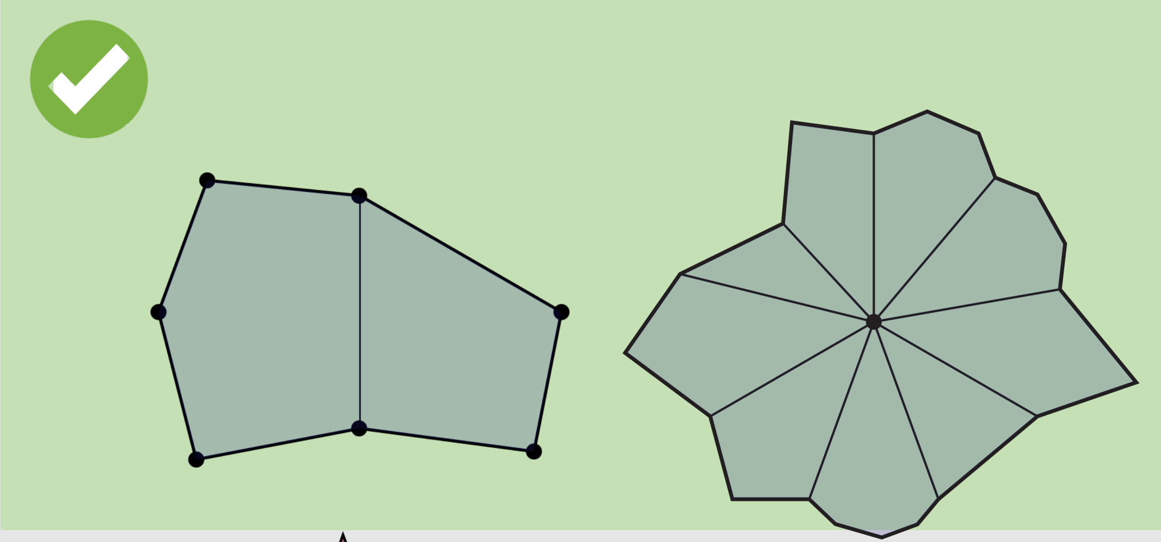
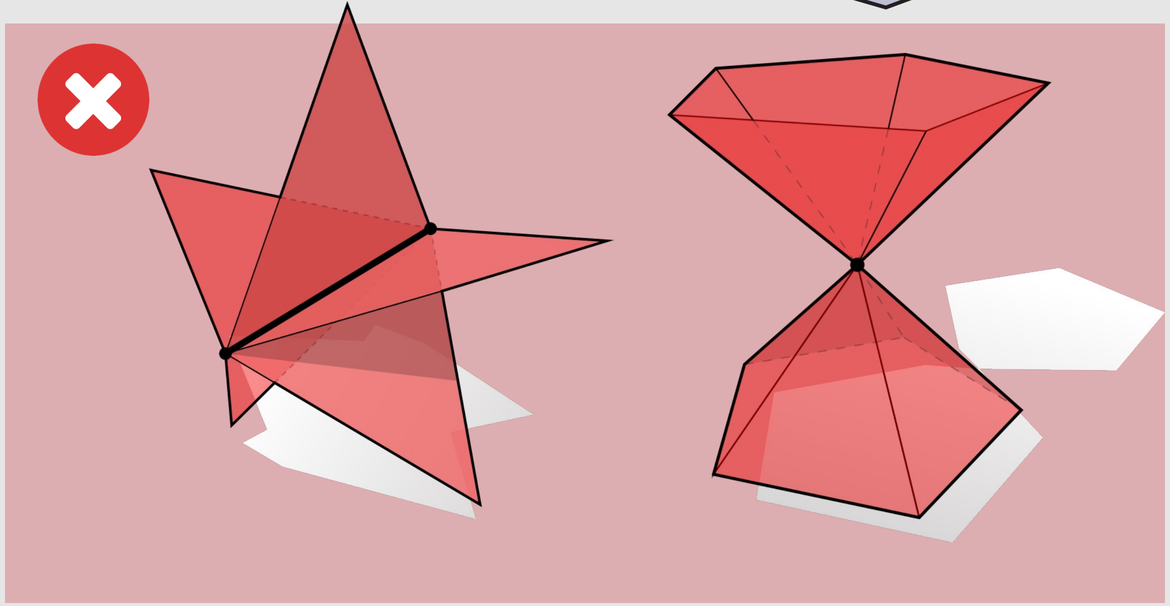
Manifolds, Mesh Representations, and Digital Geometric Processing

- Manifolds
- Mesh representations and local operations
- Digital Geometric Processing

Manifolds

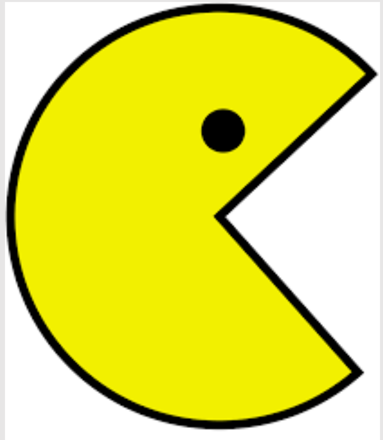
- Every edge is contained in only two polygons ("**no fins**")
  - The extra 3rd or 4th or 5th or so forth polygon is the fin of a fish
- The polygons containing each vertex make a "**single fan**"
  - We should be able to loop around the faces around a vertex in a clear way

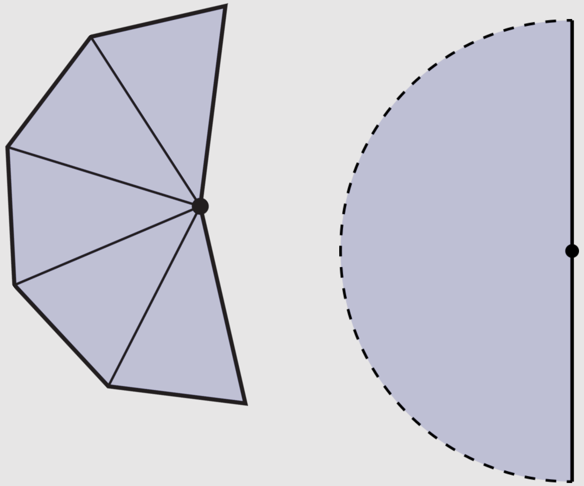




Boundary Edges are OK

- A boundary edge has 1 polygon per edge
- For each vertex, we still want a single fan (Pac-Man shape is fine)



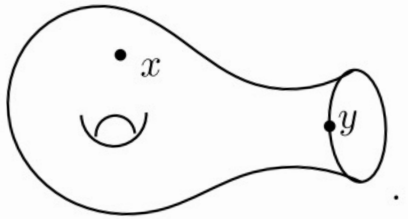


Definition 1. A manifold with boundary is a (Hausdorff, second countable) topological space X such that $\forall x \in X$ there exists an open $U \ni x$ and a homeomorphism $\phi_U : U \to \mathbb{R}^n$ or a homeomorphism $\phi_U : U \to \mathbb{R}_{\geq 0} \times \mathbb{R}^{n-1}$.

Informally, a manifold is a space where every point has a neighborhood homeomorphic to Euclidean space. Think about the surface of the earth. Locally when we look around it looks like $\mathbb{R}^2$, but globally it is not. The surface of the earth is, of course, homeomorphic to the space S^2 of unit vectors in $\mathbb{R}^3$. If a point has a neighborhood homeomorphic to $\mathbb{R}^n$ then it turns out intuition is correct and n is constant on connected components of X. Typically n is constant on all of X and is called the dimension of X. A manifold of dimension n is called an "n-manifold."

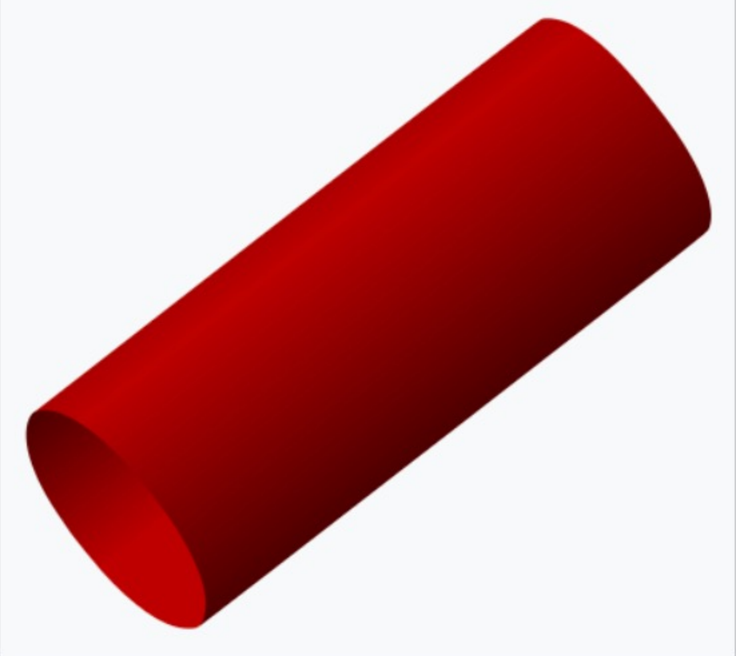
The boundary of the manifold X, denoted ∂X, is the set of points which only admit neighborhoods homeomorphic to $\mathbb{R}_{\geq 0} \times \mathbb{R}^{n-1}$.

For example, here's a (2-dimensional) manifold with boundary:



(1)

https://math.berkeley.edu/~donoghue/reu/quantum/Notes_June_26.pdf

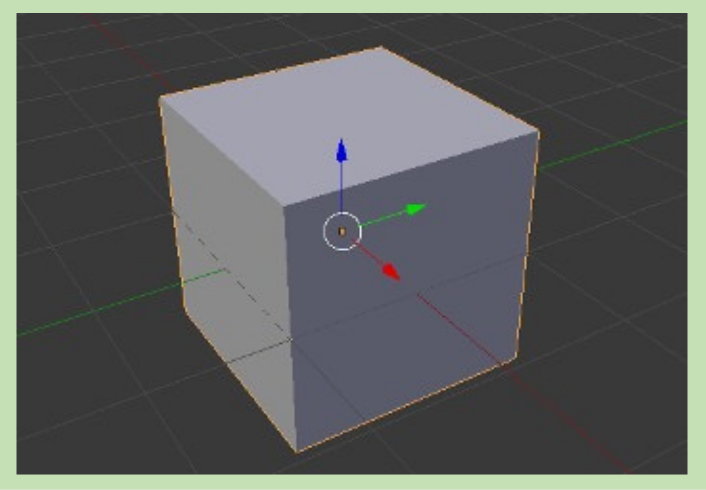


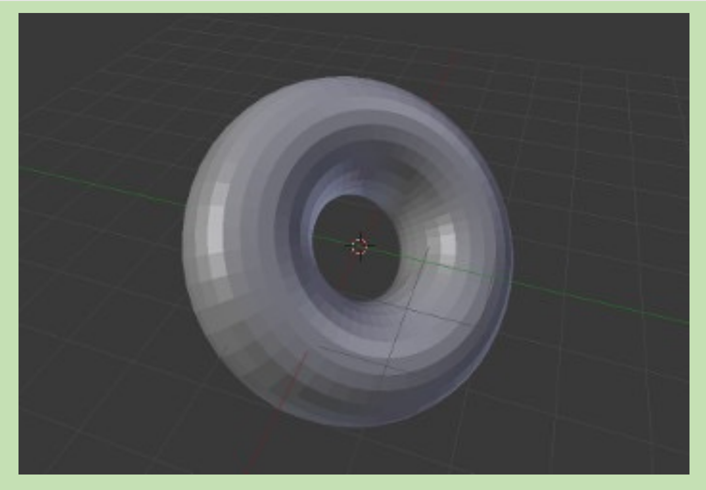
A finite cylinder is a manifold with boundary.

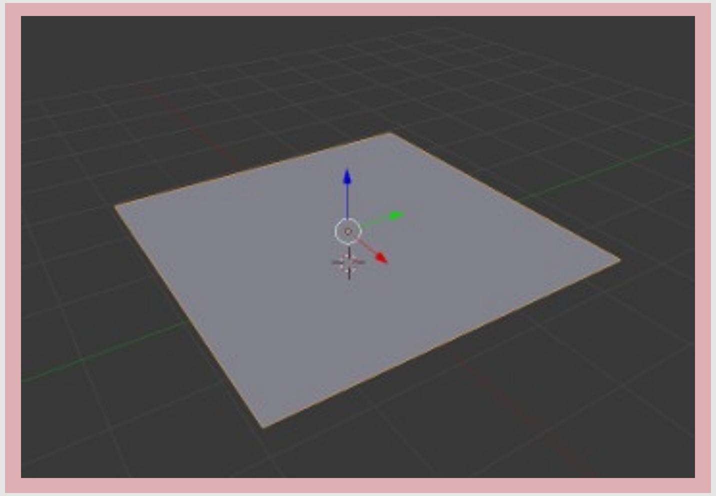
https://en.wikipedia.org/wiki/Manifold

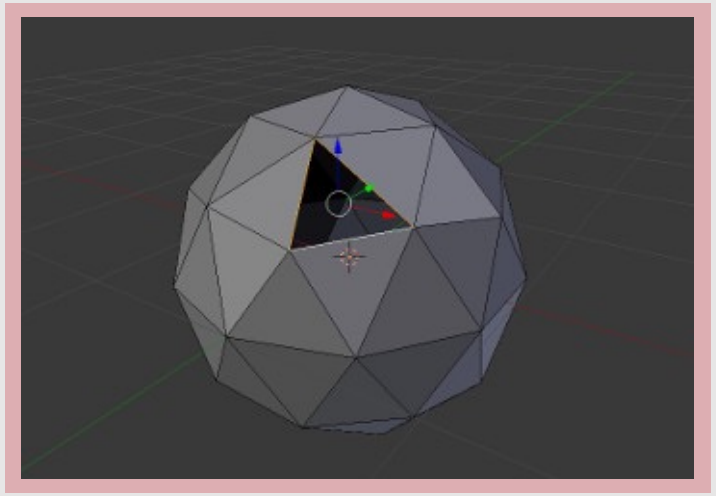
Boundaries are OK

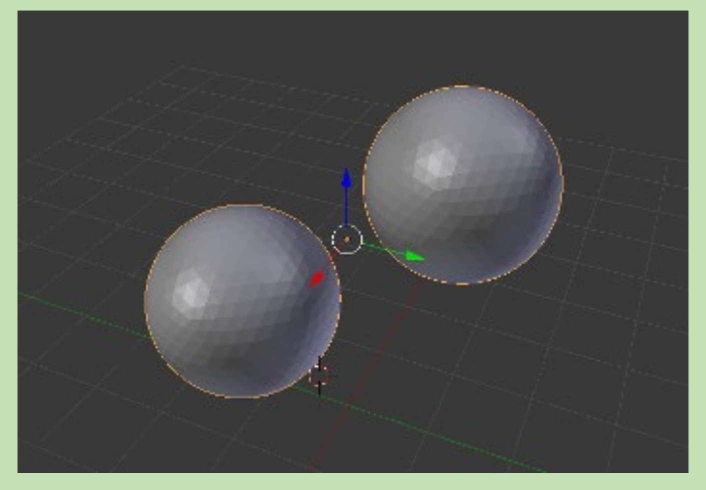
So .. What is going on with this slide?





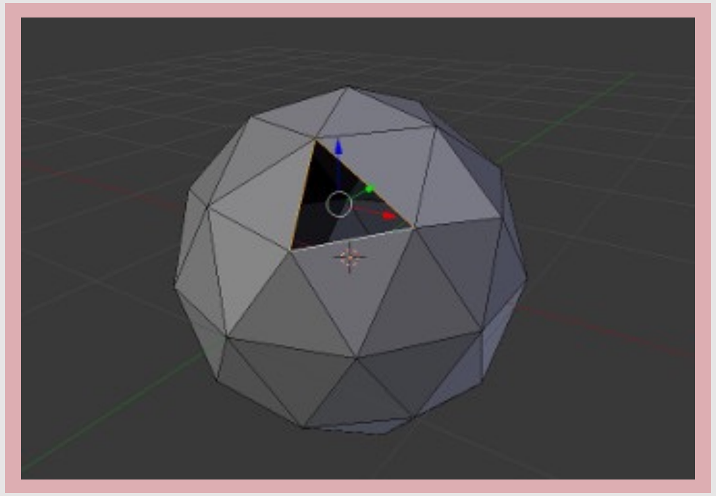


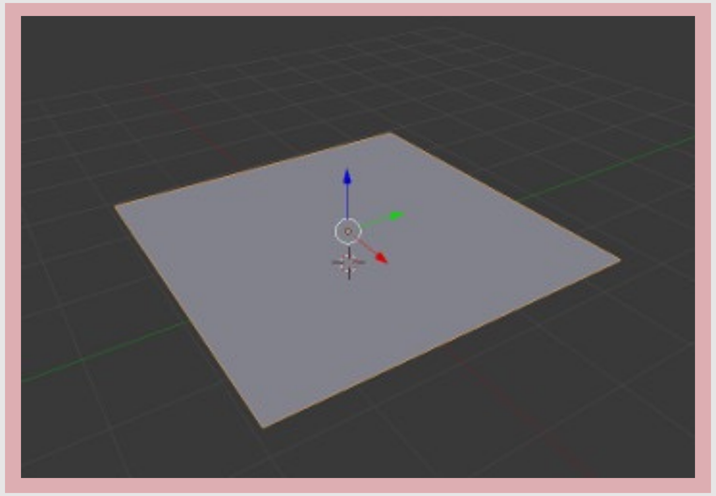




**https://github.com/rlguy/Blender-FLIP-Fluids/wiki/Manifold-Meshes

These are ***manifold*** meshes ***with a boundary***





- Some software (e.g., some simulators) requires meshes to be ***manifold*** ***without a boundary***. In this case, the mesh must appear to fully contain a volume, which might, for example, represent the volume of stuff you are simulating.

- For our project, boundaries are fine! However, it is good to know whether you have them.

**https://github.com/rlguy/Blender-FLIP-Fluids/wiki/Manifold-Meshes

- ~~Manifolds~~
- Mesh representations and local operations
- Digital Geometric Processing

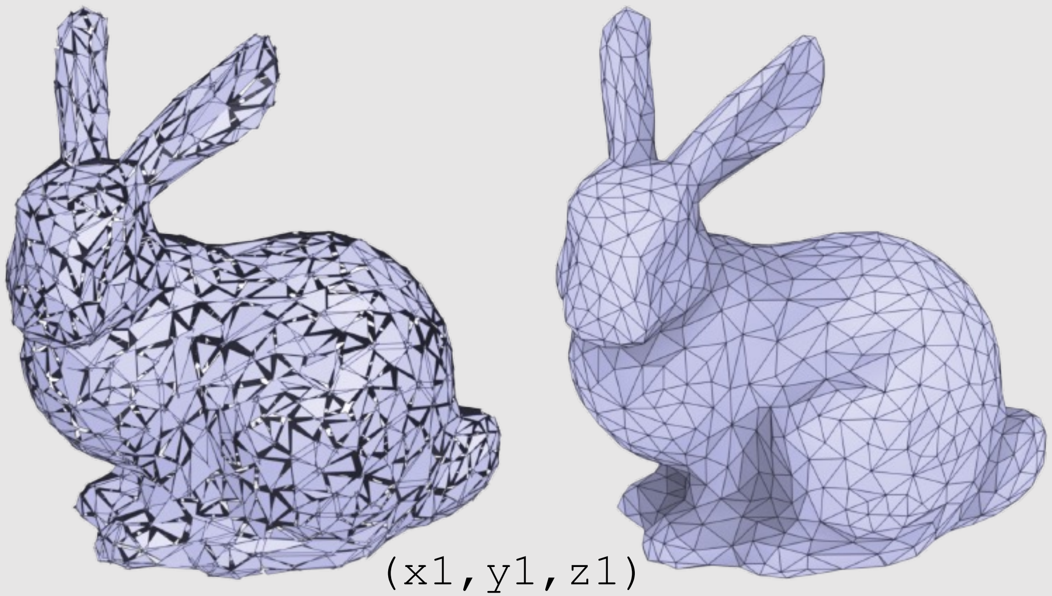
What are some ways to describe the connectivity of geometry?

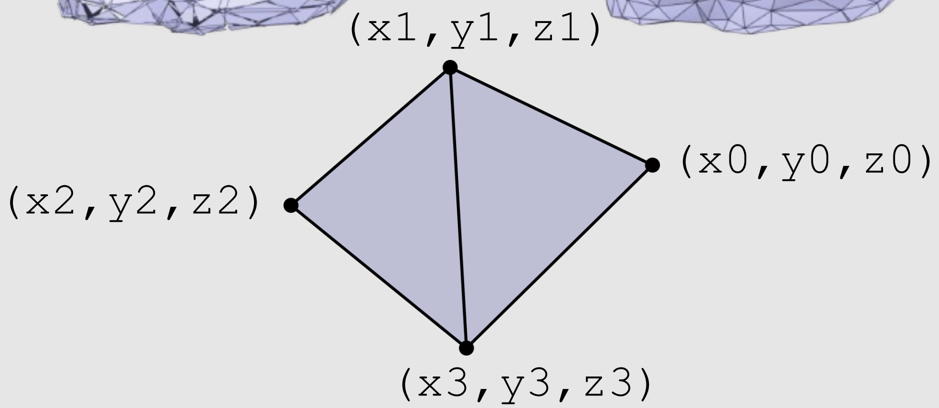
Polygon Soup

- Most basic idea imaginable:
  - For each triangle, just store three coordinates
  - No other information about connectivity
  - Not much different from point cloud
    - A “Triangle cloud”?

