Meshes and Geometry Processing

Computer Graphics
CMU 15-462/15-662

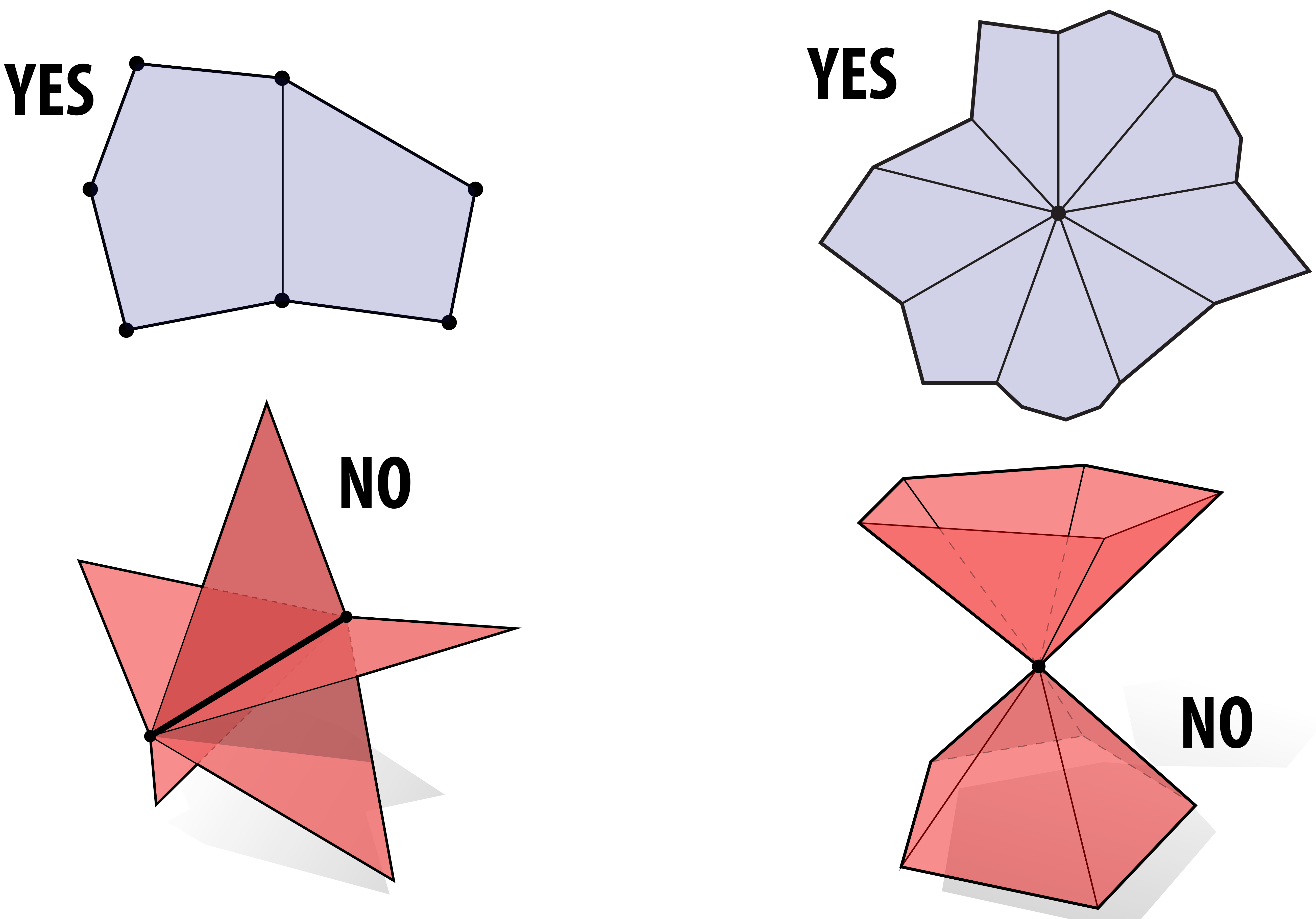
Last time: overview of geometry

- 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
- Introduction to manifold geometry
- Today:
  - nuts & bolts of polygon meshes
  - geometry processing / resampling

Geometry

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

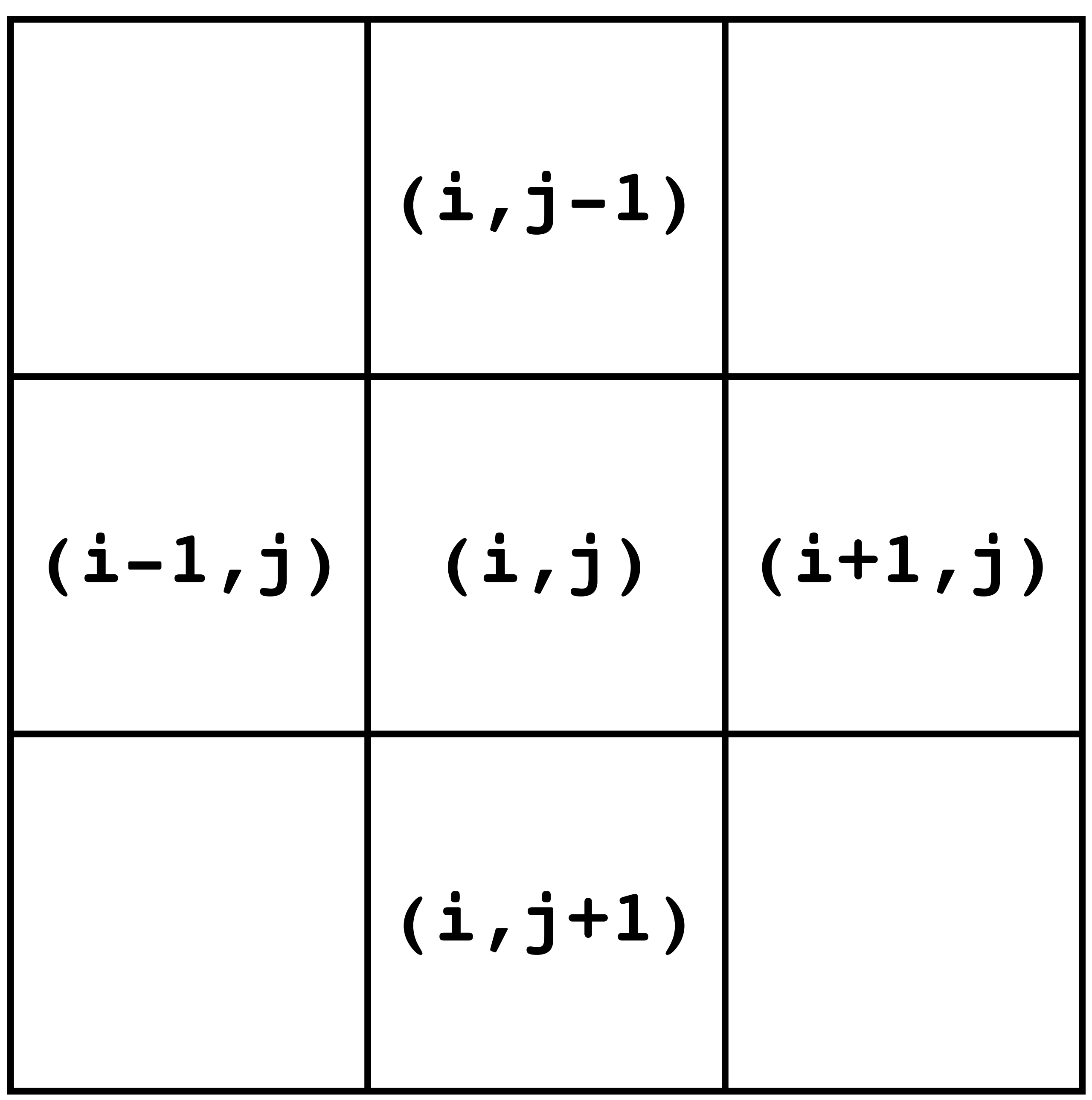
- For polygonal surfaces just two easy conditions to check:
  1. Every edge is contained in only two polygons (no “fins”)
  2. The polygons containing each vertex make a single “fan”

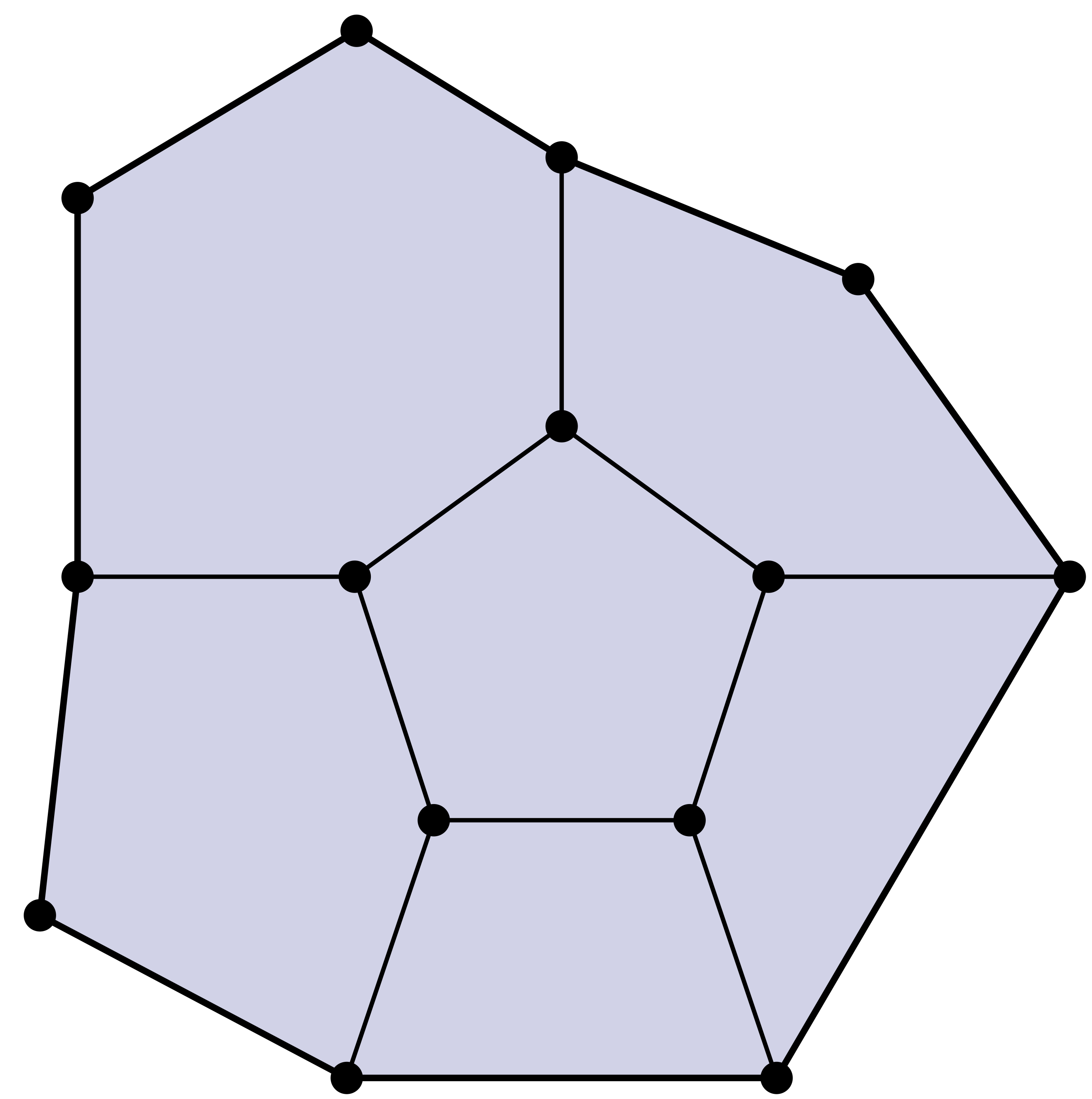


Ok, but why is the manifold assumption useful?

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

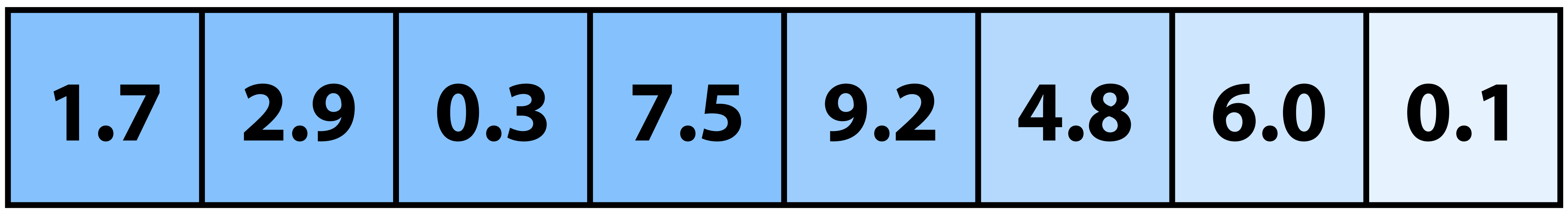




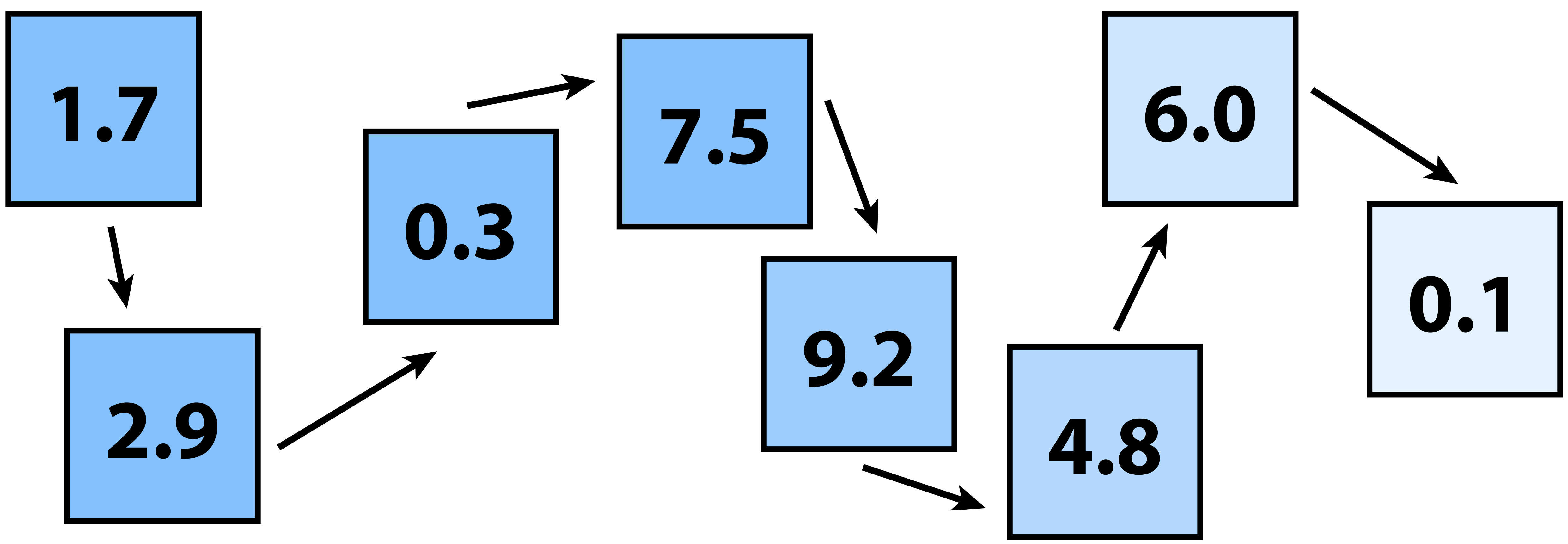
How do we actually encode all this data?

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)



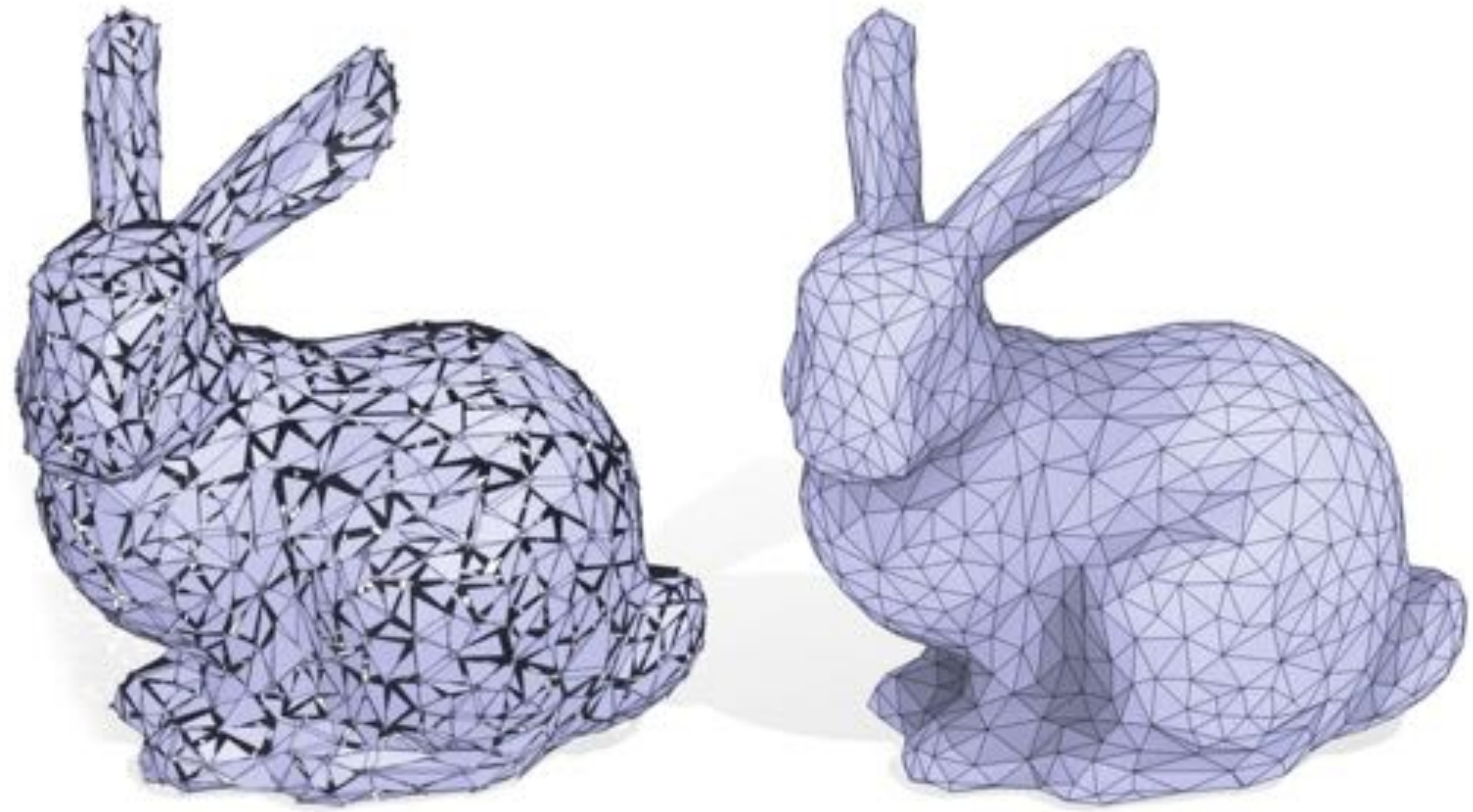
- Alternative: use a linked list (linear lookup, incoherent access)

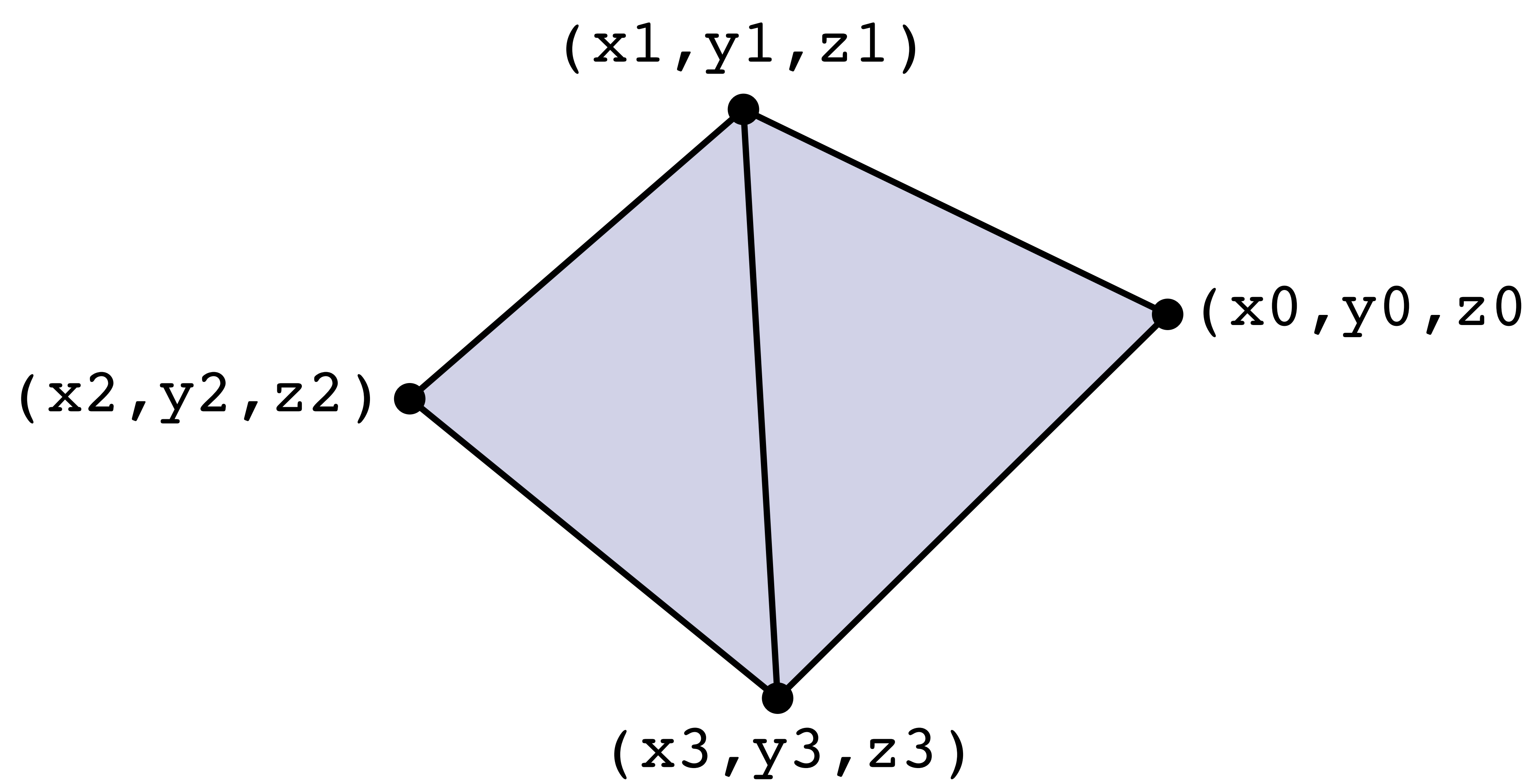


- Q: Why bother with the linked list?
- A: For one, we can easily insert numbers wherever we like...

Polygon Soup

- **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)

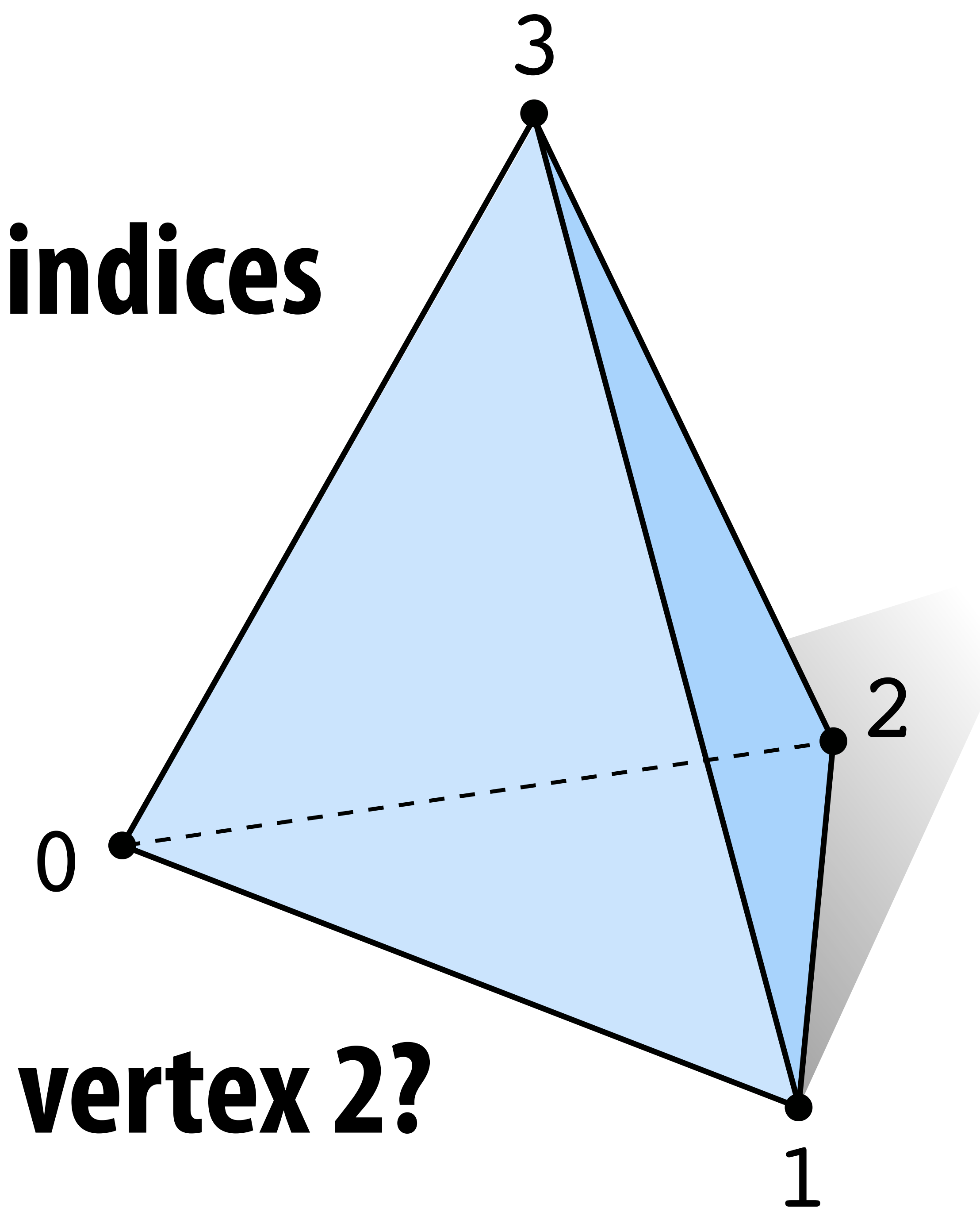
- Store triples of coordinates (x,y,z), tuples of indices
- E.g., tetrahedron:

VERTICES

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

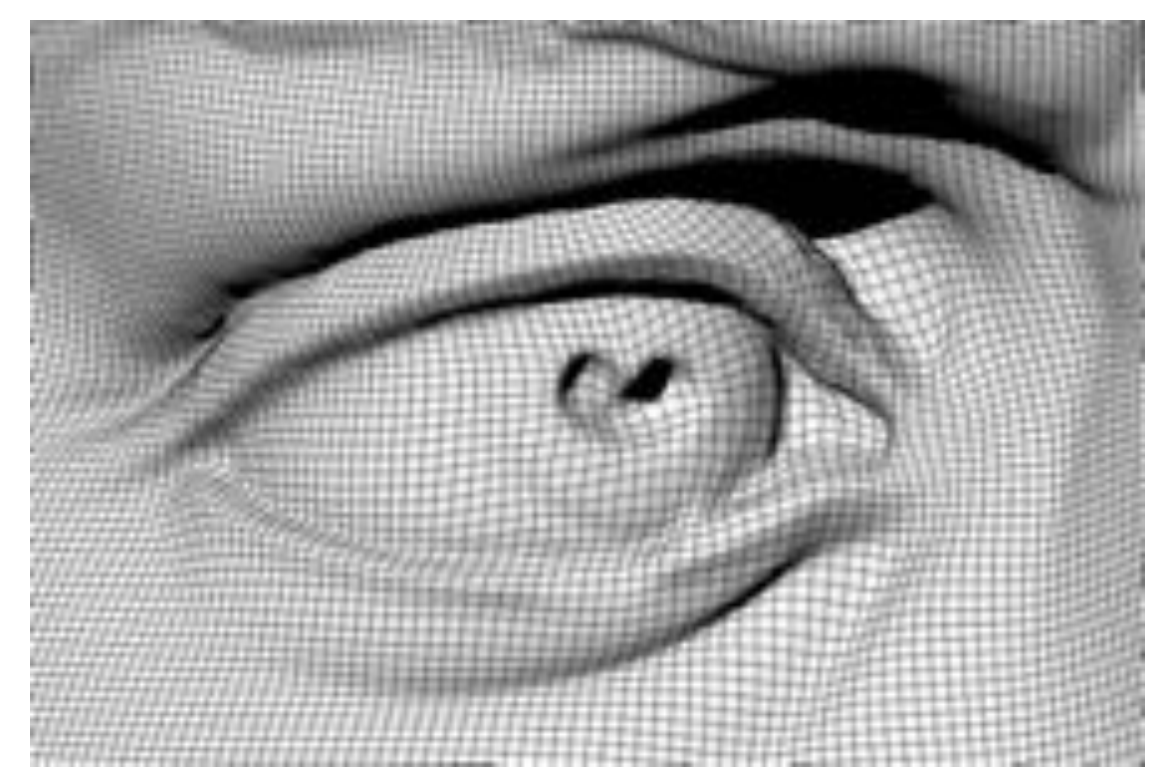
POLYGONS

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

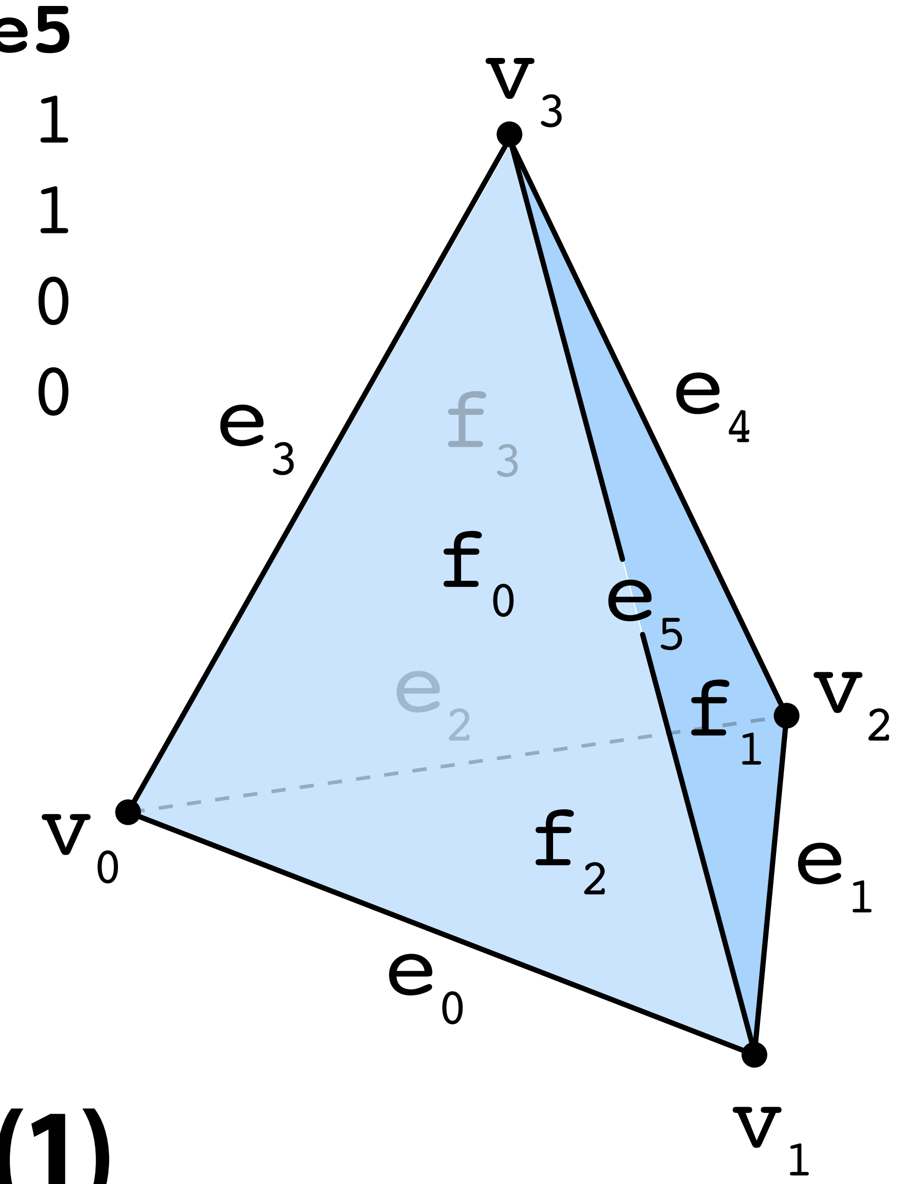
- If we want to know who our neighbors are, why not just store a list of neighbors?
- Can encode all neighbor information via incidence matrices
- E.g., tetrahedron:

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

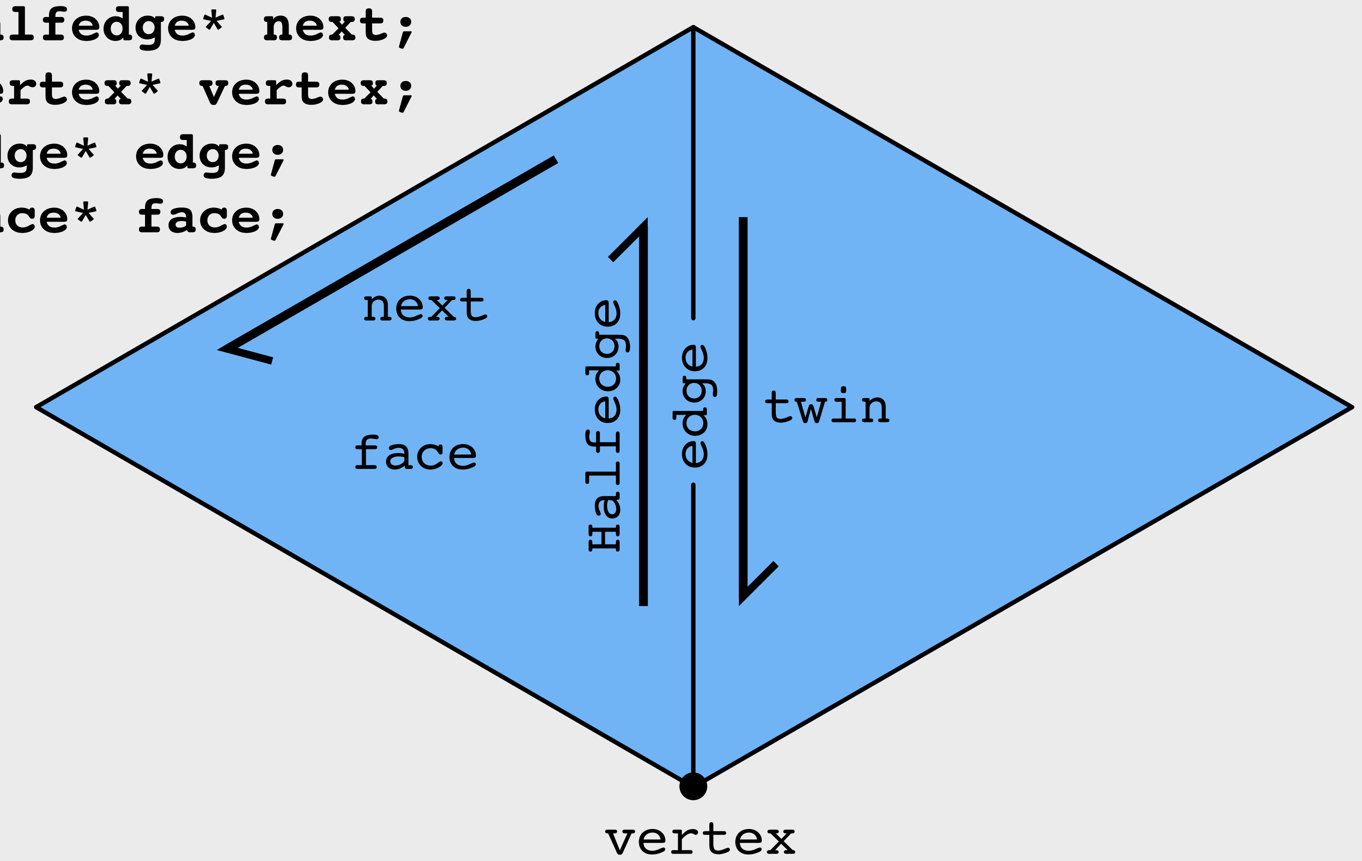


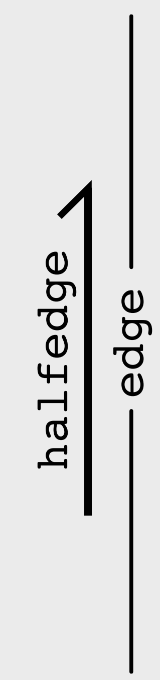
- 1 means "touches"; 0 means "does not touch"
- Instead of storing lots of 0's, use sparse matrices
- Still large storage cost, but finding neighbors is now O(1)
- Hard to change connectivity, since we used fixed indices
- Bonus feature: mesh does not have to be manifold

