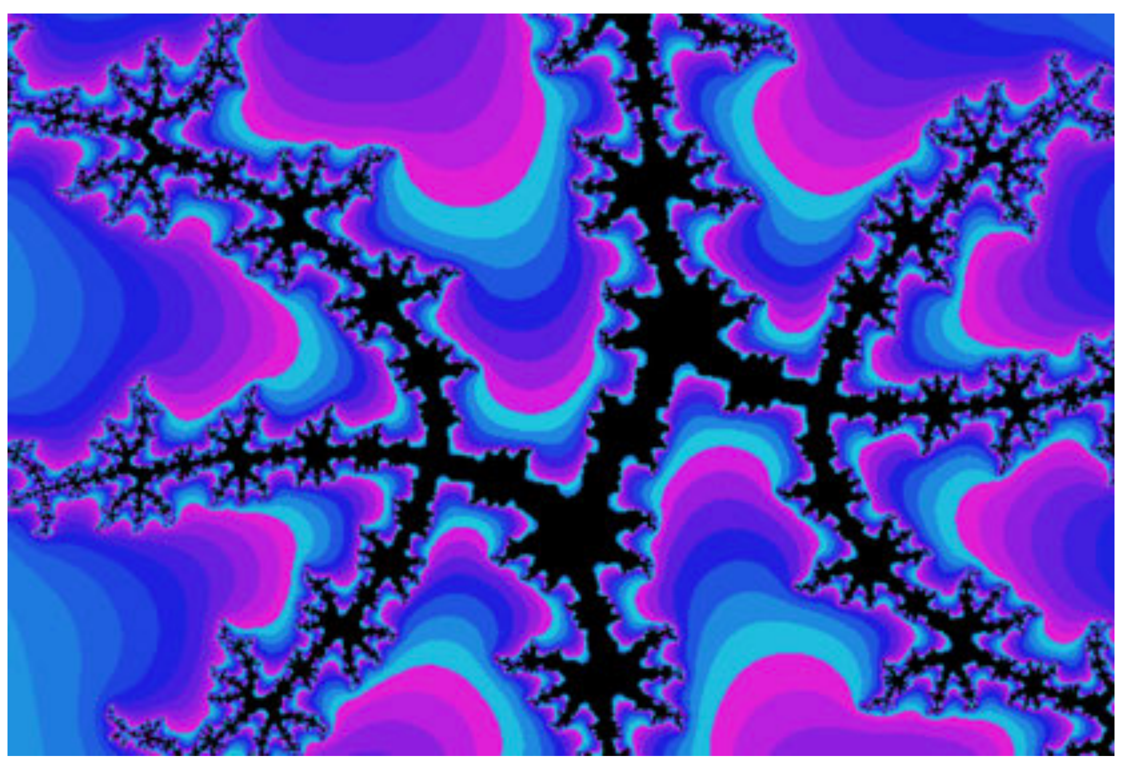
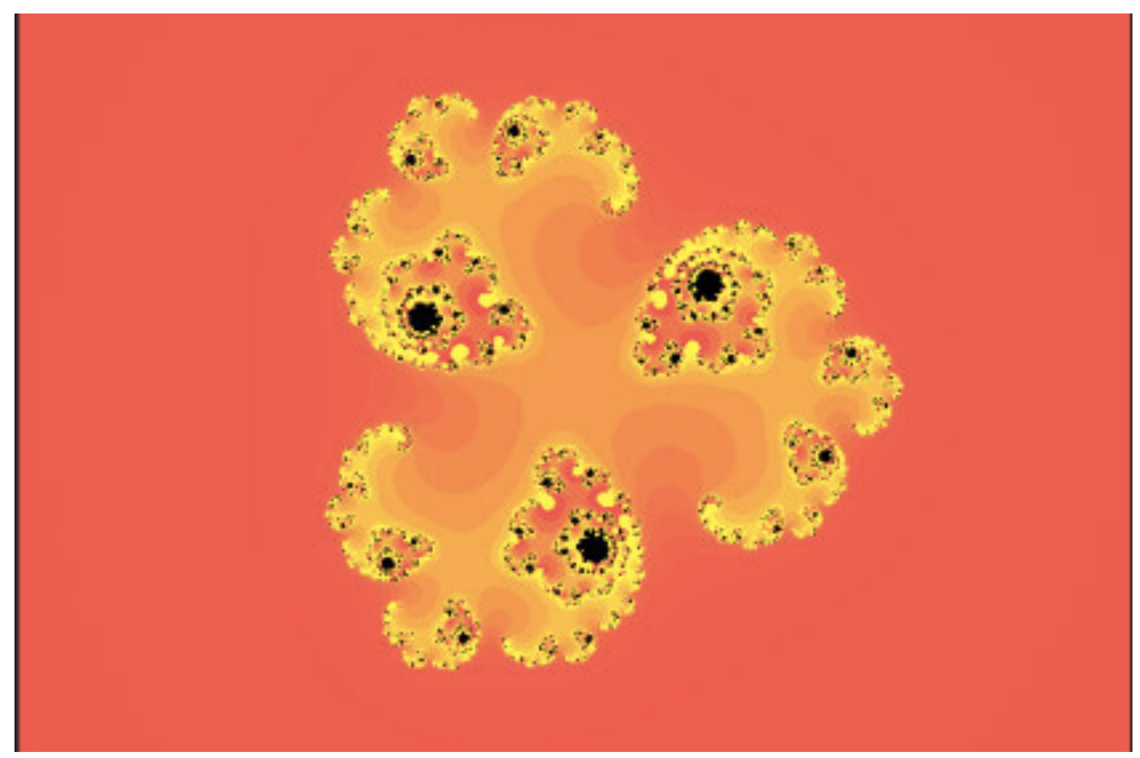
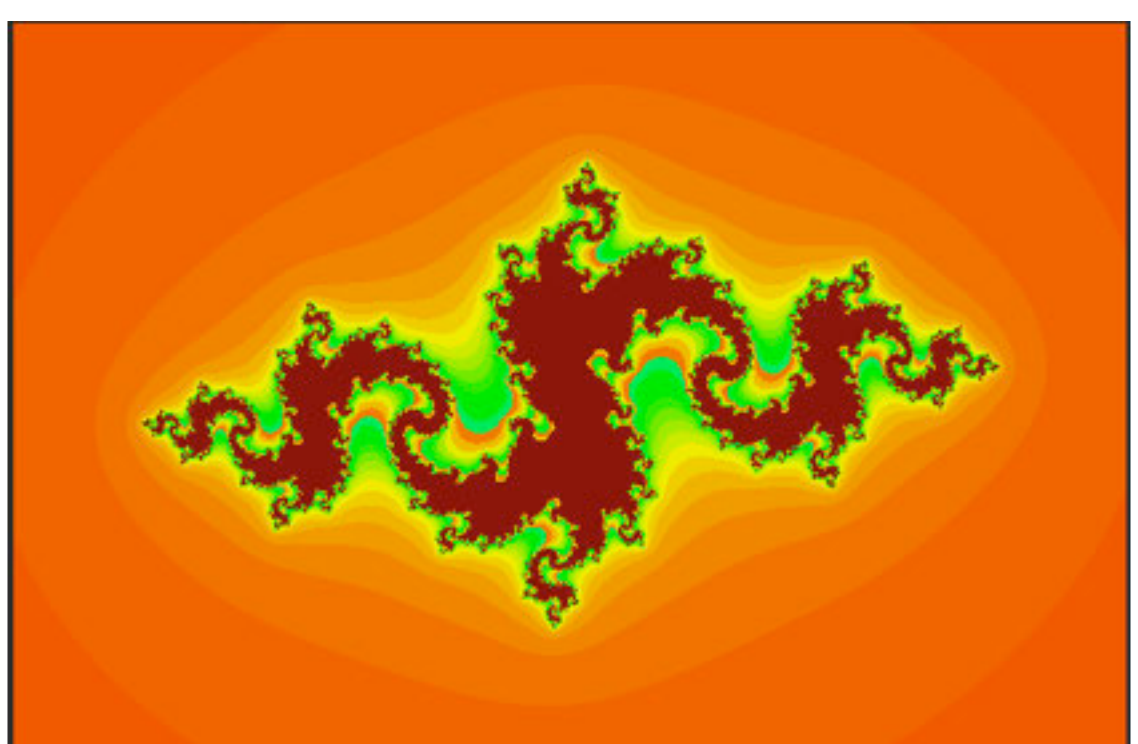
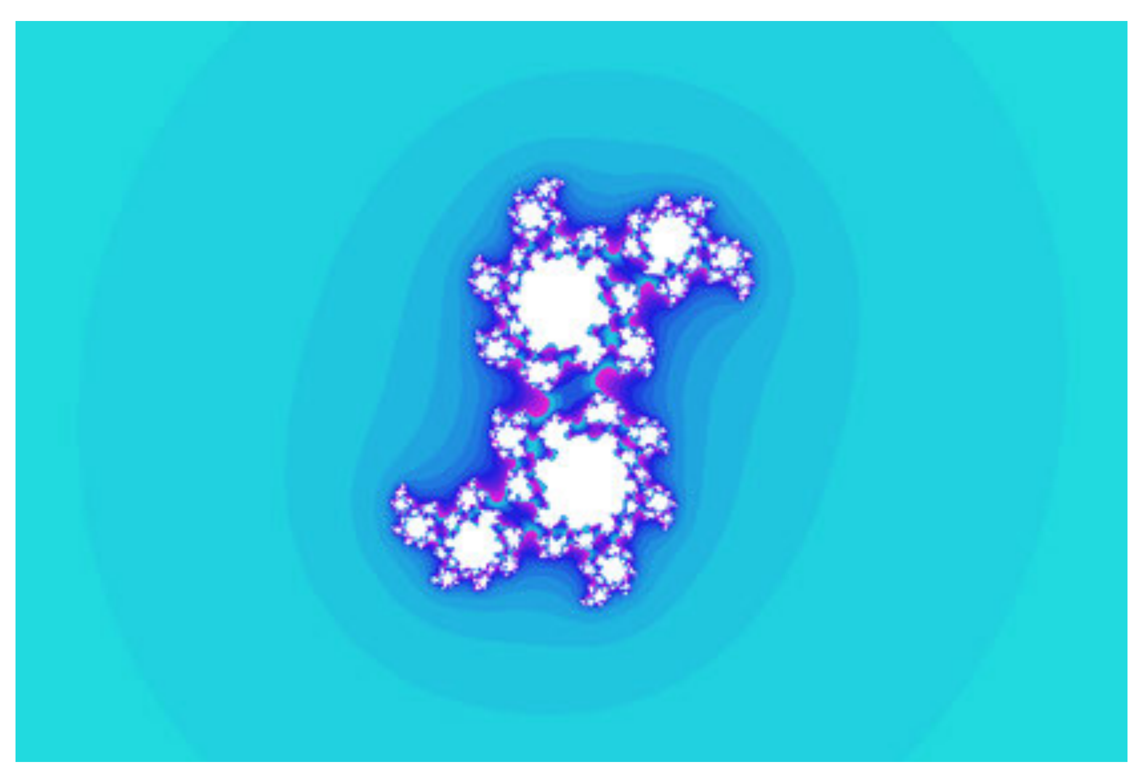
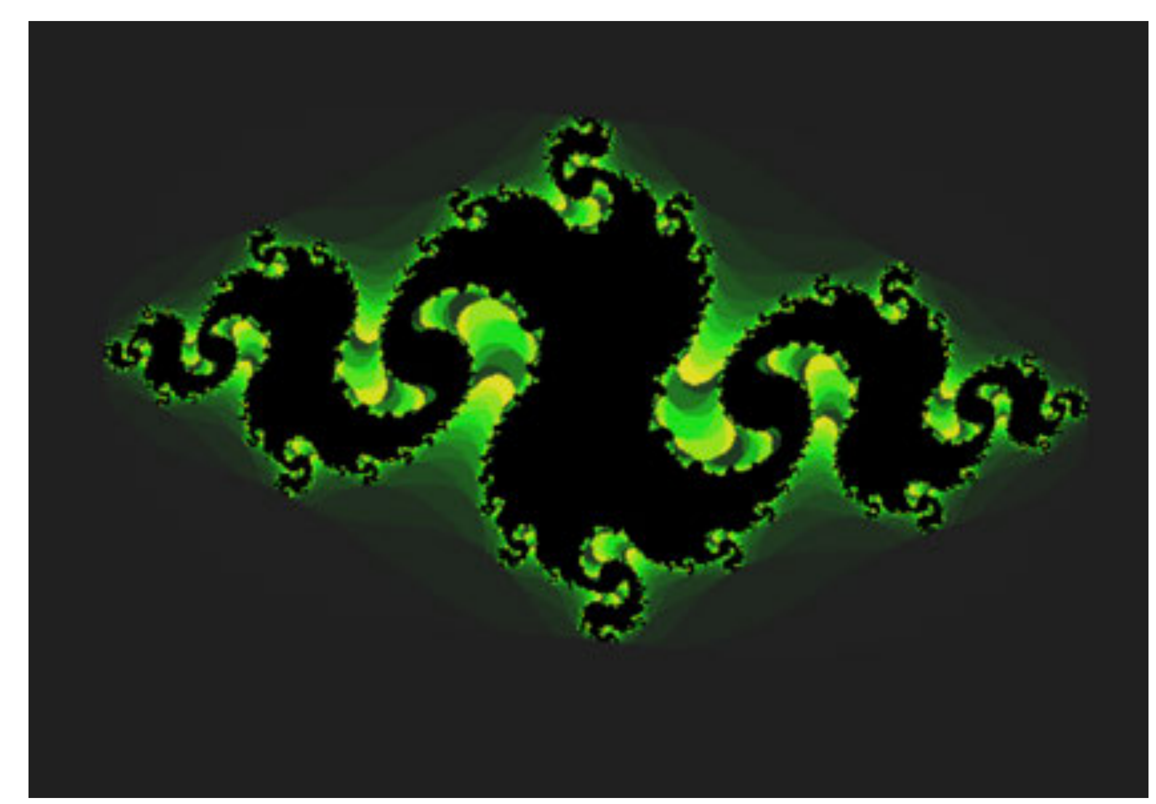
Meshes and Manifolds

