Midterm Review and Local Operations on Meshes

Computer Graphics 15-462 / 15-662

Two goals today:

- Guidance for midterm
- Local operations for A2.0

Midterm Review

Computer Graphics 15-462 / 15-662

Lecture 1: Introduction

- For any given setup where we place a camera in the environment, pointing down any of the main coordinate axes (x, y, or z), compute a projection of points in the world onto an image plane.
- Write an algorithm for drawing lines that handles all edge cases (i.e., including edges that are exactly horizontal or vertical).

Lecture 2: Math Review 1 (part 1 of 2):

- How can we measure vectors?
- What is a Vector Space?
- Draw a geometric representation of each rule that vectors seem to obey.
- Can a function be a vector? Explain
- Add and scale vectors.
- Add and scale functions
- What is the norm of a vector?
- Associate each property of a norm with a geometric interpretation.
- Compute the Euclidean norm in Cartesian coordinates.
- Compute the L2 norm of a function.
- Associate each property of the inner product with a geometric interpretation.
- Compute the inner product in Cartesian coordinates.

Lecture 2: Math Review 1 (part 2 of 2):

- Use the inner product for operations such as projection.
- Compute the inner product of functions.
- Give properties and an example of a linear map.
- Define span and basis..
- Compute an orthonormal basis from a set of vectors.
- Be able to use Gram-Schmidt orthonormalization.
- Know that orthonormalization of functions can be done by decomposing them into sinusoids.
- Be able to solve a simple system of linear equations, depict it geometrically, and represent it in matrix form.
- Be able to represent a linear map in matrix form.

Lecture 3: Math Review 2 (Part 1 of 2):

- Euclidean norm is any notion of length preserved by rotations/translations/ reflections of space. Be able to calculate it for a vector of any dimension.
- Compute the inner product of two n-dimensional vectors. What is the geometric meaning if an orthonormal basis is used?
- Compute the cross product of two three-dimensional vectors. What is the geometric meaning of this cross product? What is the geometric meaning of its magnitude?
- Use the cross product to do a quarter rotation of a vector within a plane.
- Represent the dot product using matrix notation.
- Represent the cross product using matrix notation.
- Understand what the determinant measures. What does the determinant of a linear map tell us? Give an example.
- What is a directional derivative?
- Bonus: Compute the gradient of a function.

Lecture 3: Math Review 2 (Part 2 of 2):

- Understand gradient as the best linear approximation and the direction of steepest ascent.
- When is the gradient not defined?
- Bonus: Be able to express gradients of simple matrix expressions
- What is a vector field? Give an example.
- Be able to compute divergence, curl, and the Laplacian of a vector field. Also be able to express the meaning of these terms geometrically, for example by drawing a diagram.
- What is the Hessian? Be able to compute the Hessian of a function.

Lecture 4: Drawing a Triangle

1. How should we choose the correct color for a pixel? There is not an exact right answer. However, you should be able to discuss some of the issues involved.
2. What is aliasing, and what artifacts does it produce in our images and our animations?
3. One form of aliasing is where high frequencies masquerade as low frequencies. Give an example of this phenomenon.
4. Suppose we have a single red triangle displayed against a blue background. Does this scene contain high frequencies?
5. What does the Nyqvist-Shannon theorem tell us about how image frequencies relate to required sampling rate?
6. One practical solution on your graphics card for reducing aliasing (i.e., for antialiasing) is to take multiple samples per pixel and average to get pixel color. Try to use what we learned about sampling theory to explain as precisely as you can why taking multiple samples per pixel can reduce aliasing artifacts.
7. Bonus: Be able to write an implicit representation of an edge given two points.
8. Bonus: Be able to use the implicit edge representation to determine if a point is inside a triangle.

Lecture 5: Transforms (Part 1 of 2)

1. Which of the following operations are linear transforms: scale, rotation, shear, translation, reflection, rotation about a point that is not the origin?
2. Express scale as a linear transform
3. Express rotation as a linear transform
4. Express shear as a linear transform
5. Express reflection as a linear transform
6. Express translation as an affine transform
7. Know what makes a transform linear vs. affine
8. Know how to build transformation matrices from start and end configurations of your object

Lecture 5: Transforms (Part 2 of 2)

- Create 2D and 3D transformation matrices to perform specific scale, shear, rotation, reflection, and translation operations
- Compose transformations to achieve compound effects
- Rotate an object about a fixed point
- Rotate an object about a given axis
- Create an orthonormal basis given a single vector
- Understand the equivalence of [x y 1] and [wx wy w] vectors
- Explain/illustrate how translations in 2D (x, y) are a shear operation in the homogeneous coordinate space (x, y, w)

Lecture 6: 3D Rotations

- What is the problem of gimbal lock? Give an example where this problem occurs.
- How does using quaternions solve this problem?
- Know that every rotation can be expressed as rotation by some angle about some axis.
- Know how to go between quaternions and axis-angle format for rotations.
- Know that quaternions are expressed as higher dimensional complex numbers.
- Be able to work out quaternion multiplication from the complex number representation of a quaternion.

Lecture 7: Projection

- Review:
  - Form an orthonormal basis
  - Create a rotation matrix to rotate any coordinate frame to xyz
  - Create the rotation matrix to rotate the xyz coordinate frame to any other frame
  - Know basic facts about rotation matrices / how to recognize a rotation matrix
    - Rows (also columns) are unit vectors
    - Rows (also columns) are orthogonal to one another
    - If our rows (or columns) are u, v, and w, then uXv=w
    - The inverse of a rotation matrix is its transpose
- Create a projection matrix that projects all points onto an image plane at z=1
- Propose a projection matrix that maintains some depth information
- Understand the motivation behind the projection matrix that projects the view frustum to a unit cube
- Be able to draw / discuss the details of the view frustum
- Prove that a standard projection matrix preserves some information about depth

Lecture 7: Barycentric Coordinates

- Interpolate colors using barycentric coordinates
- Bonus: Compute barycentric coordinates of a point using implicit edge functions
- Compute barycentric coordinates of a point using triangle areas
- Estimate the location of a point inside a triangle given its barycentric coordinates
- Estimate the location of a point outside a triangle given its barycentric coordinates

- Estimate barycentric coordinates of a point from a drawing.
- Show that interpolation in 3D space followed by projection can give a different result from projection followed by interpolation in screen space. In other words, explain why interpolation using barycentric coordinates in screen space may give a result that is incorrect.
- How, then, can we obtain a correct result using interpolation in screen space?

Lecture 8: Textures

- Textures are used for many things, beyond pasting images onto object surfaces.
  - Normal maps (create appearance of bumpy object on smooth surface by giving false normal to the lighting equations)
  - Displacement maps (encode offsets in the geometry of a surface, which is difficult to handle in a standard graphics pipeline)
  - Environment maps (store light information in all directions in a scene)
  - Ambient occlusion map (store exposure of geometry to ambient light for better representation of surface appearance with simple lighting models)
  - Can you think of / discover others?
- Know how to interpolate texture coordinates
- Know how to index into a texture and compute a correct color using bilinear interpolation
- Be able to create a mipmap and store it in memory
- Be able to compute color from multiple levels of mipmaps using trilinear interpolation
- What is the logic behind selecting an appropriate level in a mipmap?
- What can happen if we select a level that is too high resolution? too low resolution?

Lecture 8: Depth and Transparency

- What is the depth buffer (Z-buffer) and how is it used for hidden surface removal?
- Where does the depth for each sample / fragment come from? Where is it computed in the graphics pipeline?
- Is the depth represented in the depth buffer the actual distance from the camera? If not, what is it?
- What is the meaning of the alpha parameter in the [R G B a] color representation?
- Be able to use alpha to do compositing with the "Over" operator.
- Is "Over" commutative? If not, create a counterexample.

$$C = \alpha_B B + (1 - \alpha_B)\alpha_A A$$

- What is premultiplied alpha, and how does it work?
- Be able to use premultiplied alpha for "Over" composition.
- Why is premultiplied alpha better?
- How do we properly render a scene with mixed opaque and semi-transparent triangles? What is the rendering order we should use? When is the depth buffer updated?
- Draw a rough sketch of the graphics pipeline. Think about transforming triangles into camera space, doing perspective projection, clipping, transforming to screen coordinates, computing colors for samples, computing colors for pixels, the depth test, updating color and depth buffers.

Lecture 9: Introduction to Geometry

- List some types of implicit surface representations
- What types of operations are easy with implicit surface representations?
- List some types of explicit surface representations
- What types of operations are easy with explicit surface representations?
- What is CSG (constructive solid geometry)? Give some examples of CSG operations.
- What type of representation is best for CSG operations?
- Describe how to do union, intersection, and subtraction of geometry using simple operators on a surface representation.
- What is a level set representation? When is it useful?
- What types of splines are common in computer graphics?
- Why are they popular? What properties make them most useful?
- BONUS: Derive the equation on the slide labeled "Bézier Curves — tangent continuity" from the definitions given on the previous slides. Draw a diagram to illustrate any terms you use.