Halfedge Data Structure (Linked-list-like)

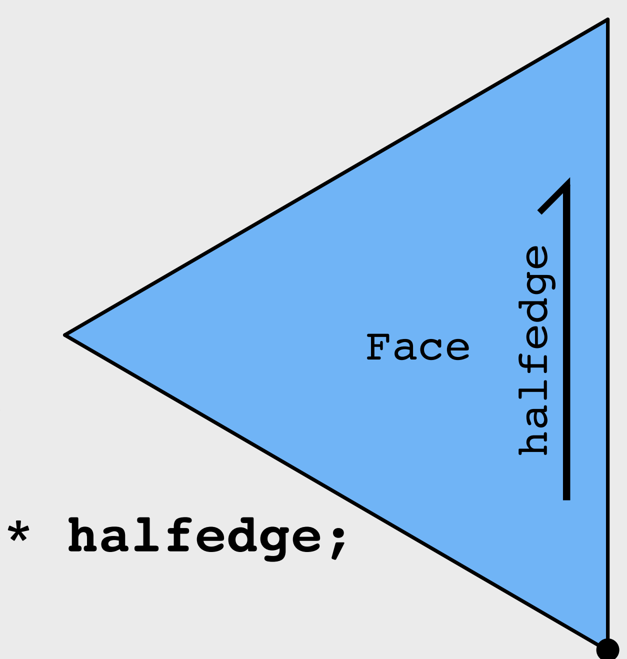
- Store some information about neighbors
- Don’t need an exhaustive list; just a few key pointers
- Key idea: two halfedges act as “glue” between mesh elements:

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

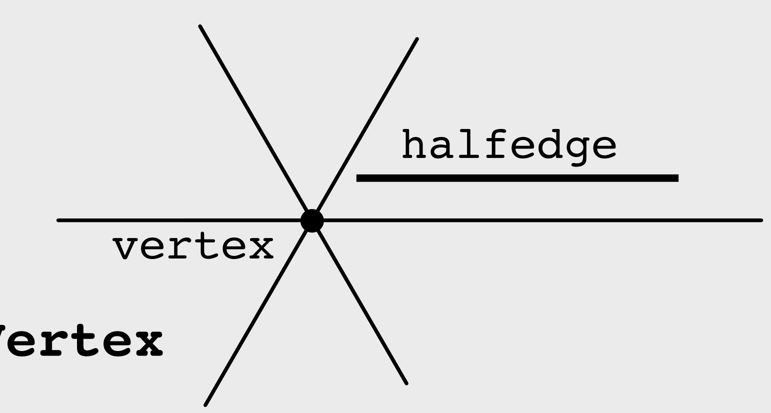





```
struct Edge
{
   Halfedge* halfedge;
};
```




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



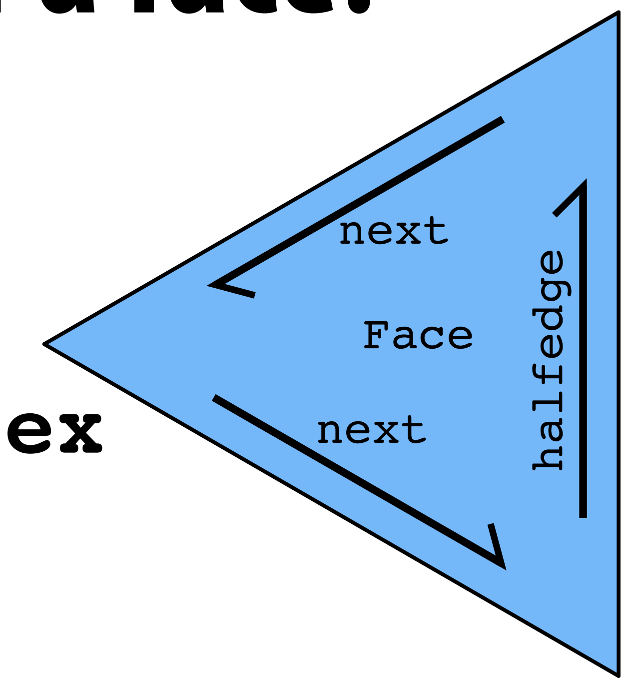

```
struct Vertex
{
   Halfedge* halfedge;
};
```