Computer Graphics
CMU 15-462/15-662

Fractal Quiz

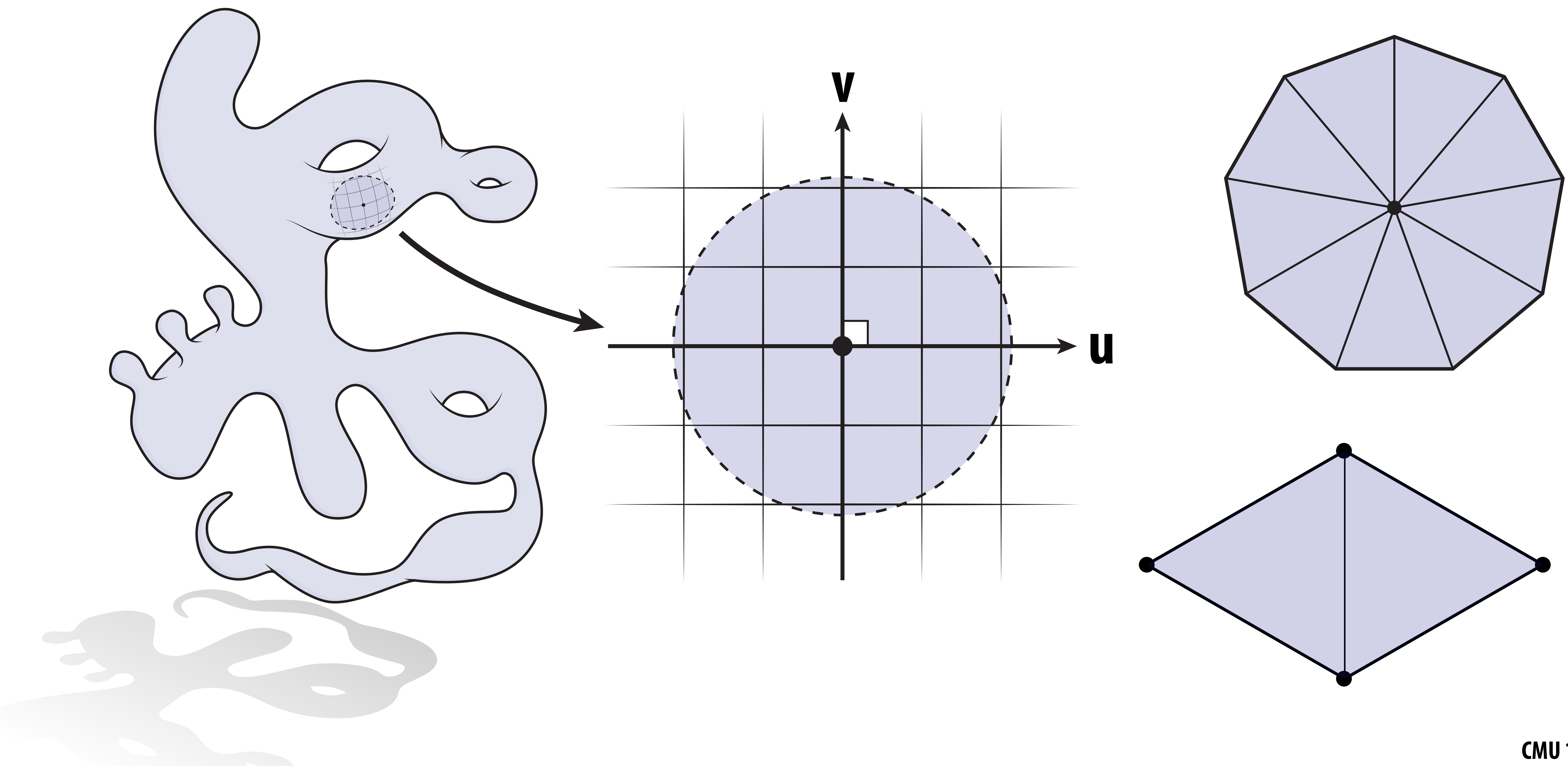
Last time: overview of geometry

- 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
- Today:
  - what is a surface, anyway?
  - nuts & bolts of polygon meshes
  - geometry processing / resampling

Geometry

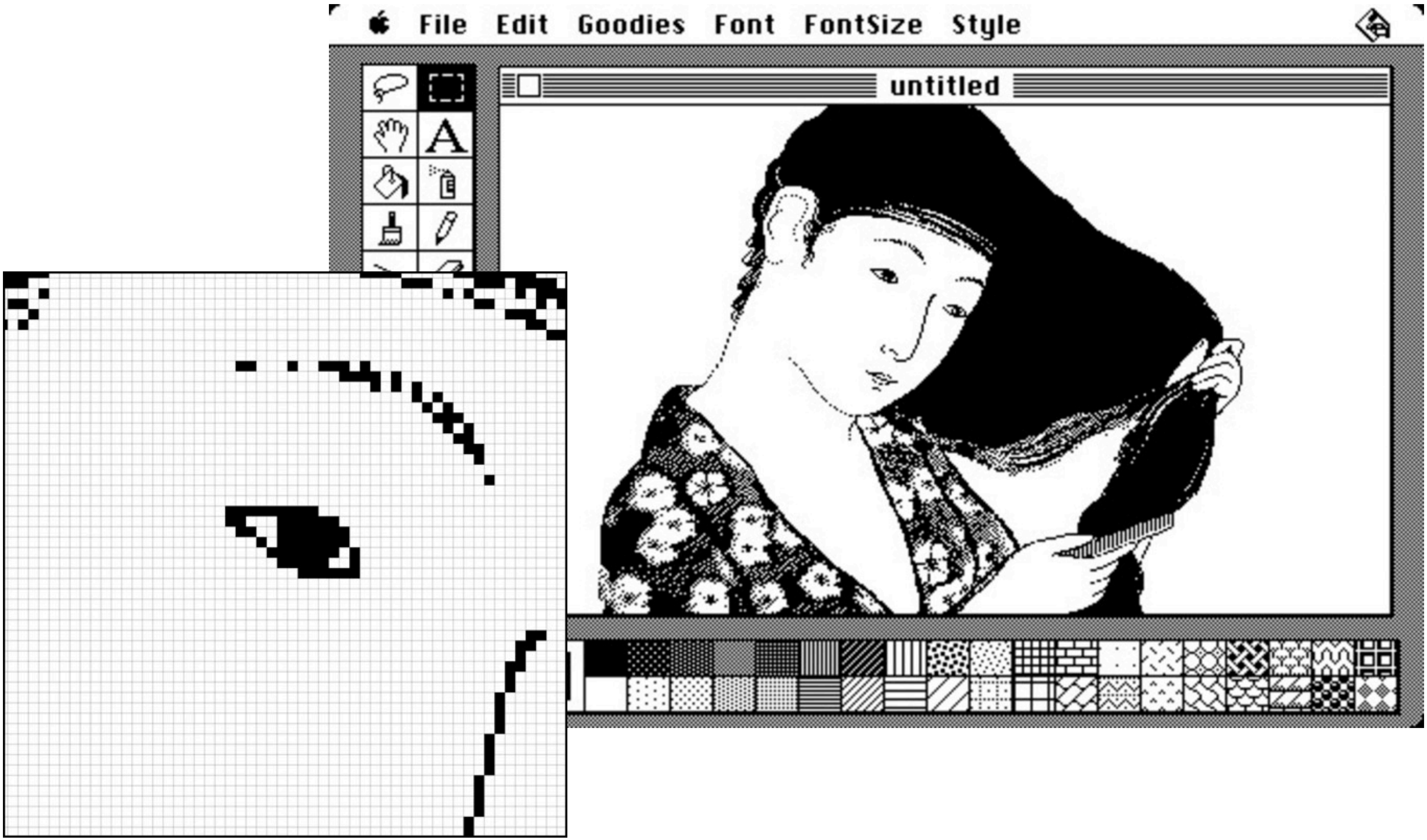
Manifold Assumption

- 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

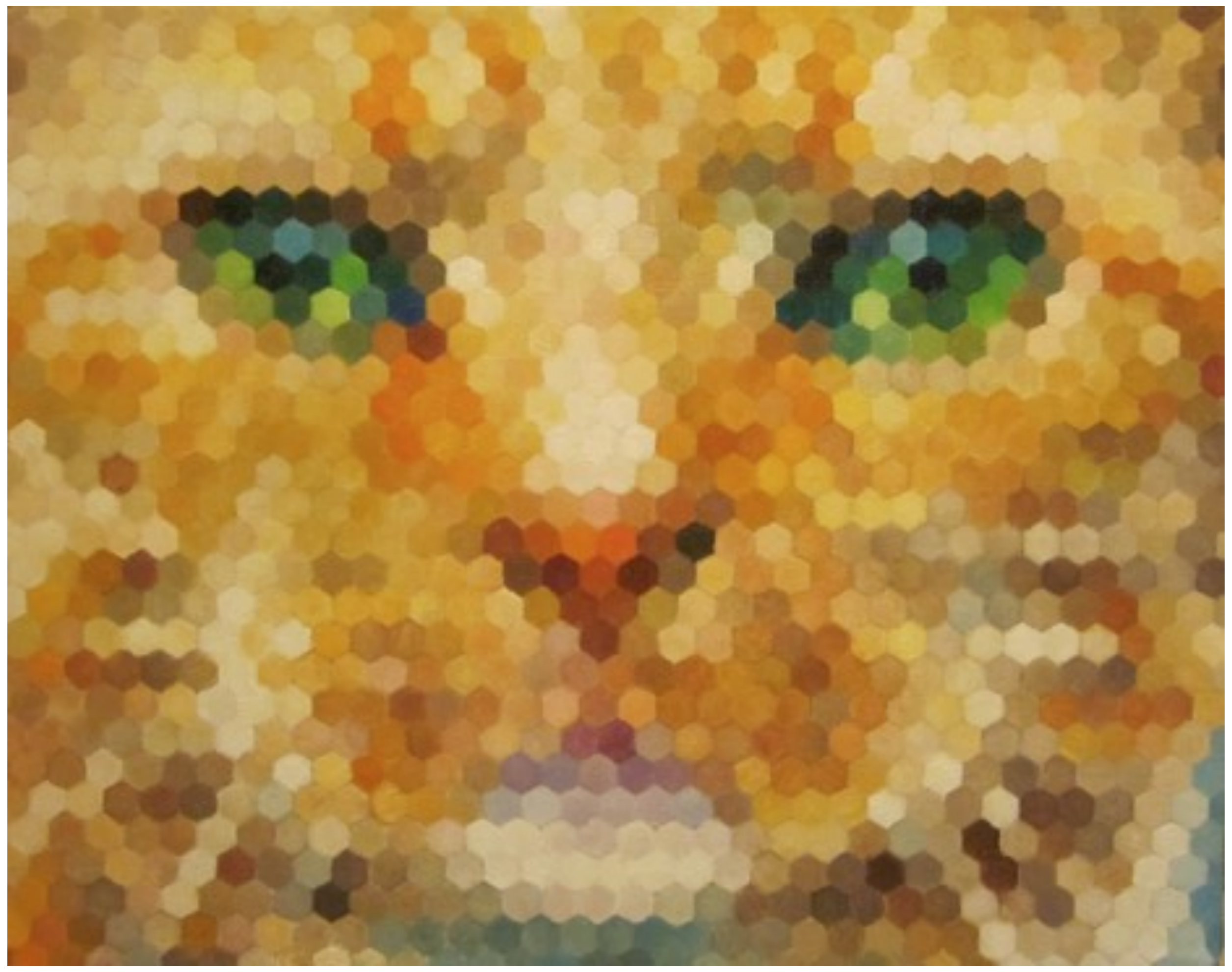
To encode images, we used a *regular grid* of pixels:



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...**

	`(i,j-1)`	
`(i-1,j)`	`(i,j)`	`(i+1,j)`
	`(i,j+1)`	

So, how should we encode surfaces?

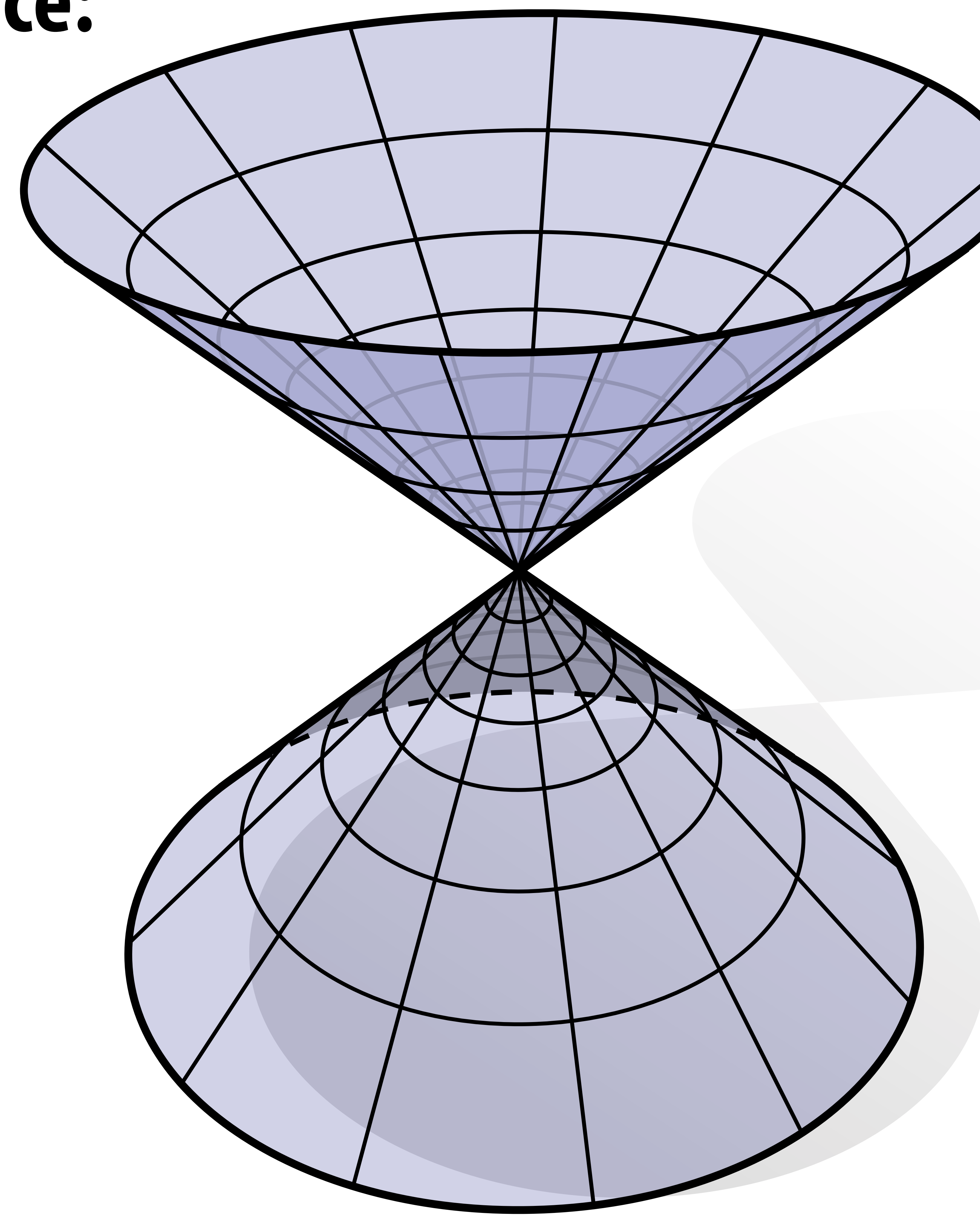
Smooth Surfaces

- 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

*...or can easily be flattened into the plane, without cutting or ripping.