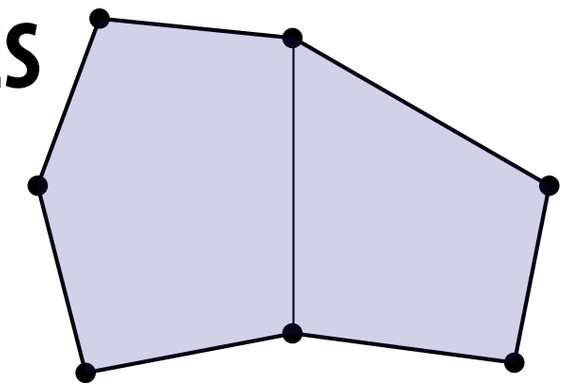
Meshes and Geometry Processing (Local Operations)

Computer Graphics
CMU 15-462/15-662

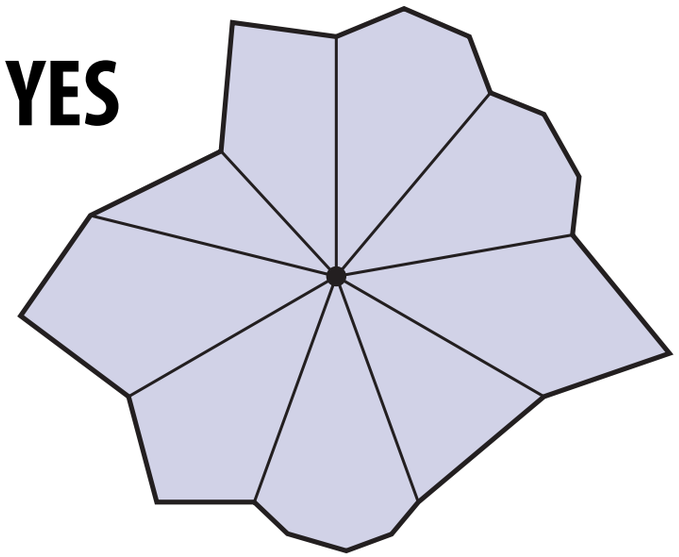
From Wednesday: A manifold polygon mesh has fans, not fins

- 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"

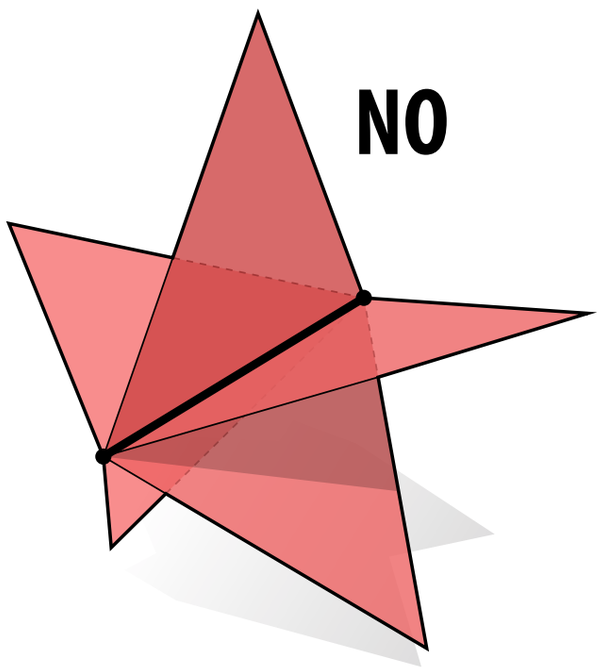
YES



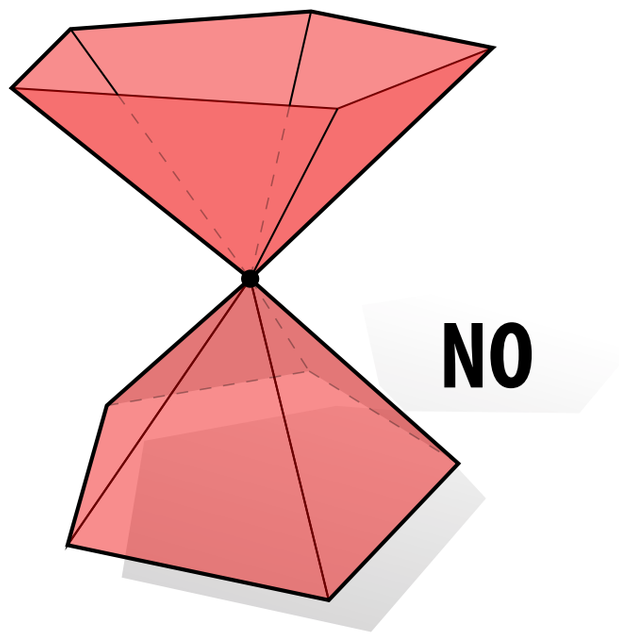
YES



NO



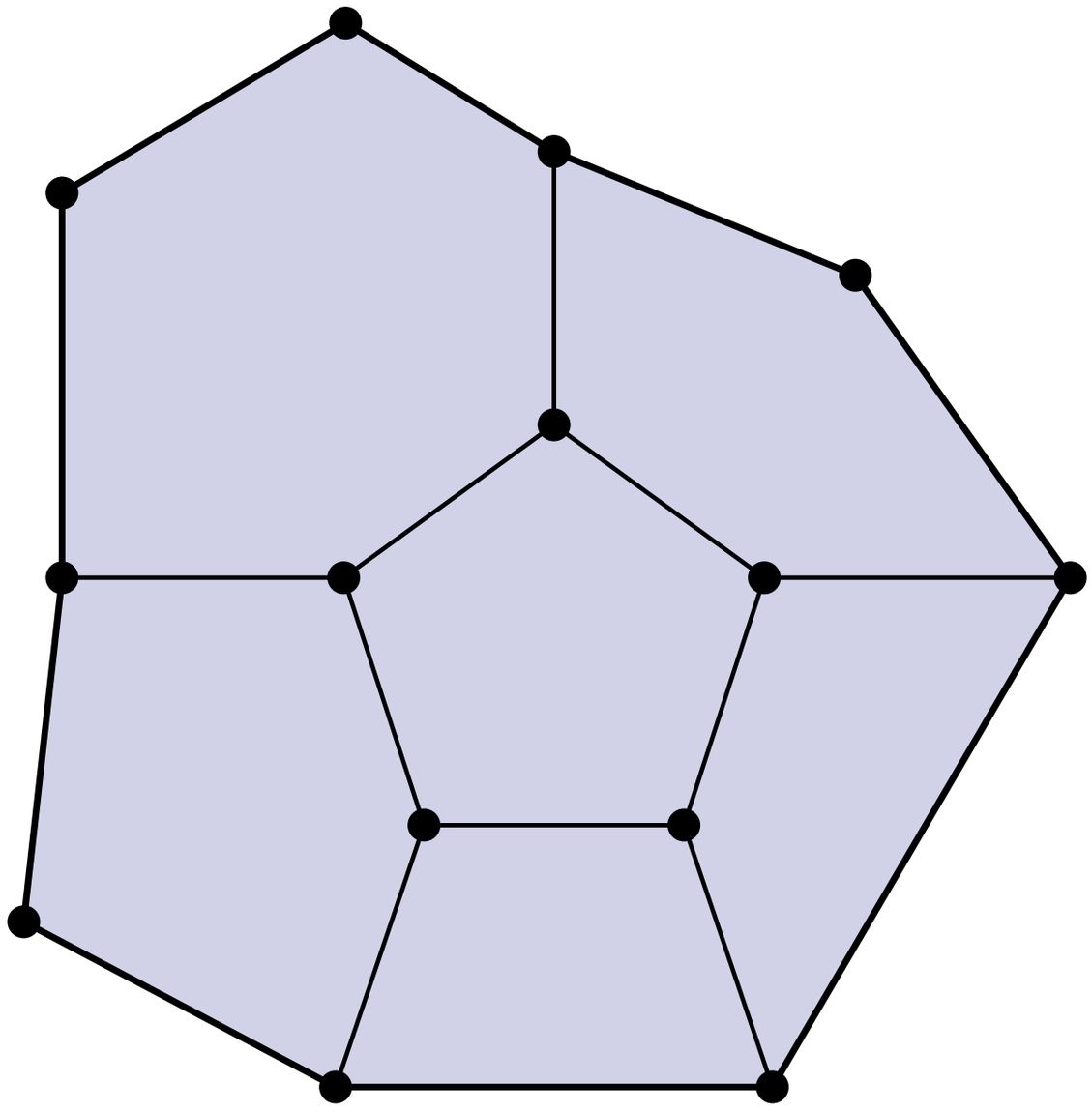
NO



Ok, but why is the manifold assumption useful?

Keep it Simple!

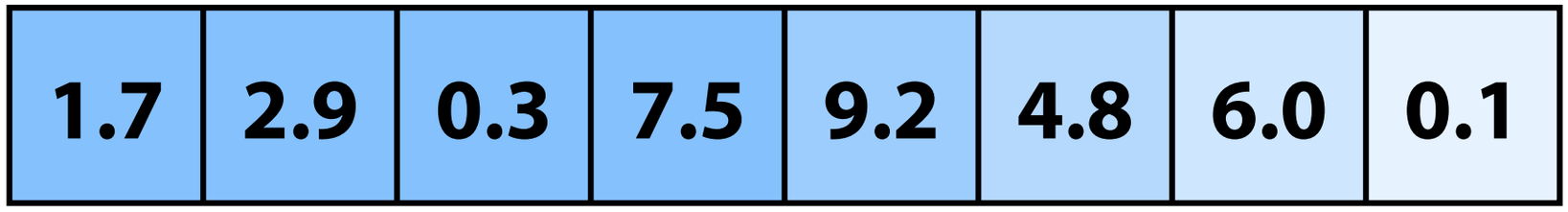
- make some assumptions about our geometry to keep data structures/algorithms simple and efficient
- in many common cases, doesn't fundamentally limit what we can do with geometry



