Lecture 9:

Mesh Representation & Geometry Processing

Computer Graphics 2025

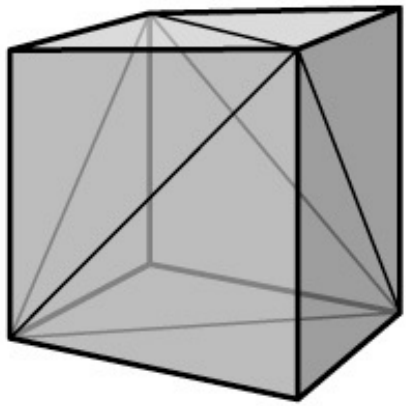
Fuzhou University - Computer Science

Mesh Representations

网格表示

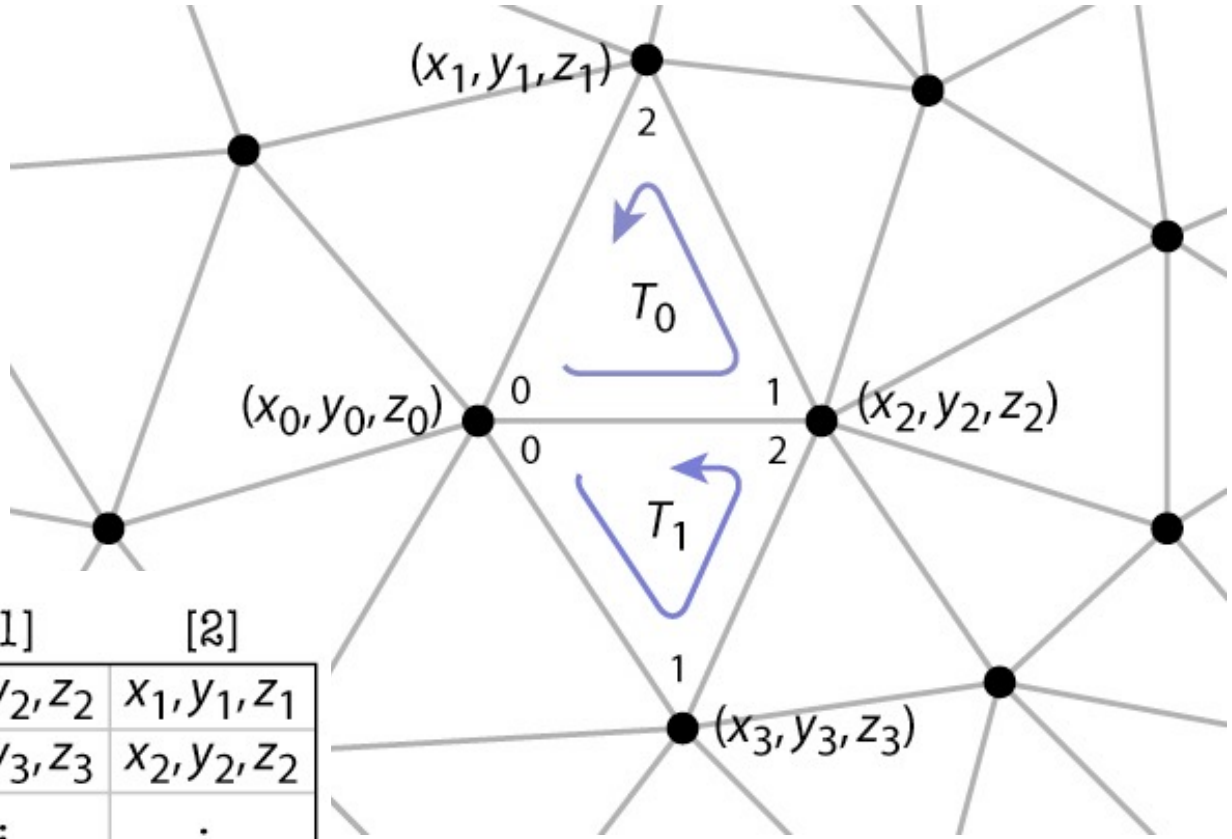

“网”格 <—> 线框

A Small Triangle Mesh



8 vertices, 12 triangles

List of Triangles

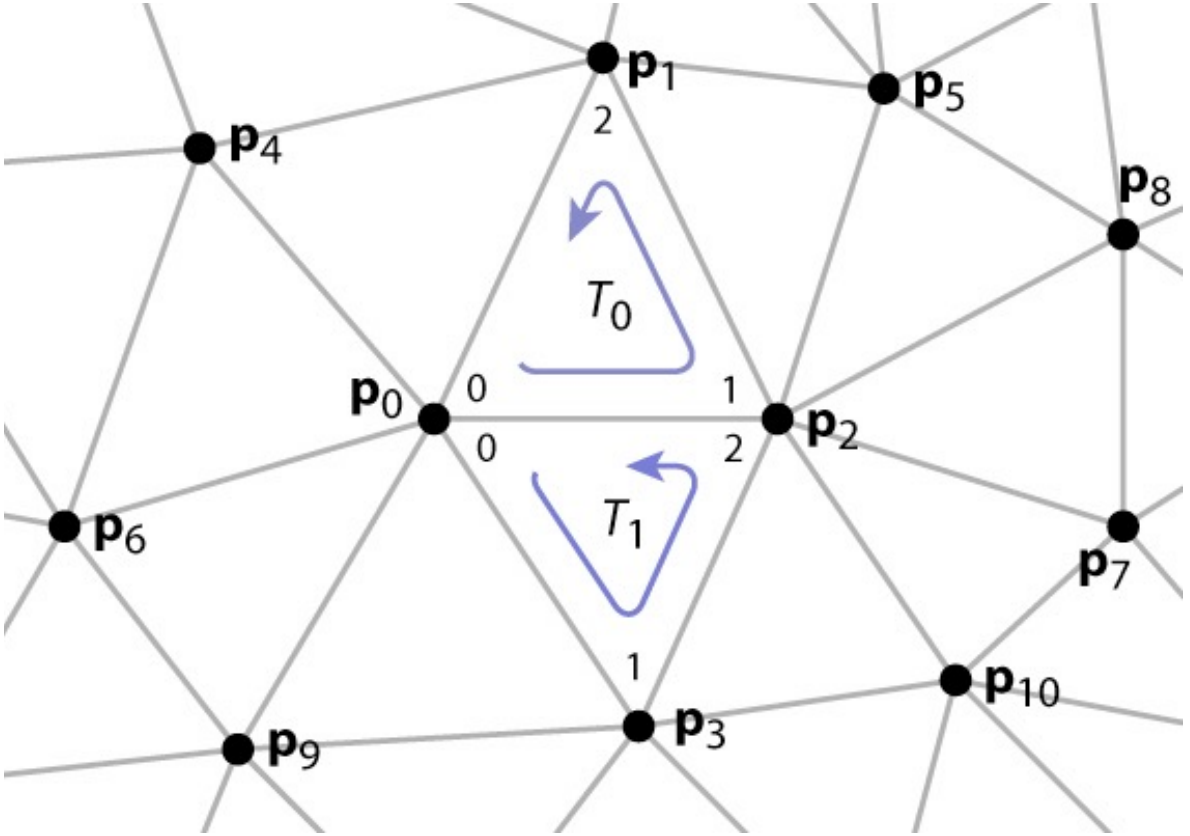


	[0]	[1]	[2]
tris[0]	x_0, y_0, z_0	x_2, y_2, z_2	x_1, y_1, z_1
tris[1]	x_0, y_0, z_0	x_3, y_3, z_3	x_2, y_2, z_2
	⋮	⋮	⋮

Lists of Points / Indexed Triangle

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	⋮

tInd[0]	0, 2, 1
tInd[1]	0, 3, 2
	⋮



Comparison

Triangles

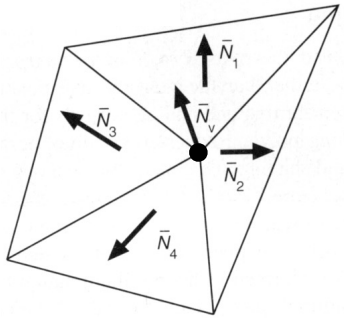
+ Simple
– Redundant information

Points + Triangles

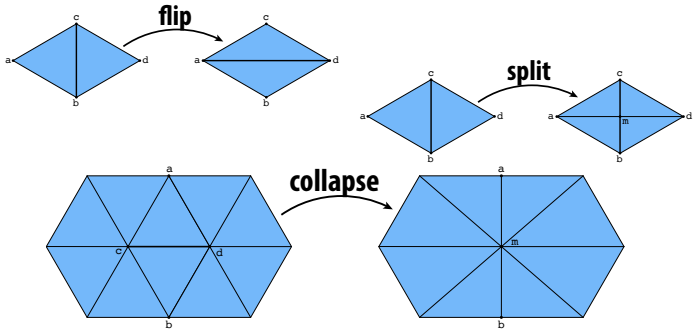
+ Sharing vertices reduces memory usage
+ Ensure integrity of the mesh (moving a vertex causes that vertex in all the polygons to move)

Good Data Structure For Mesh

- Constant time access to neighbors
  e.g. surface normal calculation, subdivision



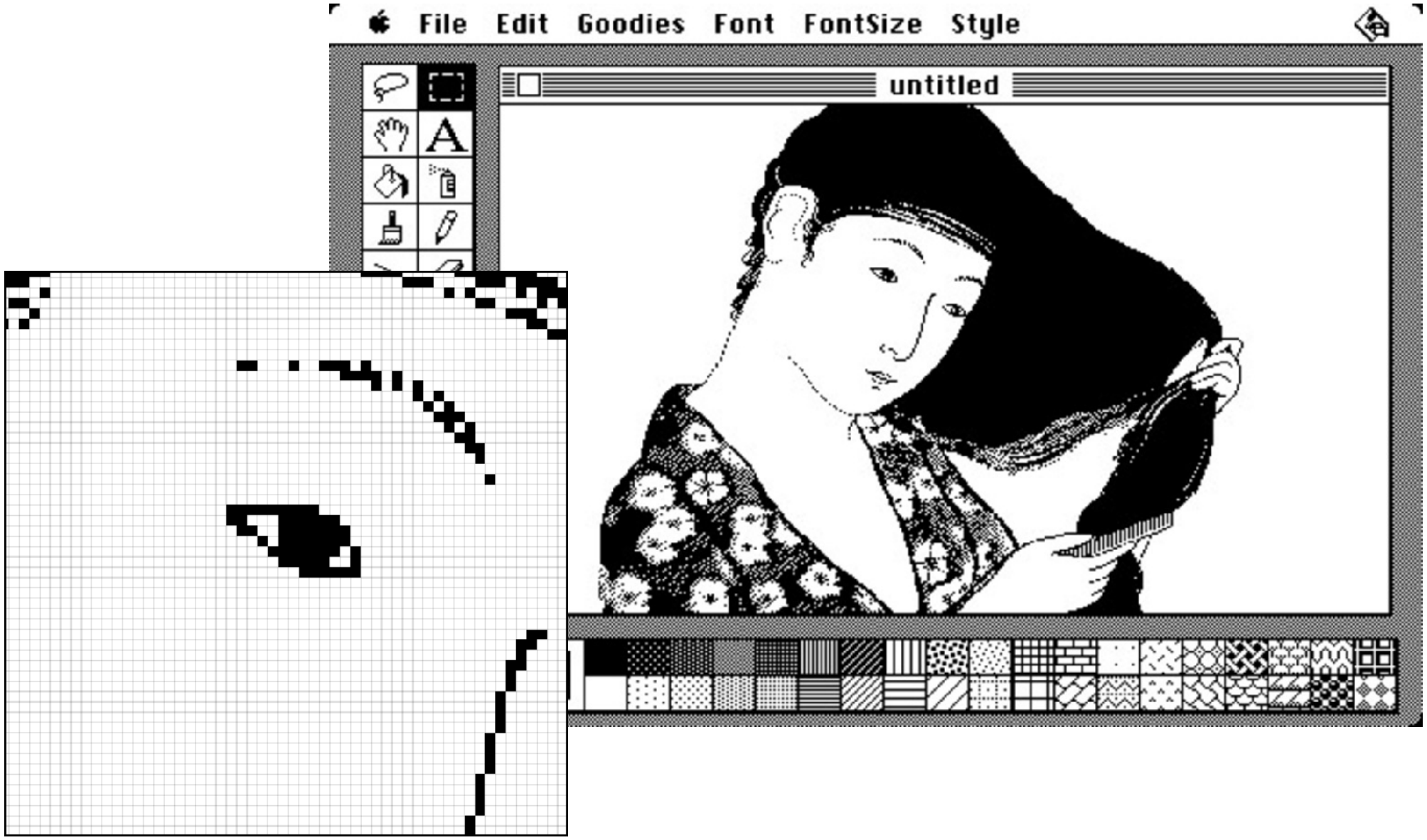
- Editing the geometry
  e.g. adding/removing vertices, faces, edges, etc.



- Can we do better?

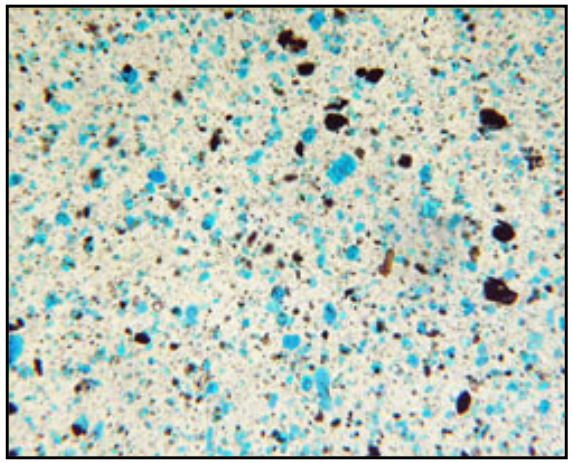
Bitmap Images

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



Bug images are not fundamentally made of little squares:

Goyō Hashiguchi, *Kamisuki* (ca 1920)



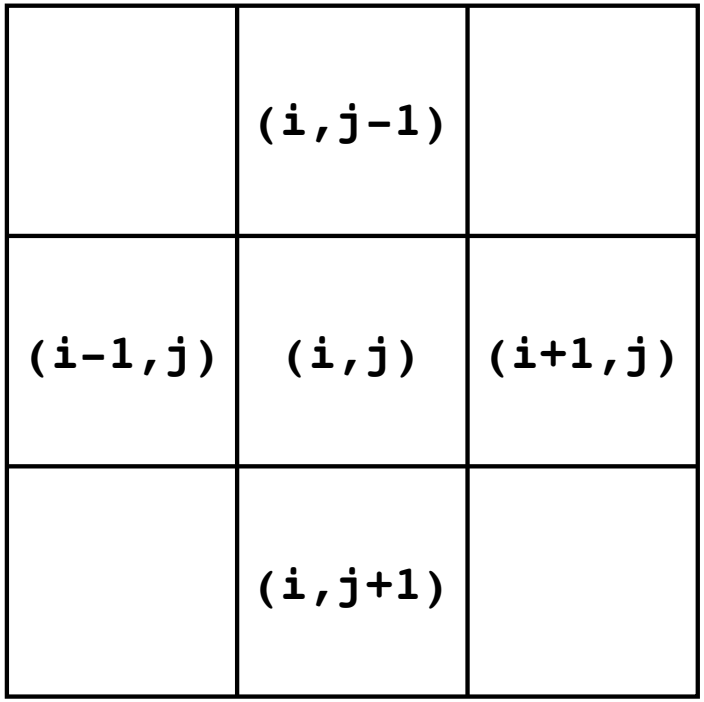
photomicrograph of paint

So why did we choose a square grid?

Rather than dozens of possible 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
- Similar story with geometry…

So, how should we encode 3d objects?

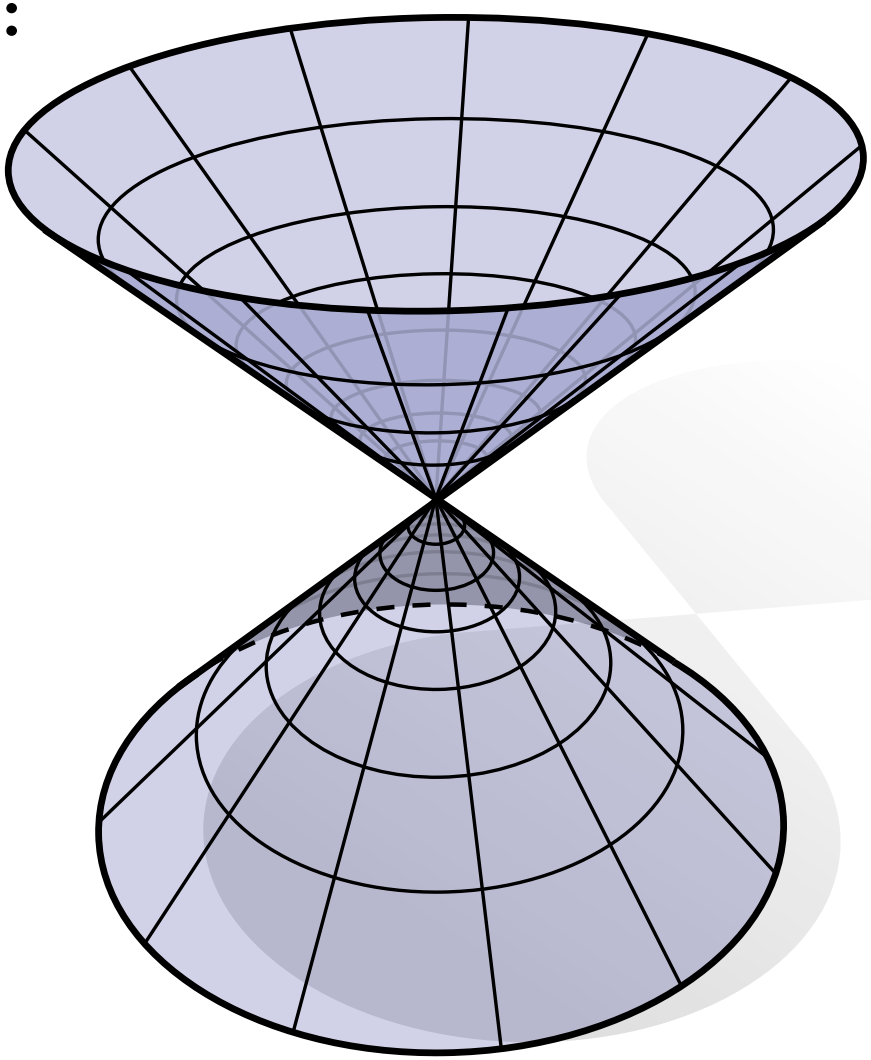
- Pixel -> Voxel
- Solid model
- Too expensive

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

Isn’t every shape manifold? 流形

- No, for instance:



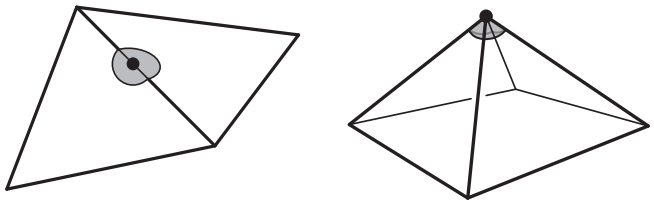
Center point never looks like the plane, no matter how close we get.

流形的局部放大后近似平面，图中形状的中心点处不能，固非流形

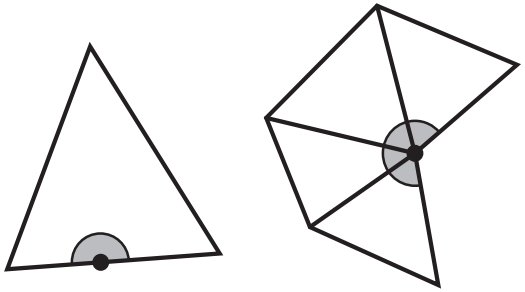
Manifold Definition

Definition: a 2D manifold is a surface that when cut with a small sphere always yields a disk.

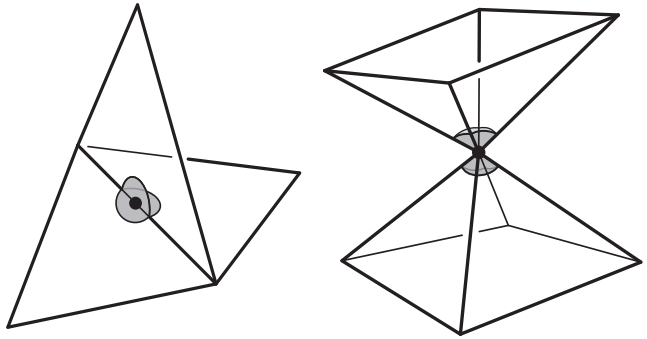
Manifold



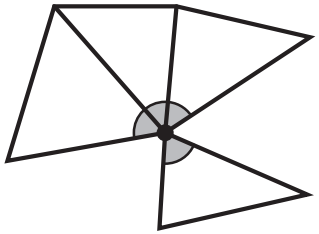
With border



Not manifold



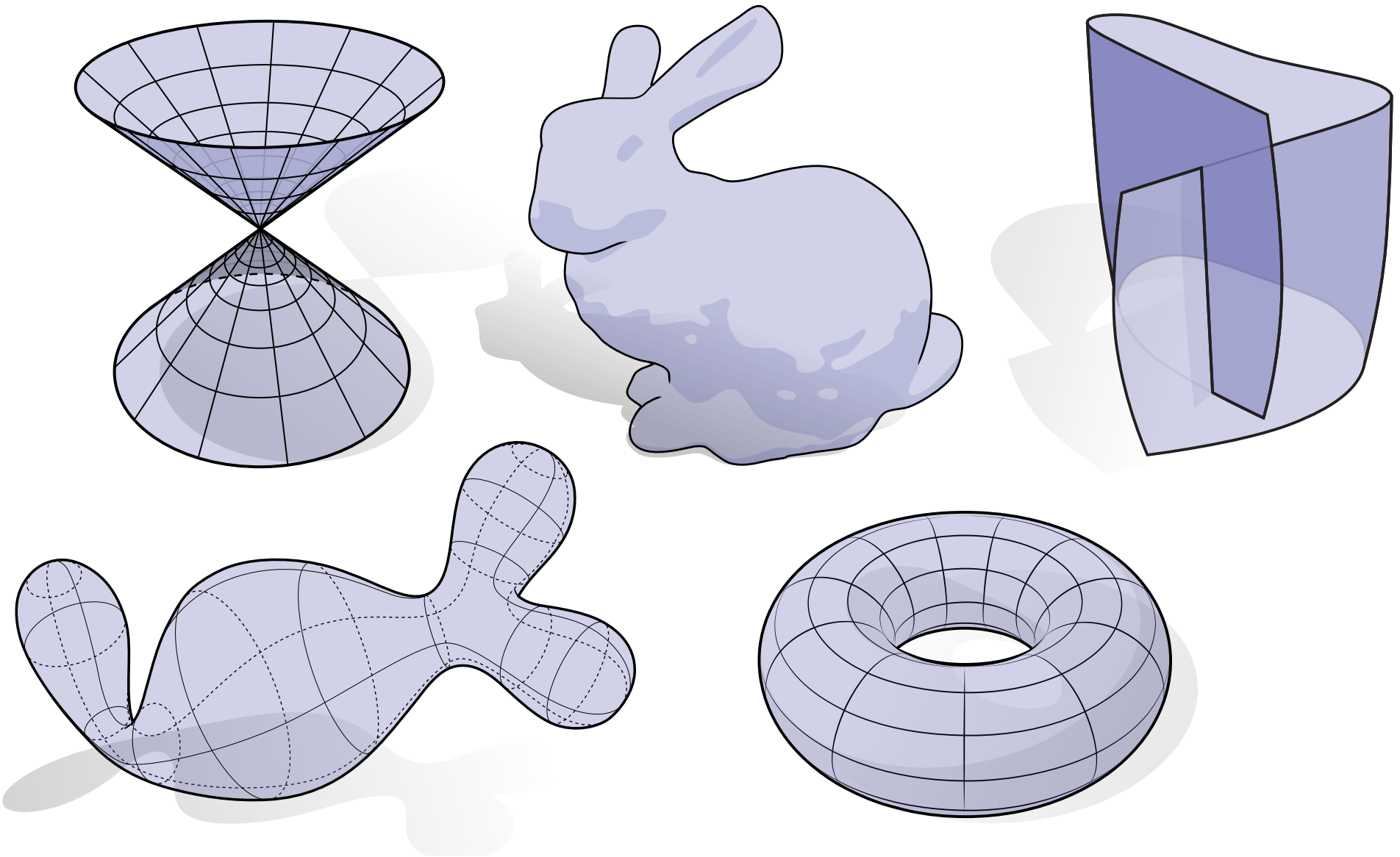
With border



Manifold with boundary: cut with a small sphere yields a *half-disc.*

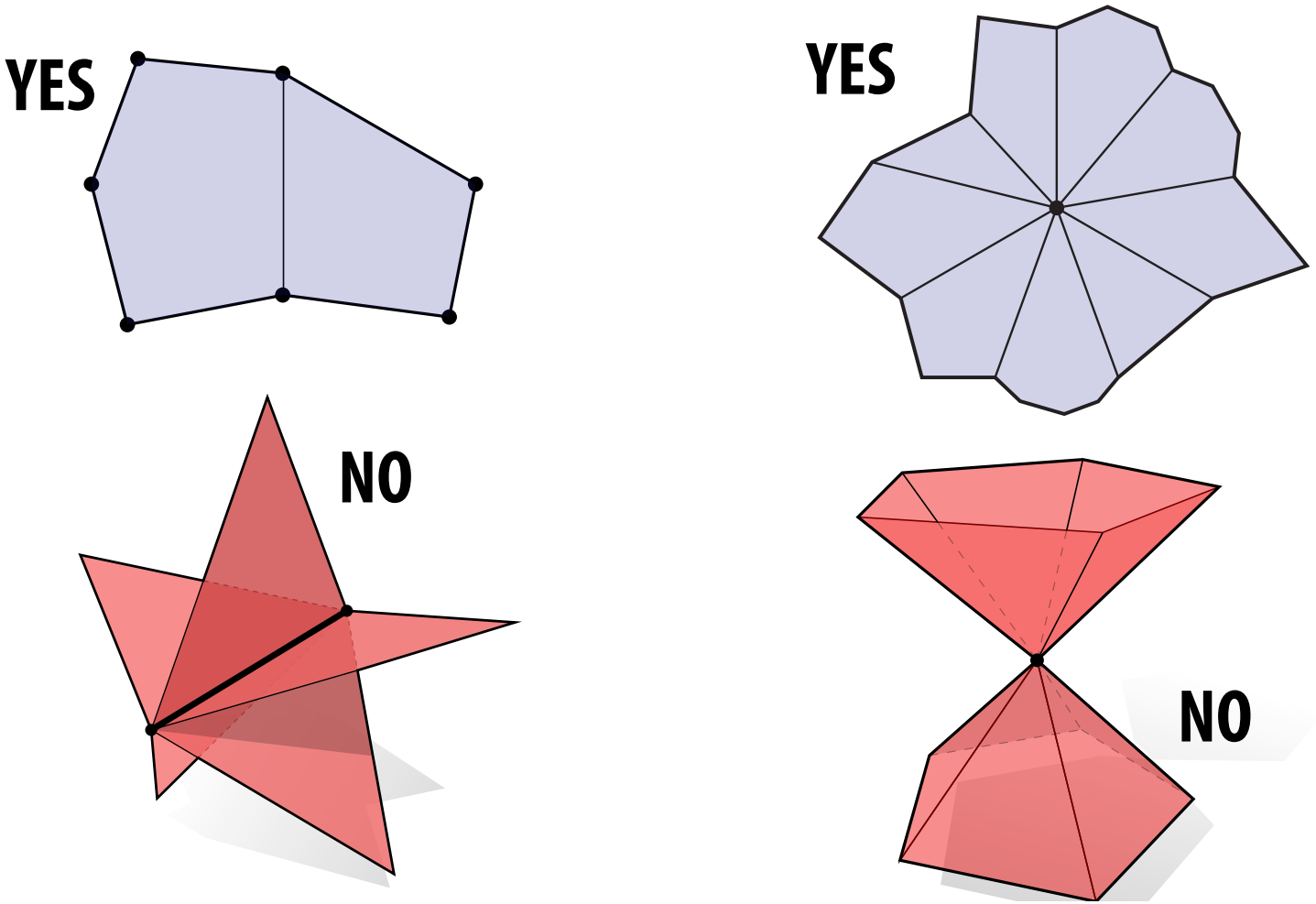
More Examples of Smooth Surfaces

- Which of these shapes are manifold?