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

Halfedge makes mesh traversal easy

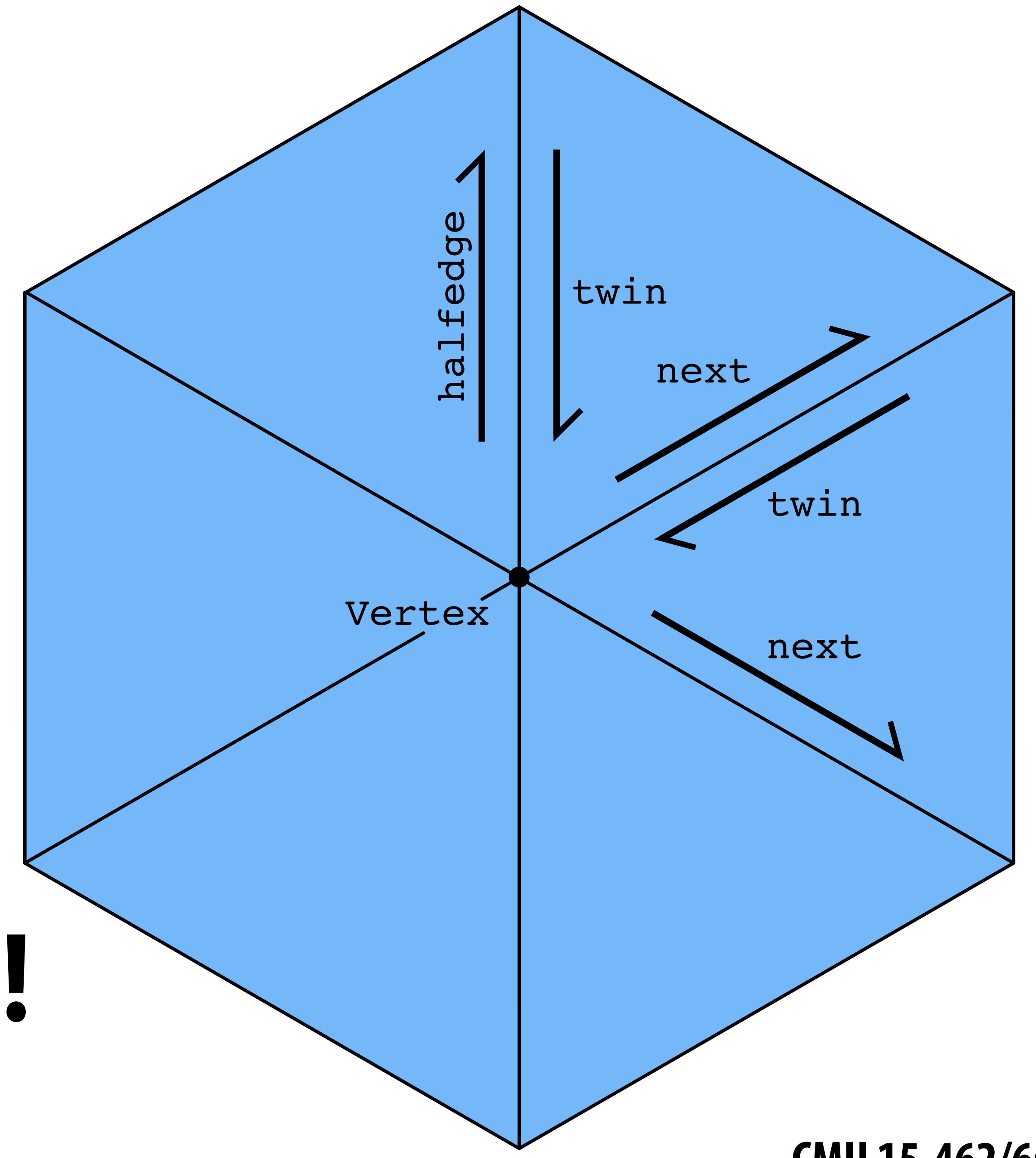
- Use "twin" and "next" pointers to move around mesh
- Use "vertex", "edge", and "face" pointers to grab element
- Example: visit all vertices of a face:

```
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

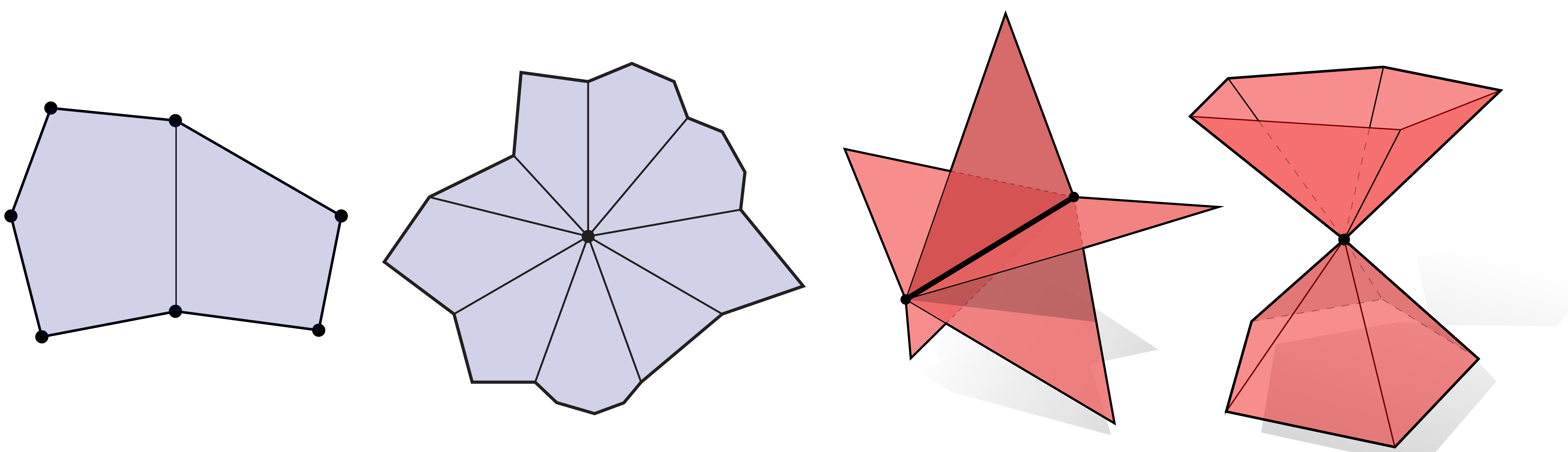
- Consider simplified halfedge data structure
- Require only "common-sense" conditions

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

(pointer to yourself!)

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

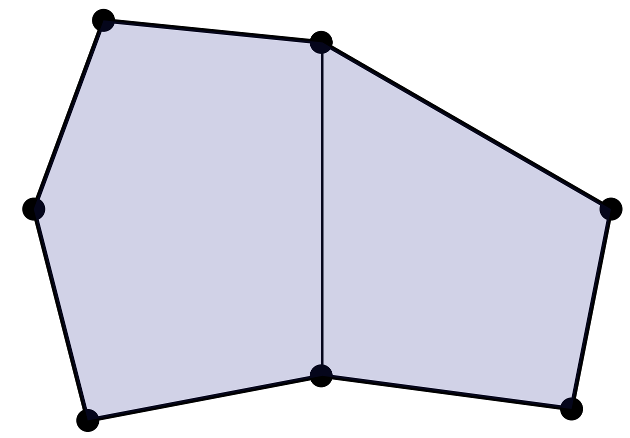
- Keep following `next`, and you'll get faces.
- Keep following `twin` and you'll get edges.
- Keep following `next->twin` and you'll get vertices.

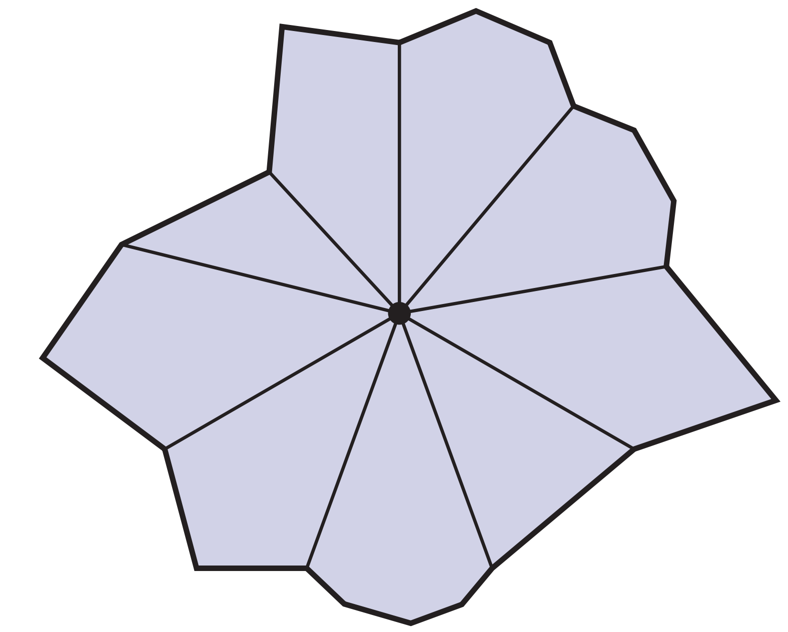


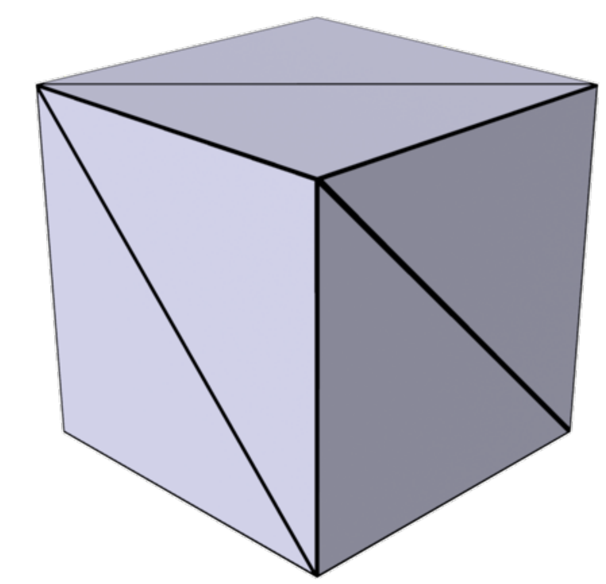
Q: Why, therefore, is it impossible to encode the red figures?

Connectivity vs. Geometry

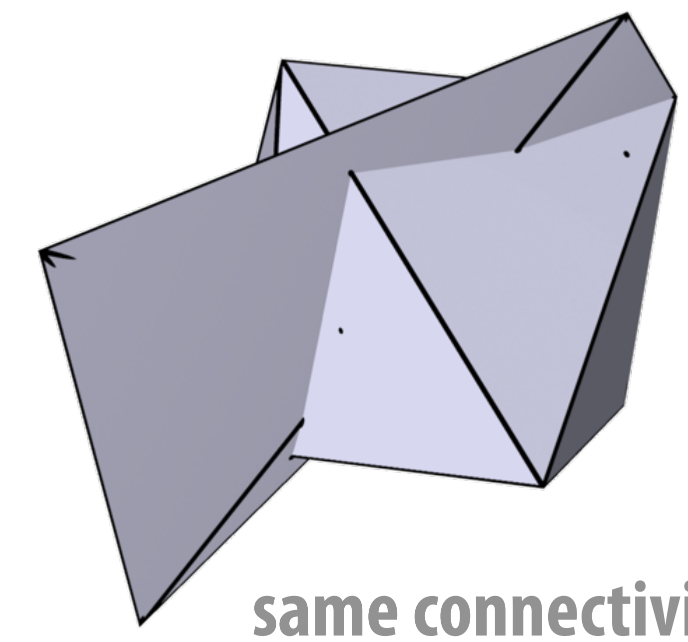
- Recall manifold conditions (fans not fins):
  - every edge contained in two faces
  - every vertex contained in one fan
- These conditions say <u>nothing</u> about vertex positions! Just connectivity
- Hence, can have perfectly good (manifold) connectivity, even if geometry is awful
- In fact, sometimes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



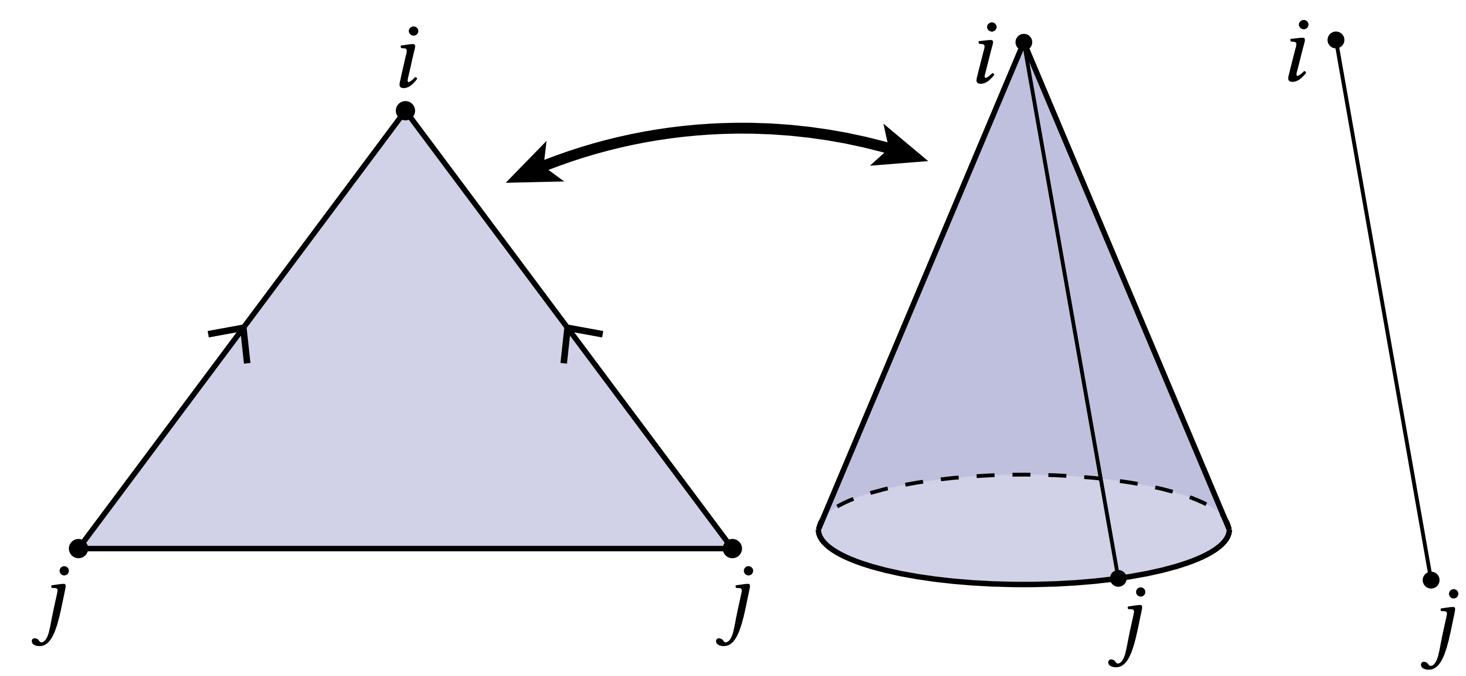


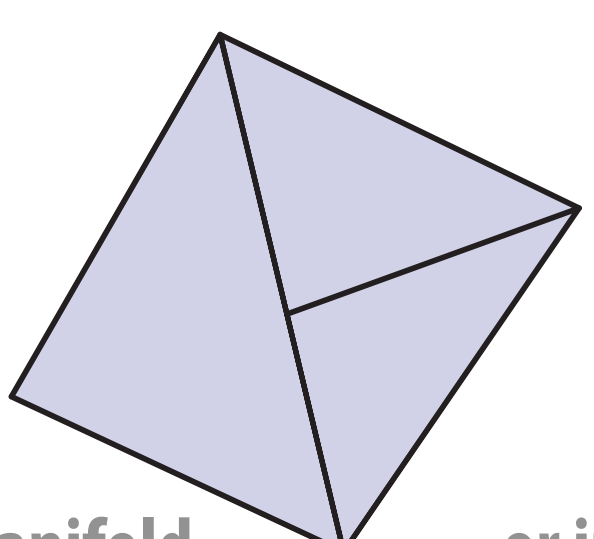


cube (manifold)

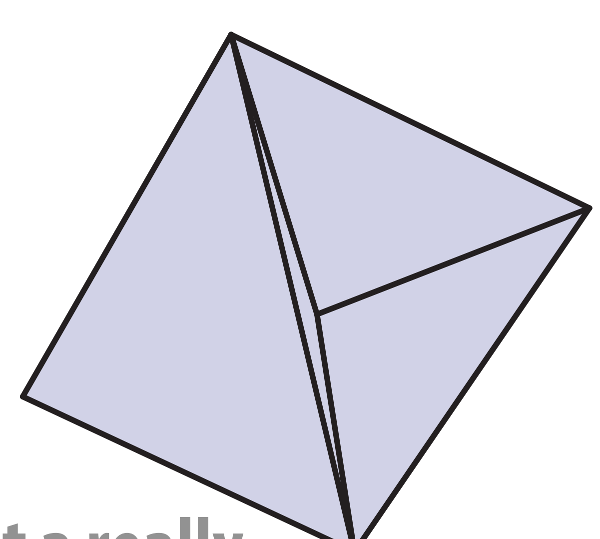


same connectivity, random vertex positions

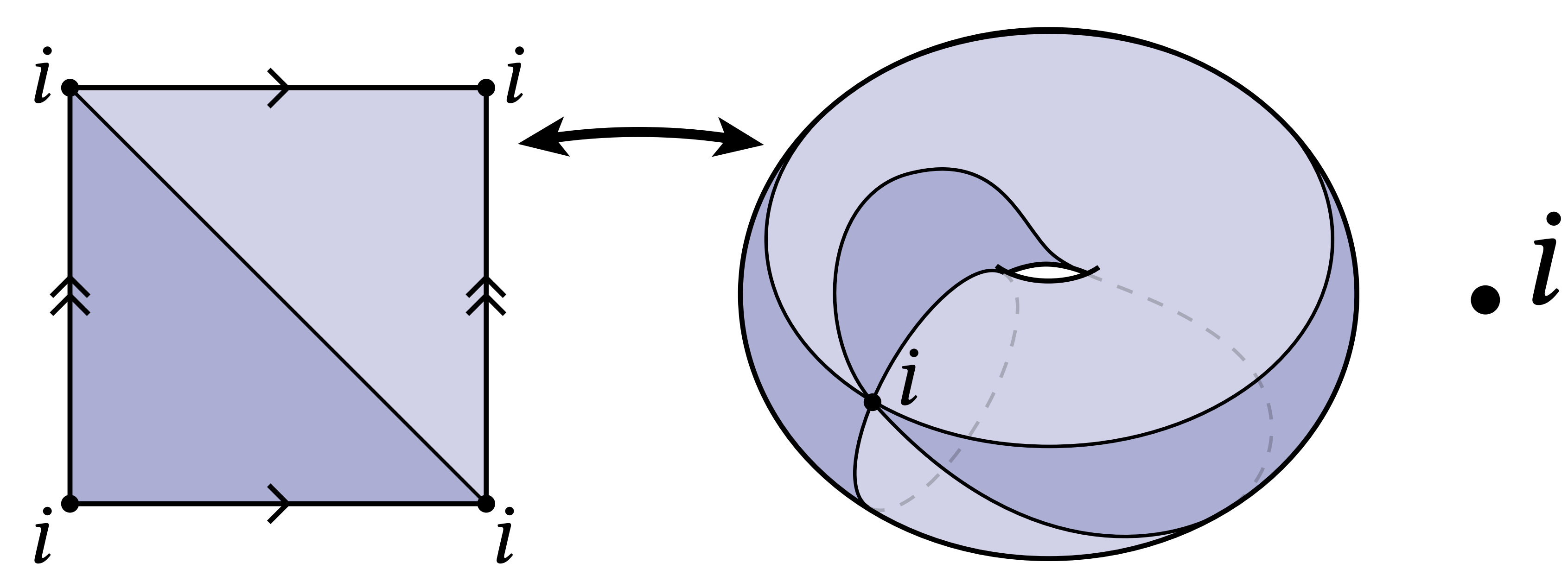




non manifold connectivity?

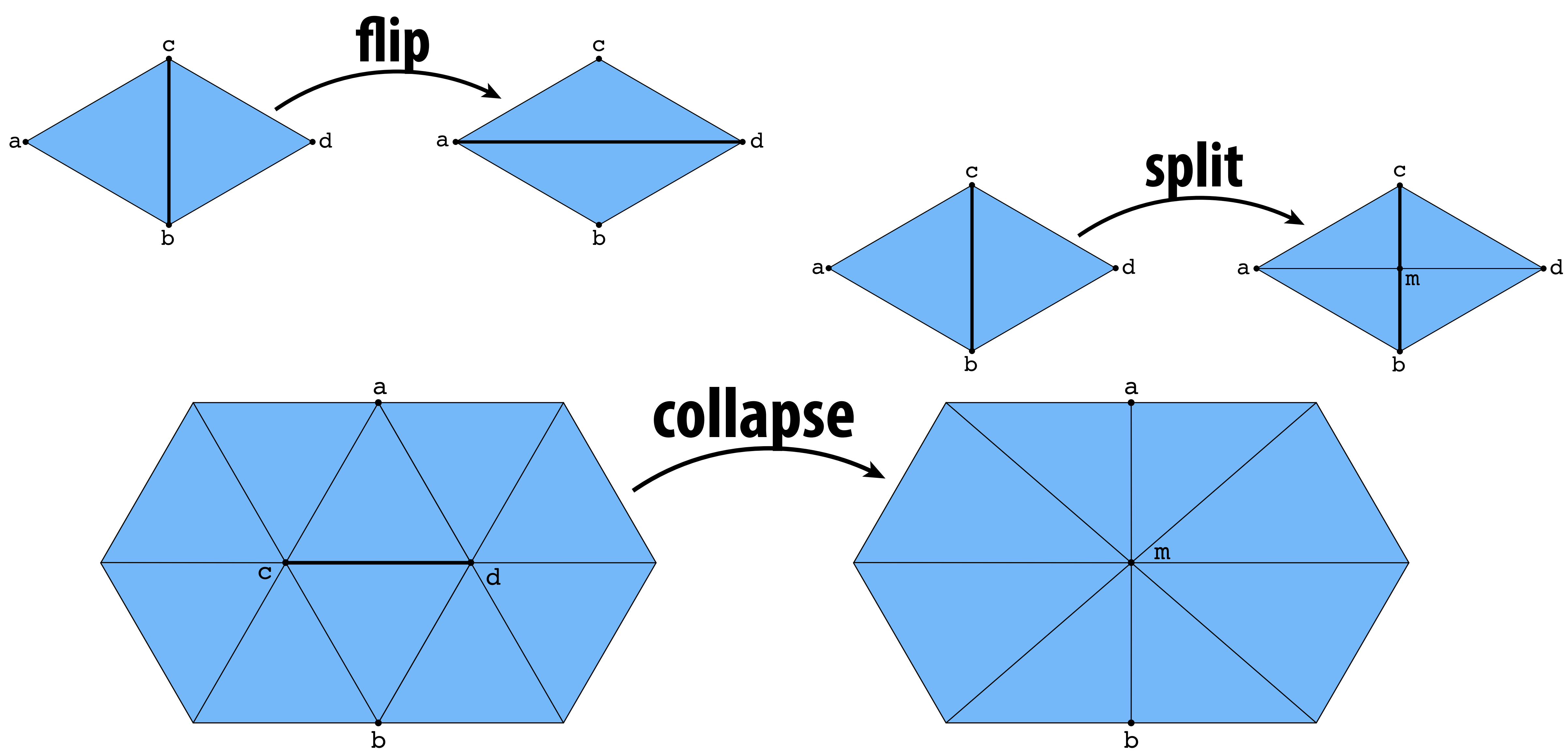


…or just a really skinny triangle?



Halfedge meshes are easy to edit

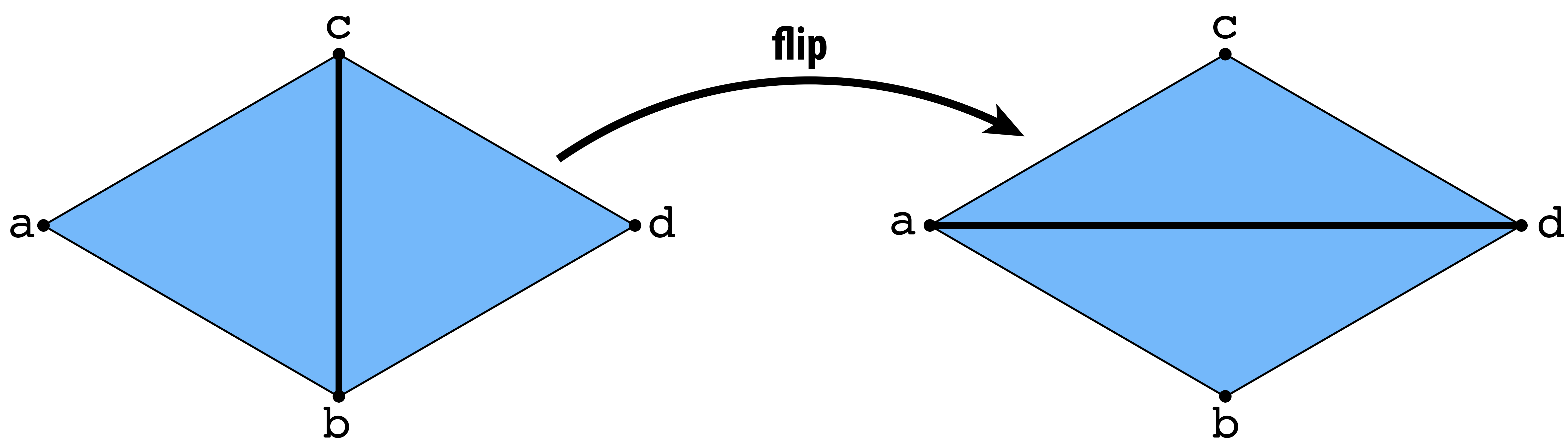
- Remember key feature of linked list: insert/delete elements
- Same story with halfedge mesh ("linked list on steroids")
- E.g., for triangle meshes, several atomic operations:



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

Edge Flip (Triangles)

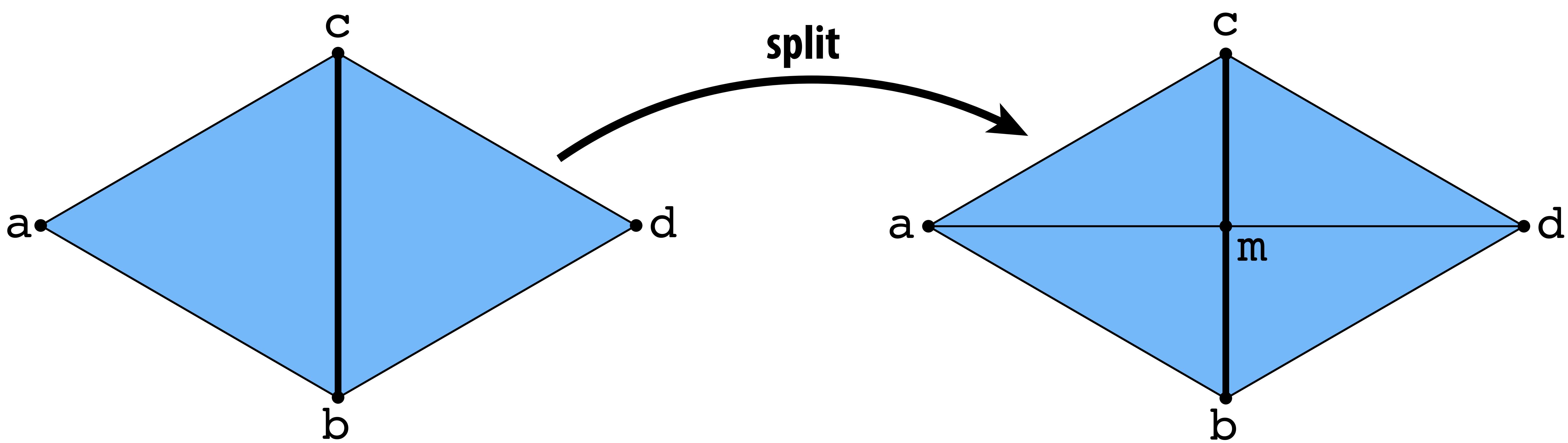
- **Triangles (a,b,c), (b,d,c) become (a,d,c), (a,b,d):**



- **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)

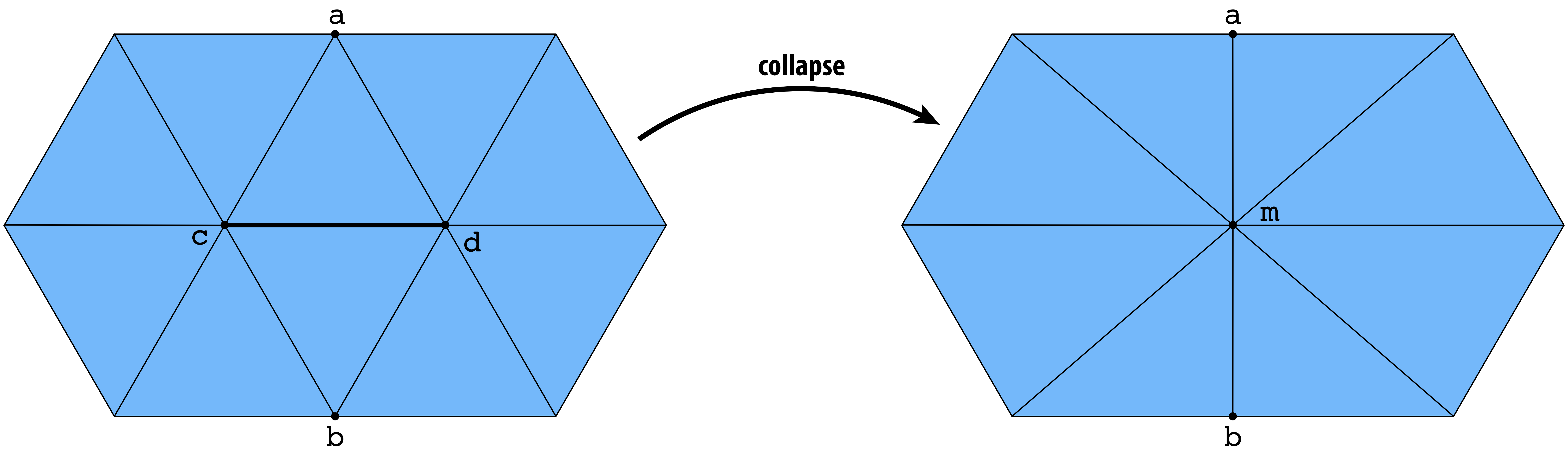
- Insert midpoint m of edge (c,b), connect to get four triangles:



- 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:

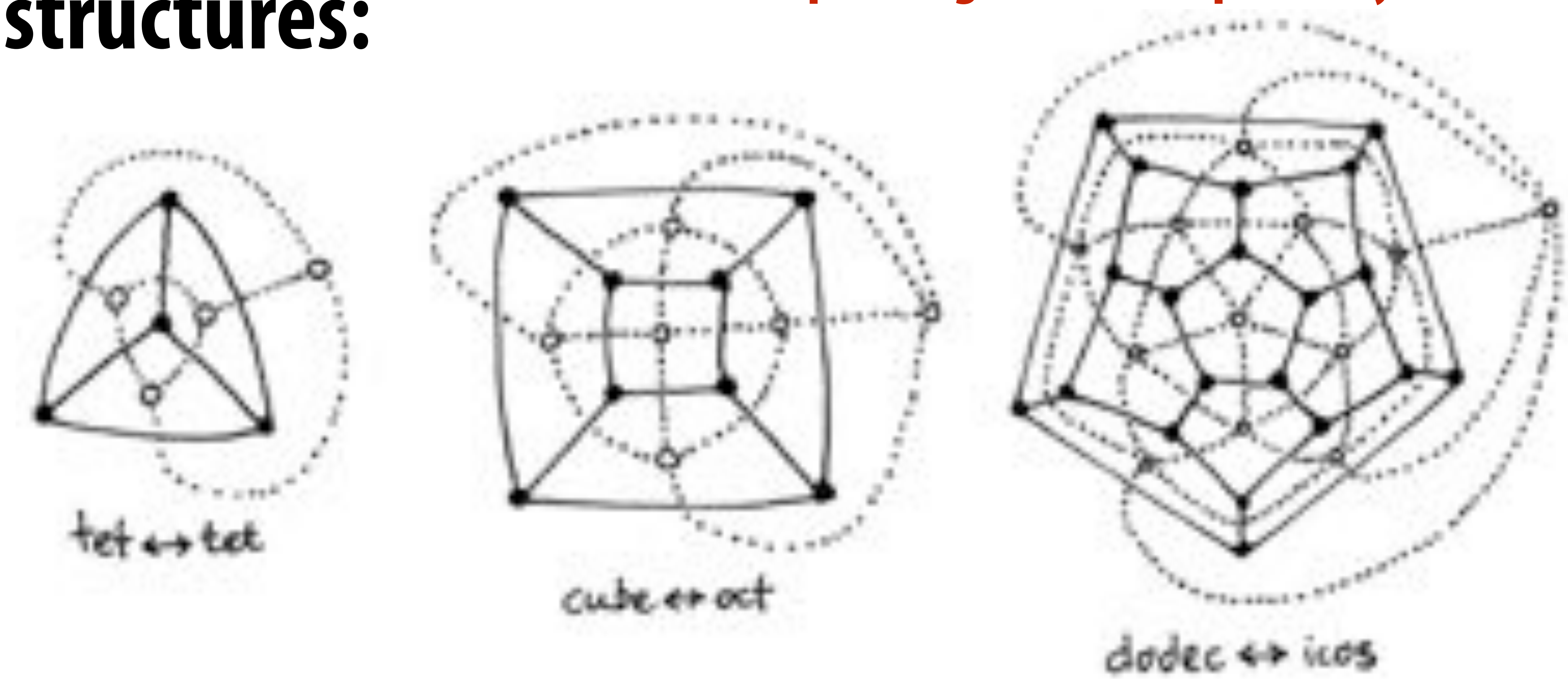


- Now have to delete elements.
- Still lots of pointer assignments!
- Q: How would we implement this with an adjacency list?
- Any other good way to do it? (E.g., different data structure?)

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

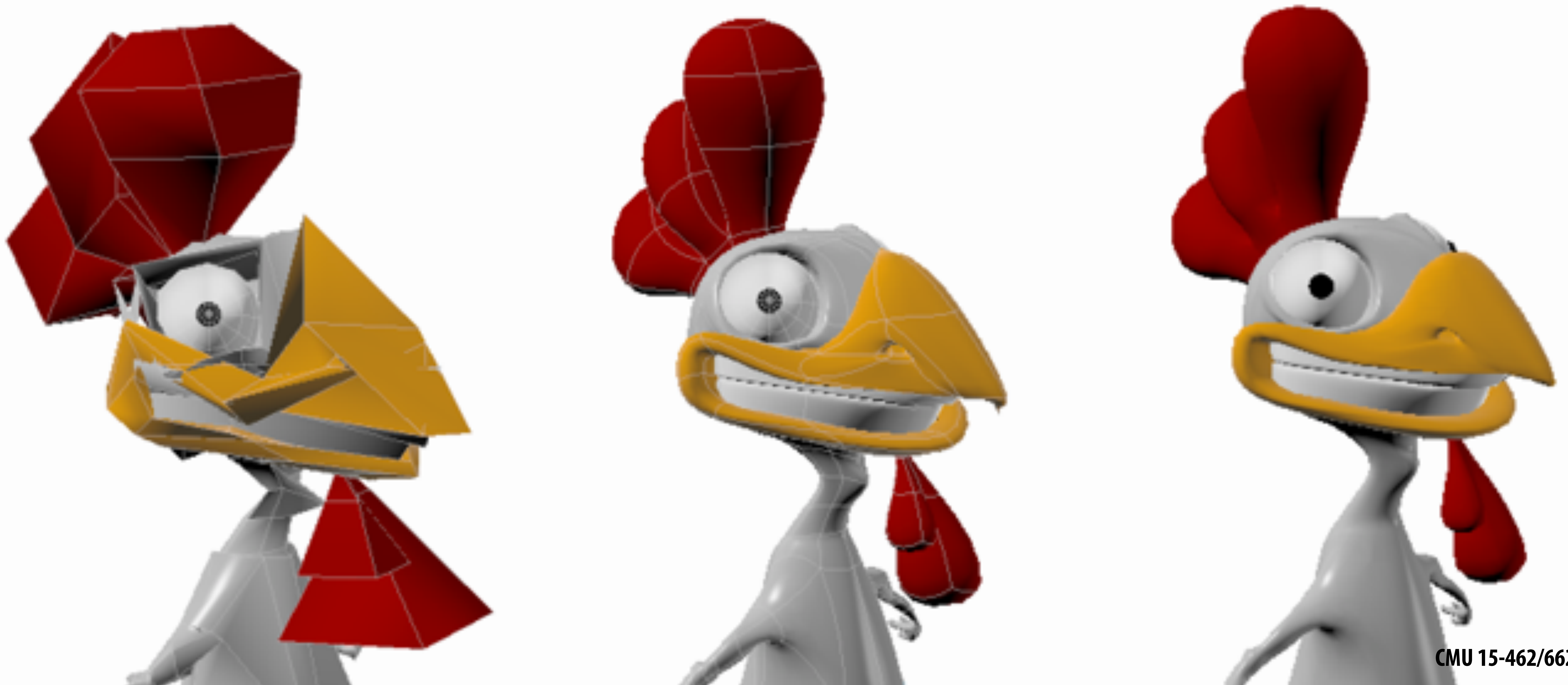
	Adjacency List	Incidence Matrices	Halfedge Mesh
constant-time neighborhood access?	NO	YES	YES
easy to add/remove mesh elements?	NO	NO	YES
nonmanifold geometry?	YES	YES	NO

Conclusion: pick the right data structure for the job!

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

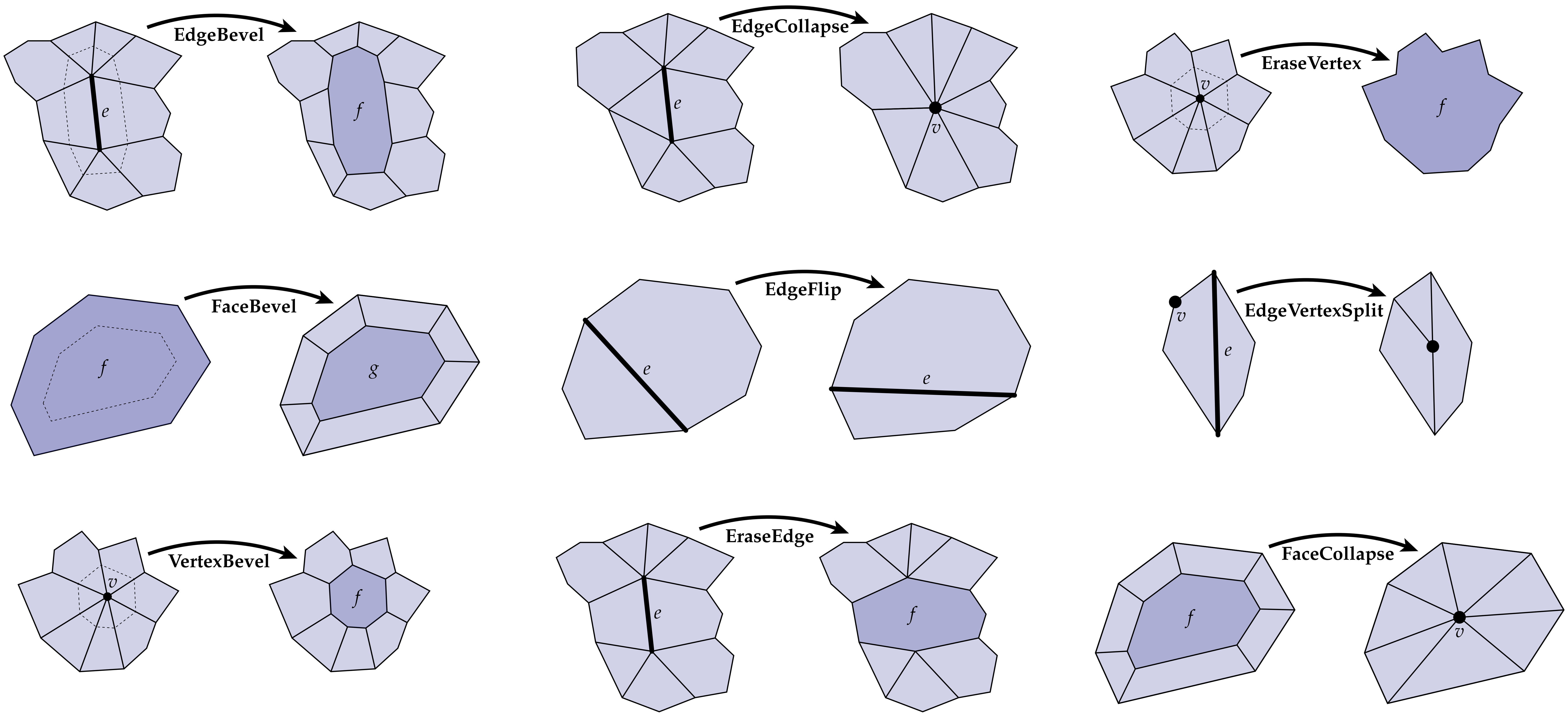
Subdivision Modeling

- 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

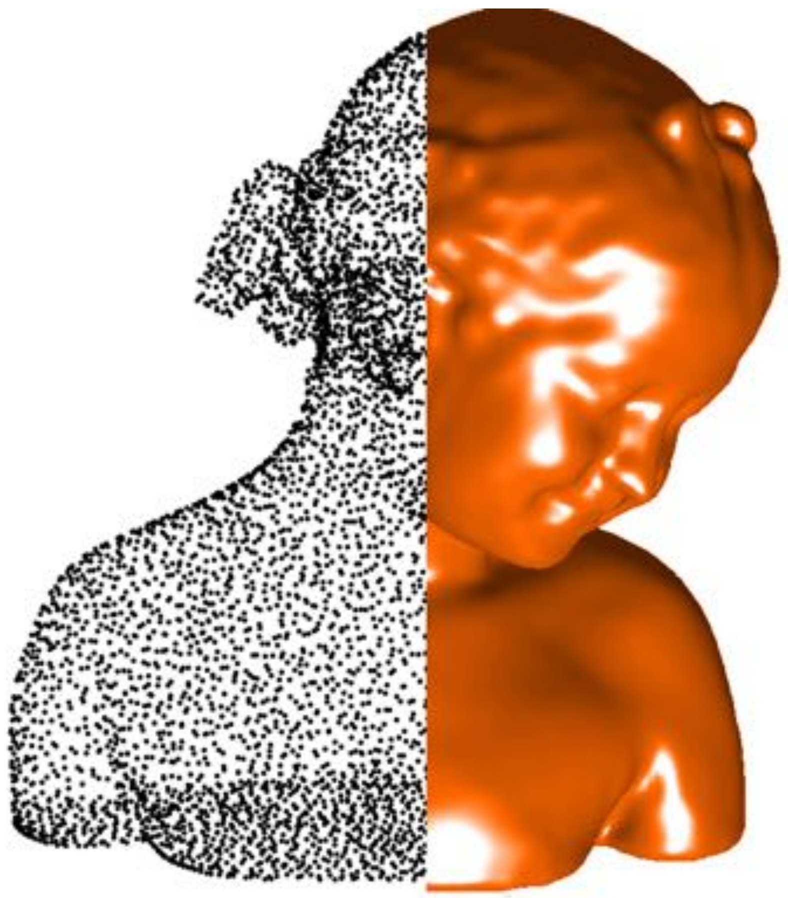
- **For general polygon meshes, we can dream up lots of local mesh operations that might be useful for modeling:**