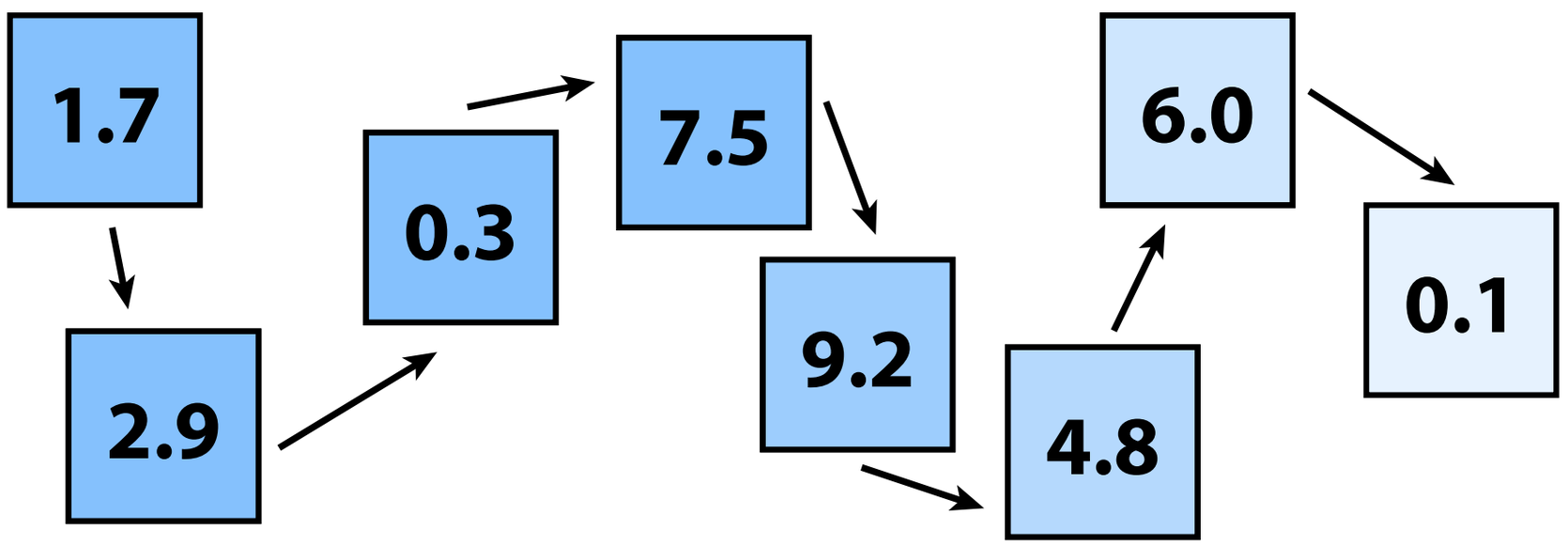
Let's talk about encoding this data

Warm up: two ways to store numbers

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



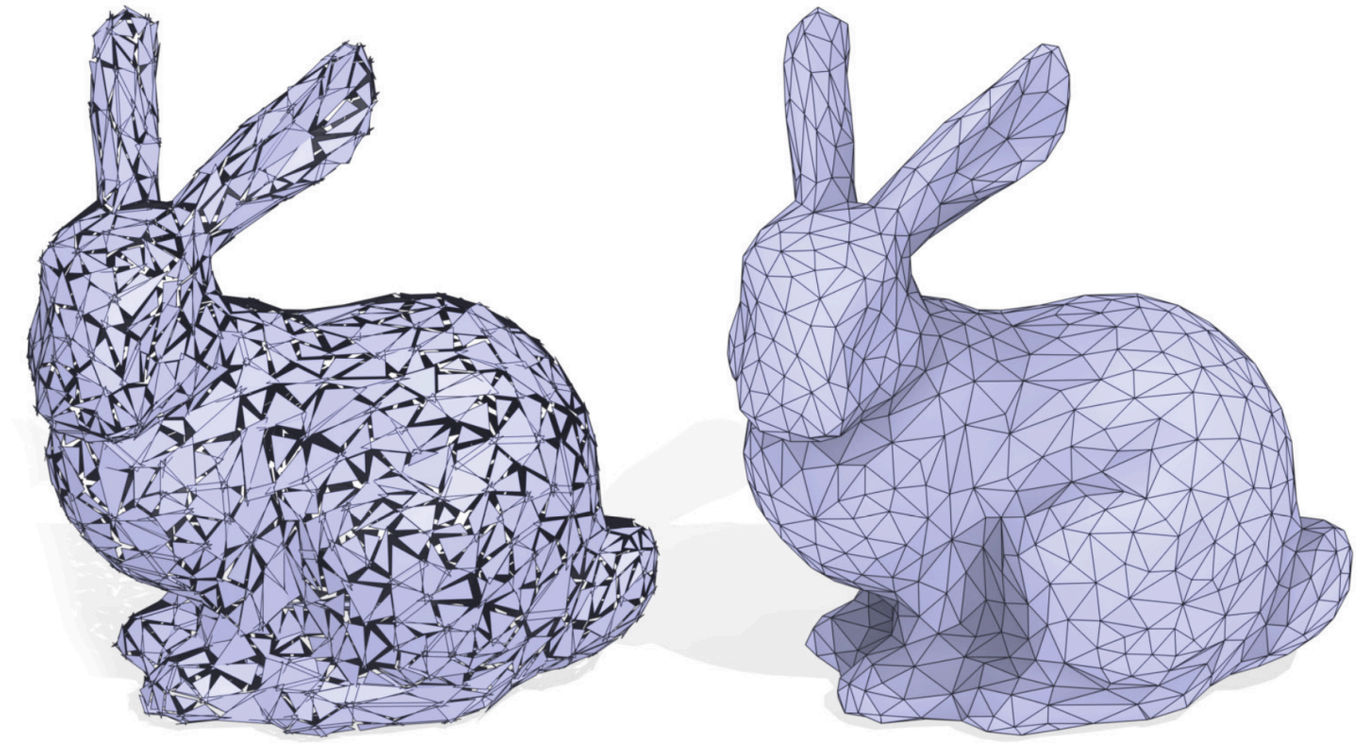
- Alternative: use a linked list (linear lookup, incoherent access)

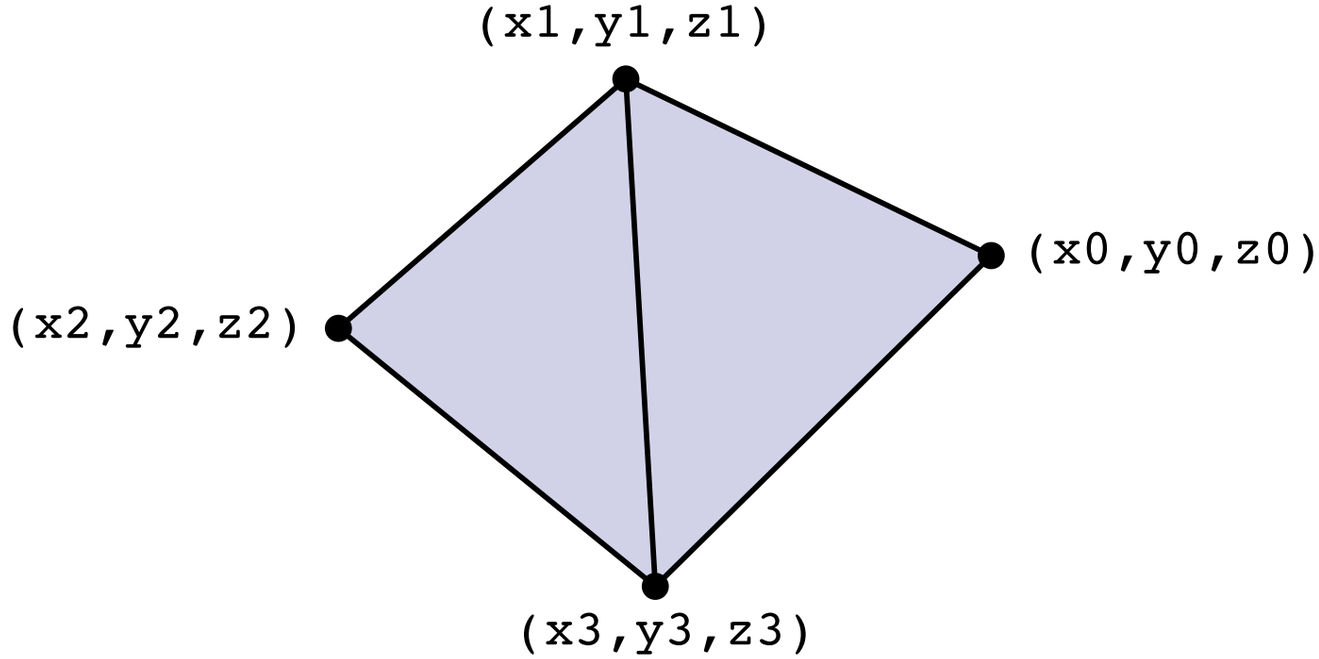


- Q: Why bother with the linked list?
- A: For one, we can easily insert numbers wherever we like…

Polygon Soup

- Most basic idea:
  - For each triangle, just store three coordinates
  - No other information about connectivity
  - Not much different from point cloud! (“Triangle cloud?”)
- Pros:
  - Really stupidly simple
- Cons:
  - Redundant storage
  - Hard to do much beyond simply drawing the mesh on screen
  - Need spatial data structures (later) to find neighbors






```
x0,y0,z0   x1,y1,z1   x3,y3,z3
x1,y1,z1   x2,y2,z2   x3,y3,z3
```

Adjacency List (Array-like)

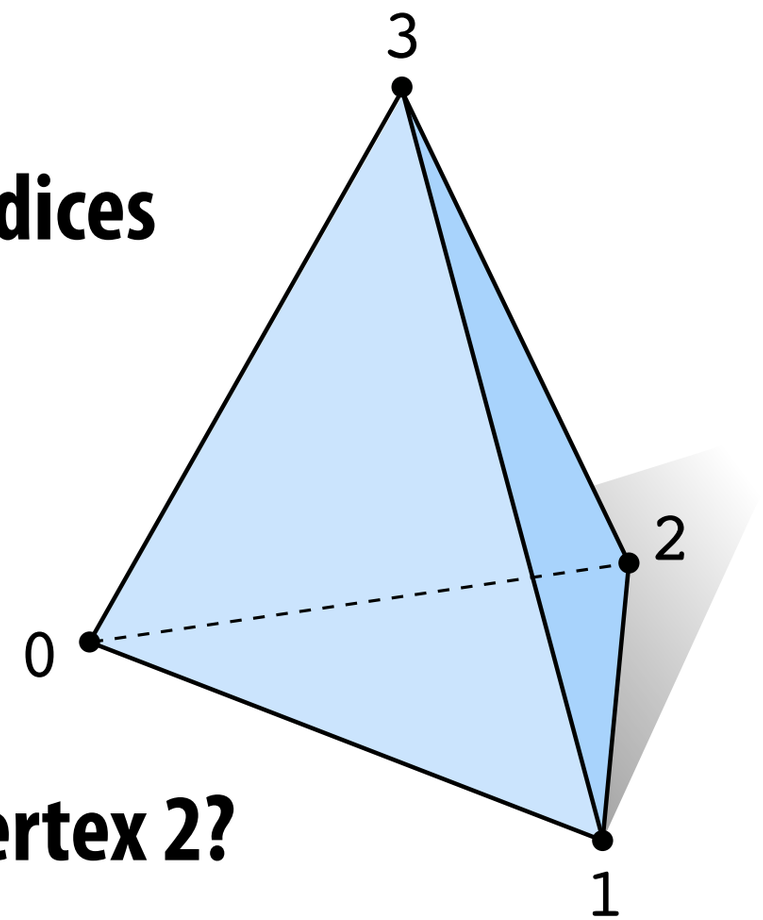
- Store triples of coordinates (x,y,z), tuples of indices
- E.g., tetrahedron:

VERTICES

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

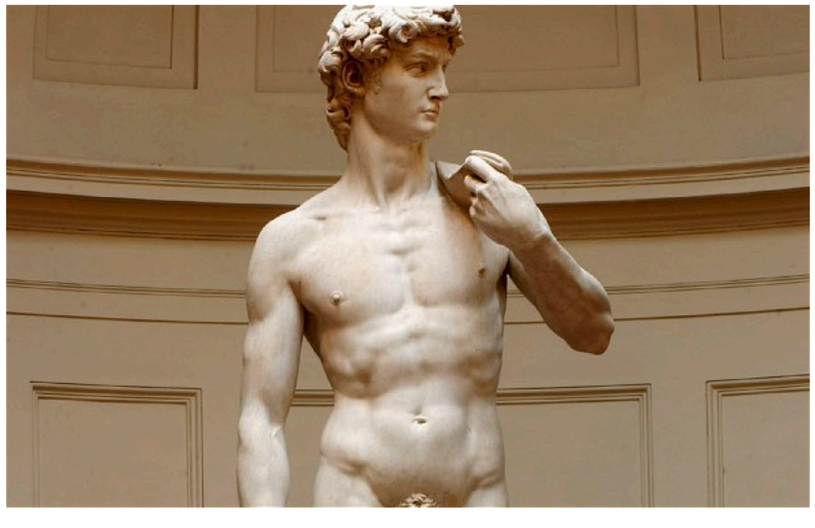
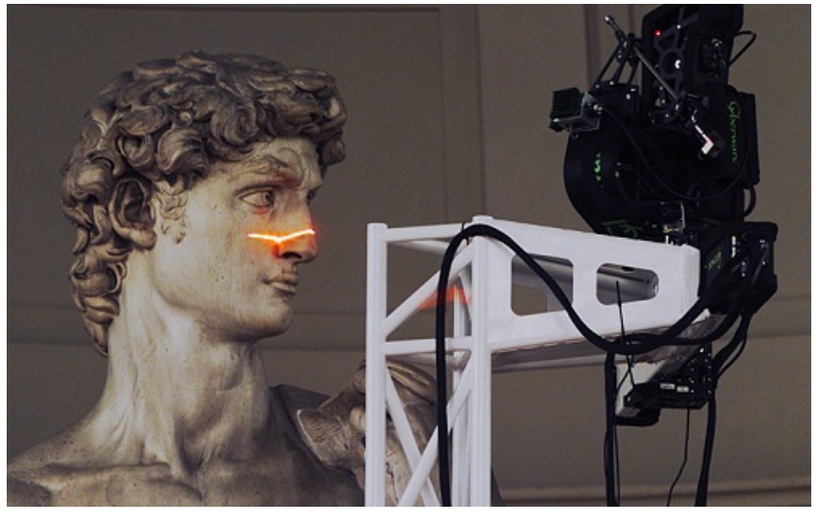
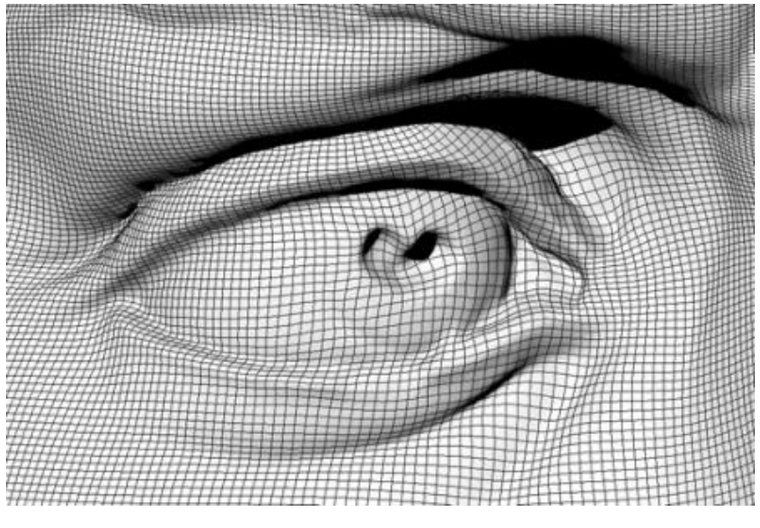
POLYGONS

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 polygons! (What's the cost?)

Incidence Matrices

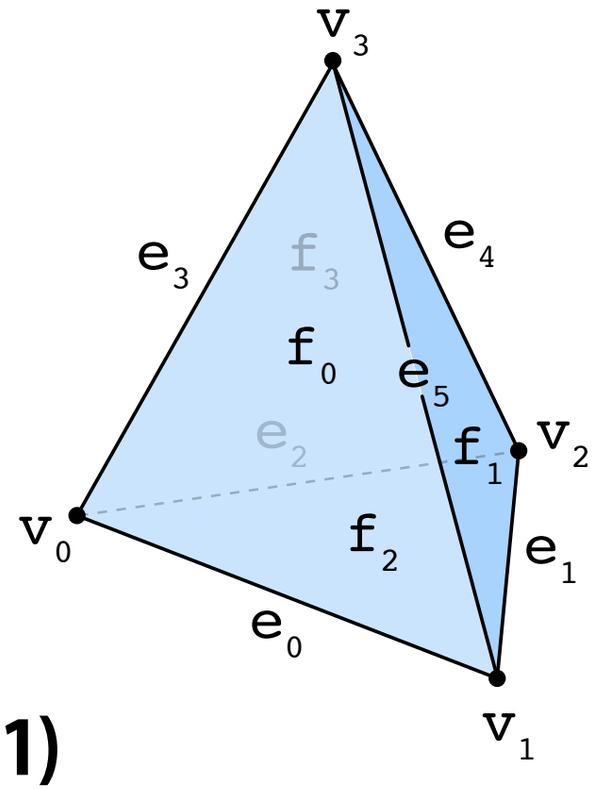
- If we want to know who our neighbors are, why not just store a list of neighbors?
- Can encode all neighbor information via incidence matrices
- E.g., tetrahedron:

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

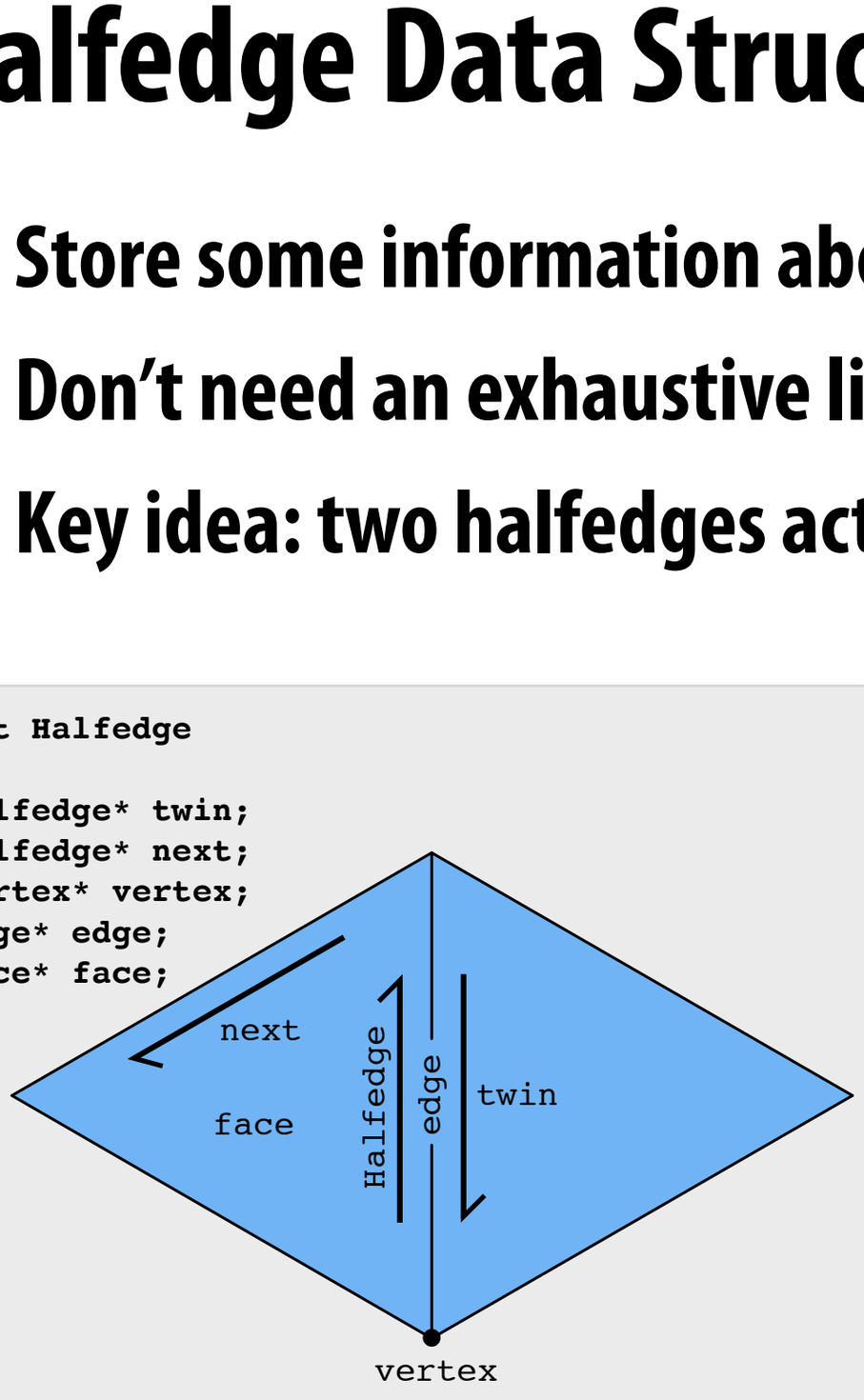


- 1 means "touches"; 0 means "does not touch"
- Instead of storing lots of 0's, use sparse matrices
- Still large storage cost, but finding neighbors is now O(1)
- Hard to change connectivity, since we used fixed indices
- Bonus feature: mesh does not have to be manifold

Halfedge Data Structure (Linked-list-like)

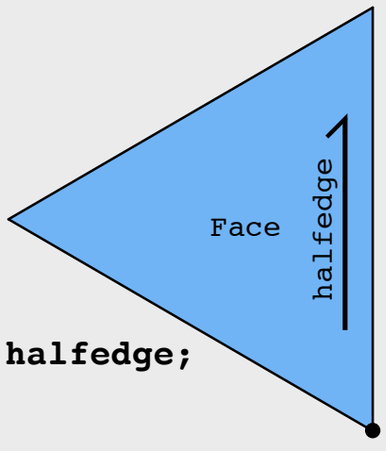
- Store some information about neighbors
- Don't need an exhaustive list; just a few key pointers
- Key idea: two halfedges act as "glue" between mesh elements:

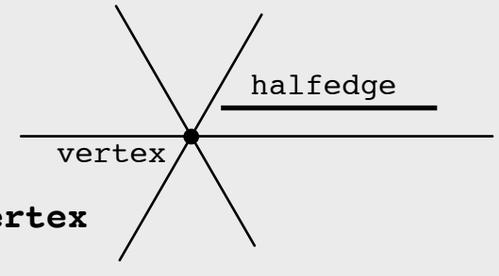
```
struct Halfedge
{
   Halfedge* twin;
   Halfedge* next;
   Vertex* vertex;
   Edge* edge;
   Face* face;
};
```




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


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





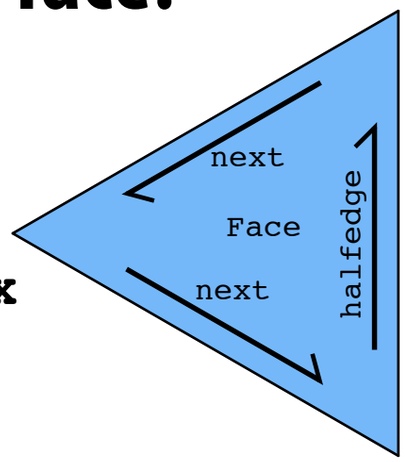

```
struct Vertex
{
   Halfedge* halfedge;
};
```

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

Halfedge makes mesh traversal easy

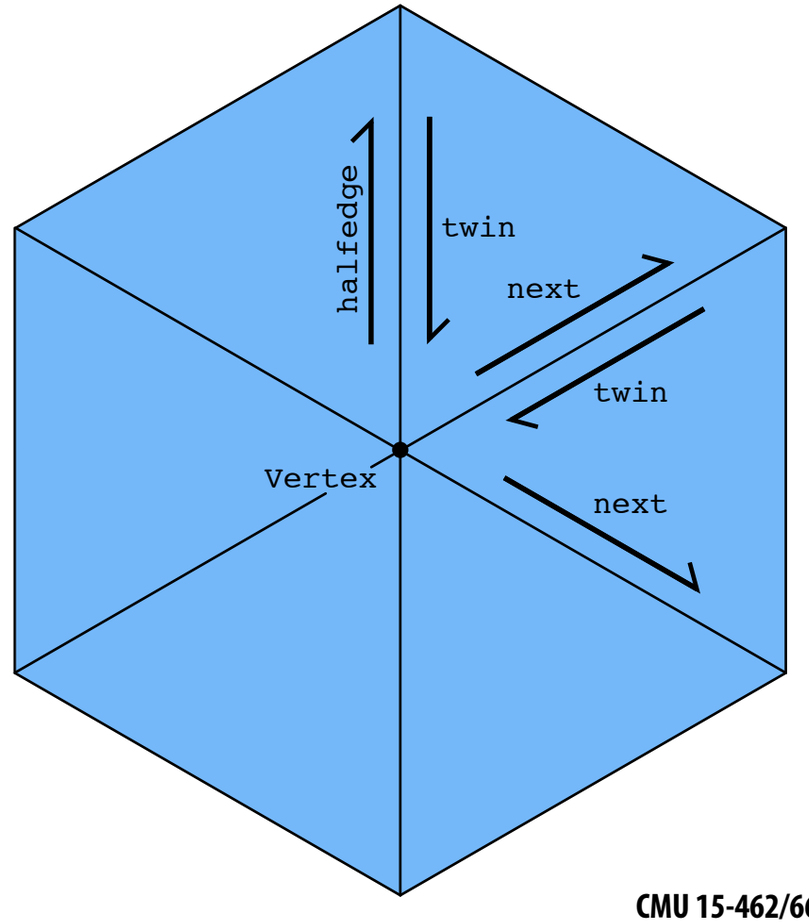
- Use “twin” and “next” pointers to move around mesh
- Use “vertex”, “edge”, and “face” pointers to grab element
- Example: visit all vertices of a face:

```
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 );
```



- Note: only makes sense if mesh is manifold!

Halfedge connectivity is always manifold

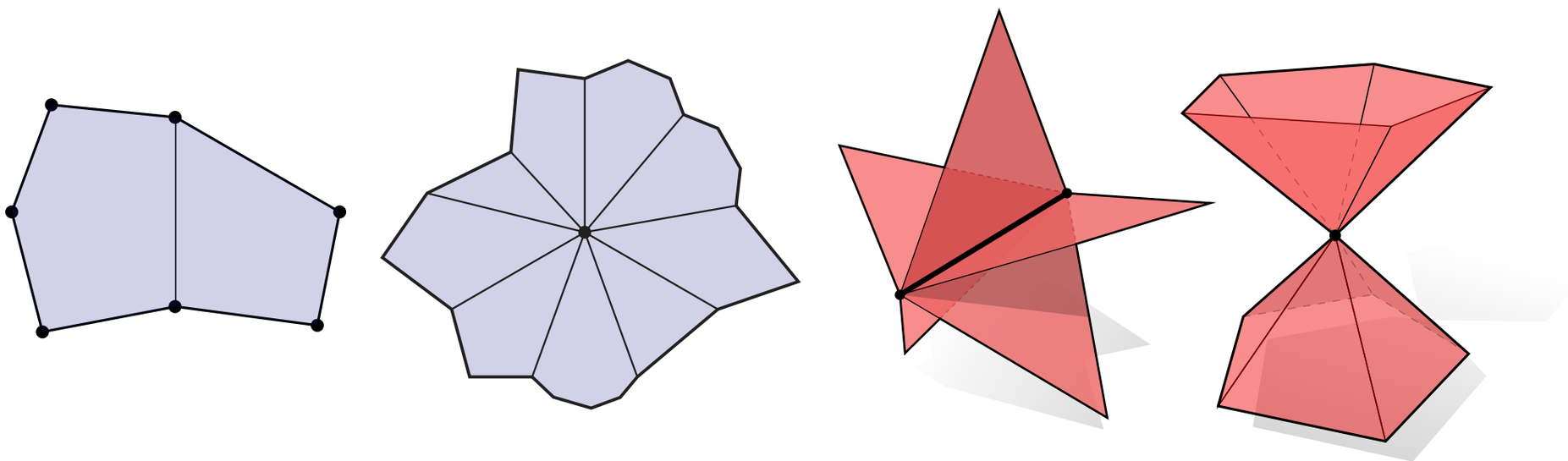
- Consider simplified halfedge data structure
- Require only "common-sense" conditions

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

(pointer to yourself!)

```
twin->twin == this
twin != this
every he is someone's "next"
```

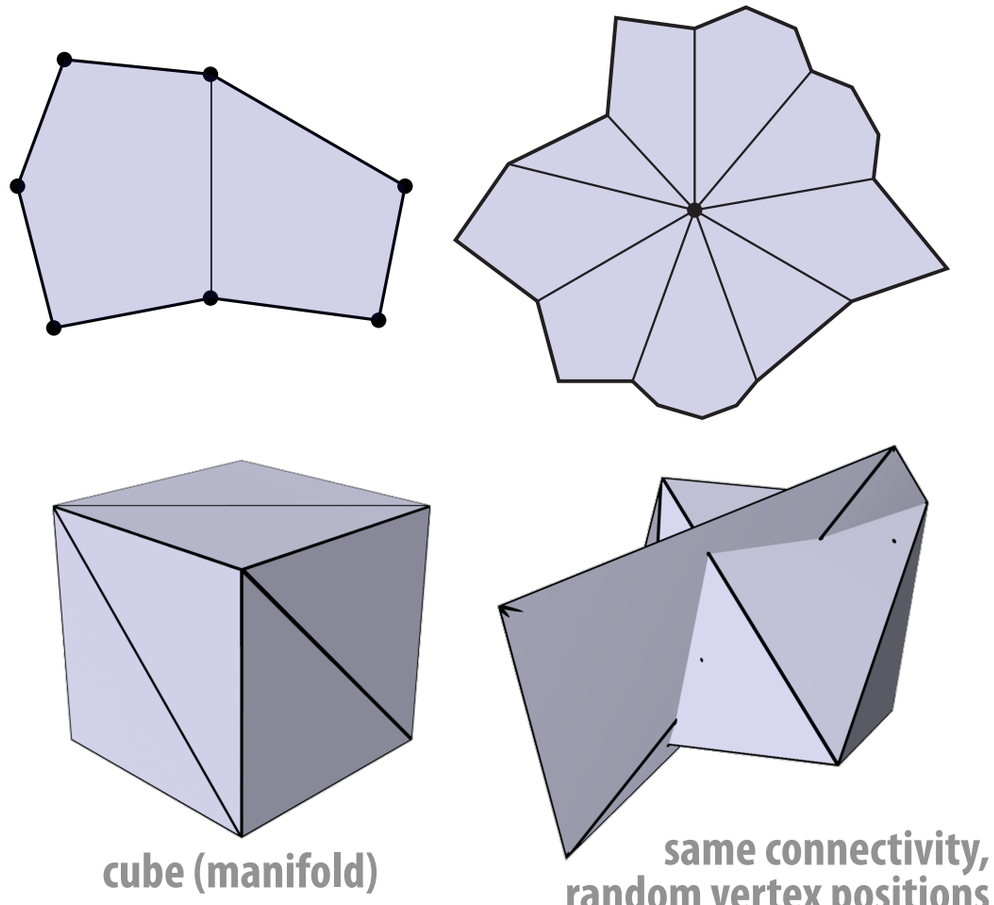
- Keep following `next`, and you'll get faces.
- Keep following `twin` and you'll get edges.
- Keep following `next->twin` and you'll get vertices.