A manifold polygon mesh has fans, not fins

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

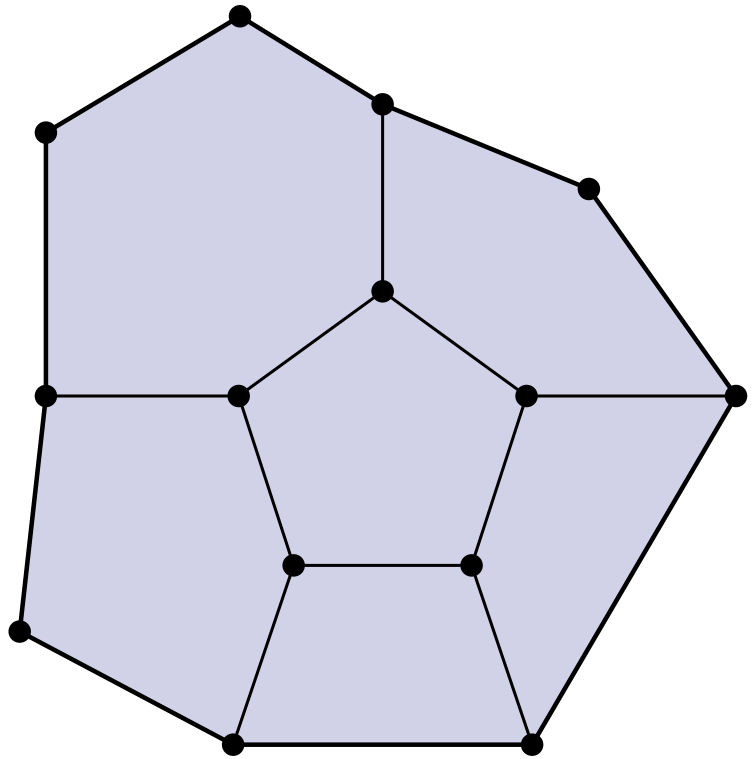


Why is the manifold assumption useful?

Keep it Simple!

- Same motivation as for images:
  - Make some assumptions about our geometry to keep data structure/algorithms simple and efficient
  - In many common cases, doesn't fundamentally limits what we can do with geometry

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



How do we actually encode all this data?

Adjacency List (Array-like) 邻接表

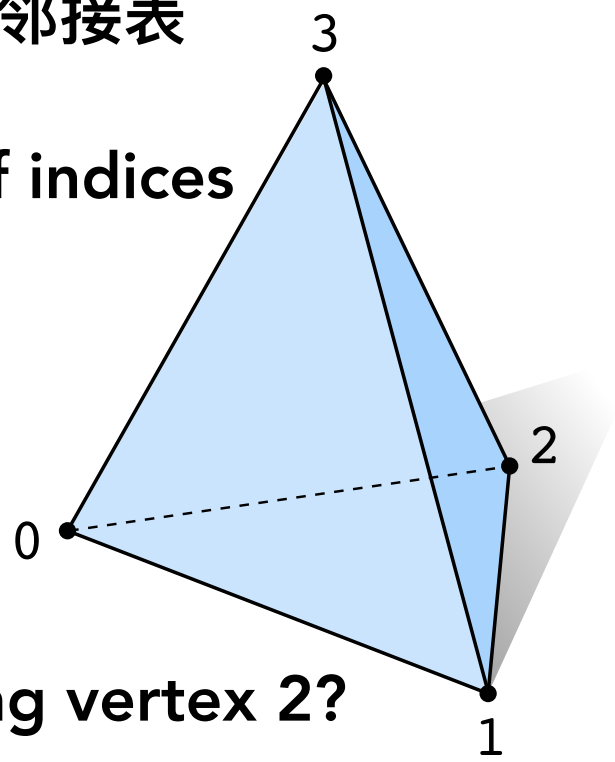
- Store triples of coordinates (x,y,z), tuples of indices
- E.g., tetrahedron:

VERTICES

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

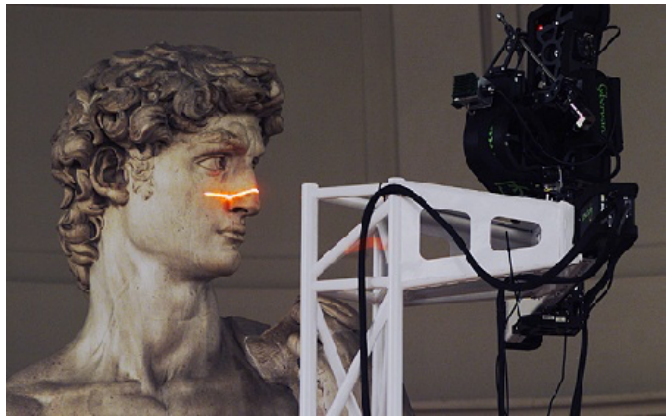
POLYGONS

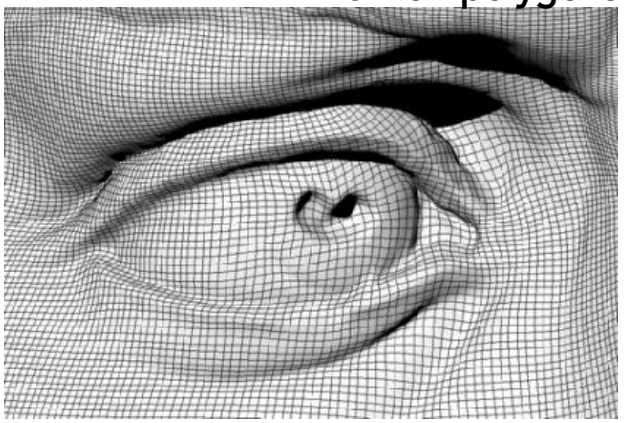
	i	j	k
	0	2	1
	0	3	2
	3	0	1
	3	1	2



- 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 关联矩阵

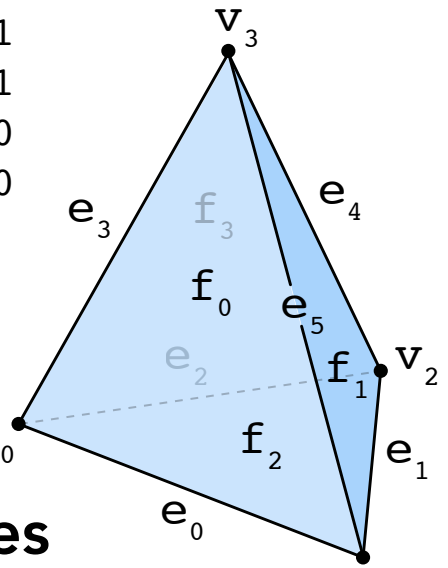
- If we want to answer neighborhood queries, why not simply store a list of neighbors?
- Can encode all neighbor information via incidence matrices
- E.g., tetrahedron:

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

Half-Edge Data Structure (Linked-list-like)

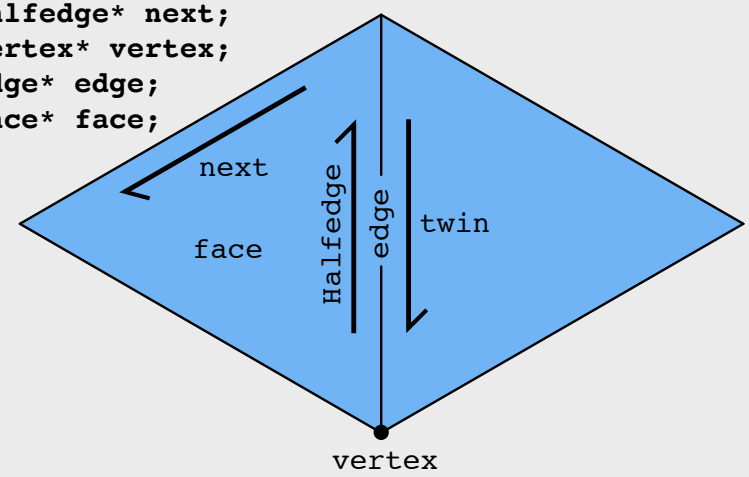
（半边）

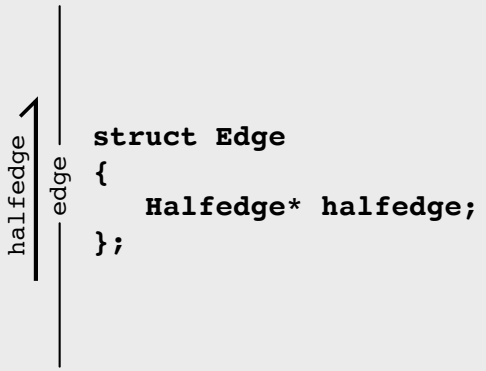
- Store some information about neighbors
- Don’t need an exhaustive list; just a few key pointers
- Key idea: two *halfedges* act as “glue” between mesh elements:
- 每条边被拆成两条“半边”， 每条半边都有方向，并且记录邻接信息。
  - 比如一条边连接顶点 A 和 B： 在半边结构中，这条边会被拆为：
    $HalfEdge_1$：从 A 指向 B
    $HalfEdge_2$：从 B 指向 A
  - 这两条半边互为对偶（twin）。

Half-Edge Data Structure (Linked-list-like)

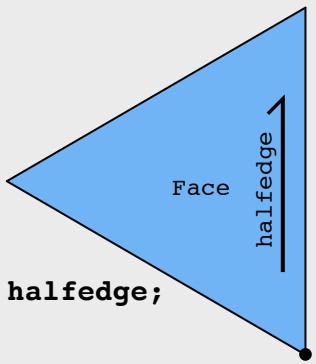
- 每条边被拆成两条“半边”， 每条半边都有方向，并且记录邻接信息。
  - 比如一条边连接顶点 A 和 B： 在半边结构中，这条边会被拆为：
    $HalfEdge_1$：从 A 指向 B
    $HalfEdge_2$：从 B 指向 A
  - 这两条半边互为对偶（twin）。

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

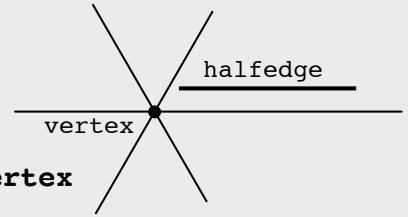





```
struct Edge
{
   Halfedge* halfedge;
};
```




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

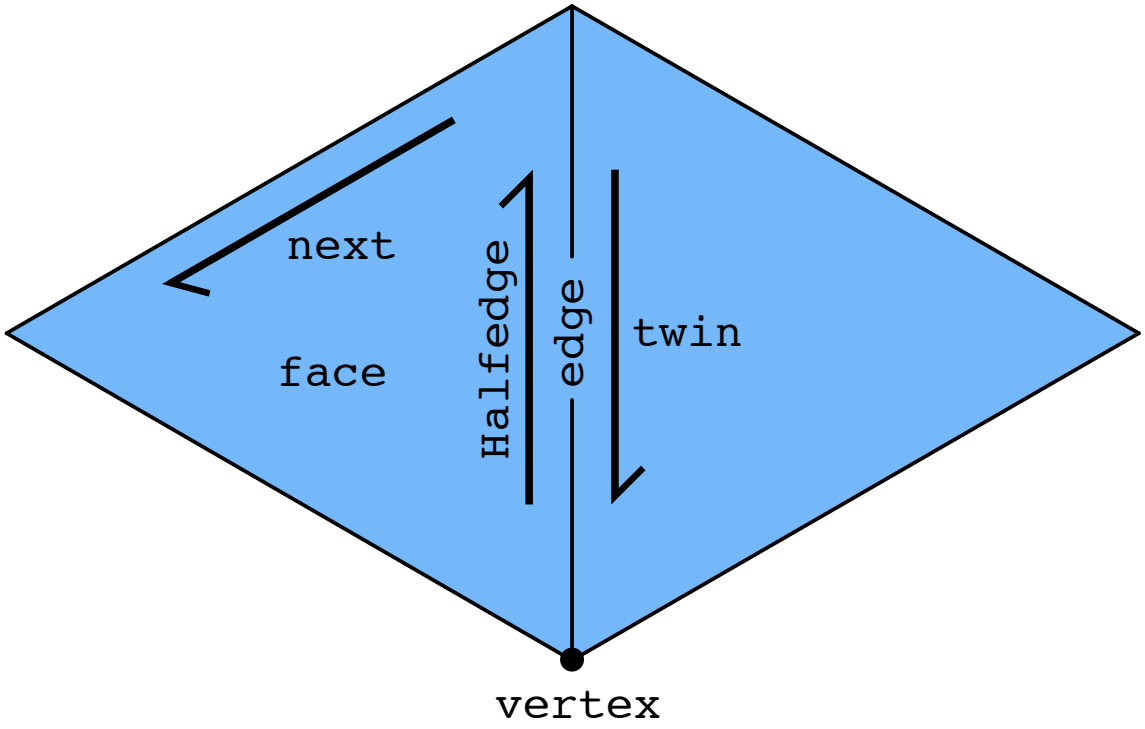



```
struct Vertex
{
   Halfedge* halfedge;
};
```

Half-Edge Data Structure

```
struct Halfedge {
   Halfedge *twin,
   Halfedge *next;
   Vertex *vertex;
   Edge *edge;
   Face *face;
}
struct Vertex {
   Point pt;
   Halfedge *halfedge;
}
struct Edge {
   Halfedge *halfedge;
}
struct Face {
   Halfedge *halfedge;
}
```

Key idea: two half-edges act as "glue" between mesh elements



Each vertex, edge and face points to one of its half edges

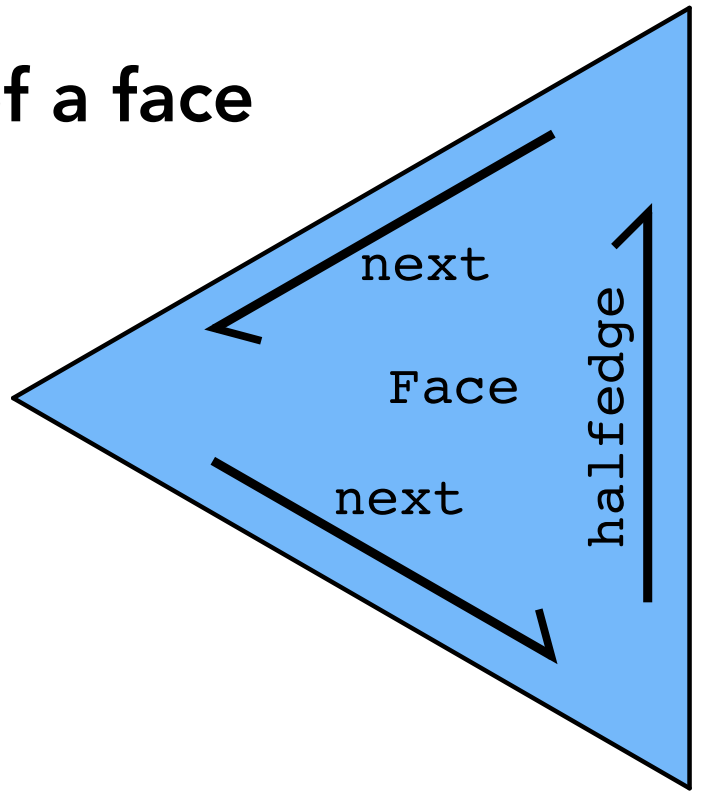
Half-Edge Facilitates Mesh Traversal

Use twin and next pointers to move around mesh

Process vertex, edge and/or face pointers

Example 1: process all vertices of a face

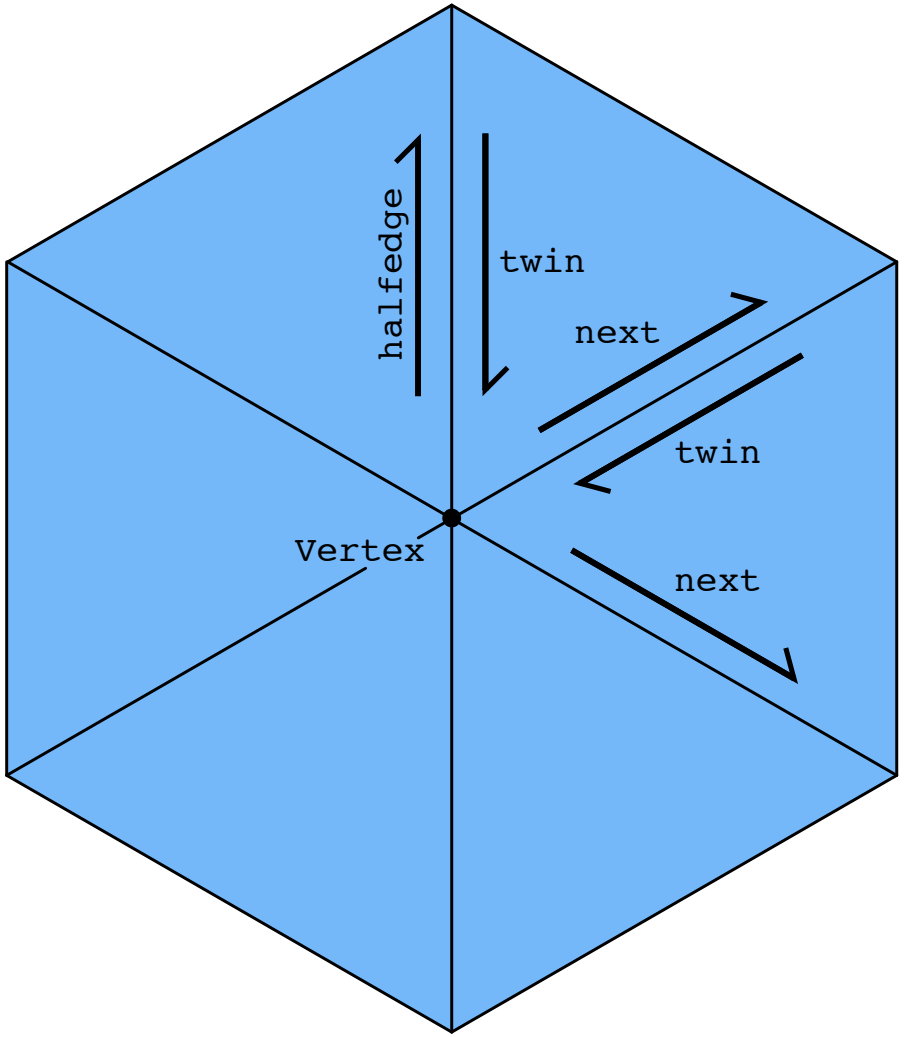
```
Halfedge* h = f->halfedge;
do {
   process(h->vertex);
   h = h->next;
}
while( h != f->halfedge );
```



Half-Edge Facilitates Mesh Traversal

Example 2: process all edges around a vertex

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



Half-Edge Data Structure

- “半边数据结构” 就像在每条边上放两条带方向的箭头， 每个箭头都知道：
  - 自己从哪个点出发（vertex）
  - 指向哪个面（face）
  - 下一个箭头是谁（next）
  - 自己的反方向箭头是谁（twin）

  这样一来，整个网格就变成一个可以“顺着箭头走”的拓扑网络。 任何点、边、面之间的关系都能通过这些指针快速找到。

Half-Edge meshes are always manifold

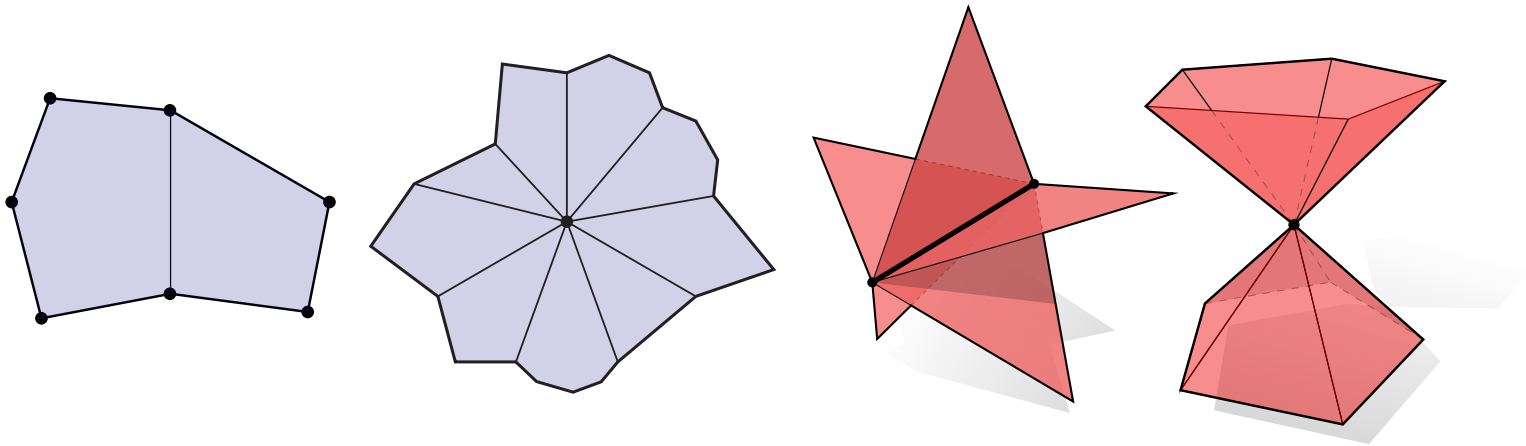
- Consider simplified halfedge data structure
- Require only common-sense conditions

(pointer to yourself!)

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

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

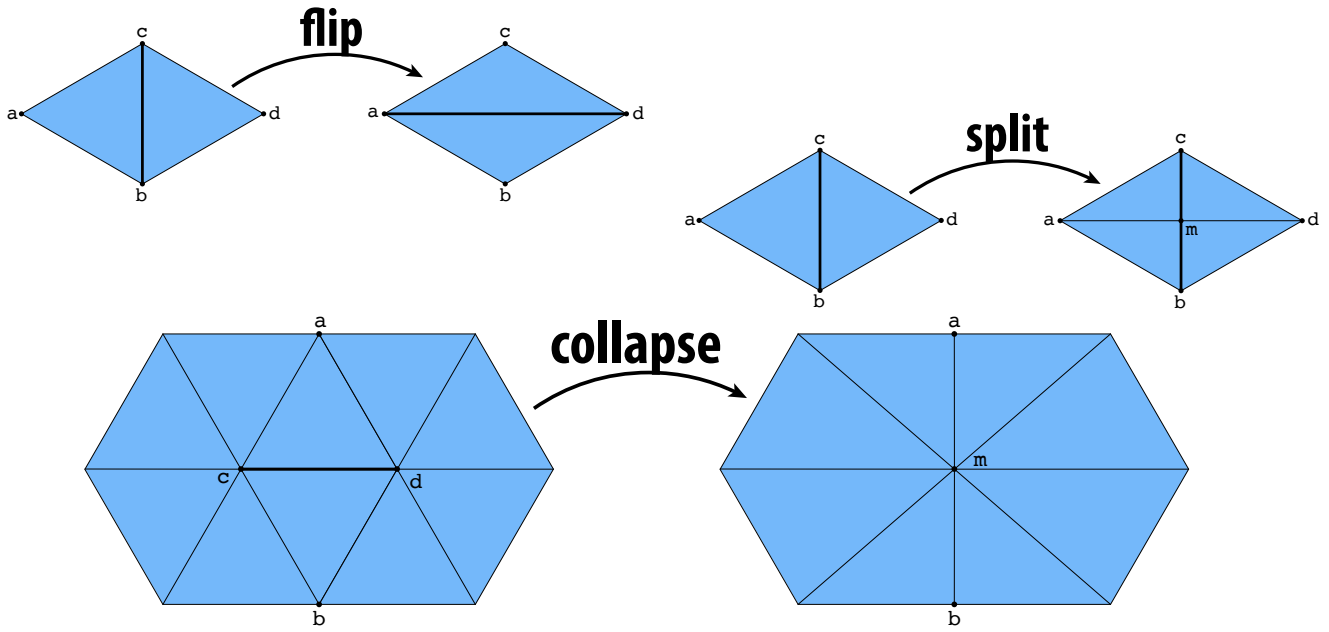
- Keep following next, and you'll get faces.
- Keep following twin, and you'll get edges.
- Keep following next->twin and you'll get vertices.



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

Half-Edge Meshes are Easy to Edit

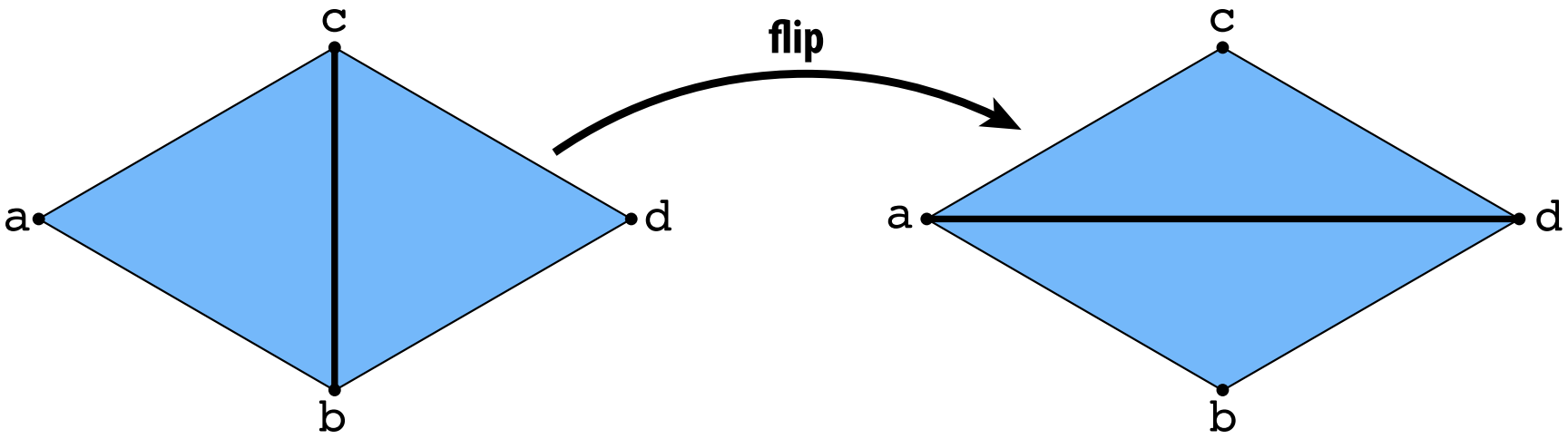
- Remember key feature of linked list: insert/delete elements
- E.g., for triangle meshes, several atomic operations:



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

Edge Flip (Triangles)

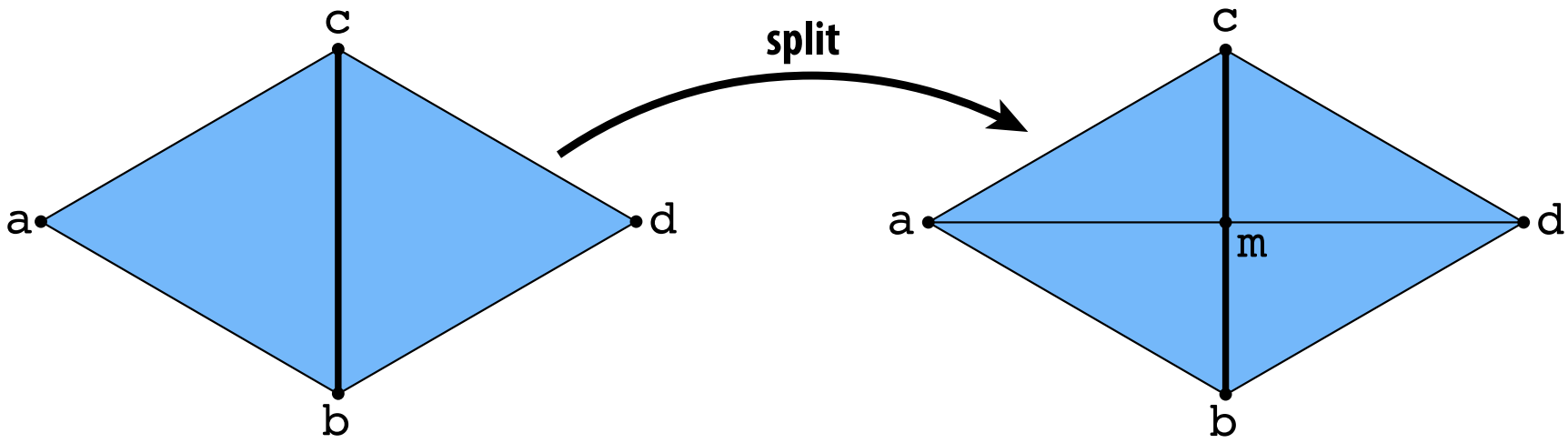
- Triangles (a,b,c), (b,d,c) become (a,d,c), (a,b,d):



- Long list of pointer reassignments (edge->halfedge = …)
- However, no elements created/destroyed.