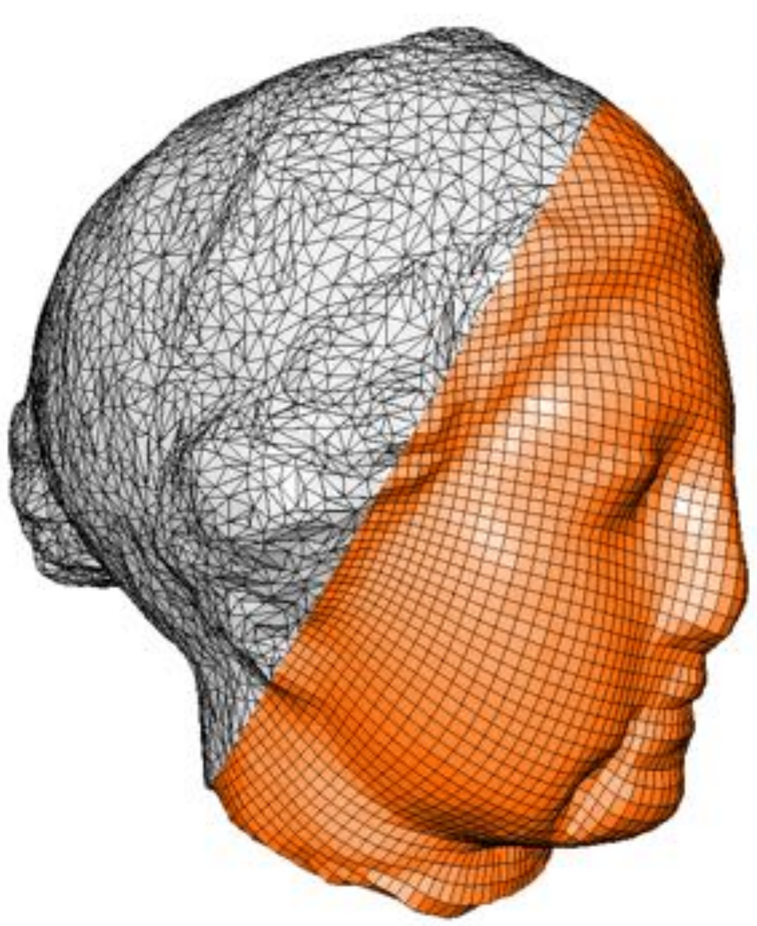
...and many, many more!

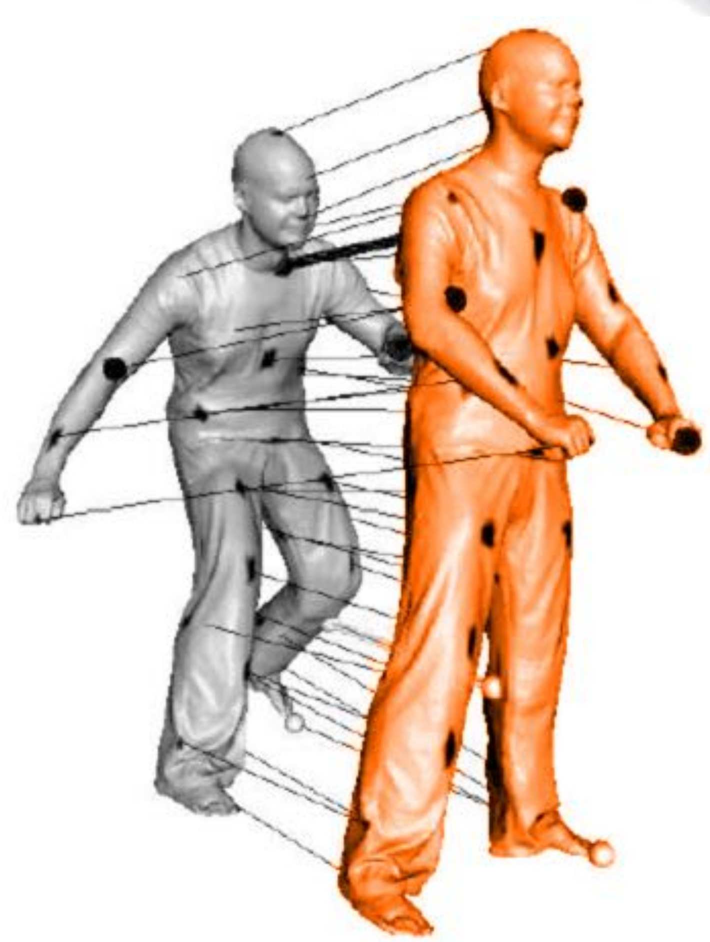
What else can we do with geometric data?

Geometry Processing

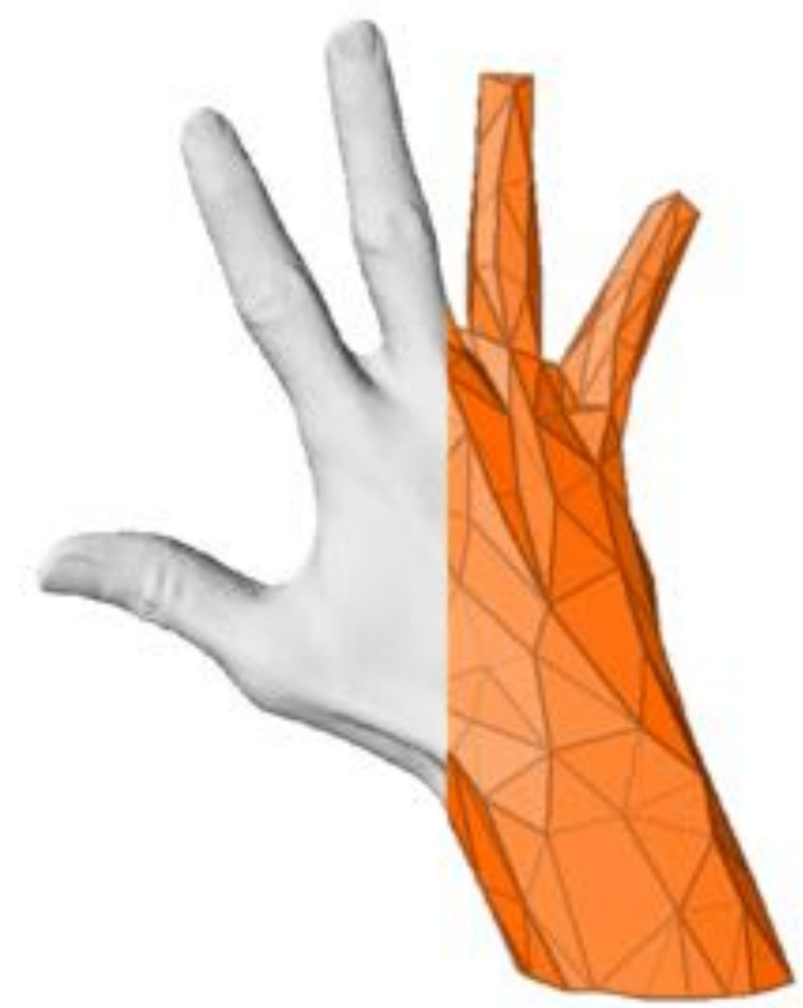

reconstruction


filtering


remeshing

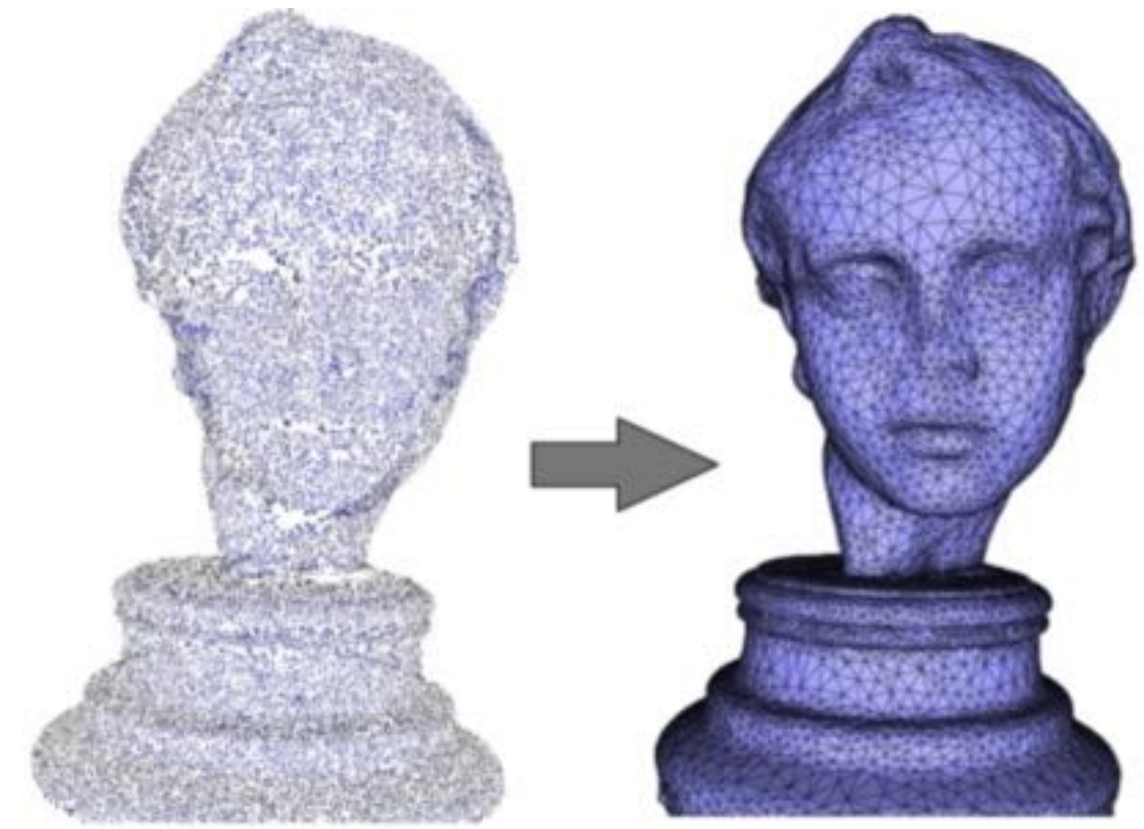

shape analysis


parameterization


compression

Geometry Processing: Reconstruction

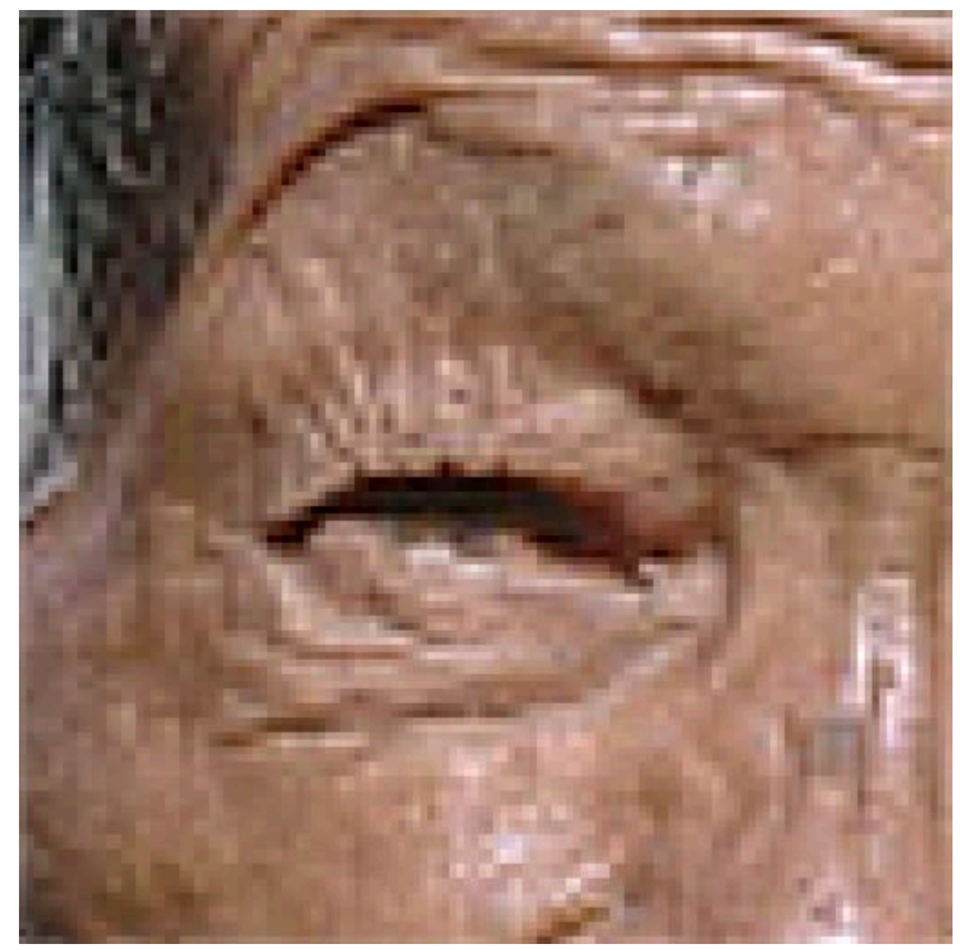
- Given samples of geometry, reconstruct surface
- What are "samples"? Many possibilities:
  - points, points & normals, ...
  - image pairs / sets (multi-view stereo)
  - line density integrals (MRI/CT scans)

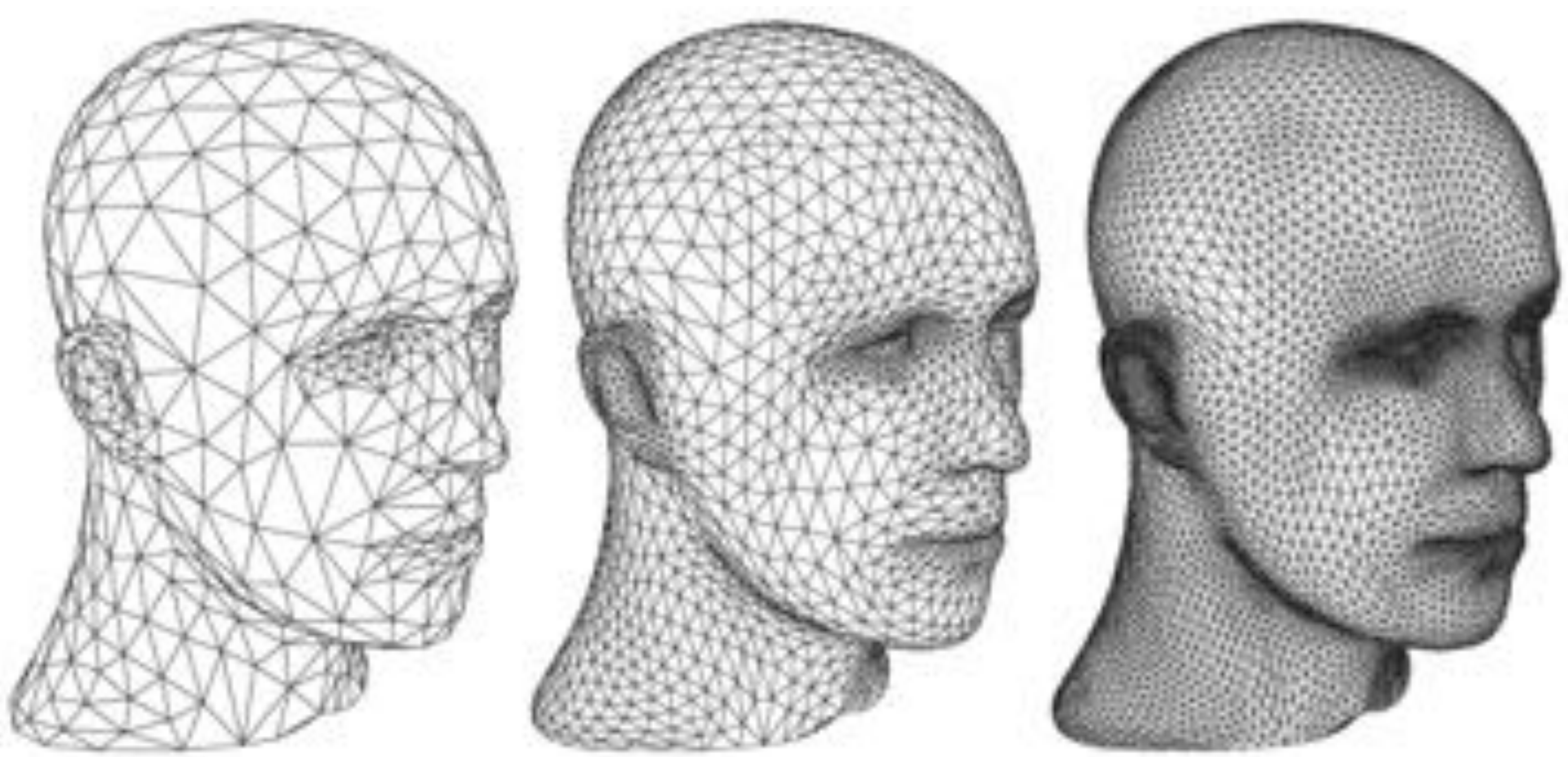


- How do you get a surface? Many techniques:
  - silhouette-based (visual hull)
  - Voronoi-based (e.g., power crust)
  - PDE-based (e.g., Poisson reconstruction)
  - Radon transform / isosurfacing (marching cubes)

Geometry Processing: Upsampling

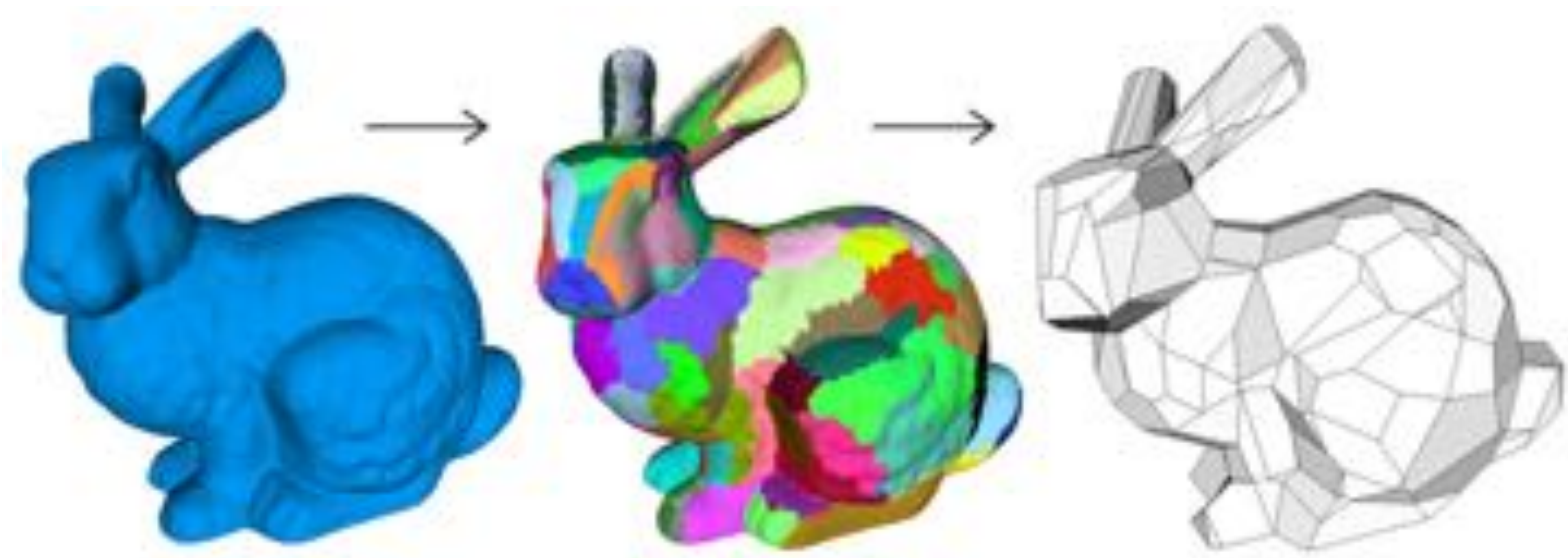
- Increase resolution via interpolation
- Images: e.g., bilinear, bicubic interpolation
- Polygon meshes:
  - subdivision
  - bilateral upsampling
  - ...





Geometry Processing: Downsampling

- Decrease resolution; try to preserve shape/appearance
- Images: nearest-neighbor, bilinear, bicubic interpolation
- Point clouds: subsampling (just take fewer points!)
- Polygon meshes:
  - iterative decimation, variational shape approximation, ...

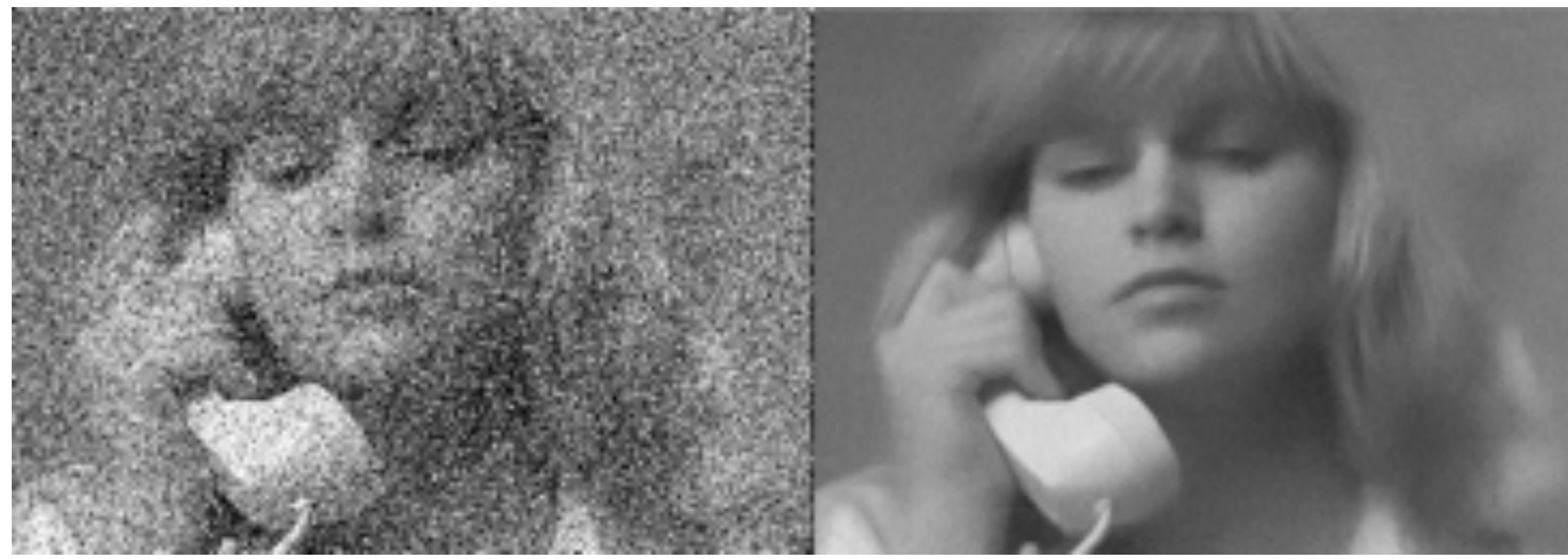


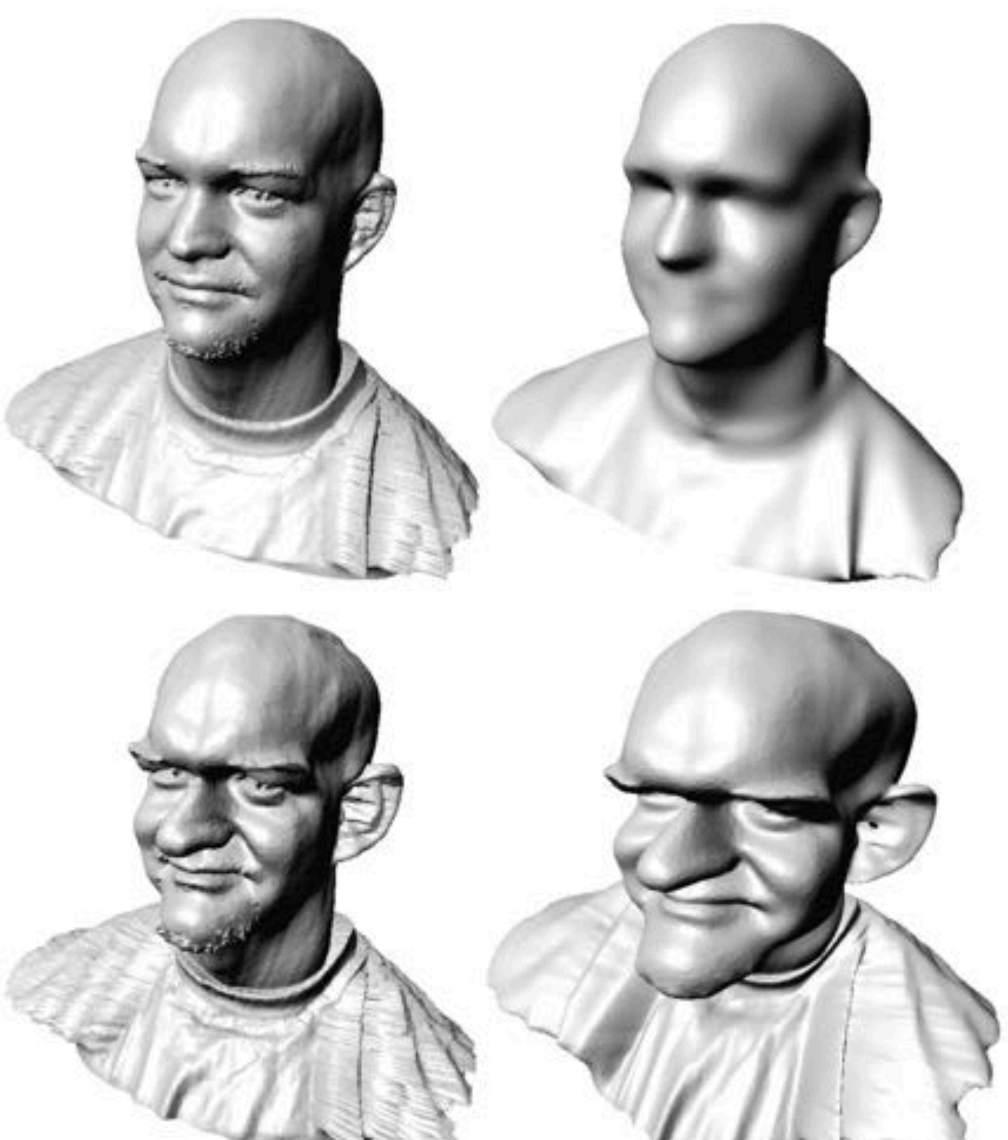
Geometry Processing: Resampling

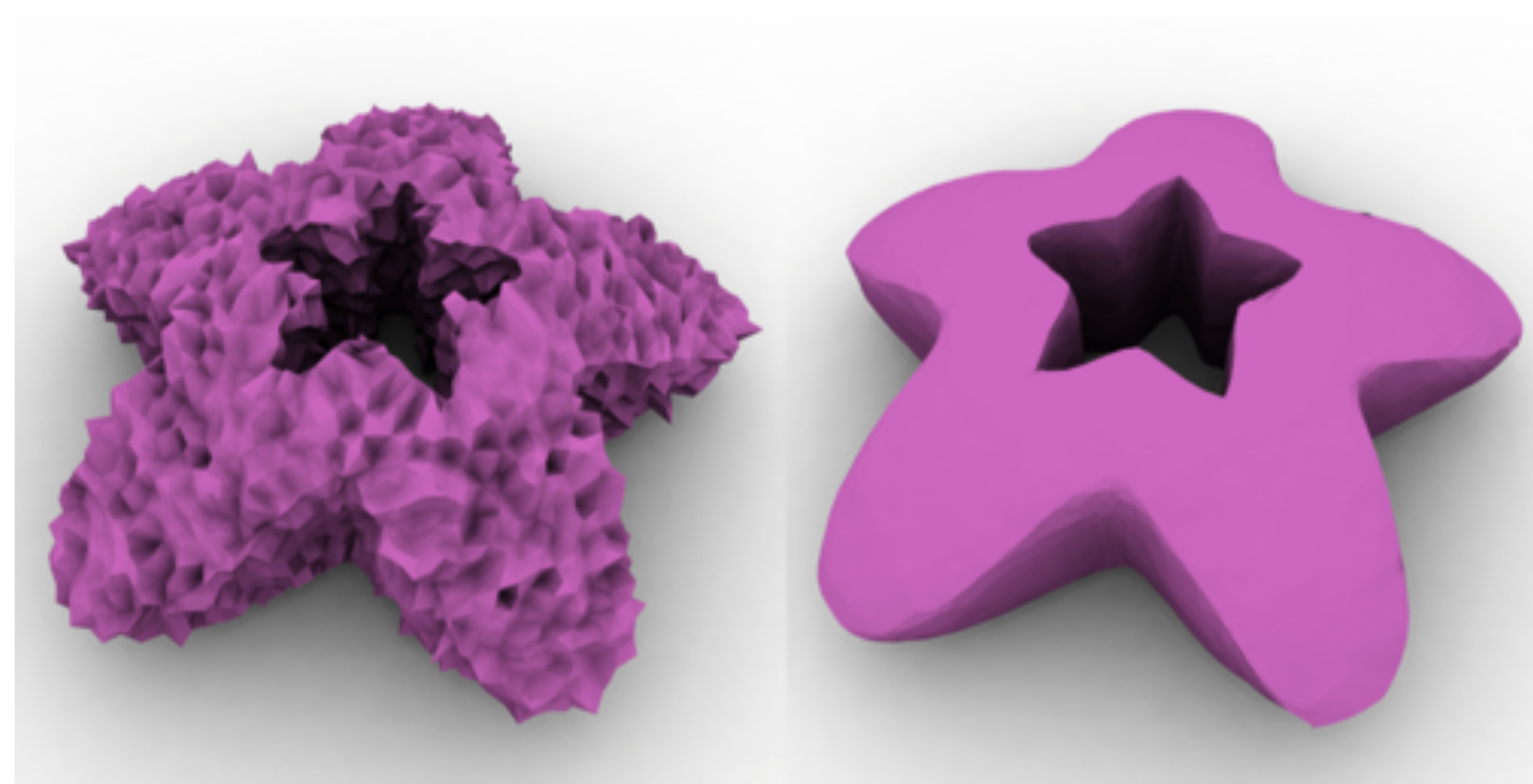
- Modify sample distribution to improve quality
- Images: not an issue! (Pixels always stored on a regular grid)
- Meshes: shape of polygons is extremely important!
  - different notion of "quality" depending on task
  - e.g., visualization vs. solving equations

Geometry Processing: Filtering

- Remove noise, or emphasize important features (e.g., edges)
- Images: blurring, bilateral filter, edge detection, ...
- Polygon meshes:
  - curvature flow
  - bilateral filter
  - spectral filter

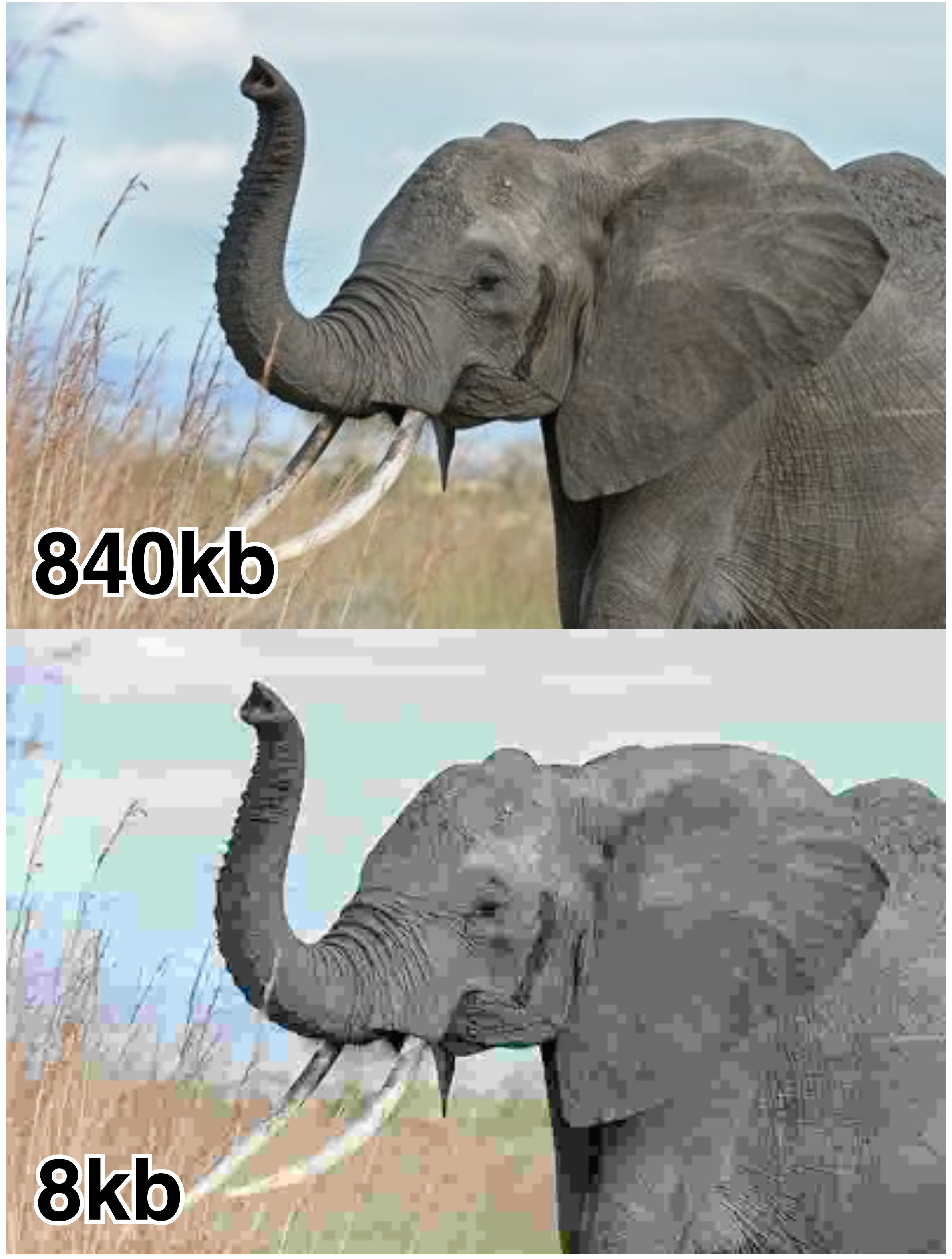


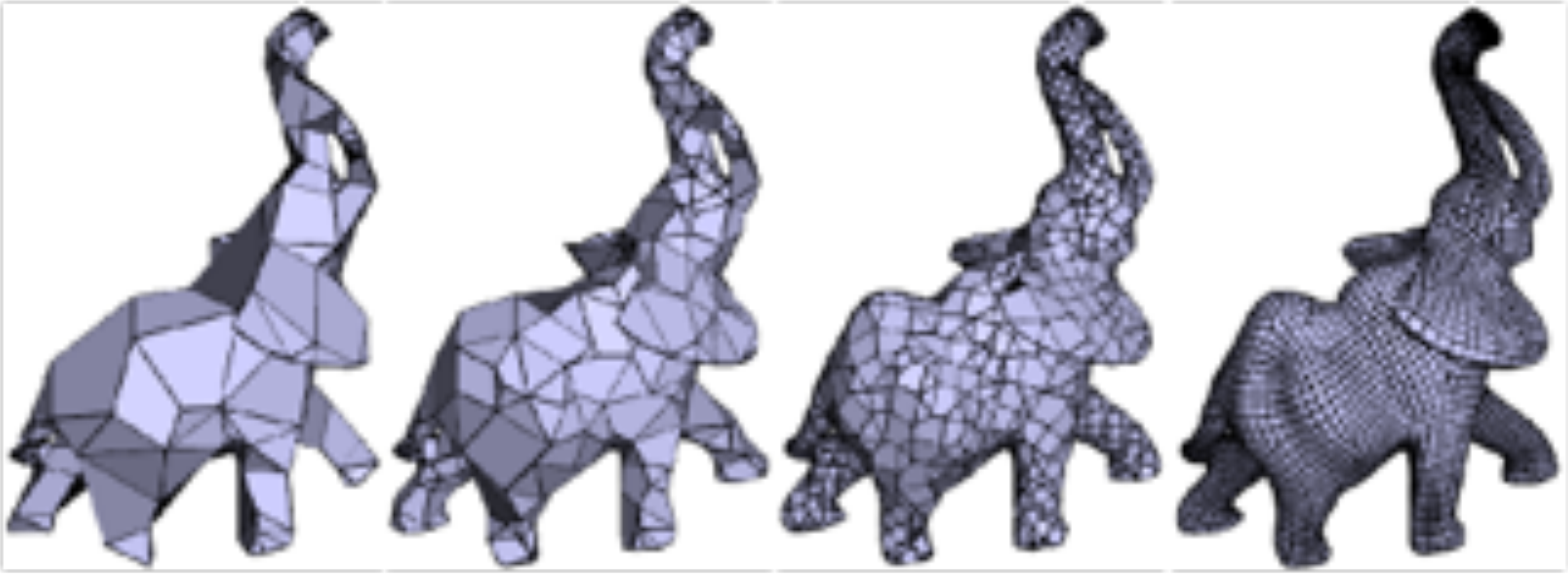




Geometry Processing: Compression

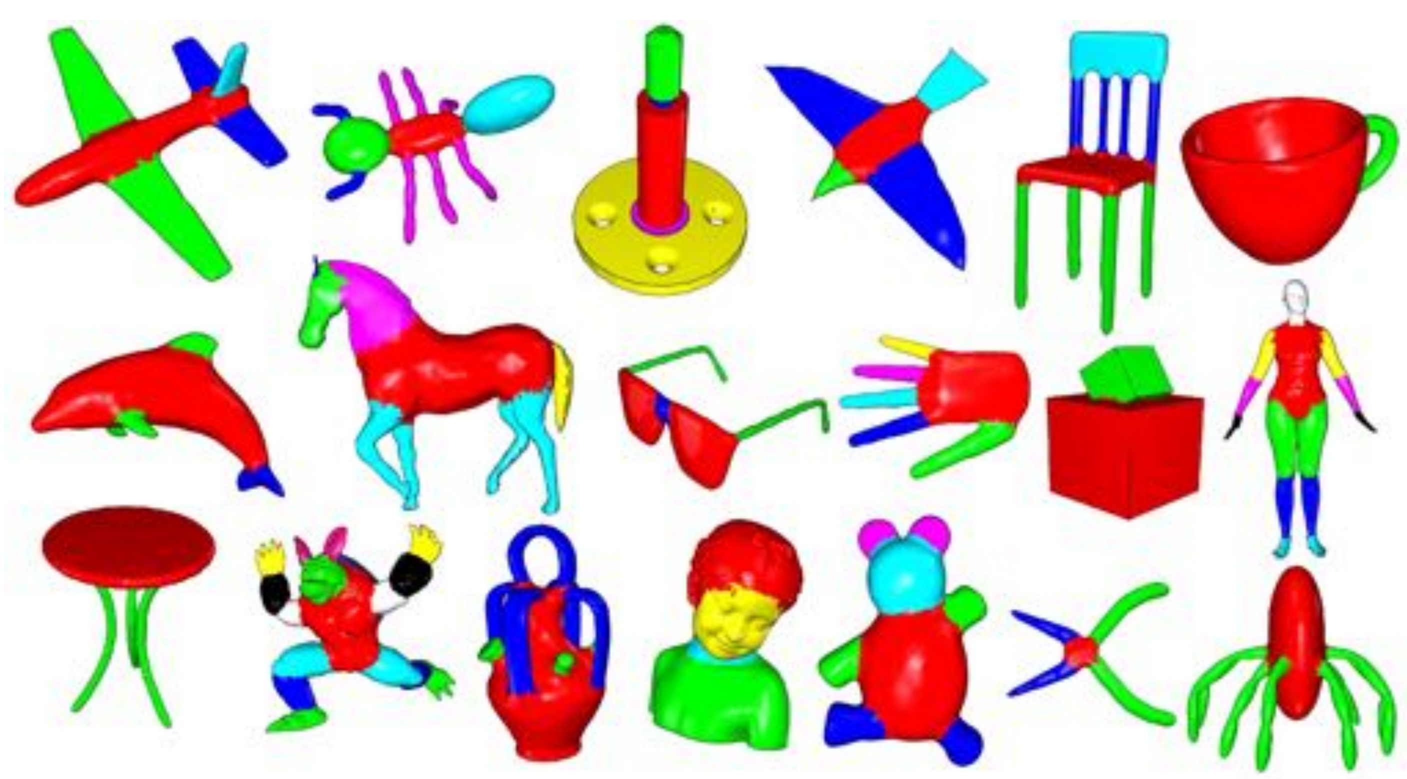
- Reduce storage size by eliminating redundant data/ approximating unimportant data
- Images:
  - run-length, Huffman coding - lossless
  - cosine/wavelet (JPEG/MPEG) - lossy
- Polygon meshes:
  - compress geometry and connectivity
  - many techniques (lossy & lossless)

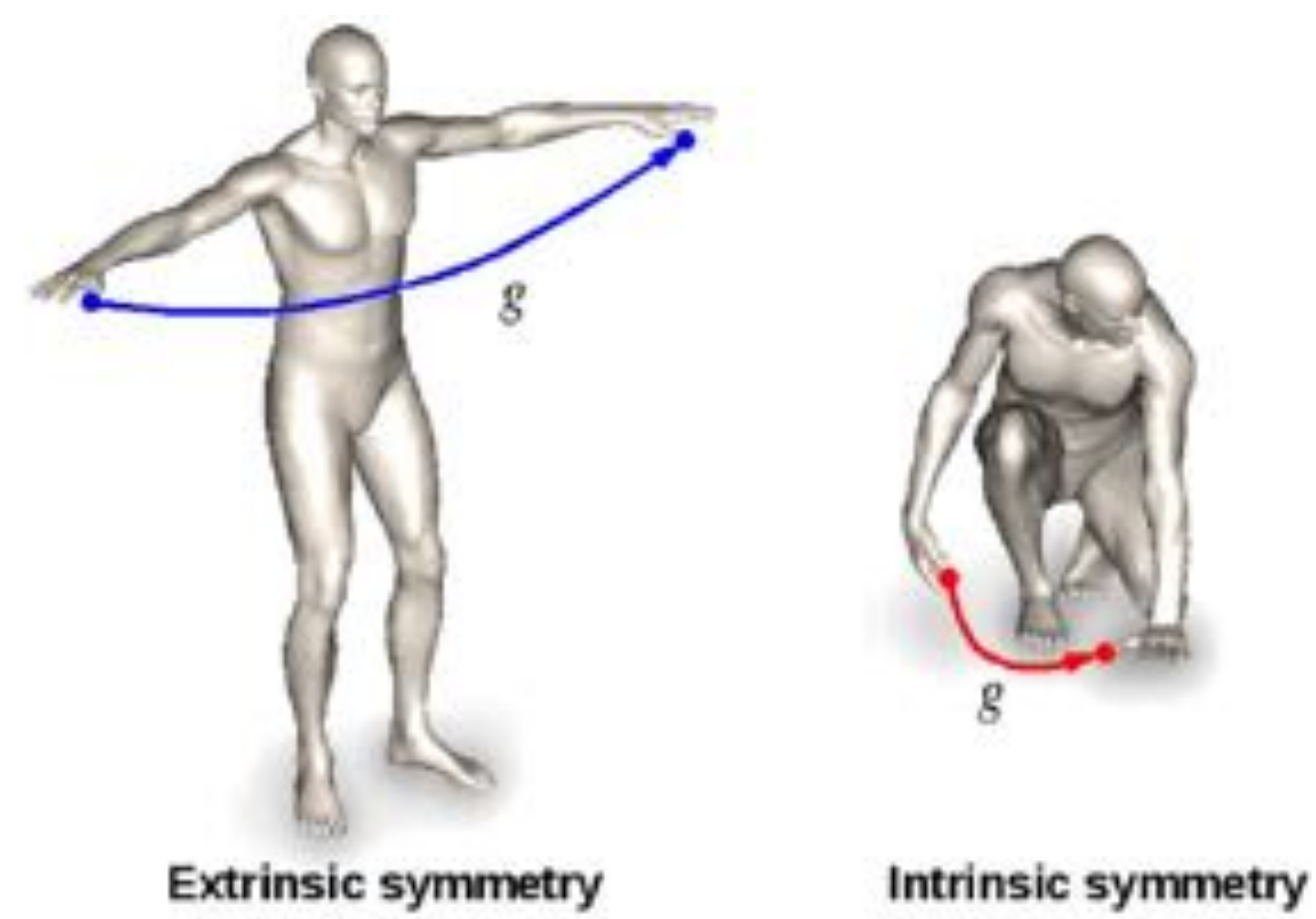


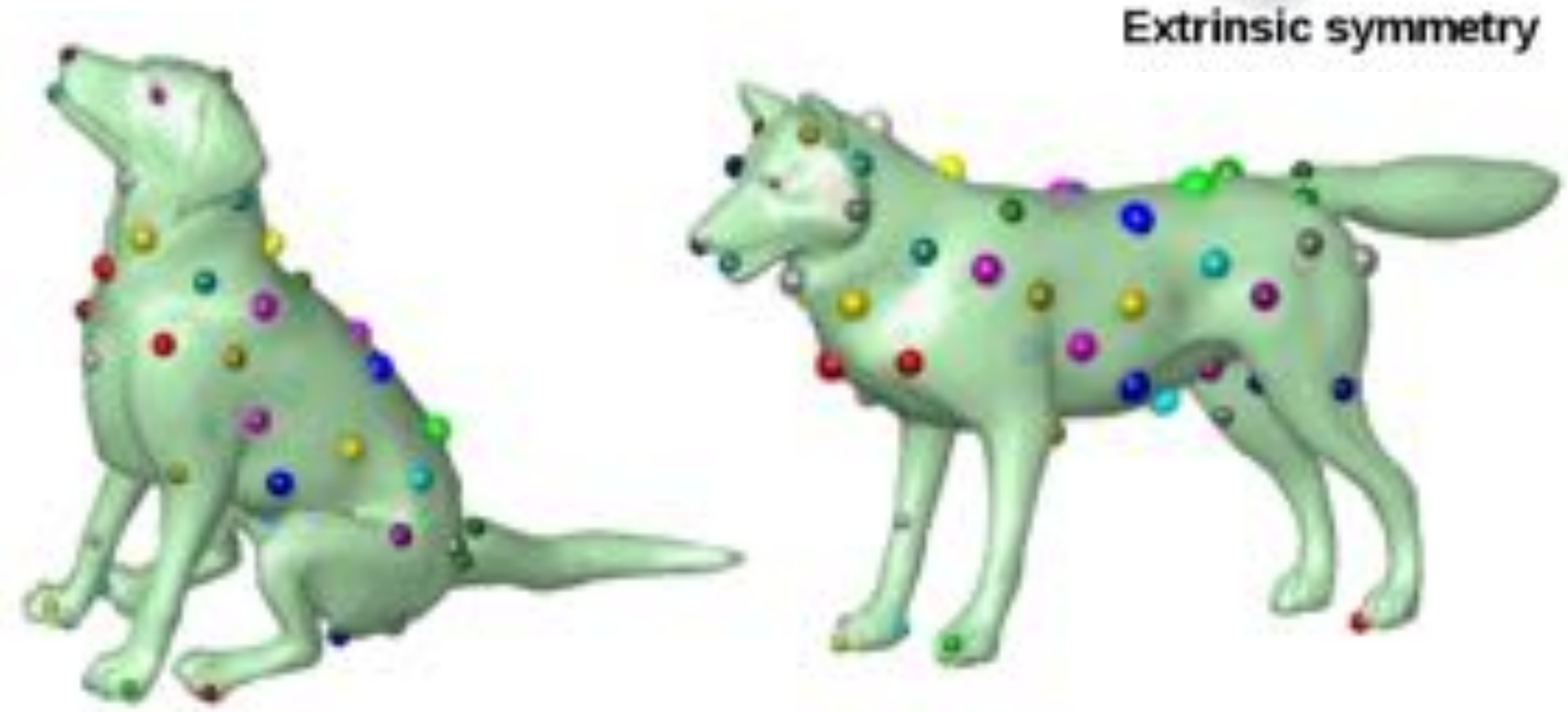


Geometry Processing: Shape Analysis

- Identify/understand important semantic features
- Images: computer vision, segmentation, face detection, ...
- Polygon meshes:
  - segmentation, correspondence, symmetry detection, ...



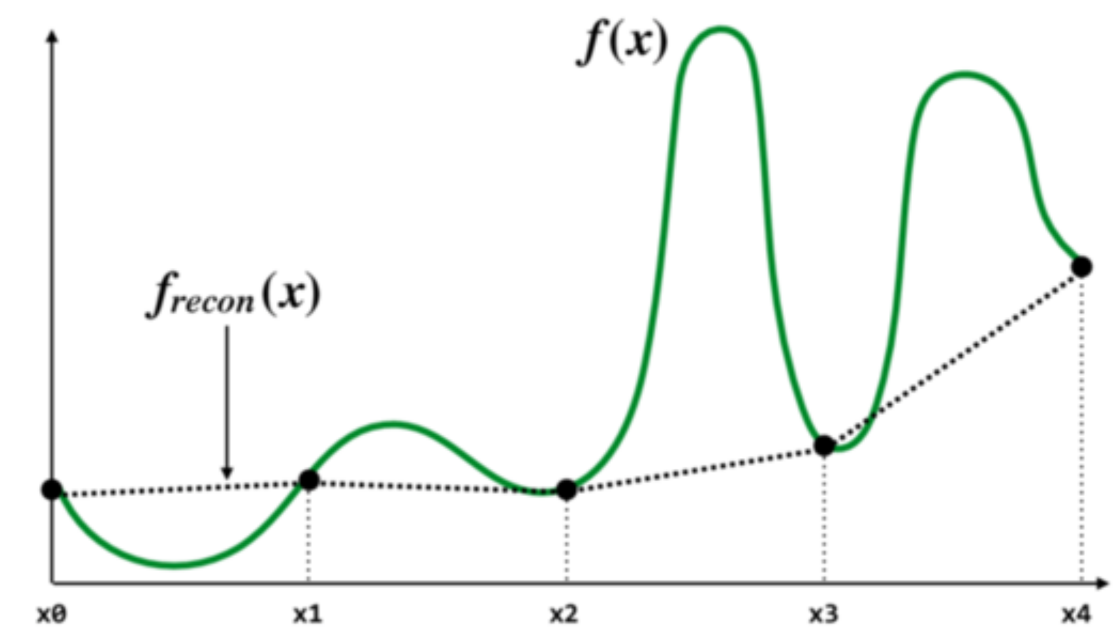




Enough overview—
Let's process some geometry!

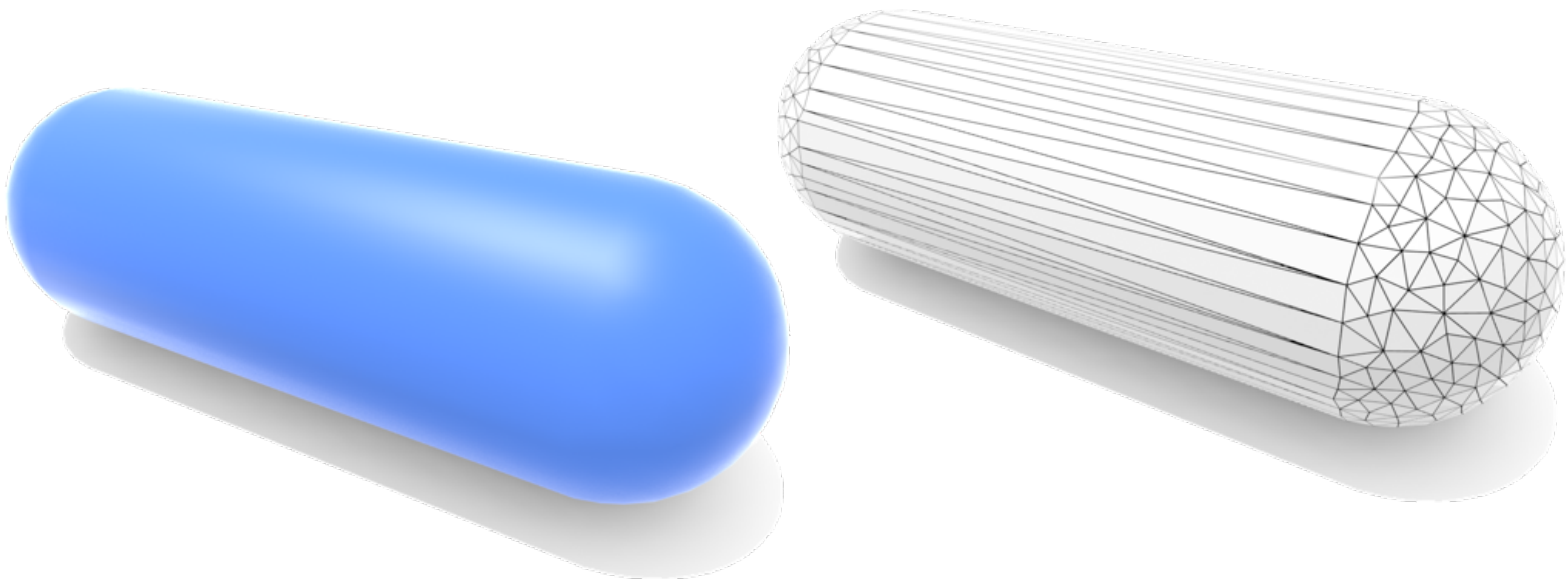
Remeshing as resampling

- Remember our discussion of aliasing
- Bad sampling makes signal appear different than it really is
- E.g., undersampled curve looks flat
- Geometry is no different!
  - undersampling destroys features
  - oversampling bad for performance



What makes a "good" mesh?

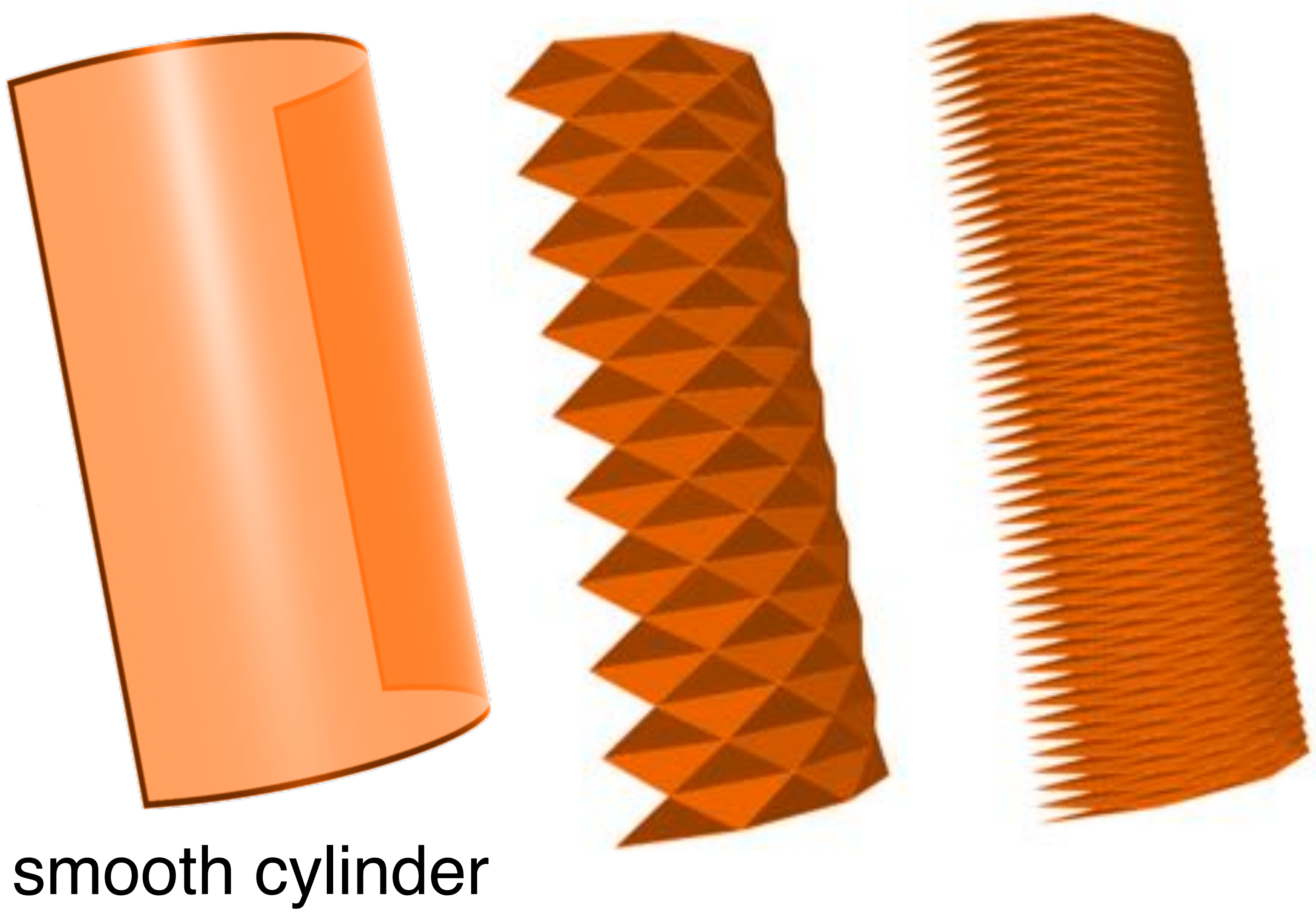