Isn't every shape manifold?

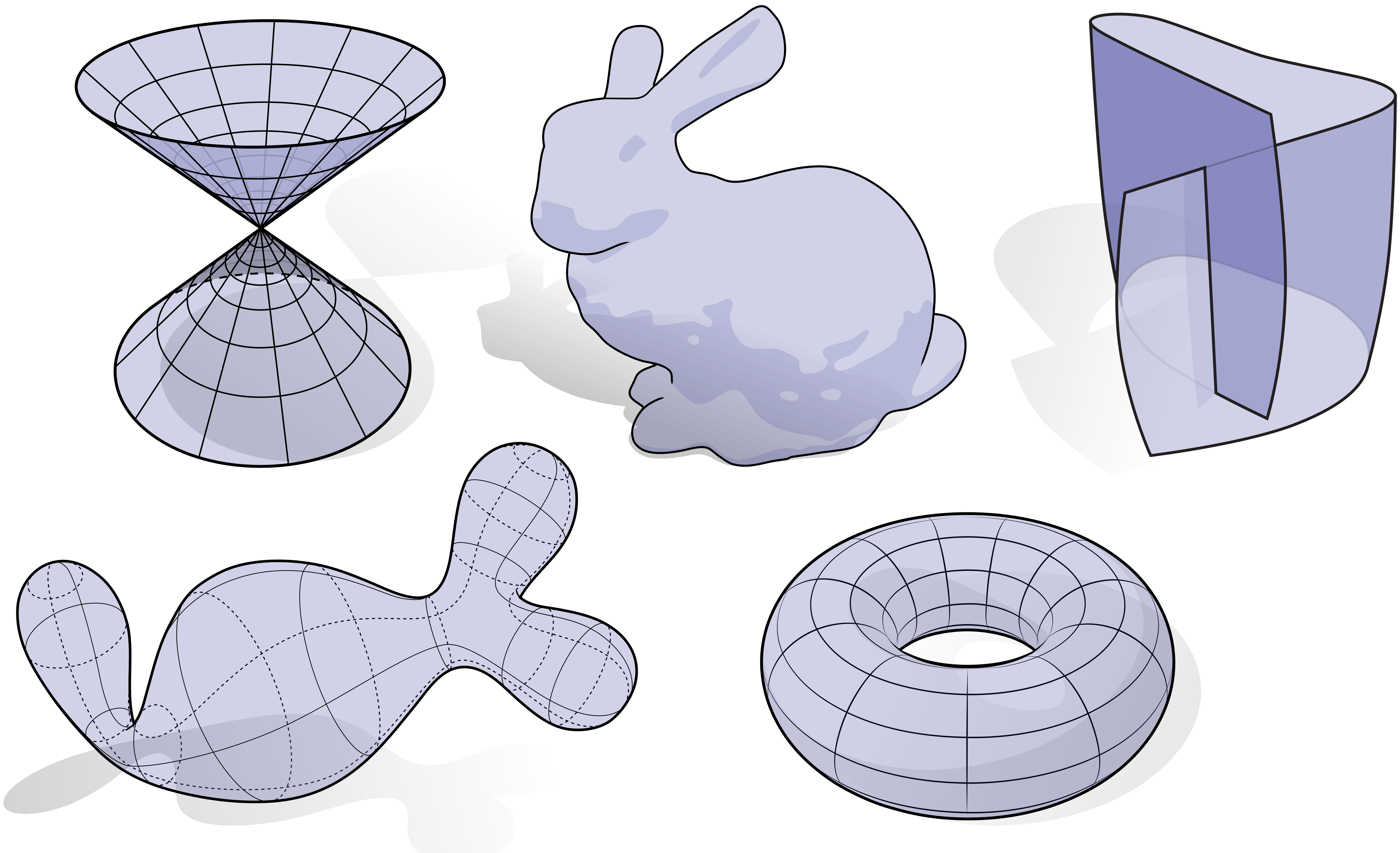
- No, for instance:



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

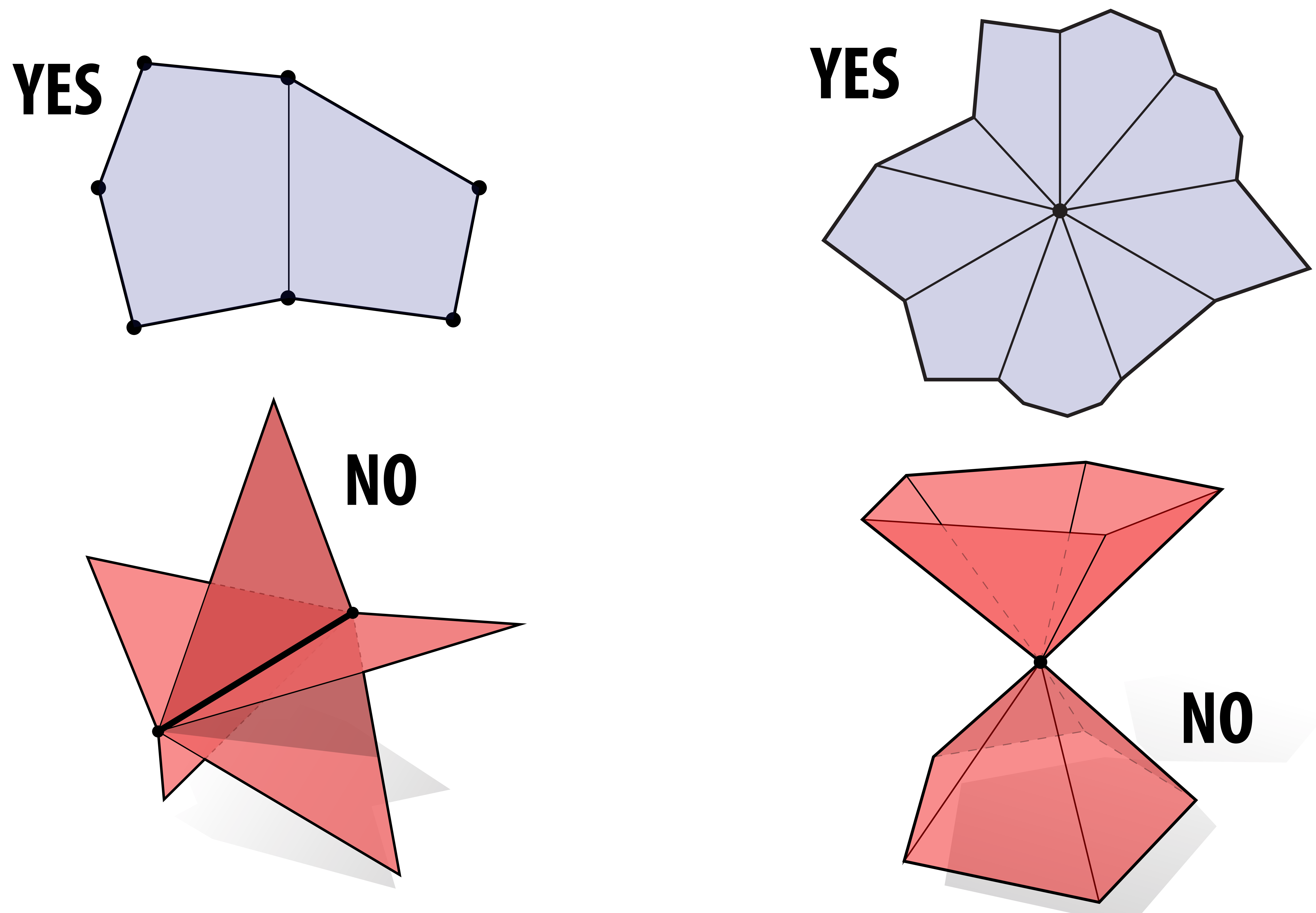
More Examples of Smooth Surfaces

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



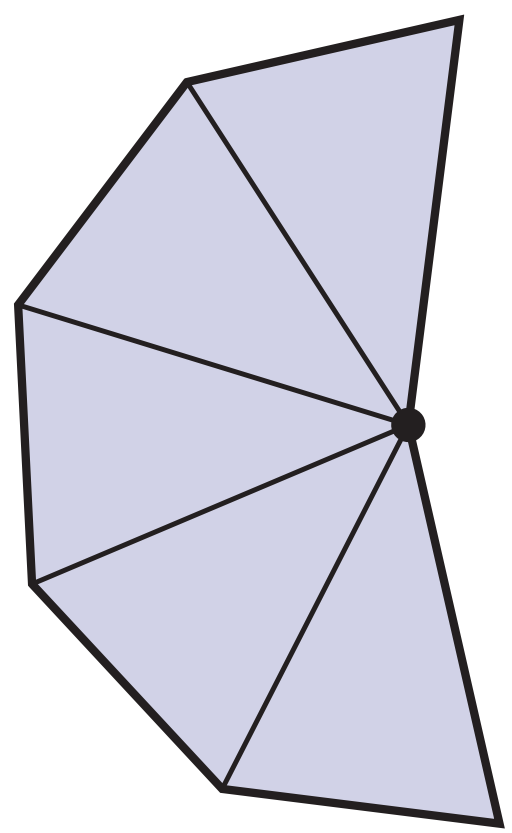
A manifold polygon mesh has fans, not fins

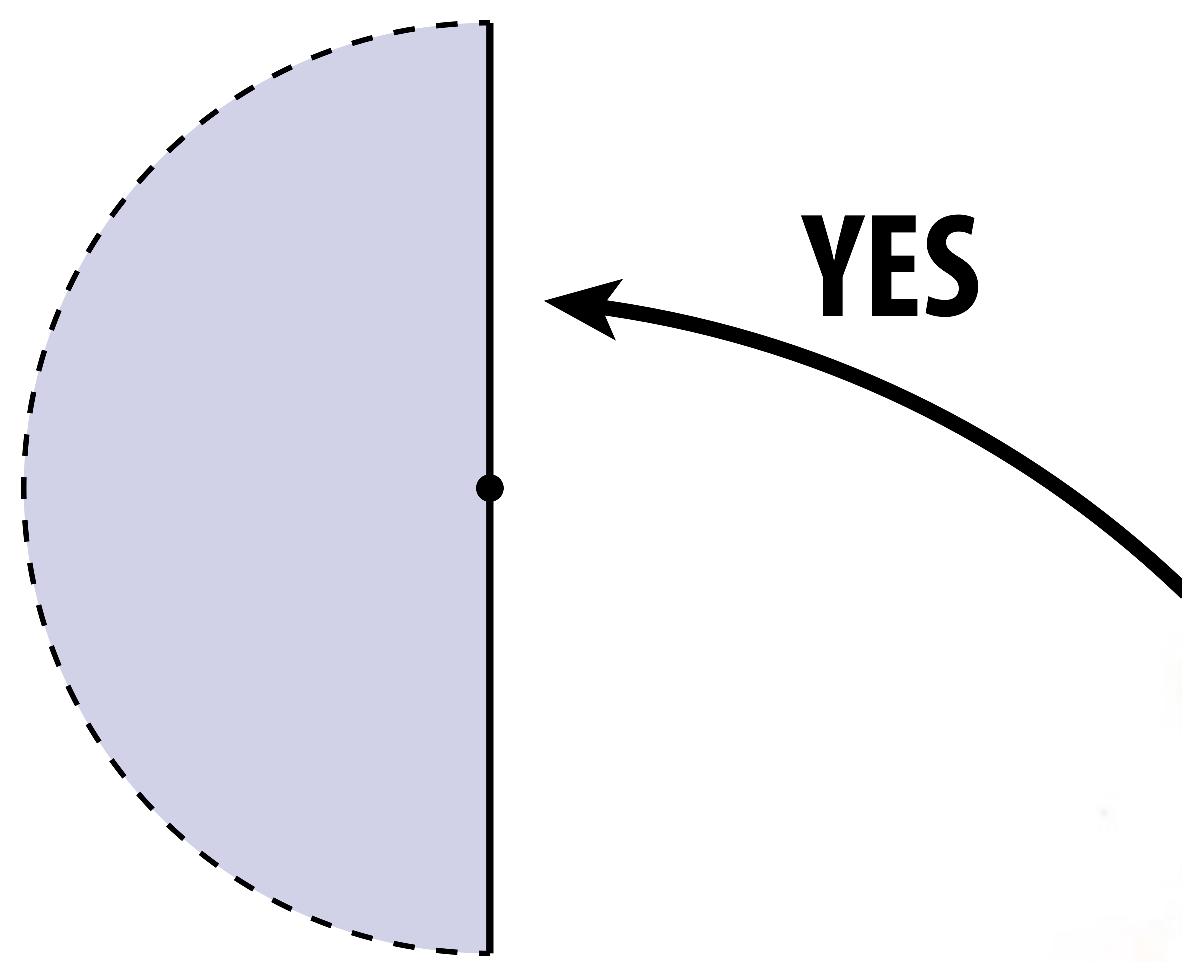
- For polygonal surfaces just two easy conditions to check:
  1. Every edge is contained in only two polygons (no “fins”)
  2. The polygons containing each vertex make a single “fan”