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

Connectivity vs. Geometry

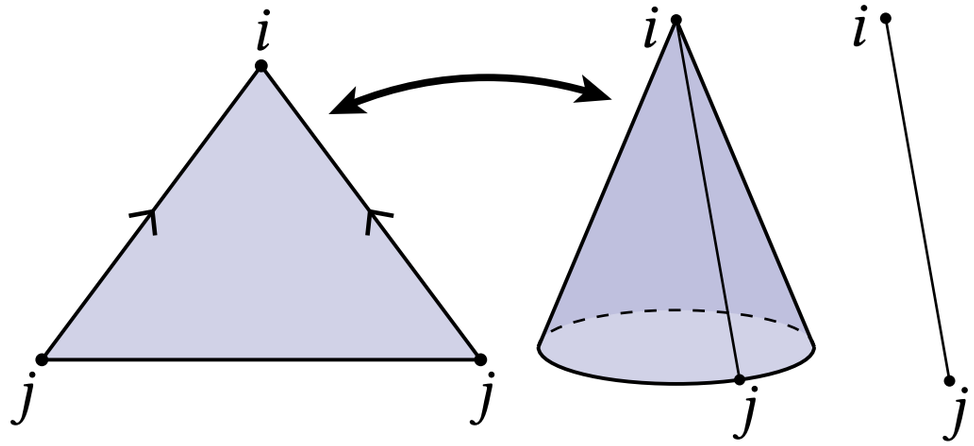
- Recall manifold conditions (fans not fins):
  - every edge contained in two faces
  - every vertex contained in one fan
- These conditions say <u>nothing</u> about vertex positions! Just connectivity
- Hence, can have perfectly good (manifold) connectivity, even if geometry is awful

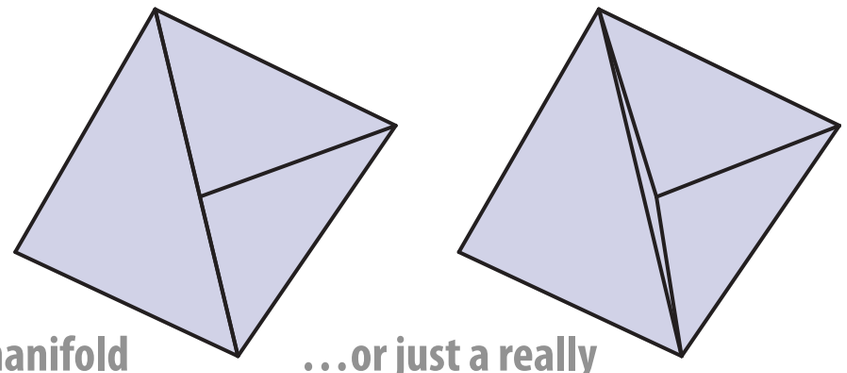


cube (manifold)

same connectivity, random vertex positions

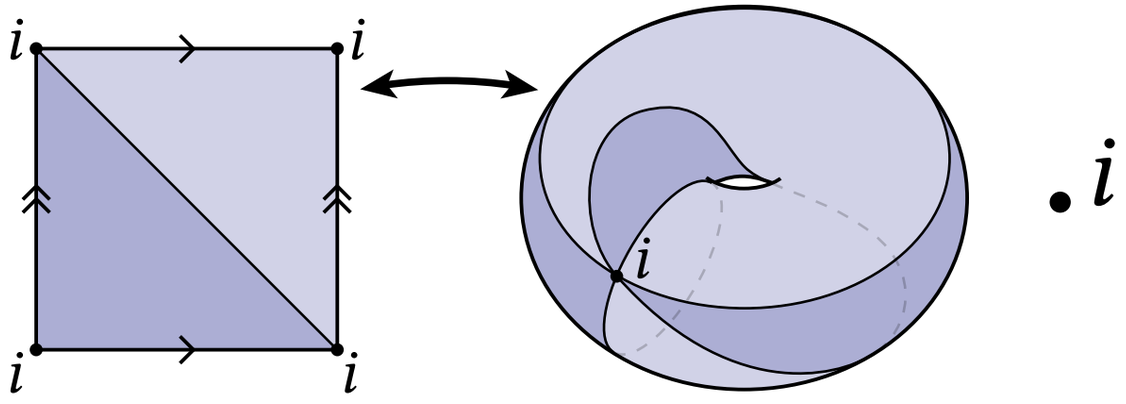
- In fact, sometimes you can have perfectly good manifold connectivity for which <u>any</u> vertex positions give "bad" geometry!
- Can lead to confusion when debugging: mesh looks "bad", even though connectivity is fine





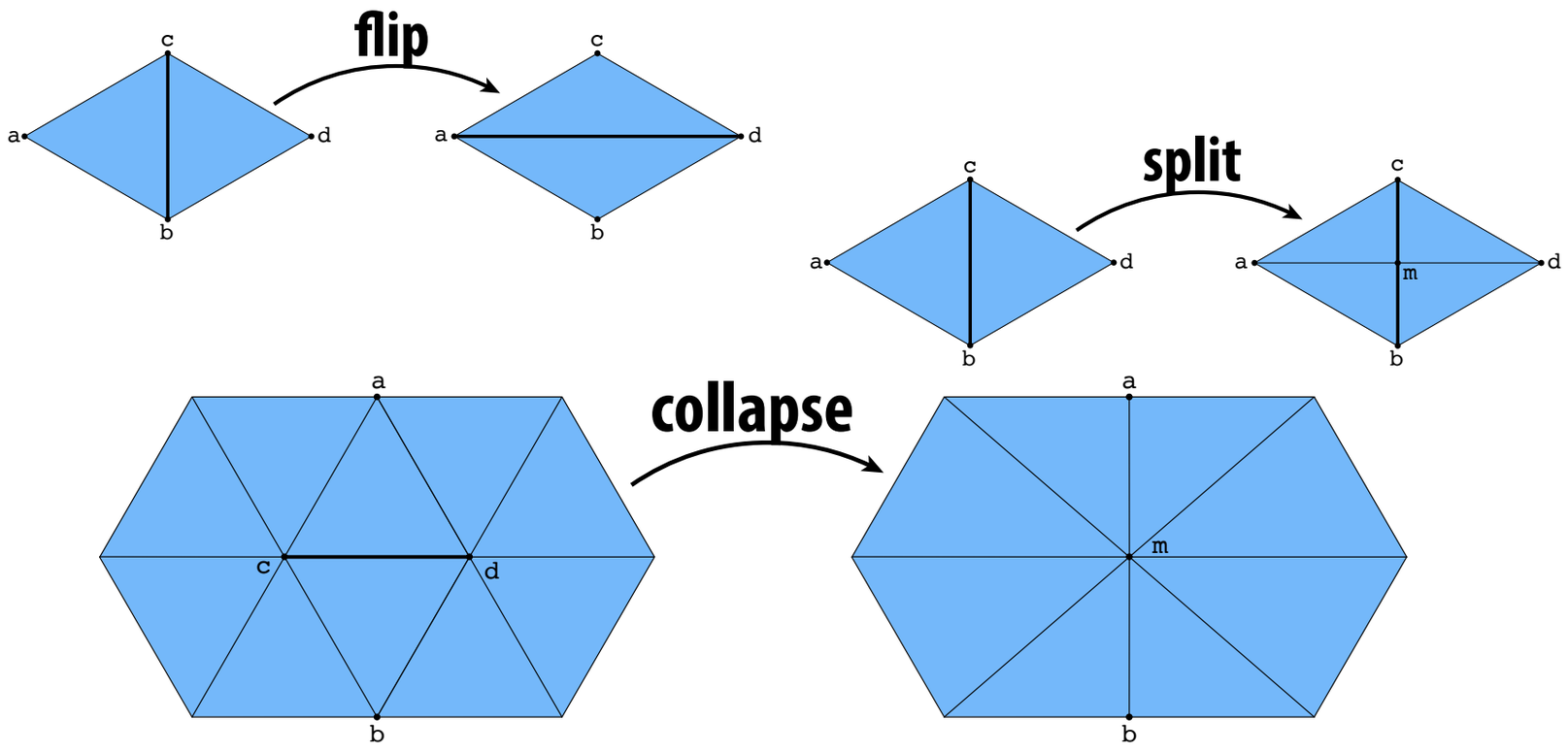
non manifold connectivity?

…or just a really skinny triangle?



Halfedge meshes are easy to edit

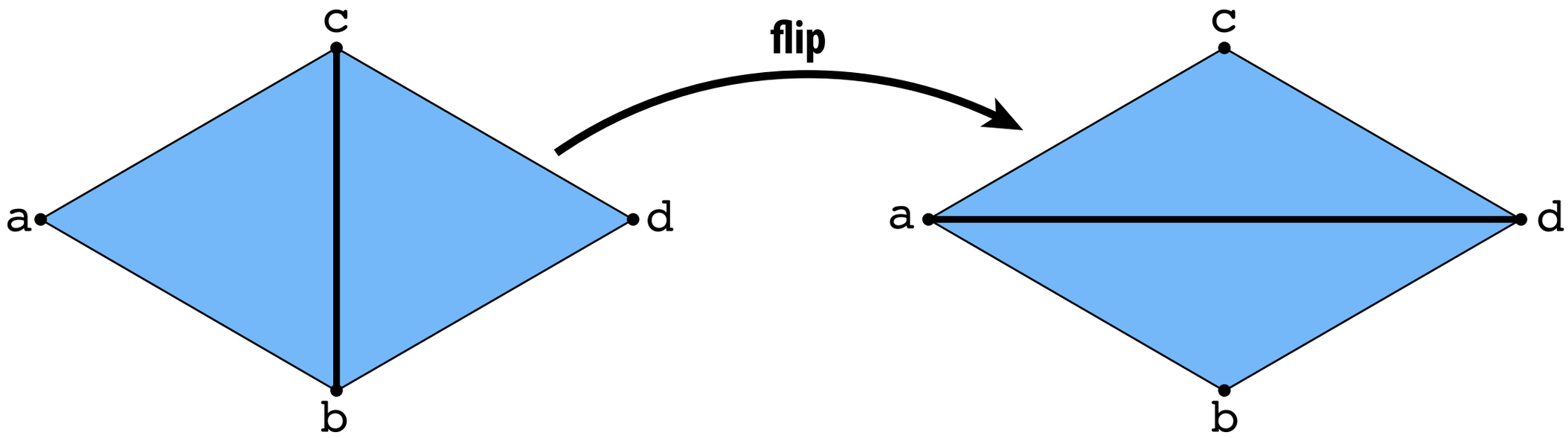
- Remember key feature of linked list: insert/delete elements
- Same story with halfedge mesh ("linked list on steroids")
- E.g., for triangle meshes, several atomic operations:



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

Edge Flip (Triangles)