- One idea: good approximation of original shape!
- Keep only elements that contribute information about shape
- Add additional information where, e.g., curvature is large



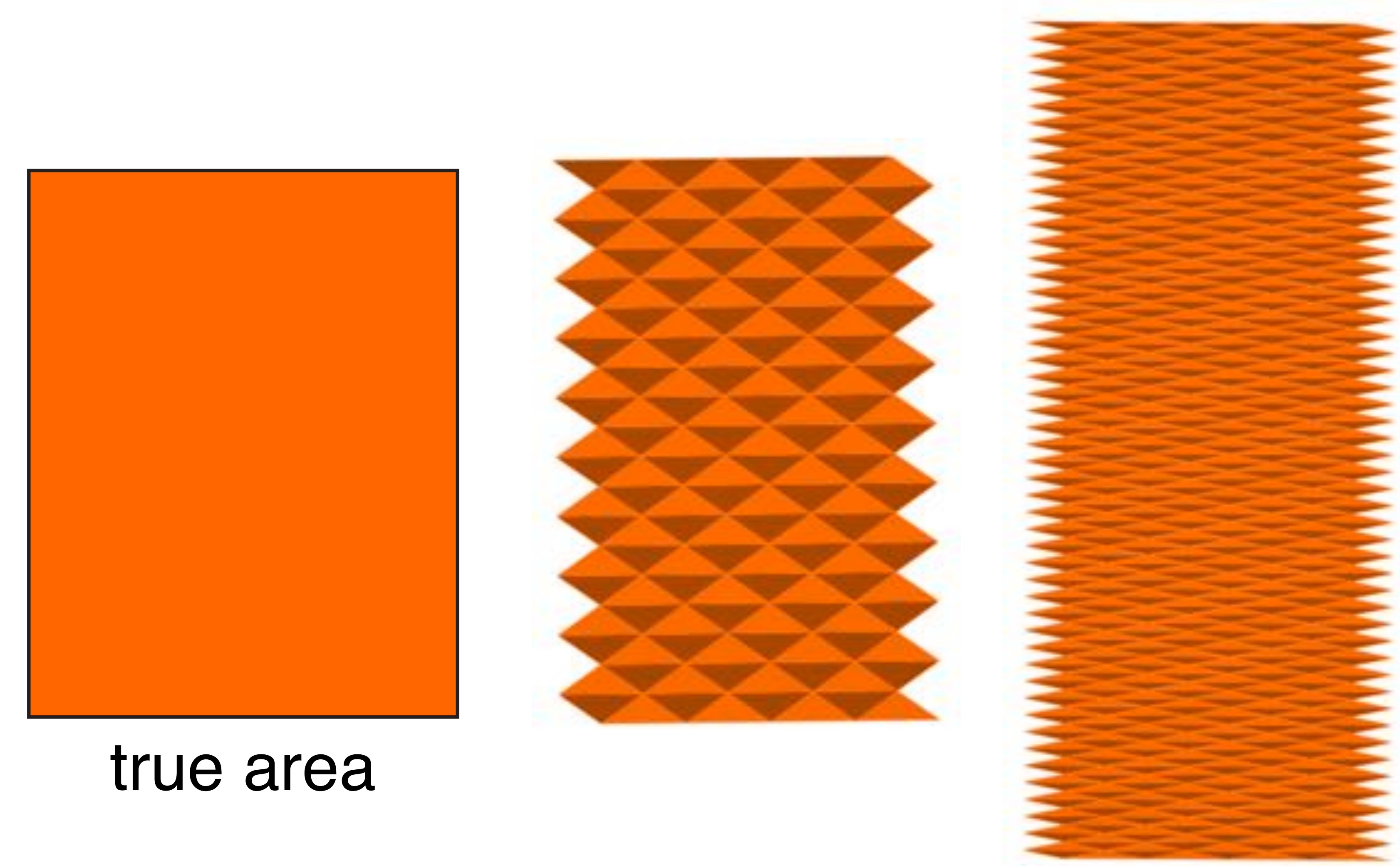
Approximation of position is not enough!

- Just because the vertices of a mesh are close to the surface it approximates does not mean it's a good approximation!
- Can still have wrong appearance, wrong area, wrong…
- Need to consider other factors*, e.g., close approximation of surface normals

vertices exactly on smooth cylinder



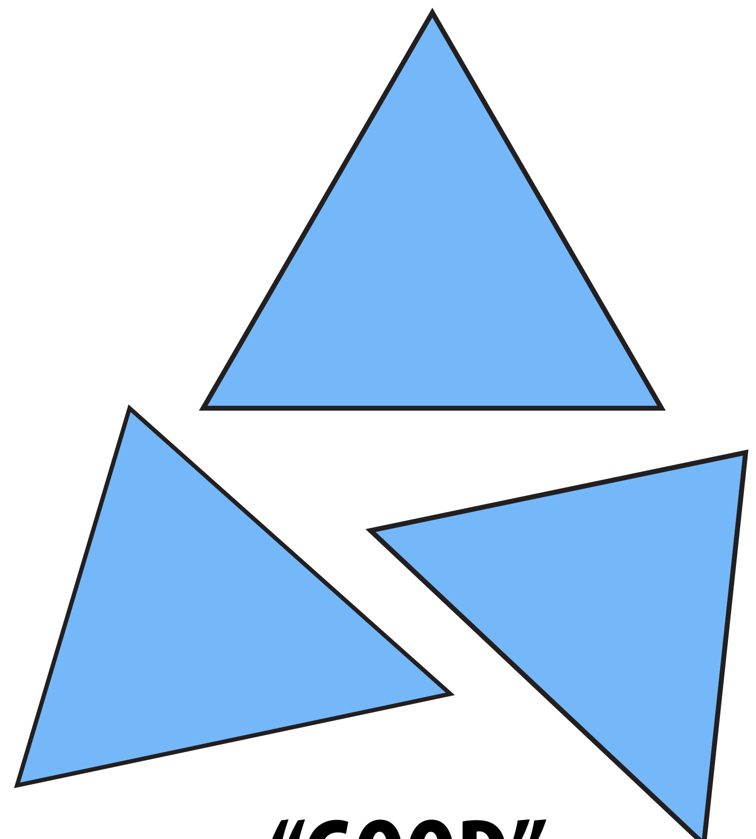
flattening of smooth cylinder & meshes



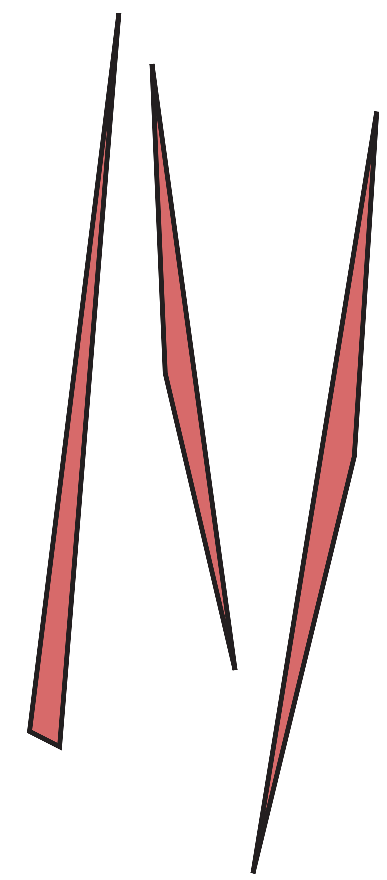
*See Hildebrandt et al (2007), "On the convergence of metric and geometric properties of polyhedral surfaces"

What else makes a “good” triangle mesh?

- Another rule of thumb: triangle



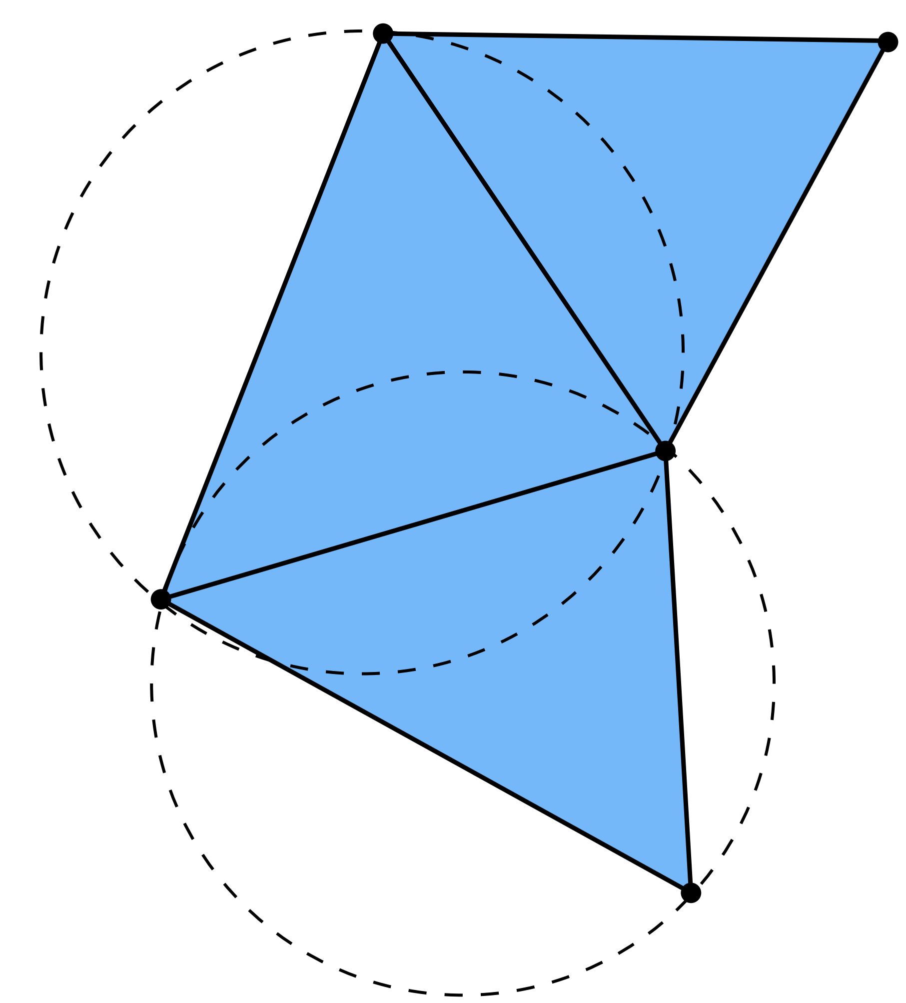
“GOOD”



“BAD”

pronunciation:

DELAUNAY

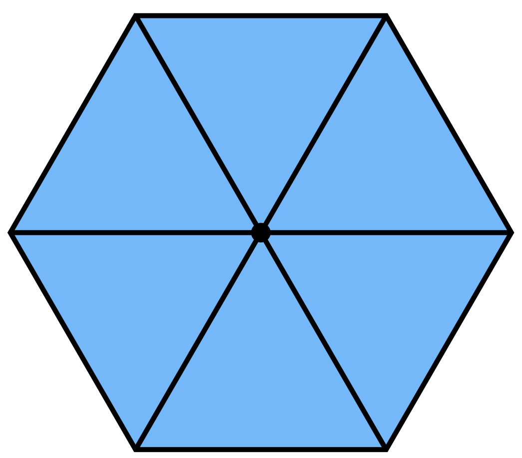


- E.g., all angles close to 60 degrees
- More sophisticated condition: Delaunay (empty circumcircles)
  – often helps with numerical accuracy/stability
  – coincides with <u>shockingly</u> many other desirable properties
  (maximizes minimum angle, provides smoothest interpolation, guarantees maximum principle…)
- Tradeoffs w/ good geometric approximation*
  –e.g., long & skinny might be “more efficient”

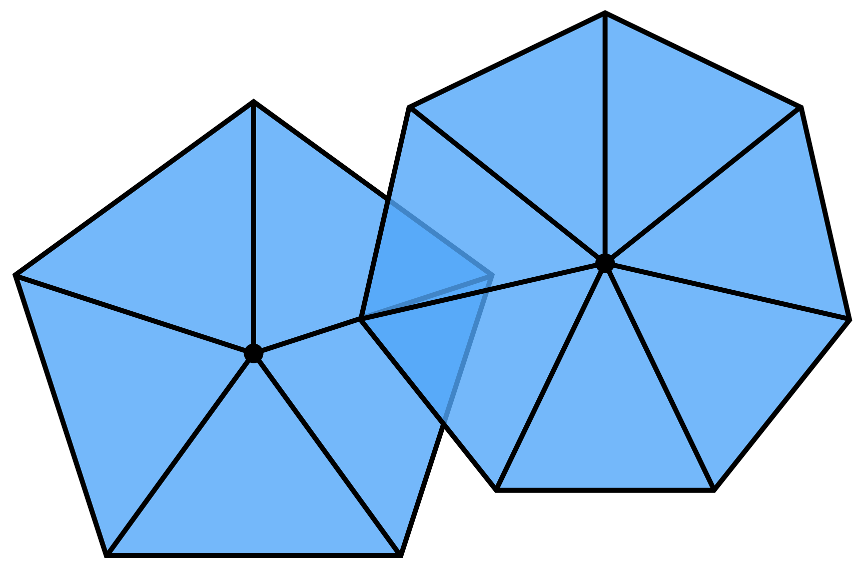
*see Shewchuk, “What is a Good Linear Element”

What else constitutes a “good” mesh?

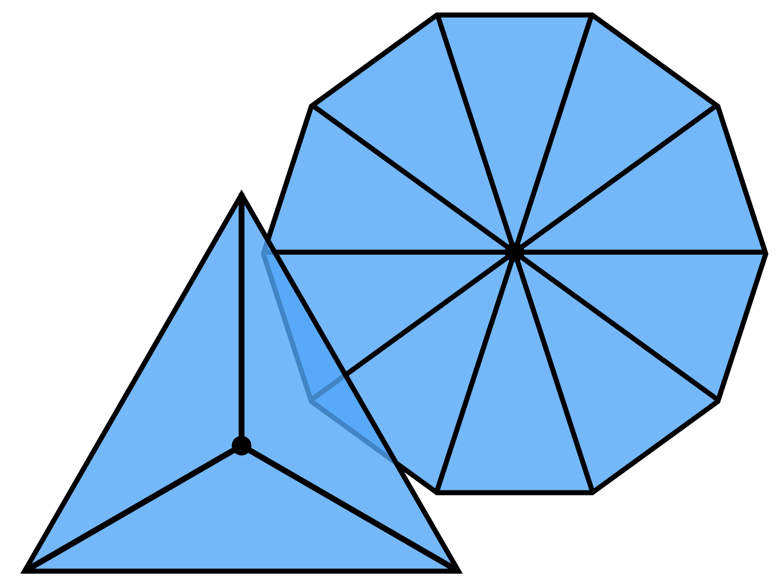
- **Another rule of thumb: regular vertex degree**
- **Degree 6 for triangle mesh, 4 for quad mesh**



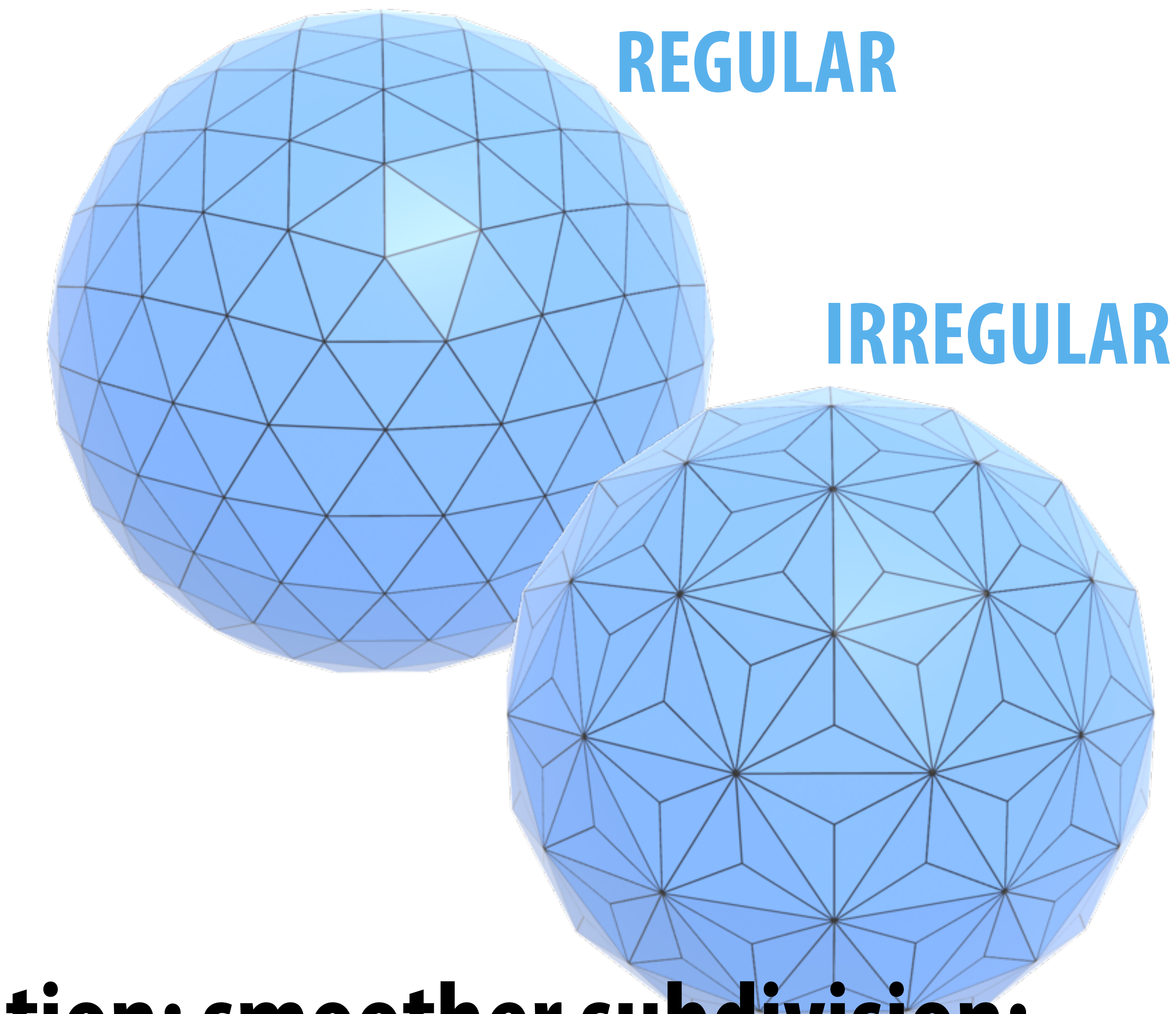
“GOOD”



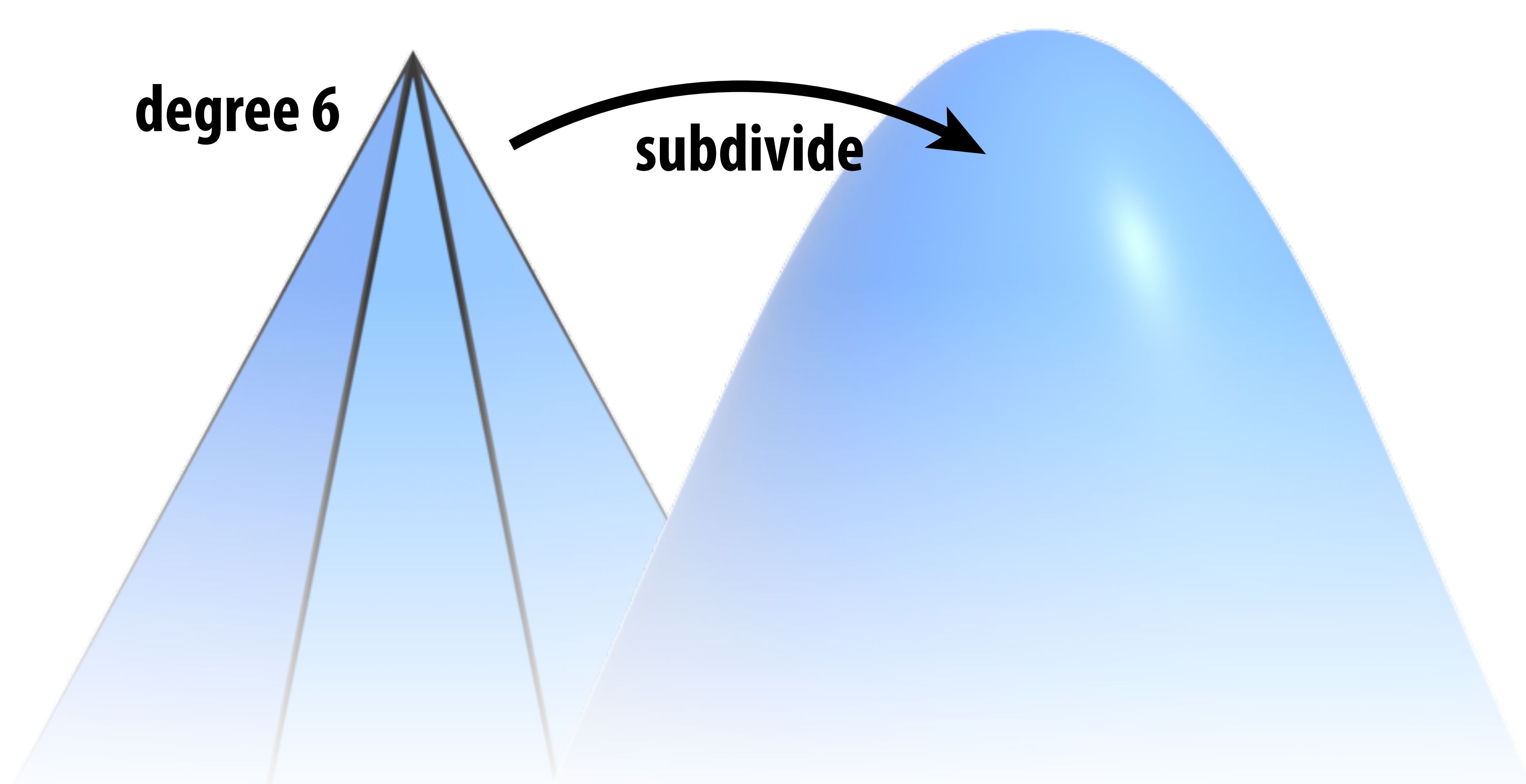
“OK”

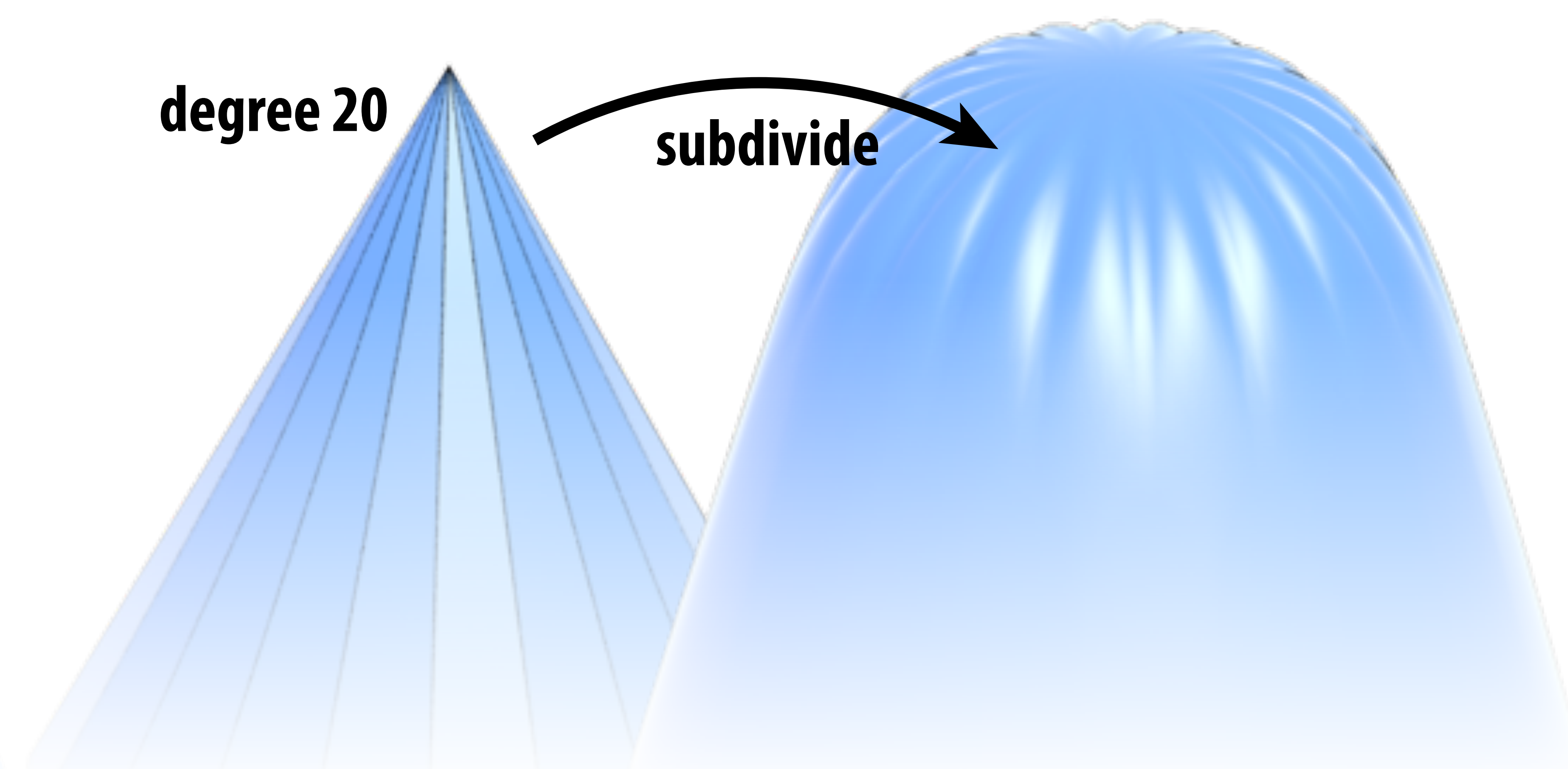


“BAD”



Why? Better polygon shape; more regular computation; smoother subdivision:

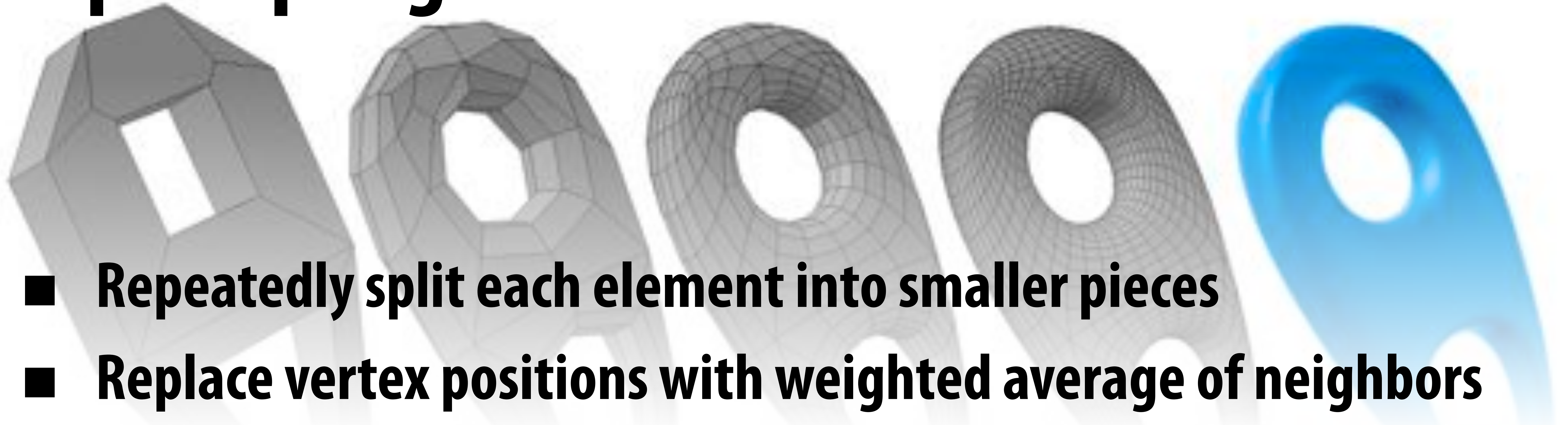




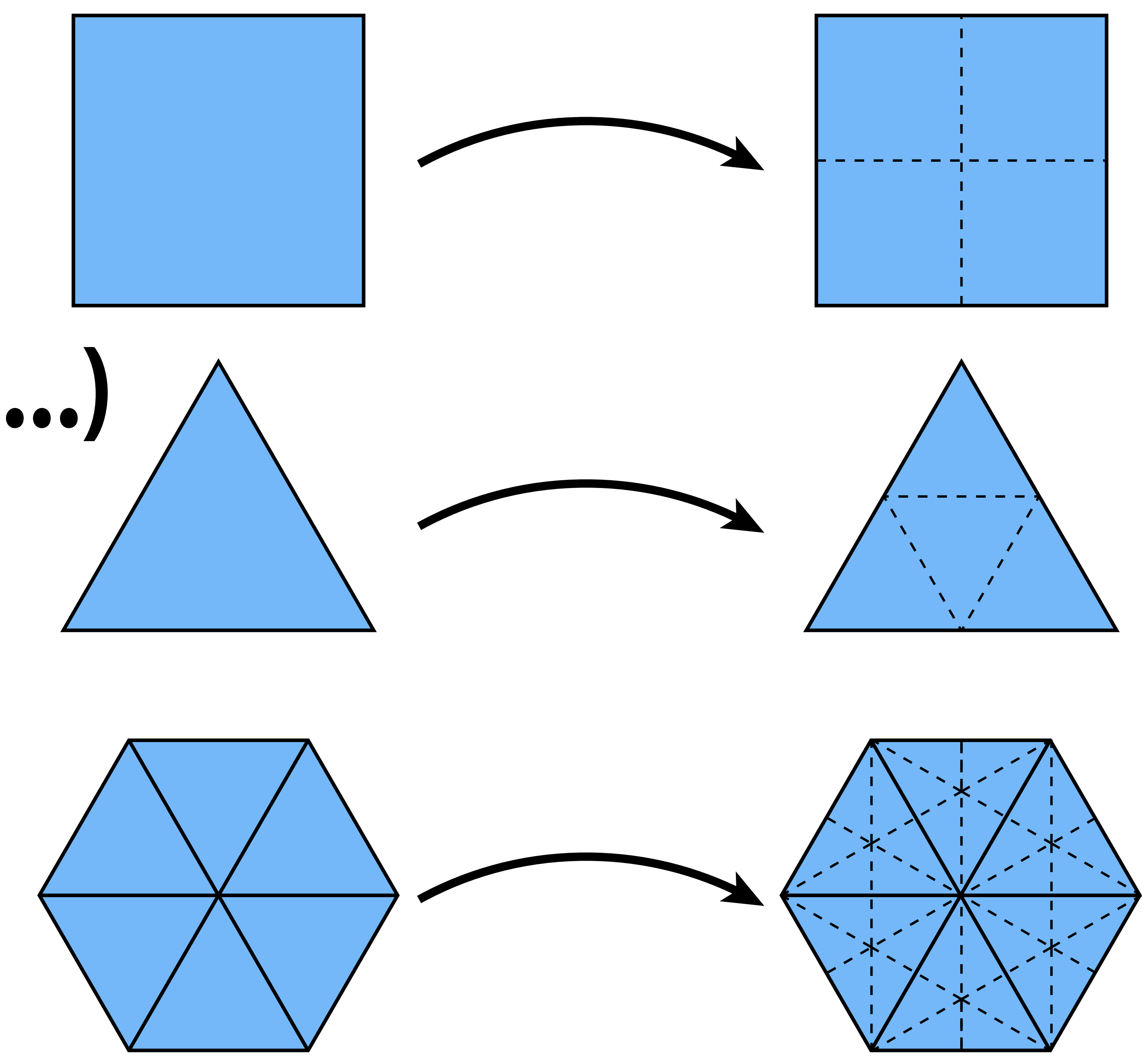
<u>Fact</u>: in general, can’t have regular vertex degree everywhere!

How do we upsample a mesh?

Upsampling via Subdivision

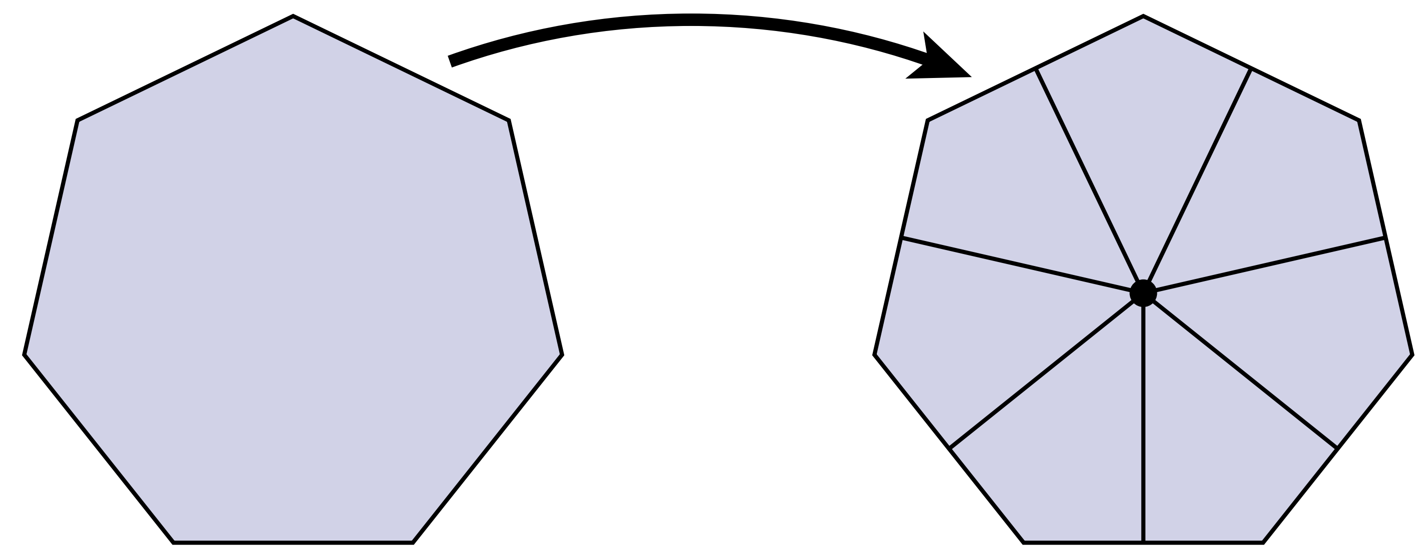


- Repeatedly split each element into smaller pieces
- Replace vertex positions with weighted average of neighbors
- Main considerations:
  - interpolating vs. approximating
  - limit surface continuity (C^1, C^2, ...)
  - behavior at irregular vertices
- Many options:
  - Quad: Catmull-Clark
  - Triangle: Loop, Butterfly, Sqrt(3)



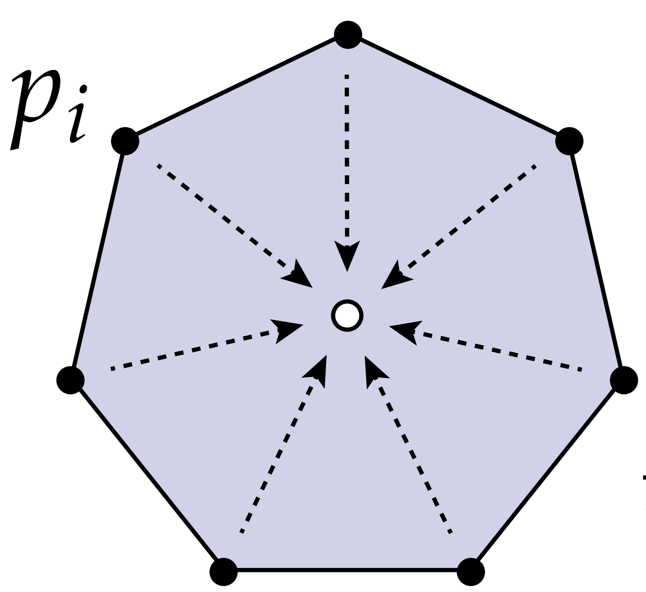
Catmull-Clark Subdivision

- **Step 0: split every polygon (any # of sides) into quadrilaterals:**



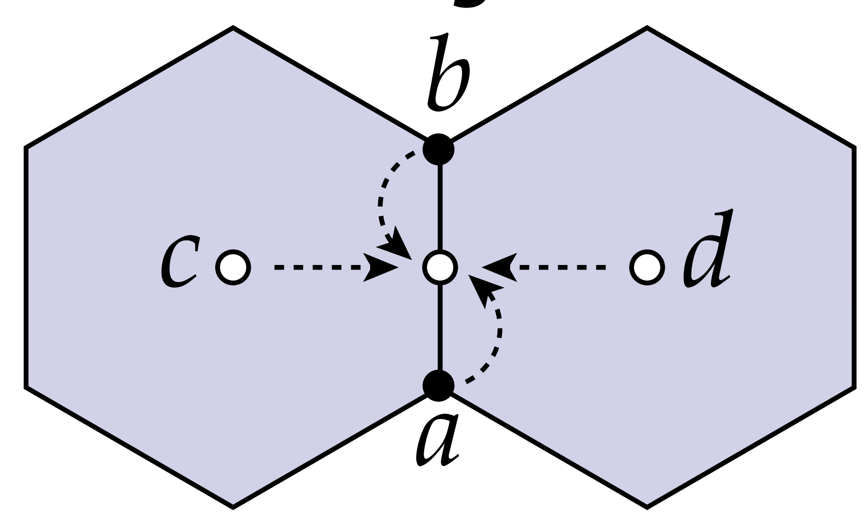
- **New vertex positions are weighted combination of old ones:**

STEP 1: Face coords



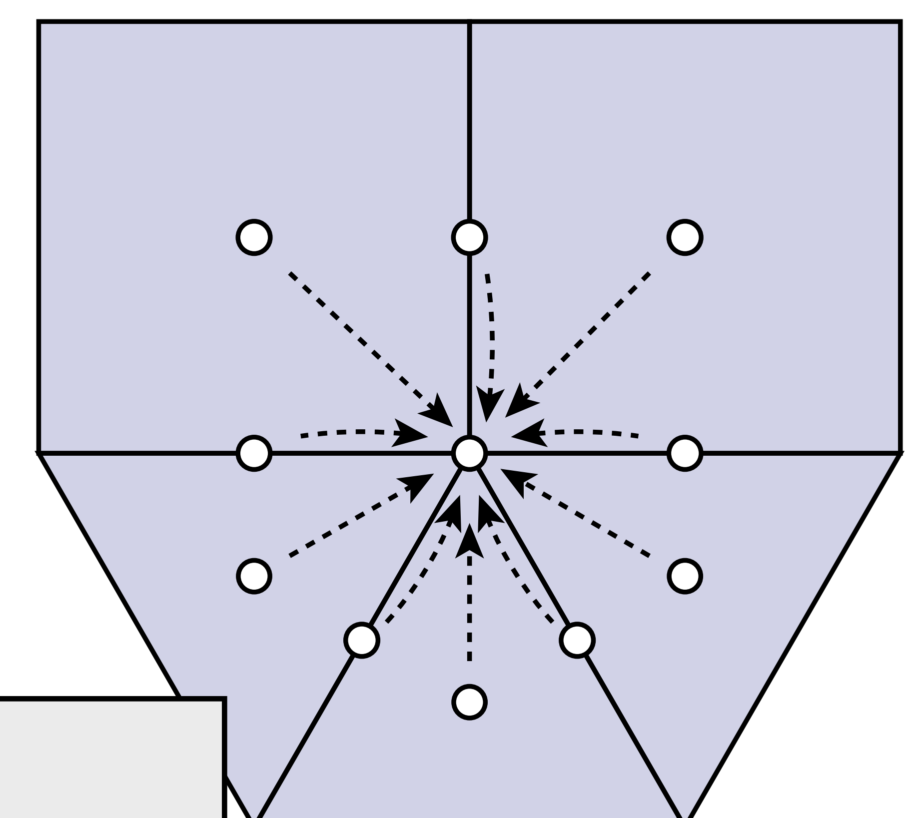
$\frac{1}{n}\sum_i p_i$

STEP 2: Edge coords



$(a+b+c+d)/4$

STEP 3: Vertex coords

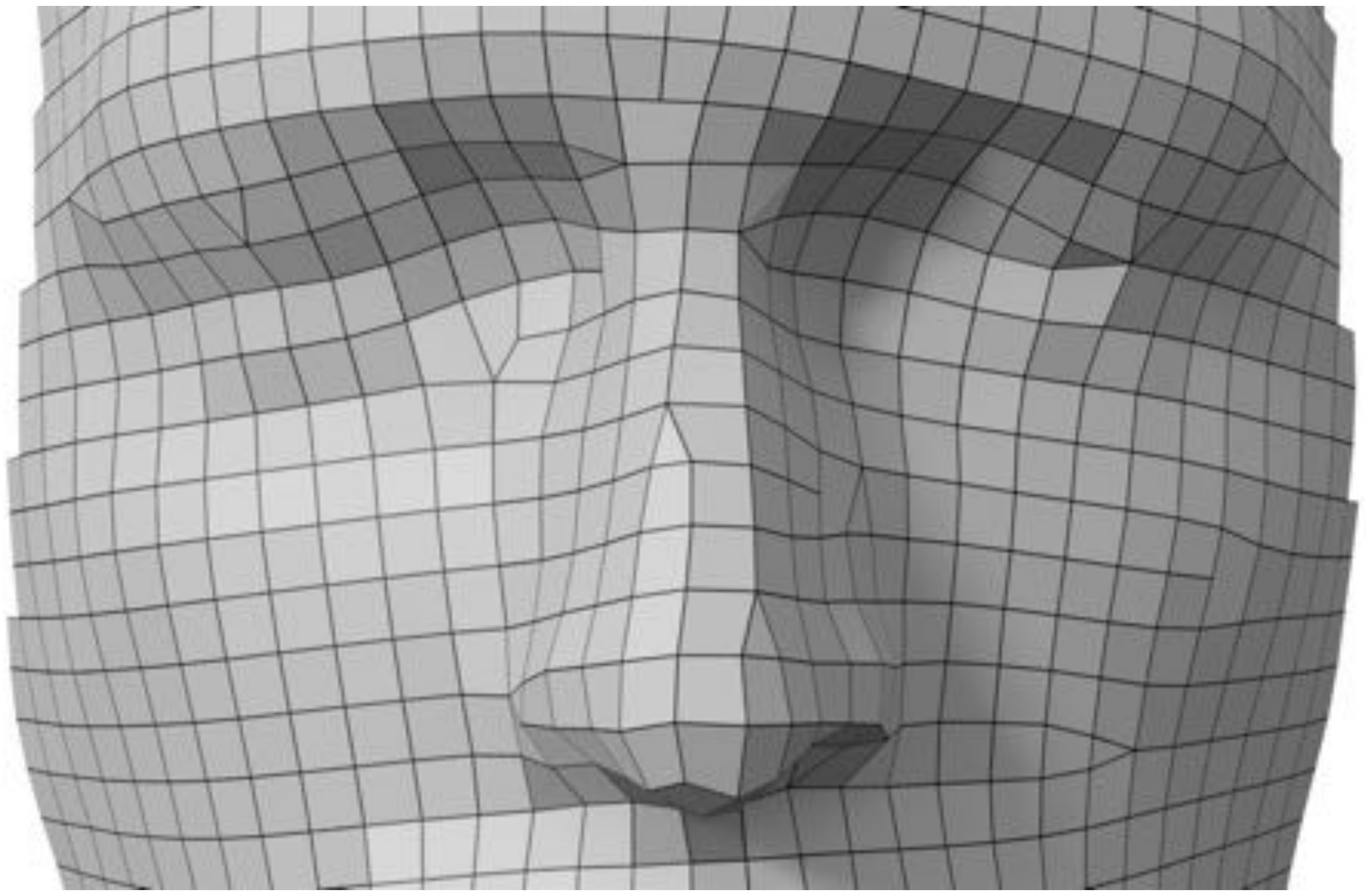


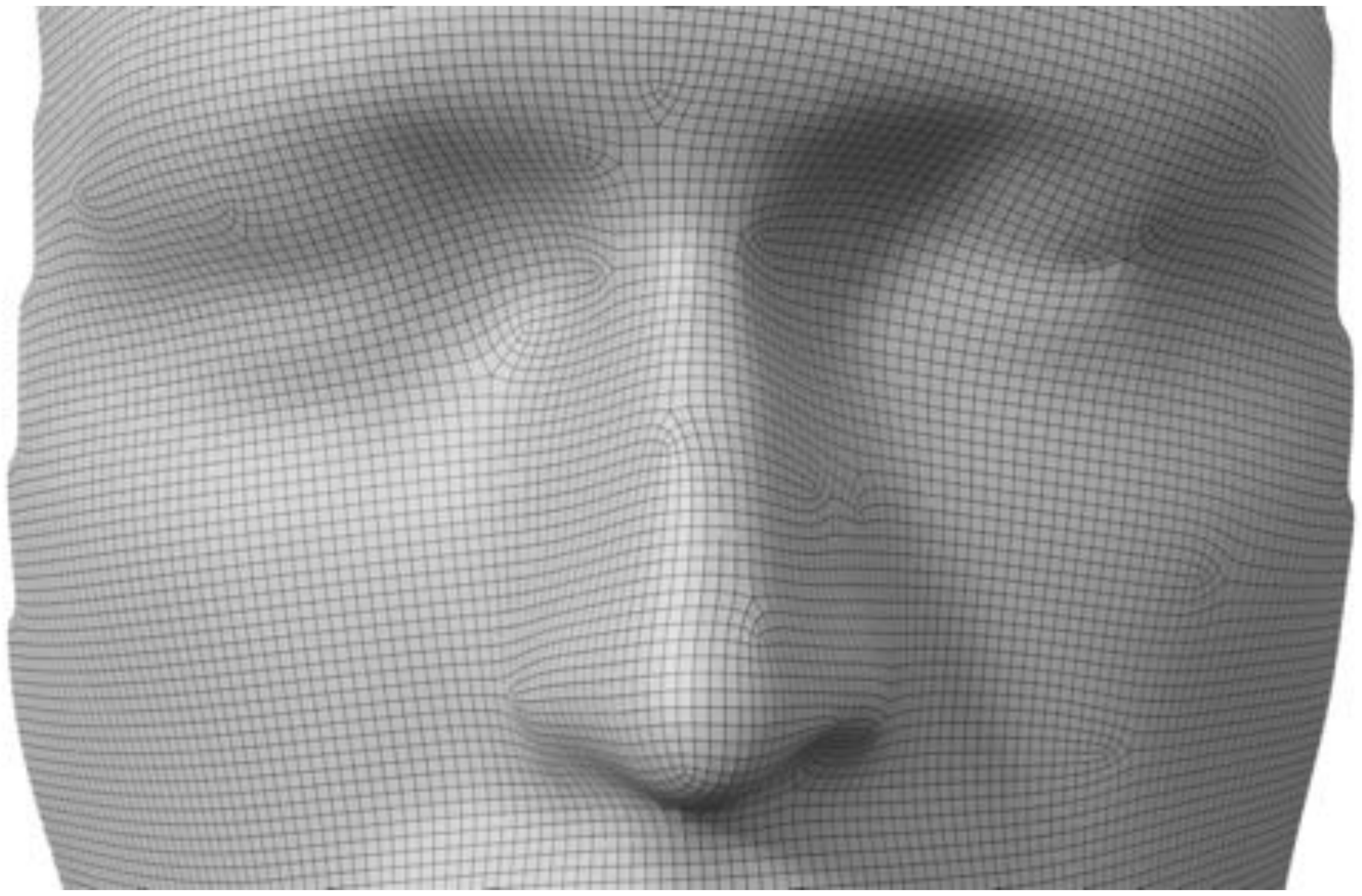
New vertex coords:

$$\frac{Q + 2R + (n-3)S}{n}$$

- n – **vertex degree**
- Q – **average of face coords around vertex**
- R – **average of edge coords around vertex**
- S – **original vertex position**

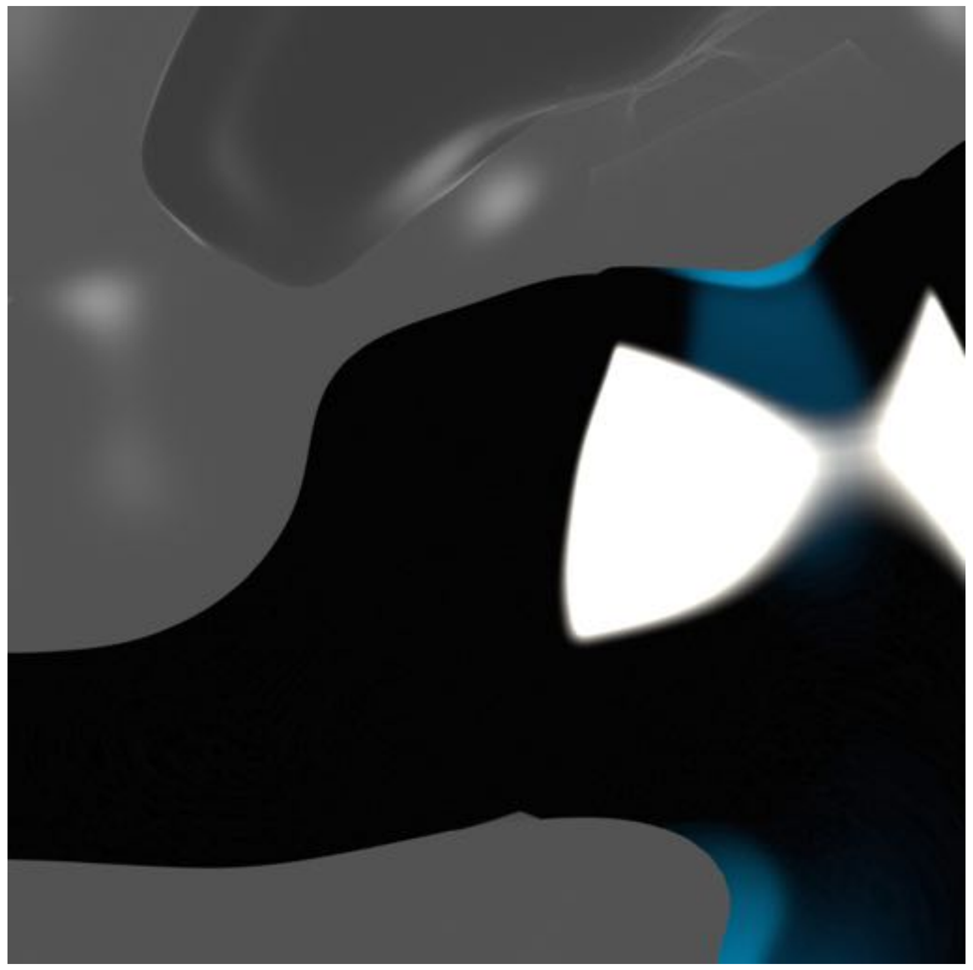
Catmull-Clark on quad mesh