Edge Split (Triangles)

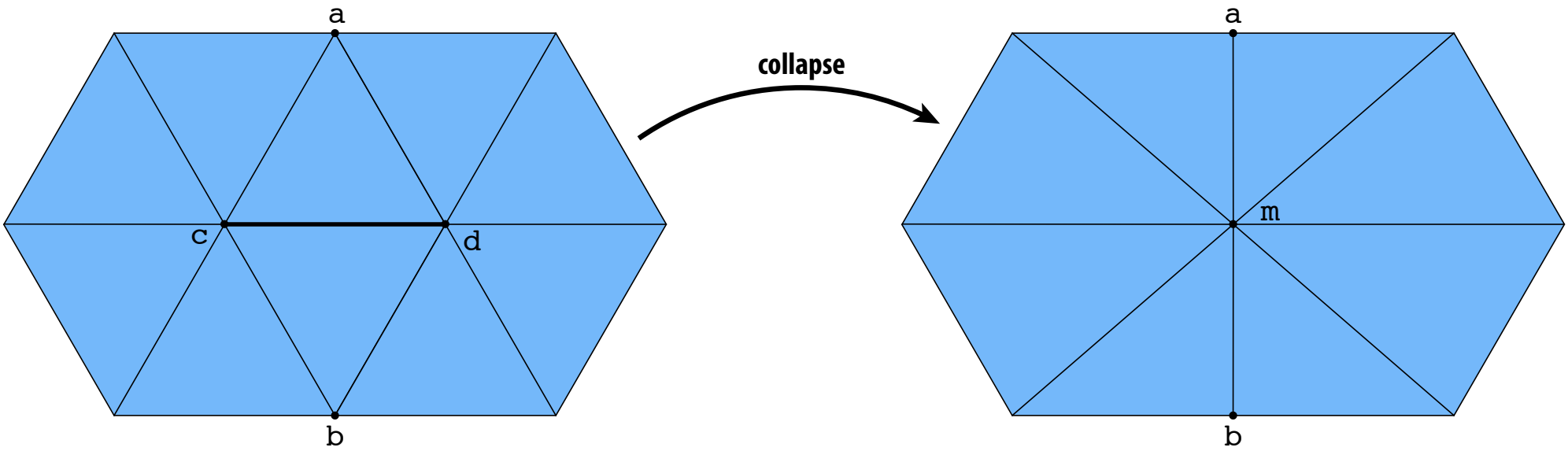
- Insert midpoint m of edge (c,b), connect to get four triangles:



- This time, have to add new elements.
- Lots of pointer reassignments.

Edge Collapse (Triangles)

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



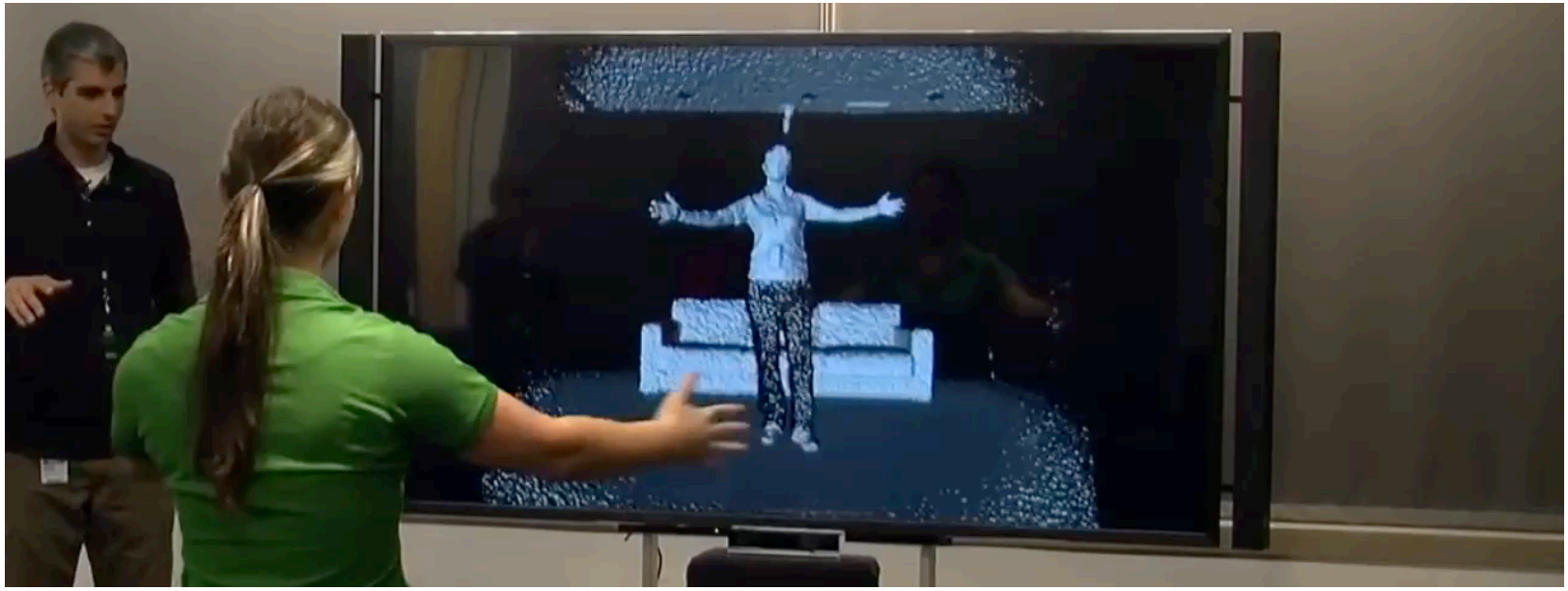
- Now have to delete elements.
- Still lots of pointer assignments!

Geometry Processing

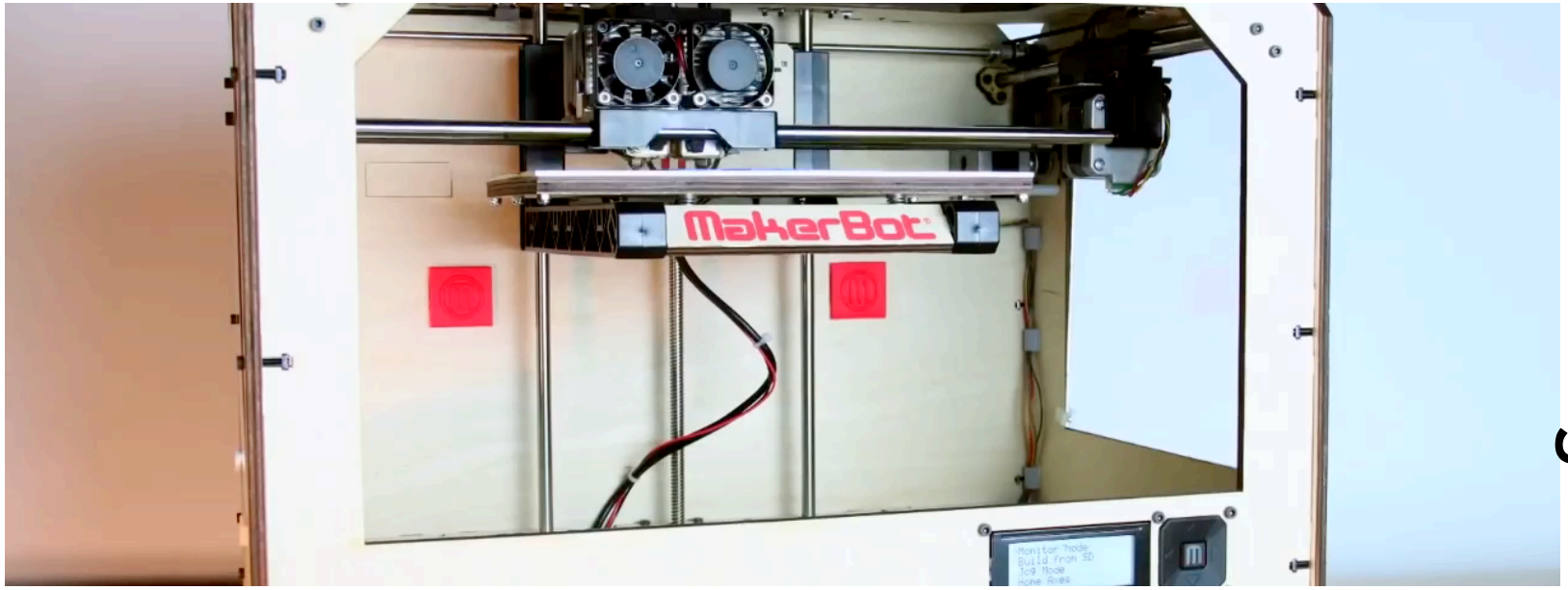
Digital Geometry Processing

- Extend traditional digital signal processing (audio, video, etc.) to deal with geometric signals:
  - Upsampling/ downsampling/ resampling/ filtering …
  - Aliasing (reconstructed surface gives “false impression”)

Digital Geometry Processing



3D Scanning

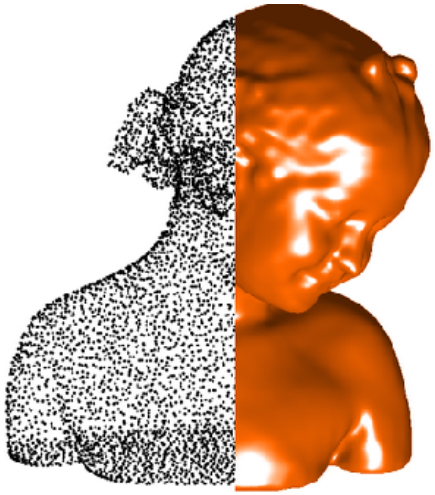


3D Printing

Geometry Processing Pipeline

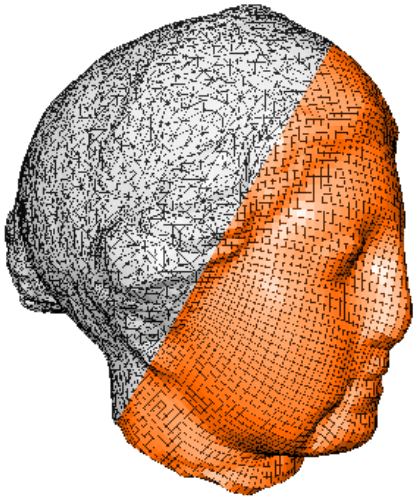
scan → process → print

Geometry Processing Tasks

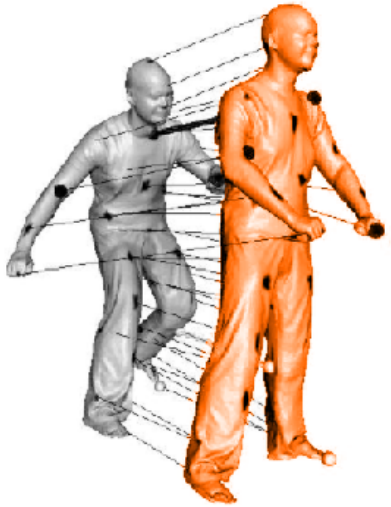


reconstruction

filtering

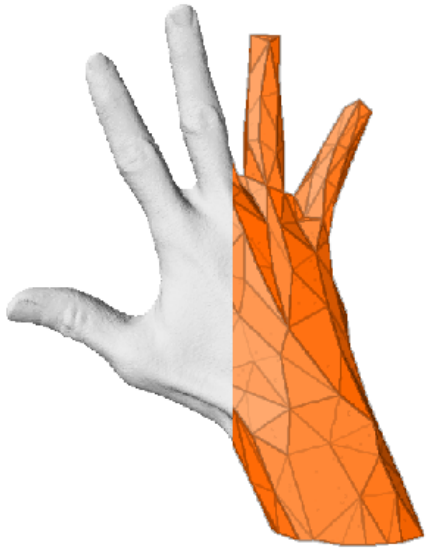


remeshing



shape analysis

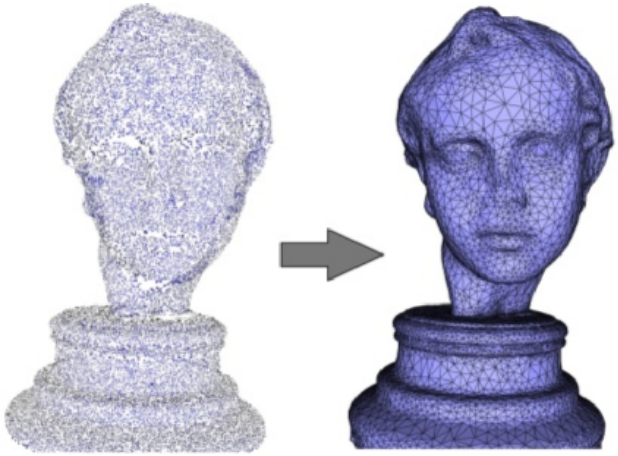
parameterization



compression

Geometry Processing: Reconstruction

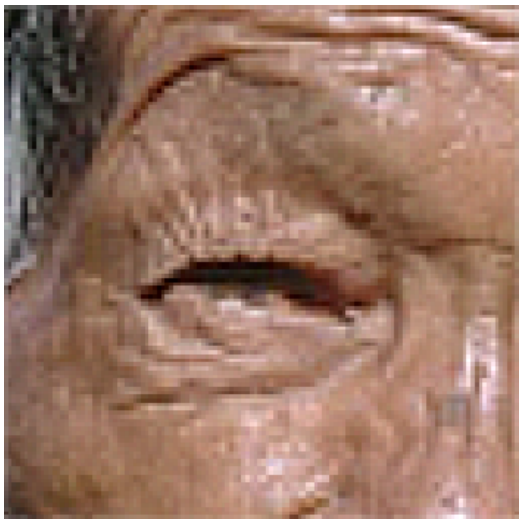
- Given samples of geometry, reconstruct surface
- What are "samples"? Many possibilities:
  - Points, points with normals, …
  - Image pairs/ sets (multi-view stereo)
  - Line density integrals (MRI/CT scans)

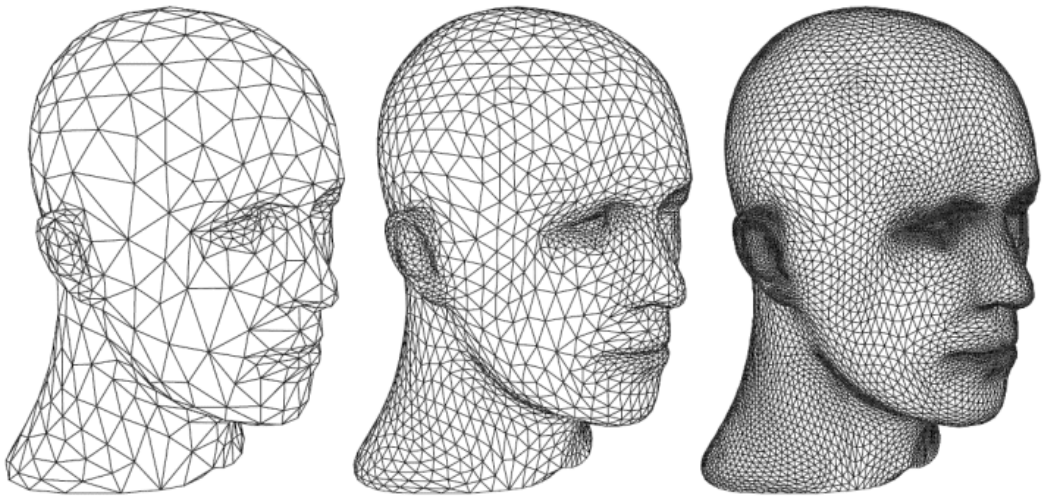


- How do you get a surface? Many techniques:
  - Silhouette-based (visual hull)
  - Voronoi-based (e.g., power crust)
  - PDE-based (e.g., Poisson reconstruction)
  - Iso-surfacing (marching cubes)

Geometry Processing: Upsampling

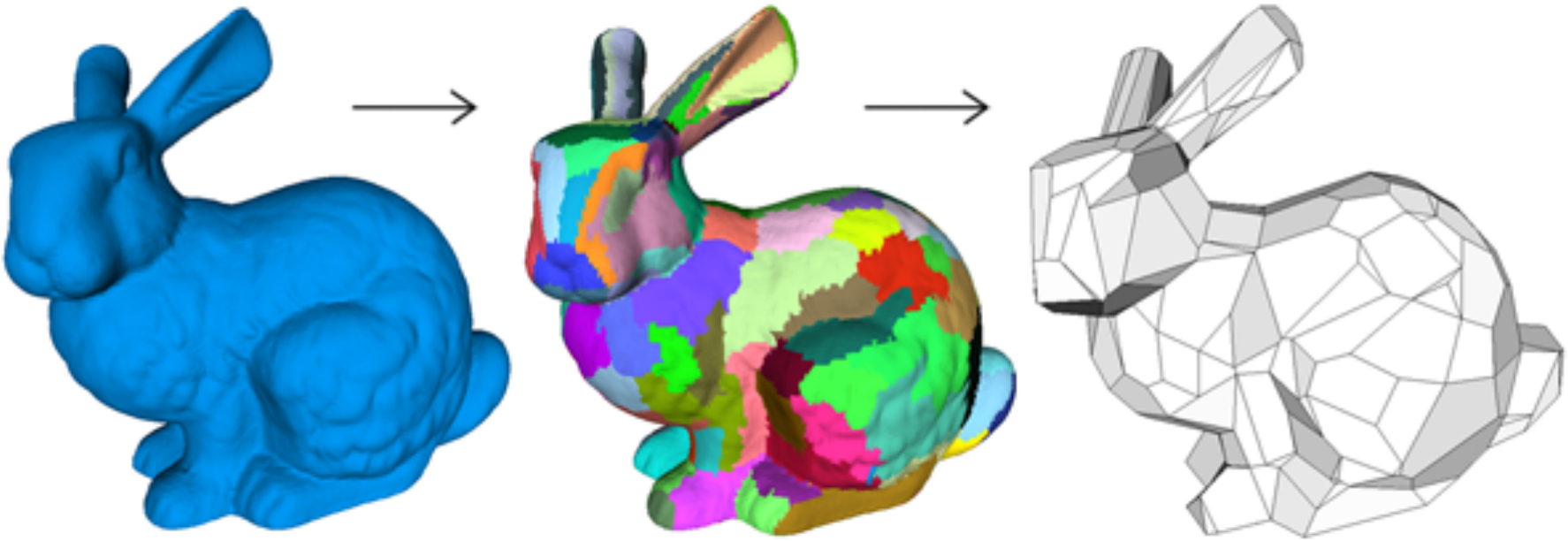
- Increase resolution via interpolation
- Images: e.g., bilinear, bicubic interpolation
- Polygon meshes:
  - Subdivision
  - Bilateral upsampling
  - …





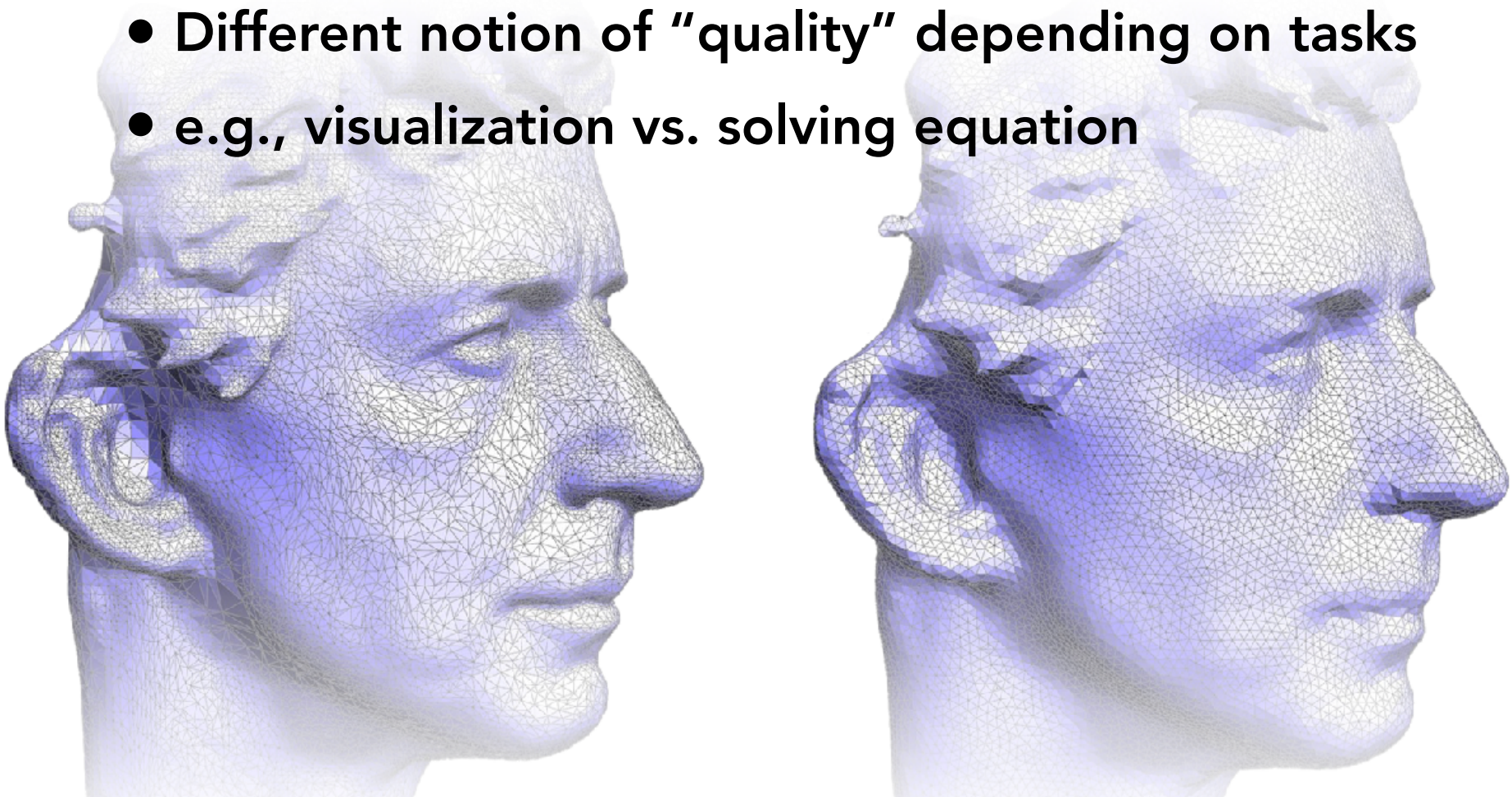
Geometry Processing: Downsampling

- 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, …



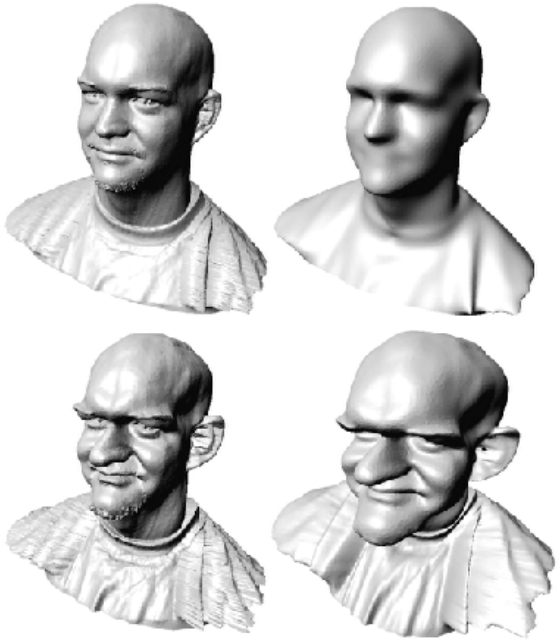
Geometry Processing: Resampling

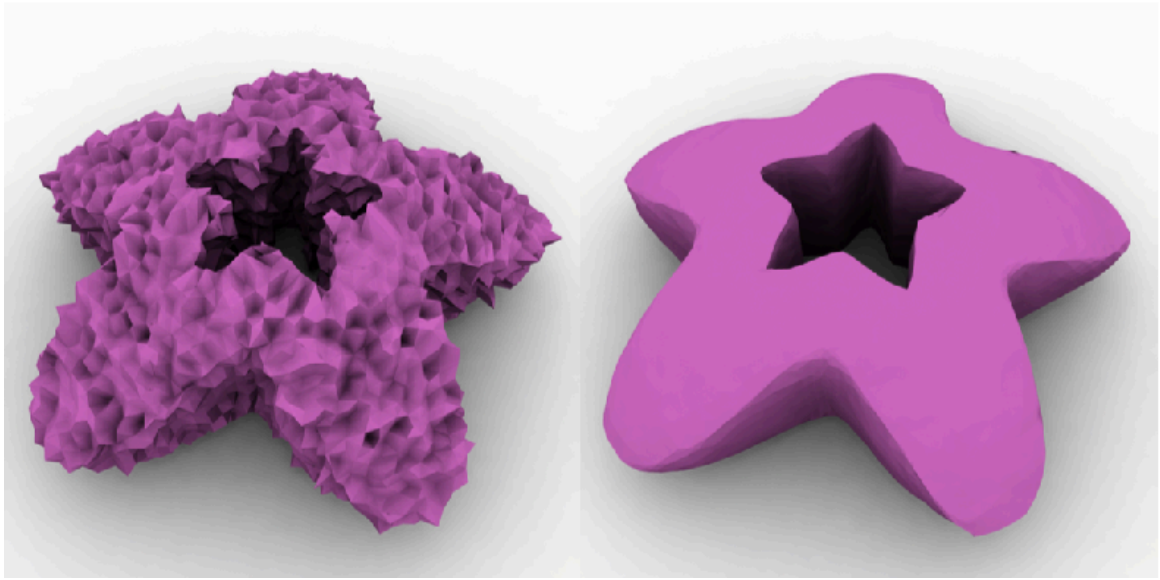
- Modify sample distribution to improve quality
- Images: not an issue! (Pixels always stored on a regular grid)
- Meshes: shape of polygons is extremely important!
  - Different notion of "quality" depending on tasks
  - e.g., visualization vs. solving equation



Geometry Processing: Filtering

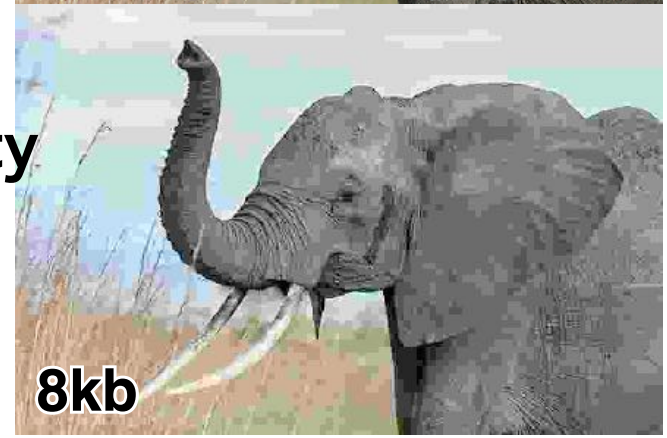
- Remove noise, or emphasize important features (e.g., edges)
- Images: blurring, bilateral filter, edge detection, …
- Polygon meshes:
  - Curvature flow
  - Bilateral filter
  - Spectral filter

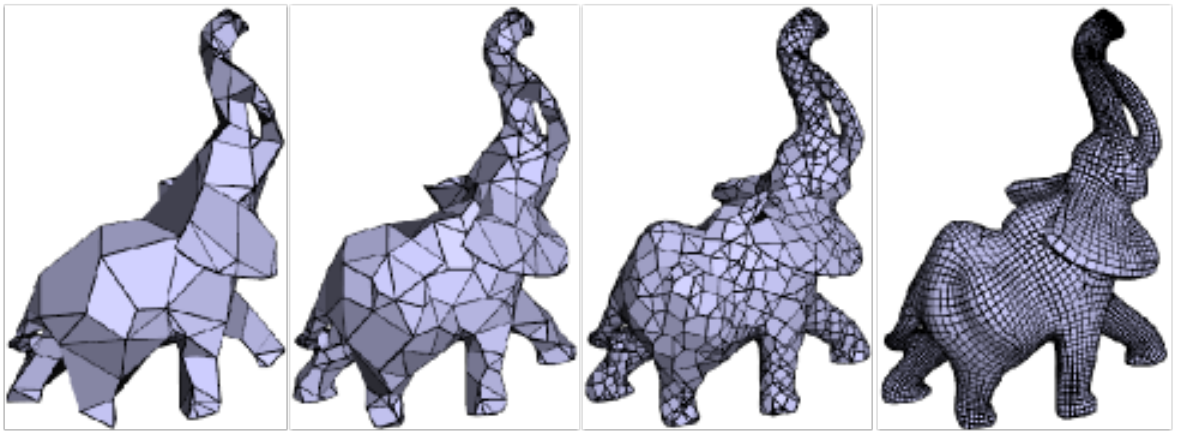




Geometry Processing: Compression

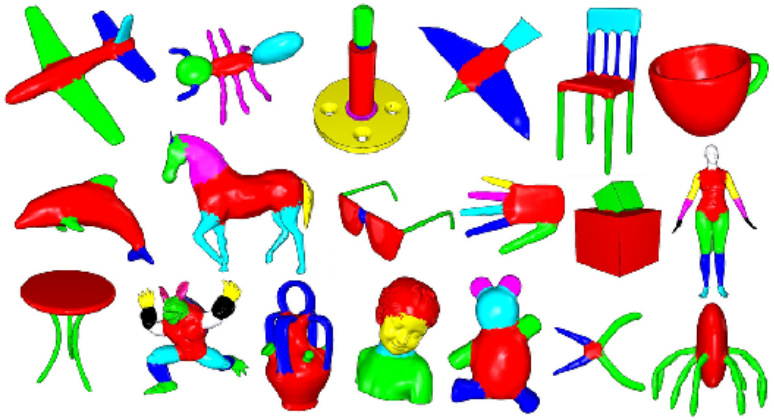
- Reduce storage size by eliminating redundant data/ approximating unimportant data
- Images:
  - Run-length, Huffman coding - lossless
  - Cosine/wavelet (JPEG/MPEG) - lossy
- Polygon meshes:
  - Compress geometry and connectivity
  - Many techniques (lossy & lossless)