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



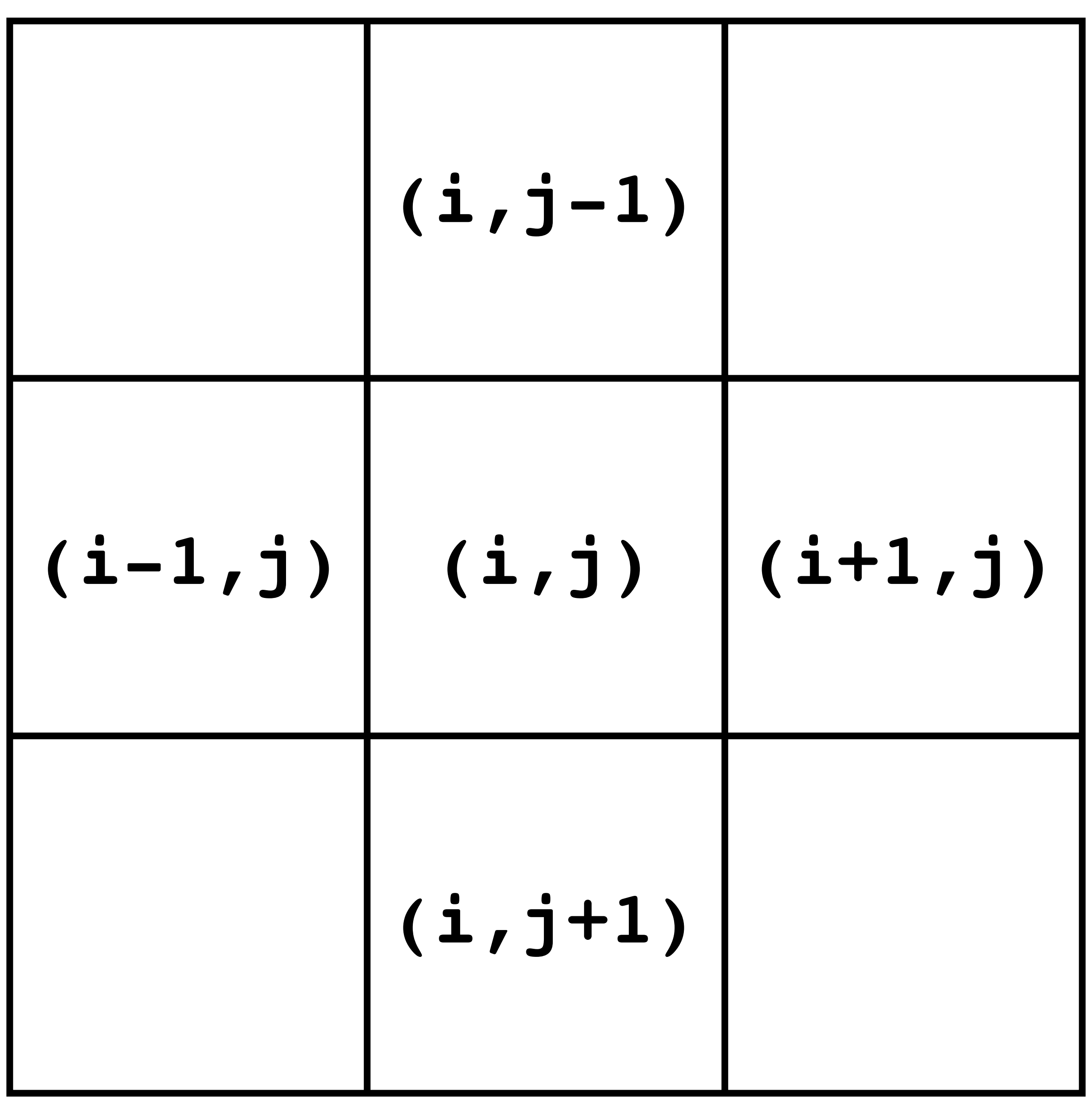


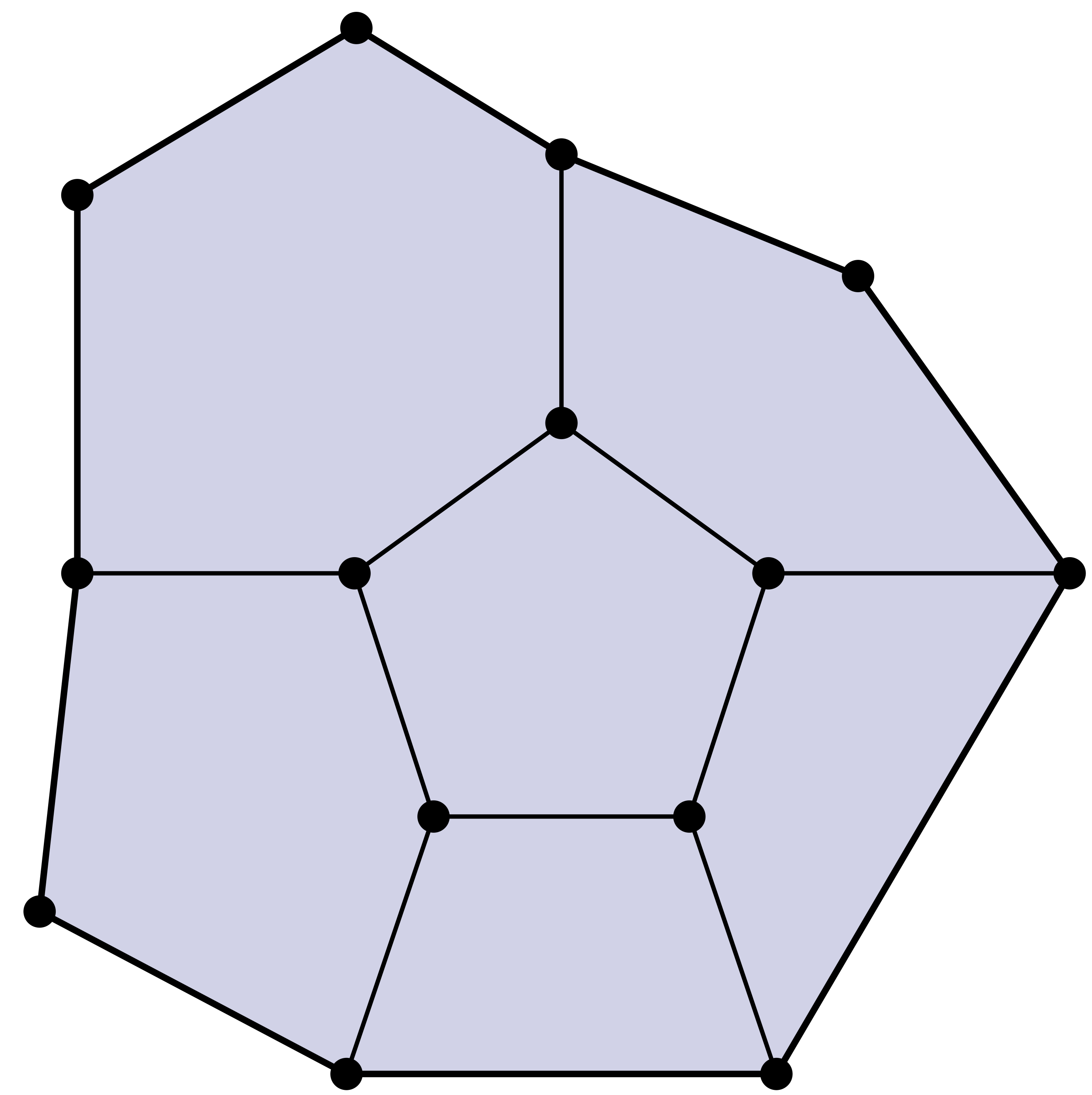
- Polygon mesh:
  - one polygon per boundary edge
  - boundary vertex looks like "pacman"

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

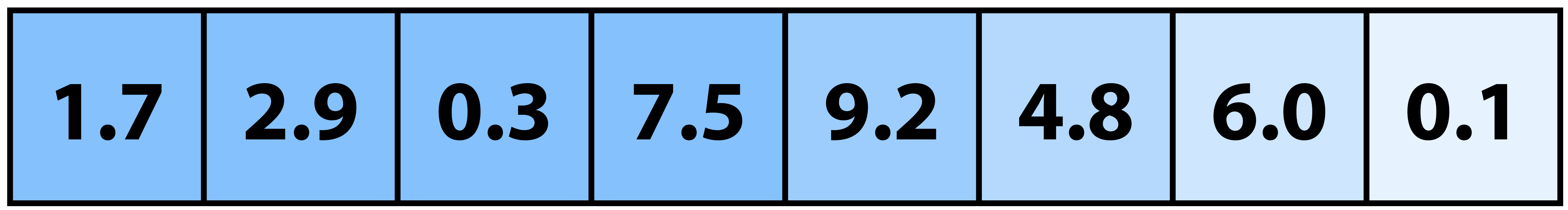




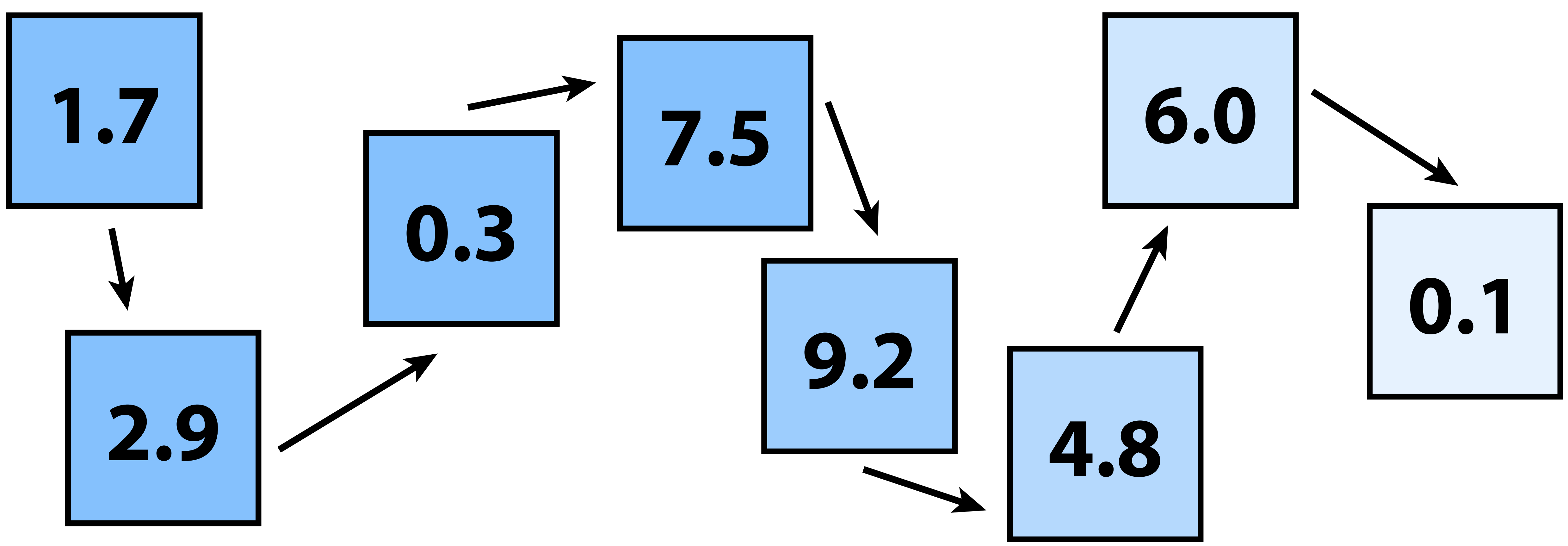
How do we actually encode all this data?

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)

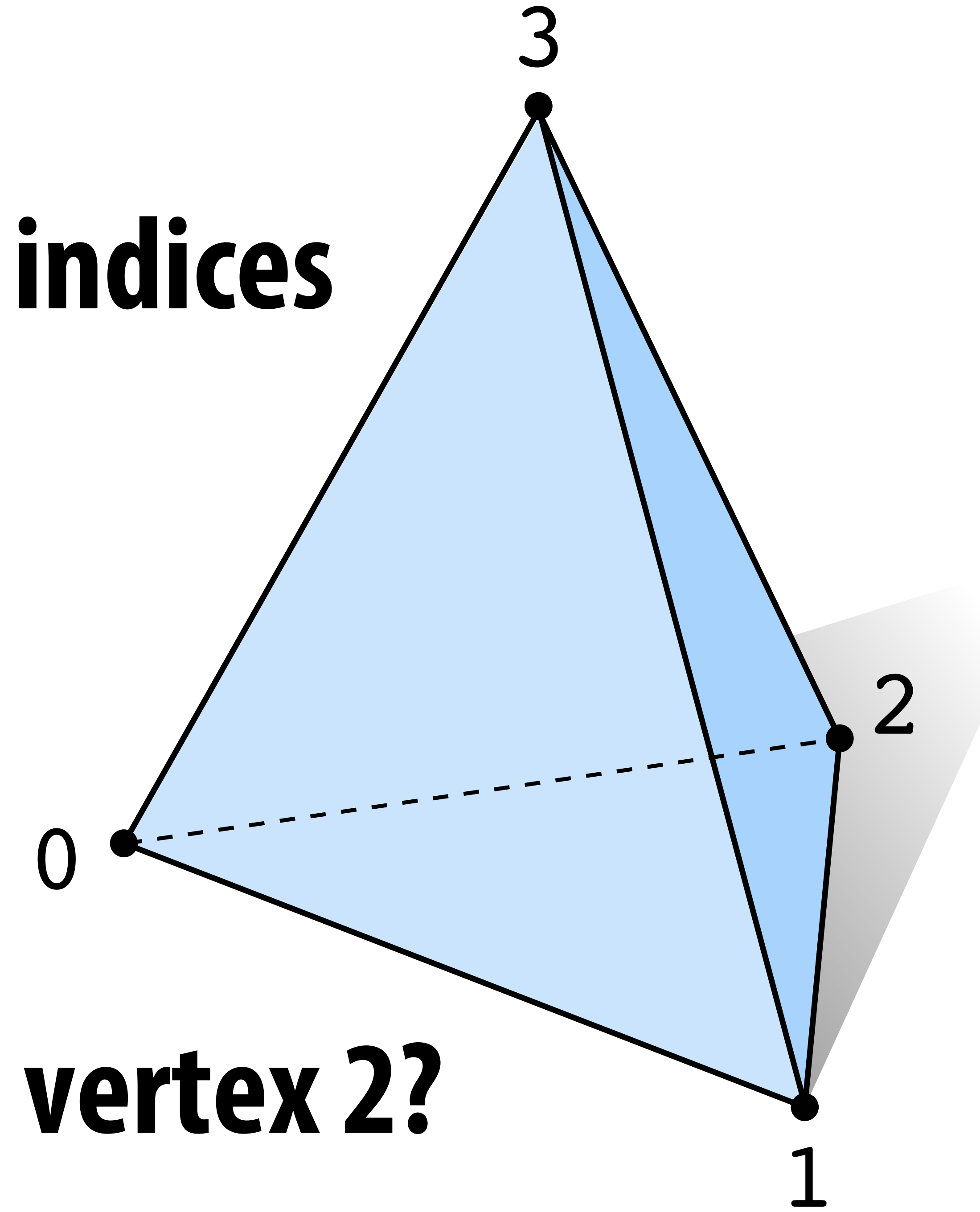
- **Store triples of coordinates (x,y,z), tuples of indices**
- **E.g., tetrahedron:**