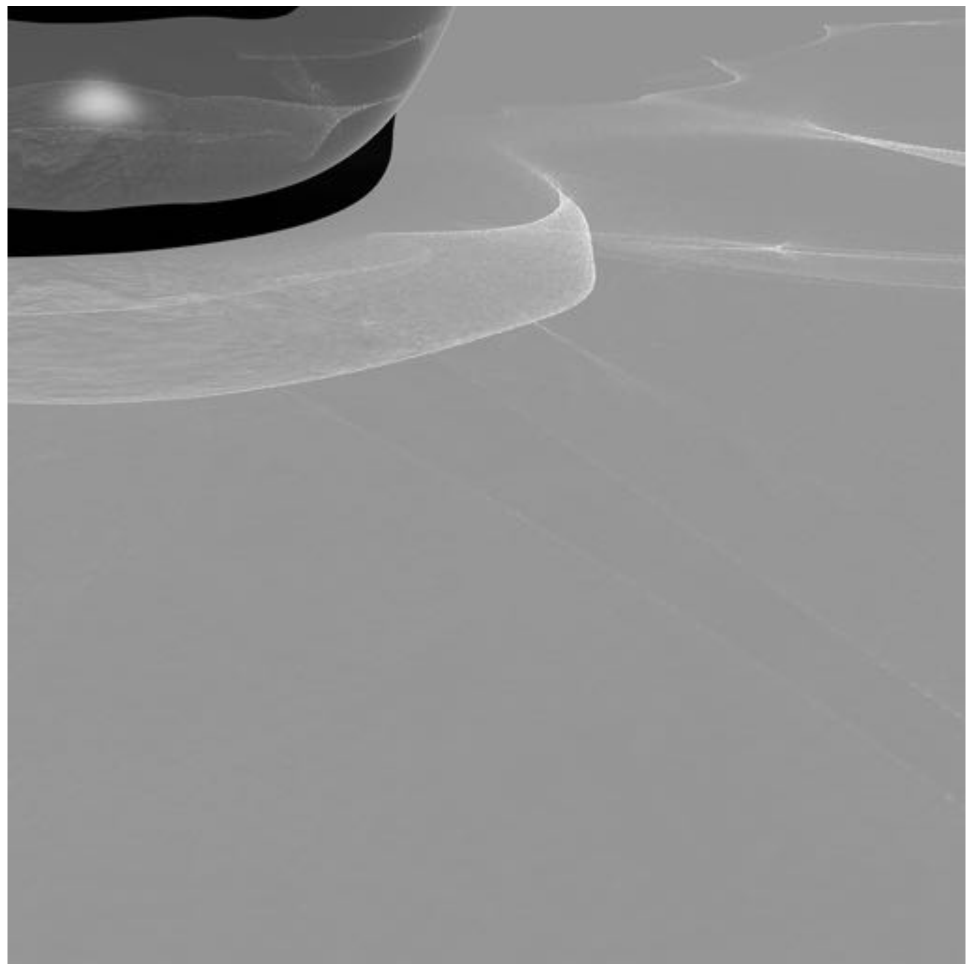


few irregular vertices

⟹ smoothly-varying surface normals

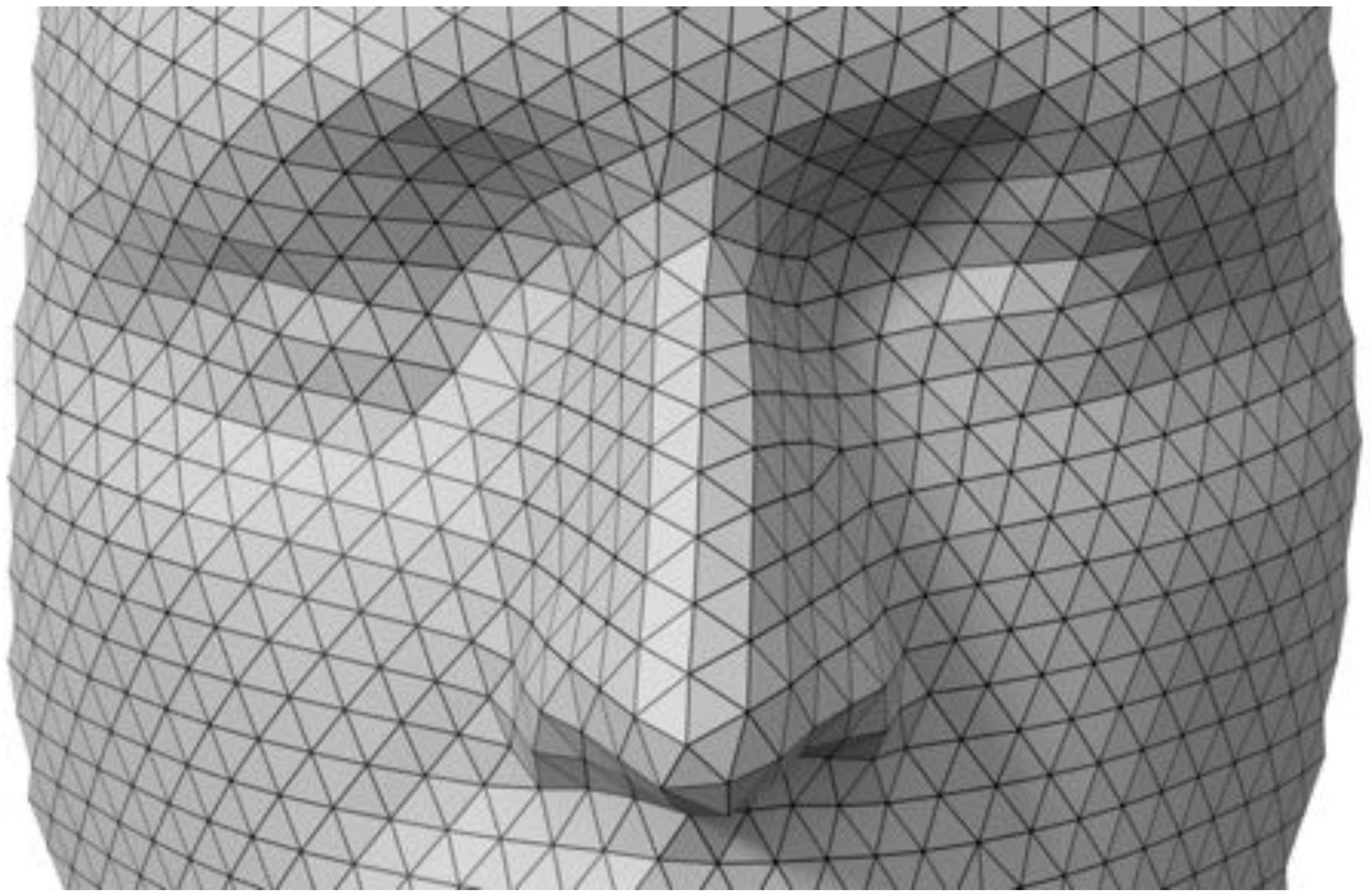


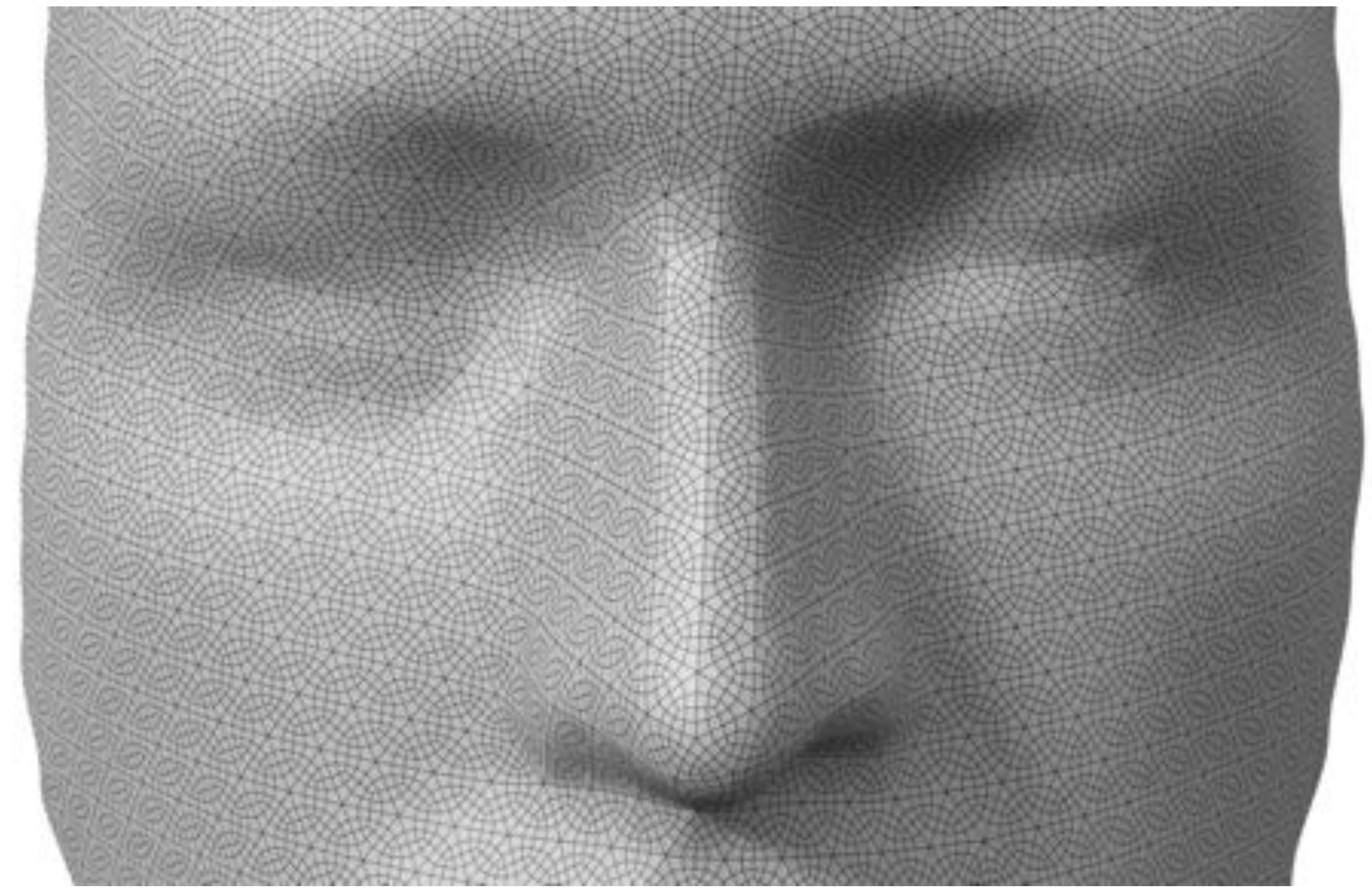
smooth
reflection lines



smooth
caustics

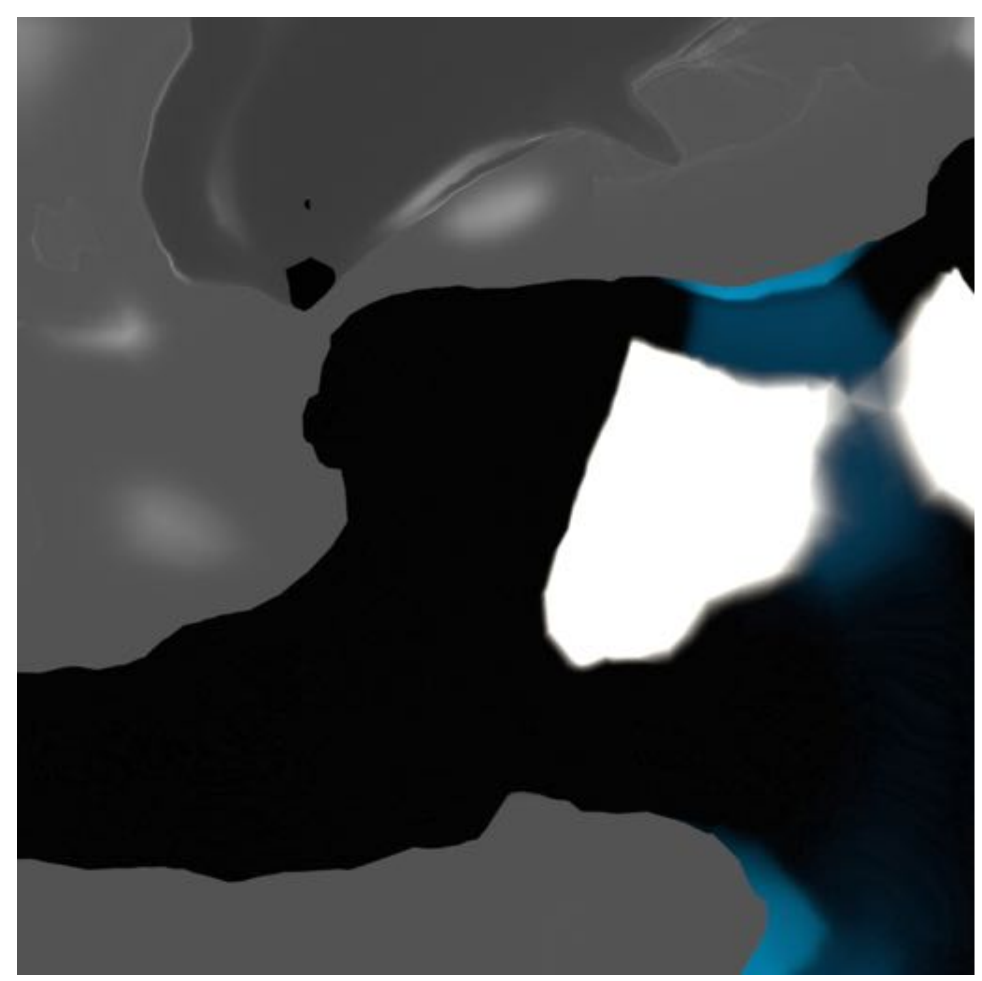
Catmull-Clark on triangle mesh



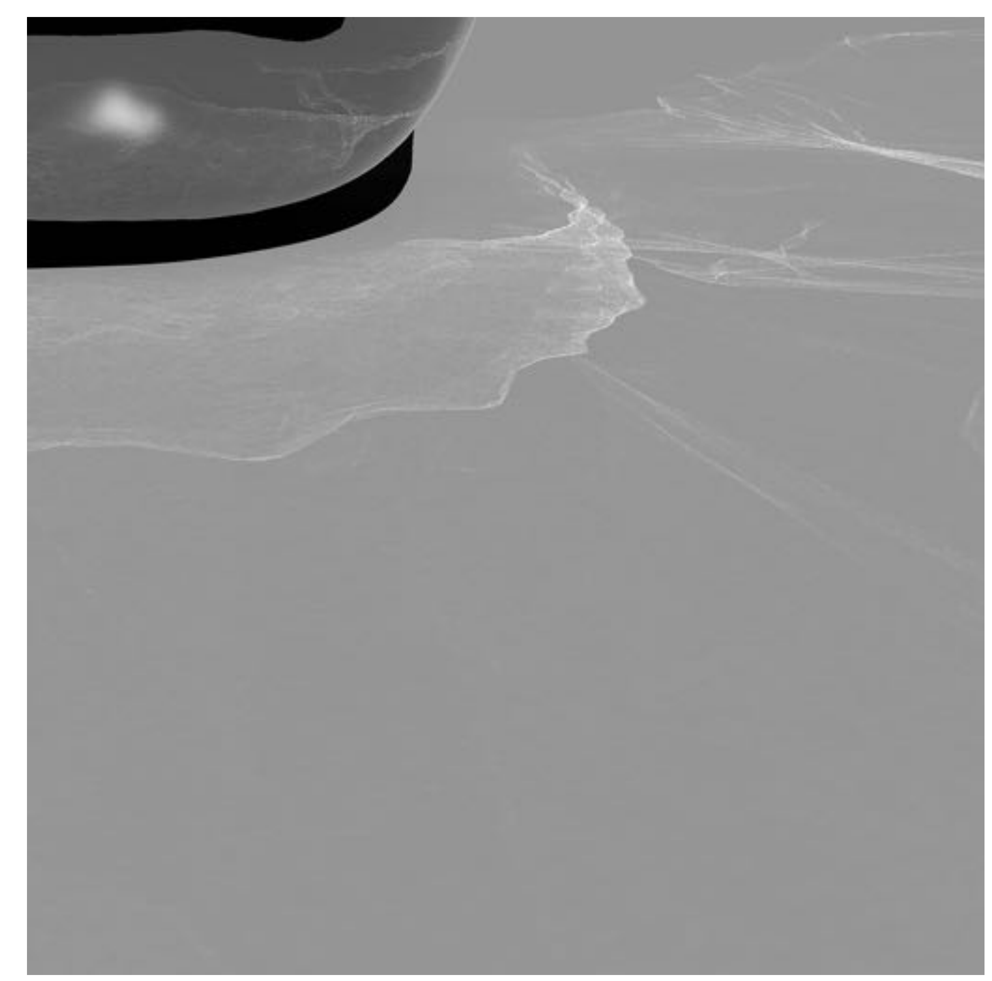


many irregular vertices

$\Longrightarrow$ **erratic surface normals**



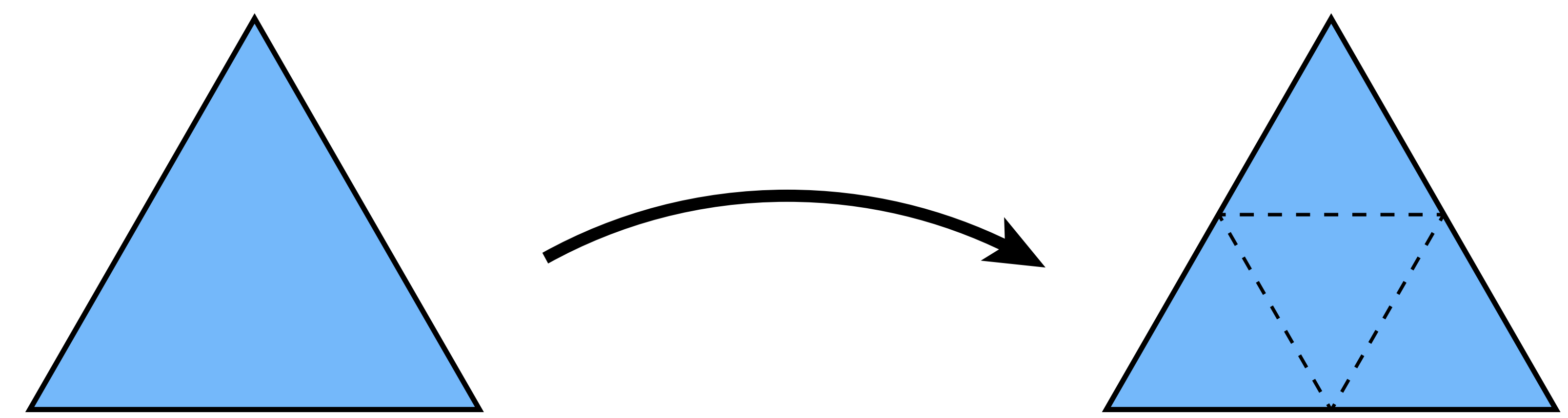
jagged
reflection lines

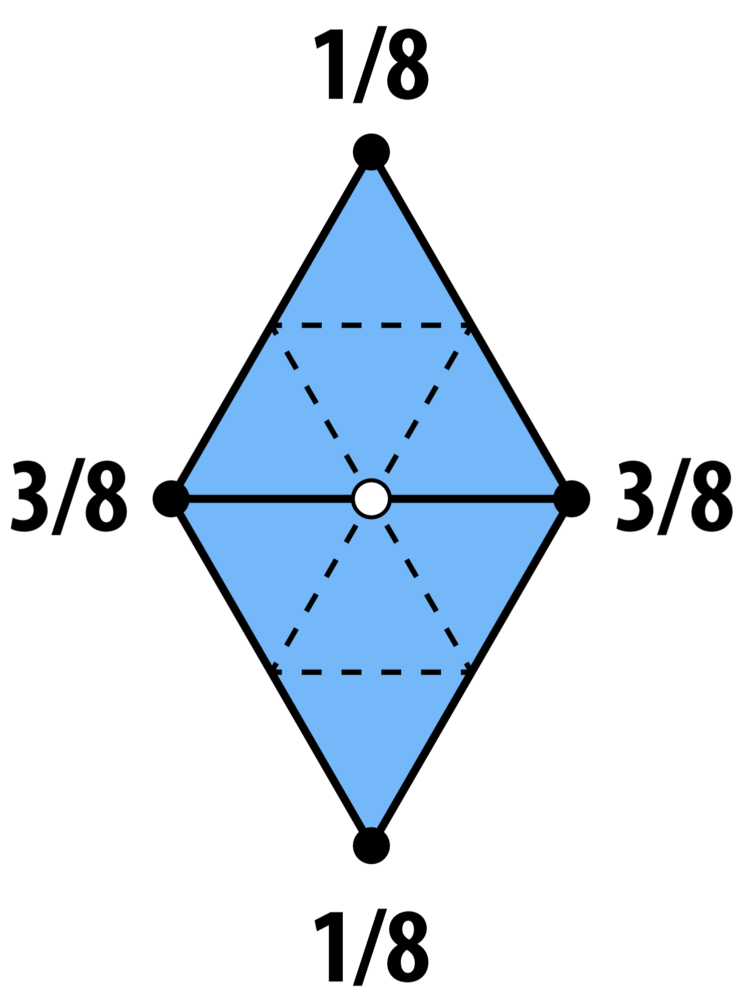


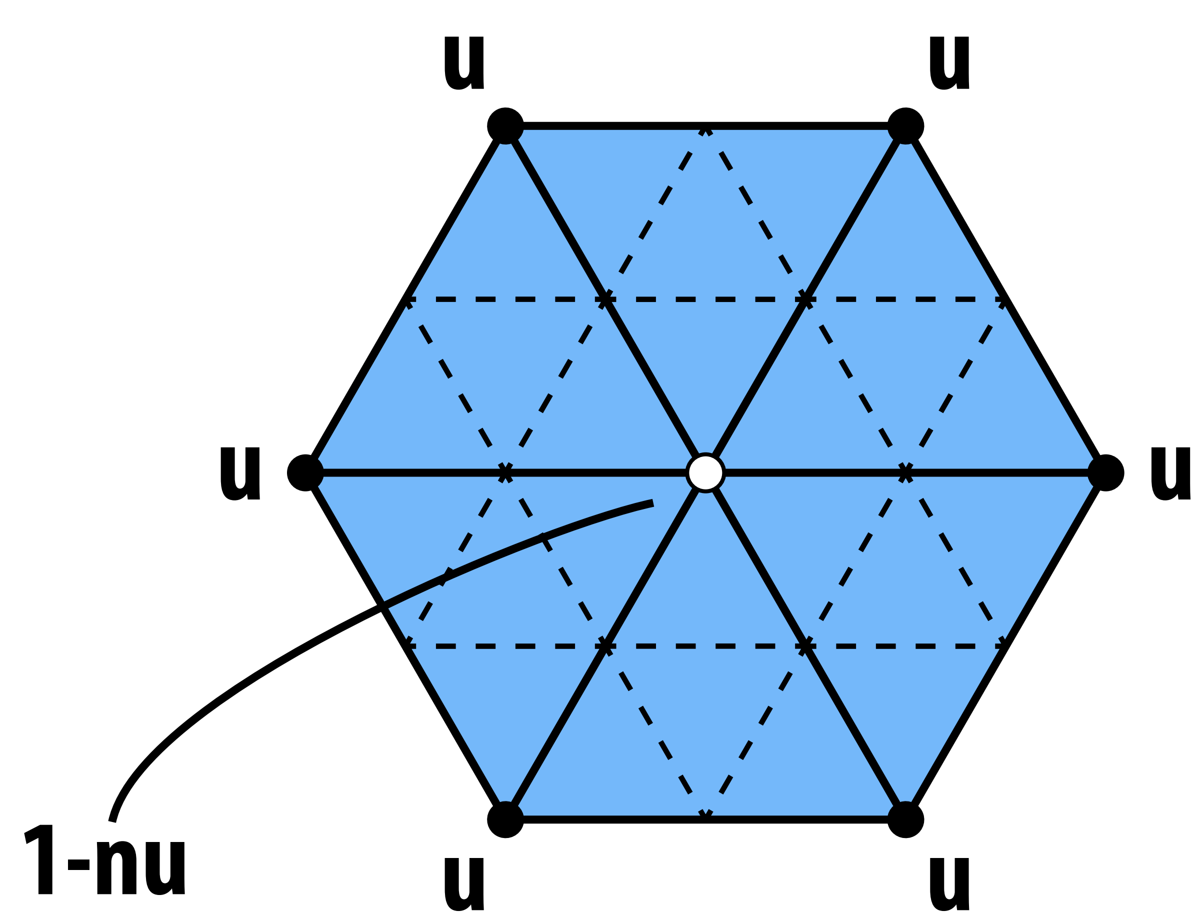
jagged
caustics

Loop Subdivision

- Alternative subdivision scheme for triangle meshes
- Curvature is continuous away from irregular vertices ("C^2")
- Algorithm:
  - Split each triangle into four
  - Assign new vertex positions according to weights:





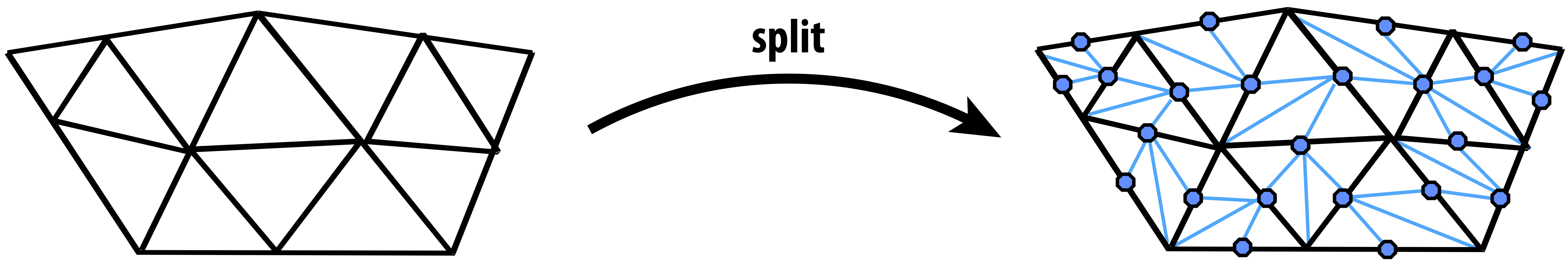


n: vertex degree

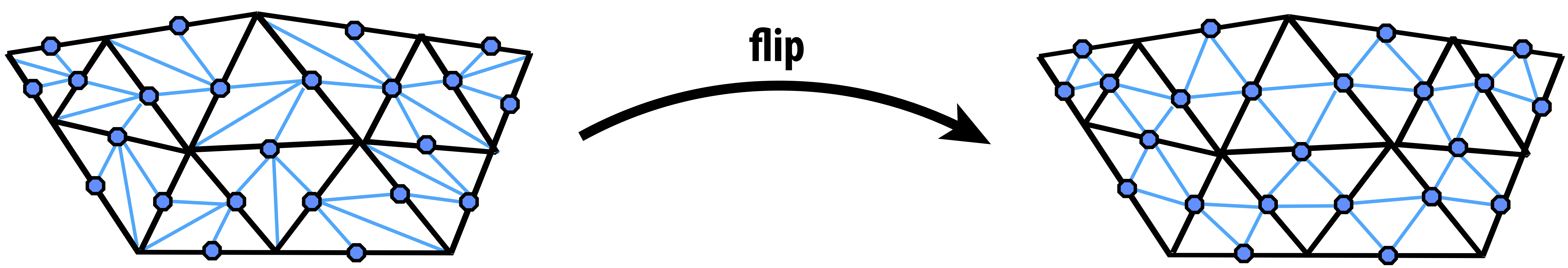
u: 3/16 if n=3, 3/(8n) otherwise

Loop Subdivision via Edge Operations

- First, split edges of original mesh in any order:



- Next, flip new edges that touch a new & old vertex:

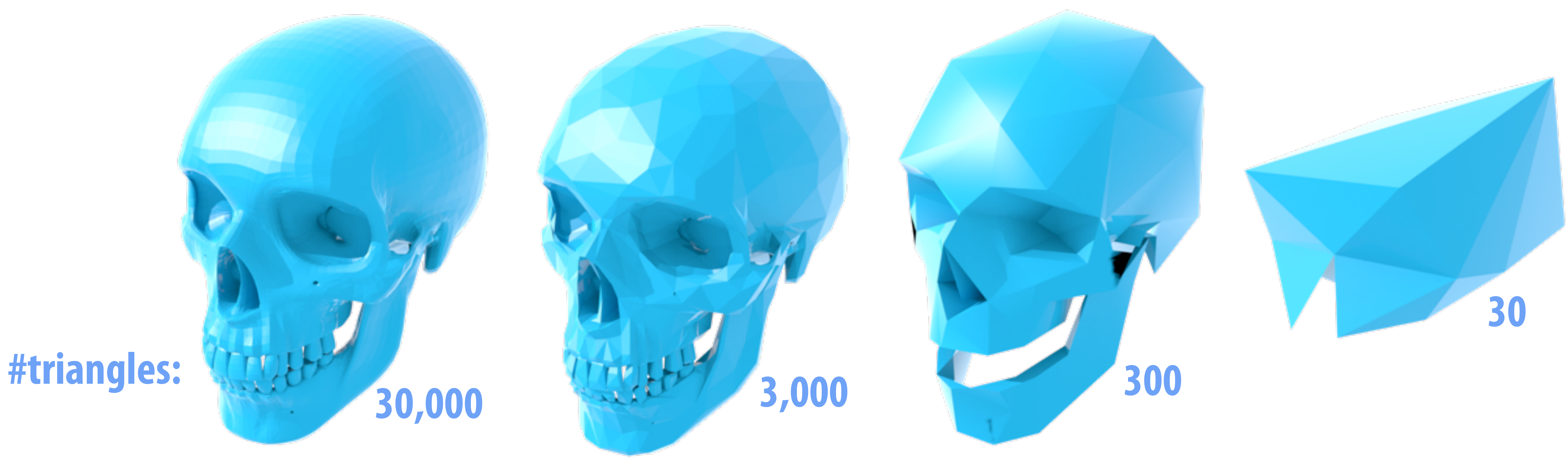


(Don't forget to update vertex positions!)

What if we want fewer triangles?

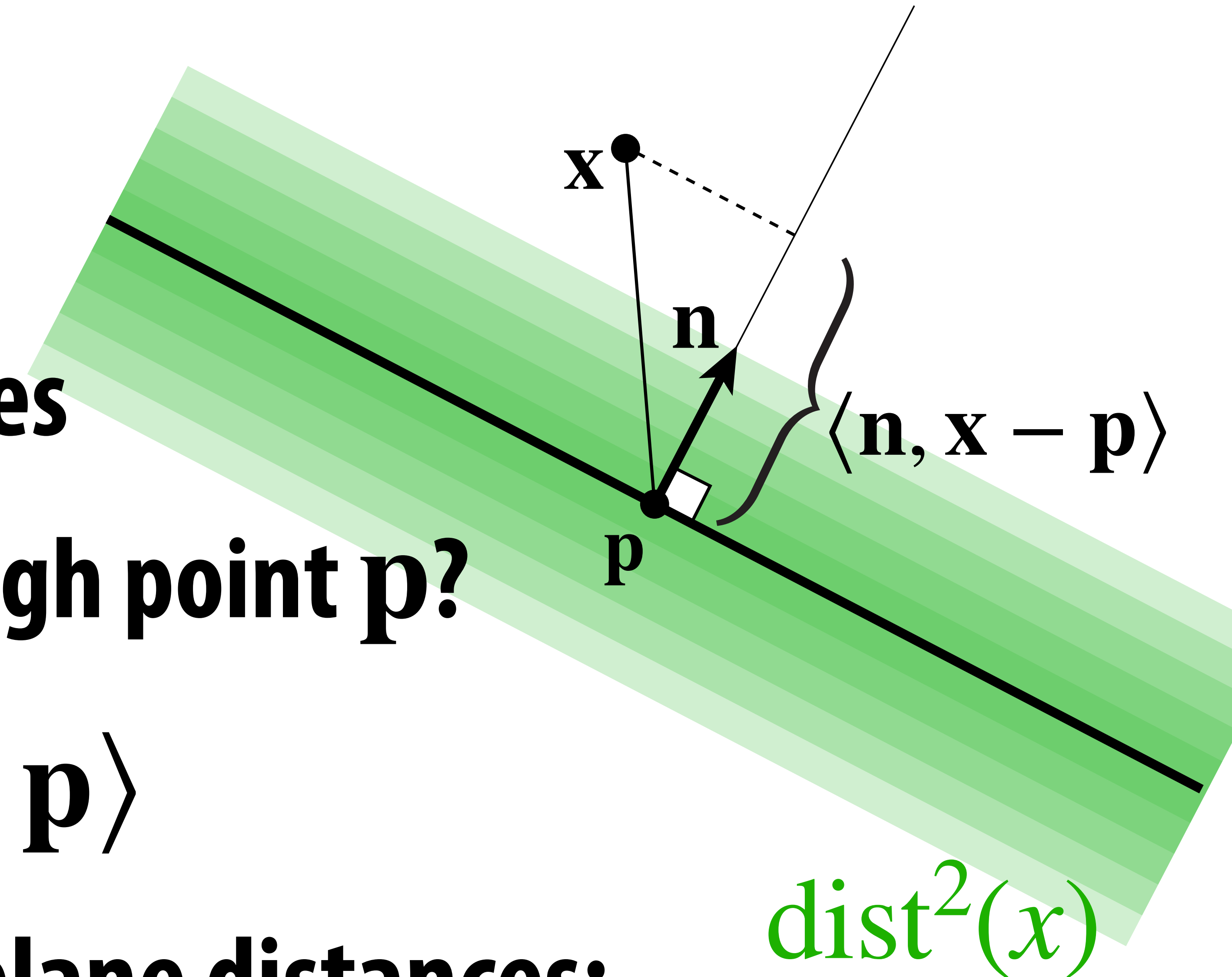
Simplification via Edge Collapse

- One popular scheme: iteratively collapse edges
- Greedy algorithm:
  - assign each edge a cost
  - collapse edge with least cost
  - repeat until target number of elements is reached
- Particularly effective cost function: quadric error metric*

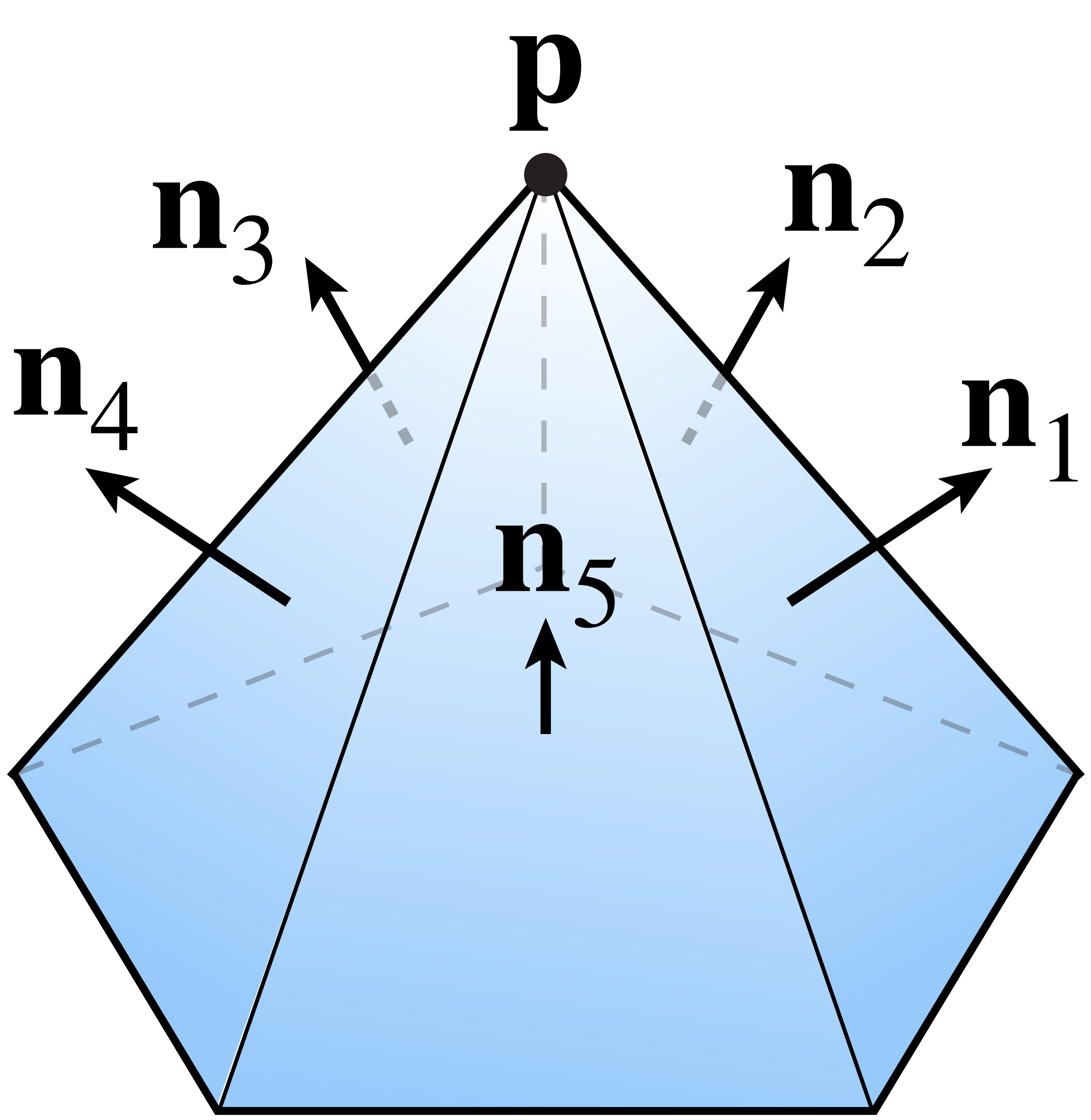


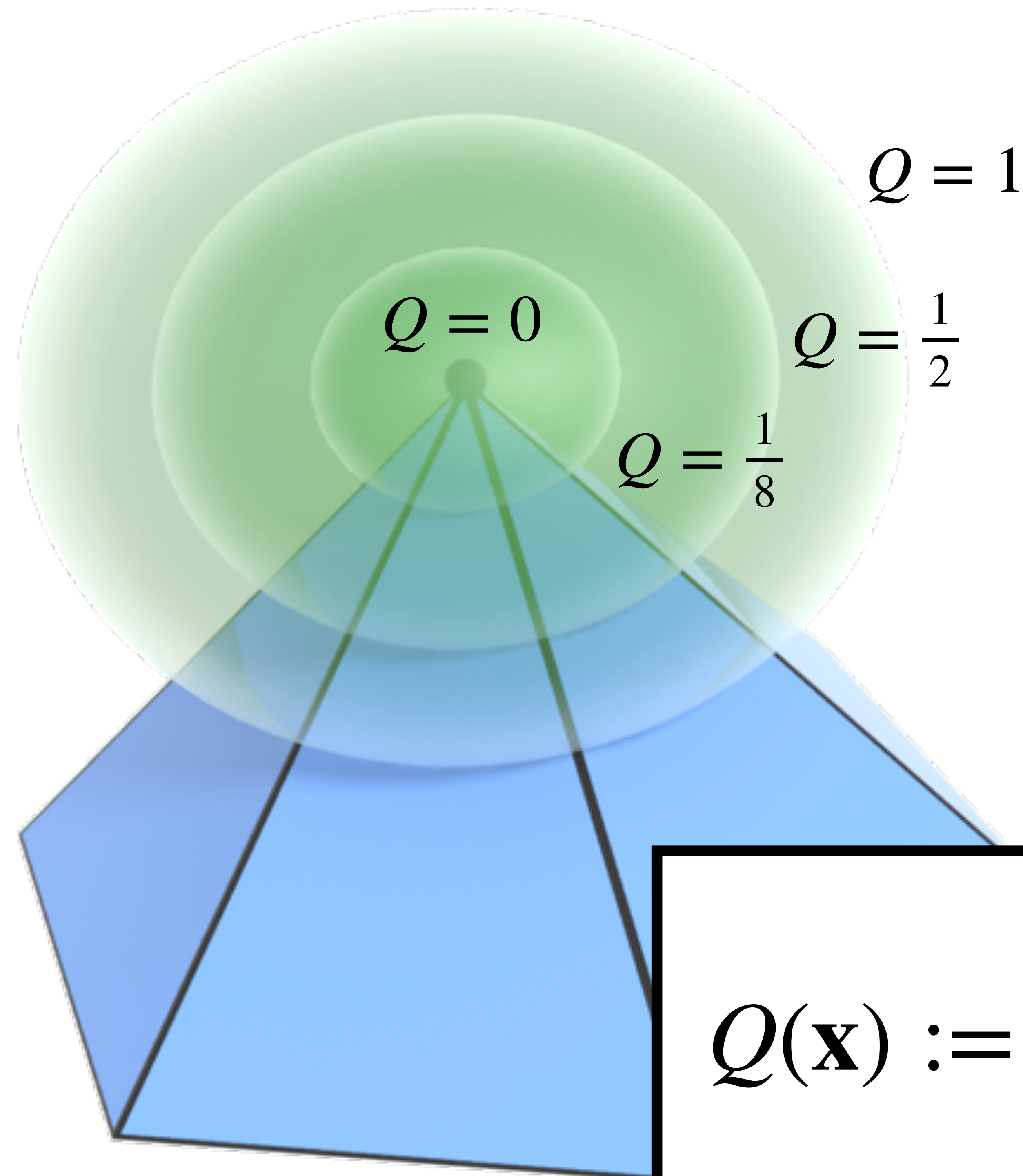
*invented at CMU (Garland & Heckbert 1997)

Quadric Error Metric



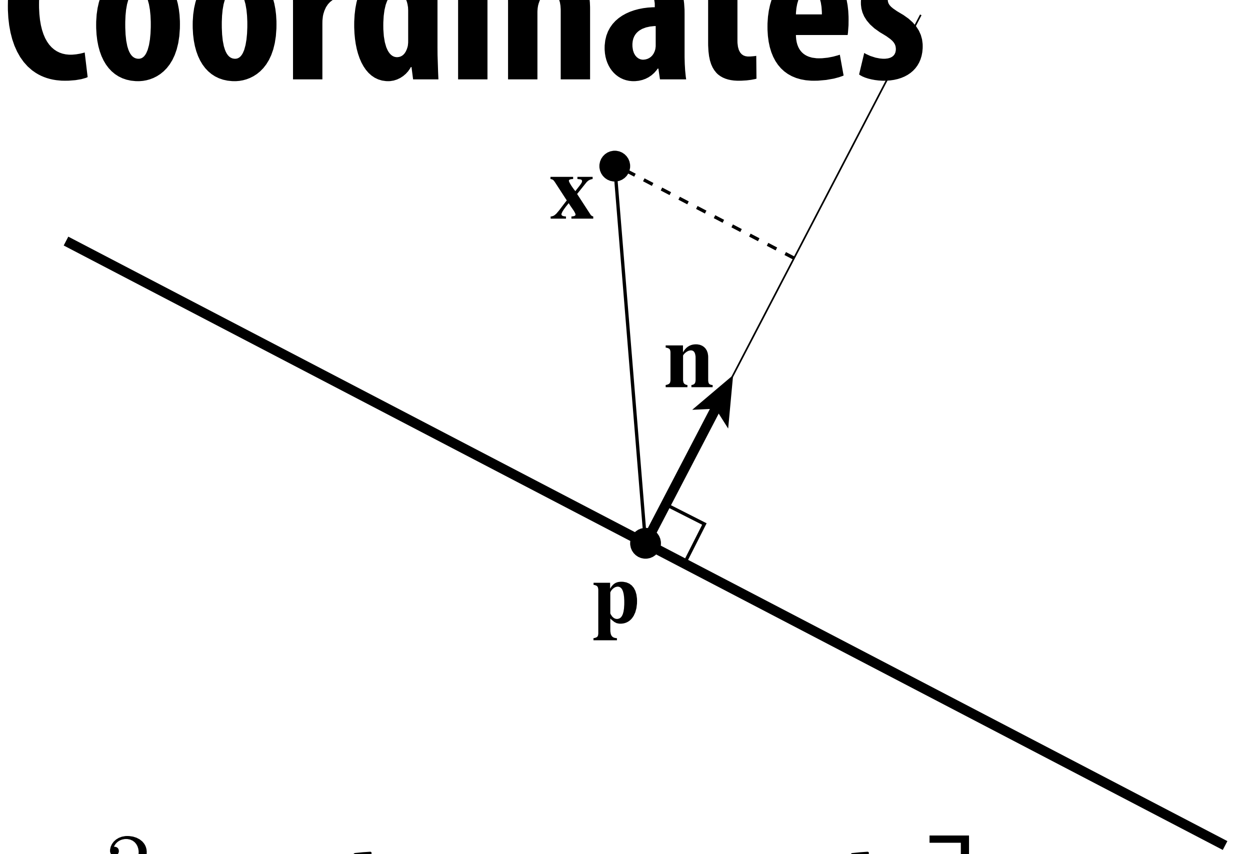
- Approximate distance to a collection of triangles
- Q: Distance to plane w/ normal $\mathbf{n}$ passing through point $\mathbf{p}$?
- A: $\mathrm{dist}(\mathbf{x}) = \langle \mathbf{n}, \mathbf{x} \rangle - \langle \mathbf{n}, \mathbf{p} \rangle = \langle \mathbf{n}, \mathbf{x} - \mathbf{p} \rangle$
- Quadric error is then sum of squared point-to-plane distances:





$$Q(\mathbf{x}) := \sum_{i=1}^{k} \langle \mathbf{n}_i, \mathbf{x} - \mathbf{p} \rangle^2$$

Quadric Error - Homogeneous Coordinates



- Suppose in coordinates we have
  - a query point $\mathbf{x} = (x, y, z)$
  - a normal $\mathbf{n} = (a, b, c)$
  - an offset $d := -\langle \mathbf{n}, \mathbf{p} \rangle$
- In homogeneous coordinates, let
  - $\mathbf{u} := (x, y, z, 1)$
  - $\mathbf{v} := (a, b, c, d)$

$$K = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

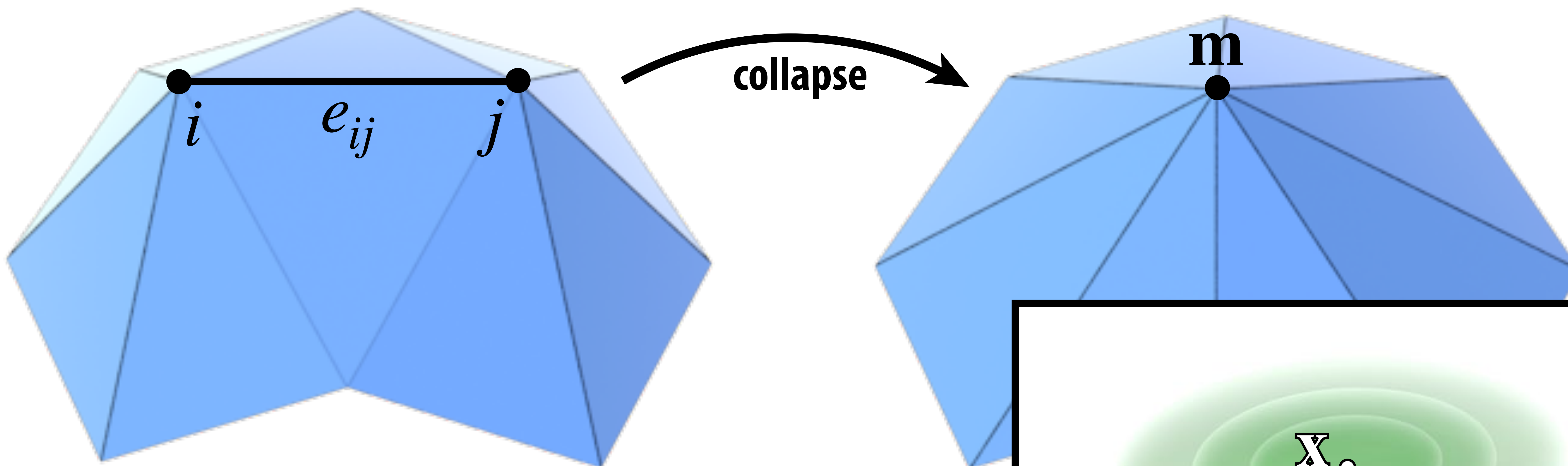
- Signed distance to plane is then just $\langle \mathbf{u}, \mathbf{v} \rangle = ax + by + cz + d$
- Squared distance is $\langle \mathbf{u}, \mathbf{v} \rangle^2 = \mathbf{u}^\top (\mathbf{v}\mathbf{v}^\top)\mathbf{u} =: \mathbf{u}^\top K \mathbf{u}$
- Matrix $K = \mathbf{v}\mathbf{v}^T$ encodes squared distance to plane

Key idea: <u>sum</u> of matrices $K \iff$ distance to <u>union</u> of planes

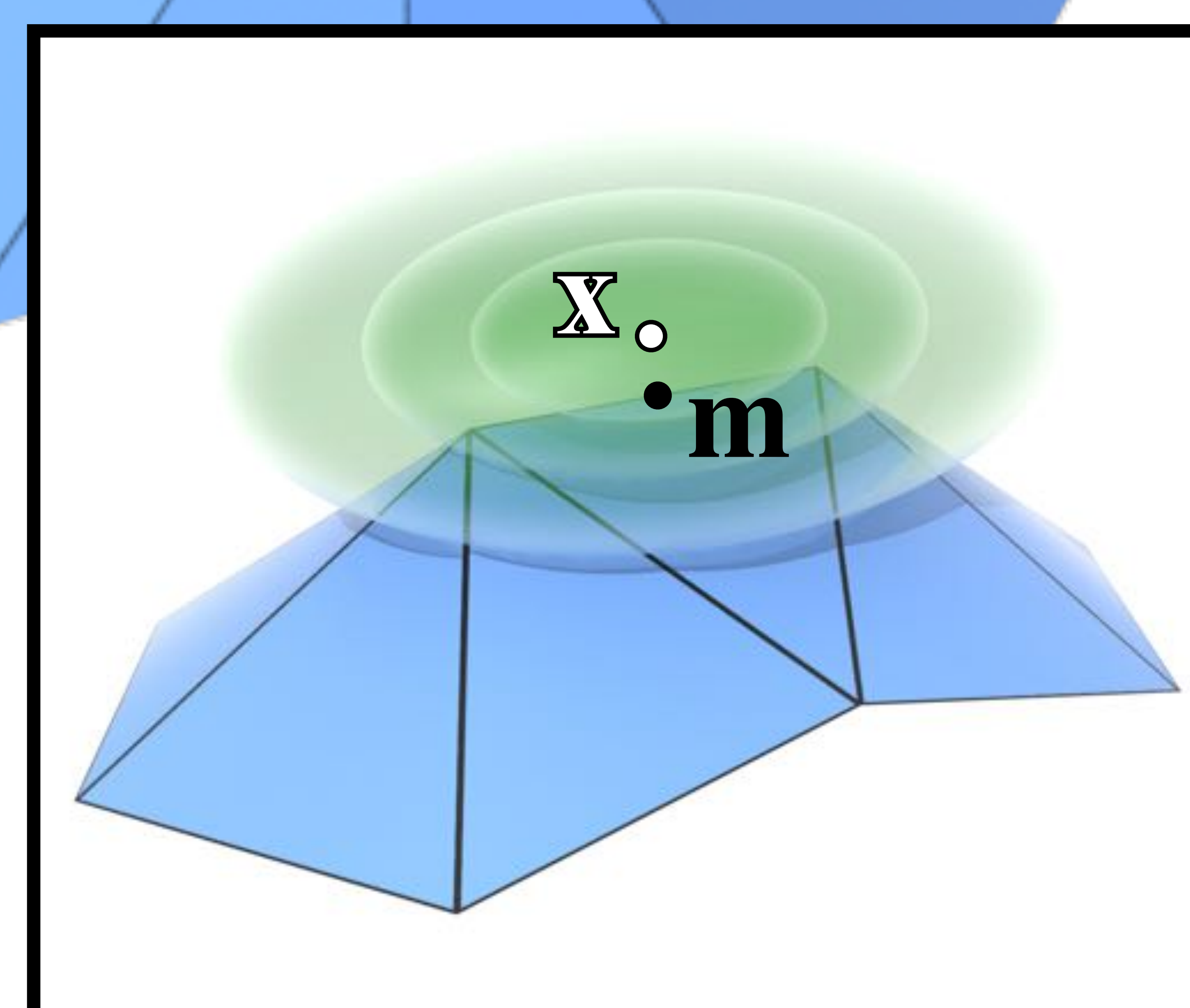
$$\mathbf{u}^\top K_1 \mathbf{u} + \mathbf{u}^\top K_2 \mathbf{u} = \mathbf{u}^\top (K_1 + K_2)\mathbf{u}$$

Quadric Error of Edge Collapse

- How much does it cost to collapse an edge e_{ij}?