- **Pros:**
  - [+] Really stupid simple

- **Cons:**
  - [-] Really stupid
  - [-] Redundant storage of vertices
  - [-] Very difficult to find neighboring polygons





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

Adjacency List

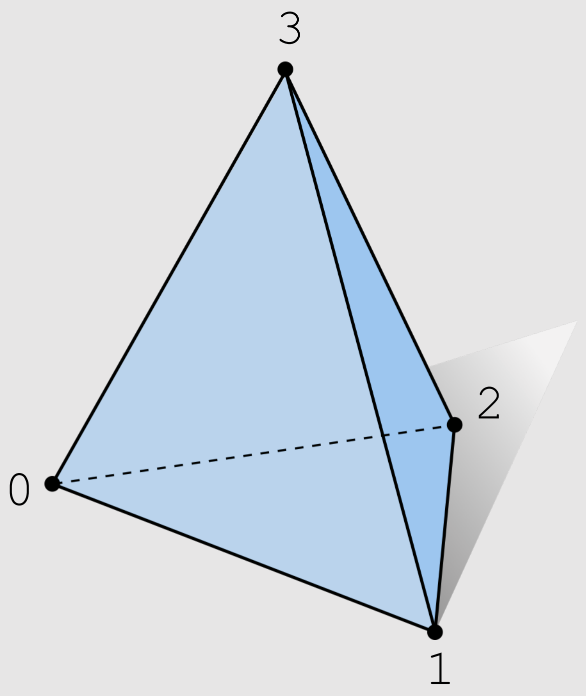
- A little more complicated:
  - Store triples of coordinates (x,y,z)
  - Store tuples of indices referencing the coordinates needed to build each triangle
- **Pros:**
  - [+] No duplicate coordinates
  - [+] Lower memory footprint
  - [+] Easy to keep geometry manifold
  - [+] Supports nonmanifold geometry
  - [+] Easy to change connectivity of geometry
- **Cons:**
  - [-] Very difficult to find neighboring polygons
  - [-] Difficult to add/remove mesh elements

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



Incidence Matrices

- If we want to know our neighbors, let's store them:
  - Store triples of coordinates (x,y,z) Store incidence matrix between vertices + edges, and edges + faces
    - 1 means touch, 0 means no touch
    - Store as sparse matrix

- **Pros:**
  - [+] No duplicate coordinates
  - [+] Finding neighbors is O(1)
  - [+] Easy to keep geometry manifold
  - [+] Supports nonmanifold geometry

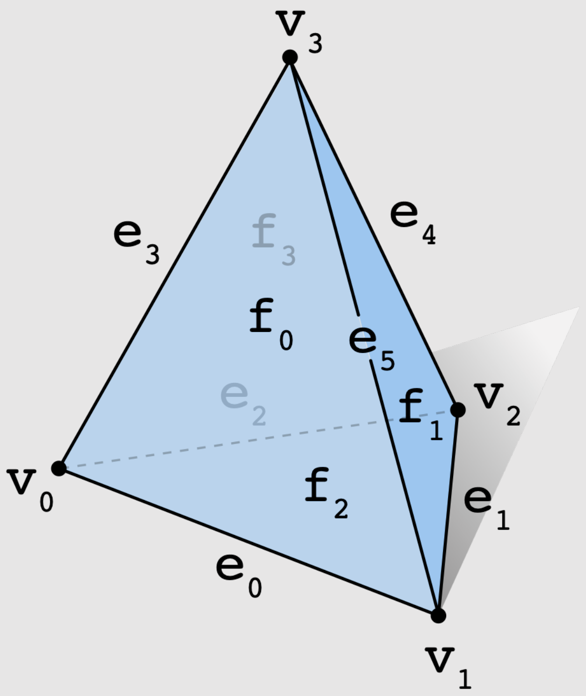
- **Cons:**
  - [-] Larger memory footprint
  - [-] Hard to change connectivity with fixed indices
  - [-] Difficult to add/remove mesh elements

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



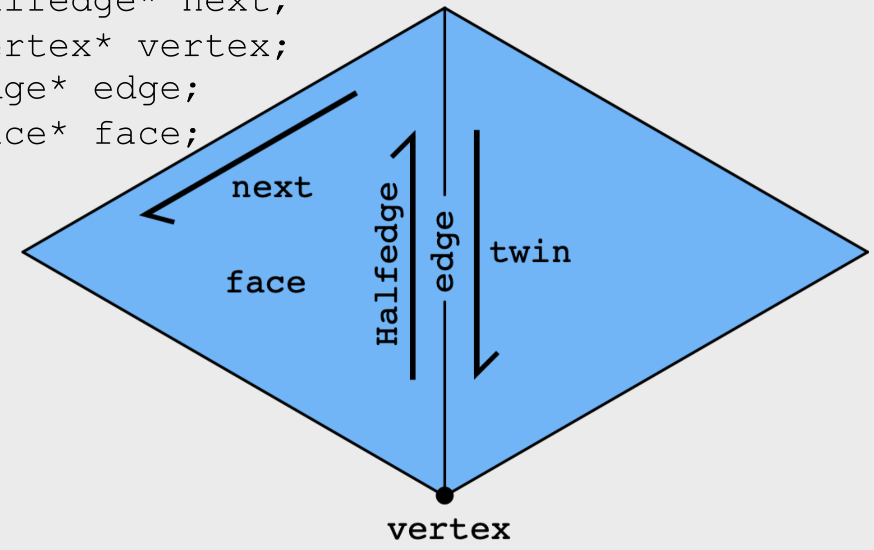
Halfedge Data Structure

- Let's store a little, but not a lot, about our neighbors:
  - Halfedge data structure added to our geometry
  - Each edge gets 2 halfedges
    - Each halfedge "glues" an edge to a face

- **Pros:**
  - [+] No duplicate coordinates
  - [+] Finding neighbors is O(1)
  - [+] Easy to traverse geometry
  - [+] Easy to change mesh connectivity
  - [+] Easy to add/remove mesh elements
  - [+] Easy to keep geometry manifold

- **Cons:**
  - [-] Does not support nonmanifold geometry

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



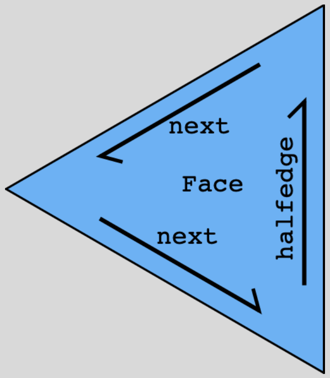
Halfedge Data Structure

- Makes mesh traversal easy
  - Use “twin” and “next” pointers to move around the mesh
  - Use “vertex”, “edge”, and “face” pointers to grab element

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

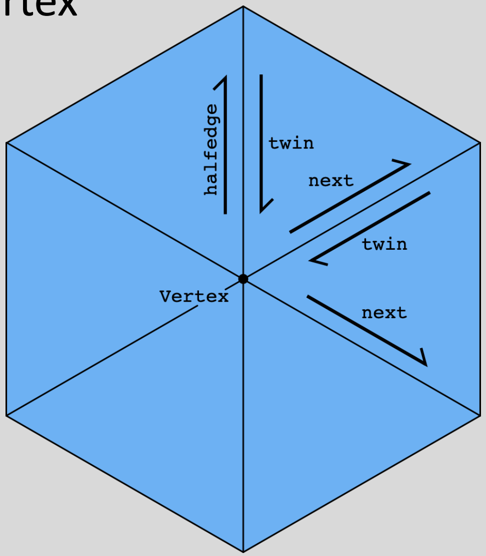
Example: visit all vertices in a face

```
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!

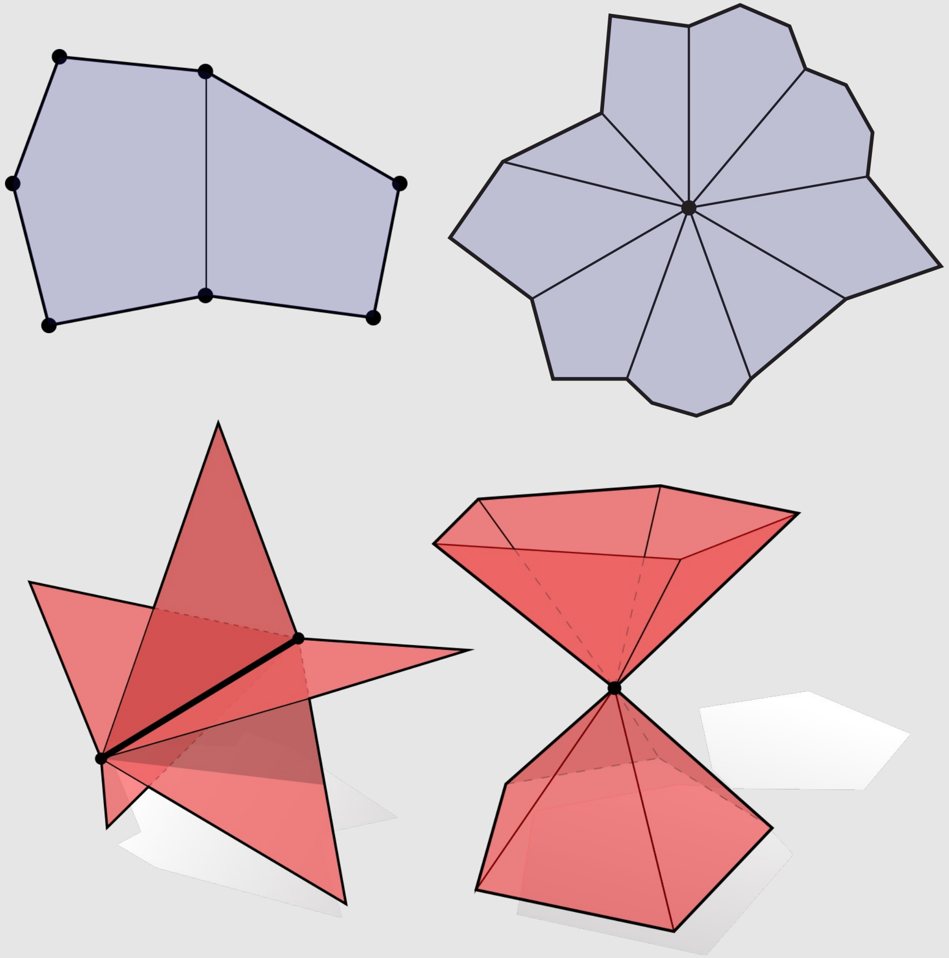
Halfedge Data Structure

- Halfedge meshes are always manifold!
- Halfedge data structures have the following constraints:

```
h->twin->twin == h // my twin's twin is me
h->twin != h // I am not my own twin
h2->next = h //every h is someone's "next"
```

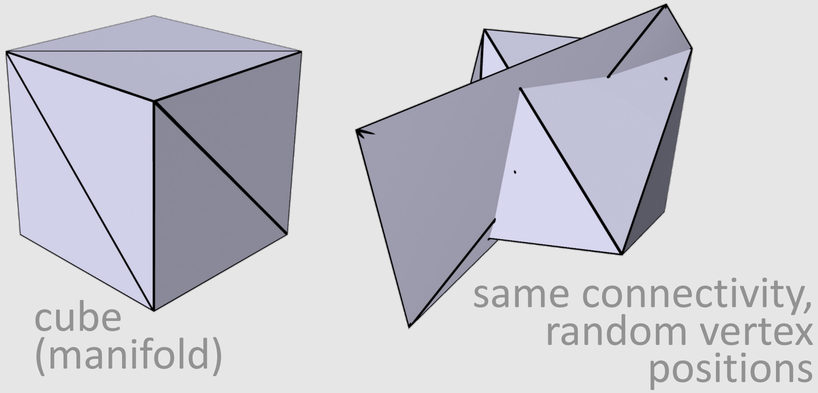
- Keep following **next** and you'll traverse a face
- Keep following **twin** and you'll traverse an edge
- Keep following **next->twin** and you'll traverse a vertex

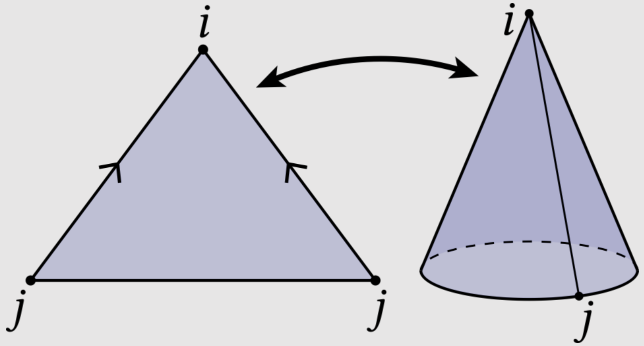
- **Q: Why, therefore, is it impossible to encode the red figures?**
  - First shape violates first 2 conditions
  - Second shape violates 3rd condition

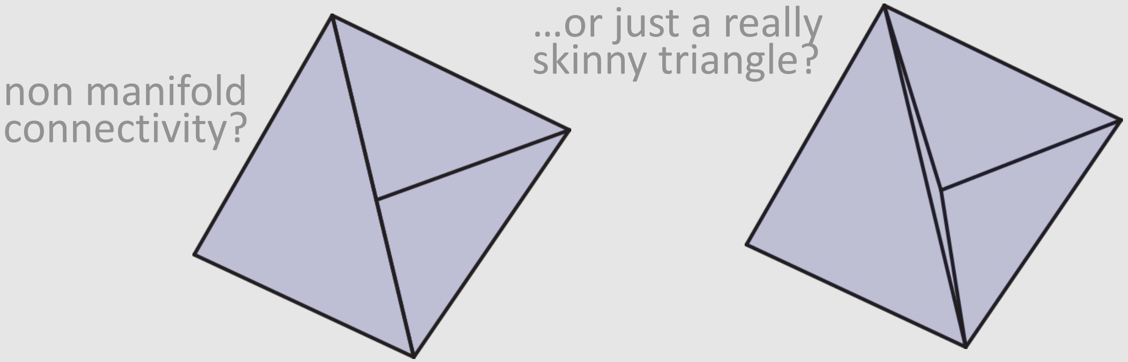


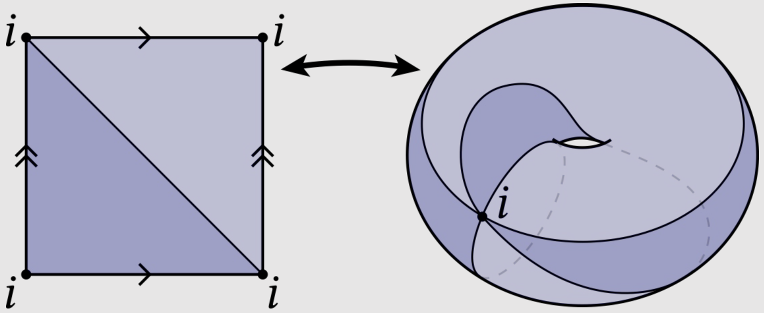
Connectivity vs Geometry

- Recall manifold conditions (fans not fins):
  - These conditions say nothing about vertex positions! Just connectivity
- Can have perfectly good (manifold) connectivity, even if geometry is awful
  - Can have perfectly good manifold connectivity for which any vertex positions give “bad” geometry!
- Leads to confusion when debugging:
  - Mesh looks “bad”, even though connectivity is fine









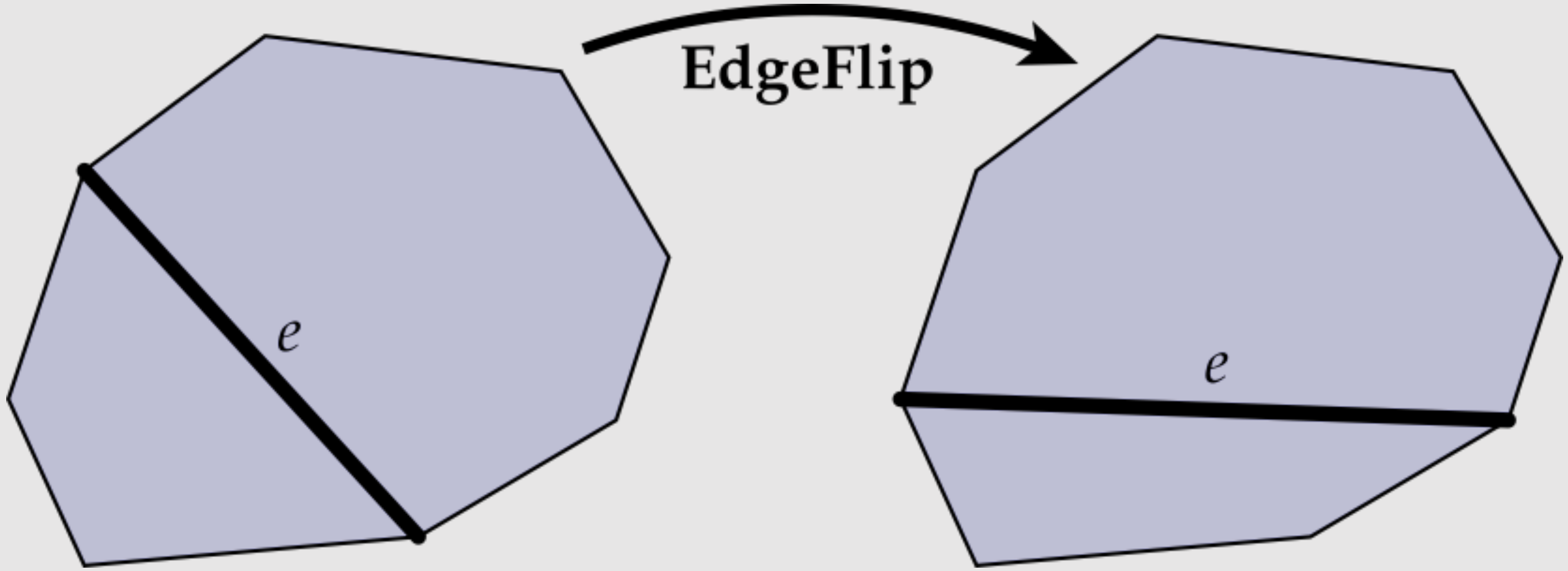
- ~~Manifolds~~

- ~~Mesh representations~~ and local operations

- Digital Geometric Processing

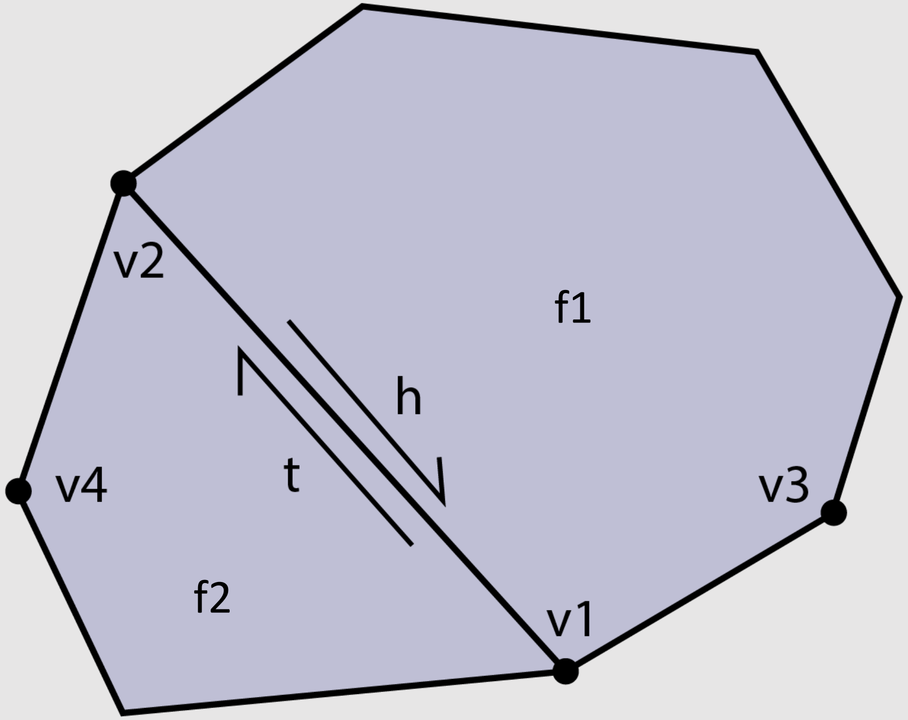
Edge Flip

Goal: Move edge e around faces adjacent to it:



- No elements created/destroyed, just pointer reassignment
- Flipping the same edge multiple times yields original results

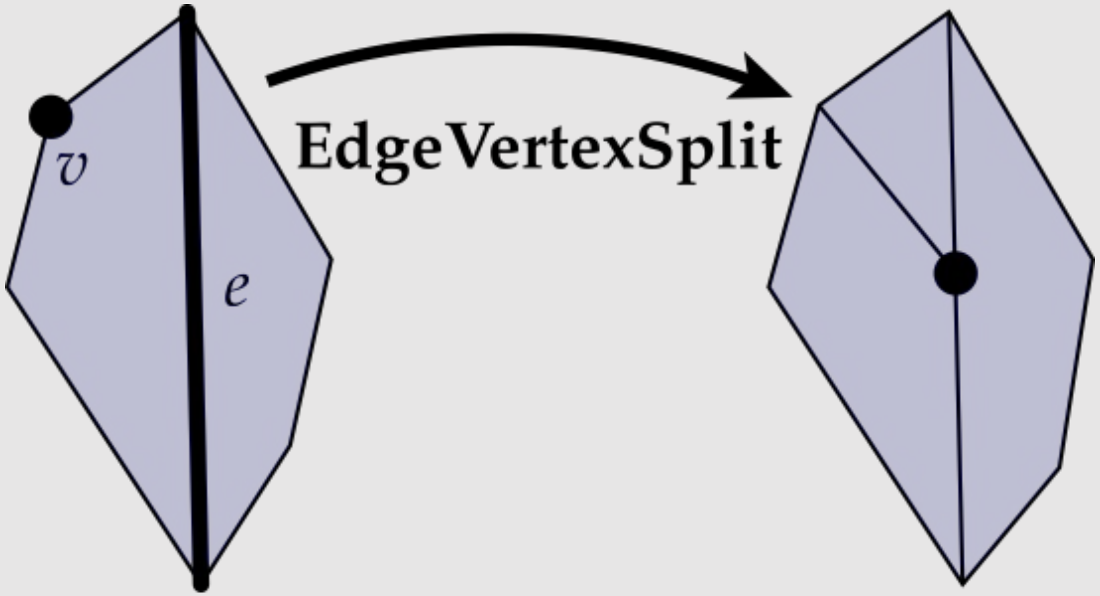
Edge Flip




```
// collect
h = e->halfedge;
t = h->twin;
v1 = h->next->vertex;
v2 = t->next->vertex;
v3 = h->next->next->vertex;
v4 = t->next->next->vertex;
f1 = h->face;
f2 = t->face;

// disconnect
v1->halfedge = h->next;
v2->halfedge = t->next;
f1->halfedge = h;
f2->halfedge = t;