VERTICES

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

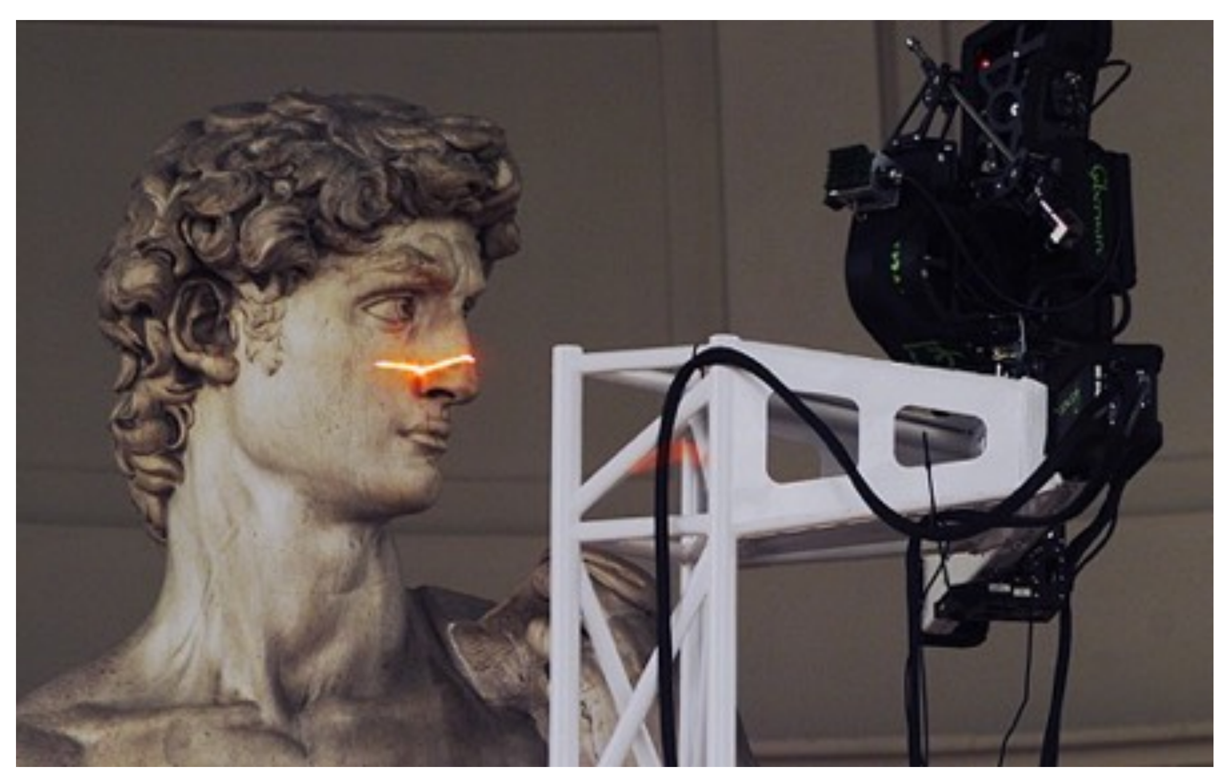
POLYGONS

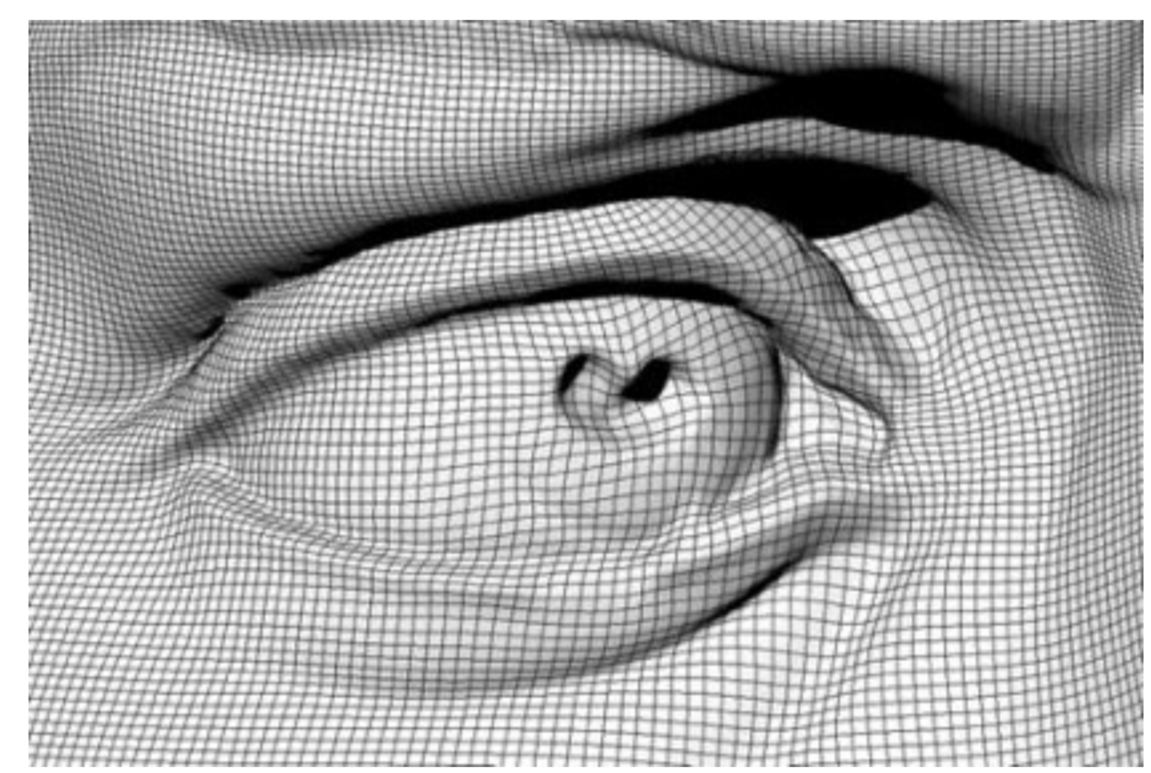
i	j	k
0	2	1
0	3	2
3	0	1
3	1	2



- **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?)**

Incidence Matrices

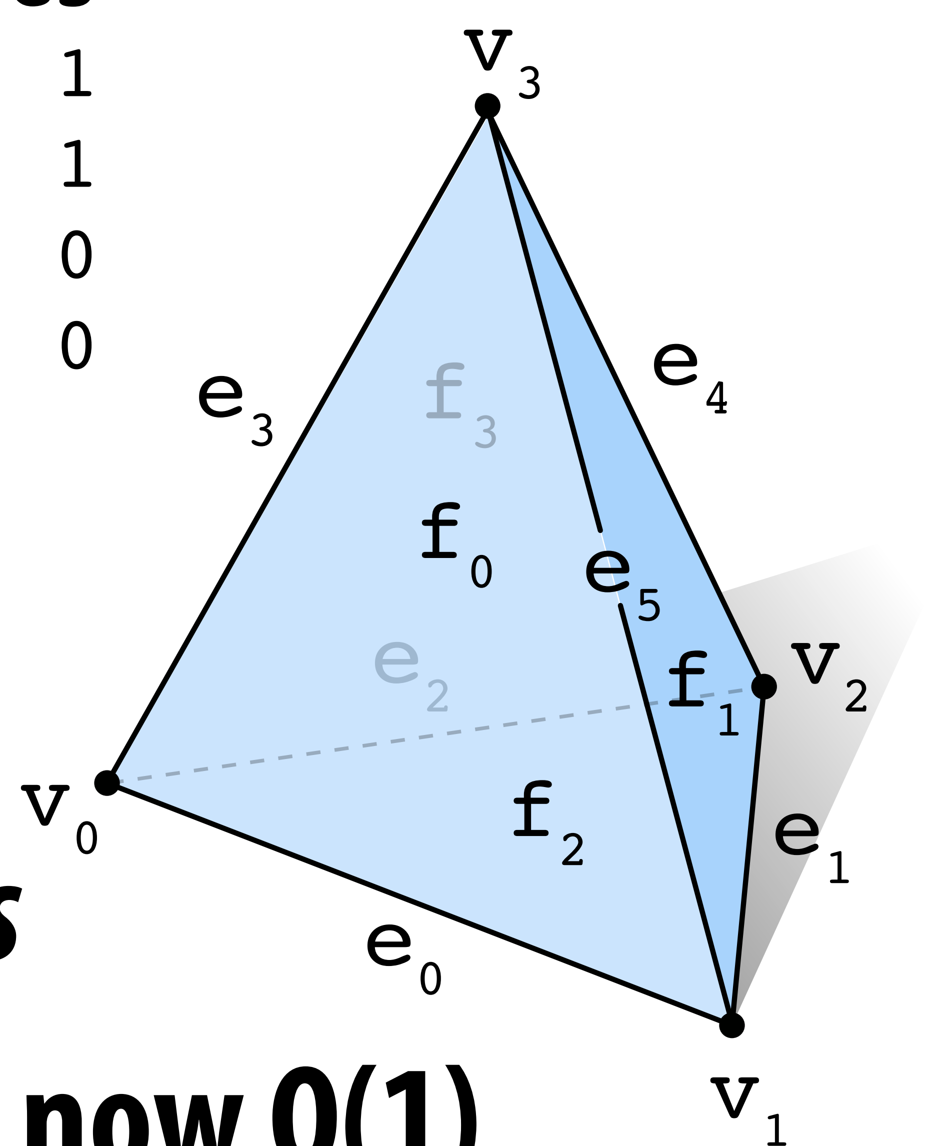
- If we want to answer neighborhood queries, why not simply store a list of neighbors?
- Can encode all neighbor information via *incidence matrices*
- E.g., tetrahedron:

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

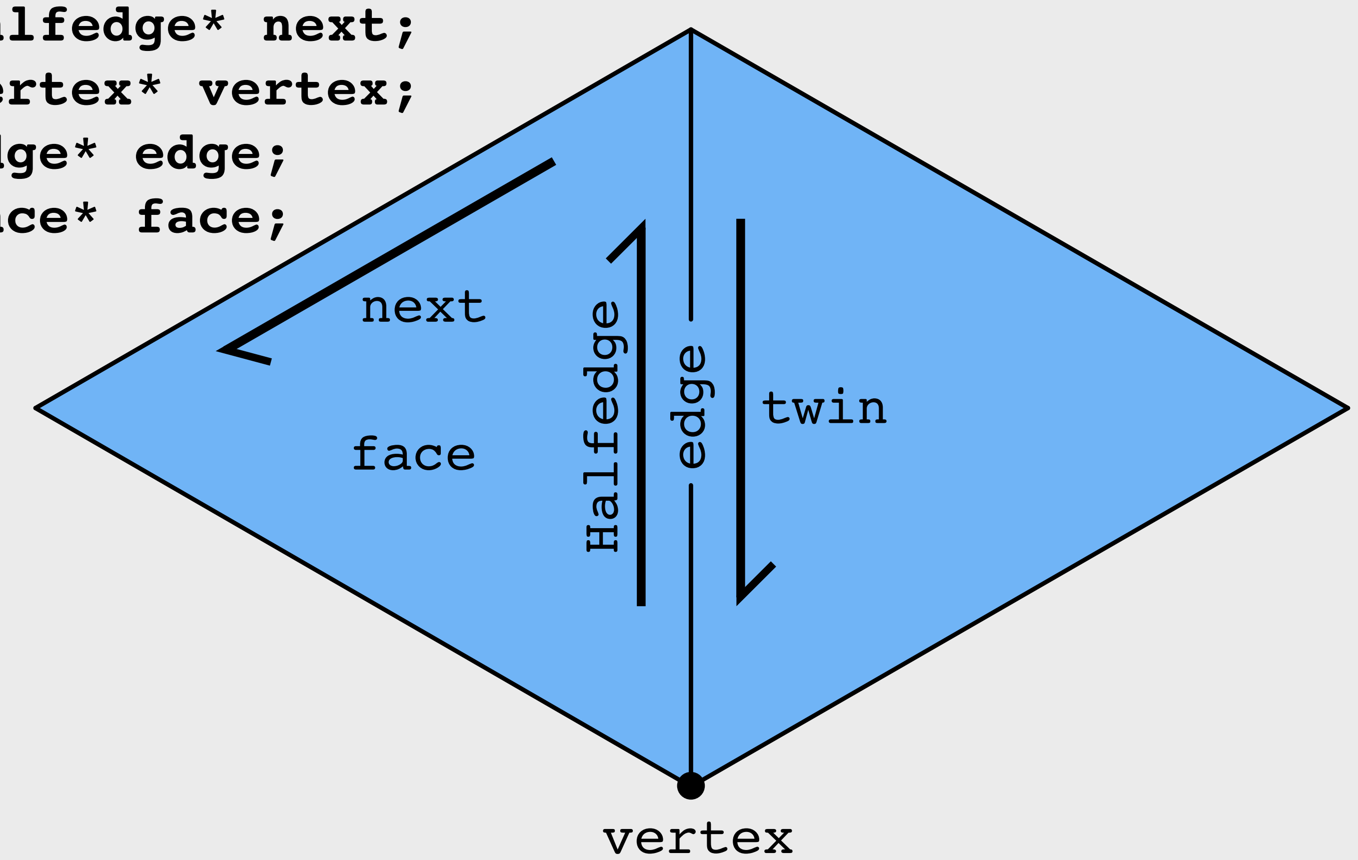


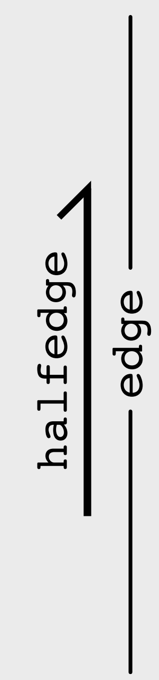
- 1 means "touches"; 0 means "does not touch"
- Instead of storing lots of 0's, use *sparse matrices*
- Still large storage cost, but finding neighbors is now O(1)
- Hard to change connectivity, since we used fixed indices
- Bonus feature: mesh does not have to be manifold