- Idea: compute midpoint $\mathbf{m}$, measure error $Q(\mathbf{m}) = \mathbf{m}^\mathsf{T}(K_i + K_j)\mathbf{m}$
- Error becomes "score" for e_{ij}, determining priority

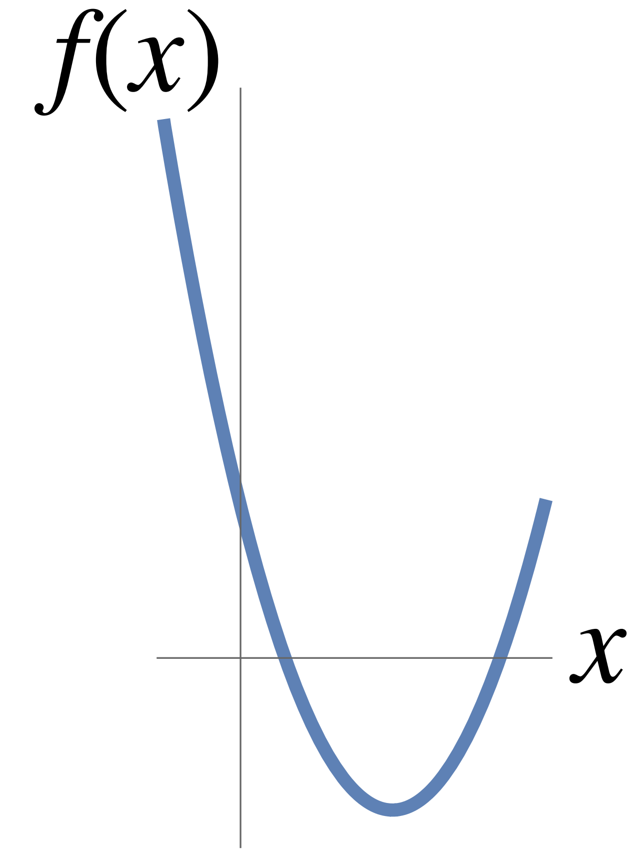


- Better idea: find point $\mathbf{x}$ that minimizes error!
- Ok, but how do we minimize quadric error?

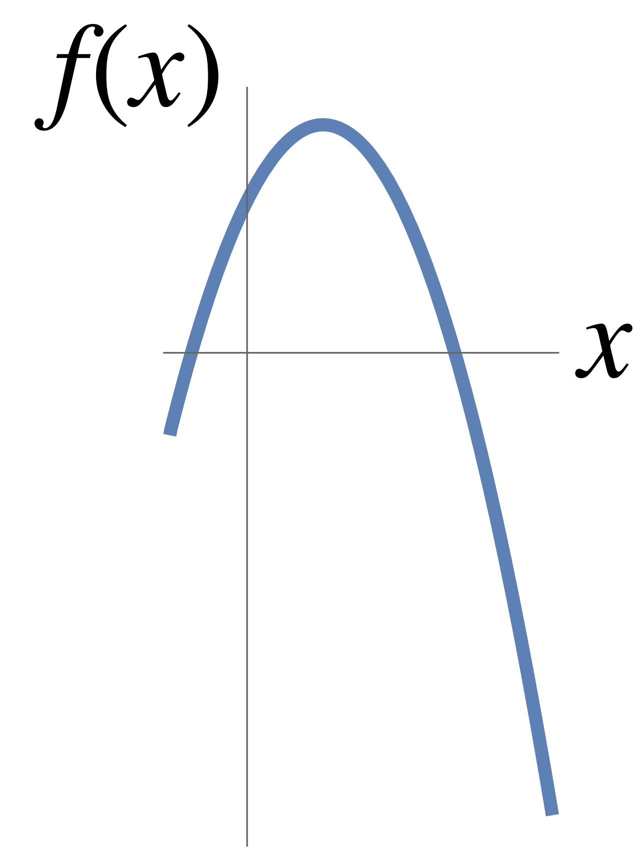


Review: Minimizing a Quadratic Function

- Suppose you have a function $f(x) = ax^2 + bx + c$
- Q: What does the graph of this function look like?



- Could also look like this!



- Q: How do we find the minimum?
- A: Find where the function looks "flat" if we zoom in really close
- I.e., find point x where 1st derivative vanishes:

$$f'(x) = 0$$

$$2ax + b = 0$$

$$x = -b/2a$$

(What does x describe for the second function?)

Minimizing Quadratic Polynomial

- Not much harder to minimize a quadratic polynomial in n variables
- Can always write in terms of a symmetric matrix A
- E.g., in 2D: $f(x,y) = ax^2 + bxy + cy^2 + dx + ey + g$

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} \quad A = \begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} d \\ e \end{bmatrix}$$

$$f(x,y) = \mathbf{x}^\mathsf{T} A \mathbf{x} + \mathbf{u}^\mathsf{T}\mathbf{x} + g$$

(will have this same form for any n)

- Q: How do we find a critical point (min/max/saddle)?
- A: Set derivative to zero!

$$2A\mathbf{x} + \mathbf{u} = 0$$

$$\mathbf{x} = -\frac{1}{2}A^{-1}\mathbf{u}$$

(compare with our 1D solution)

$$x = -b/2a$$

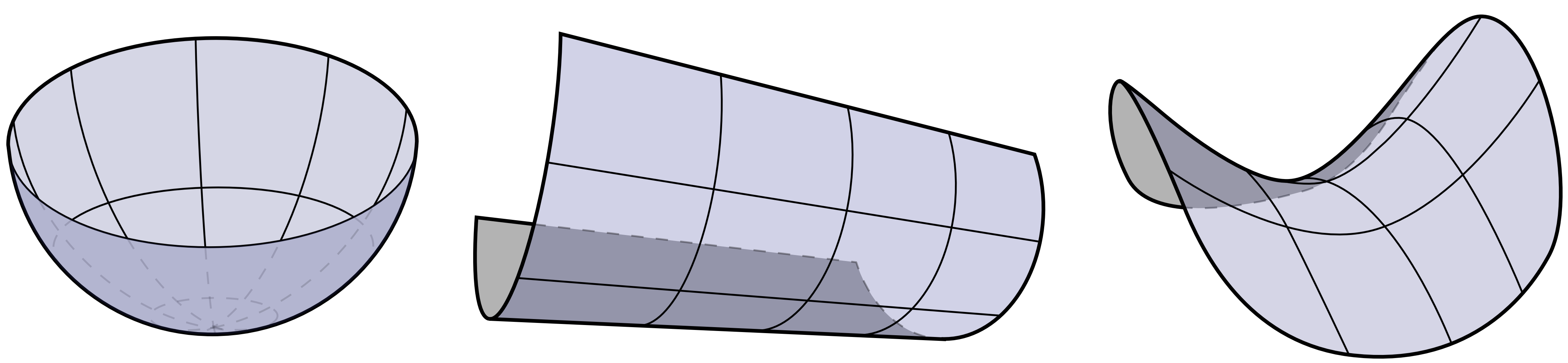
(Can you show this is true, at least in 2D?)

Positive Definite Quadratic Form

- **Just like our 1D parabola, critical point is not always a min!**
- **Q: In 2D, 3D, nD, when do we get a minimum?**
- **A: When matrix A is positive-definite:**

$$\mathbf{x}^{\mathsf{T}} A\, \mathbf{x} > 0 \quad \forall\, \mathbf{x}$$

- **1D: Must have $xax = ax^2 > 0$. In other words: a is positive!**
- **2D: Graph of function looks like a "bowl":**



positive definite | positive semidefinite | indefinite

Positive-definiteness extremely important in computer graphics: means we can find minimizers by solving linear equations. Starting point for many algorithms (geometry processing, simulation, ...)

Minimizing Quadric Error

- Find "best" point for edge collapse by minimizing quadratic form

$$\min_{\mathbf{u}\in\mathbb{R}^4} \mathbf{u}^T K \mathbf{u}$$

- Already know fourth (homogeneous) coordinate for a point is 1
- So, break up our quadratic function into two pieces:

$$\begin{bmatrix} \mathbf{x}^\mathsf{T} & 1 \end{bmatrix} \begin{bmatrix} B & \mathbf{w} \\ \mathbf{w}^\mathsf{T} & d^2 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$$

$$= \mathbf{x}^\mathsf{T} B \mathbf{x} + 2\mathbf{w}^\mathsf{T}\mathbf{x} + d^2$$

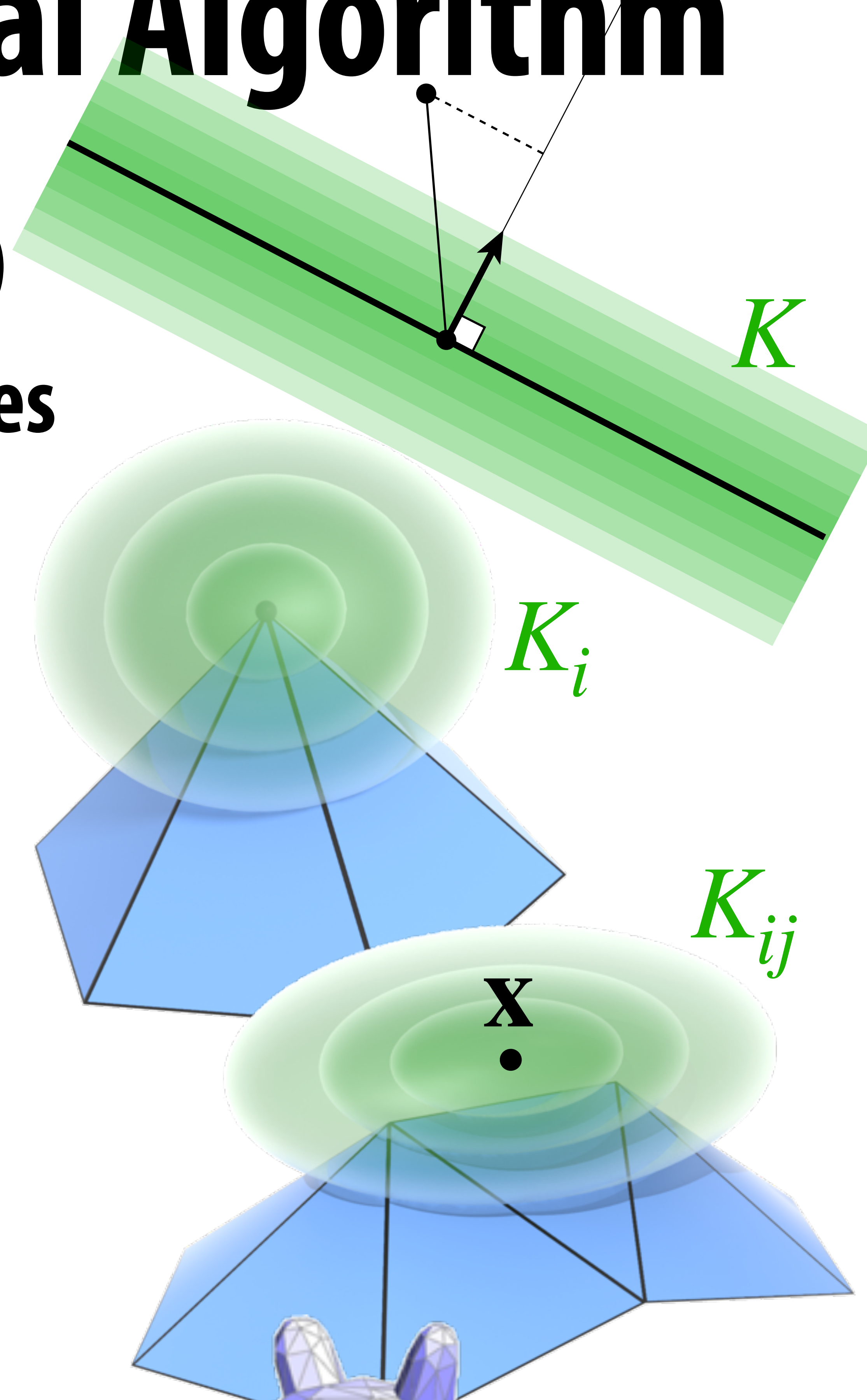
- Now we have a quadratic polynomial in the unknown position $\mathbf{x} \in \mathbb{R}^3$
- Can minimize as before:

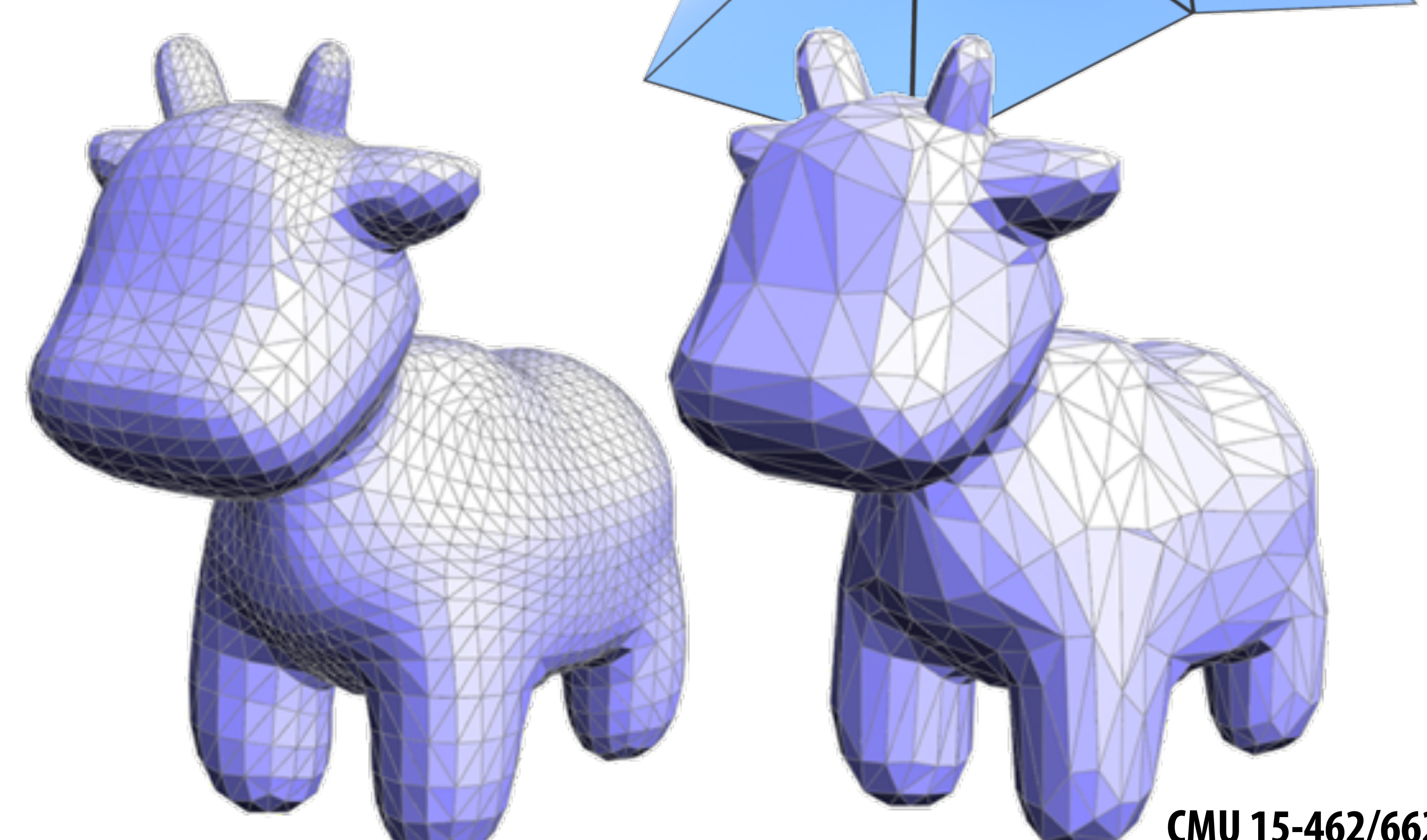
$$2B\mathbf{x} + 2\mathbf{w} = 0 \qquad \Longleftrightarrow \qquad \mathbf{x} = -B^{-1}\mathbf{w}$$

Q: Why should B be positive-definite?

Quadric Error Simplification: Final Algorithm

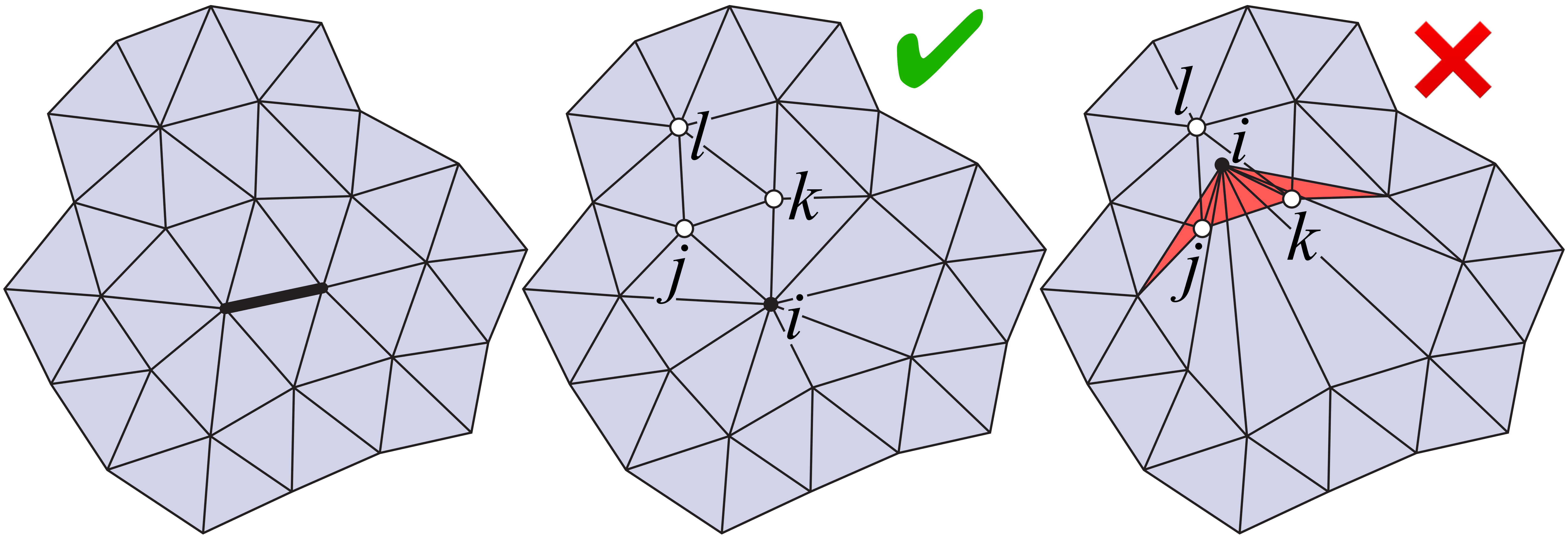
- Compute K for each triangle (squared distance to plane)
- Set K_i at each vertex to sum of Ks from incident triangles