// connect
t->vertex = v3;
h->vertex = v4;
f1->halfedge = h;
f2->halfedge = t;
```

Edge Vertex Split

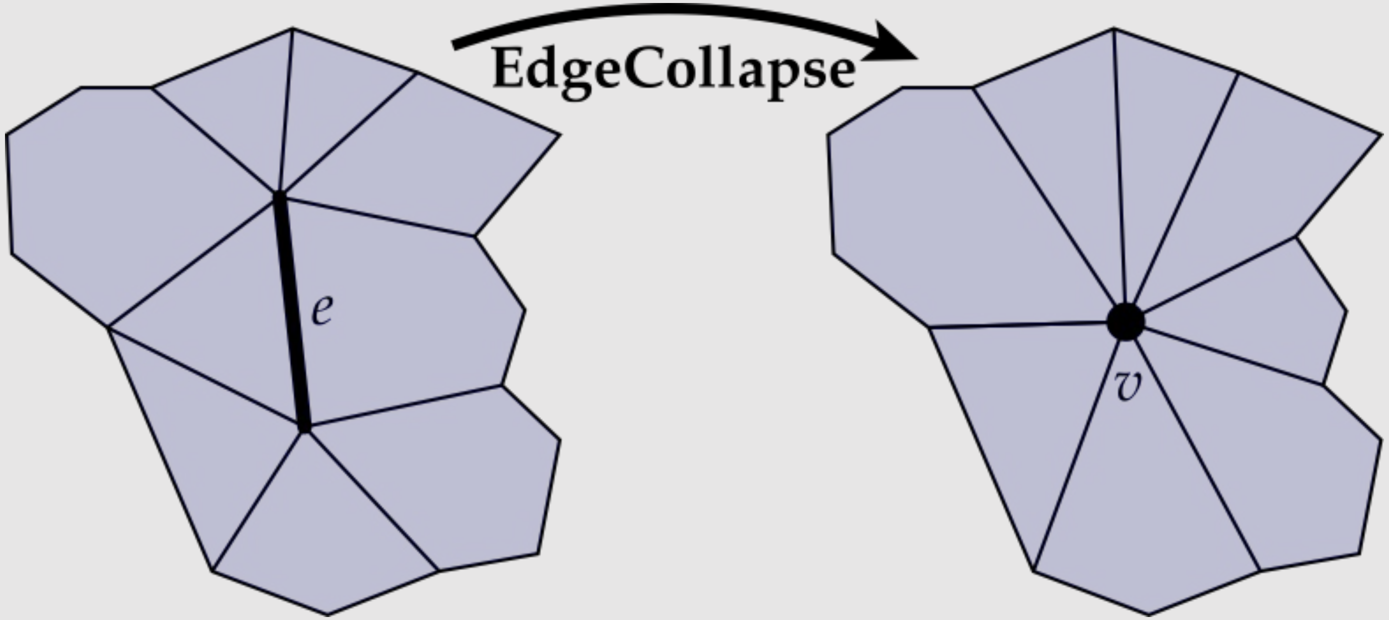
Goal: Insert edge between vertex v and midpoint of edge e:



- Creates a new vertex, new edge, and new face
- Involves much more pointer reassignments

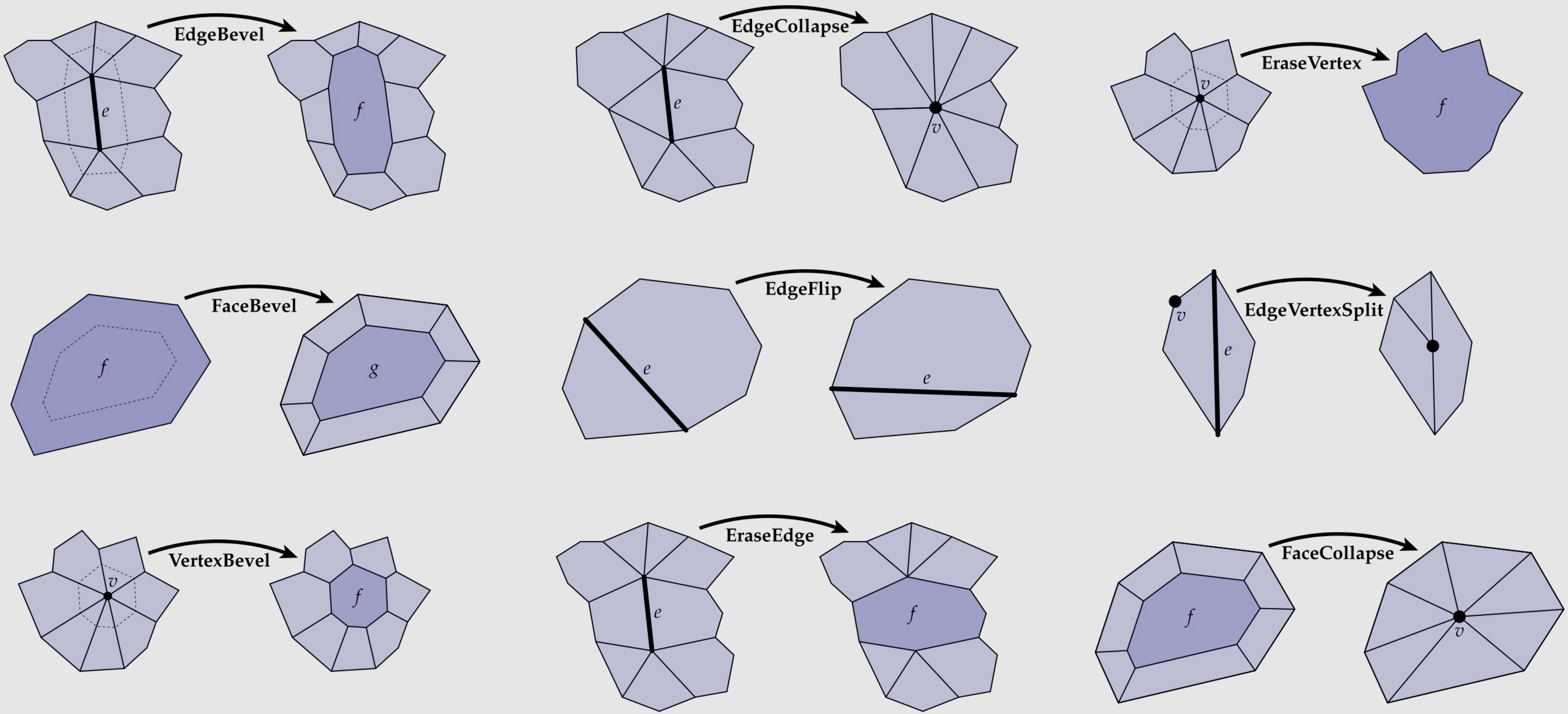
Edge Collapse

Goal: Replace edge (c,d) with a single vertex m:



- Deletes a vertex, (up to) 3 edges, and (up to) 2 faces
  - Depends on the degree of the original faces

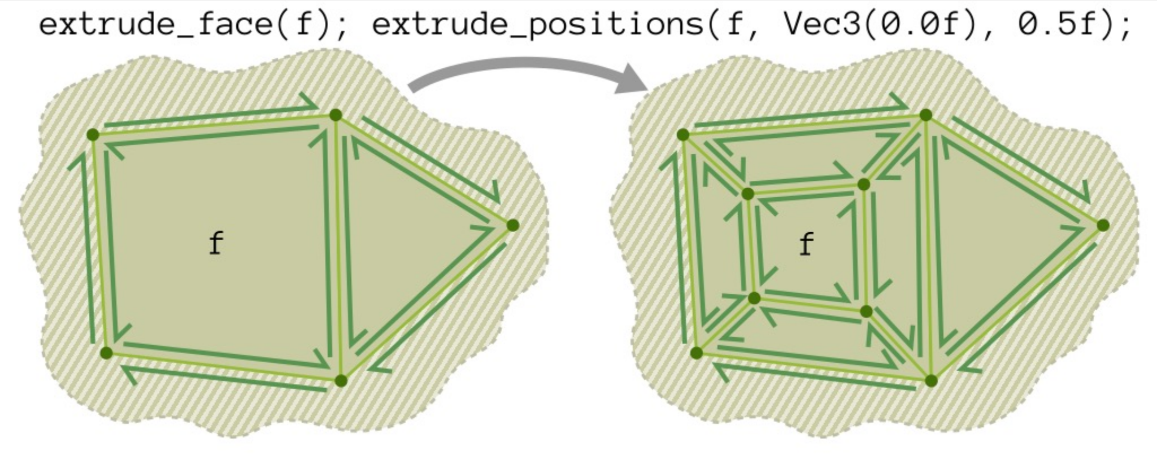
Local Operations



Many other local operations you will explore in your homework…

Local Operation Tips

- Always draw out a diagram
  - We've given you some unlabeled diagrams
  - With pen + paper, label the elements you'll need to collect/create



- Stage your code in the following way:
  - Create
  - Collect
  - Disconnect
  - Connect
  - Delete

- Write asserts around your code
  - Check if elements that should be deleted were deleted
  - Make sure there are no dangling references to anything that has been deleted
  - Make sure every element that you disconnected or reconnected is still valid
    - What it means for a vertex to be valid is not the same as what it means for an edge to be valid, etc.

- ~~Manifolds~~

- ~~Mesh representations and local operations~~

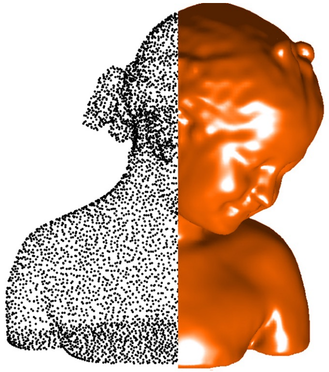
- Digital Geometric Processing

- ~~Manifolds~~

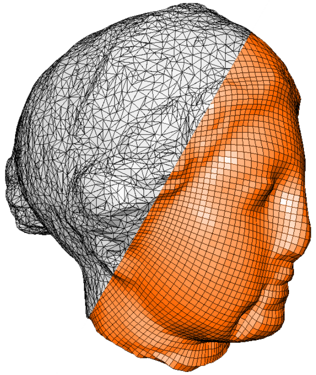
- ~~Mesh representations and local operations~~

- Digital Geometric Processing
  - Good Geometry
  - Geometric Subdivision
  - Geometric Simplification
  - Geometric Remeshing
  - Geometric Queries

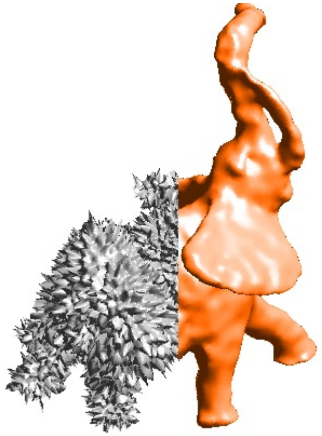
Geometry Processing Tasks



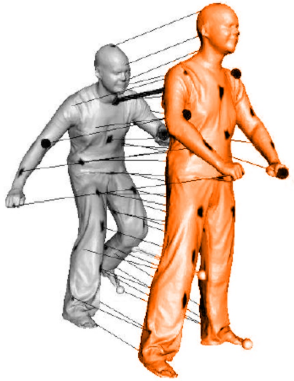
[reconstruction]



[remeshing]

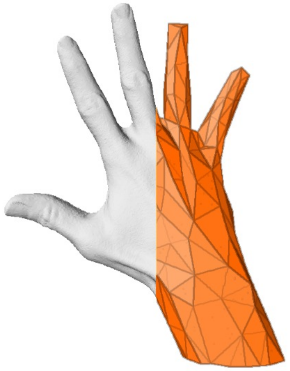


[filtering]



[shape analysis]

[parameterization]



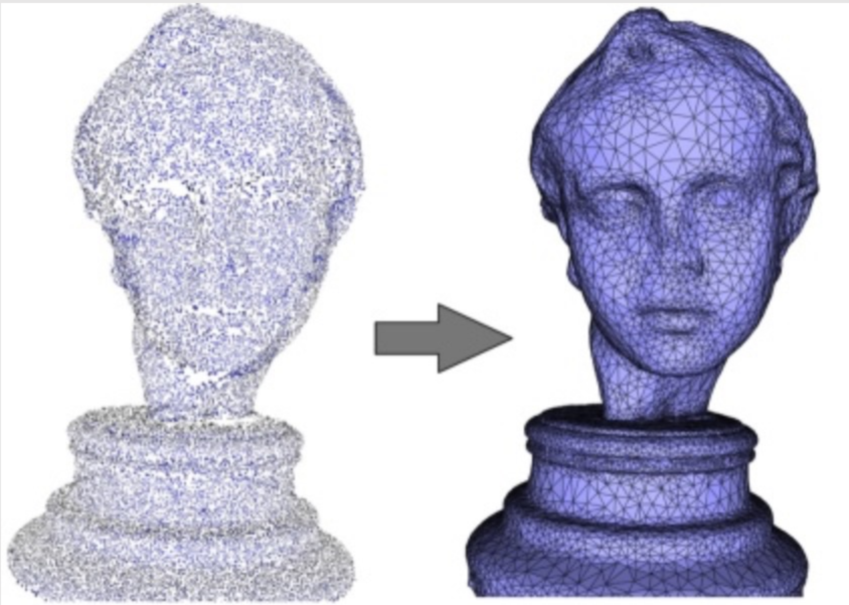
[compression]

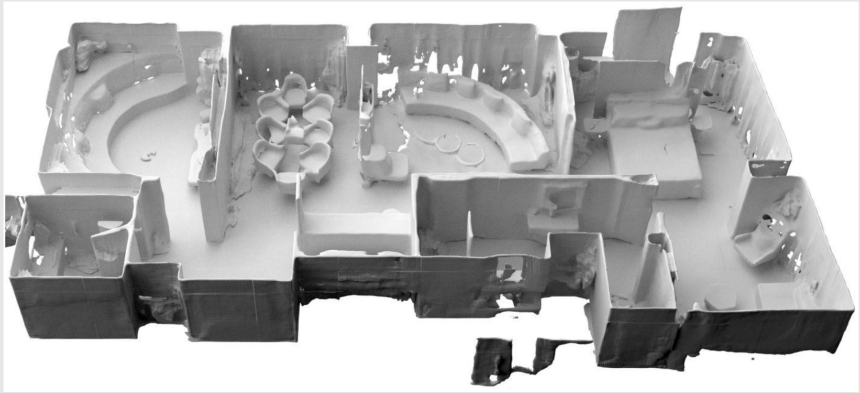
Geometry Processing: Reconstruction

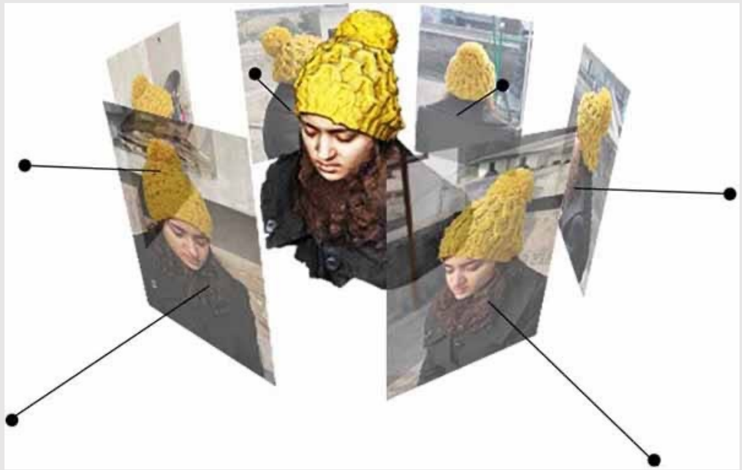
- Given samples of geometry, reconstruct surface

- **Data:** What are “samples”?
    - Points & normals
    - Image pairs / sets (multi-view stereo)
    - Line density integrals (MRI/CT scans)

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

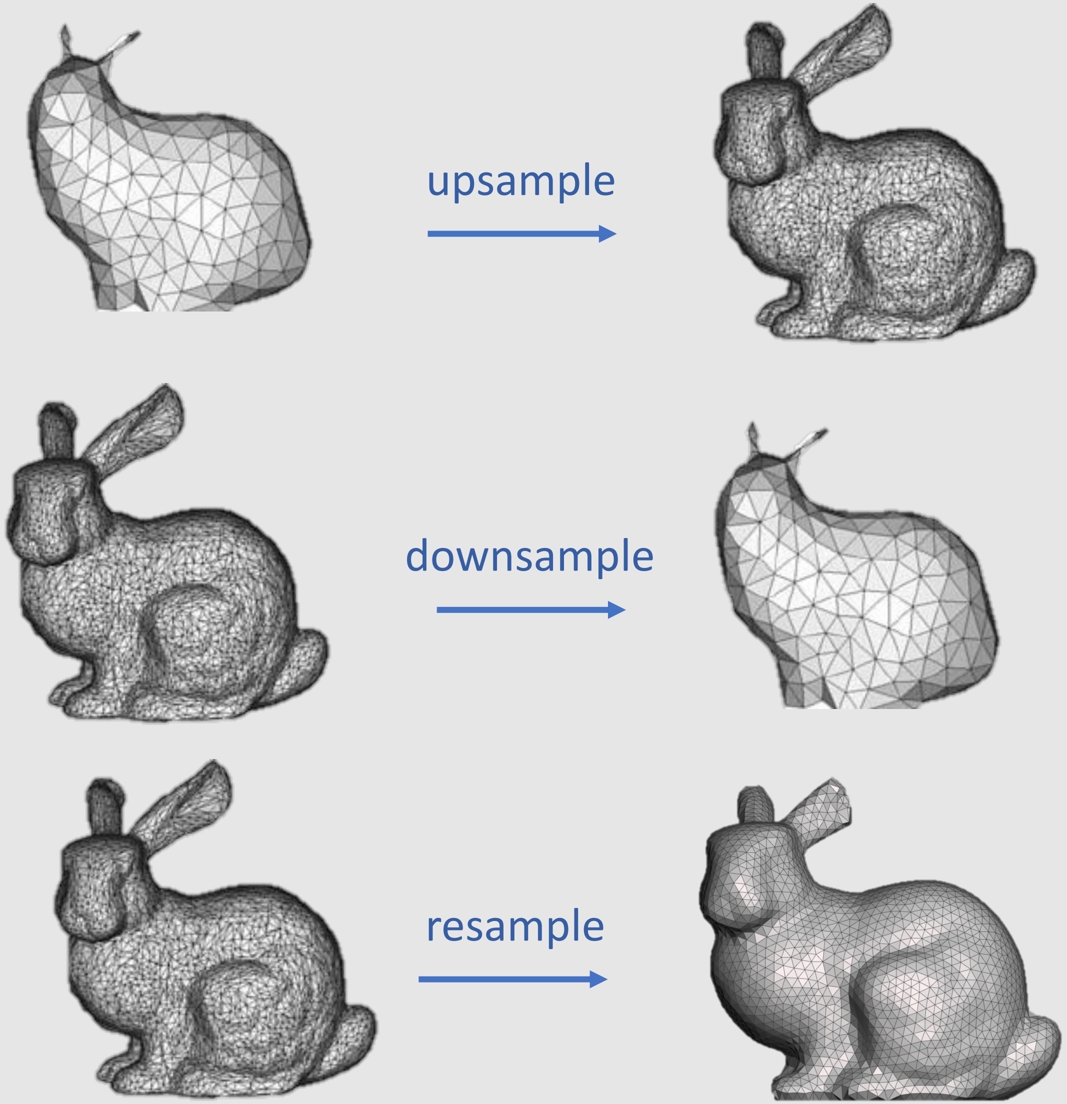






Geometry Processing: Remeshing

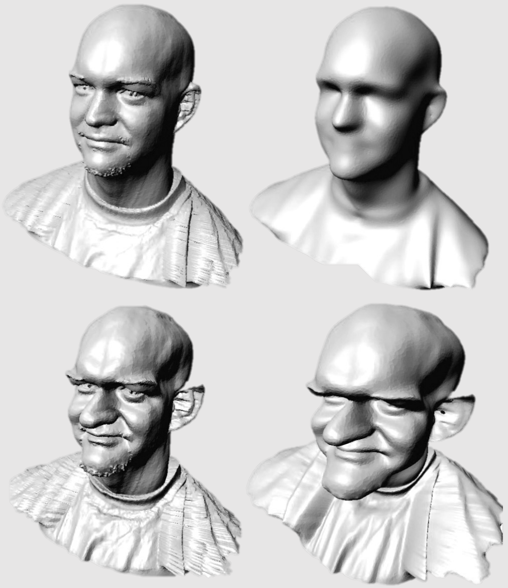
- **Upsampling**: increase resolution via interpolation
  - Subdivision
  - Bilateral upsampling
- **Downsampling**: decrease resolution via averaging
  - Subsampling
  - Iterative decimation
- **Resampling:** modify sample distribution to improve quality
  - Remeshing

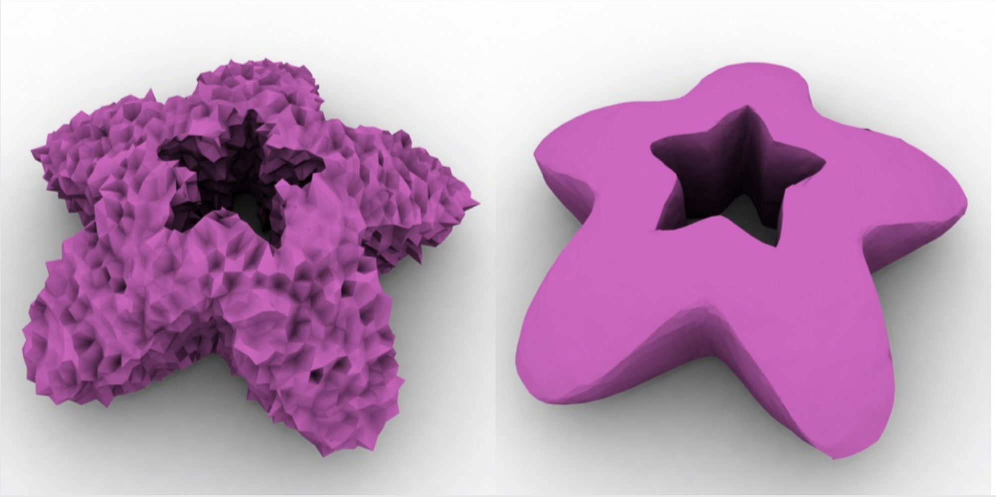


Geometry Processing: Filtering

- Remove noise, or emphasize important features (e.g., edges)
  - Curvature flow
  - Bilateral filtering
  - Spectral filtering

- Useful for cleaning up noisy 3D scans
  - **Example:** Kinect
    - Search for key facial components while smoothening out artifacts in between



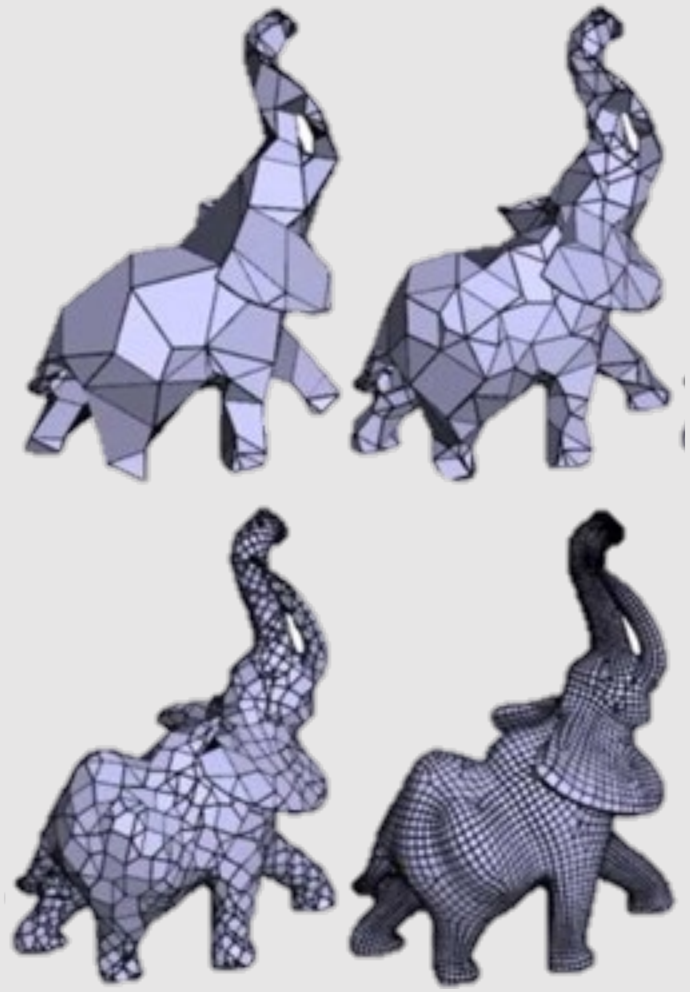


Geometry Processing: Compression

- Reduce storage size by eliminating redundant data/approximating unimportant data

- Techniques may be either lossy or lossless:
  - **Lossy:** unable to reconstruct original mesh
    - Able to compress the mesh better
  - **Lossless:** able to reconstruct original mesh
    - Not as good compression results

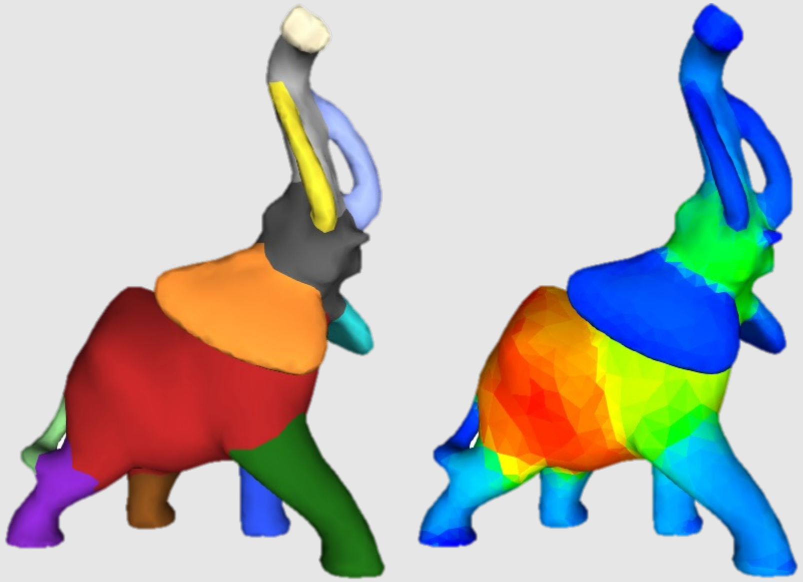
- Somewhat similar idea to downsampling
  - Added objective of wanting to recover the original mesh perfectly (lossless) or as best as possible (lossy)

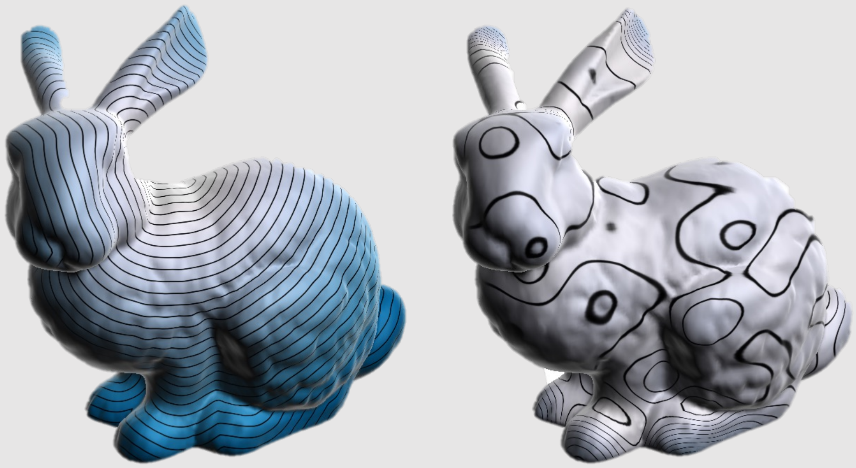


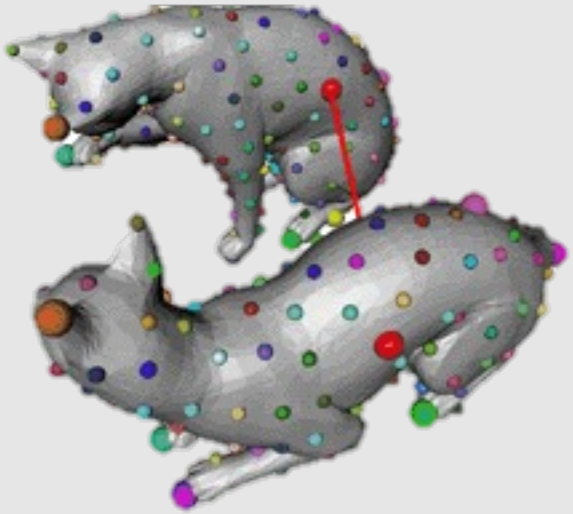
Geometry Processing: Shape Analysis