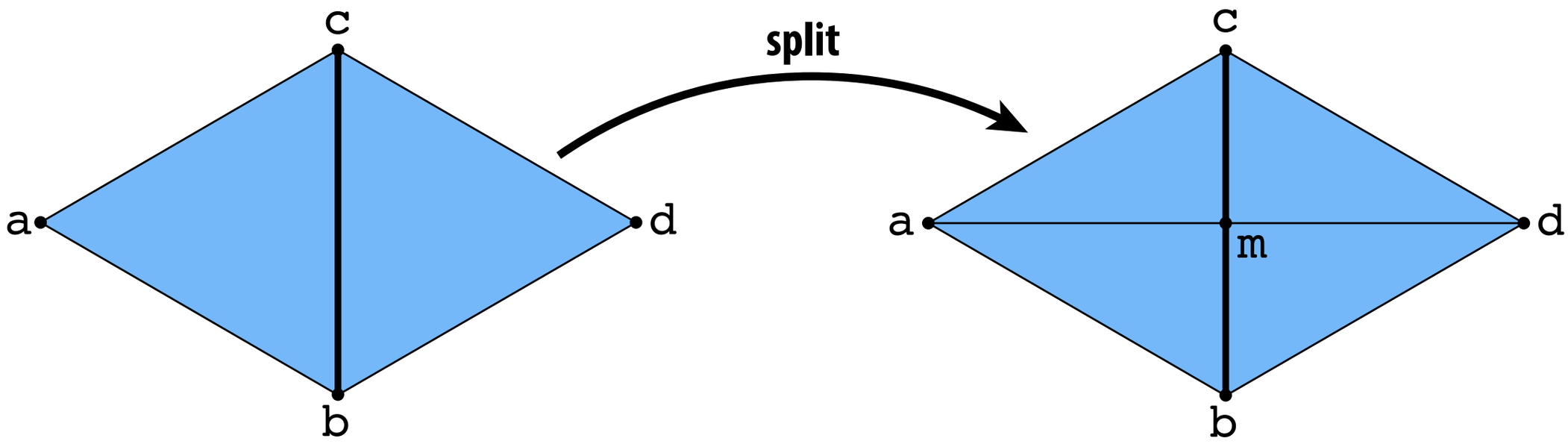
- Triangles (a,b,c), (b,d,c) become (a,d,c), (a,b,d):



- 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)

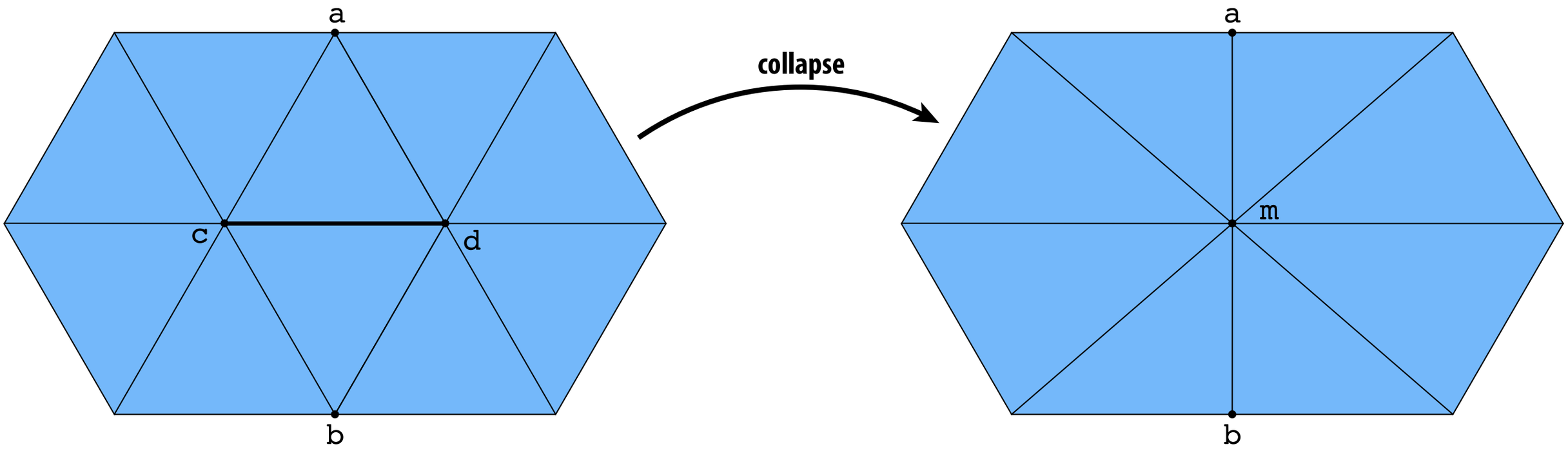
- Insert midpoint m of edge (c,b), connect to get four triangles:



- 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:

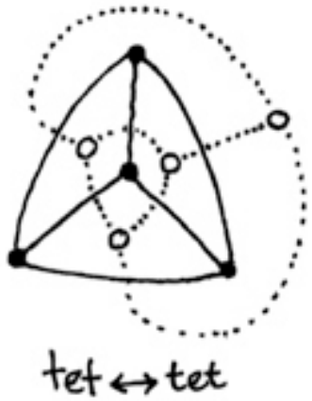


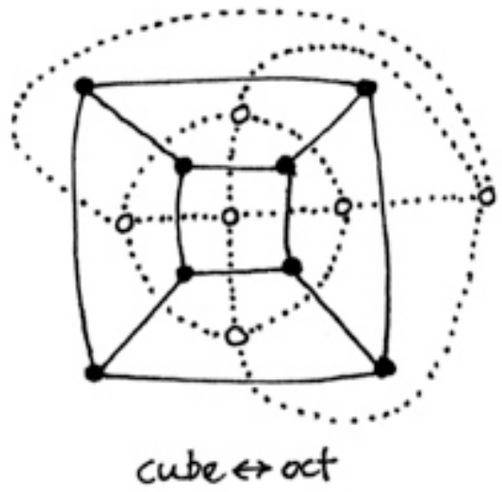
- Now have to delete elements.
- Still lots of pointer assignments!
- Q: How would we implement this with an adjacency list?
- Any other good way to do it? (E.g., different data structure?)

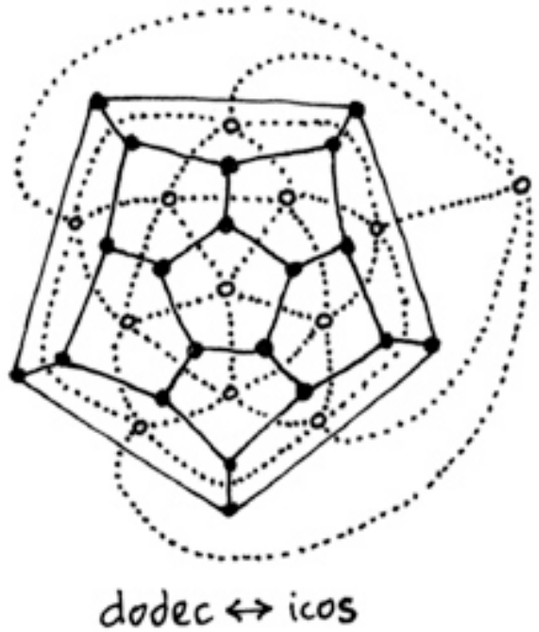
Alternatives to Halfedge

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
- With some thought*, can design halfedge-type data structures with coherent data storage, support for non manifold connectivity, etc.

*see for instance http://geometry-central.net/

Comparison of Polygon Mesh Data Strucutres

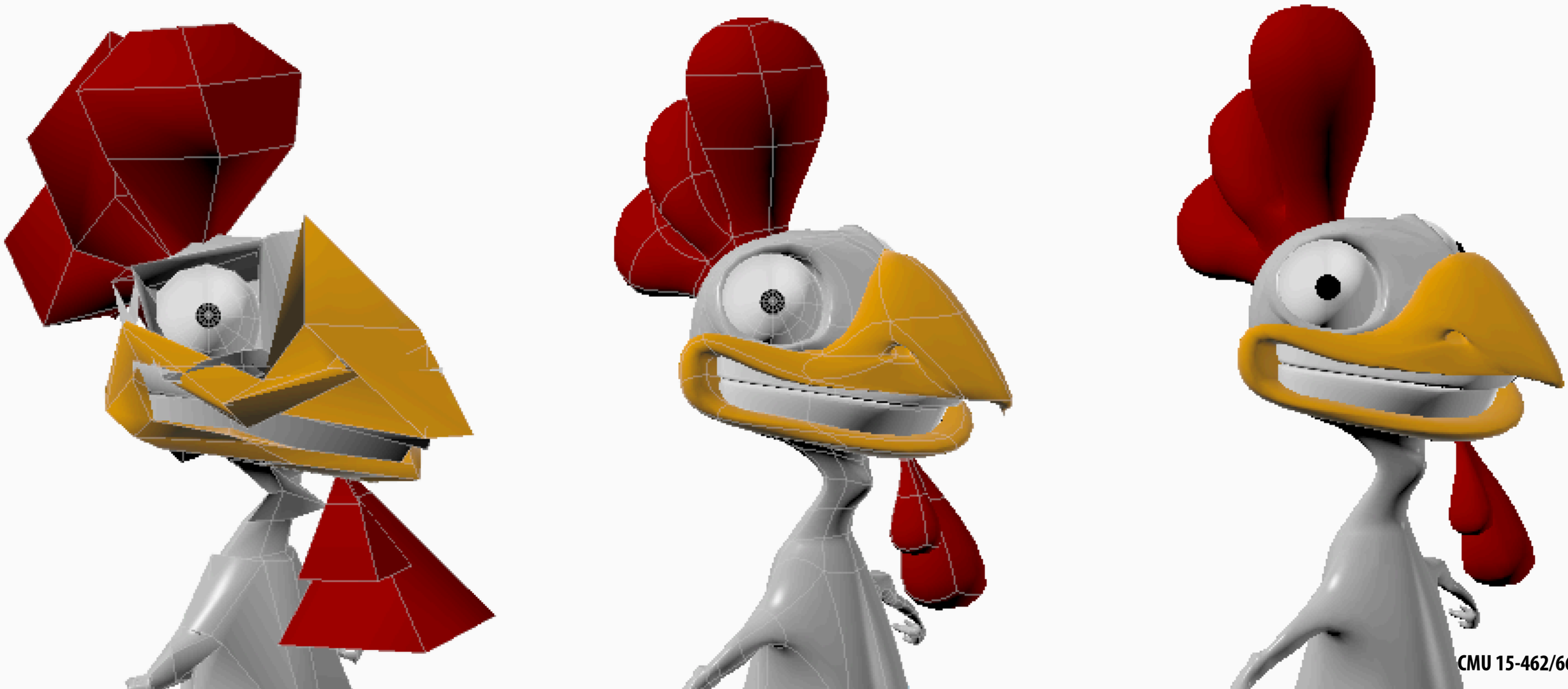
	Adjacency List	Incidence Matrices	Halfedge Mesh
constant-time neighborhood access?	NO	YES	YES
easy to add/remove mesh elements?	NO	NO	YES
nonmanifold geometry?	YES	YES	NO

Conclusion: pick the right data structure for the job!