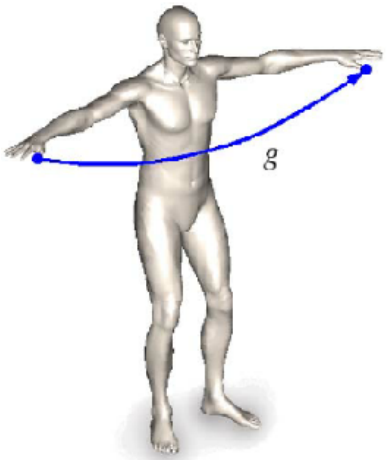




Geometry Processing: Shape Analysis

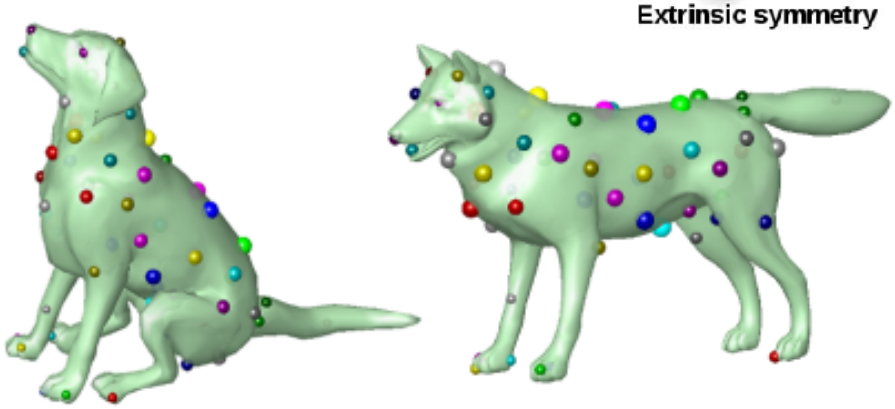
- Identify/ understand important semantic features
- Images: computer vision, segmentation, face detection, …
- Polygon meshes:
  - Segmentation, correspondence, symmetry detection, …





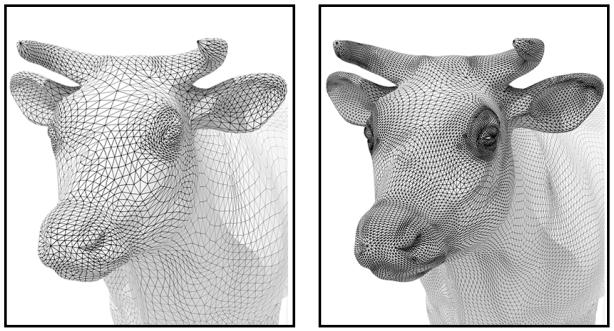
Extrinsic symmetry

Intrinsic symmetry



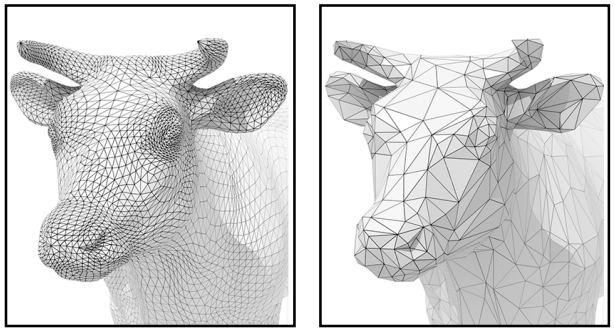
Geometry Processing Tasks: 3 Examples

Mesh Upsampling – Subdivision



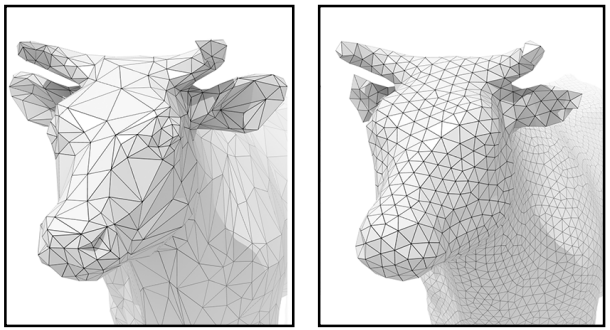
Increase resolution via subdivision

Mesh Downsampling – Simplification



Decrease resolution; try to preserve shape/appearance

Mesh Regularization

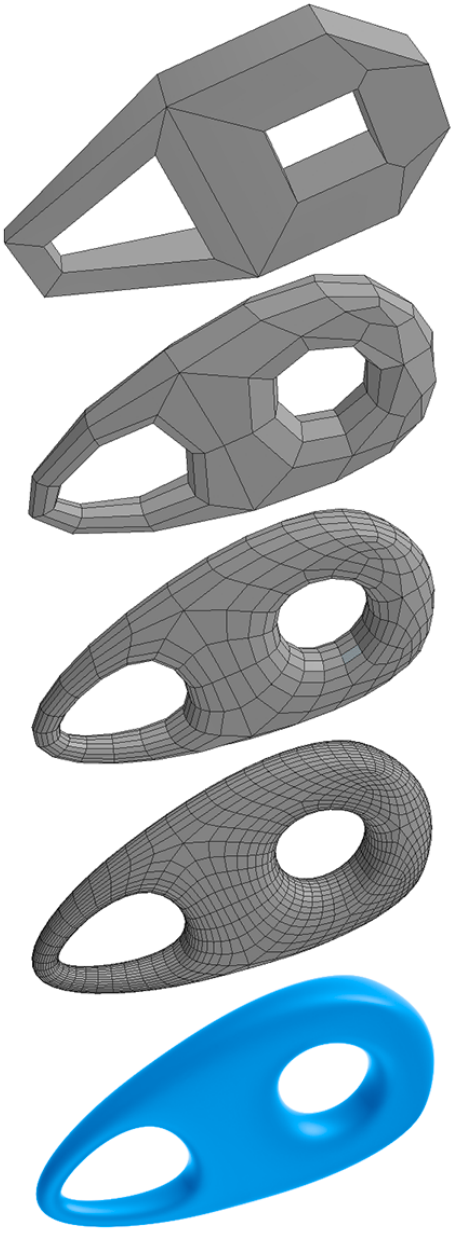


Modify sample distribution to improve quality

Subdivision Surfaces

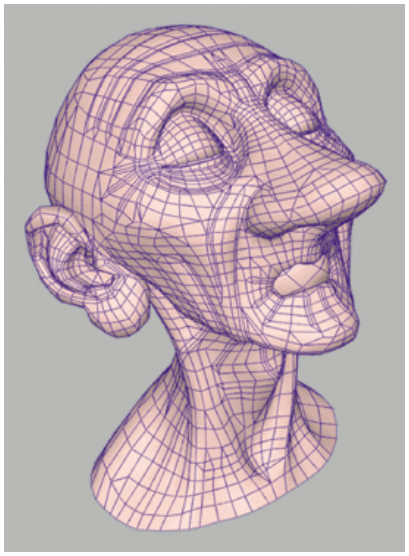
细分曲面

Subdivision Surfaces



Start with coarse polygon mesh ("control cage")

- Subdivide each element
- Update vertices via local averaging



Many possible rule:

- Catmull-Clark (quads)
- Loop (triangles)
- ...

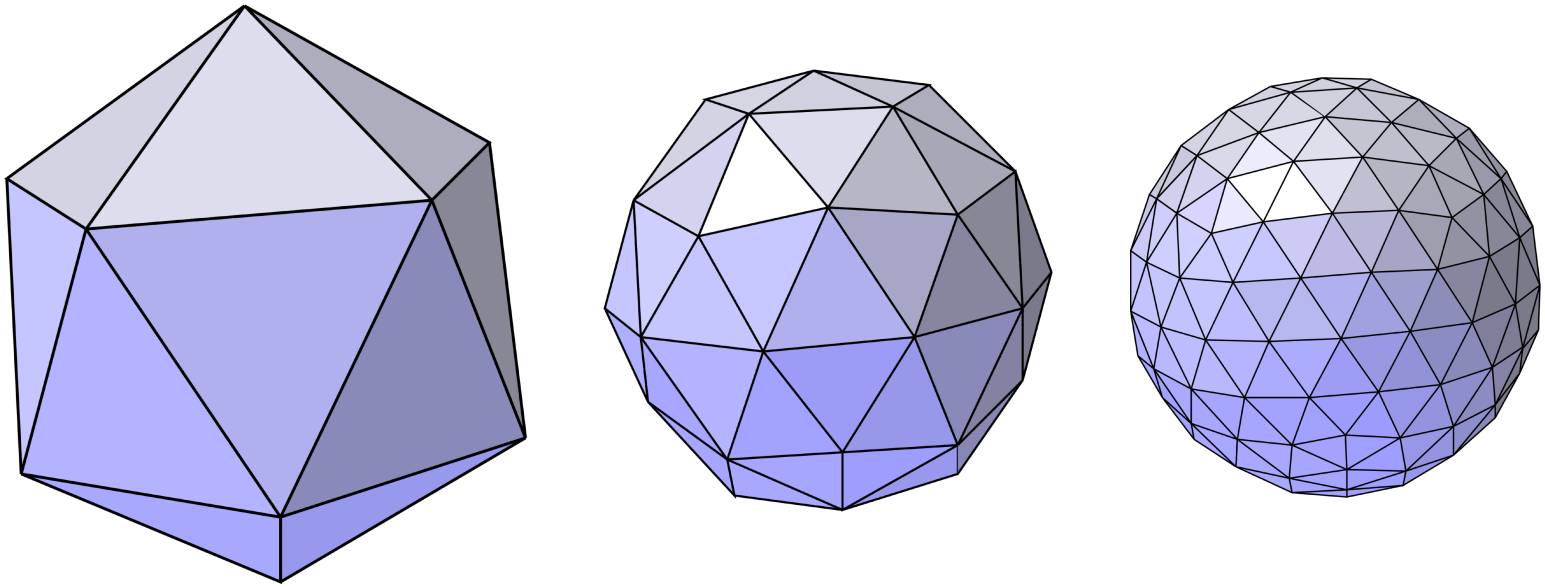
Common issues:

- interpolating or approximating?
- continuity at vertices?

Relatively easy for modeling; harder to guarantee continuity

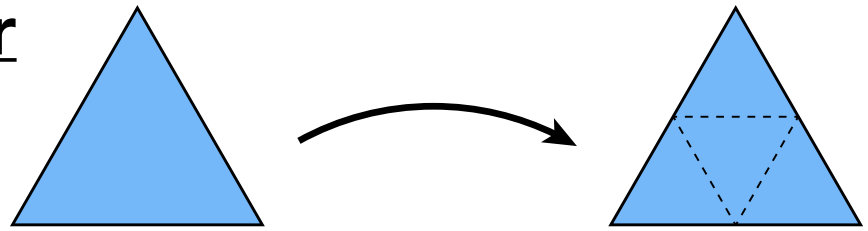
Loop Subdivision

Loop Subdivision



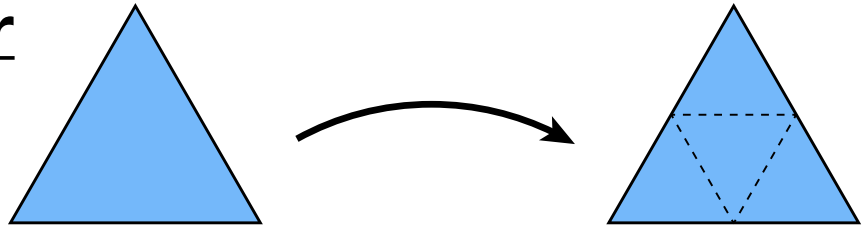
Loop Subdivision Algorithm

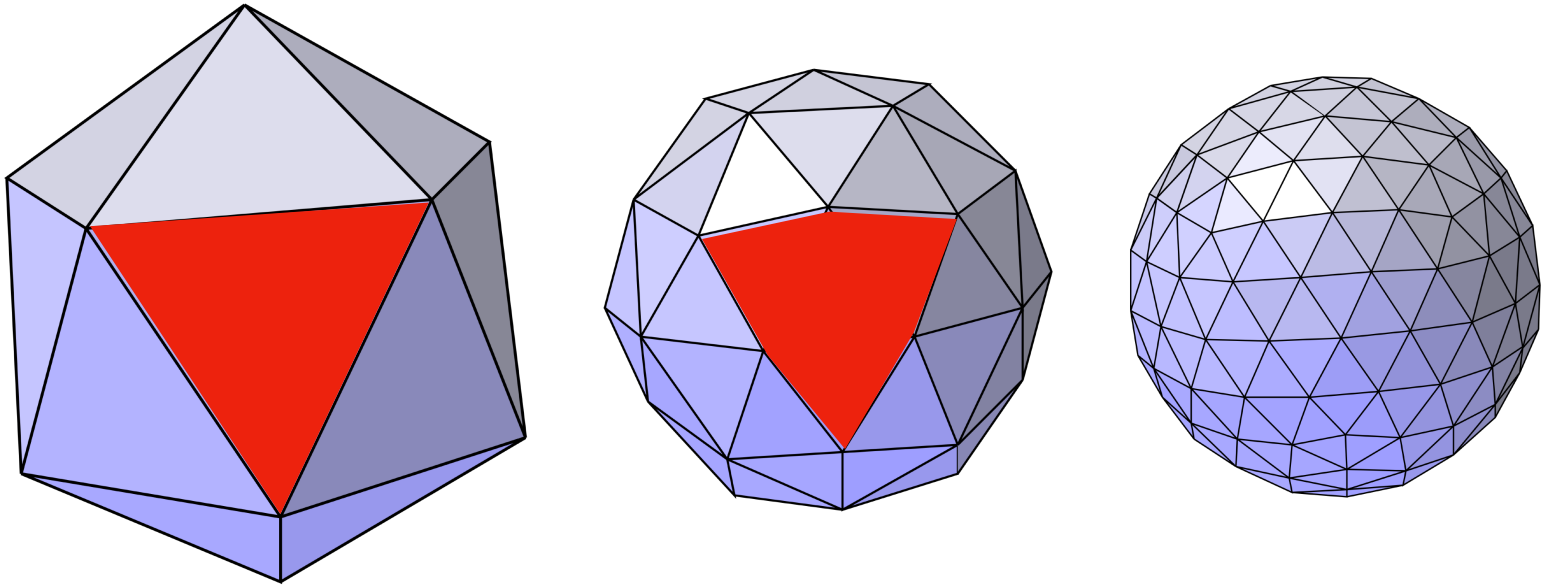
- Split each triangle into four



Loop Subdivision Algorithm

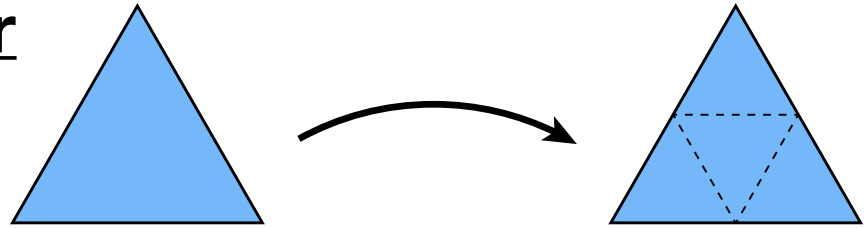
- Split each triangle into four



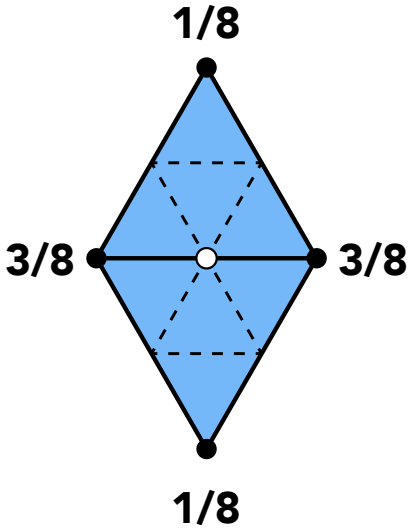


Loop Subdivision Algorithm

- Split each triangle into four



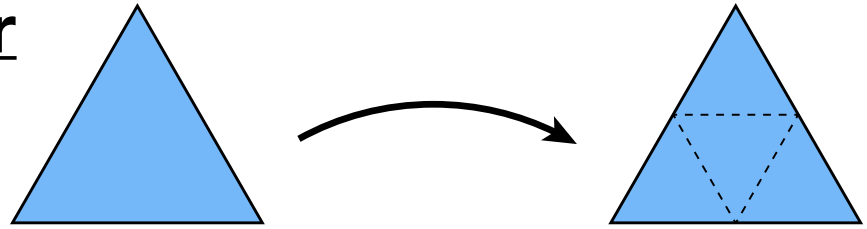
- Assign new vertex positions according to weights:



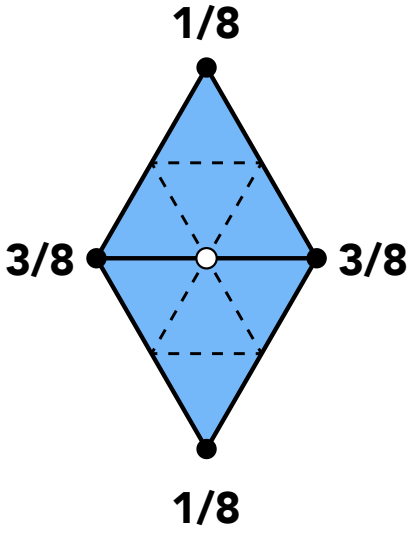
New vertices

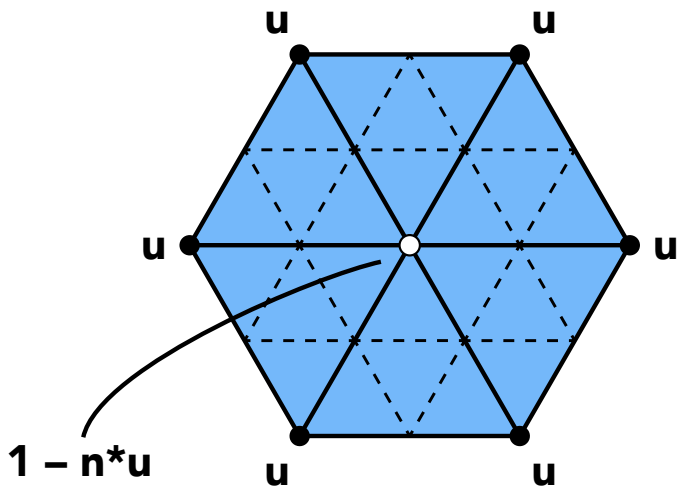
Loop Subdivision Algorithm

- Split each triangle into four



- Assign new vertex positions according to weights:





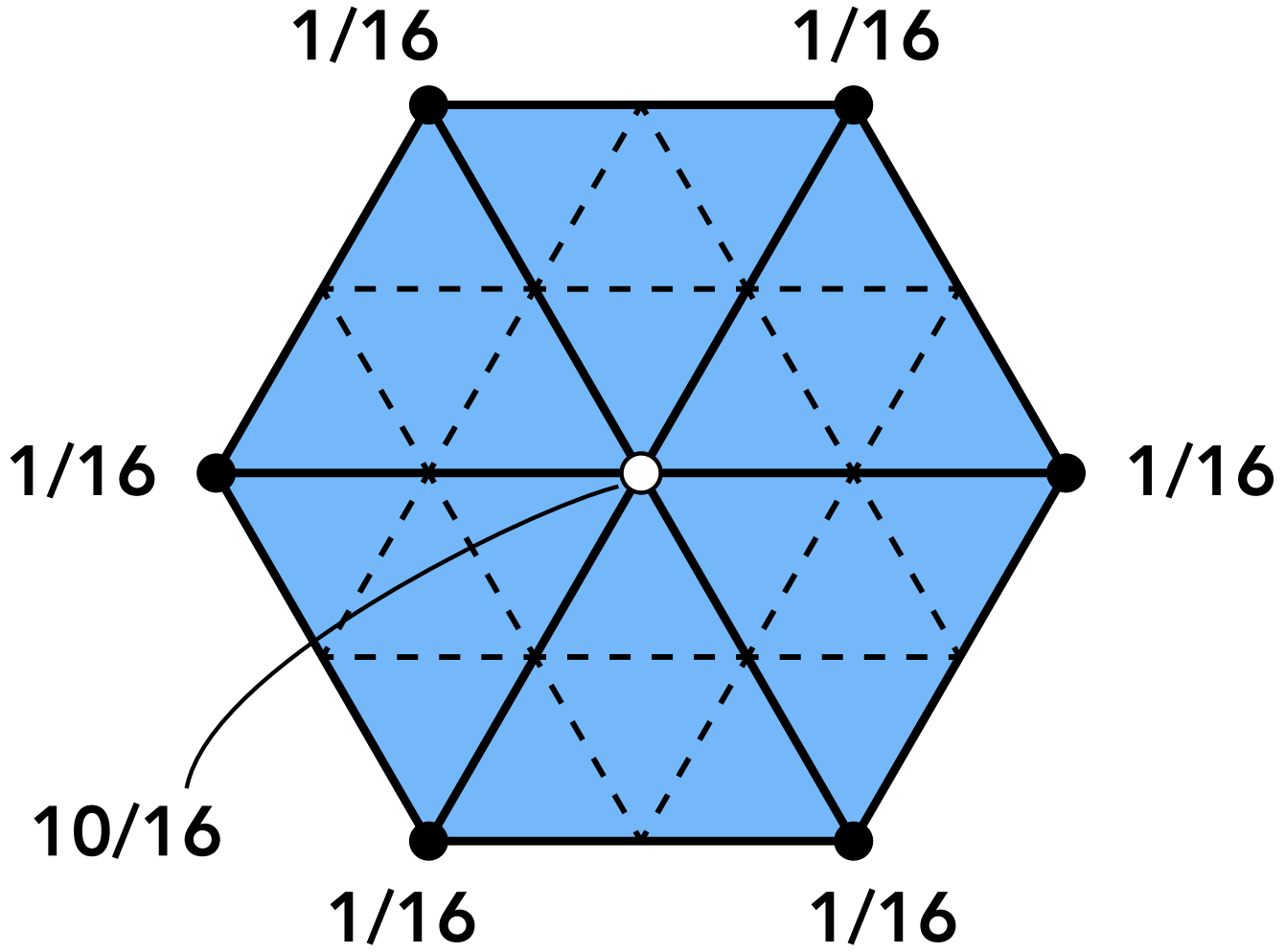
n: vertex degree

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

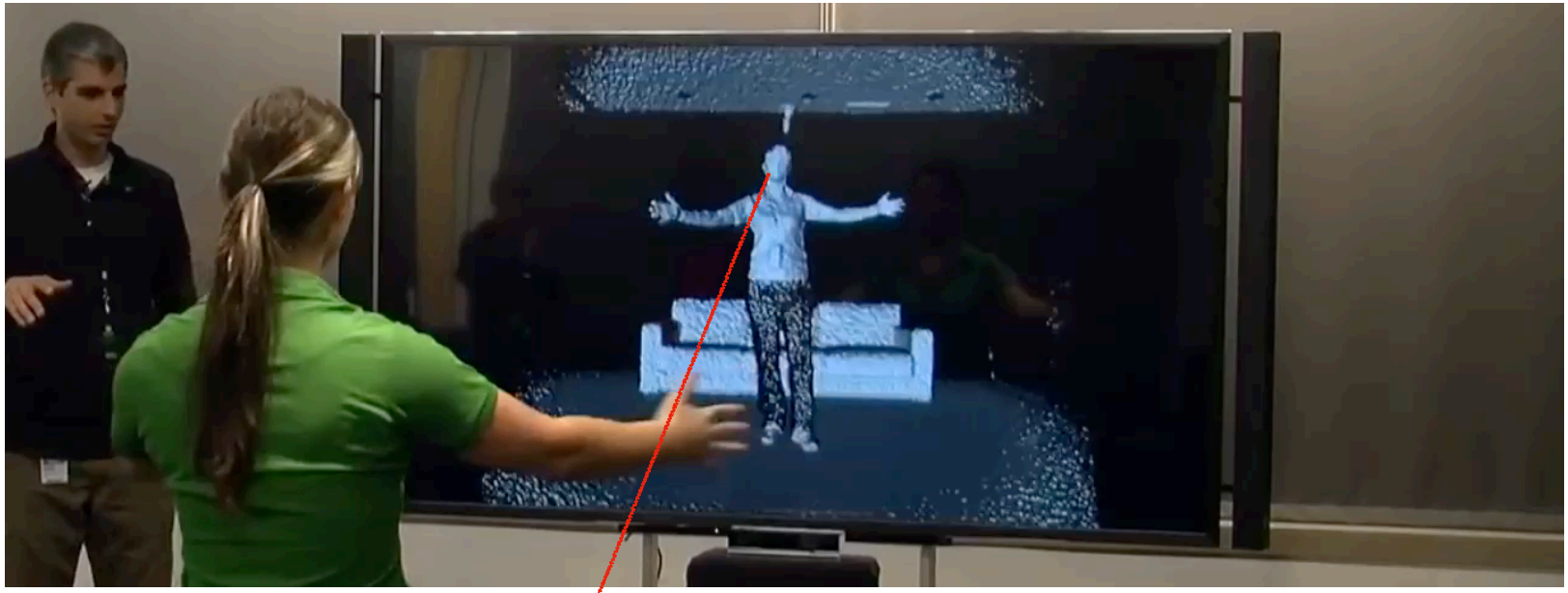
New vertices Old vertices

Loop Subdivision Algorithm

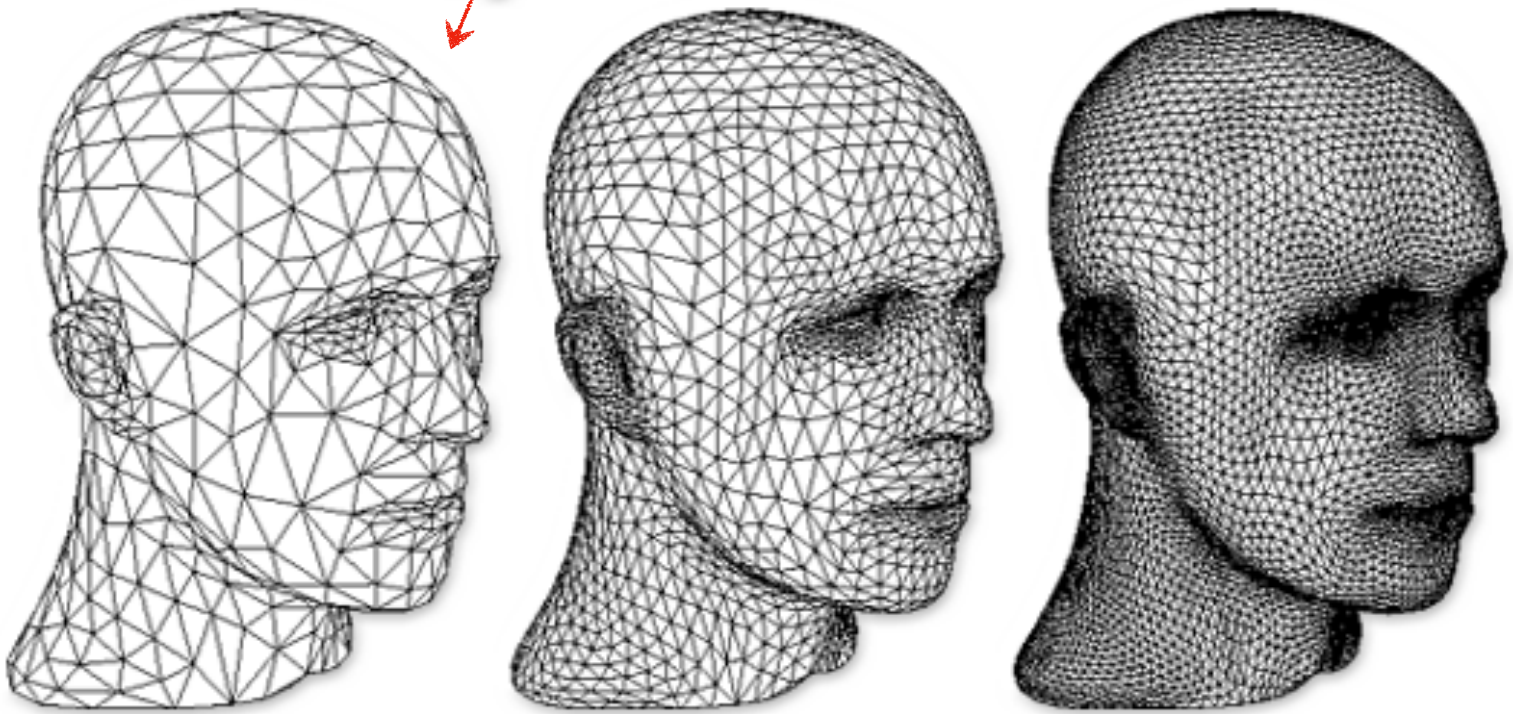
Example, for degree 6 vertices



Loop Subdivision Results

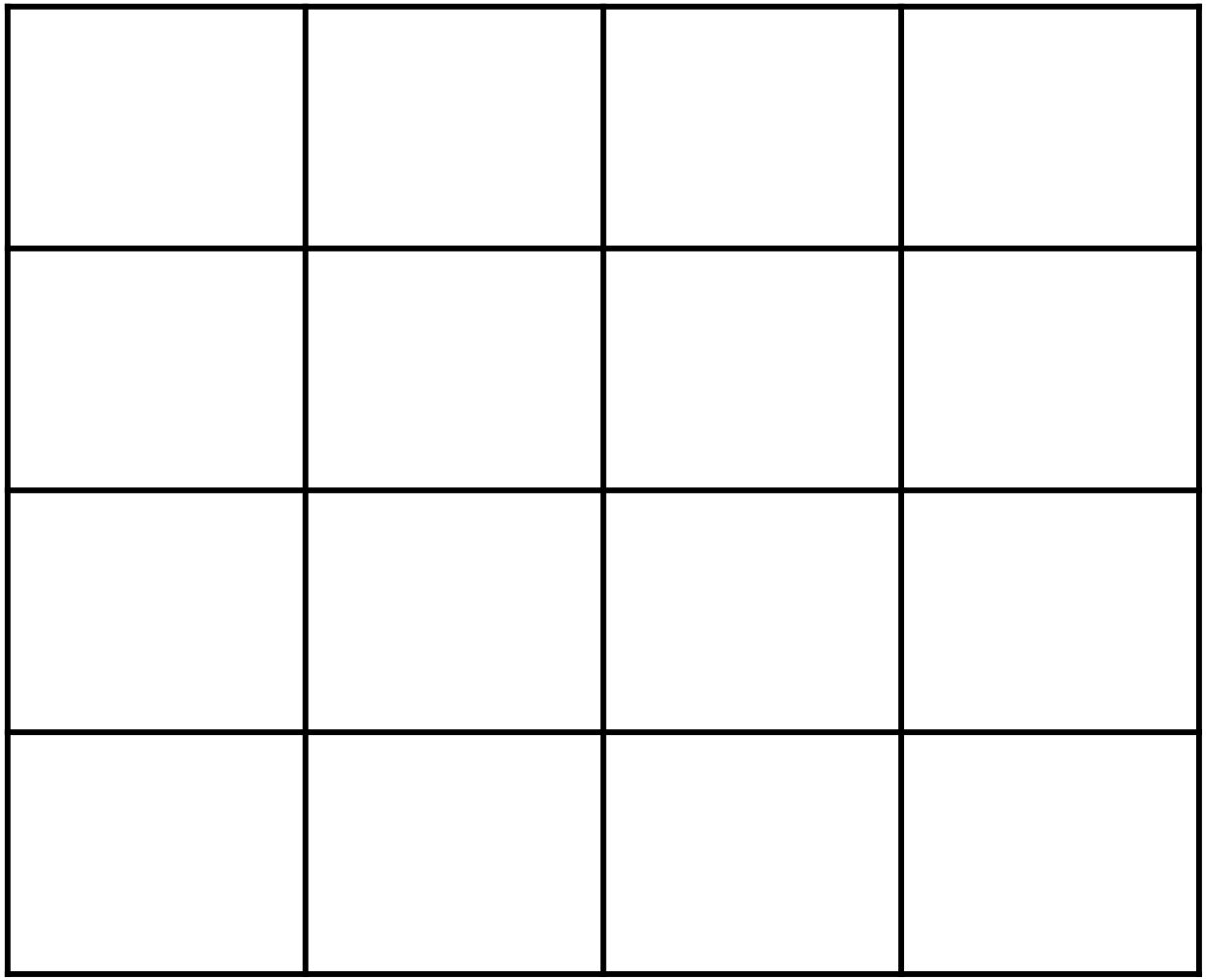


3D Scanning

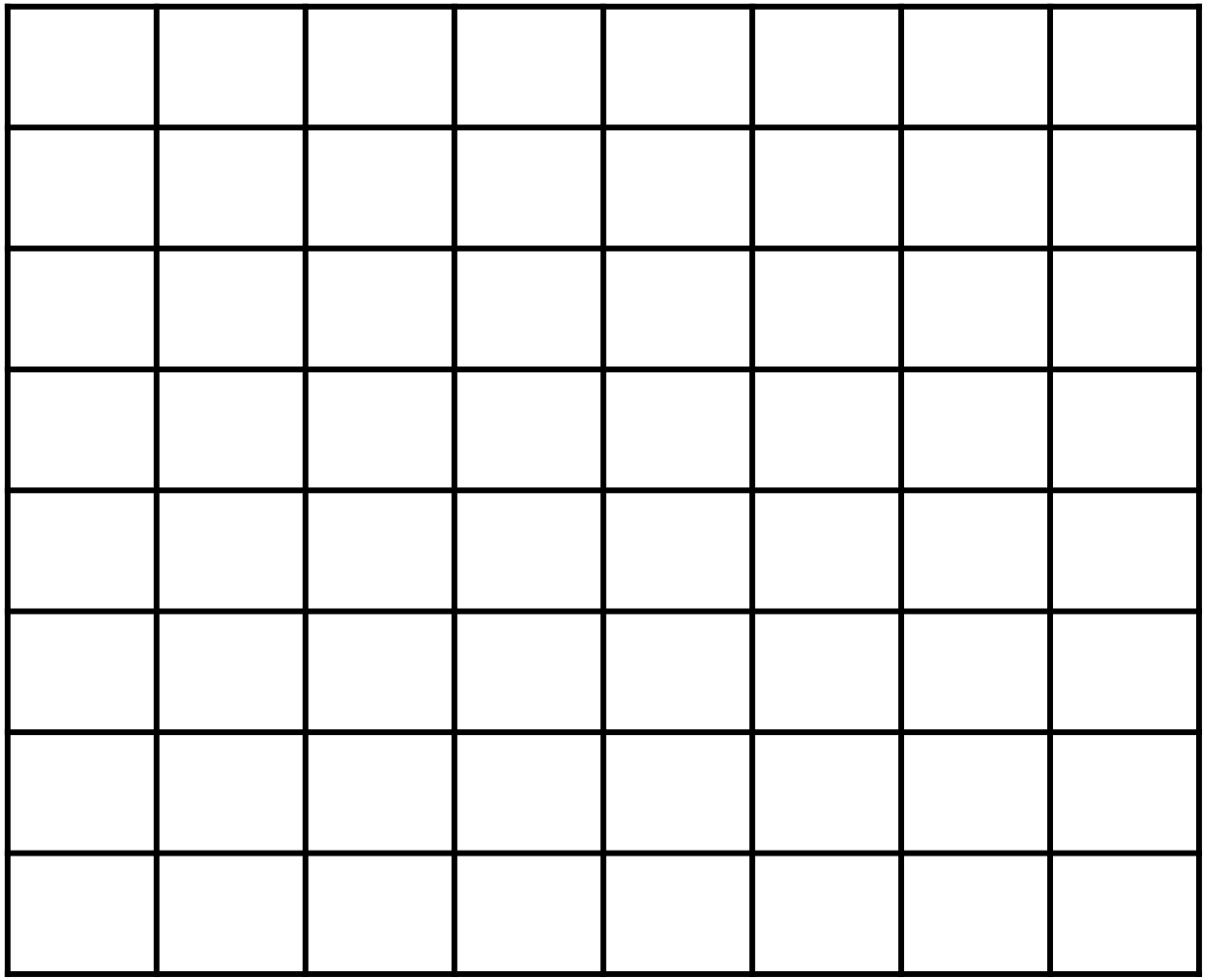


Catmull-Clark Subdivision

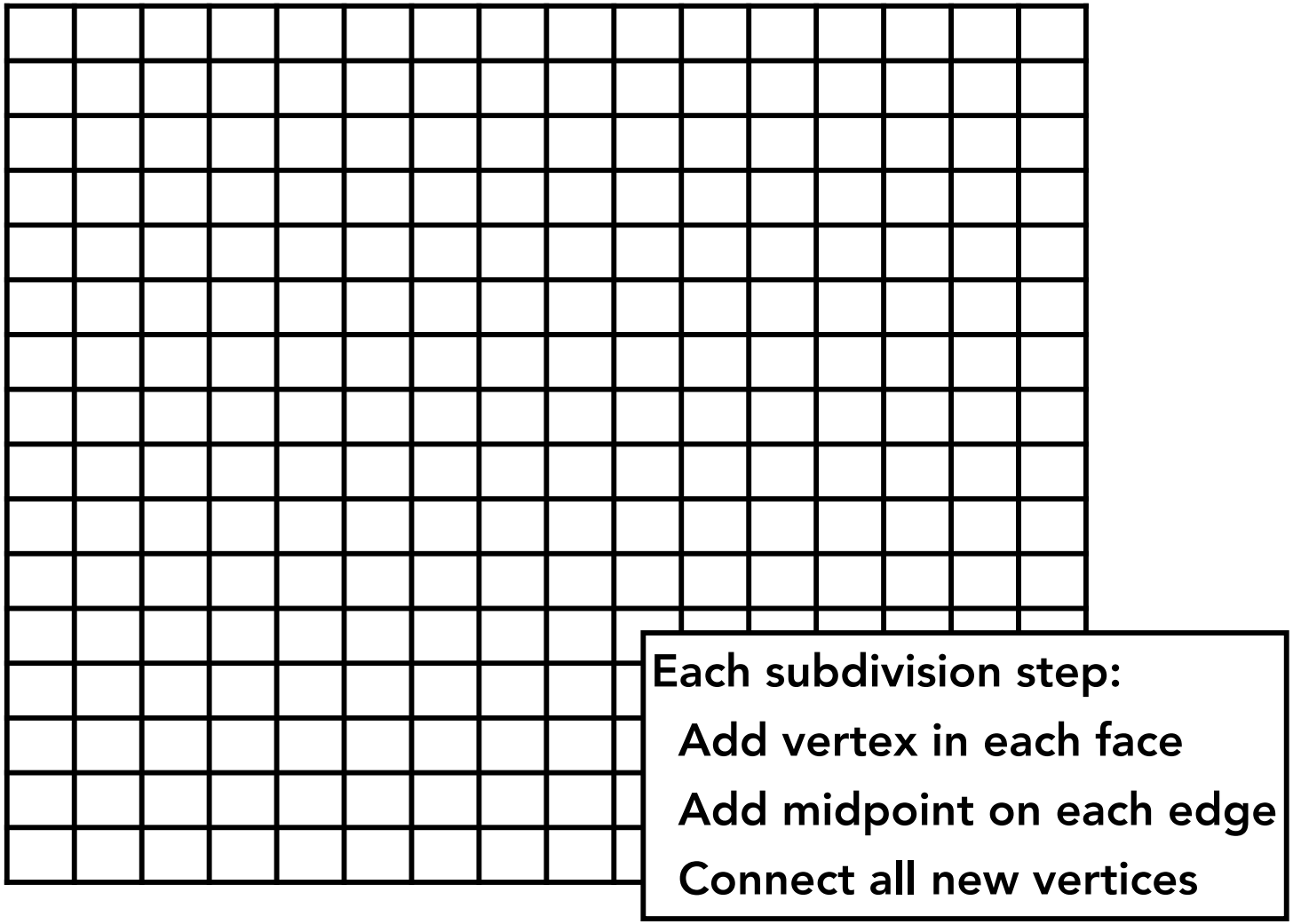
Catmull-Clark Subdivision (Regular Quad Mesh)



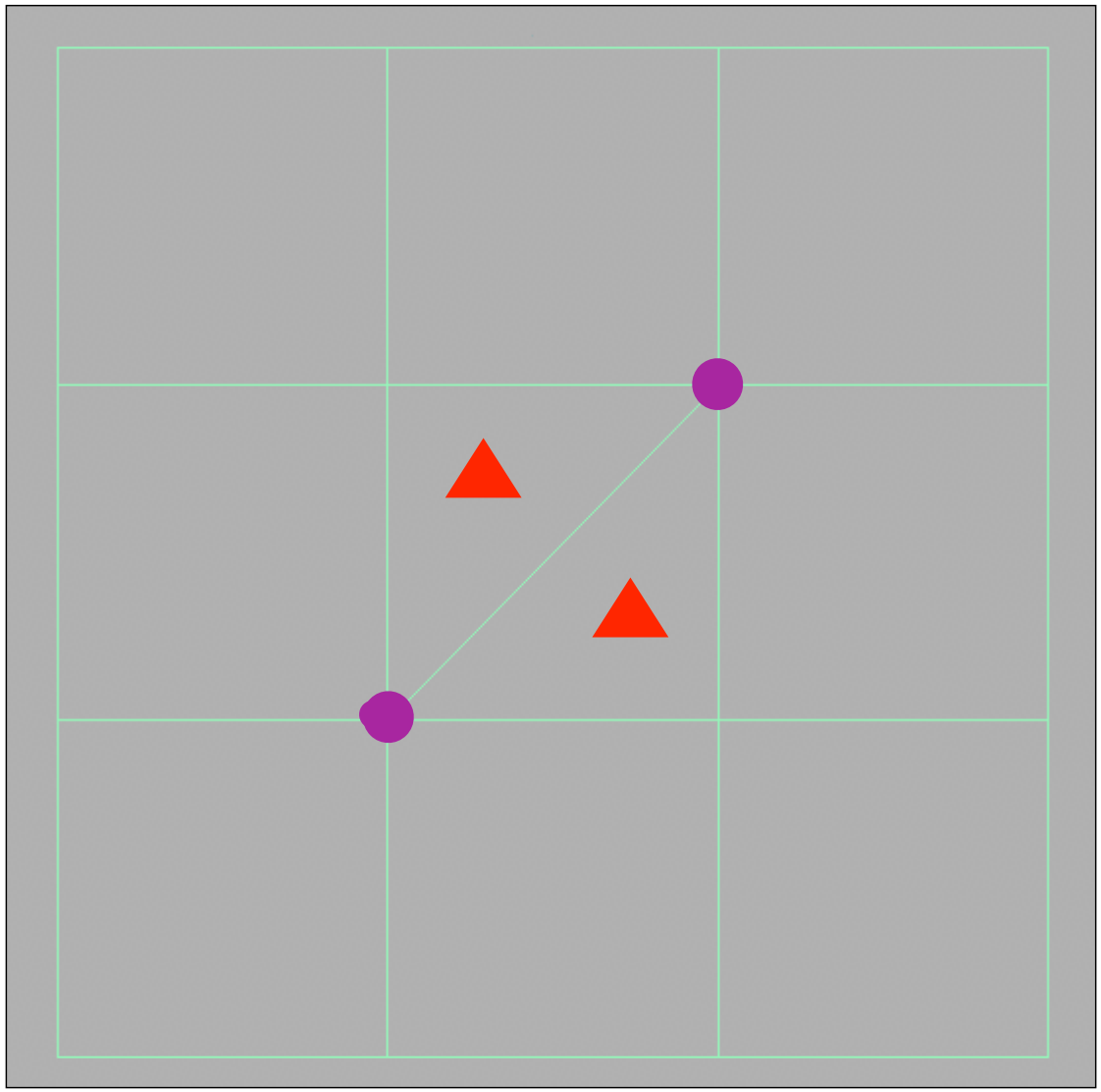
Catmull-Clark Subdivision (Regular Quad Mesh)



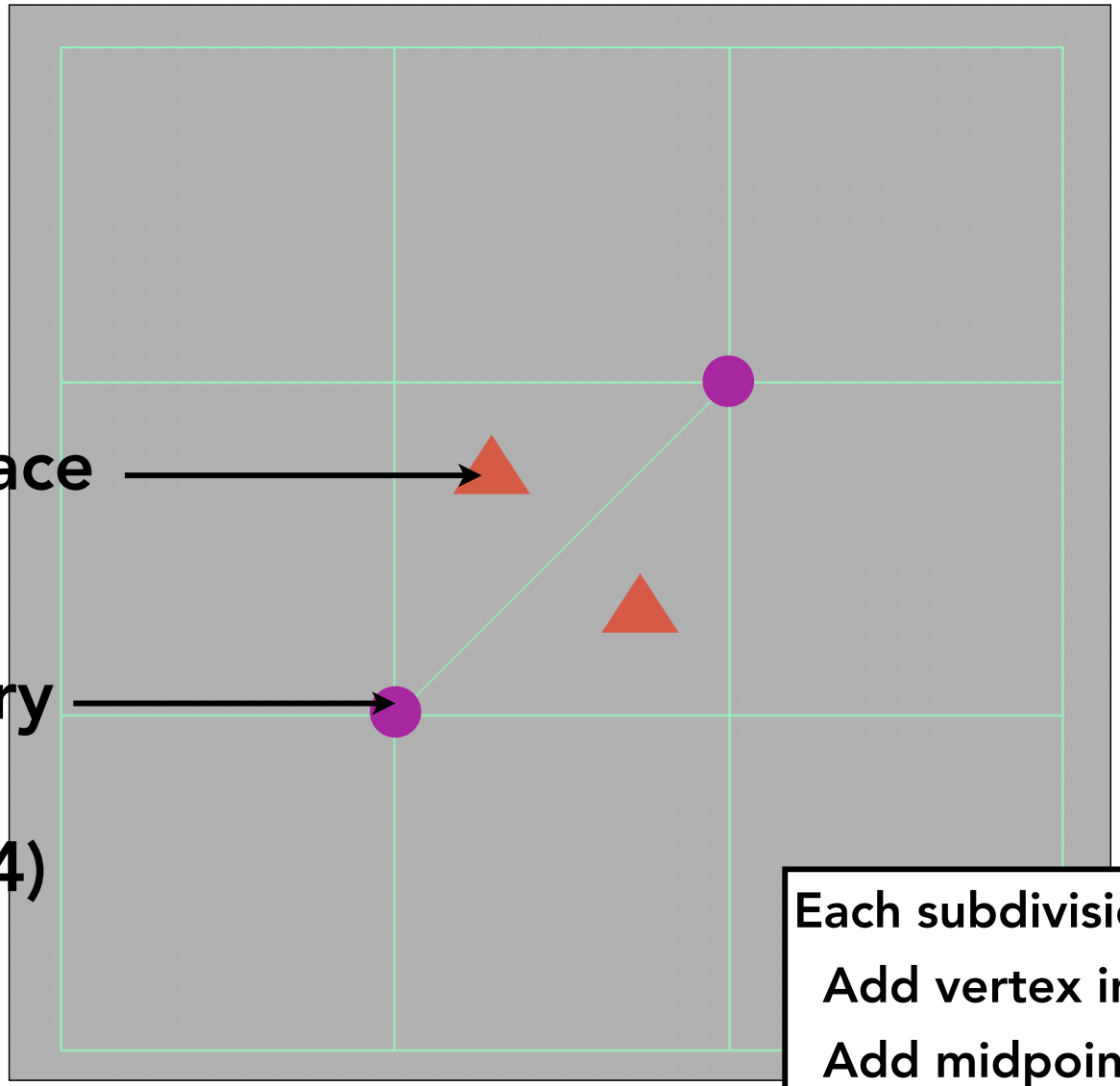
Catmull-Clark Subdivision (Regular Quad Mesh)



Catmull-Clark Subdivision (General Mesh)



Catmull-Clark Subdivision (General Mesh)



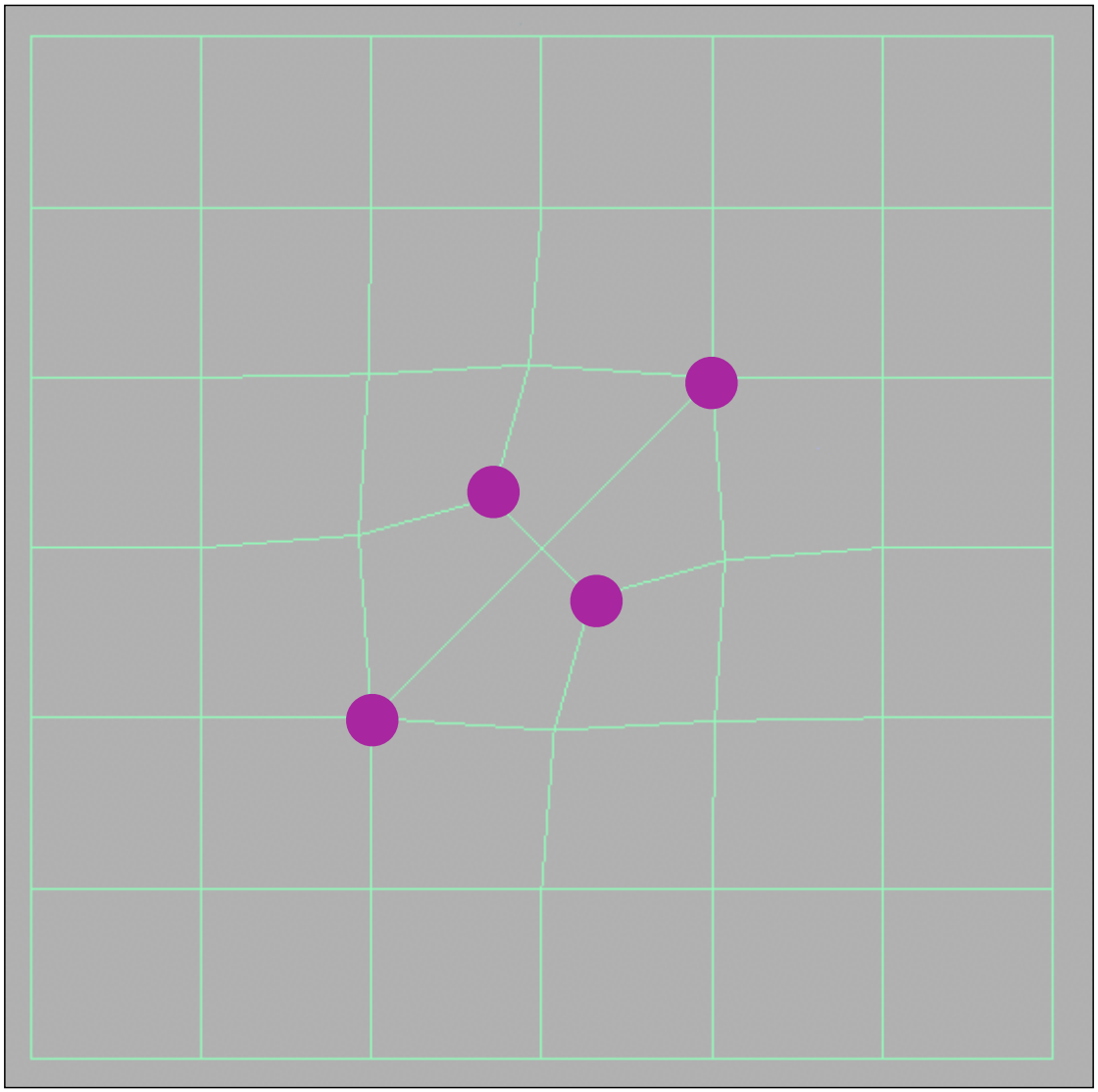
Non-quad face

Extraordinary vertex (valence != 4)

Each subdivision step:

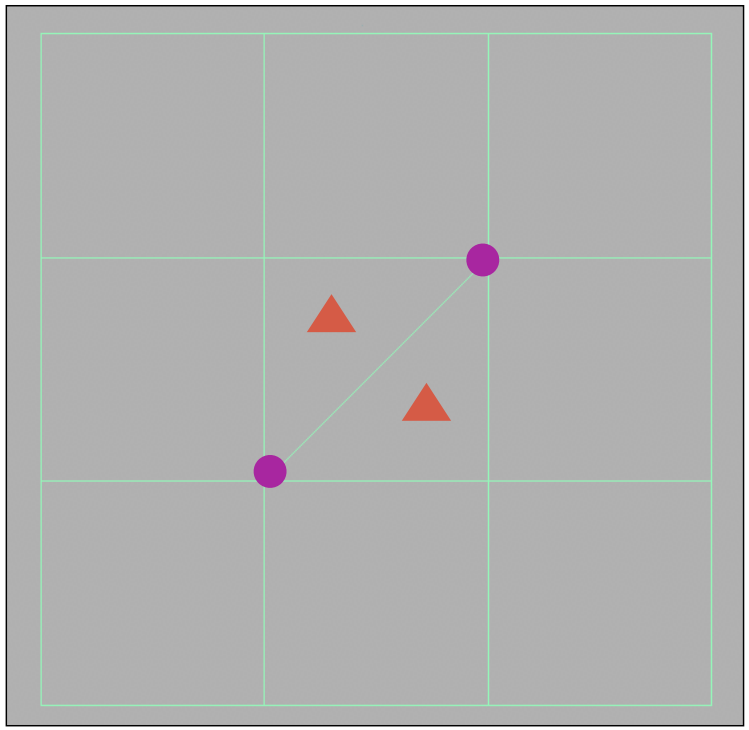
- Add vertex in each face
- Add midpoint on each edge
- Connect all new vertices

Catmull-Clark Subdivision (General Mesh)

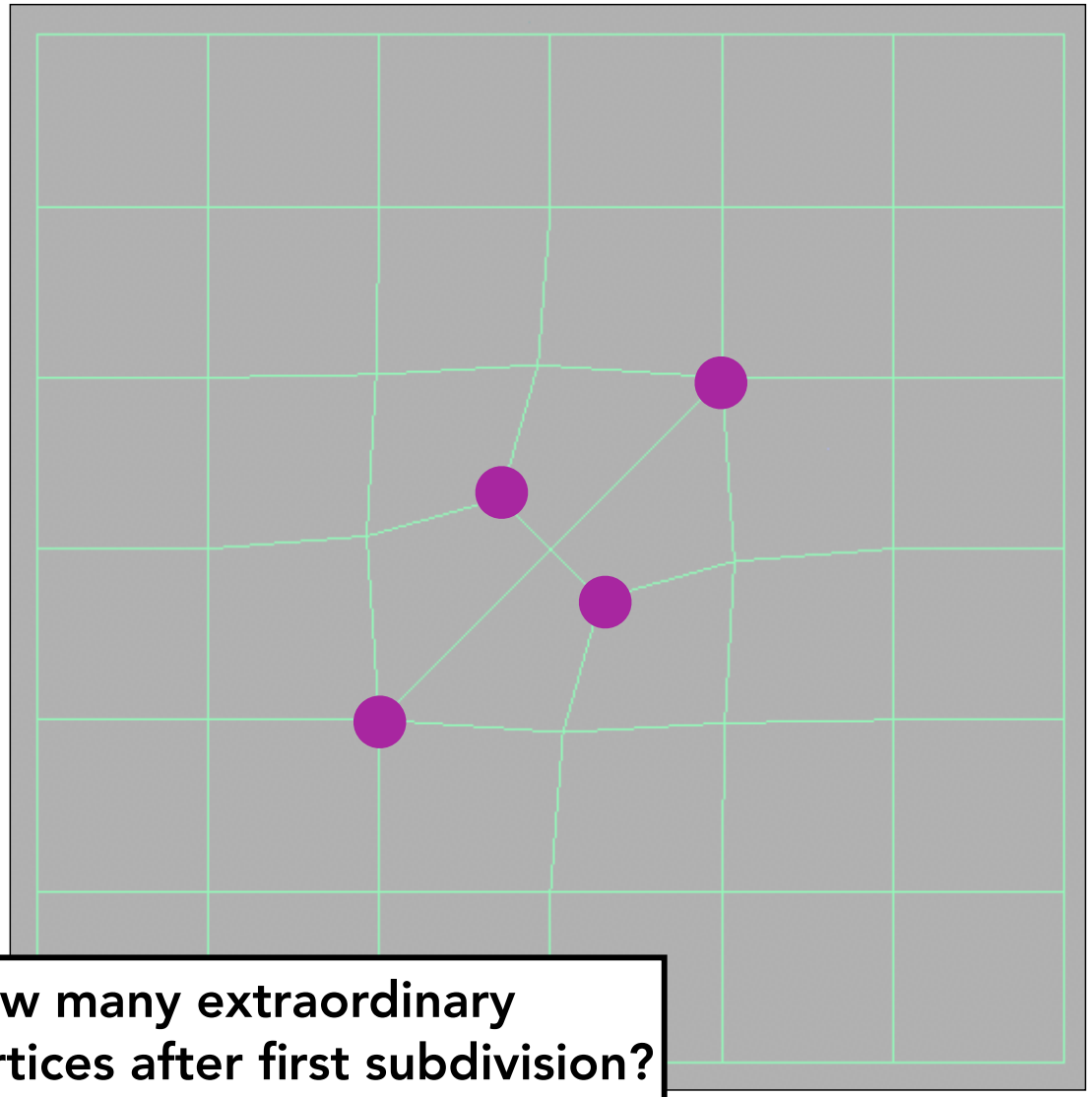


Catmull-Clark Subdivision (General Mesh)