- Identify/understand important semantic features
  - Segmentation
  - Correspondence
  - Symmetry detection
  - Alignment
    - **Objective:** Compute similarities between two meshes

- Starting to become AI-driven





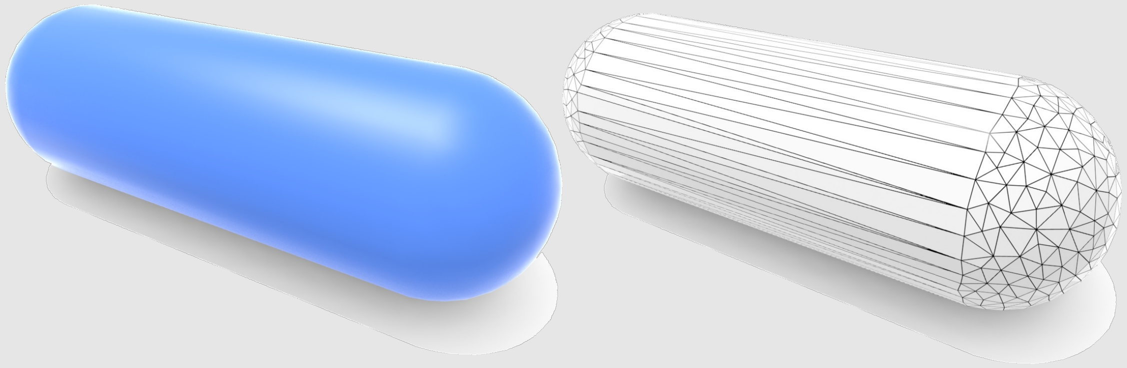


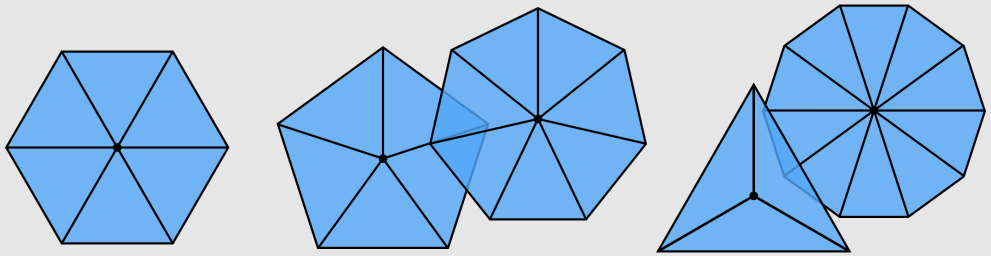
But what makes a good mesh?

A Good Mesh Has...

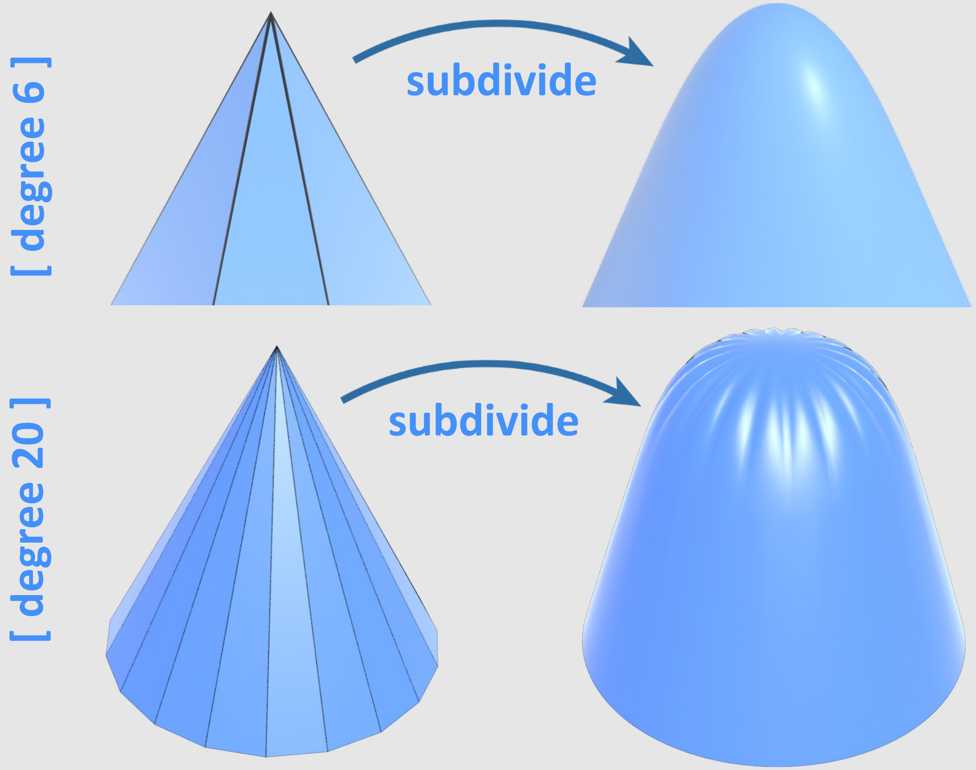
- **Good approximation of original shape**
  - Keep elements that contribute shape info
  - More elements where curvature is high

- **Regular vertex degree**
  - Degree 6 for triangle mesh, 4 for quad mesh
    - Better polygon shape
    - More regular computation
    - Smoother subdivision



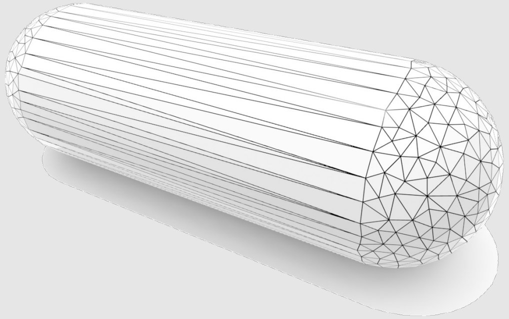


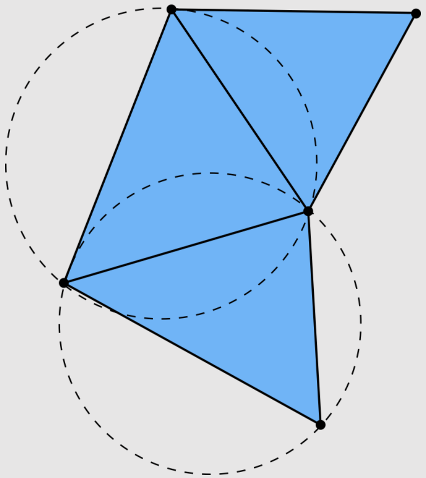
[good] [okay] [bad]



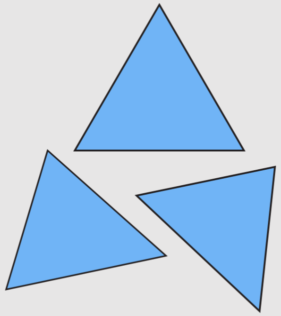
A Good Mesh Has...

- **Good triangle shape**
  - All angles close to 60 degrees
- More sophisticated condition: **Delaunay**
  - For every triangle, the unique circumcircle (circle passing through all vertices of the triangle) does not encase any other vertices
  - Many nice properties:
    - Maximizes minimum angle
    - Smoothest interpolation
- **Tradeoff:** sometimes a mesh can be approximated best with long & skinny triangles
  - Doesn't make the mesh Delaunay anymore
  - **Example:** cylinder





[delaunay]

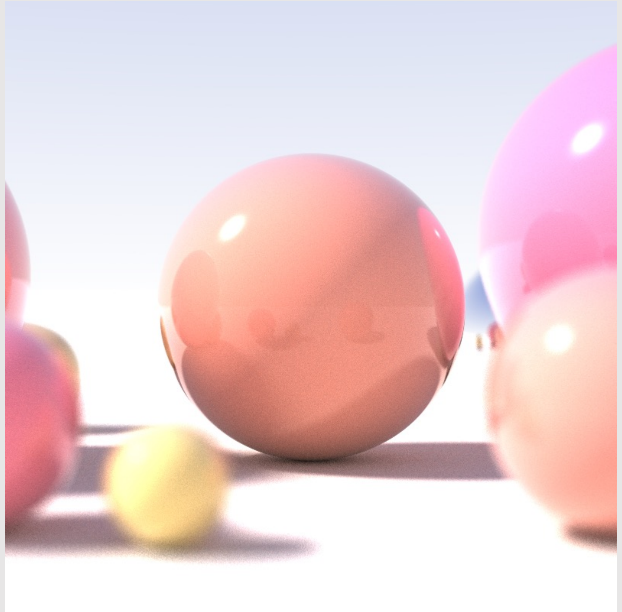


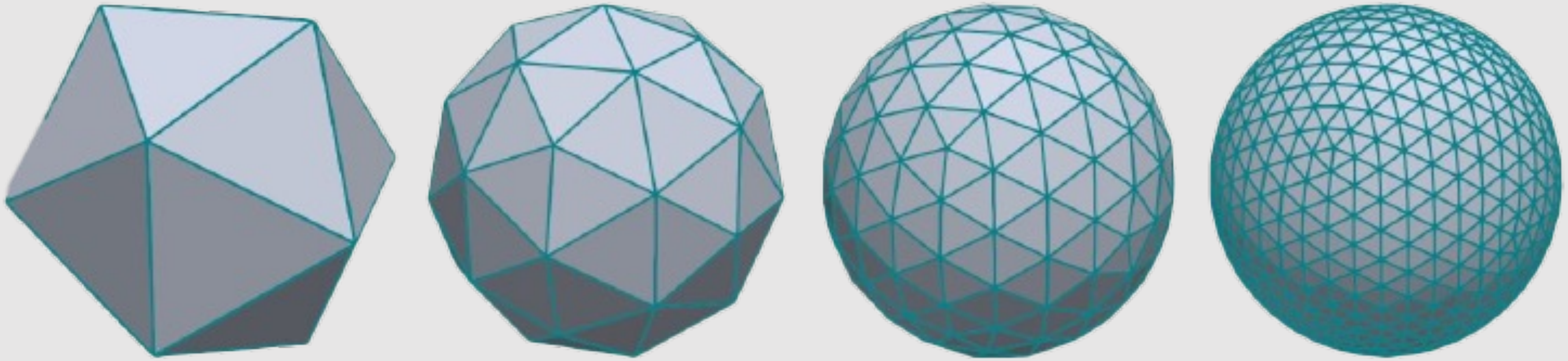
[good]

[bad]

A Good Mesh Has…

- **Good approximation on the vertices & interpolation**
  - Placing vertices on a sphere and linearly interpolating is not enough
    - Adding more vertices yields better approximation, but now too much data to store/process!
  - Need to apply correct **surface normals**



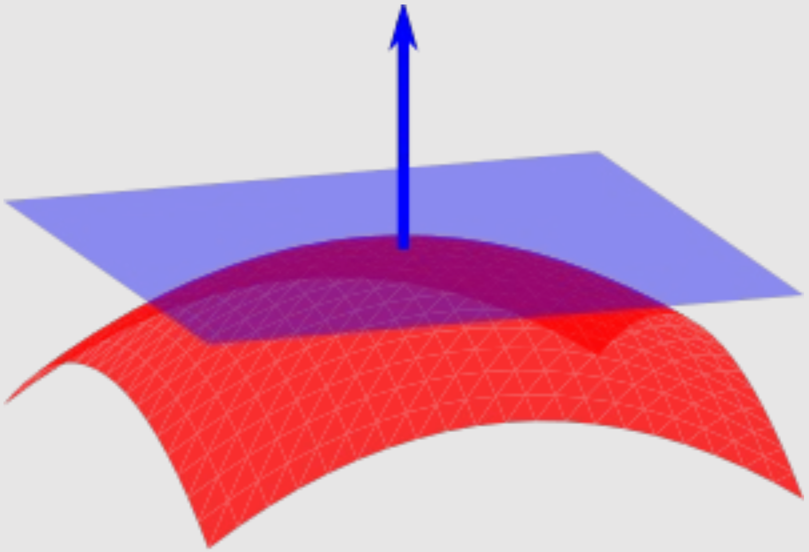


Surface Normals

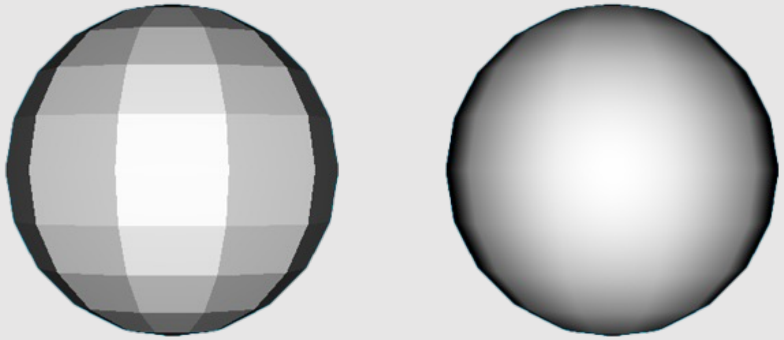
- A **surface normal** is a vector that is perpendicular to the surface at a given point
  - The surface normal for a surface $z = f(x, y)$ at point (x', y') is:

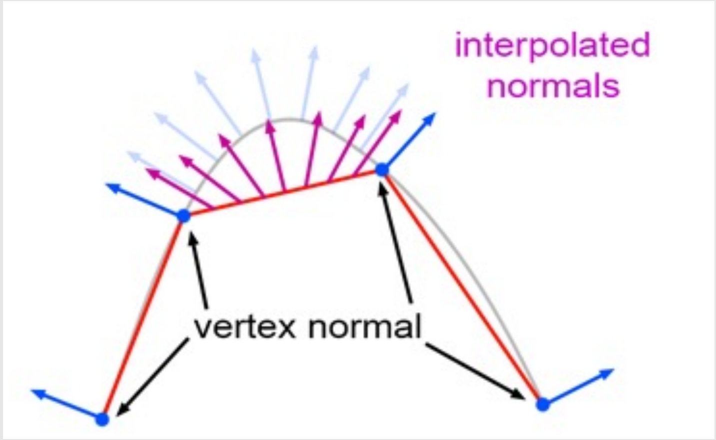
$$N_s = \begin{bmatrix} f_x(x', y') \\ f_y(x', y') \\ -1 \end{bmatrix}$$

  - Value assigned per-vertex



- Surface normal are interpolated via-barycentric coordinates and extruded in that direction to provide the appearance of curvature during rendering





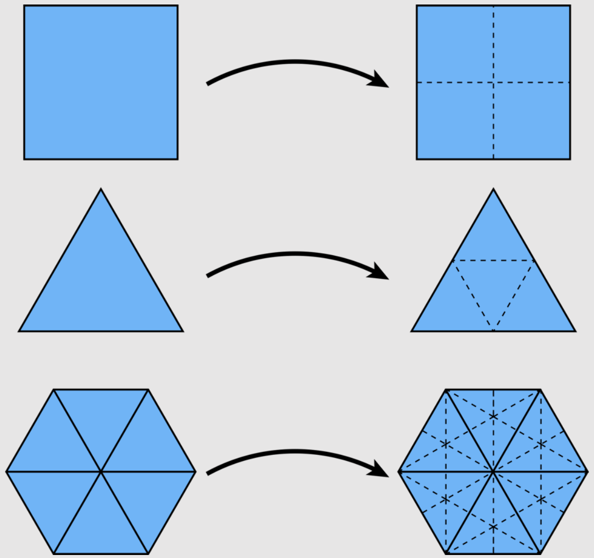
- ~~Manifolds~~

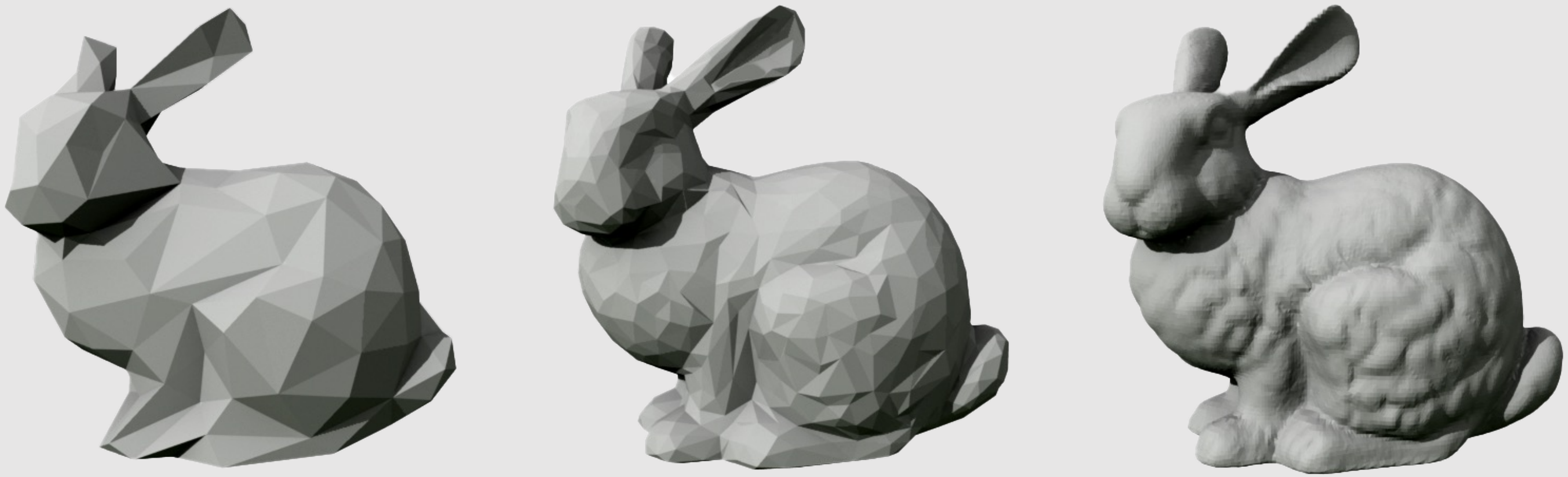
- ~~Mesh representations and local operations~~

- Digital Geometric Processing
    - ~~Good Geometry~~
    - Geometric Subdivision
    - Geometric Simplification
    - Geometric Remeshing
    - Geometric Queries

Subdivision

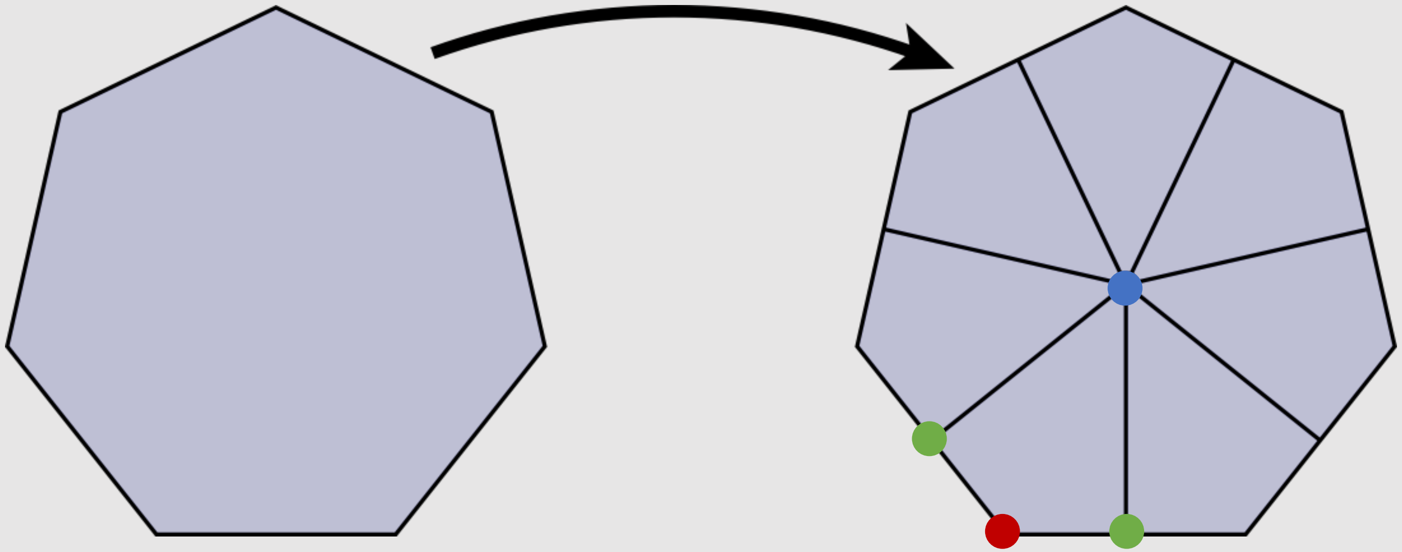
- Subdivison is the process of **upsampling** a mesh
- General formula:
  - **Split Step:** split faces into smaller faces
  - **Move Step:** replace vertex positions/properties with weighted average of neighbors





Linear Subdivision [Split Step]

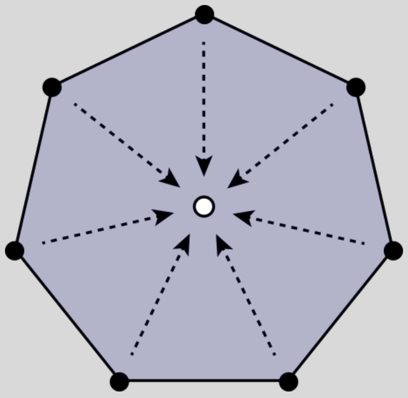
- Split every polygon (any # of sides) into quadrilaterals



- Each new quadrilateral now has:
  - **[face coords]   : 1 new vertex from the mesh face center**
  - **[edge coords]   : 2 new vertices from the new edges**
  - **[vertex coords] : 1 new vertex from the original mesh face**

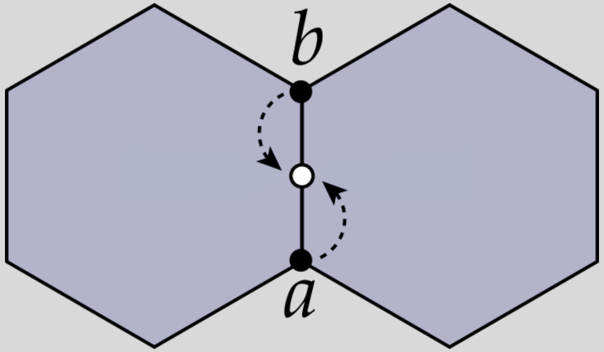
Linear Subdivision [Move Step]

Step 1: Face Coords



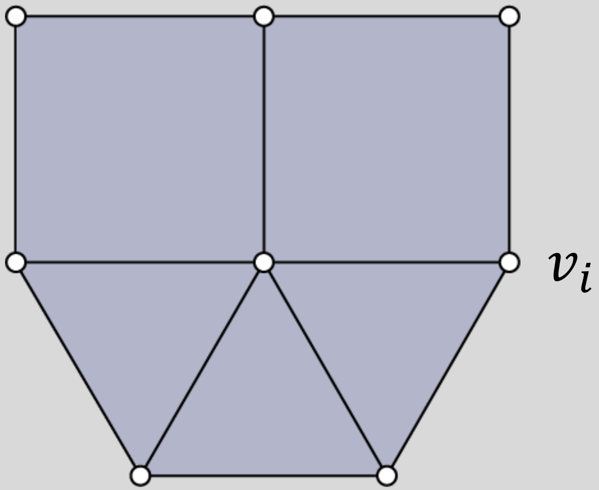
$$\frac{1}{n}\sum_i p_i$$

Step 2: Edge Coords



$$(a + b)\ /\ 2$$

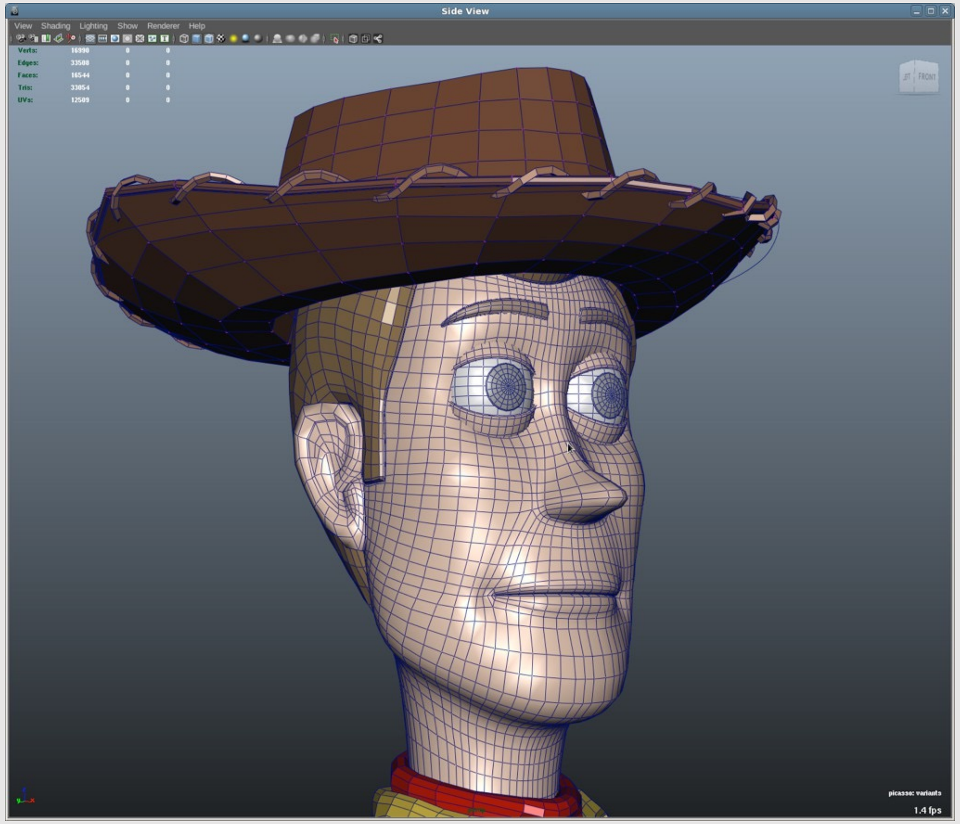
Step 3: Vertex Coords



$$v_i = v_i$$

Catmull Clark Subdivision

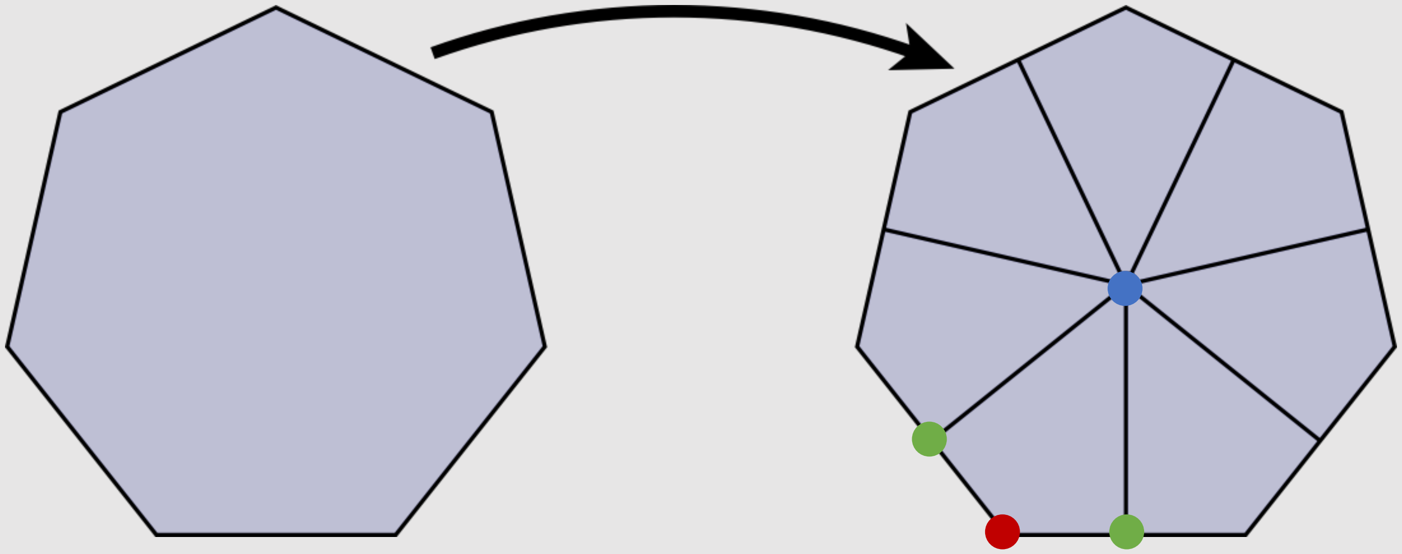
- In 1978, Edwin Catmull (Pixar co-founder) and Jim Clark wanted to create a generalization of **uniform bi-cubic b-splines** for 3D meshes