Halfedge Data Structure (Linked-list-like)

- 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:

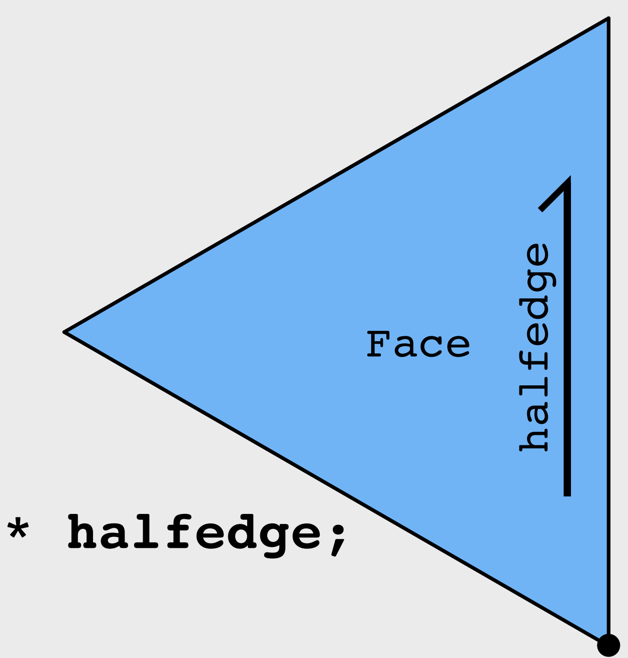
```
struct Halfedge
{
   Halfedge* twin;
   Halfedge* next;
   Vertex* vertex;
   Edge* edge;
   Face* face;
};
```

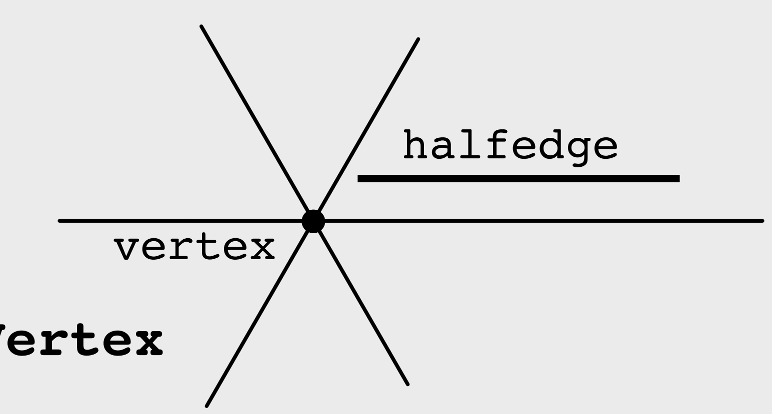





```
struct Edge
{
   Halfedge* halfedge;
};
```

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





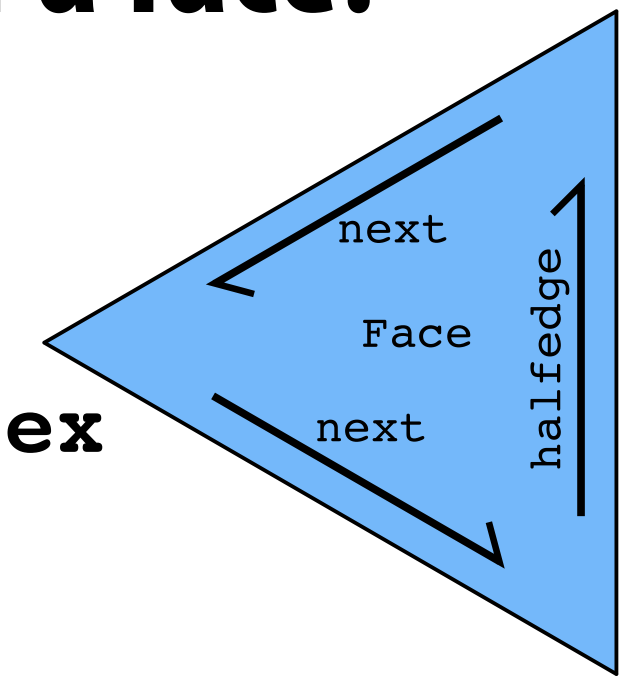

```
struct Vertex
{
   Halfedge* halfedge;
};
```

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

Halfedge makes mesh traversal easy

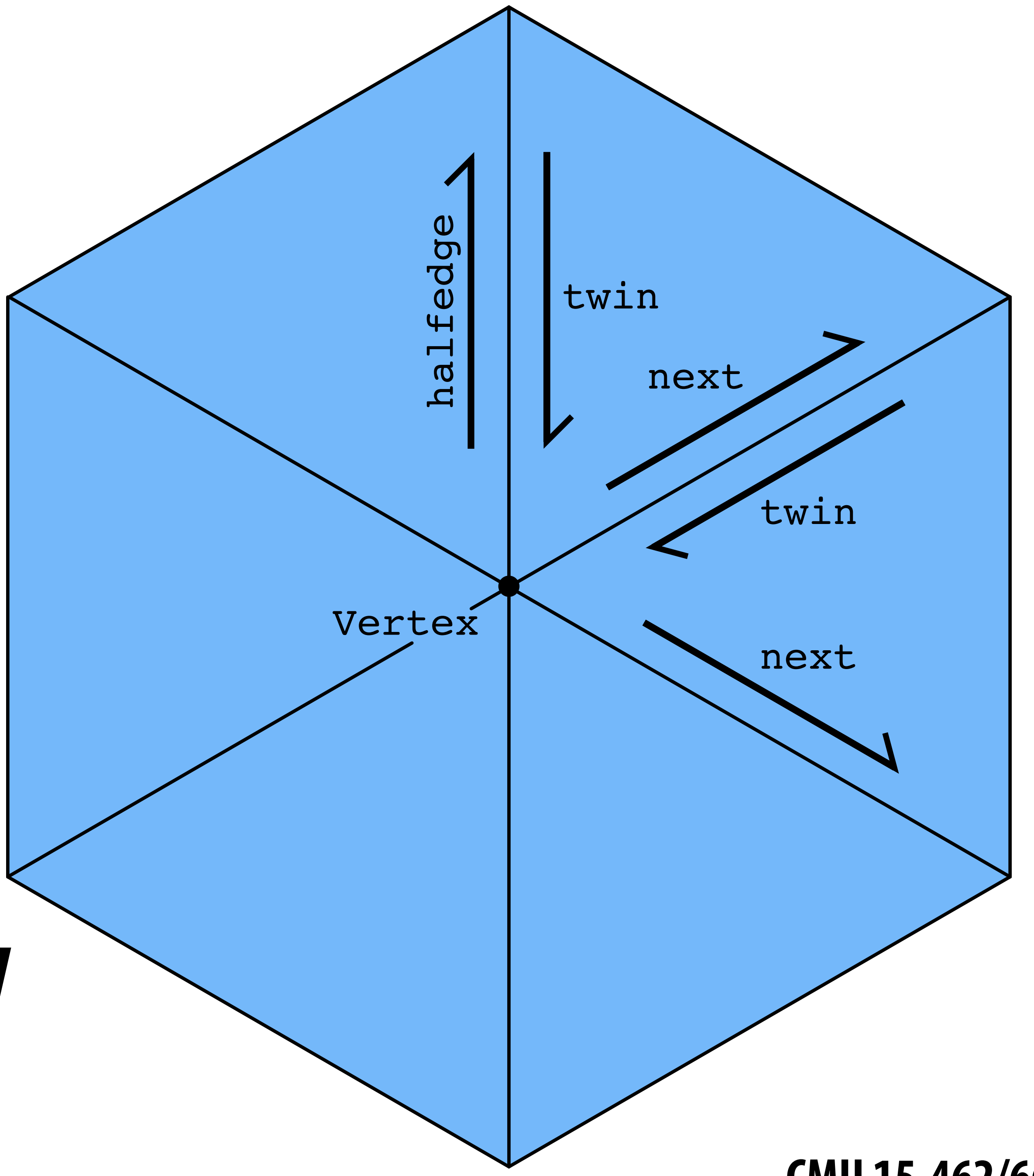
- Use "twin" and "next" pointers to move around mesh
- Use "vertex", "edge", and "face" pointers to grab element
- Example: visit all vertices of a face:

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



- Example: visit all neighbors of a vertex:

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



- [DEMO]
- Note: only makes sense if mesh is *manifold!*

Halfedge meshes are *always* manifold

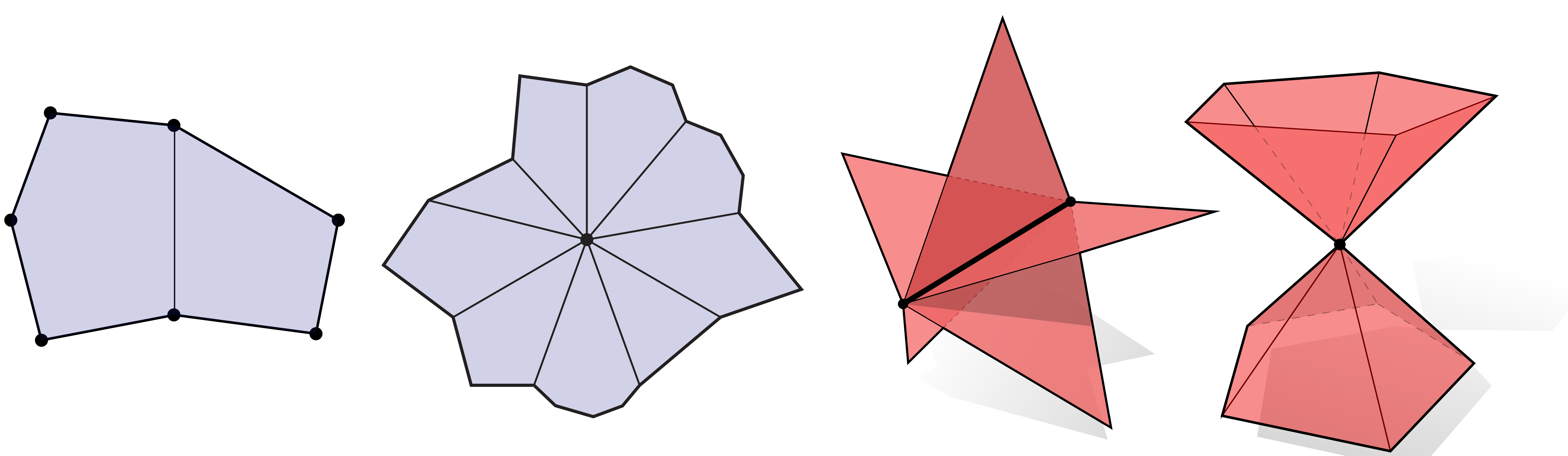
- Consider simplified halfedge data structure
- Require only "common-sense" conditions

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

(pointer to yourself!)

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

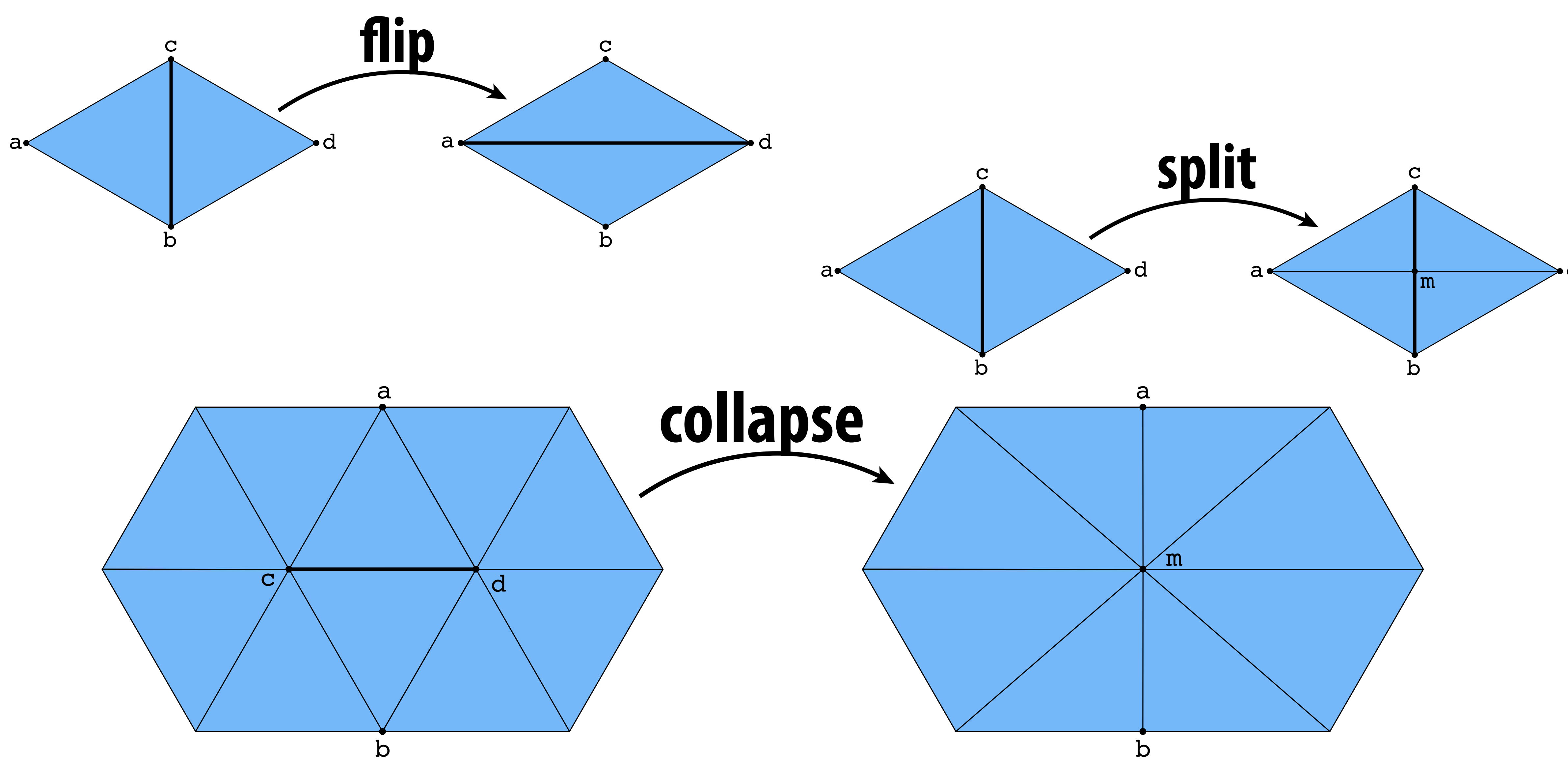
- Keep following `next`, and you'll get faces.
- Keep following `twin` and you'll get edges.
- Keep following `next->twin` and you'll get vertices.