Before



After

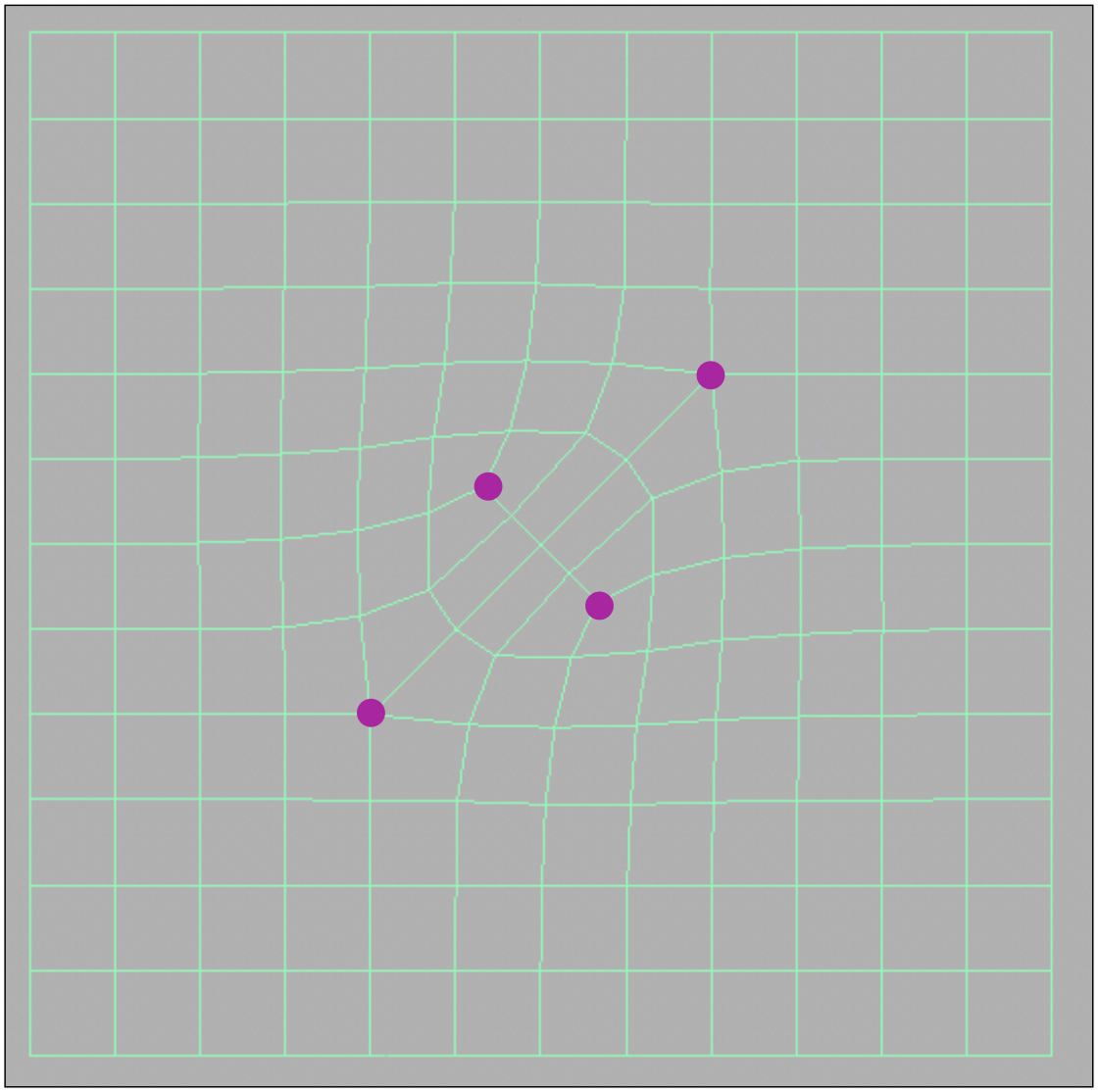


How many extraordinary vertices after first subdivision?

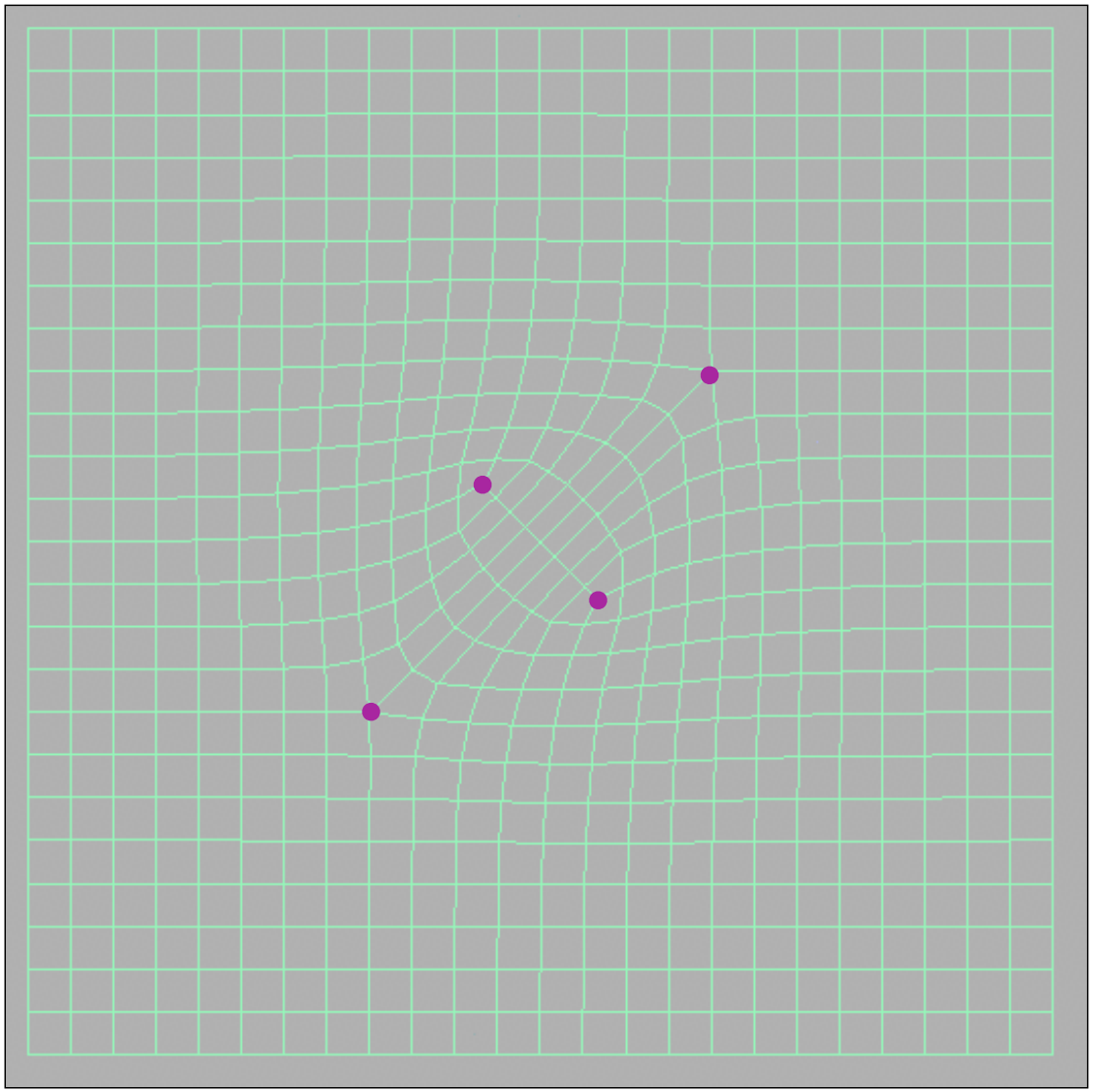
What are their valences?

How many non-quad faces?

Catmull-Clark Subdivision (General Mesh)

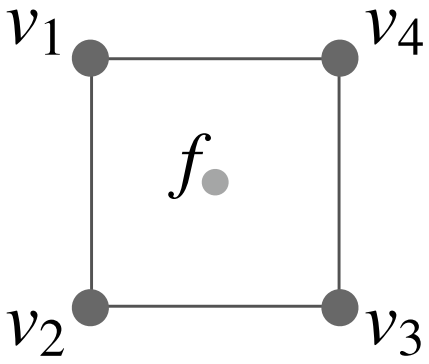


Catmull-Clark Subdivision (General Mesh)



Catmull-Clark Vertex Update Rules (Quad Mesh)

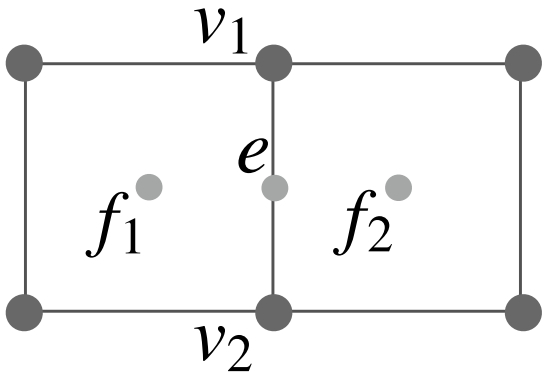
Face point

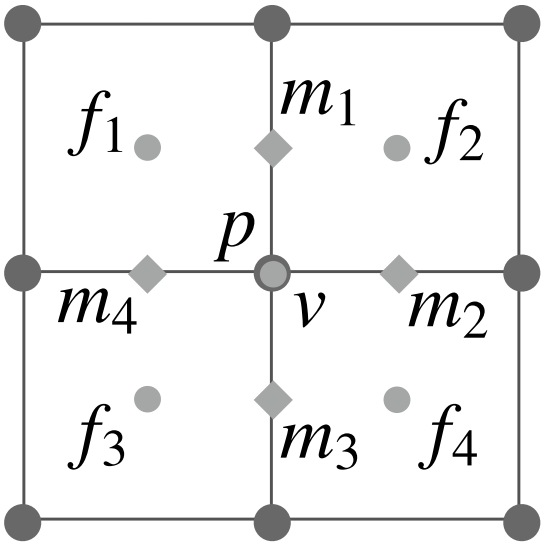


$$f = \frac{v_1 + v_2 + v_3 + v_4}{4}$$

$$e = \frac{v_1 + v_2 + f_1 + f_2}{4}$$

Edge point





Vertex point

$$v = \frac{f_1 + f_2 + f_3 + f_4 + 2(m_1 + m_2 + m_3 + m_4) + 4p}{16}$$

m midpoint of edge, not "edge point"

p old "vertex point"

Catmull-Clark Vertex Update Rules (General Mesh)

$f =$ average of surrounding vertices

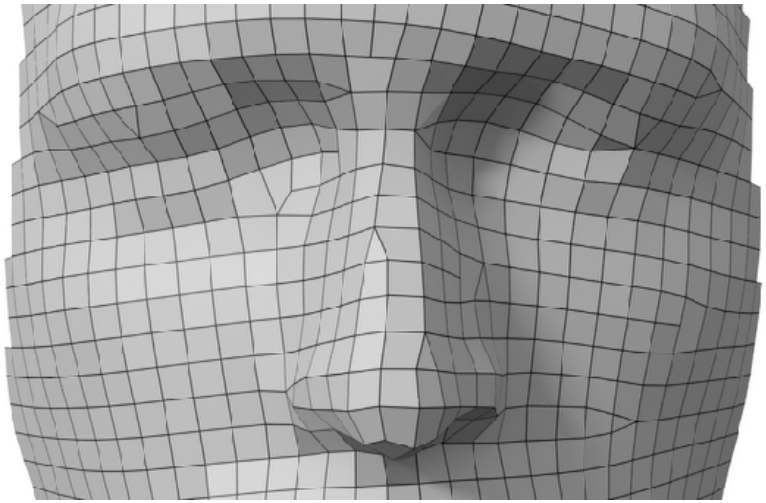
$$e = \frac{f_1 + f_2 + v_1 + v_2}{4}$$

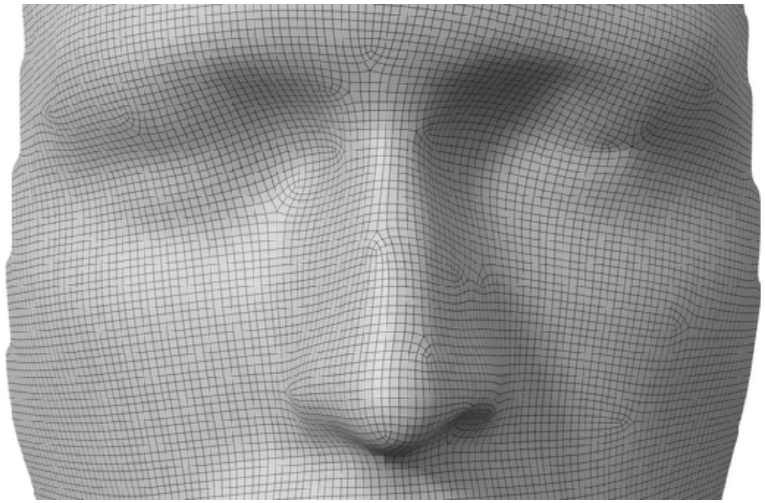
$$v = \frac{\bar{f}}{n} + \frac{2\bar{m}}{n} + \frac{p(n-3)}{n}$$

These rules reduce to earlier quad rules for ordinary vertices / faces

$\bar{m} =$ average of adjacent midpoints
$\bar{f} =$ average of adjacent face points
$n =$ valence of vertex
$p =$ old "vertex" point

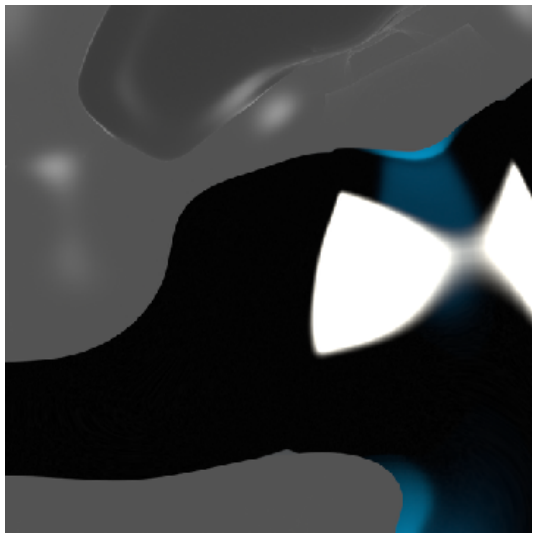
Catmull-Clark Subdivision Results



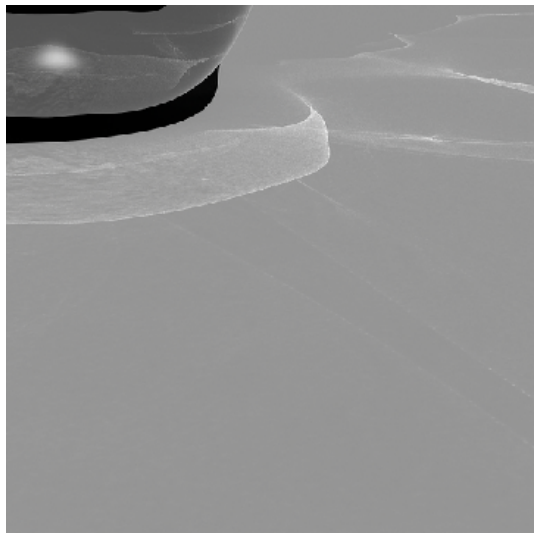


(Very few irregular vertices)

Good normal approximation almost everywhere:

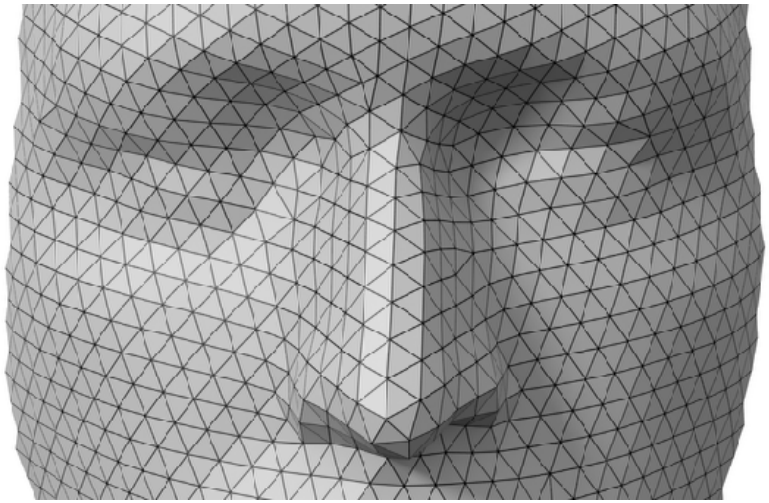


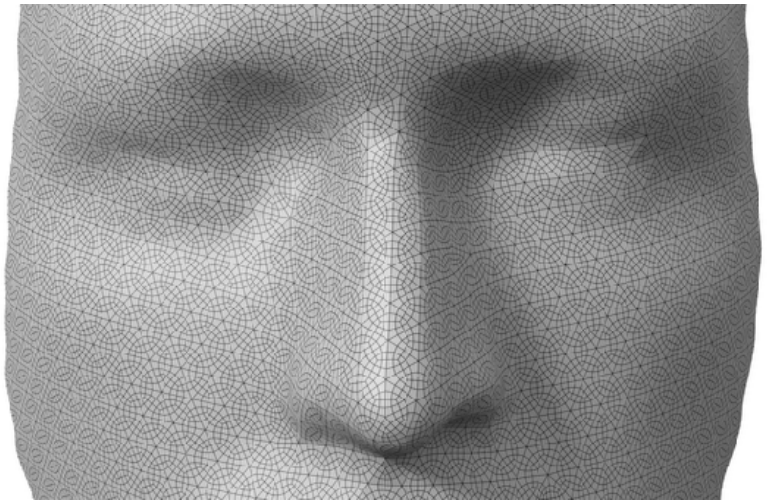
Smooth
reflection lines



Smooth
caustics

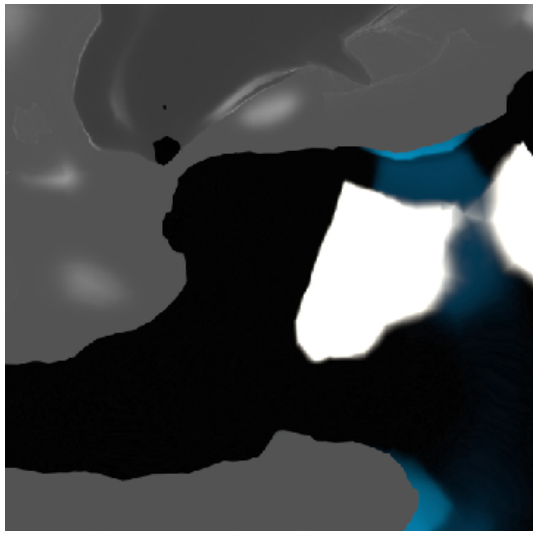
Catmull-Clark on triangle mesh



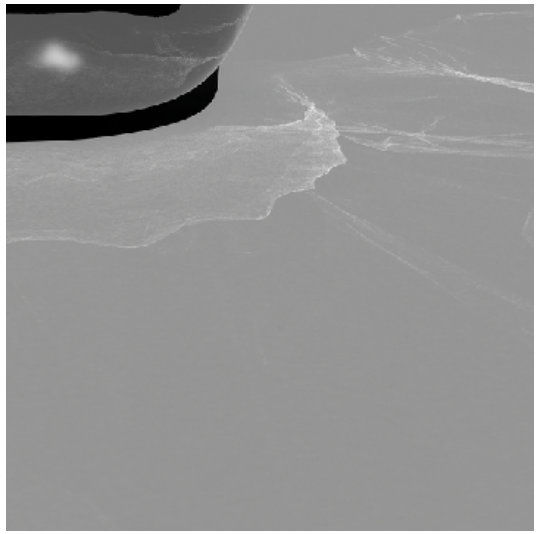


(Huge number of irregular vertices!)

Poor normal approximation almost everywhere:



Jagged
reflection lines

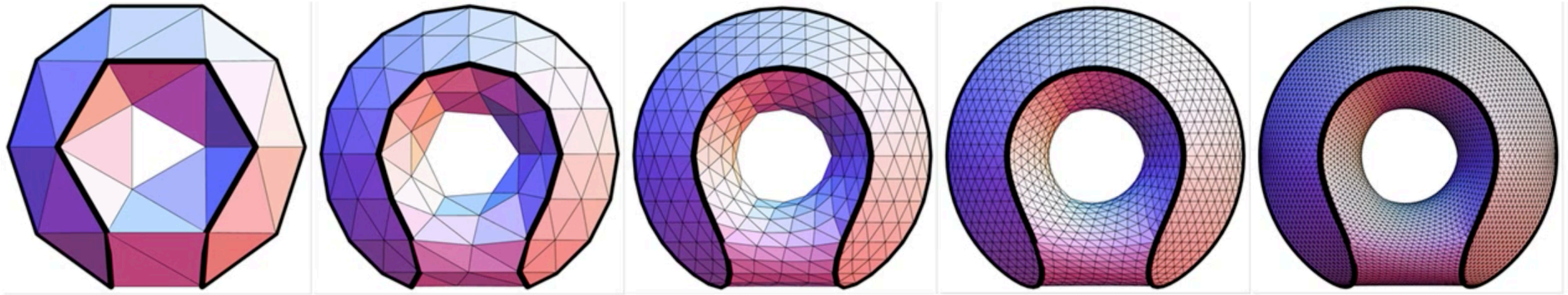


Jagged
caustics

ALIASING!

What About Sharp Creases?

Loop with Sharp Creases



Catmull-Clark with Sharp Creases

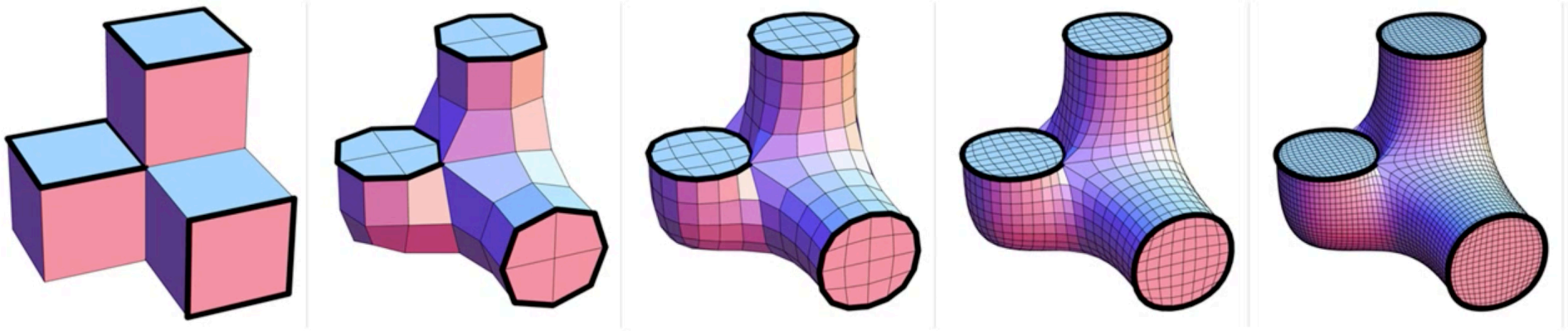
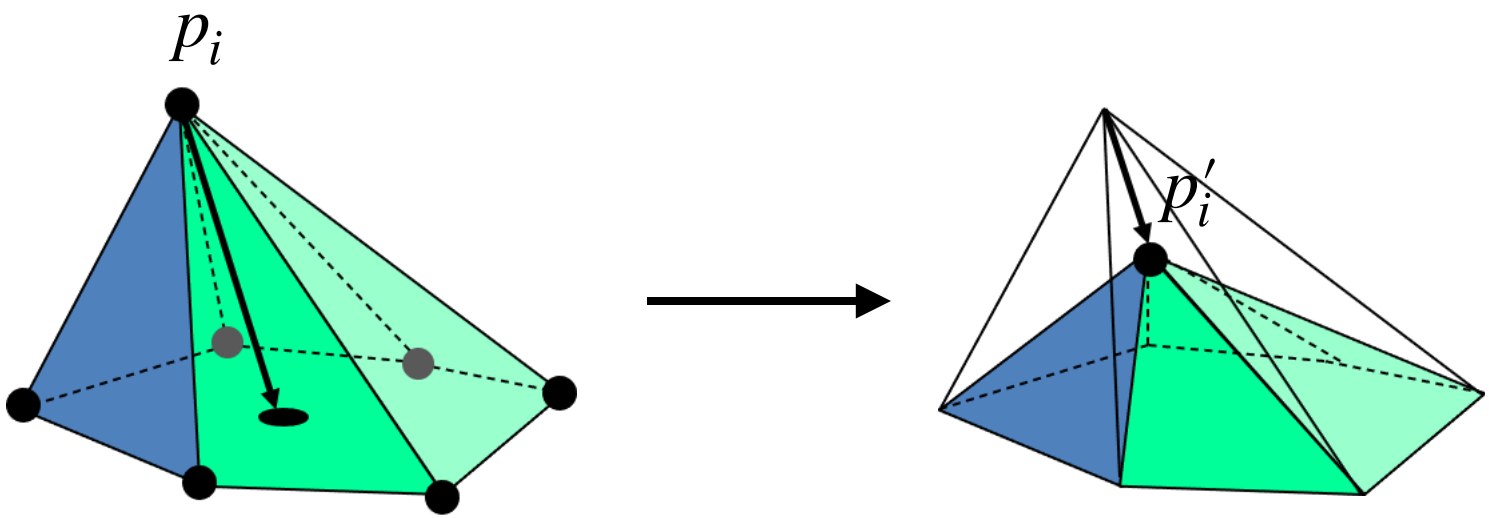


Figure from: Hakenberg et al. Volume Enclosed by Subdivision Surfaces with Sharp Creases

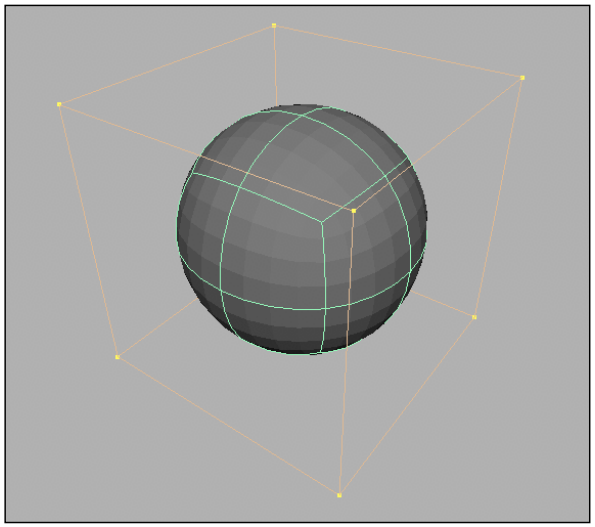
Update vertex positions in 3D

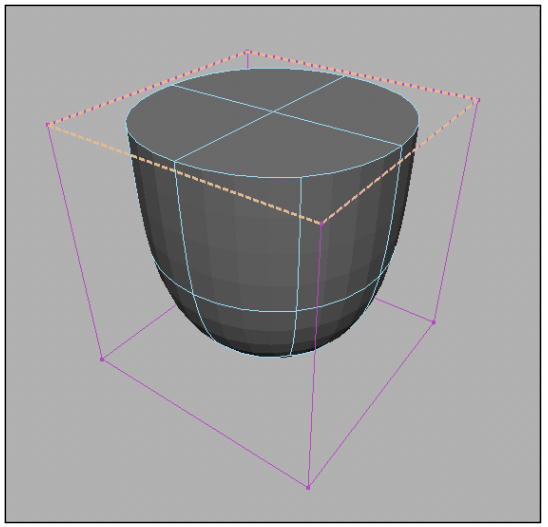


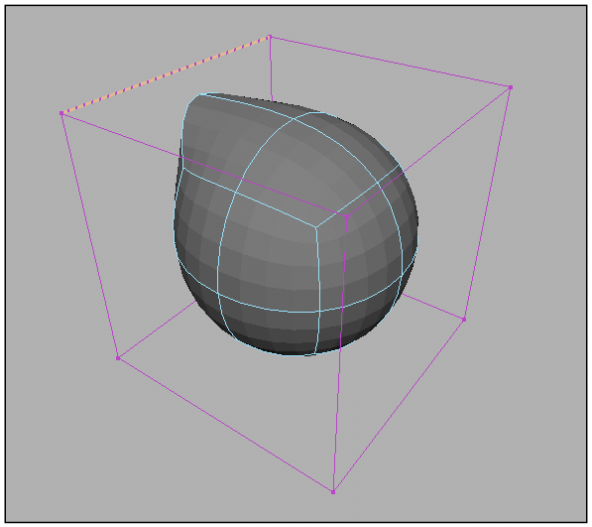
$$p_i' = \alpha p_i + (1 - \alpha)\Sigma(p_{ij})$$

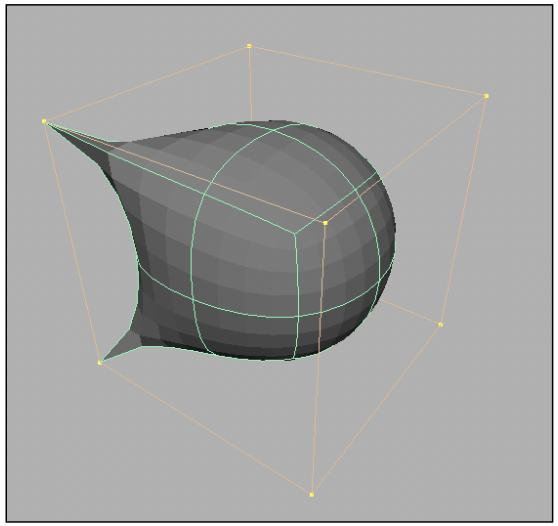
What About Sharp Creases?