    - We will cover what this means in a future lecture :)
- Became ubiquitous in graphics
    - Helped Catmull win an Academy Award for Technical Achievement in 2005



OpenSubdiv V2 (2018) Pixar

Catmull-Clark Subdivision [Split Step]

- Split every polygon (any # of sides) into quadrilaterals



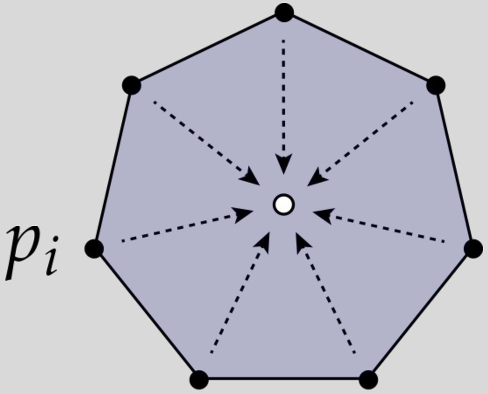
- Each new quadrilateral now has:
  - [face coords] : 1 new vertex from the mesh face center
  - [edge coords] : 2 new vertices from the new edges
  - [vertex coords] : 1 new vertex from the original mesh face

No different than Linear Subdivision!

Catmull-Clark Subdivision [Move Step]

Step 1:

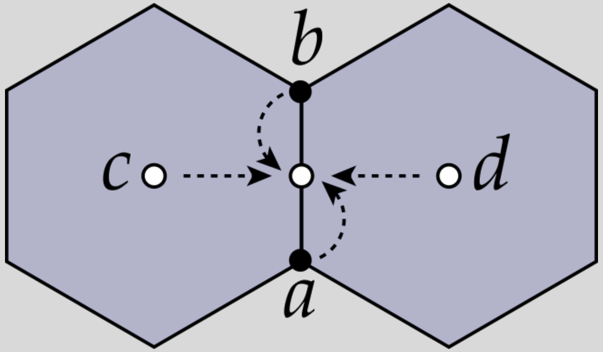
Face Coords



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

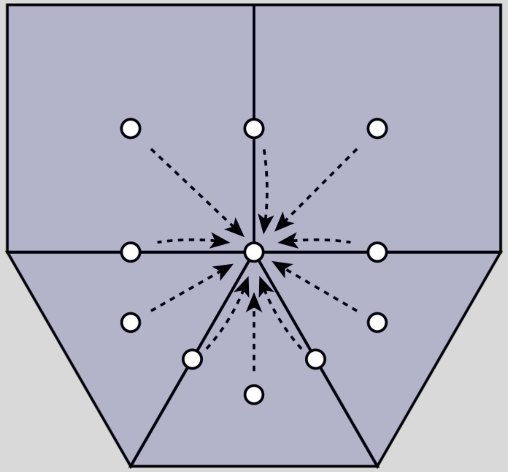
Step 2:

Edge Coords



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

Step 3:



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

Vertex Coords

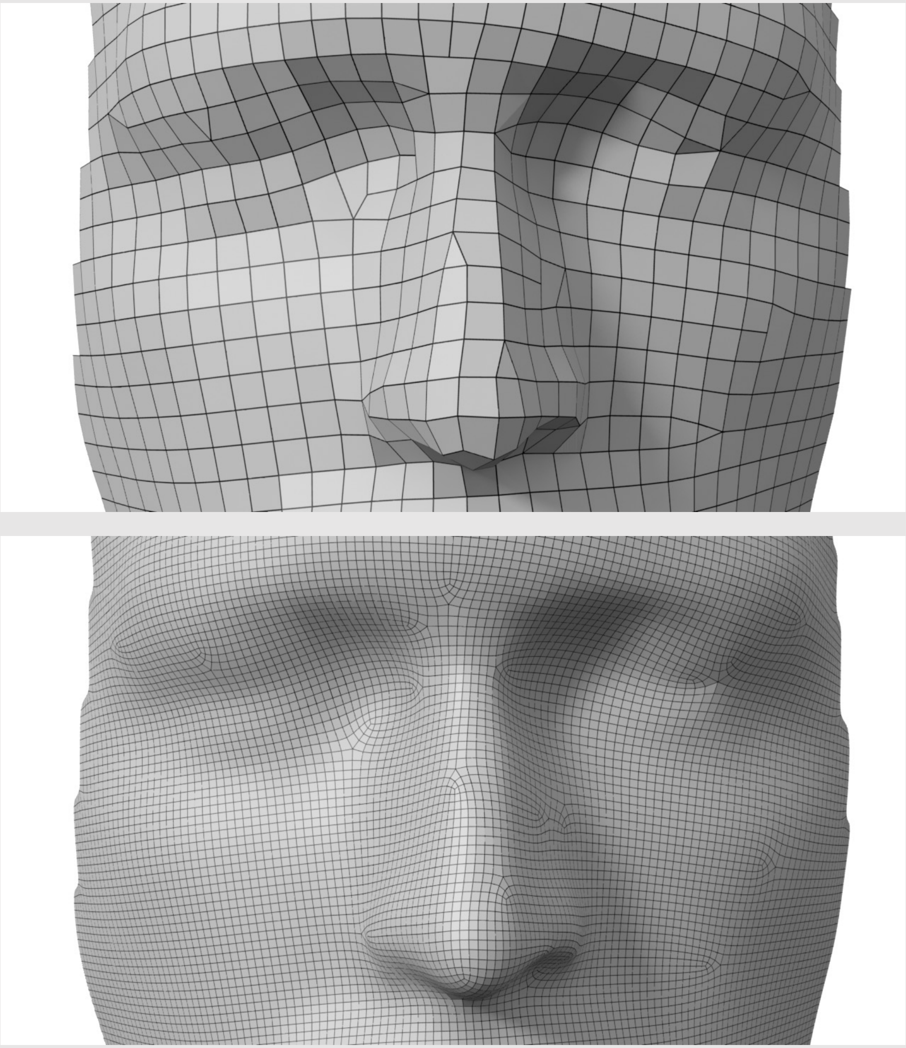
n - vertex degree

Q - average of face coords around vertex

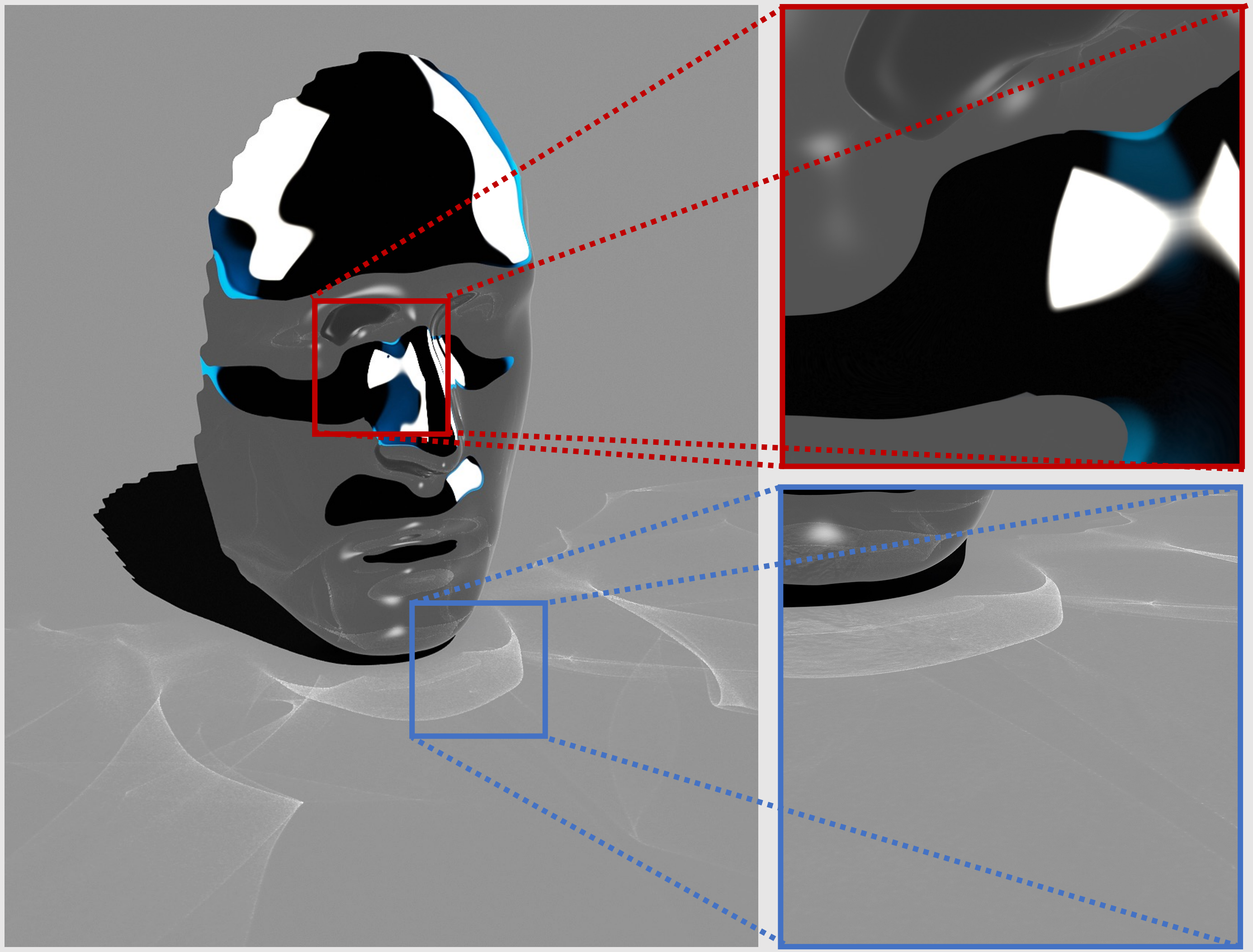
R - average of edge coords around vertex

S - original vertex position

Catmull-Clark Subdivision [Quads]



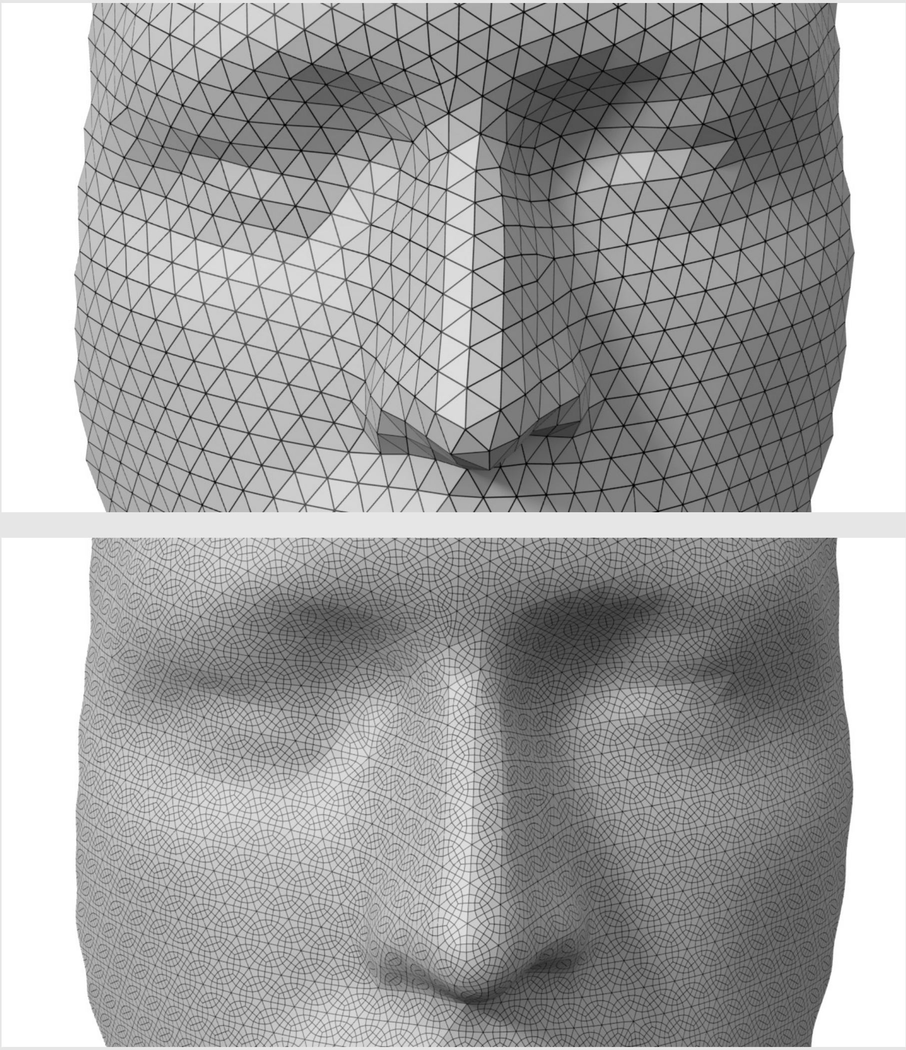
Few irregular vertices



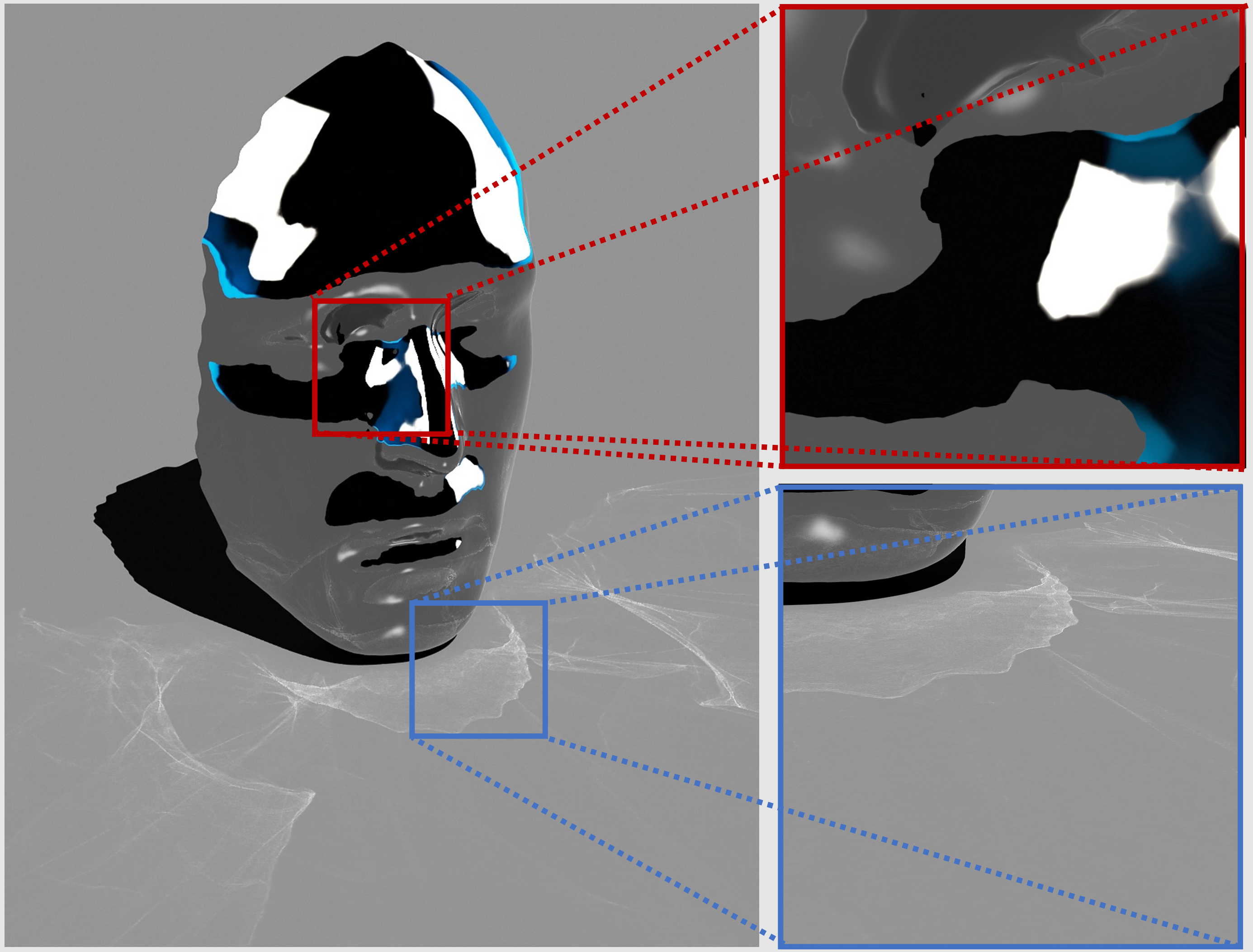
Smoothly-varying surface normals

Smooth reflections/caustics

Catmull-Clark Subdivision [Triangles]



Many irregular vertices



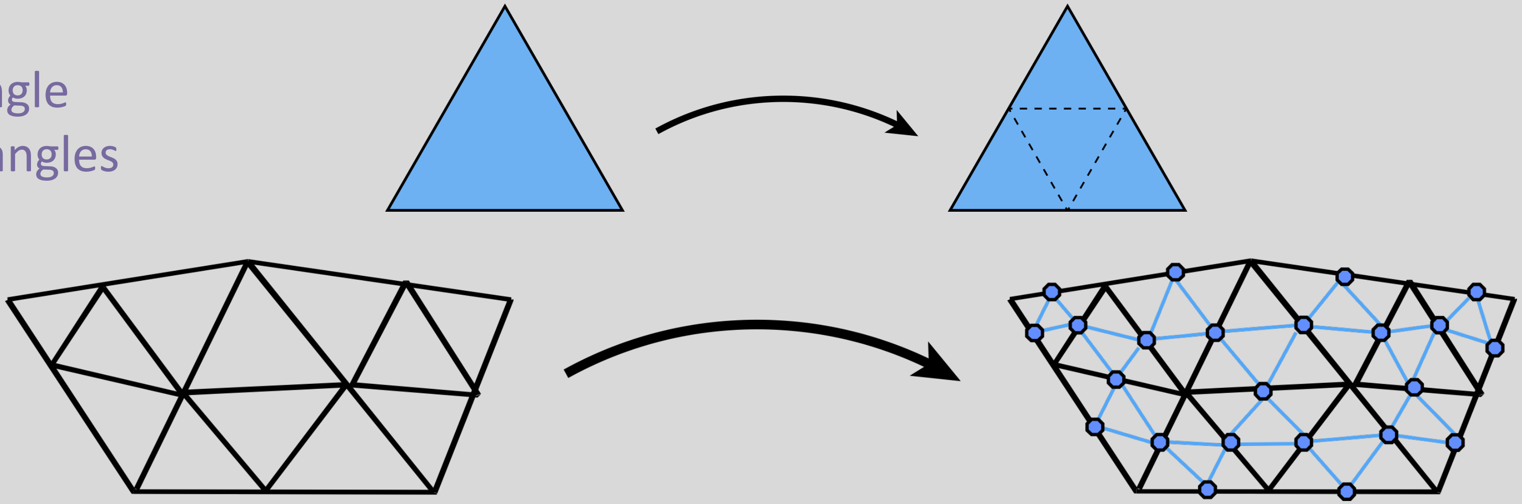
Erratic surface normals

Jagged reflections/caustics

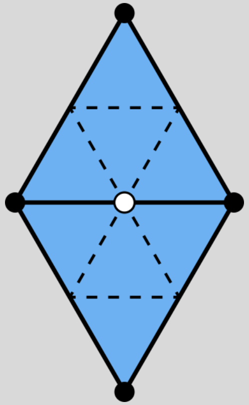
Is there a better subdivision scheme we can use for triangulated meshes?

Loop Subdivision

Step 1:
Split triangle
into 4 triangles

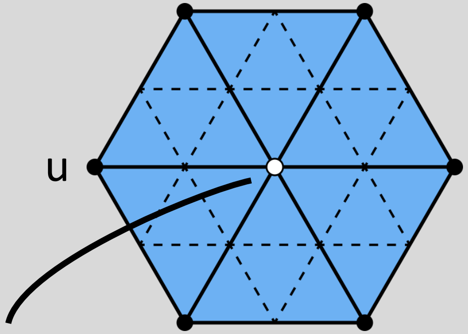


Step 2:
Assign new coords



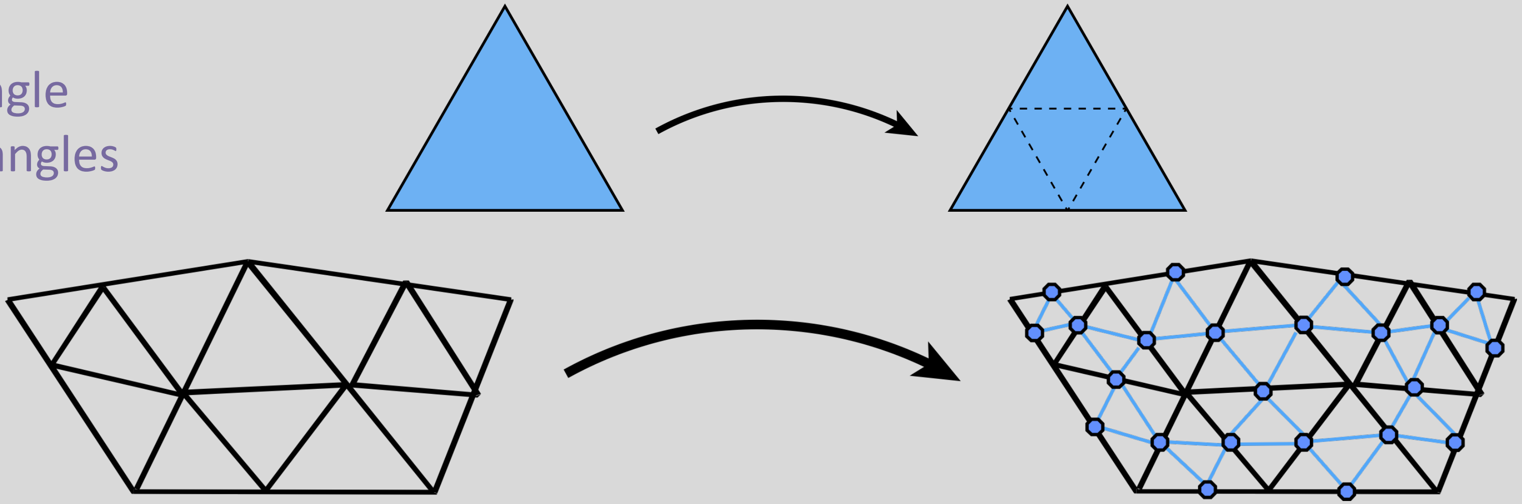
Step 3:
Assign old coords

n - vertex degree
u - 3/16 if n=3
3/(8n) otherwise

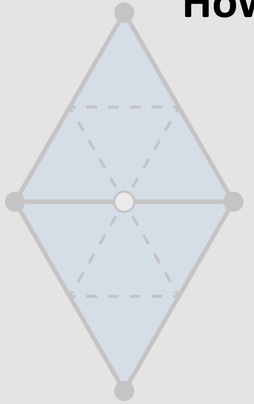


Loop Subdivision

Step 1:
Split triangle
into 4 triangles



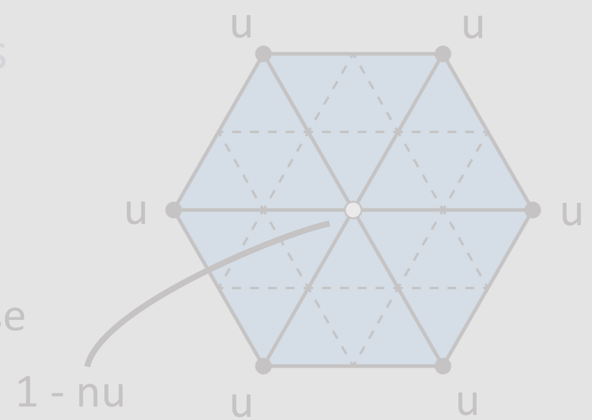
Step 2:
Assign new coords



How do we efficiently do Step 1?

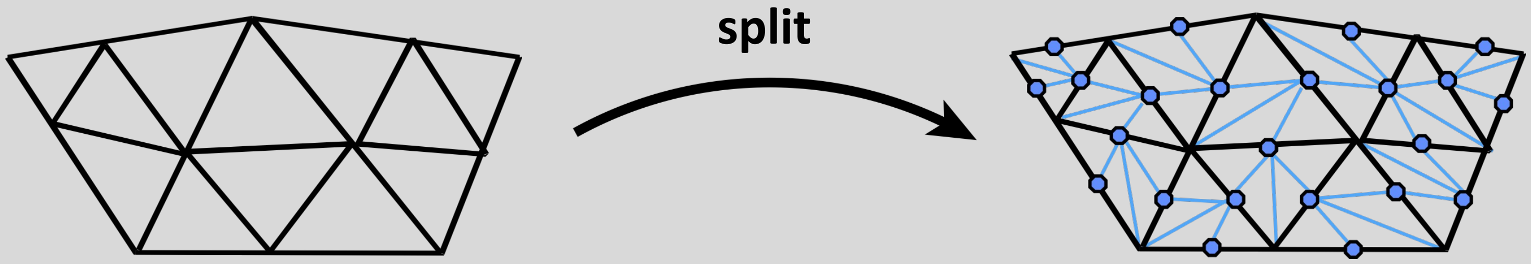
Step 3:
Assign old coords

n - vertex degree
u - 3/16 if n=3
3/(8n) otherwise

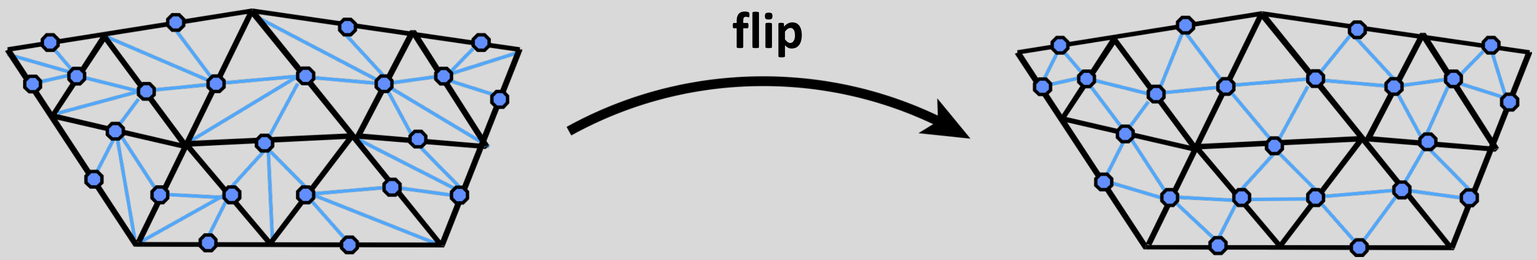


Loop Subdivision Using Local Ops

Step 1:
Split all edges in any order

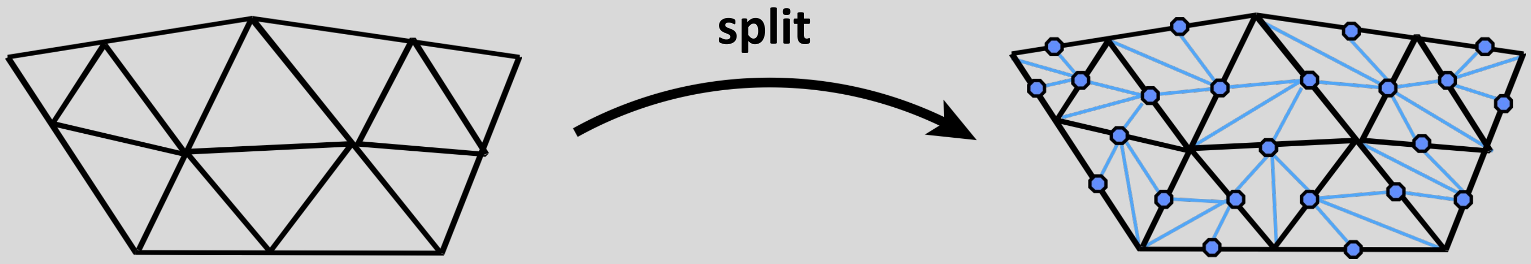


Step 2:
Flip new edges until they touch two new vertices



Loop Subdivision Using Local Ops

Step 1:
Split all edges in any order



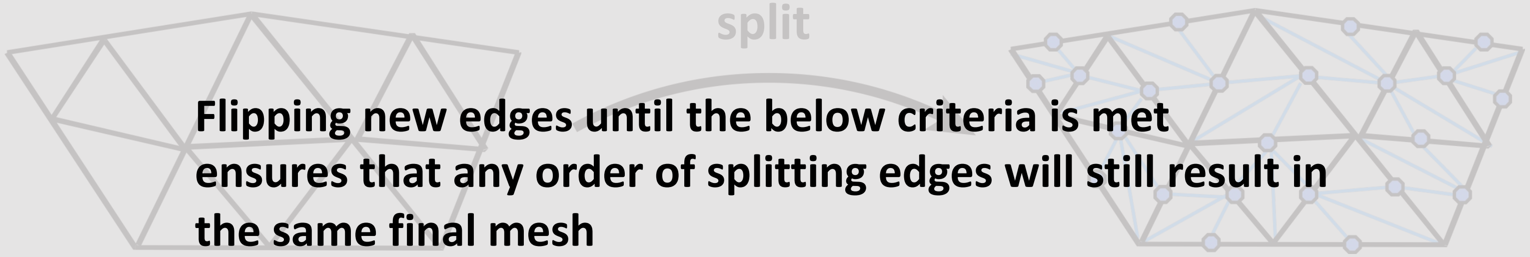
Step 2:
Flip new edges until they touch two new vertices

The order we traverse the edges and split them matter!

Traversing edges forward and splitting vs traversing them backwards and splitting will yield different meshes

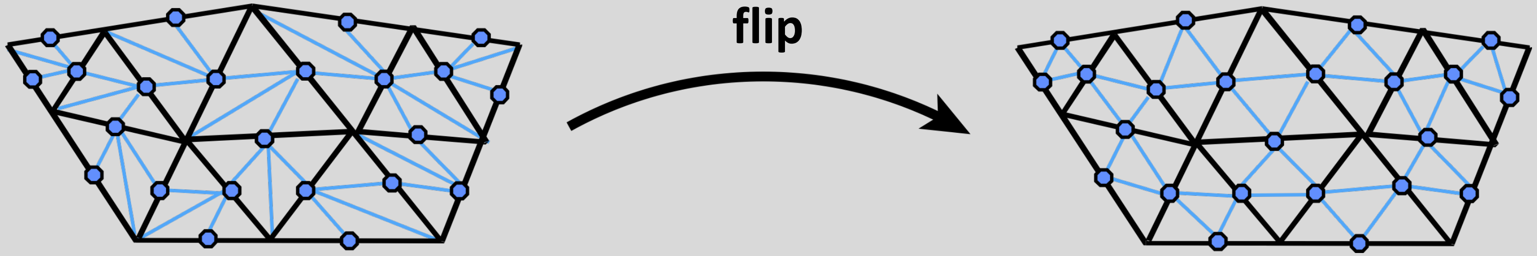
Loop Subdivision Using Local Ops

Step 1:
Split all edges in any order



Flipping new edges until the below criteria is met ensures that any order of splitting edges will still result in the same final mesh

Step 2:
Flip new edges until they touch two new vertices



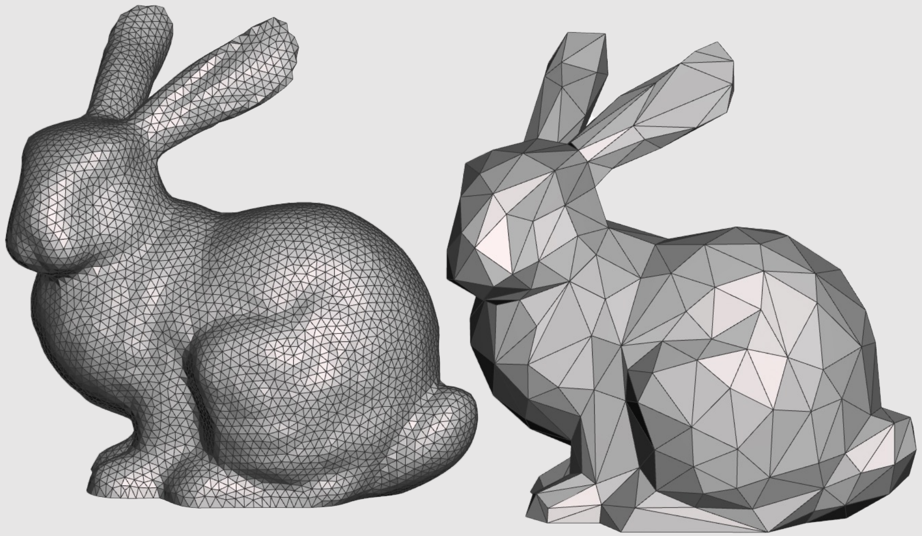
- ~~Manifolds~~

- ~~Mesh representations and local operations~~

- Digital Geometric Processing
    - ~~Good Geometry~~
    - ~~Geometric Subdivision~~
    - Geometric Simplification
    - Geometric Remeshing
    - Geometric Queries

Simplification

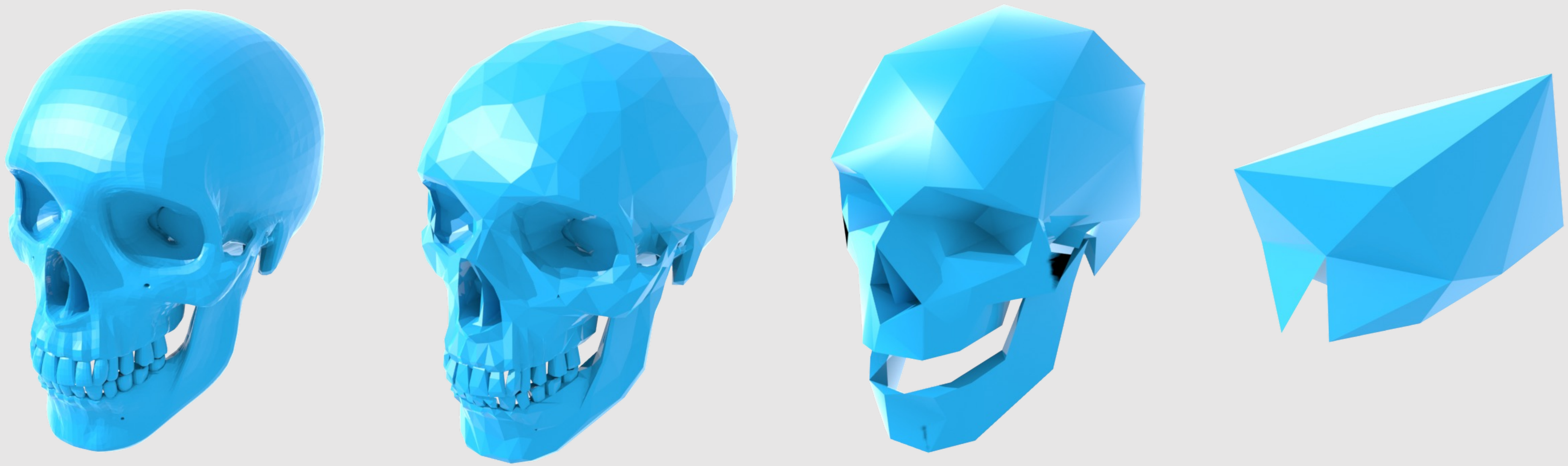
- Simplification is the process of **downsampling** a mesh
  - Less Storage overhead
    - Smaller file sizes
  - Less Processing overhead
    - Less elements to iterate over
  - Larger mesh modifications
    - Instead of moving tens of smaller mesh elements, move one larger mesh element



Simplification Algorithm Basics