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

Halfedge meshes are easy to edit

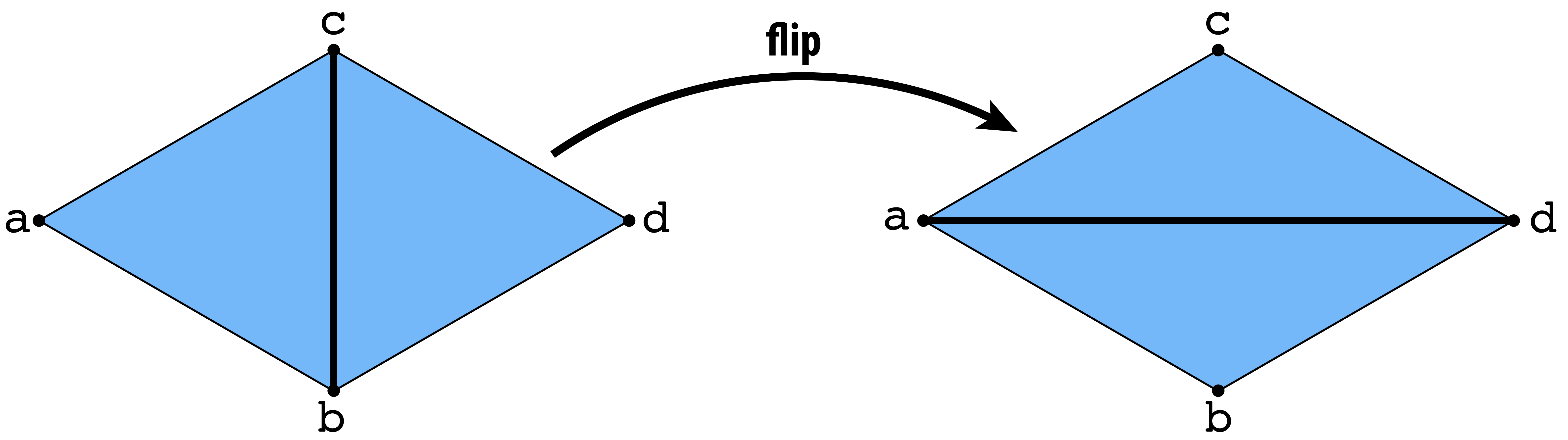
- Remember key feature of linked list: insert/delete elements
- Same story with halfedge mesh (“linked list on steroids”)
- E.g., for triangle meshes, several atomic operations:



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

Edge Flip (Triangles)

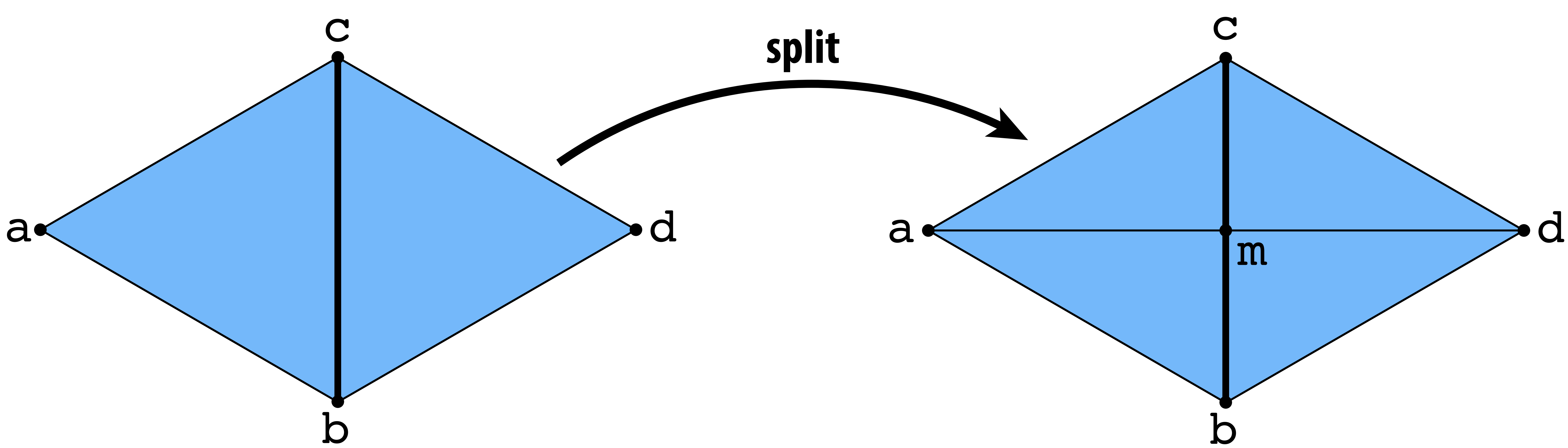
- Triangles (a,b,c), (b,d,c) become (a,d,c), (a,b,d):



- 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)

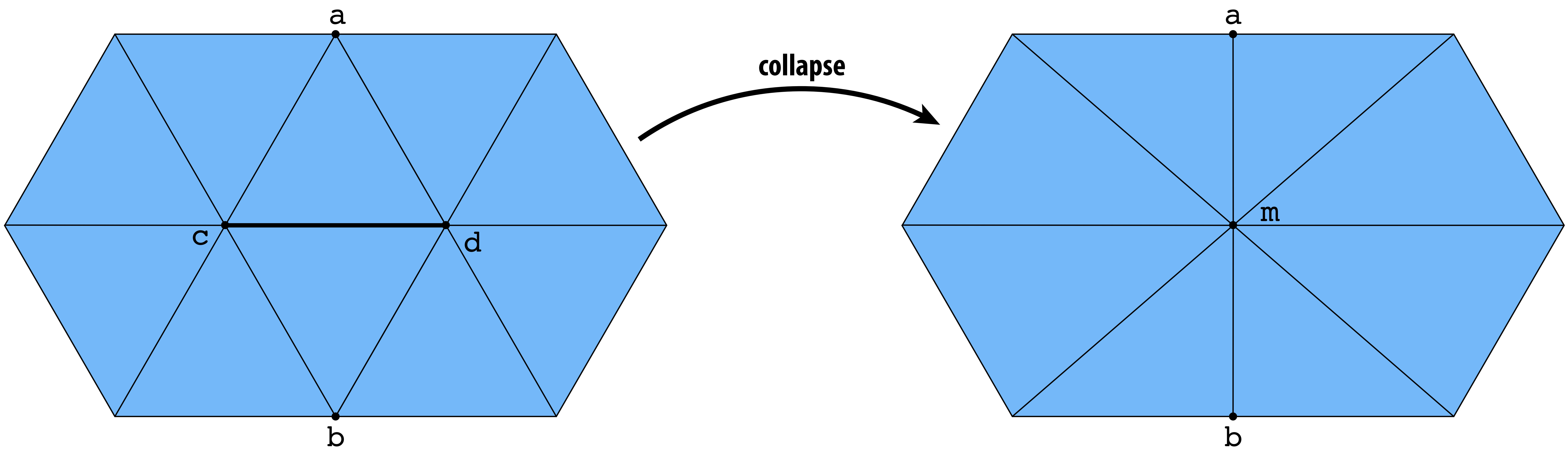
- Insert midpoint m of edge (c,b), connect to get four triangles:



- 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?)

Comparison of Polygon Mesh Data Strucutres

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

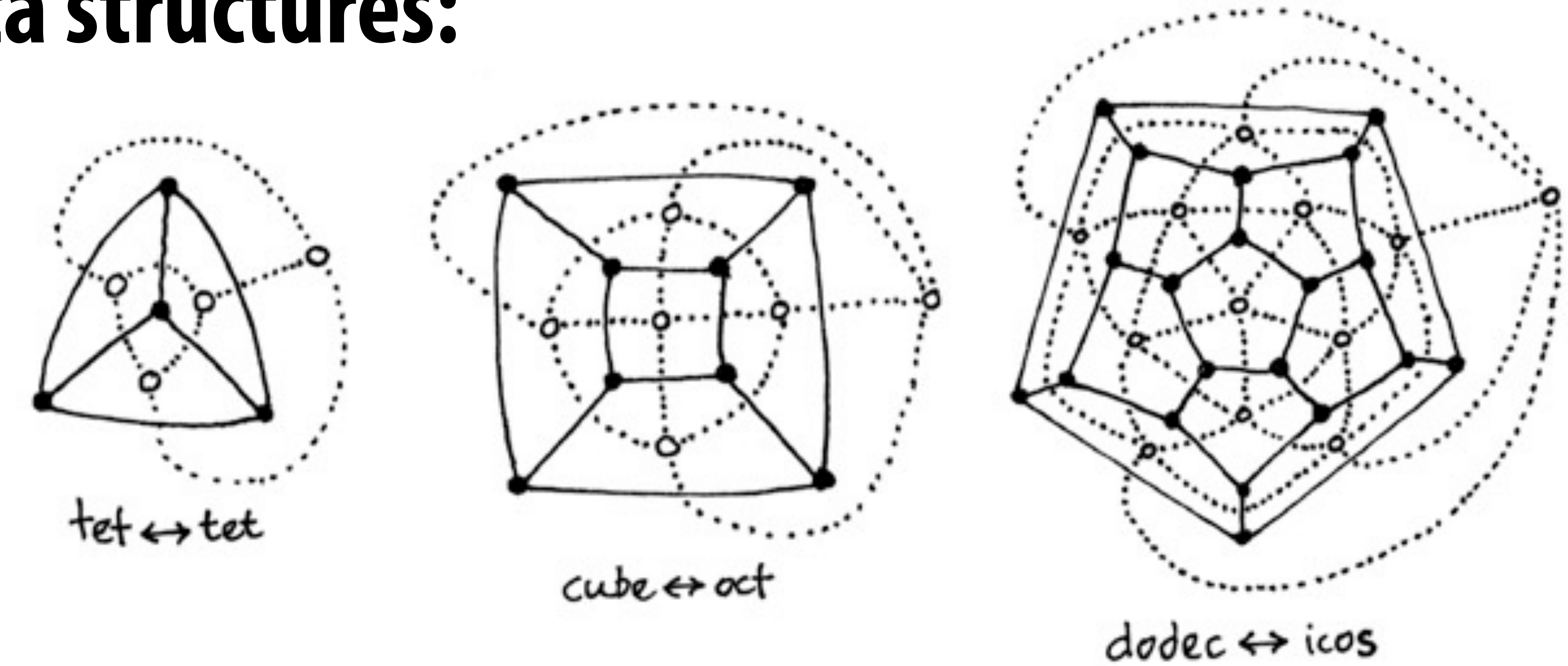
Conclusion: pick the right data structure for the job!

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

Alternatives to Halfedge

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

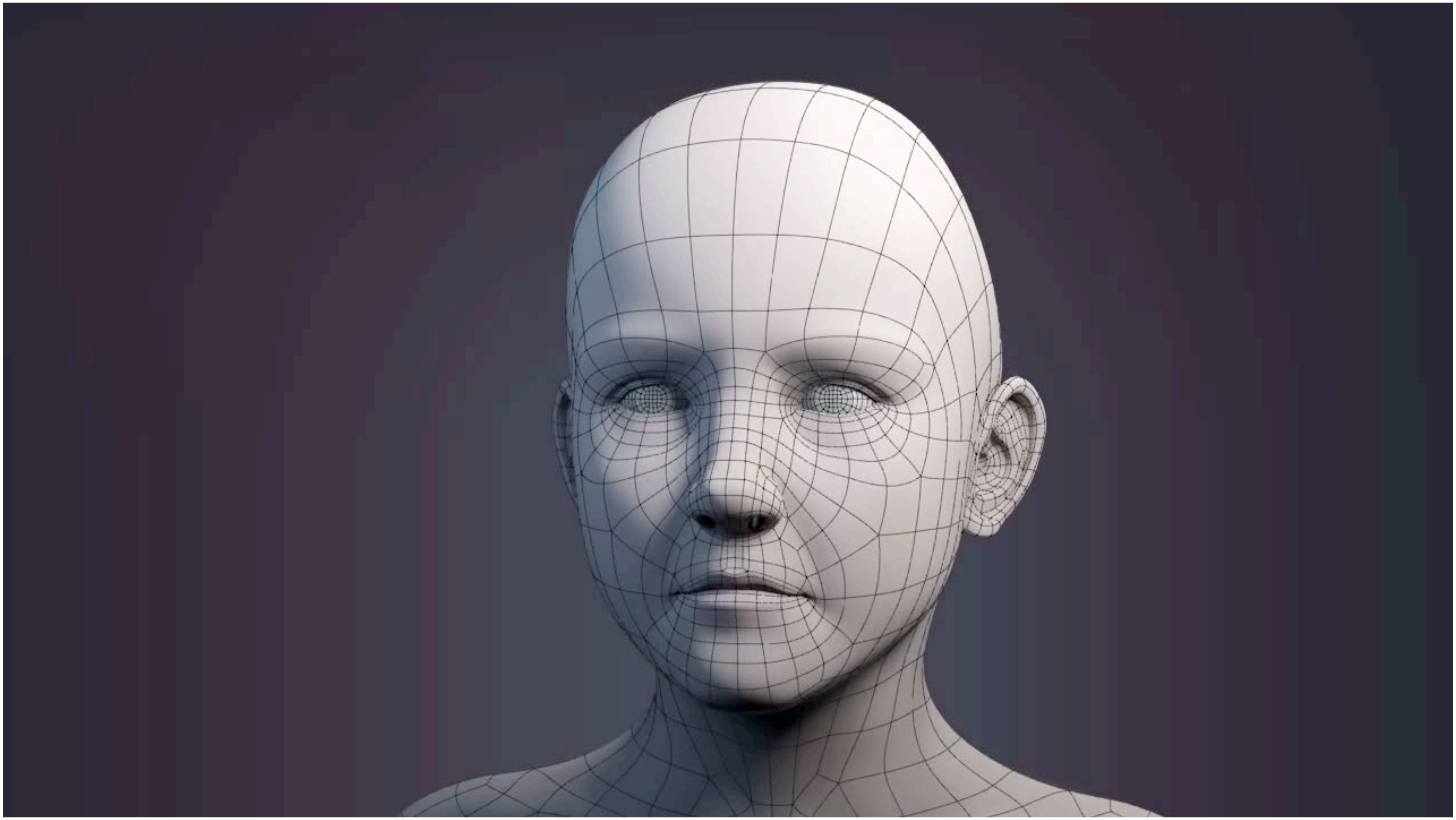
- Many very similar data structures:
  - winged edge
  - corner table
  - quadedge
  - ...