- For each edge e_{ij}:
  - set $K_{ij} = K_i + K_j$
  - find point $\mathbf{x}$ minimizing error, set cost to $K_{ij}(\mathbf{x})$
- Until we reach target number of triangles:
  - collapse edge e_{ij} with smallest cost to optimal point $\mathbf{x}$
  - set quadric at new vertex to K_{ij}
  - update cost of edges touching new vertex
- More details in assignment writeup!





Quadric Simplification—Flipped Triangles

- Depending on where we put the new vertex, one of the new triangles might be "flipped" (normal points in instead of out):

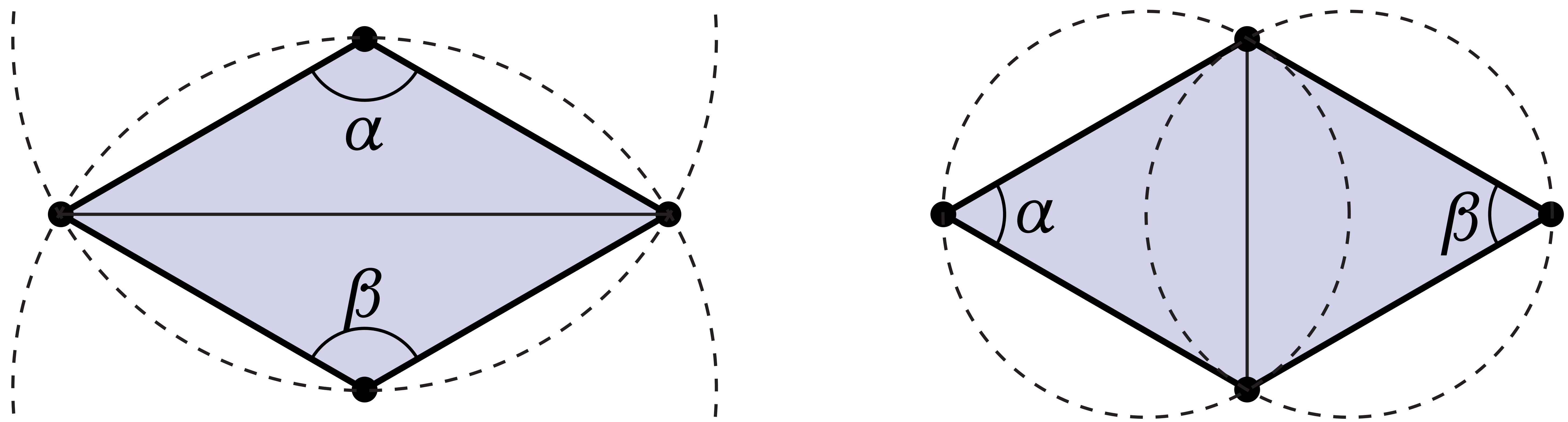


- Easy solution: for each triangle ijk touching collapsed vertex i, consider normals N_{ijk} and N_{kjl} (where kjl is other triangle containing edge jk)
- If $\langle N_{ijk}, N_{kjl} \rangle$ is negative, don't collapse this edge!

What if we're happy with the number of triangles, but want to improve quality?

How do we make a mesh “more Delaunay”?

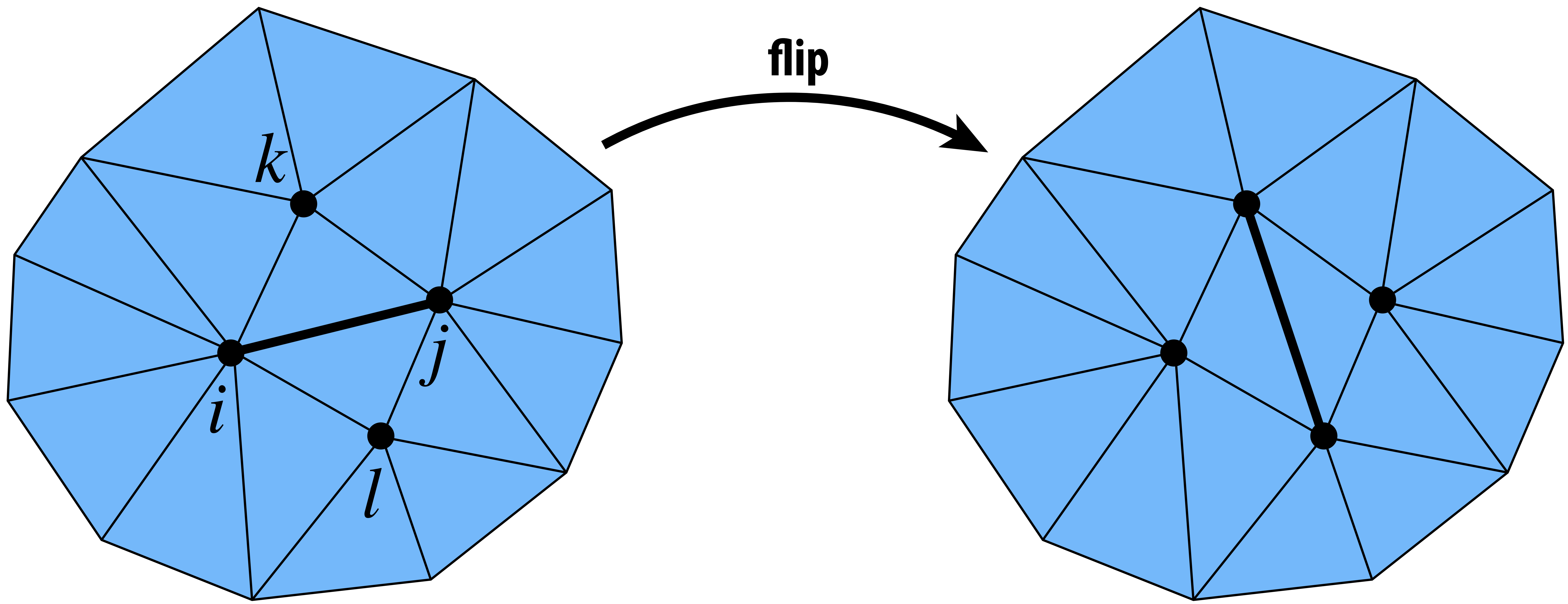
- Already have a good tool: edge flips!
- If $\alpha + \beta > \pi$, flip it!



- FACT: in 2D, flipping edges eventually yields Delaunay mesh
- Theory: worst case $O(n^2)$; doesn’t always work for surfaces in 3D
- Practice: simple, effective way to improve mesh quality

Alternatively: how do we improve degree?

- Same tool: edge flips!
- If total deviation from degree-6 gets smaller, flip it!

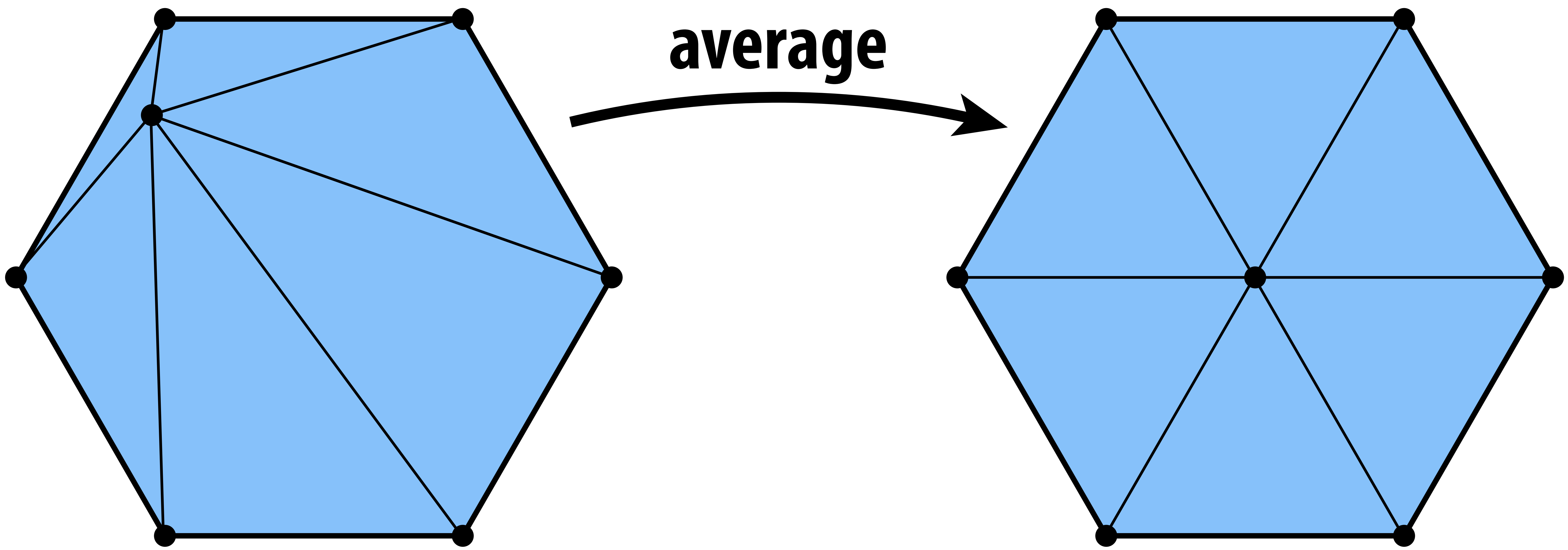


total deviation: $|d_i - 6| + |d_j - 6| + |d_k - 6| + |d_l - 6|$

- FACT: average degree approaches 6 as number of elements increases
- Iterative edge flipping acts like "discrete diffusion" of degree
- No (known) guarantees; works well in practice

How do we make a triangles “more round”?

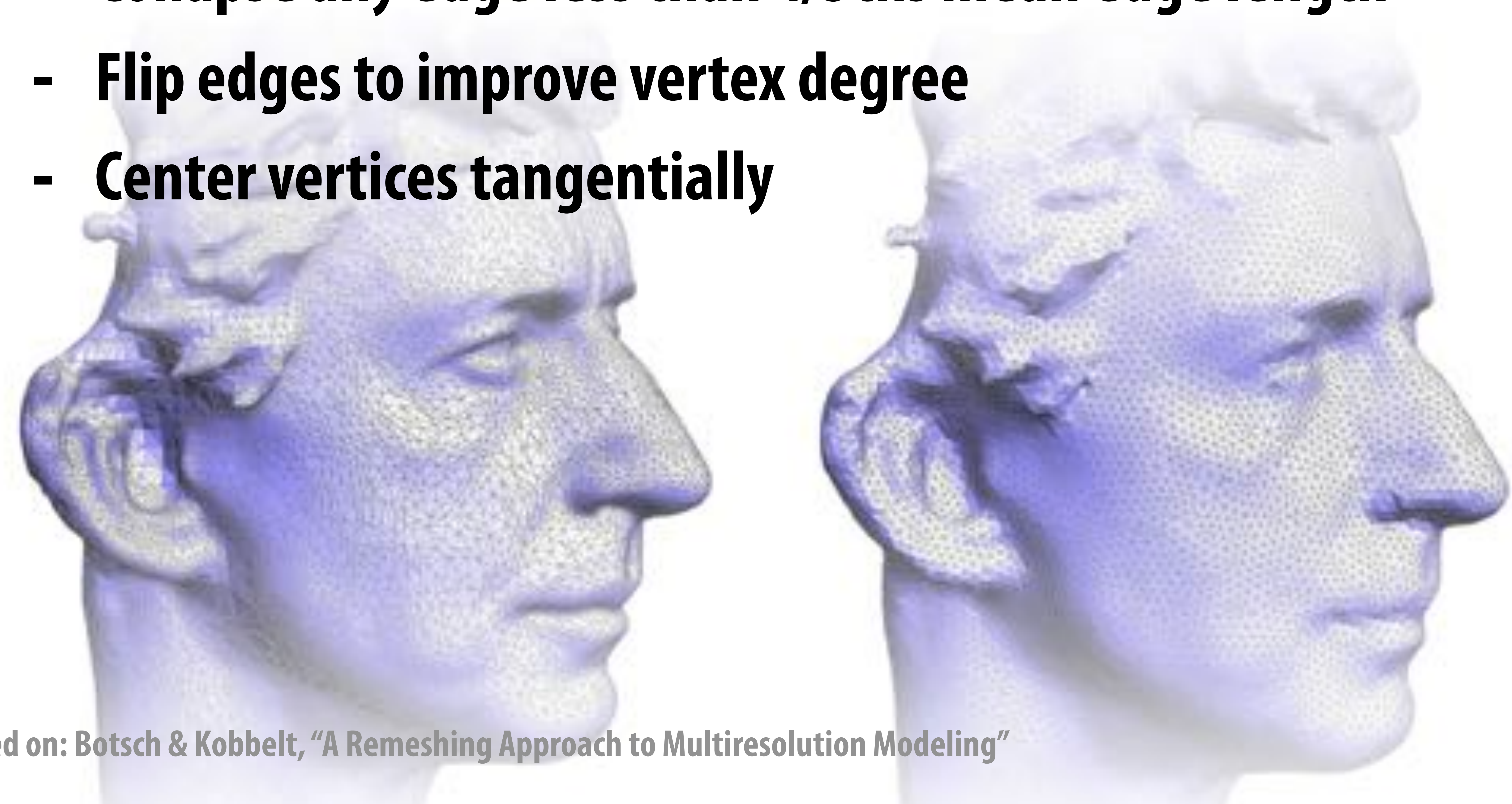
- Delaunay doesn’t guarantee triangles are “round” (angles near 60°)
- Can often improve shape by centering vertices:



- Simple version of technique called “Laplacian smoothing”
- On surface: move only in tangent direction
- How? Remove normal component from update vector

Isotropic Remeshing Algorithm

- Try to make triangles uniform shape & size
- Repeat four steps:
  - Split any edge over 4/3rds mean edge length
  - Collapse any edge less than 4/5ths mean edge length
  - Flip edges to improve vertex degree
  - Center vertices tangentially

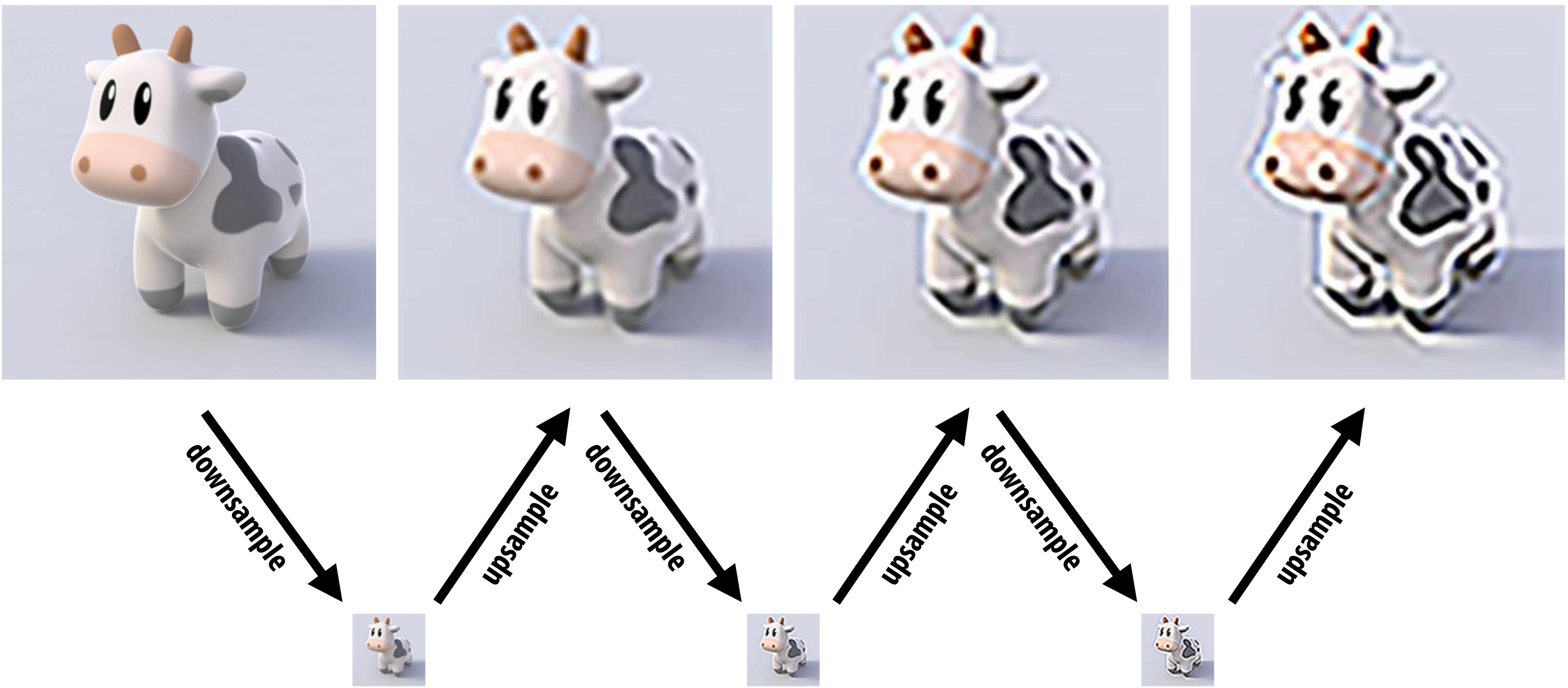


Based on: Botsch & Kobbelt, "A Remeshing Approach to Multiresolution Modeling"

What can go wrong when you resample a signal?

Danger of Resampling

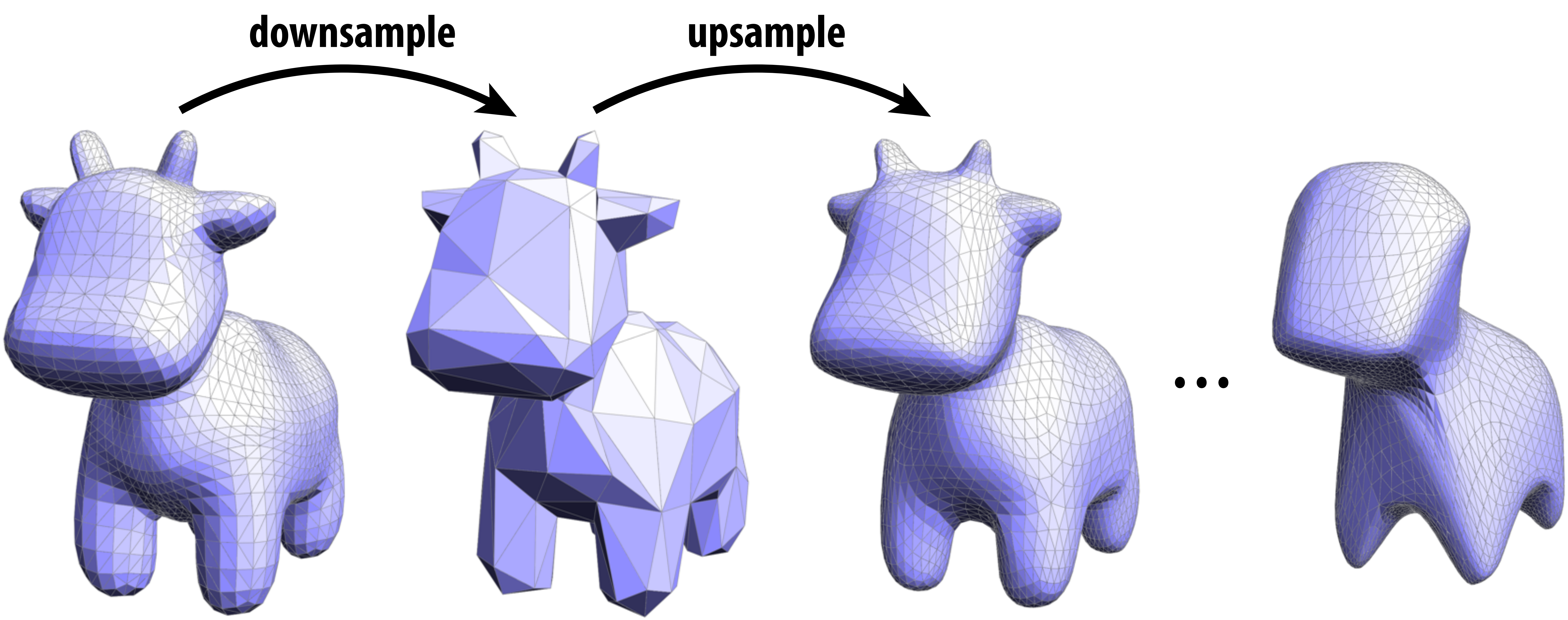
Q: What happens if we repeatedly resample an image?



A: Signal quality degrades!

Danger of Resampling

Q: What happens if we repeatedly resample a mesh?



A: Signal also degrades!

But wait: we have the original signal (mesh).

Why not just project each new sample point onto the closest point of the original mesh?

Next Time: Geometric Queries

- **Q: Given a point, in space, how do we find the closest point on a surface? Are we inside or outside the surface? How do we find intersection of two triangles? Etc.**
- **Do implicit/explicit representations make such tasks easier?**
- **What's the cost of the naïve algorithm, and how do we <u>accelerate</u> such queries for large meshes?**
- **So many questions!**