- 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****



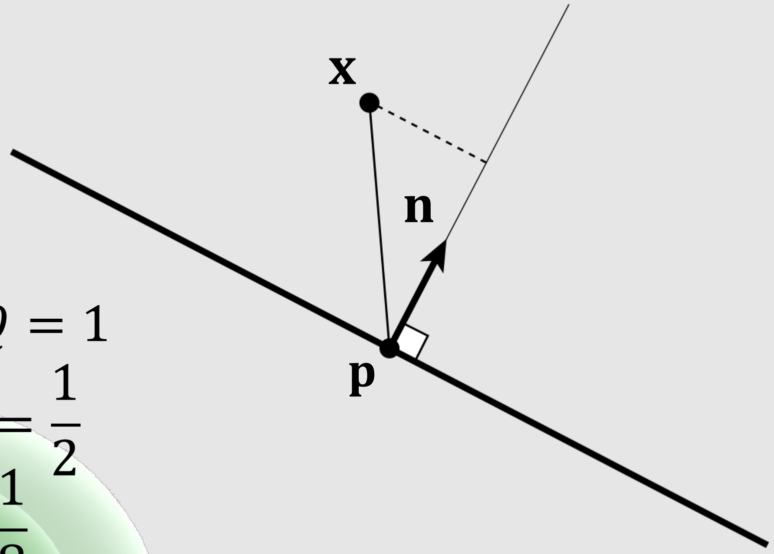
[30,000 triangles]   [3,000 triangles]   [300 triangles]   [30 triangles]

**invented at CMU (Garland & Heckbert 1997)

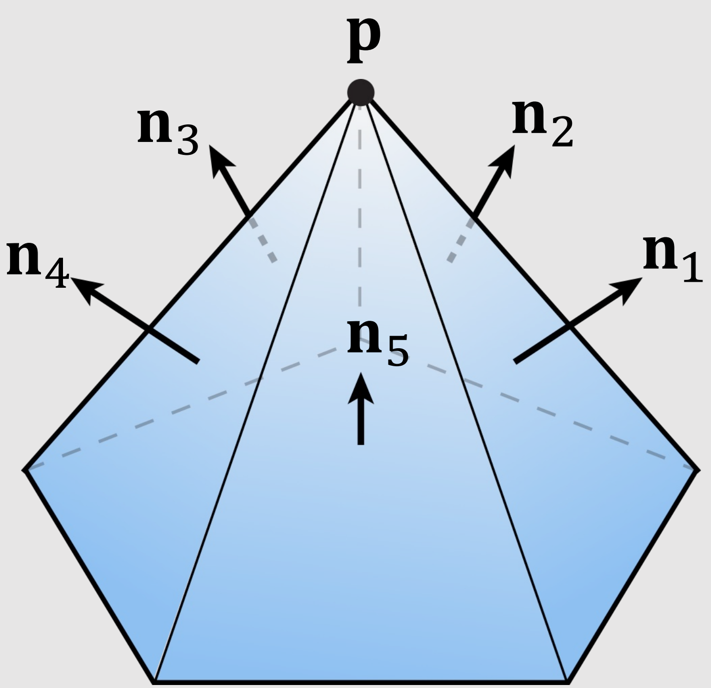
Quadric Error Metric

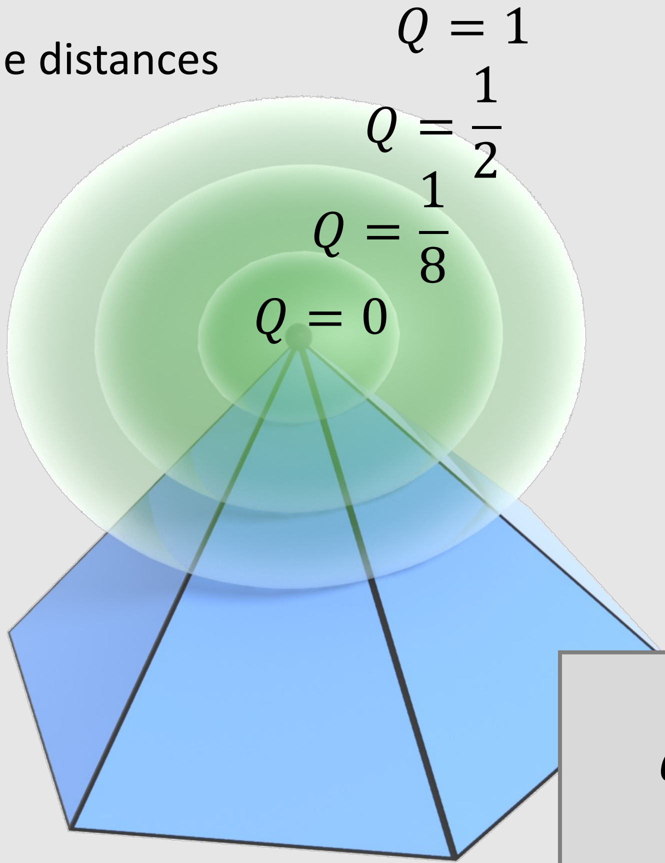
- **Goal:** approximate a point’s distance from a collection of triangles
  - **Review:** what is the distance of a point $\mathbf{x}$ from a plane $\mathbf{p}$ with normal $\mathbf{n}$?

$$\mathrm{dist}(\mathbf{x}) = \langle \mathbf{n}, \mathbf{x} \rangle - \langle \mathbf{n}, \mathbf{p} \rangle = \langle \mathbf{n}, \mathbf{x} - \mathbf{p} \rangle$$



- Quadric error is the sum of squared point-to-plane distances

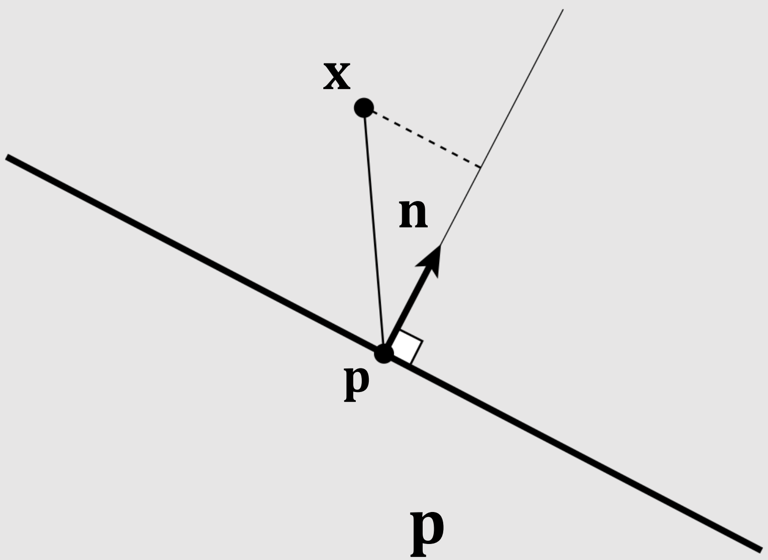




$$Q(\mathbf{x}) := \sum_{i=1}^{k} \langle \mathbf{n}_i, \mathbf{x} - \mathbf{p} \rangle^2$$

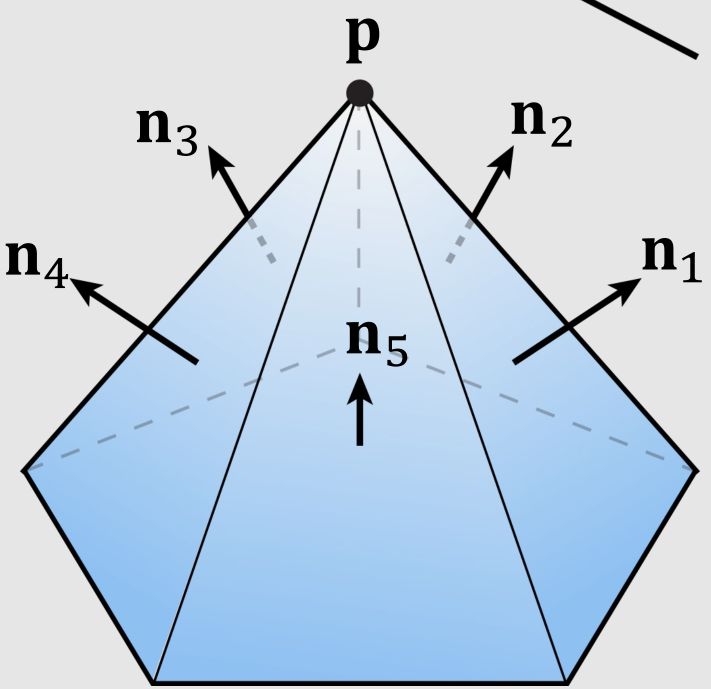
Quadric Error Metric

- Given:
  - Query point $\mathbf{x} = (x, y, z)$
  - Normal $\mathbf{n} = (a, b, c)$
  - Offset from origin $d = \langle \mathbf{n}, \mathbf{p} - 0 \rangle = \langle \mathbf{n}, \mathbf{p} \rangle$
- We can rewrite in homogeneous coordinates:
  - $\mathbf{u} = (x, y, z, 1)$
  - $\mathbf{v} = (a, b, c, d)$



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

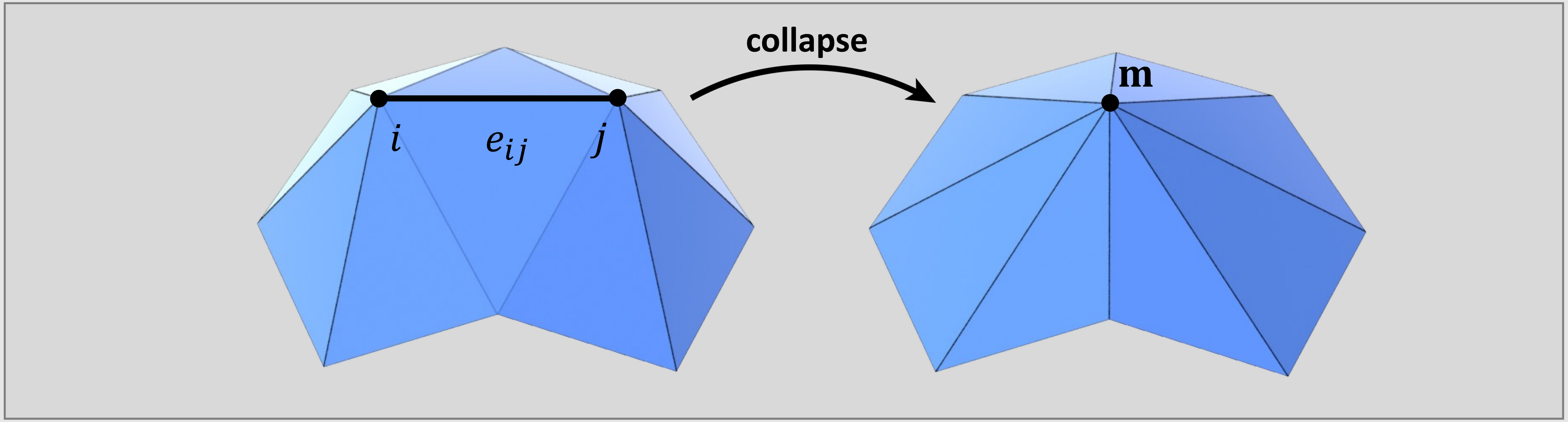
$$K = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$



- **Key Idea:** sum of matrices K represents distance to a union of planes

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

Quadric Error of Edge Collapse

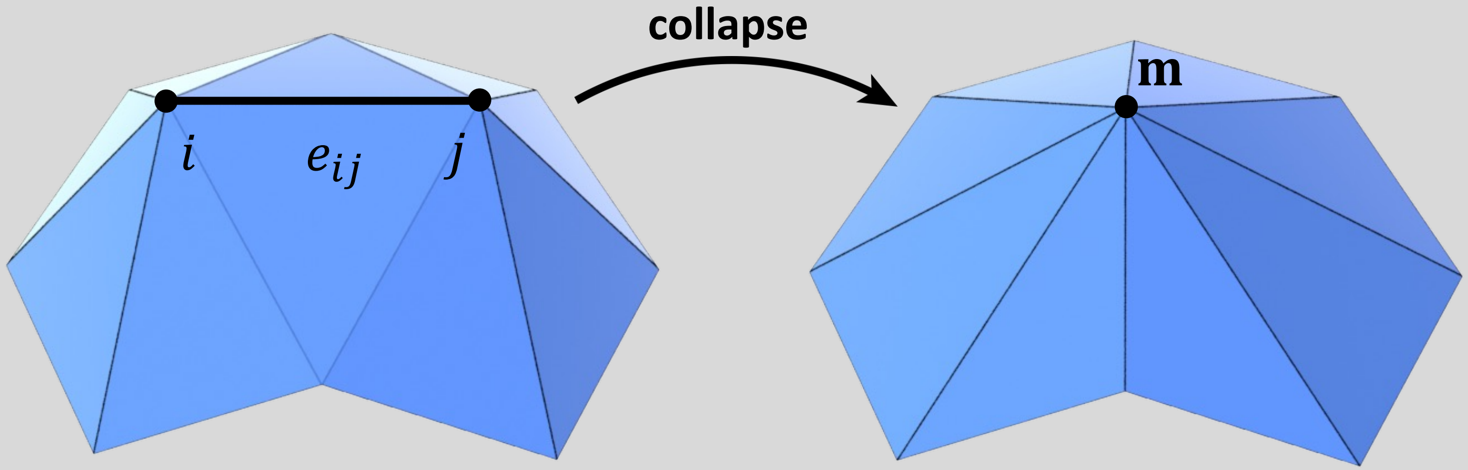


- How much does it cost to collapse an edge e_{ij}?
  - Compute midpoint $\mathbf{m}$, measure error as

$$Q(\mathbf{m}) = \mathbf{m}^\mathsf{T}(K_i + K_j)\mathbf{m}$$

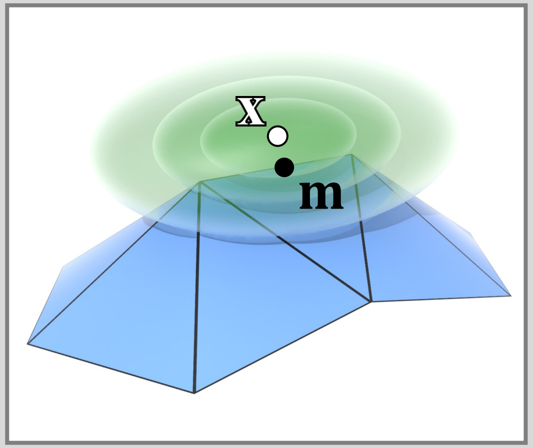
- Error becomes "score" for e_{ij}, determining priority
  - Q: where to put $\mathbf{m}$?

Quadric Error of Edge Collapse



$$Q(\mathbf{m}) = \mathbf{m}^{\mathsf{T}}(K_i + K_j)\mathbf{m}$$

- Find point $\mathbf{x}$ that minimizes error
  - Take derivatives!



How to take a derivative of a function involving matrices?

Minimizing a Quadratic Function

To find the min of a function $f(x)$

$$f(x) = ax^2 + bx + c$$

take derivative $f'(x)$ and set equal to 0

$$f'(x) = 2ax + b = 0$$

$$\boxed{x = -b/2a}$$

same structure

can also write any quadratic function of n variables as a symmetric matrix A
consider the multivariable function

$$f(x, y) = ax^2 + bxy + cy^2 + dx + ey + g$$

we can rewrite it as:

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

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

take derivative $f'(x)$ and set equal to 0

$$f'(x, y) = 2A\mathbf{x} + \mathbf{u} = 0$$

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

same structure

Positive Definite Quadratic Form

How do we know if our solution minimizes quadratic error?

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

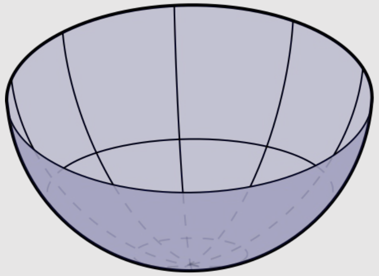
In the 1D case, we minimize the function if

$$xax = ax^2 > 0$$
$$a > 0$$

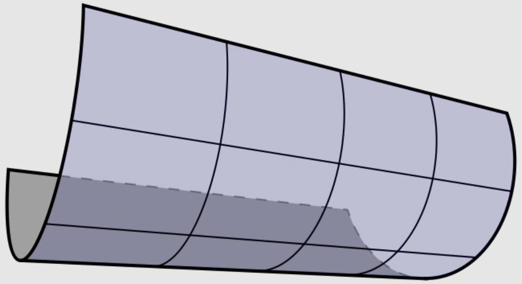
In the ND case, we minimize the function if

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

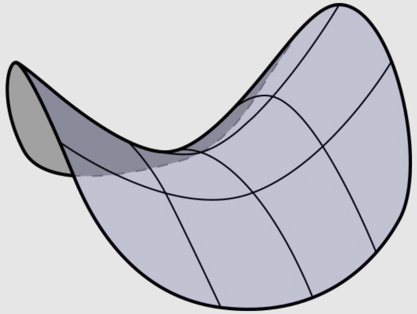
This is known as the function being positive semidefinite



[positive definite]



[positive semidefinite]



[indefinite]

Minimizing Quadratic 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
Break up our quadratic function into two pieces

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

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

Can minimize as before

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

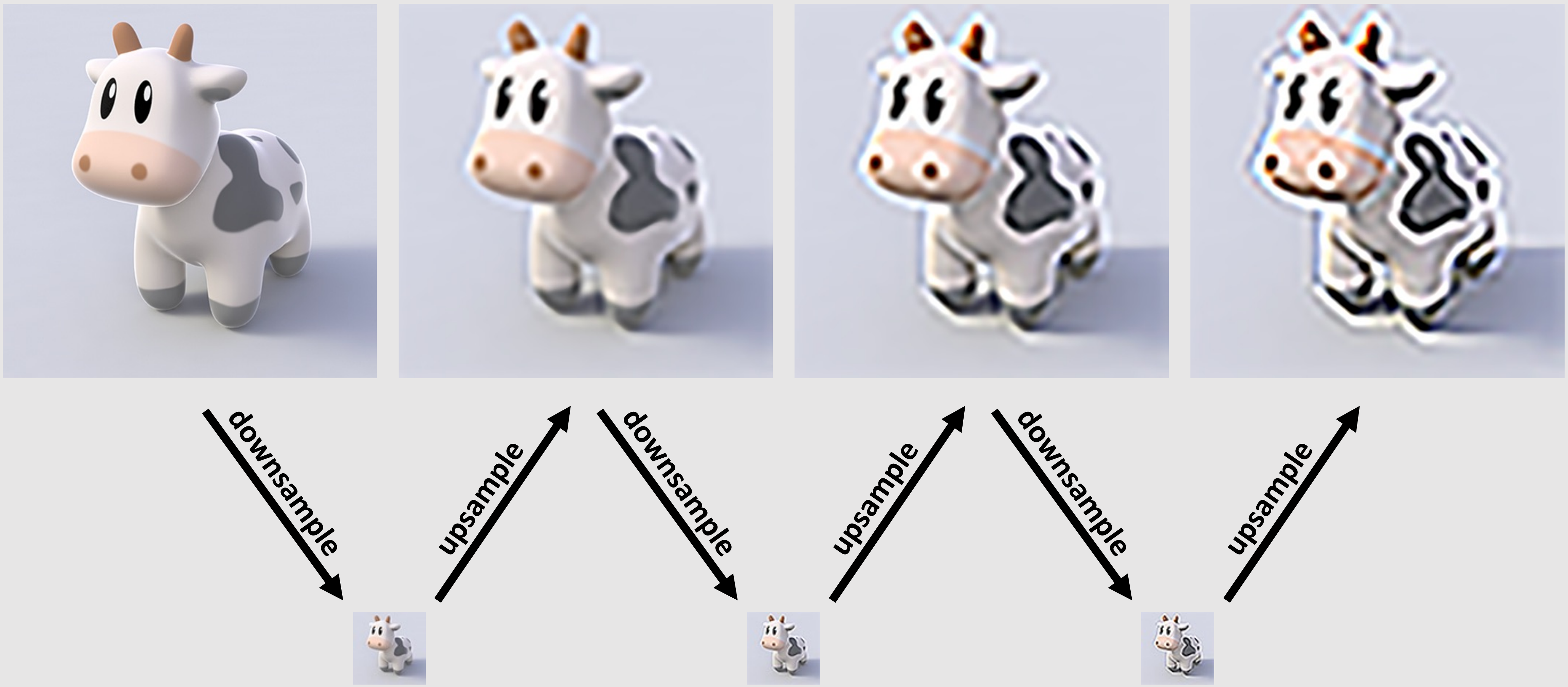
Quadric Error Simplification Algorithm