- 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?)

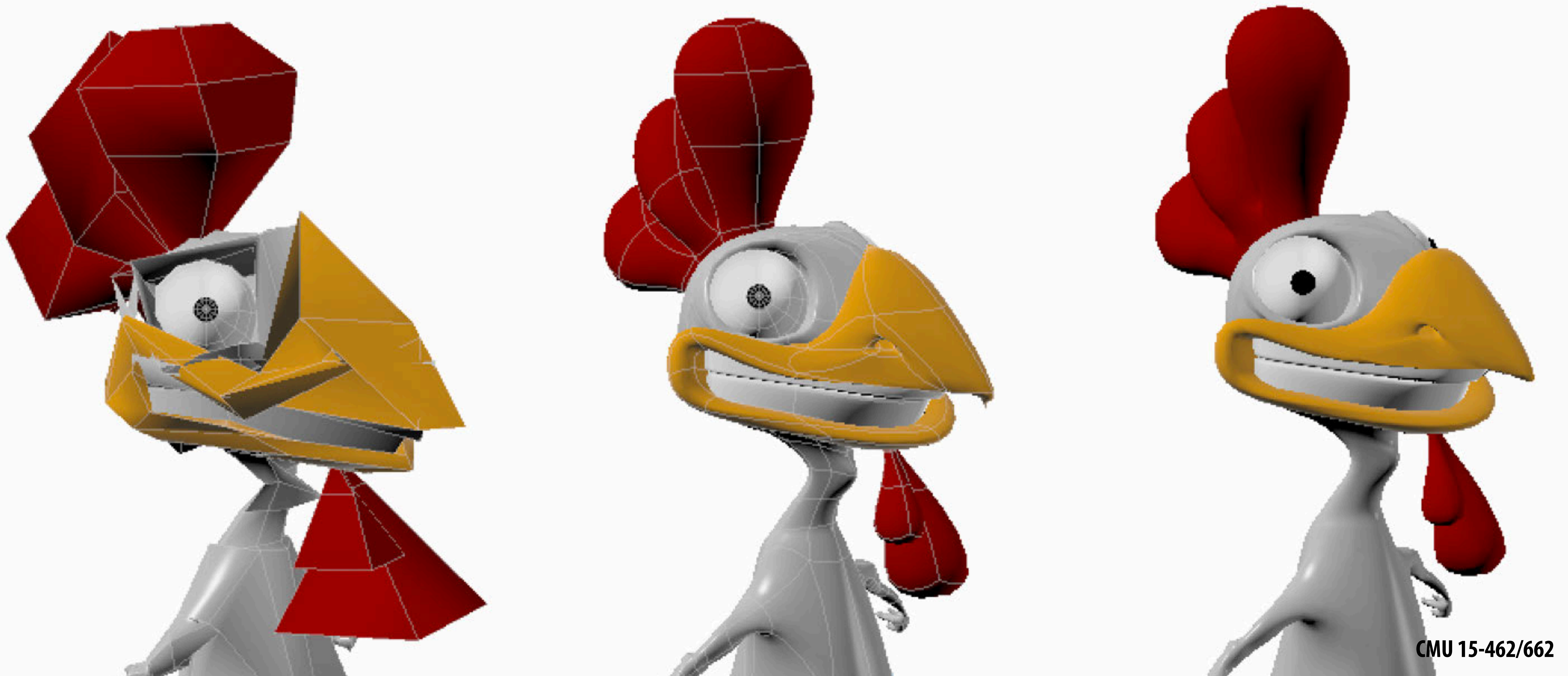
Ok, but what can we actually *do* with our fancy new data structure?

Subdivision Modeling



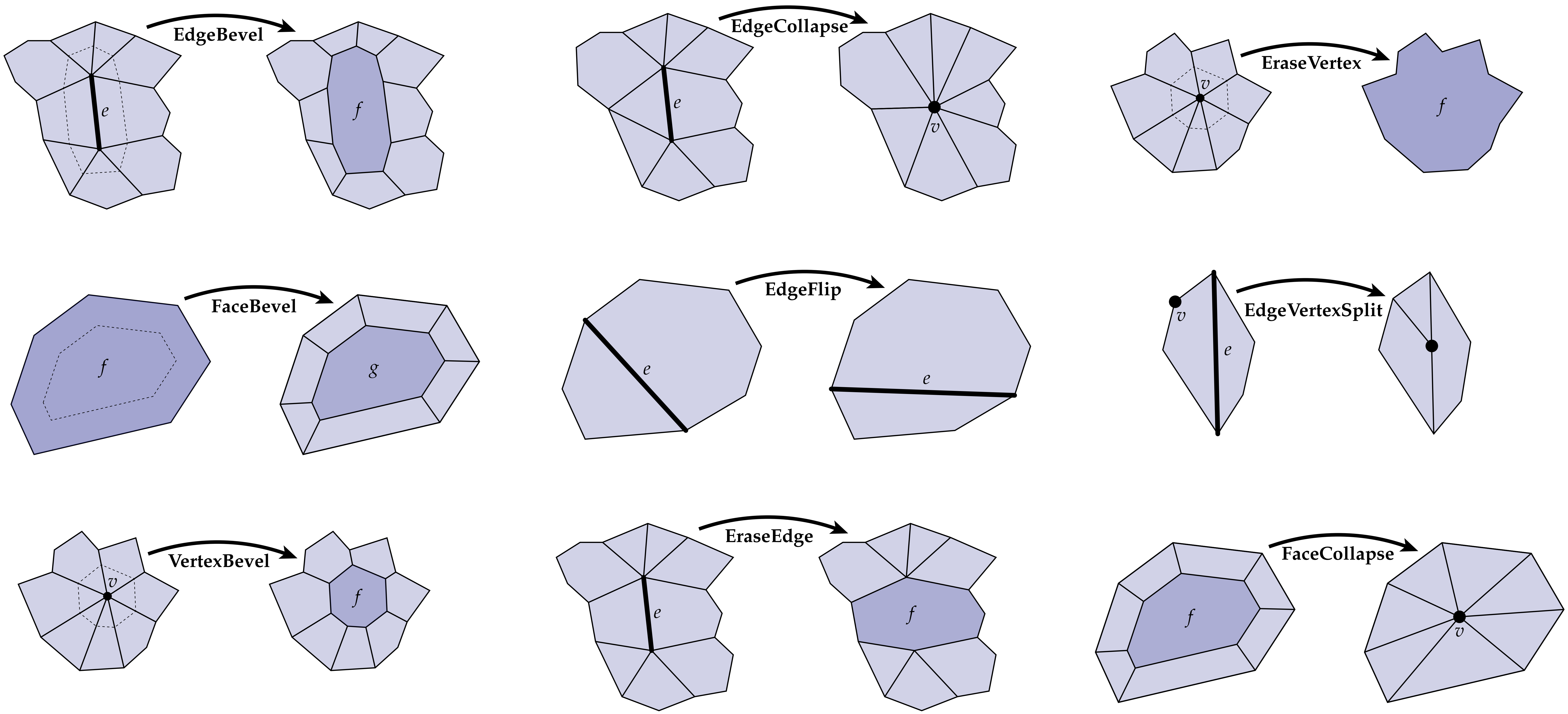
Subdivision Modeling

- 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

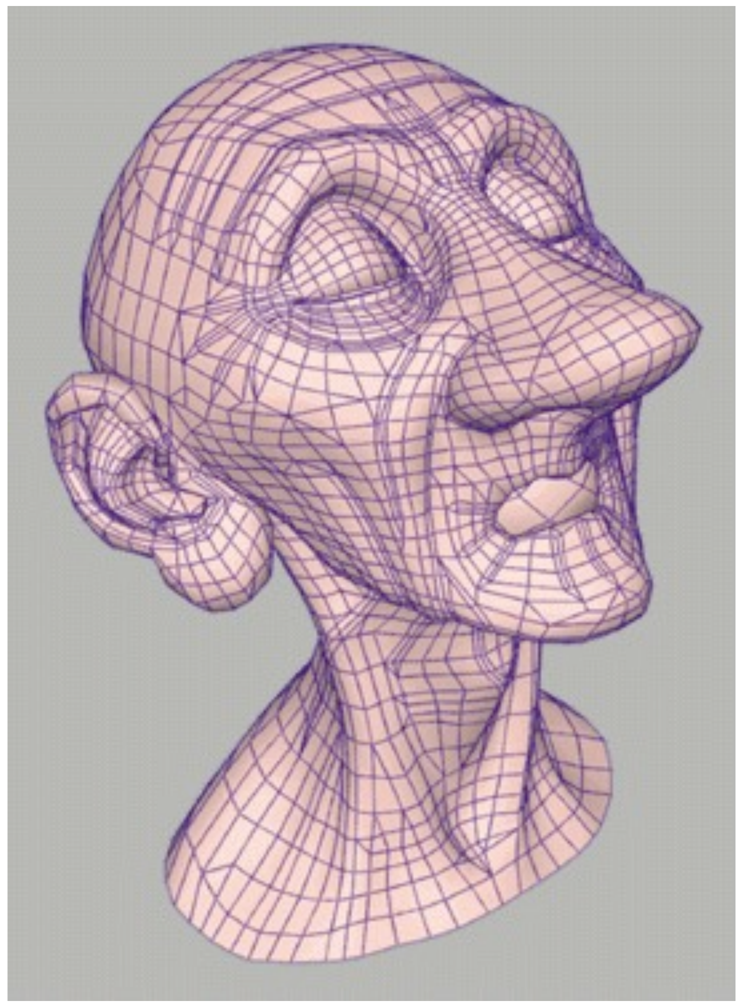
- **For general polygon meshes, we can dream up lots of local mesh operations that might be useful for modeling:**

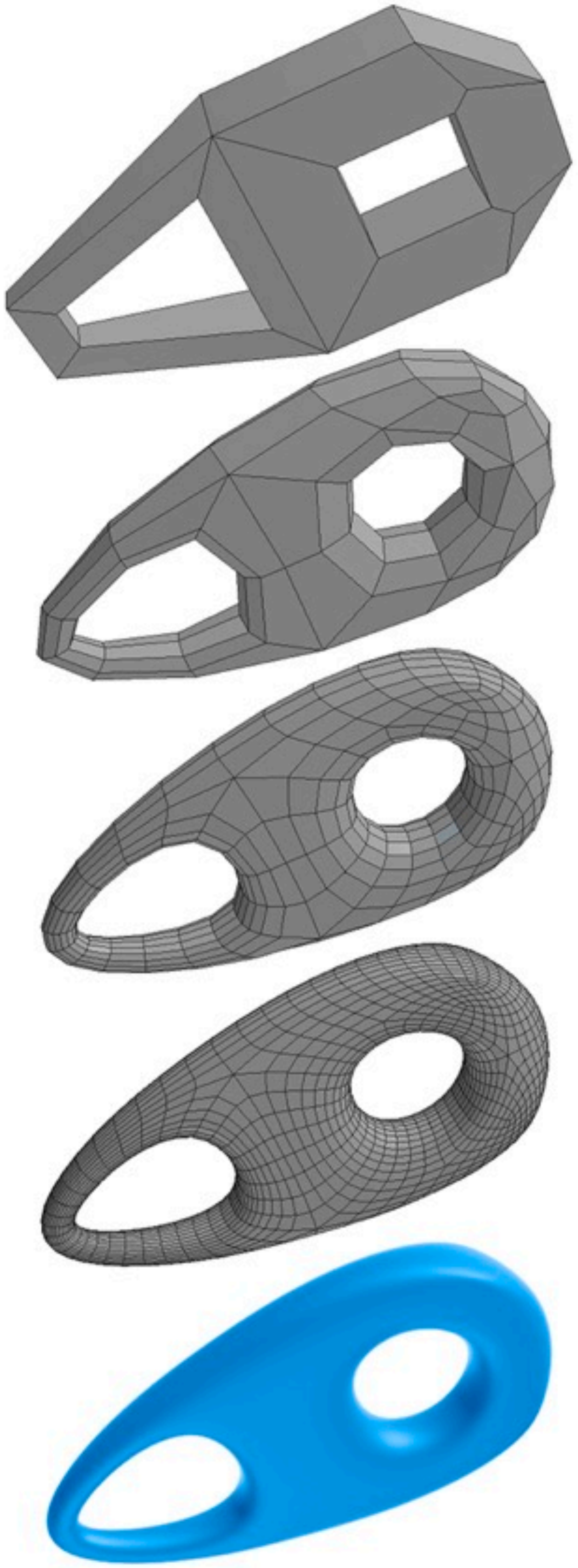


…and many, many more!

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





Next Time: Digital Geometry Processing

- Extend traditional digital signal processing (audio, video, etc.) to deal with *geometric* signals:
  - upsampling / downsampling / resampling / filtering ...
  - aliasing (reconstructed surface gives "false impression")
- Also some new challenges (very recent field!):
  - over which domain is a geometric signal expressed?
  - no terrific sampling theory, no fast Fourier transform, ...
- Often need new data structures & new algorithms