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

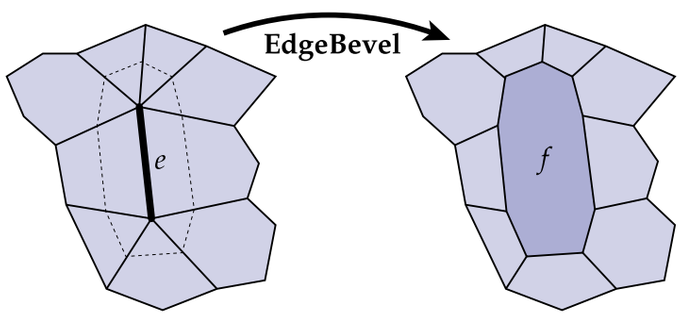
Subdivision Modeling

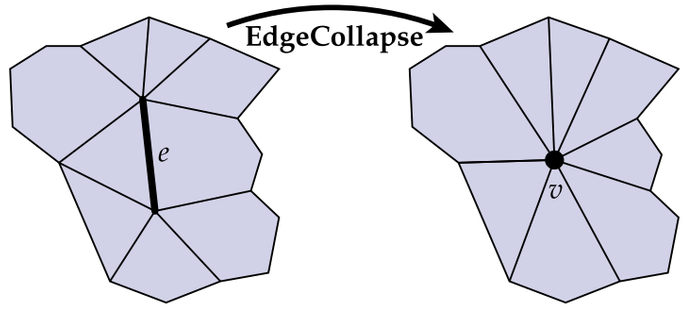
- Common modeling paradigm in modern 3D tools:
  - Coarse "control cage"
  - Perform local operations to control/edit shape
  - Global subdivision process determines final surface

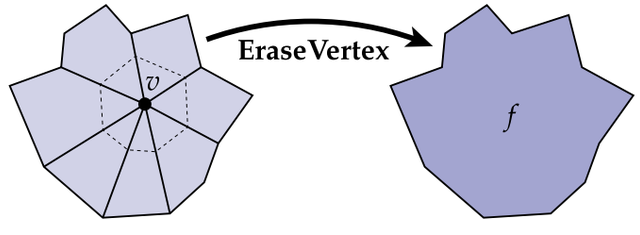


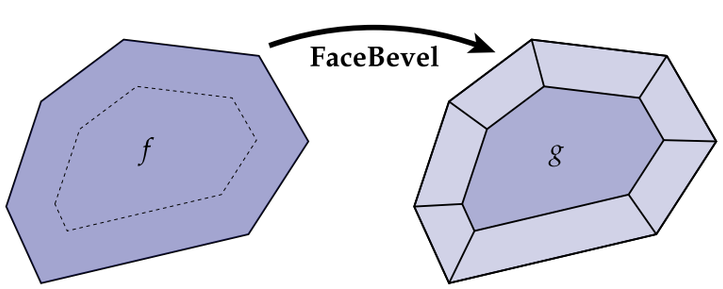
Subdivision Modeling—Local Operations

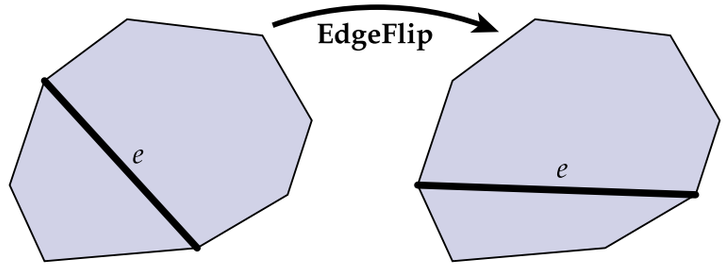
- **For general polygon meshes, we can dream up lots of local mesh operations that might be useful for modeling:**

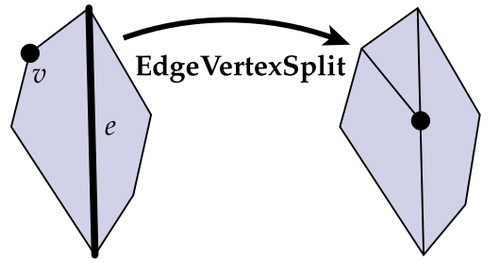


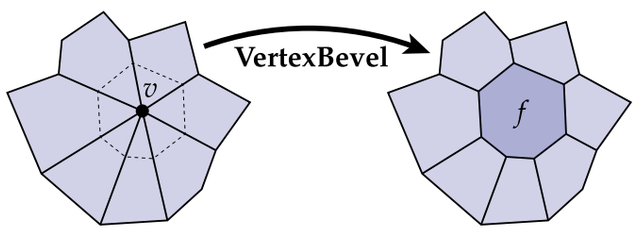


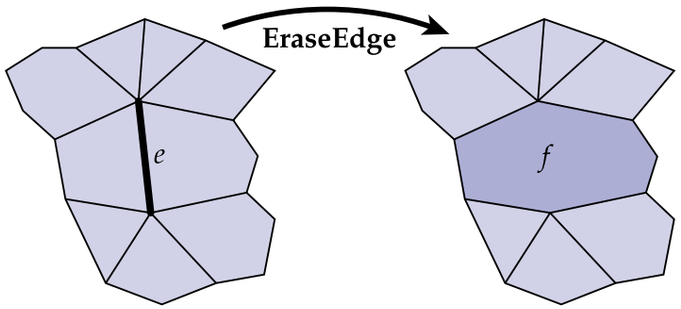


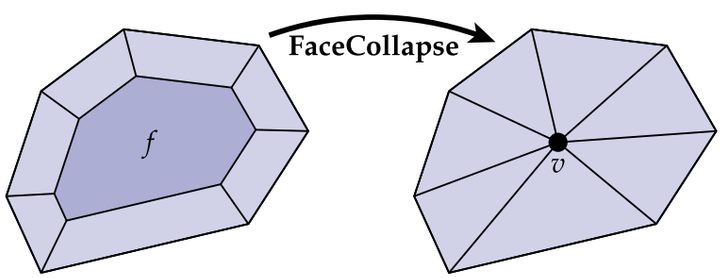






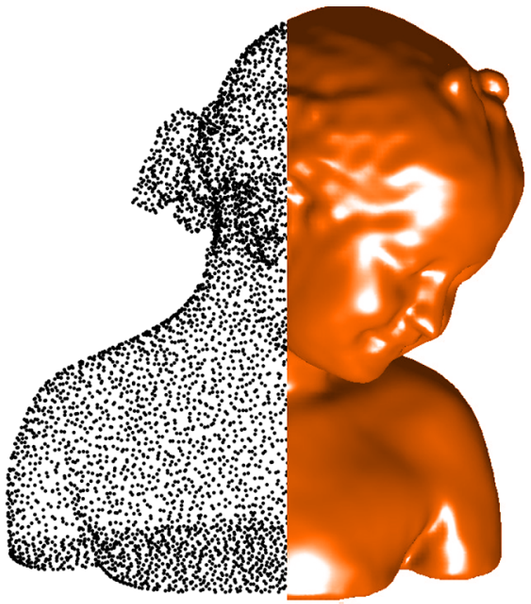






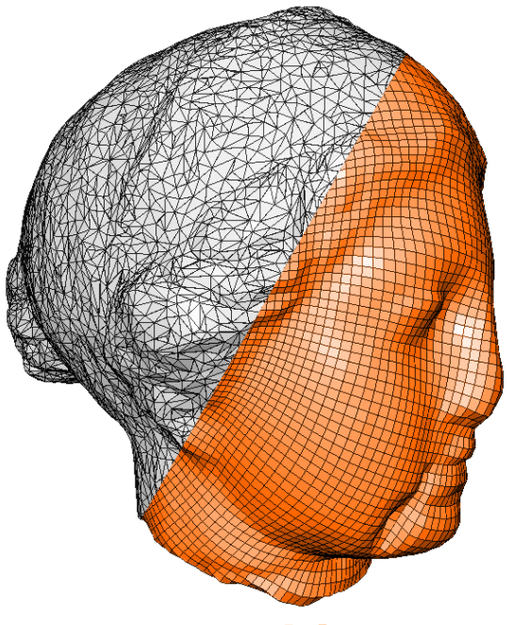
…and many, many more!

Geometry Processing

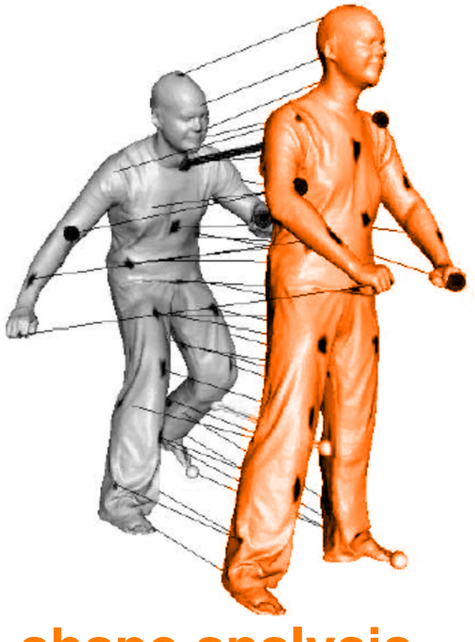


reconstruction

filtering

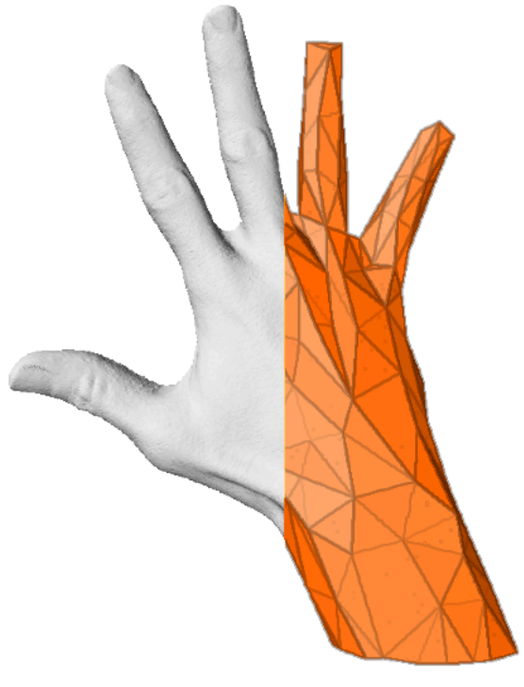


remeshing



shape analysis

parameterization



compression

Next time…

- **Wednesday: GOOD LUCK WITH YOUR MIDTERM!**

- **Monday:**
  - **Midterm review**
  - **More on subdivision and geometry processing**
  - **Quadric error metric for simplification**