```
// compute K for each face
for(v : vertices) {
        for(f : faces) {
                Vec4 ve(N, d);
                f->K = outer(ve, ve);
        }
}

// compute K for each vertex
for(v : vertices)
        for(f : v->faces())
                v->K += f->K;

// compute K for each edge
// place into priority queue
PriorityQueue pq;
for(e : edge) {
        for(v : e->vertices())
                e->K += v->K;
        pq.push(e->K, e);
}
```

```
// iterate until mesh is a target size
while(faces.length() < target_size) {

        // collapse edge with smallest cost
        e = pq.pop();
        K = e->K;
        v = collapse(e);

        // position new vertex to optimal pos
        v->pos = -B.inv() * w

        // update K for vertex
        // update K for edges touching vertex
        v->K = K;
        for(e2 : v->edges()) {
                e2->K = 0
                for(v2 : e2->vertices())
                        e2->K += v2->K;
        }
}
```

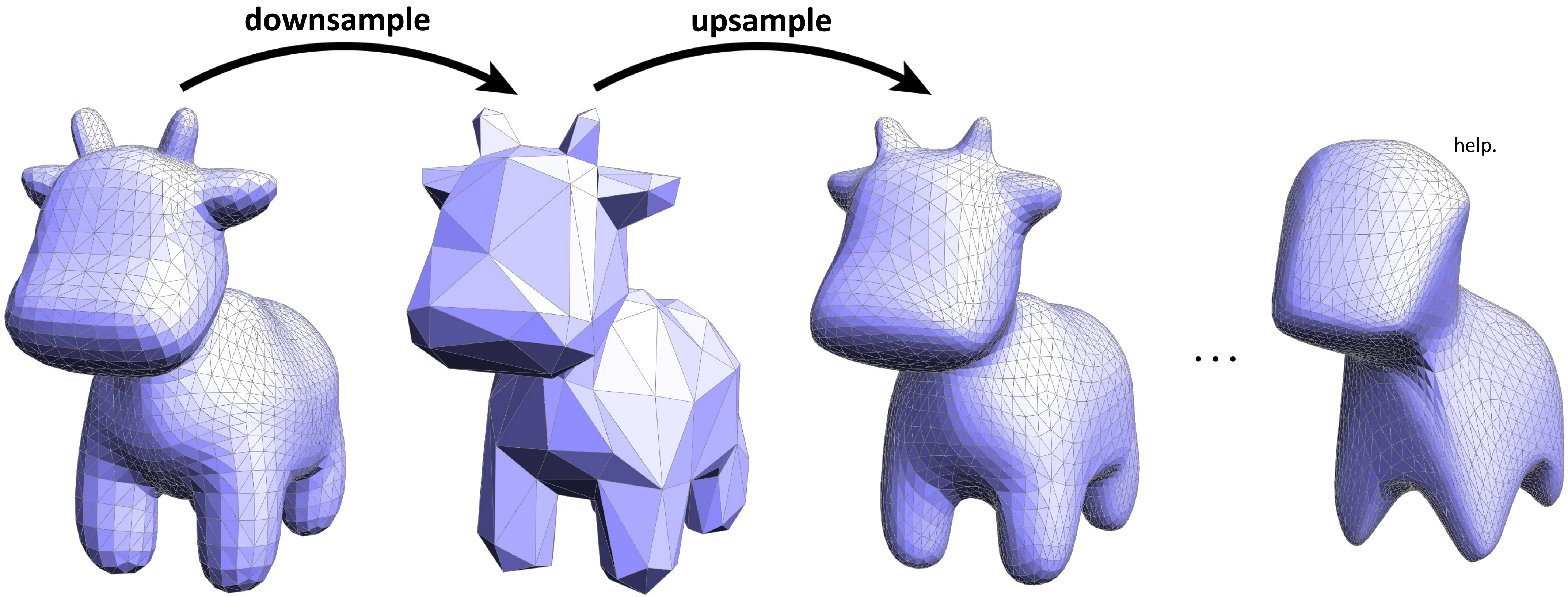
Is simplification the inverse operation of subdivision?

Dangers of Resampling



Repeatedly resampling an image degrades signal quality!

Dangers of Resampling



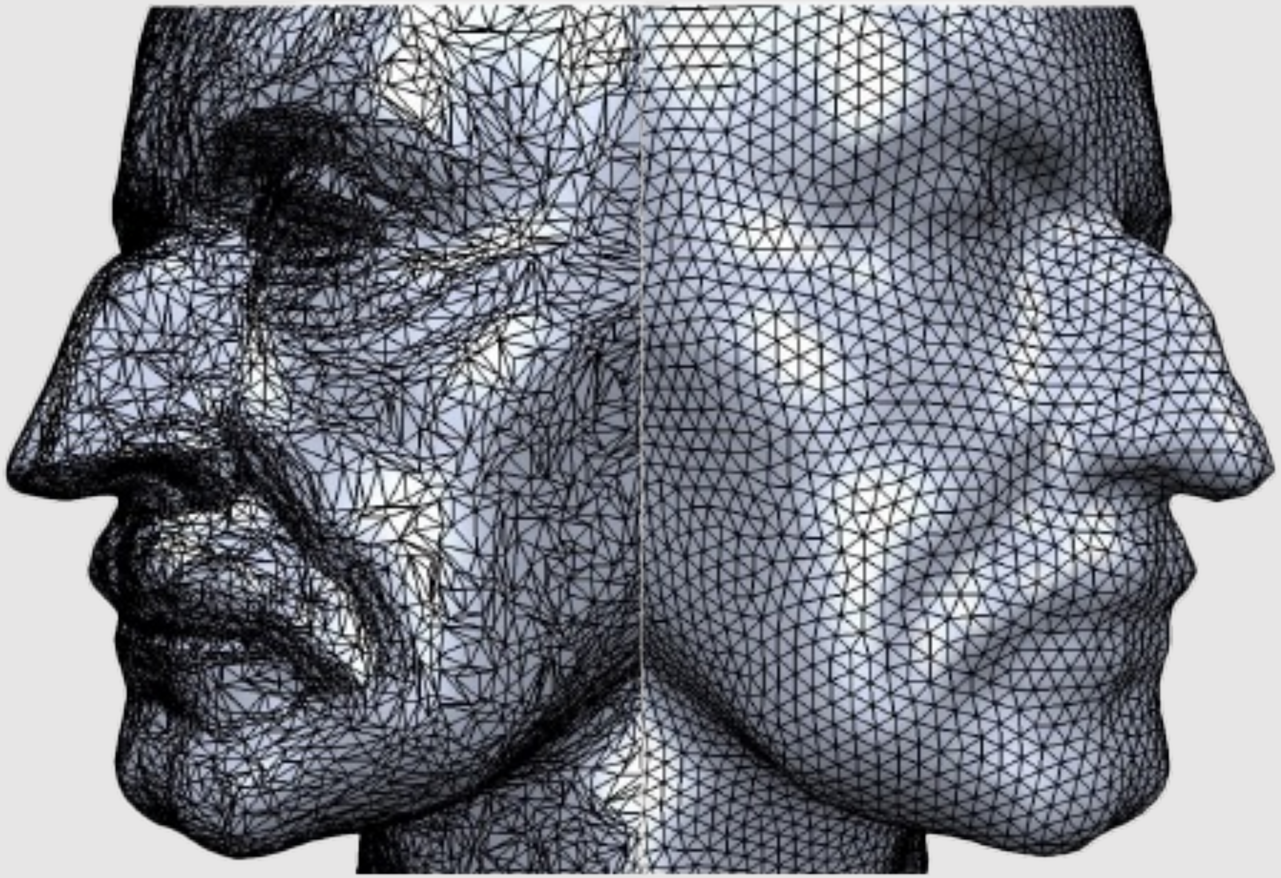
Repeatedly resampling a mesh also degrades signal quality!

- ~~Manifolds~~
- ~~Mesh representations and local operations~~
- Digital Geometric Processing
  - ~~Good Geometry~~
  - ~~Geometric Subdivision~~
  - ~~Geometric Simplification~~
  - Geometric Remeshing
  - Geometric Queries

Isotropic Remeshing

- **Isotropic:** same value when measured in any direction
- **Remeshing:** a change in the mesh
  - **Goal:** change the mesh to make triangles more uniform shape and size

- Helps achieve good mesh properties:
  - Good approximation of original shape
  - Vertex degrees close to 6
  - Angles close to 60deg
  - Delaunay triangles

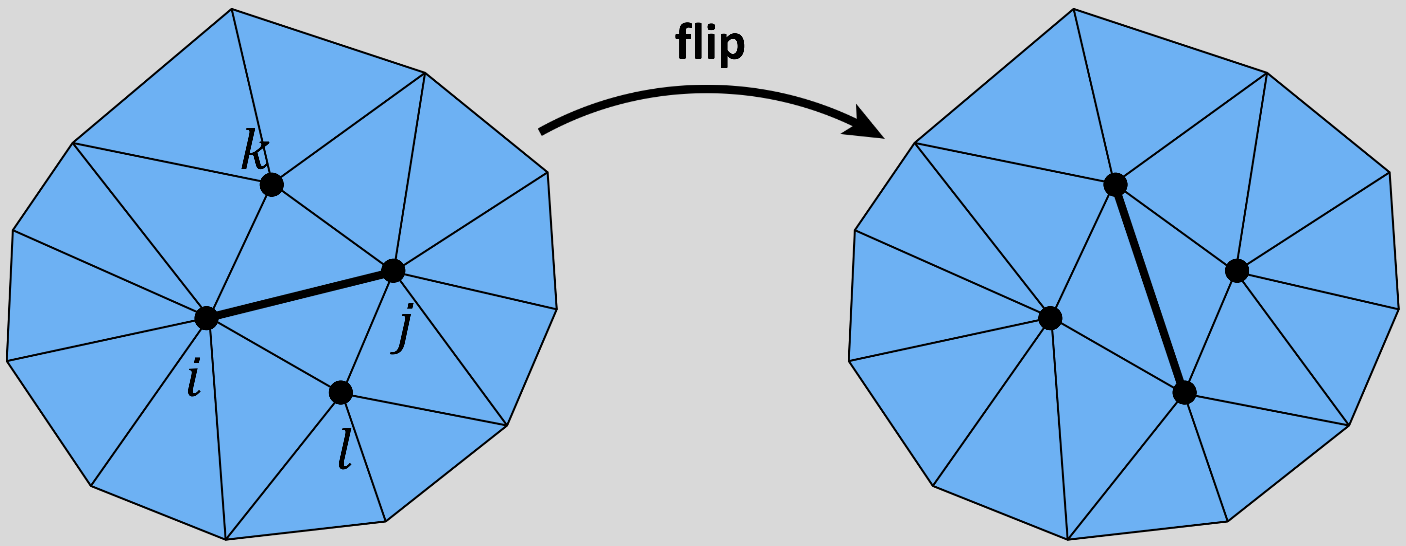


Improving Degree

Vertices with degree 6 makes triangles more regular

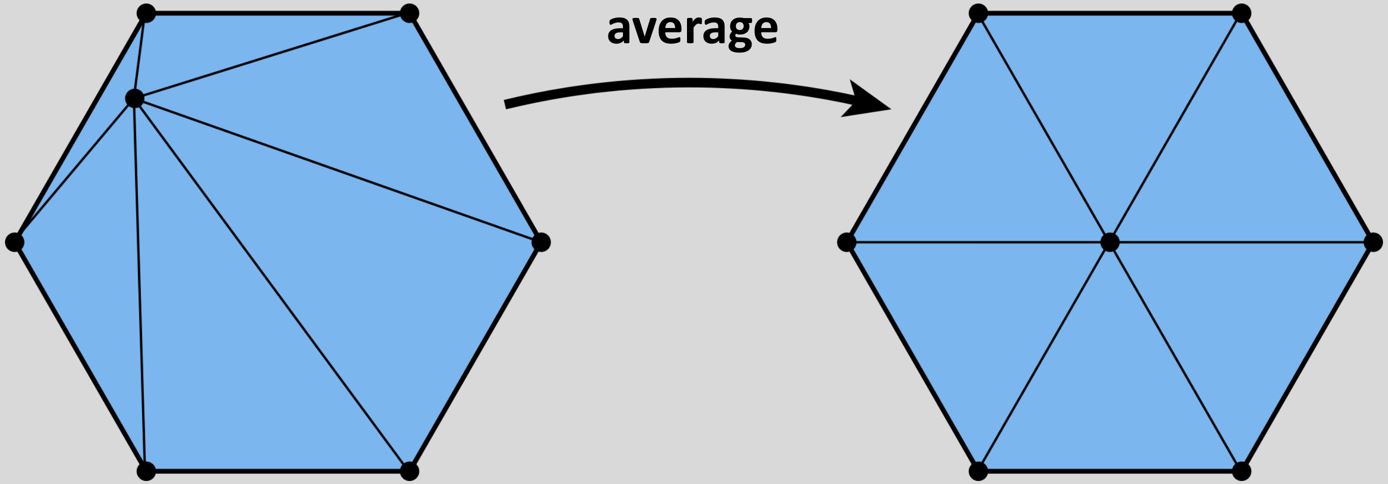
Deviation function: $|d_i - 6| + |d_j - 6| + |d_k - 6| + |d_l - 6|$

If flipping an edge reduces deviation function, flip edge



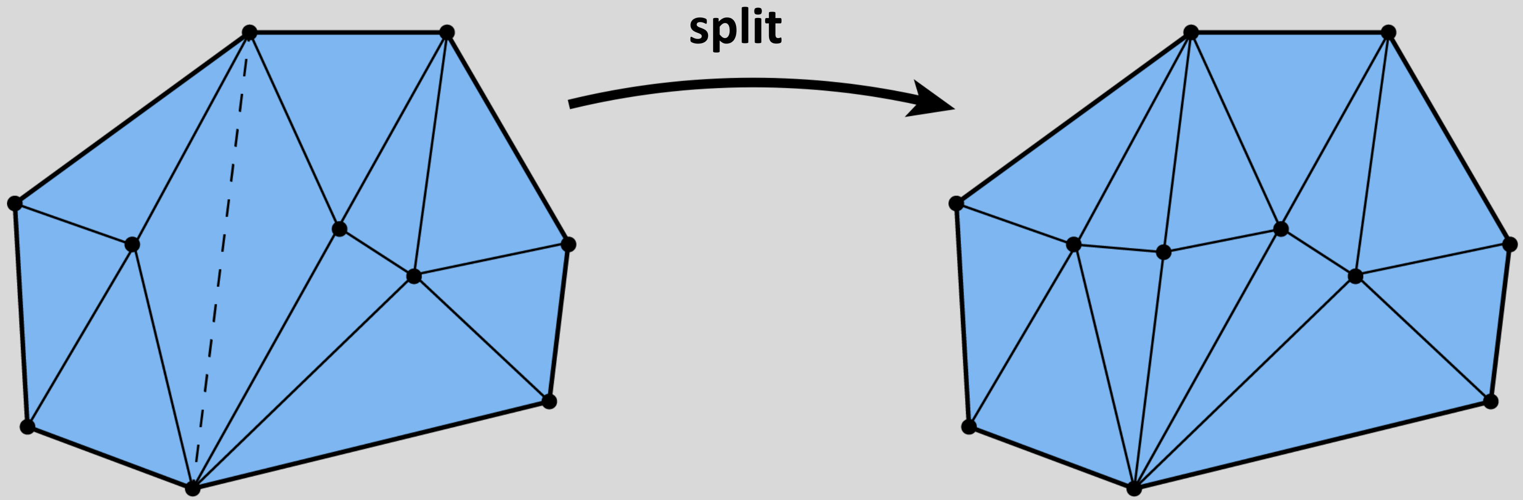
Improving Vertex Positioning

Center vertices to make triangles more even in size



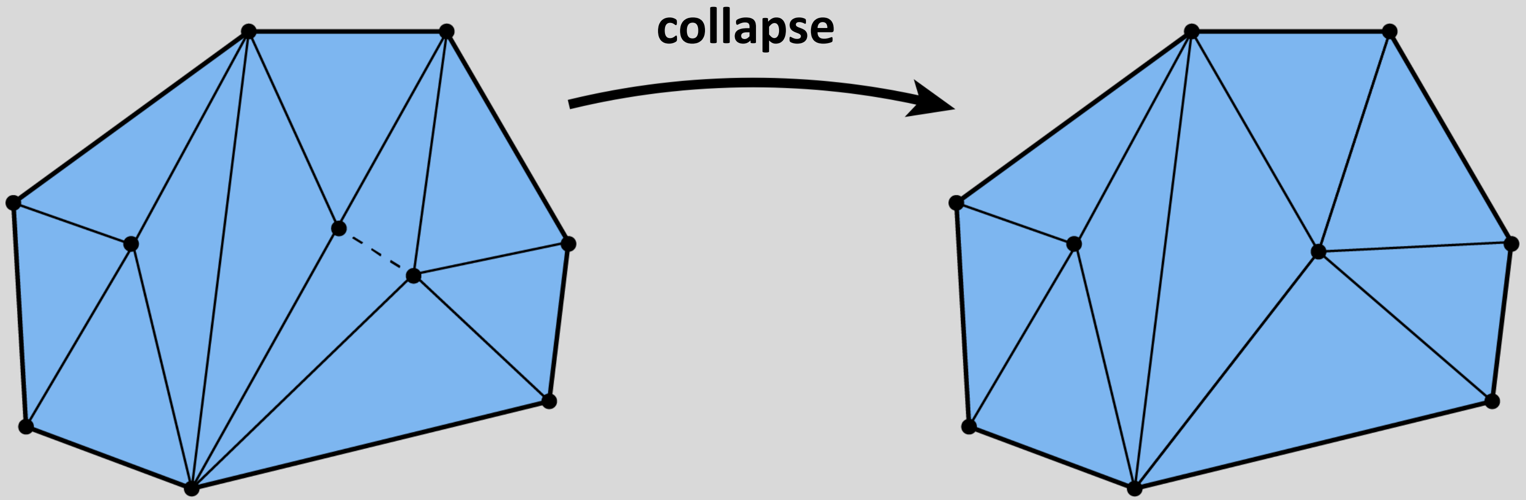
Improving Edge Length

If an edge is longer than (4/3 * mean) length, split it

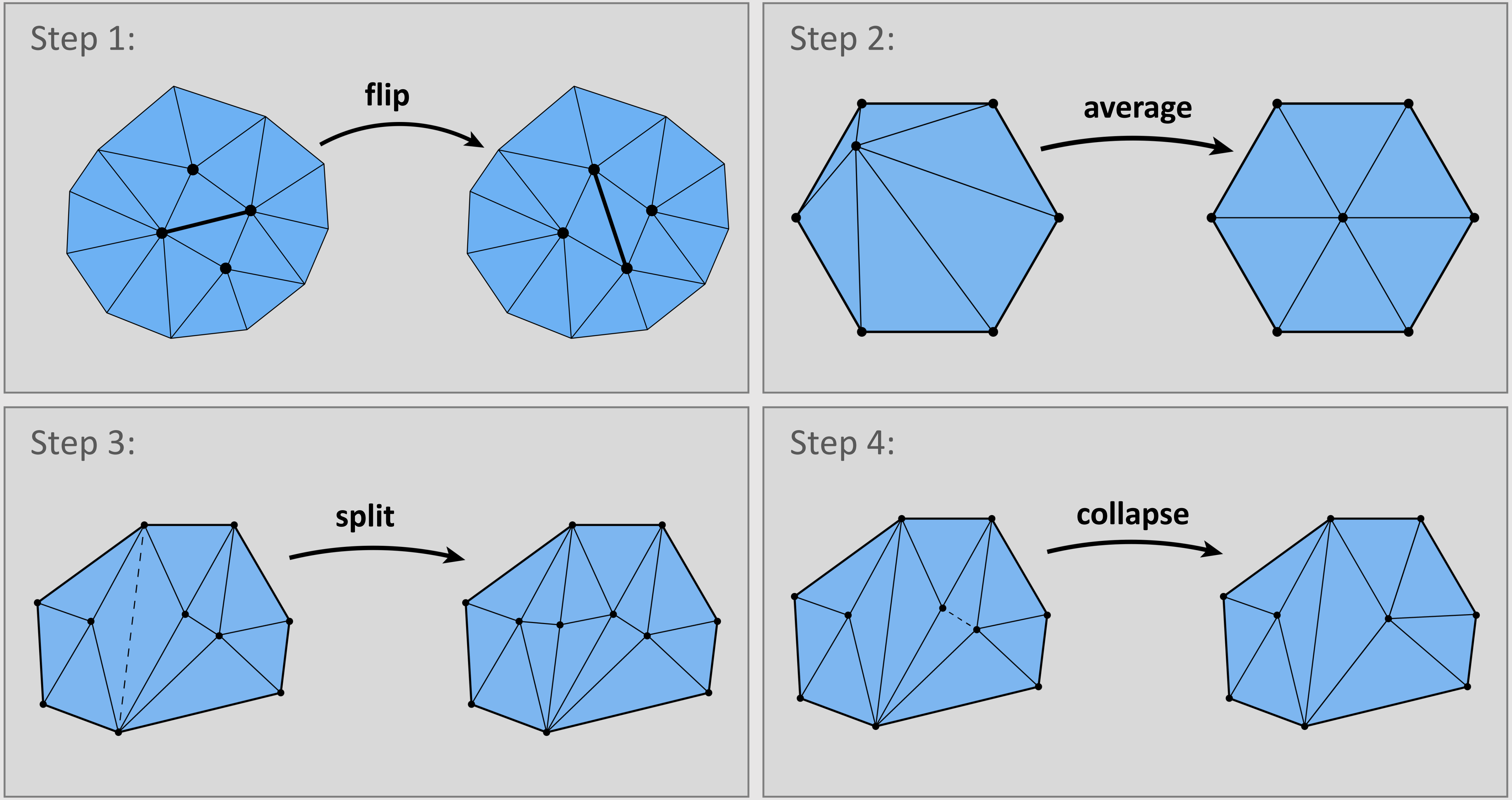


Improving Edge Length

If an edge is shorter than (4/5 * mean) length, collapse it



Isotropic Remeshing



- ~~Manifolds~~

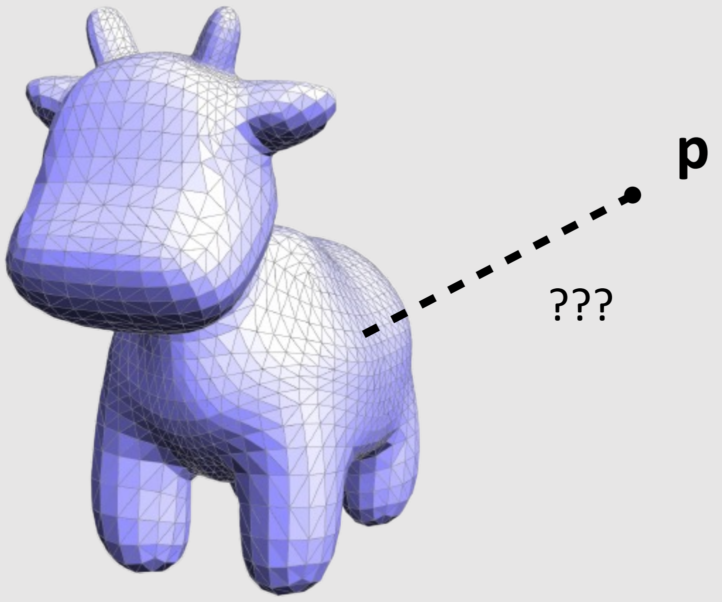
- ~~Mesh representations and local operations~~

- Digital Geometric Processing
    - ~~Good Geometry~~
    - ~~Geometric Subdivision~~
    - ~~Geometric Simplification~~
    - ~~Geometric Remeshing~~
    - Geometric Queries

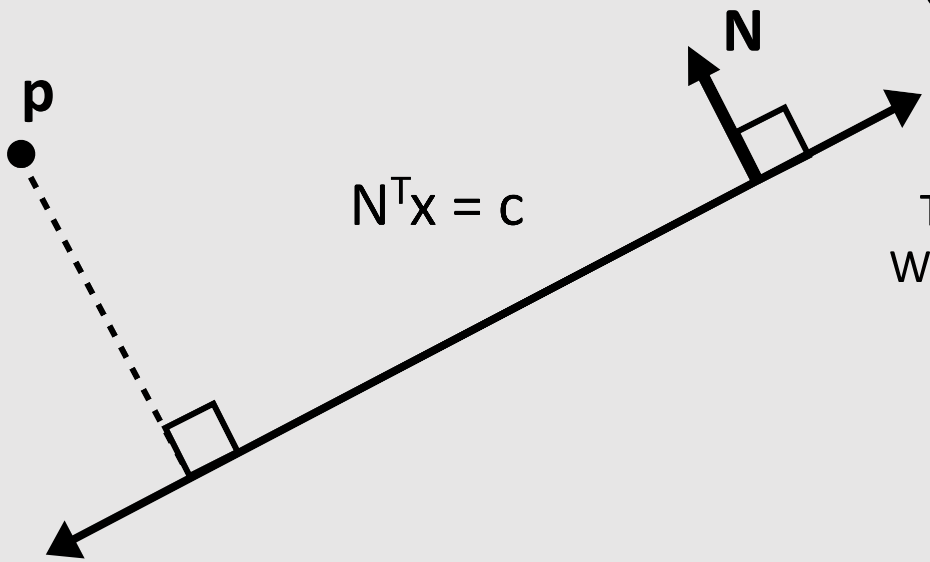
Closest Point Queries

- **Problem:** given a point, in how do we find the closest point on a given surface?

- Several use cases:
    - Ray/mesh intersection in pathtracing
    - Kinematics/animation
    - GUI/user selection
        - When I click on a mesh, what point am I actually clicking on?



Closest Point on a Line