Sharp vertices, edges, and faces





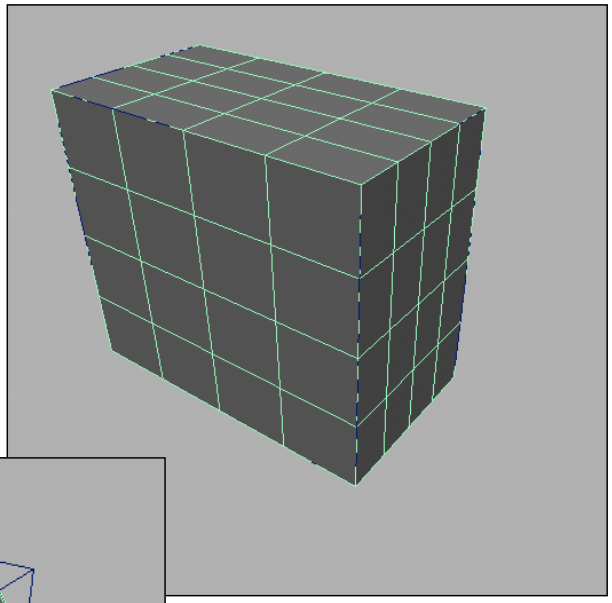


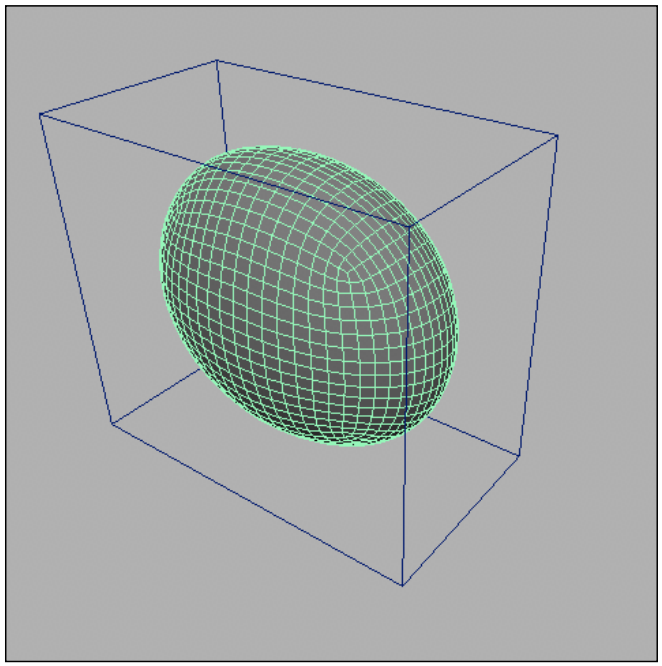


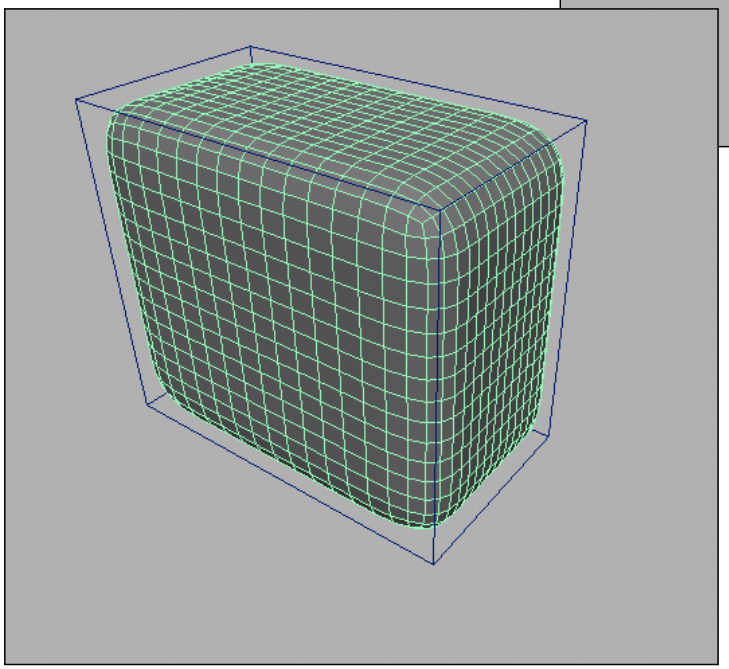
What About Controlled Smoothing?

Partial sharpness

Use sharp rule for N levels, then use smooth rules.



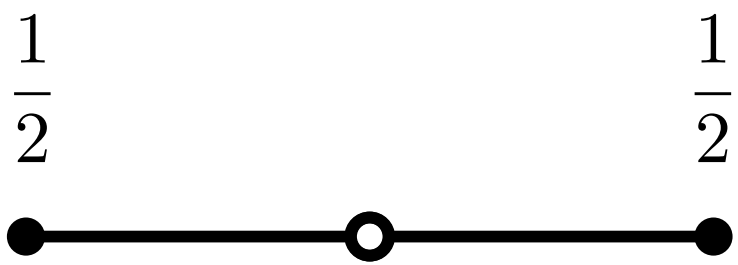




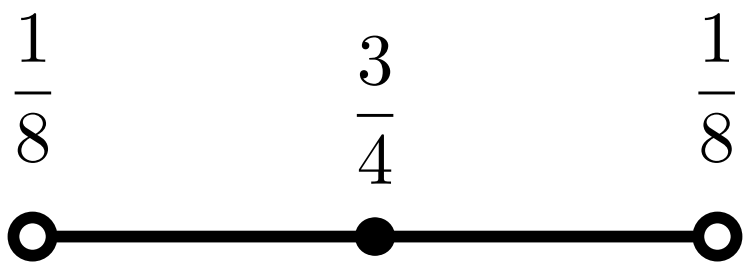
Creases + Boundaries

Can create creases in subdivision surfaces by marking certain edges as “sharp”. Boundary edges can be handled the same way

- Use different subdivision rules for vertices along these “sharp” edges

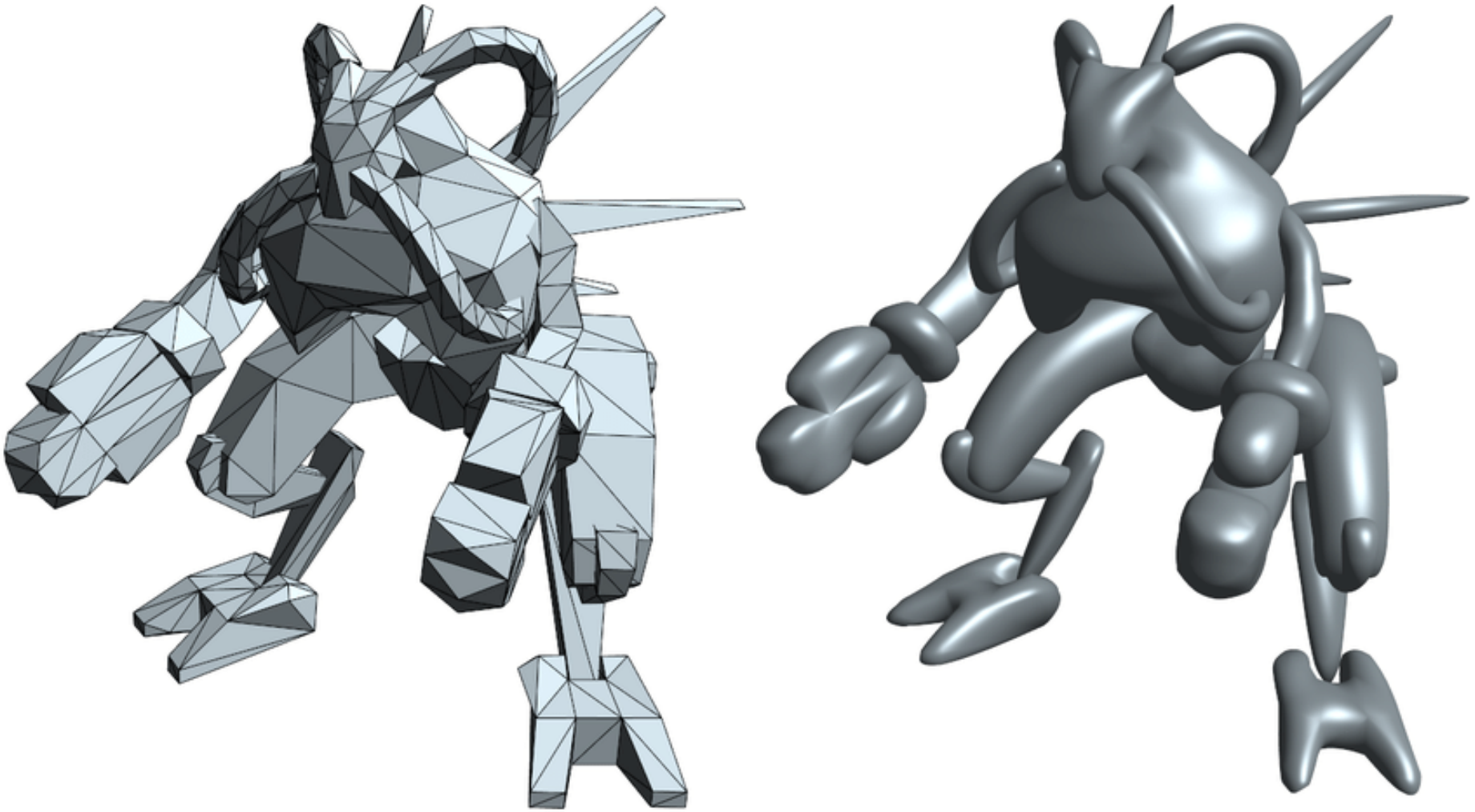


Insert new midpoint vertex, weights as shown



Update existing vertices, weights as shown

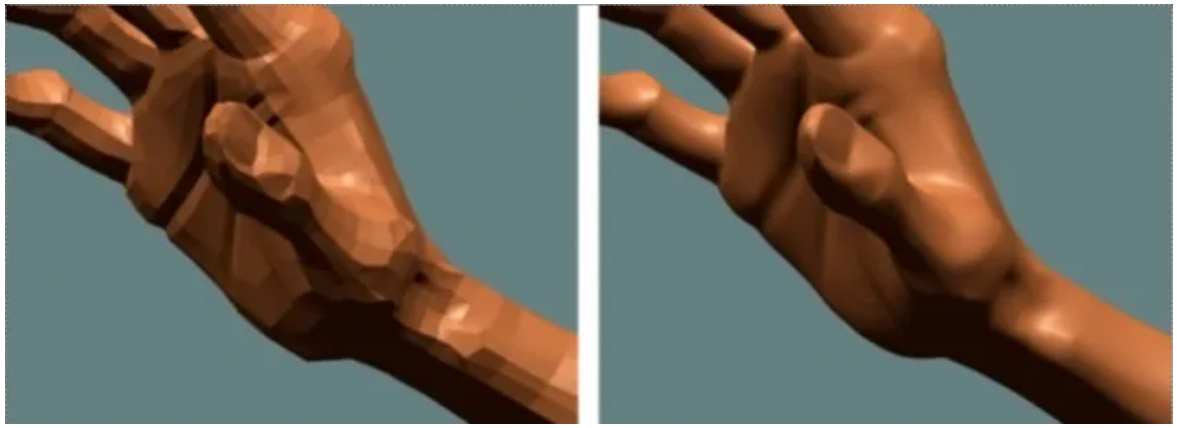
Subdivision in Modeling



Subdivision in Action ("Geri's Game", Pixar)

Subdivision used for entire character:

- Hands and head
- Clothing, tie, shoes



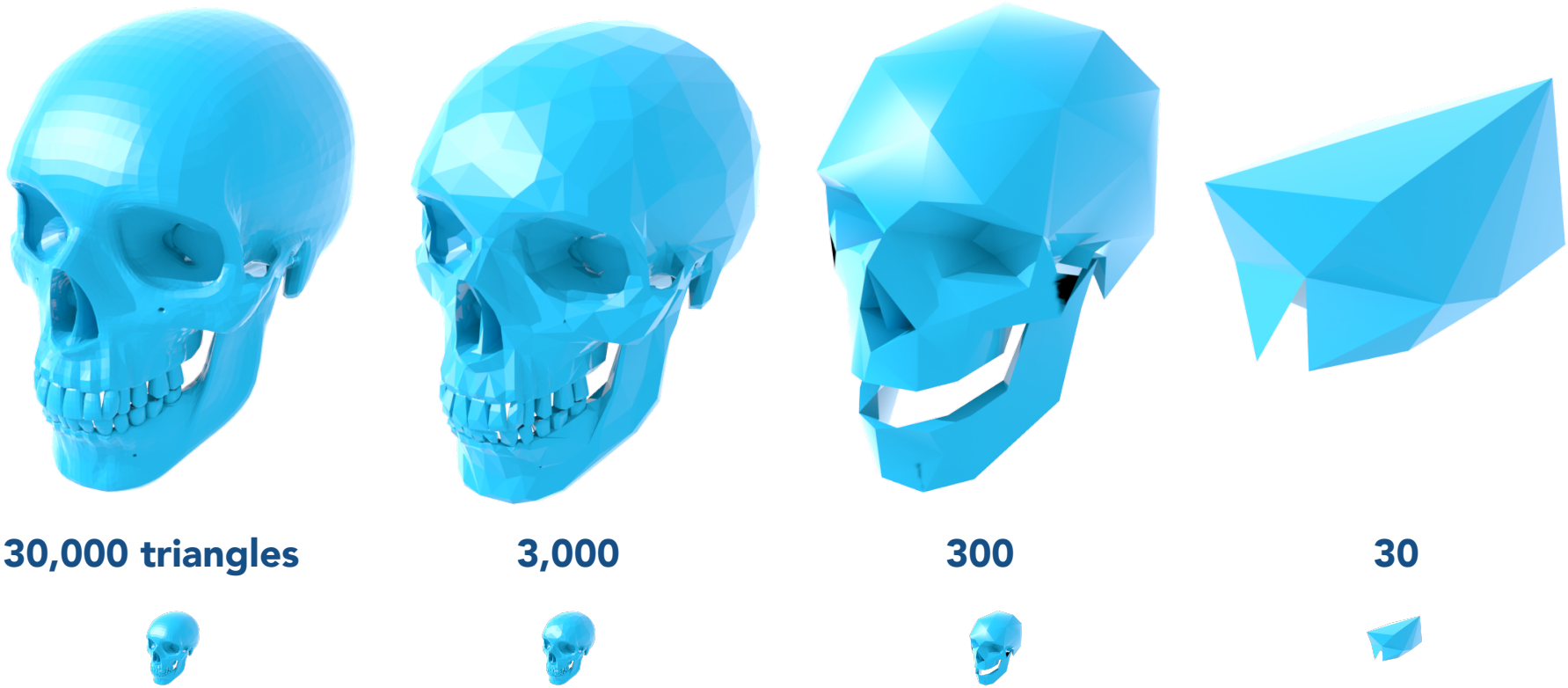
Subdivision in Action (Pixar's "Geri's Game")

1997

Mesh Simplification

Mesh Simplification

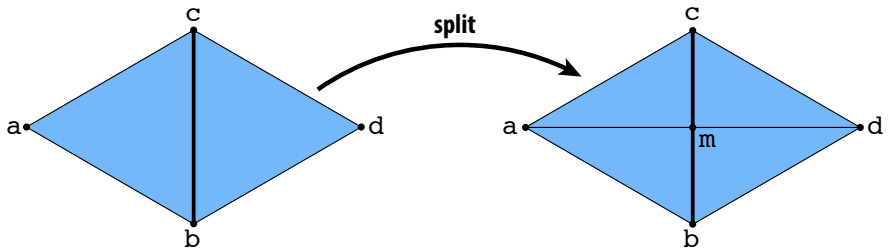
Goal: reduce number of mesh elements while maintaining overall shape



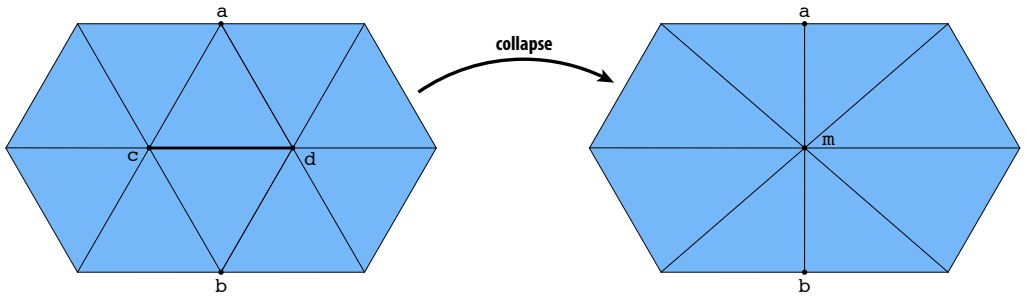
How to compute?

How Do We Resample Meshes? (Reminder)

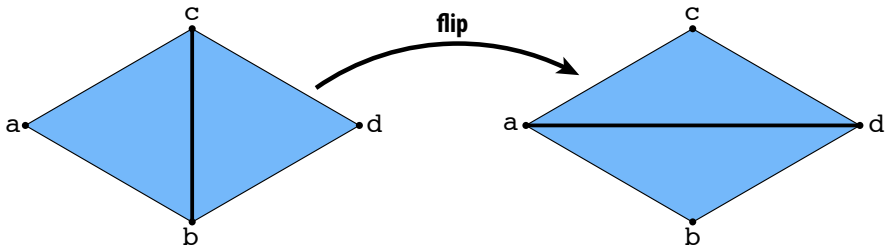
Edge split is (local) upsampling:



Edge collapse is (local) downsampling:



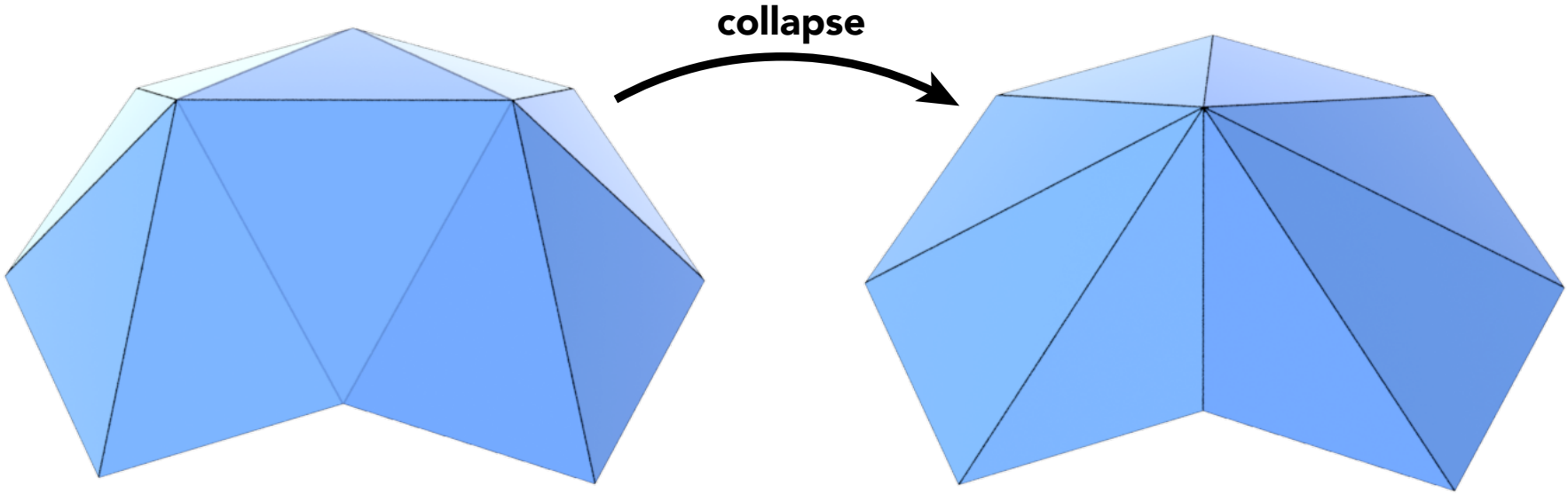
Edge flip is (local) resampling:



Still need to intelligently decide which edges to modify!

Estimate: Error Introduced by Collapsing An Edge?

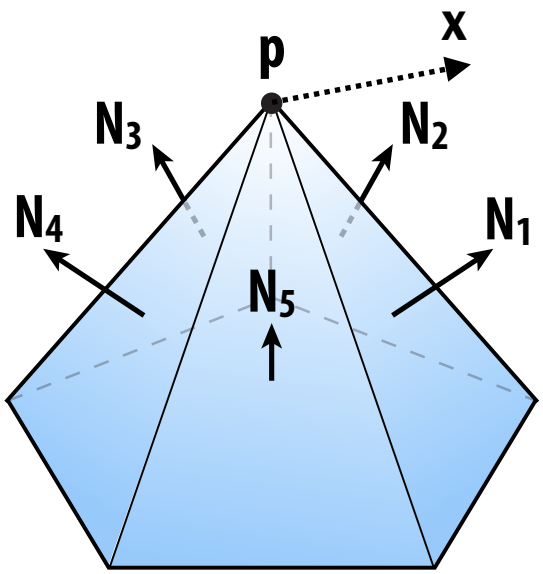
- Suppose we simplify a mesh using edge collapsing
- How much geometric error for collapsing an edge?
  - Where to place the new vertex that minimize the error?

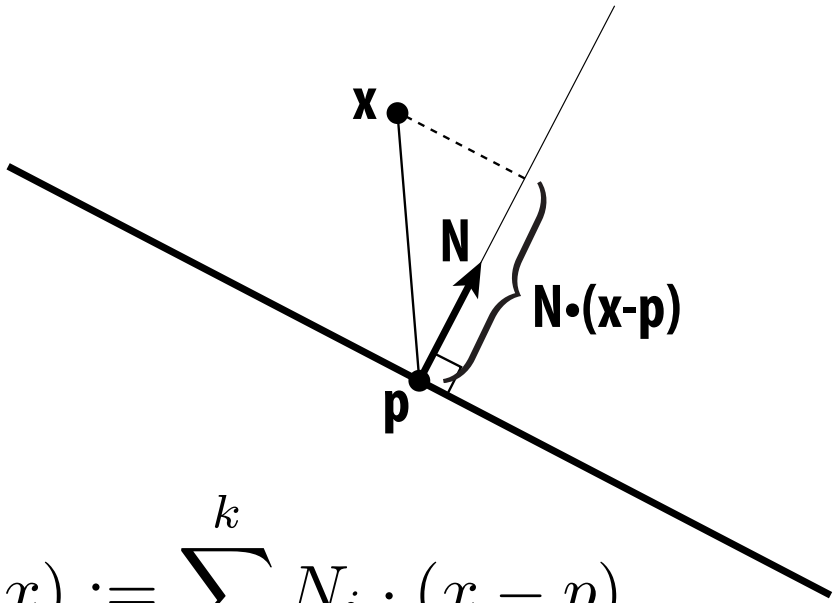


- Idea: Compute edge midpoint.
- Better idea: new vertex should minimize its sum of square distance to previously related triangle planes -> Quadric error

Quadric Error Metrics

- Quadric error: new vertex should minimize its sum of square distance to previously related triangle planes.





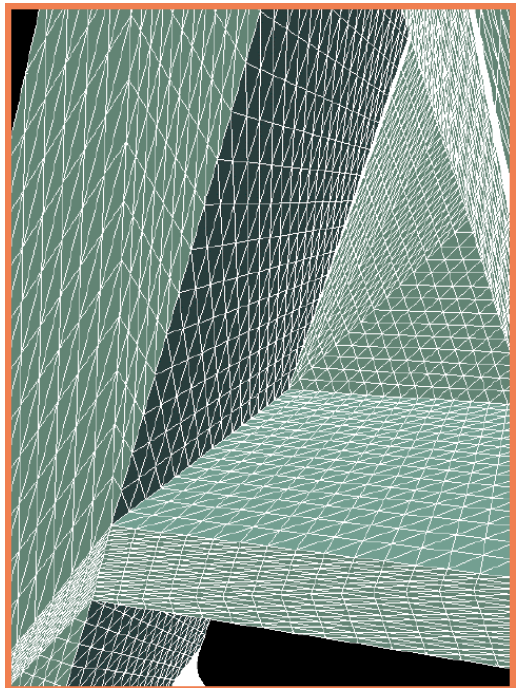
$$d(x) := \sum_{i=1}^{k} N_i \cdot (x - p)$$

误差最小化的顶点可以写成矩阵形式，可参考论文
https://www.cs.cmu.edu/~garland/Papers/quadrics.pdf
或http://graphics.stanford.edu/courses/cs468-10-fall/LectureSlides/08_Simplification.pdf

Simplification via Quadric Error

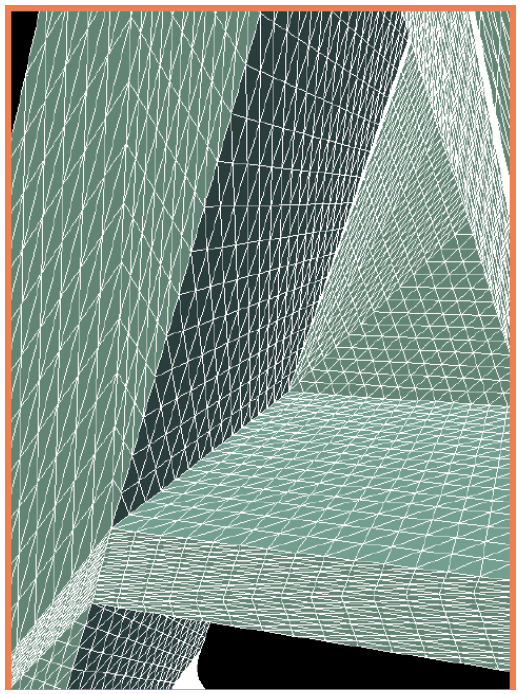
Iteratively collapse edges

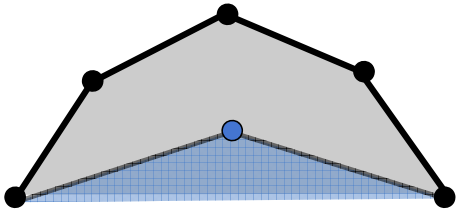
Which edges? Assign score with quadric error metric*



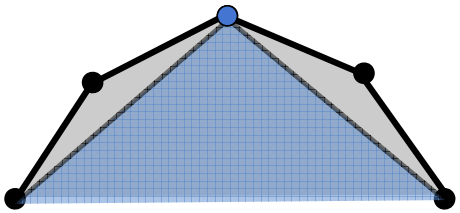
- Iteratively collapse edge with smallest score
- Greedy algorithm… great results!