To find the closest point to **p** along $N^T x = c$
We can have **p** travel along **N** for some time t

$$N^T(p + tN) = c$$

Multiplying the terms out

$$N^T p + tN^T N = c$$

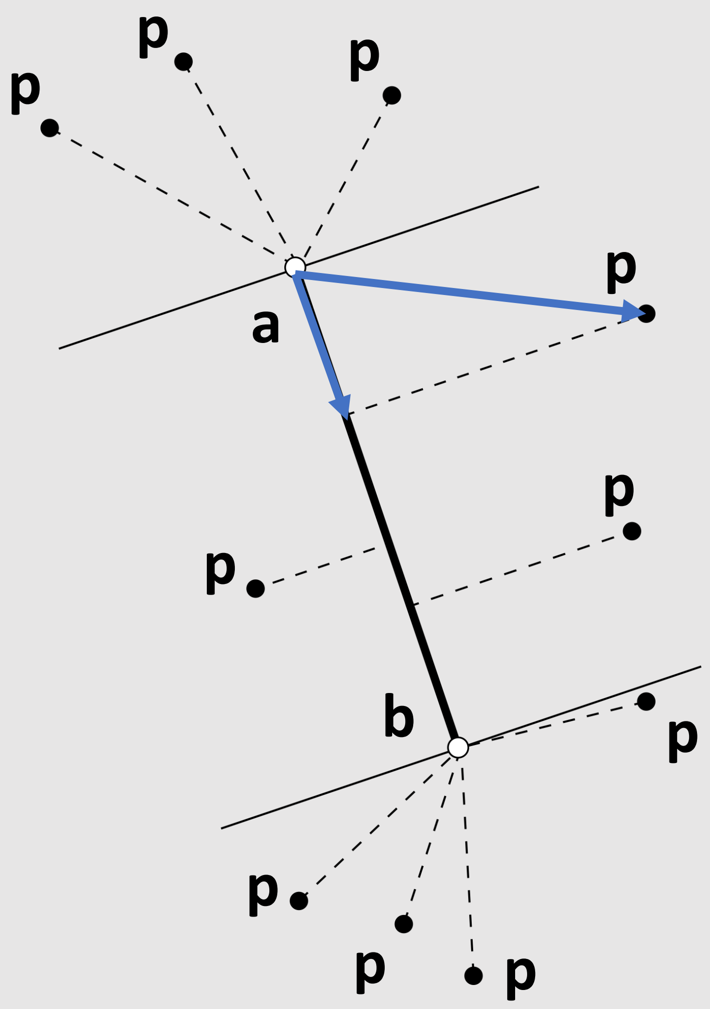
The unit norm multiplied by itself is 1
Solve for t

$$t = c - N^T p$$

Propagate **p** along **N** for time t

$$p + tN$$
$$p + (c - N^T p)N$$

Closest Point on a Line Segment



Compute the vector **p** from the line base **a** along the line

$$\langle \mathbf{p} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle$$

Normalize to get a time

$$t = \frac{\langle \mathbf{p} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle}{\langle \mathbf{b} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle}$$

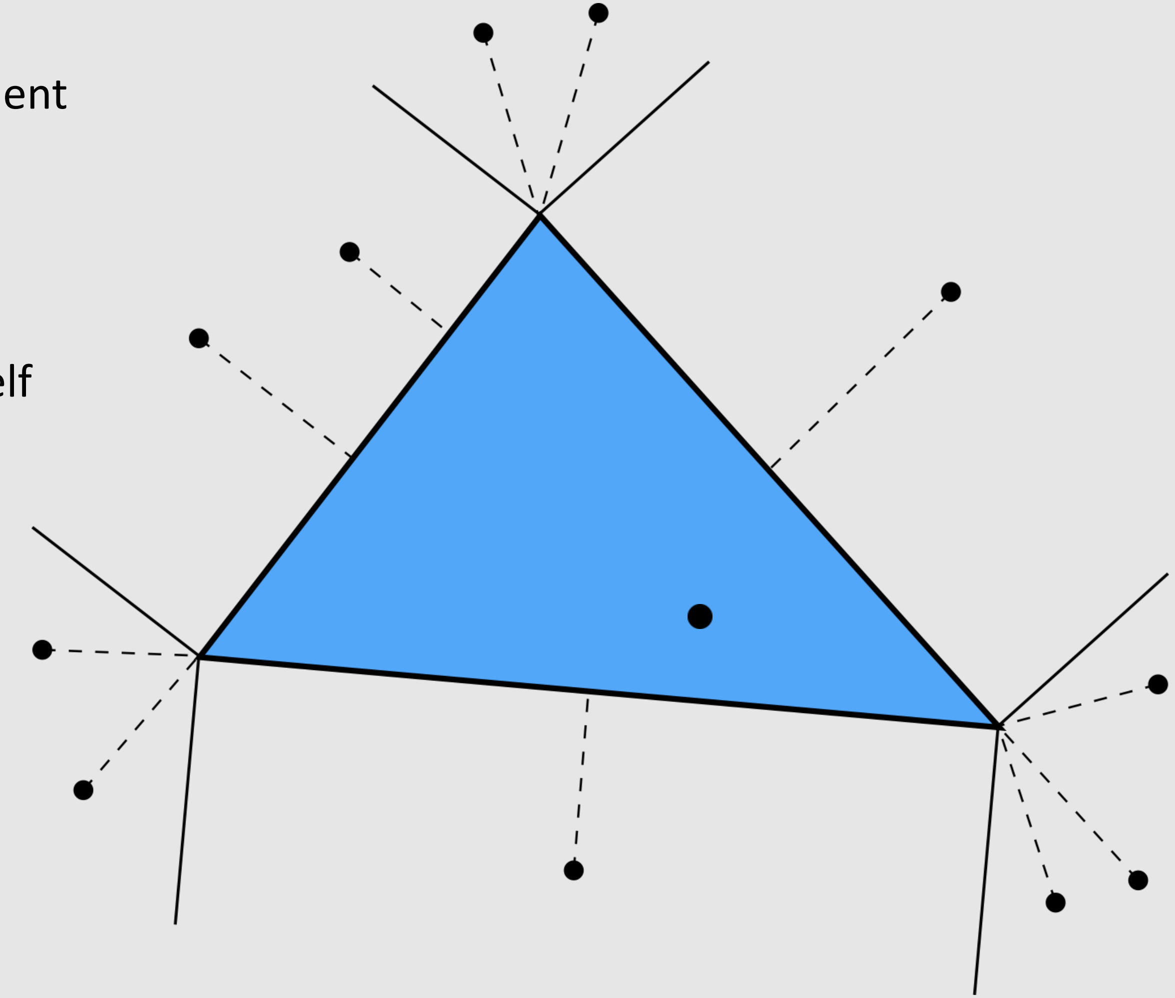
Clip time to range [0,1]and interpolate

$$\boldsymbol{a} + (\mathbf{b} - \mathbf{a})t$$

Closest Point on a 2D Triangle

- Easy! Just compute closest point to each line segment
  - For each point, compute distance
  - Point with smallest distance wins

- What if the point is inside the triangle?
  - Even easier! The closest point is the point itself
  - Recall point-in-triangle tests



Closest Point on a 3D Triangle

- **Method #1:** Projection**
  - Construct a plane that passes through the triangle
    - Can be done using cross product of edges
  - Project the point to the closest point on the plane
    - Same expression as with a line: $p + (c - N^T p)N$
    - Check if point is in triangle using half-plane test
  - Else, compute distance from each line segment in 3D
    - Same expression as with a 2D line segment

- **Method #2:** Rotation**
  - Translate point + triangle so that triangle vertex v1 is at the origin
  - Rotate point + triangle so that triangle vertex v2 sits on the z-axis
  - Rotate point + triangle so that triangle vertex v3 sits on the yz-axis
  - Disregard x-coordinate of point
    - Problem reduces to closest point on 2D triangle

**https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.4264&rep=rep1&type=pdf

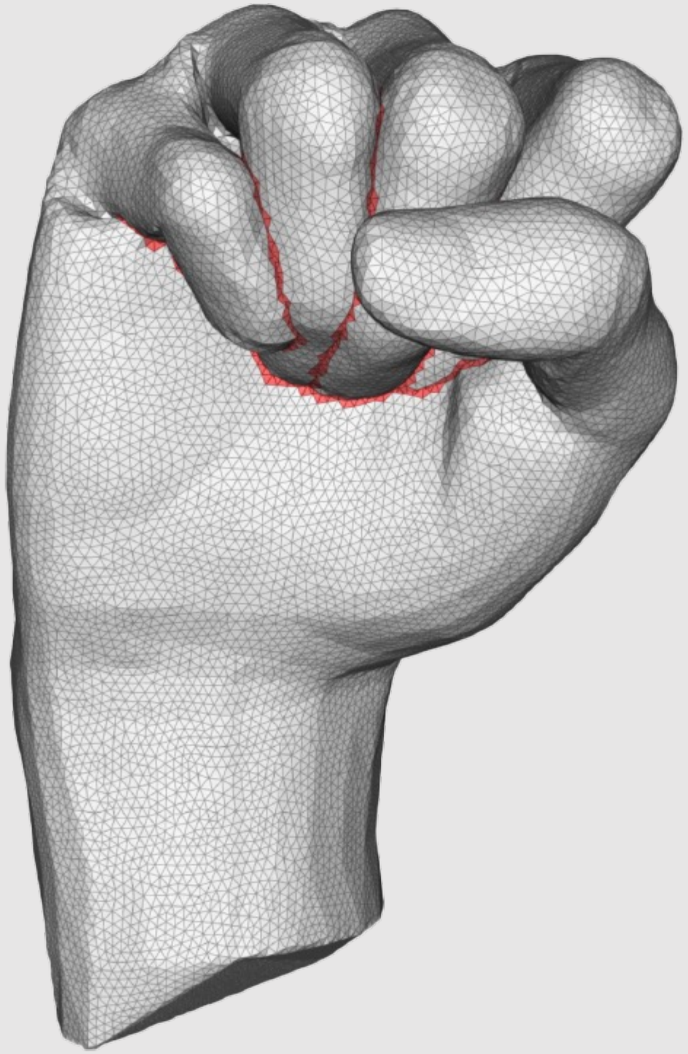
Closest Point on a 3D Triangle Mesh

- Conceptually easy!
  - Loop over every triangle
  - Compute closest point to current triangle
  - Keep track of globally closest point
- Not practical in real world
  - Meshes have billions of triangles
  - Programs make thousands of geometric queries a second
- Will look at better solutions next time

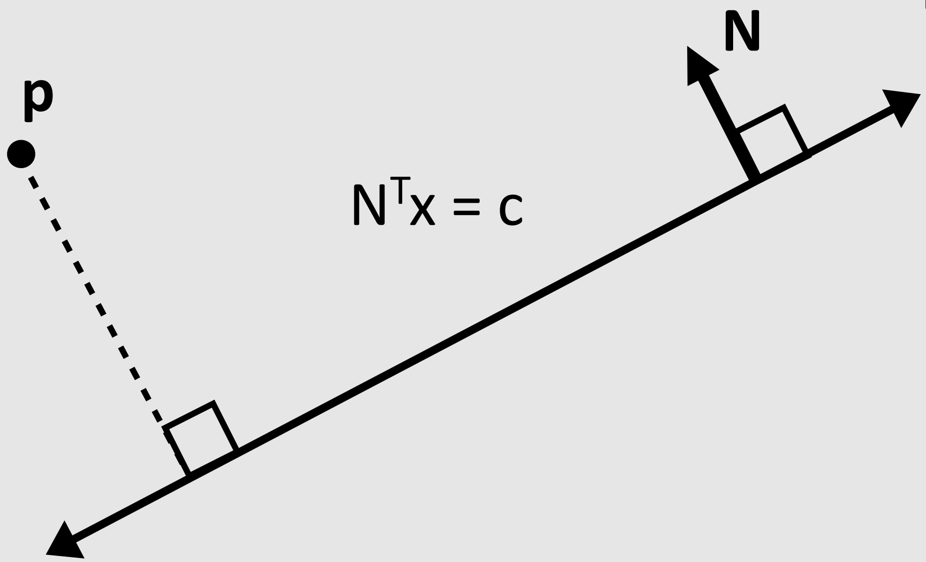
Mesh-Mesh Intersections

- Sometimes when editing geometry, a mesh will intersect with itself
- Likewise, sometimes when animating geometry, meshes will collide

- How do we check for/prevent collisions?



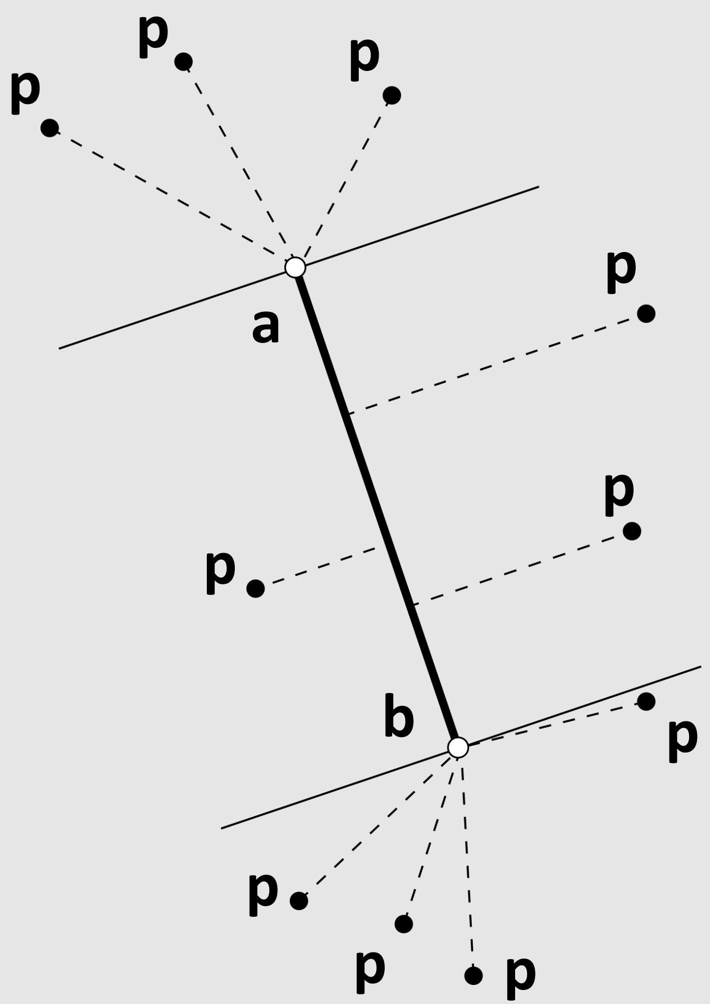
Point-Line Intersection



Just plug point in

$$N^{T}p = c?$$

Point-Line Segment Intersection



Check if adding distances equals net distance**

$$dist(a,p) + dist(p,b) = dist(a,b)$$

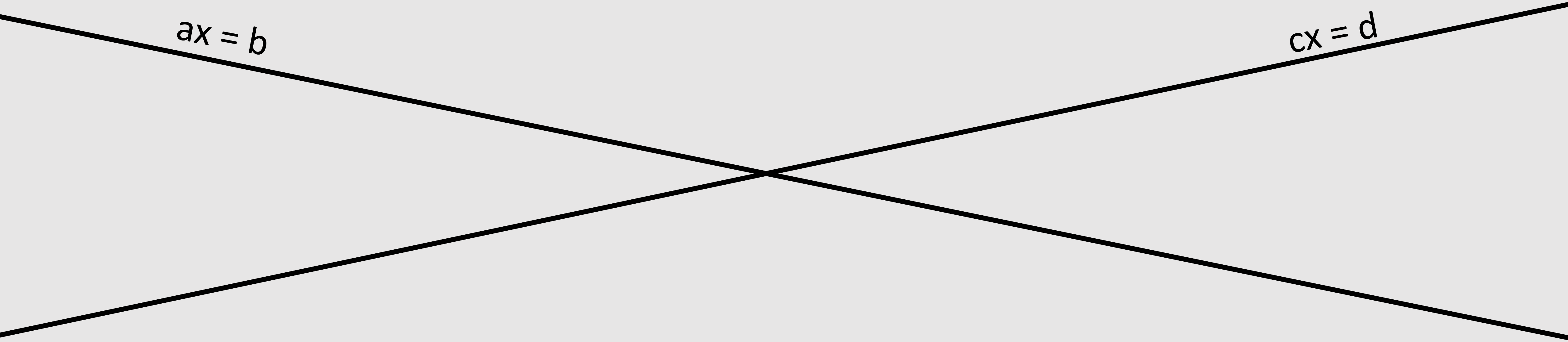
**Potential numeric stability issues

Line-Line Intersection

Two equations, two unknowns

Solve a linear system

$$\begin{bmatrix} a_1 & a_2 \\ c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$$



Point-Triangle Intersection

You know this :)