Comparison of Different Errors

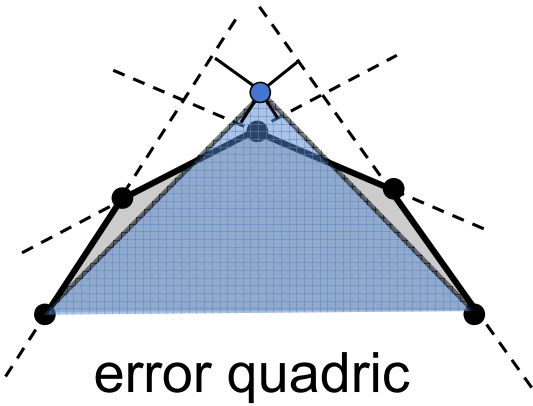




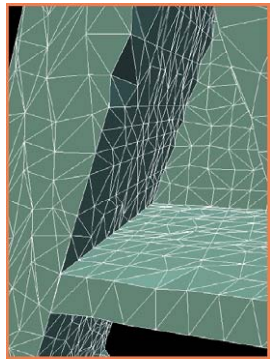
average

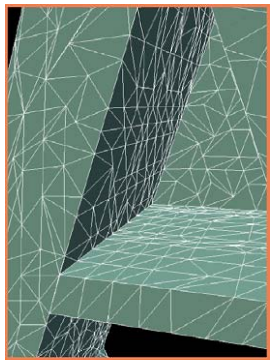


median

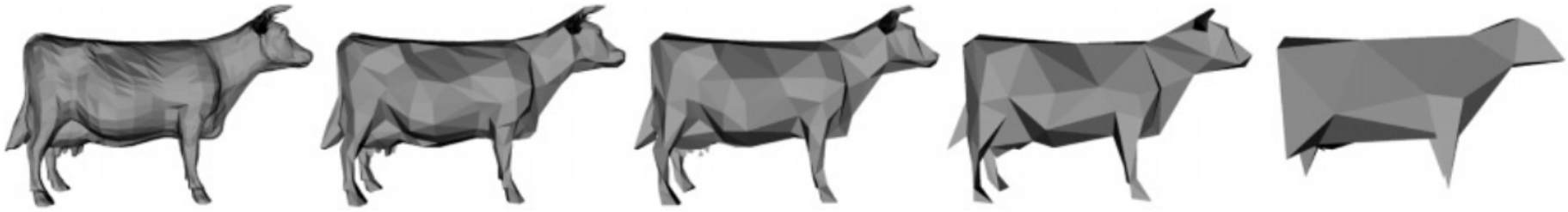


error quadric

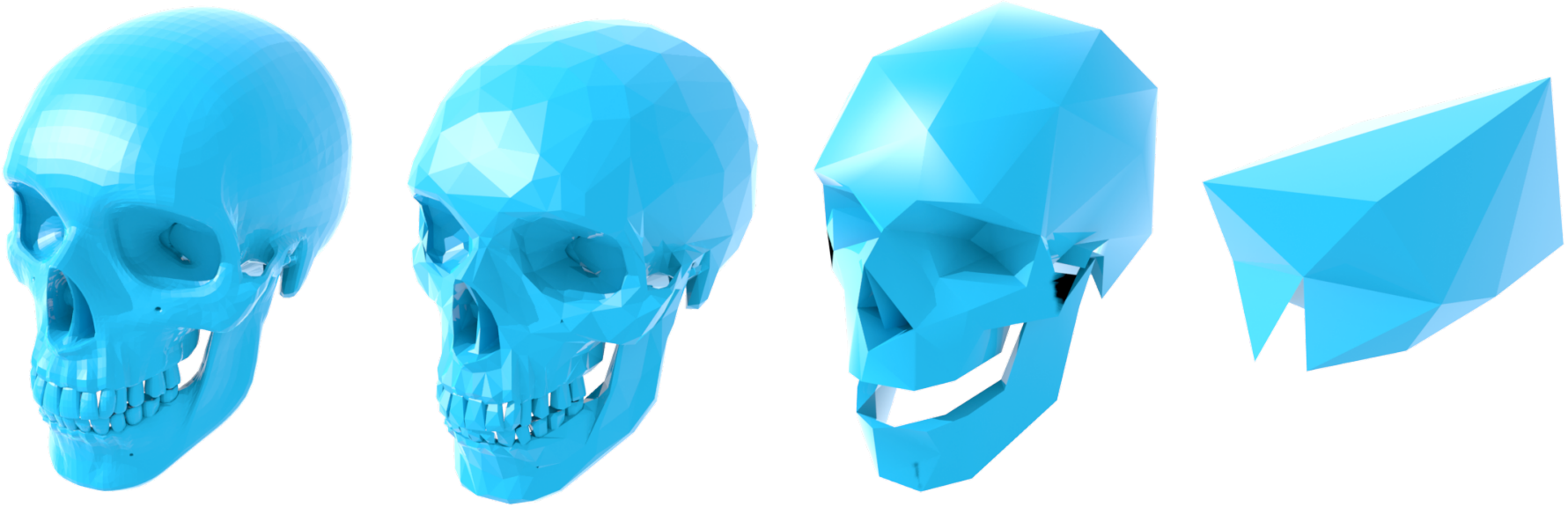




Quadric Error Mesh Simplification



5,804 994 532 248 64



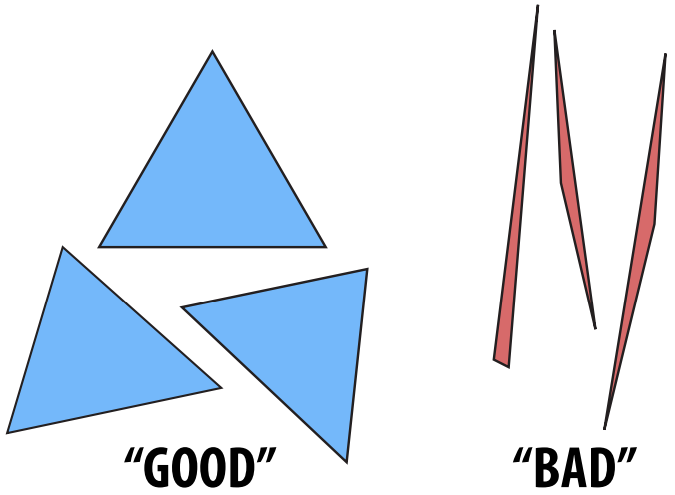
30,000 triangles 3,000 300 30

Garland and Heckbert '97

Mesh Regularization

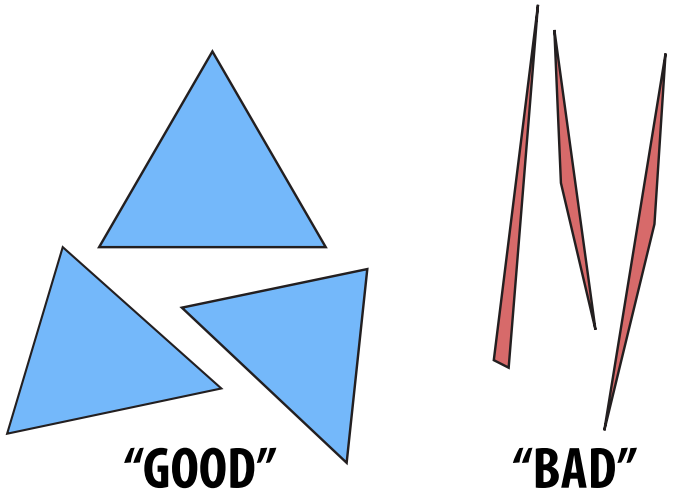
What Makes a “Good” Triangle Mesh?

One rule of thumb: triangle shape



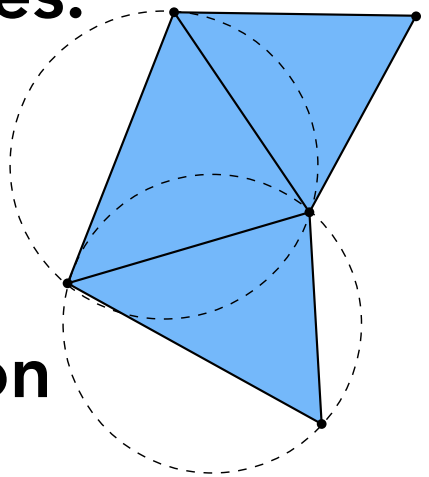
What Makes a "Good" Triangle Mesh?

One rule of thumb: triangle shape



More specific condition: Delaunay

- "Circumcircle interiors contain no vertices."



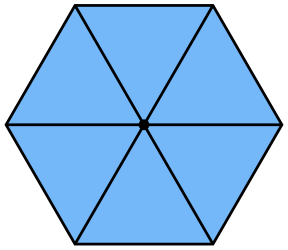
Not always a good condition, but often*

- Good for simulation
- Not always best for shape approximation

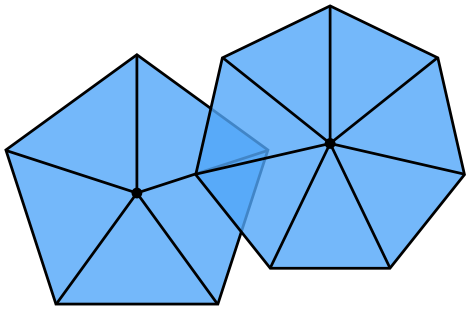
What Else Constitutes a Good Mesh?

Rule of thumb: regular vertex degree

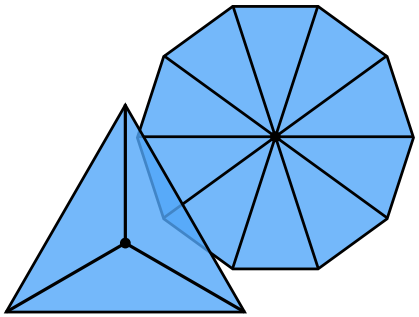
Triangle meshes: ideal is every vertex with valence 6:



"GOOD"



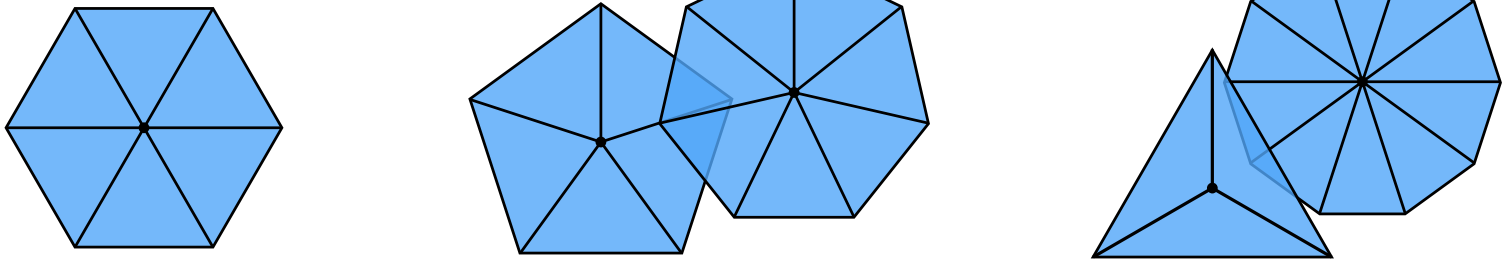
"OK"



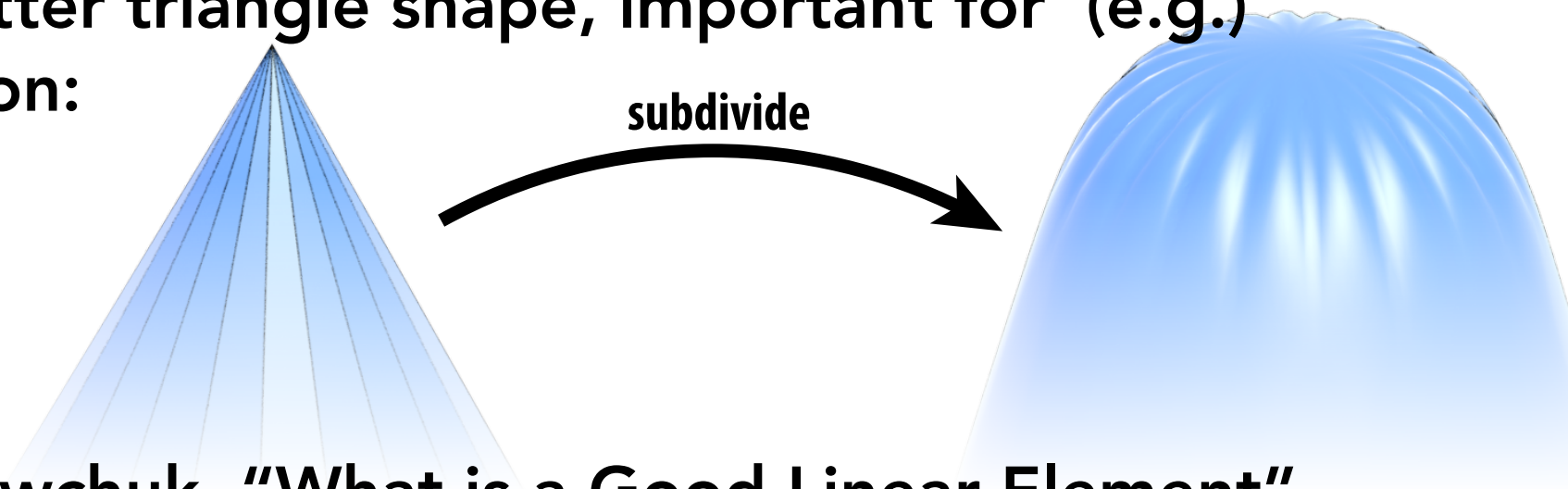
"BAD"

What Else Constitutes a Good Mesh?

Rule of thumb: regular vertex degree
Triangle meshes: ideal is every vertex with valence 6:



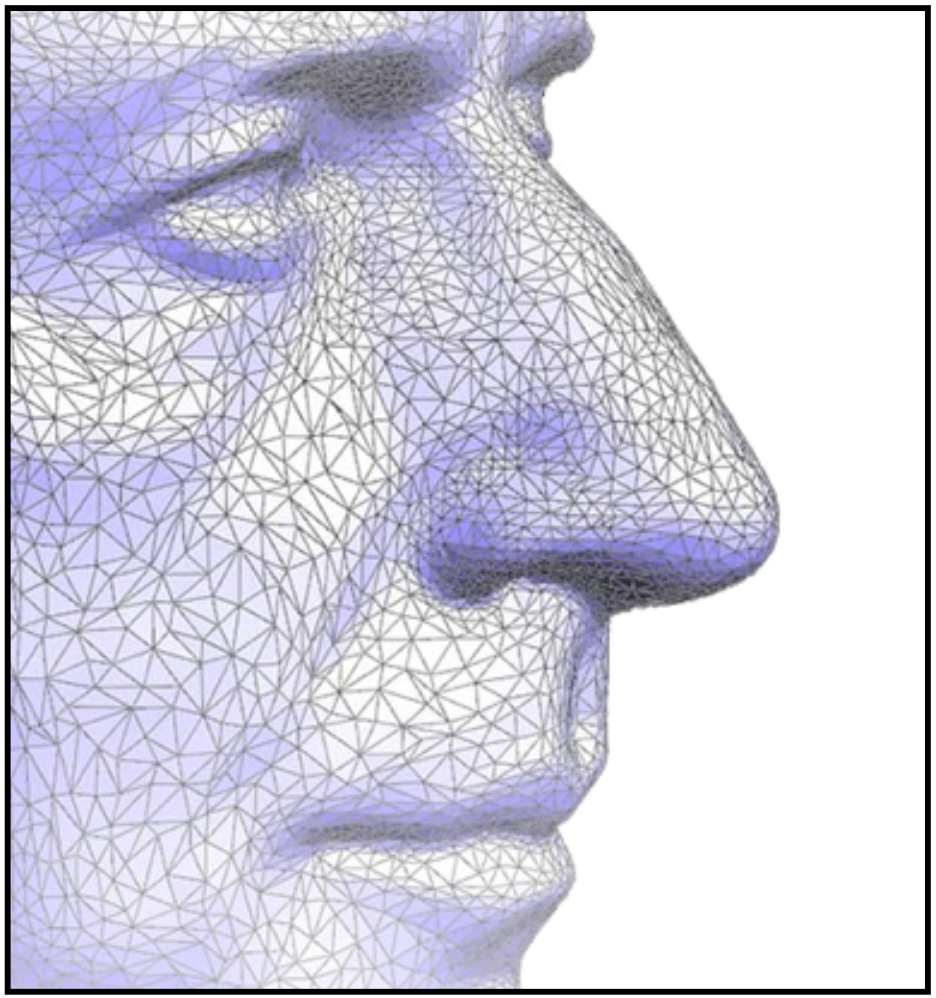
Why? Better triangle shape, important for (e.g.) subdivision:

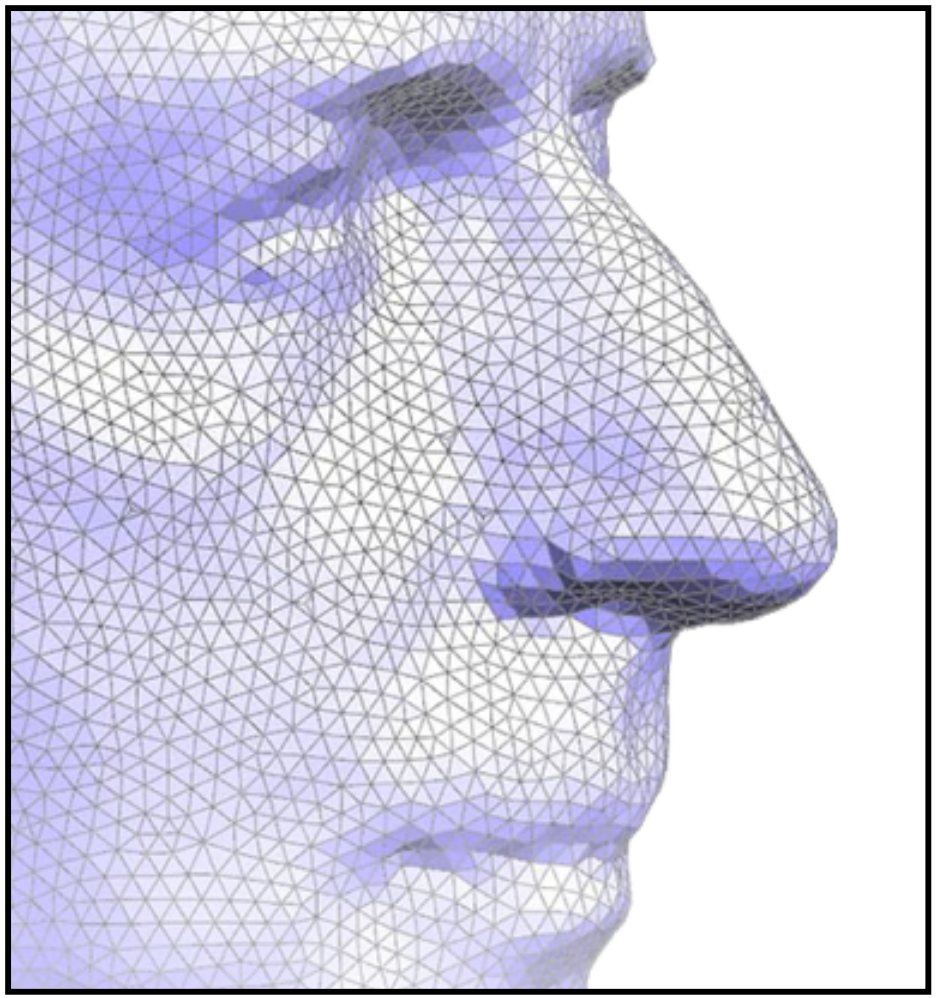


*See Shewchuk, "What is a Good Linear Element"

Isotropic Remeshing

Try to make triangles uniform in shape and size

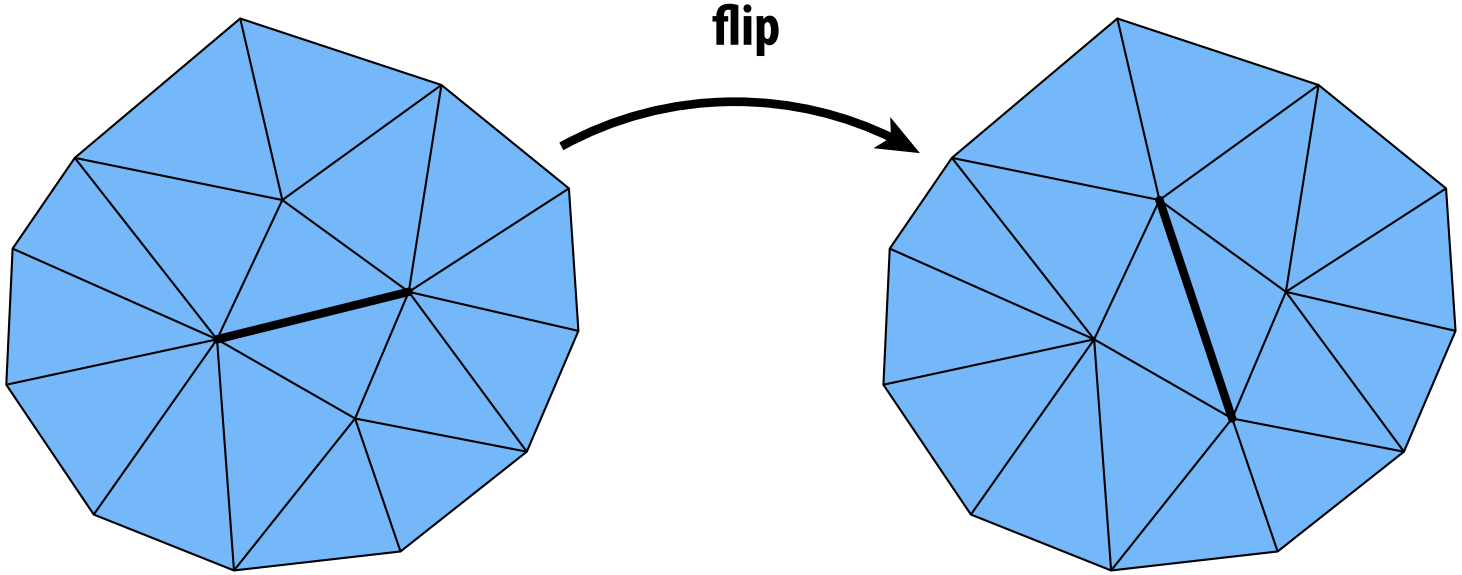




How Do We Improve Degree?

Edge flips!

If total deviation from degree 6 gets smaller, flip it!



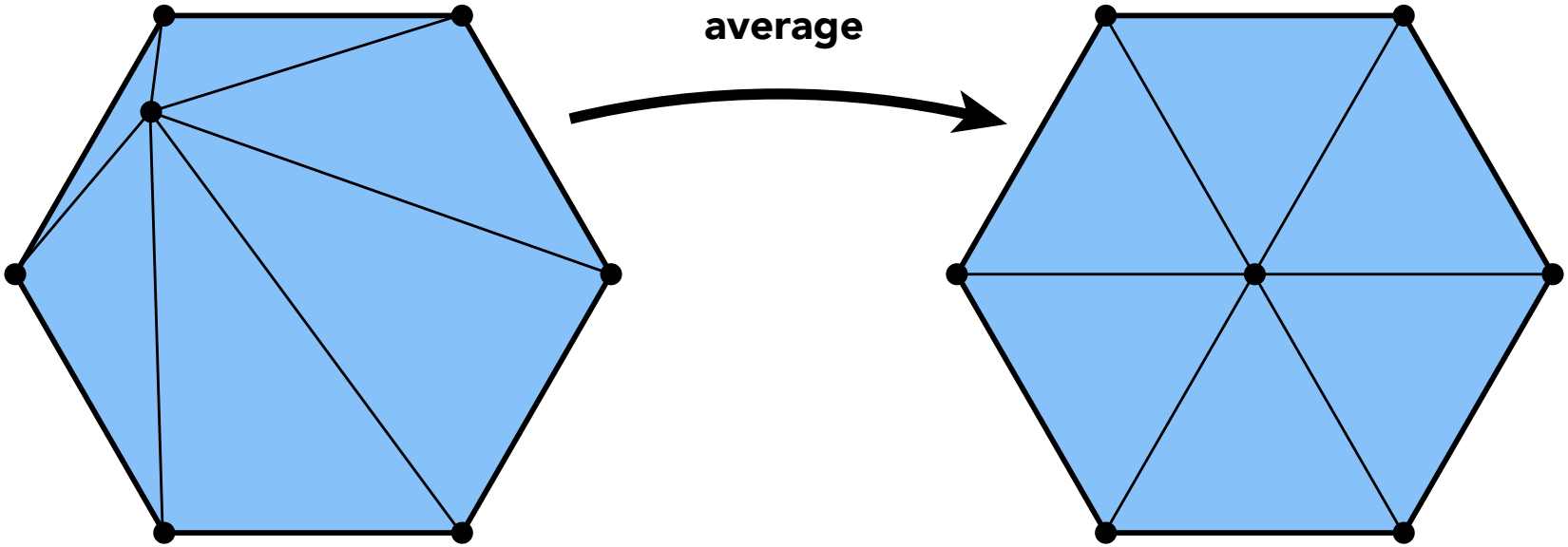
Iterative edge flipping acts like “discrete diffusion” of degree

No (known) guarantees; works well in practice

How Do We Make Triangles “More Round”?

Delaunay doesn’t mean equilateral triangles

Can often improve shape by centering vertices:

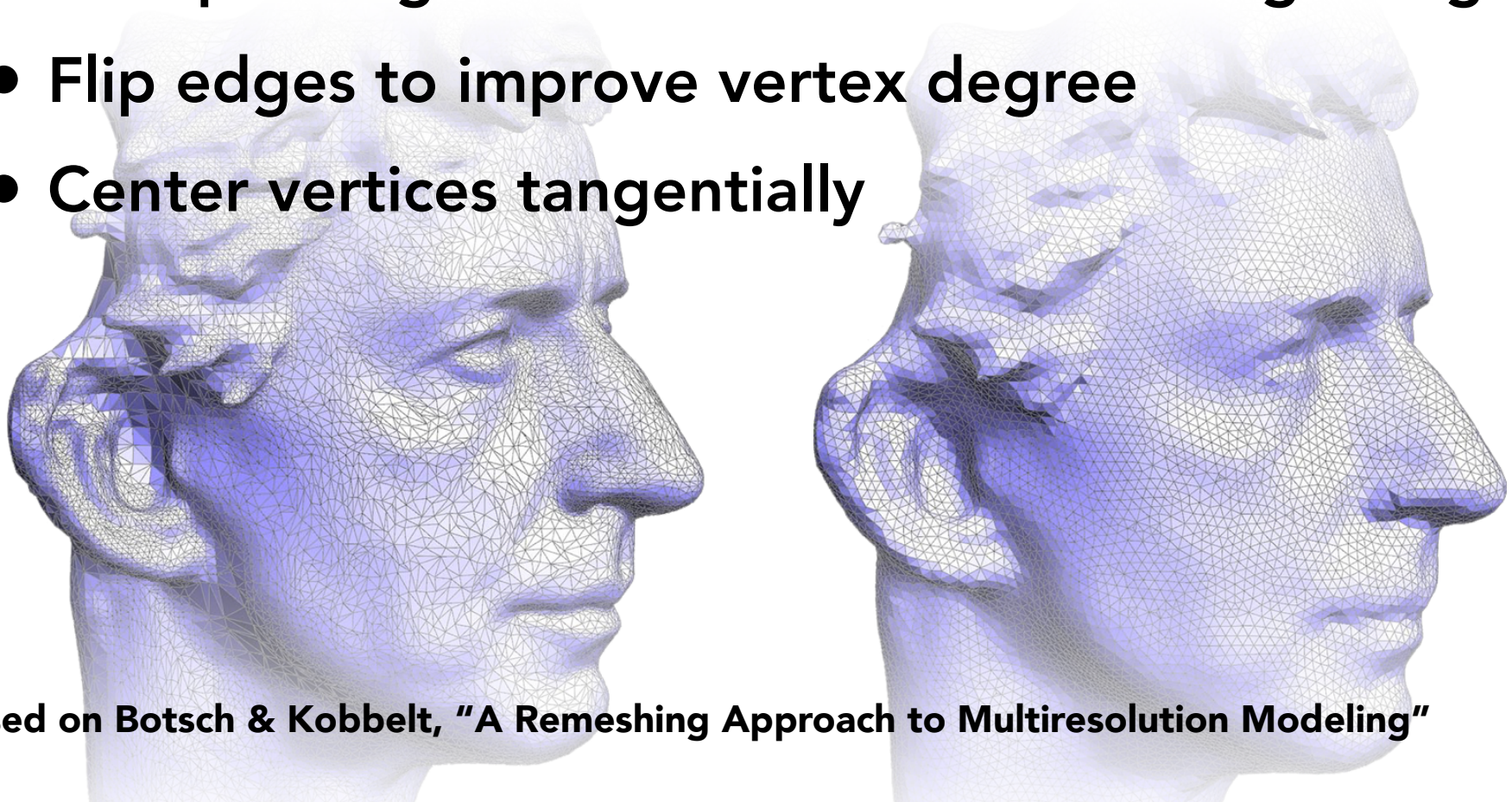


Consider vertex motion in-surface and off-surface

Isotropic Remeshing Algorithm*

Repeat four steps:

- Split edges over 4/3rds mean edge legth
- Collapse edges 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”