Introduction To Geometry

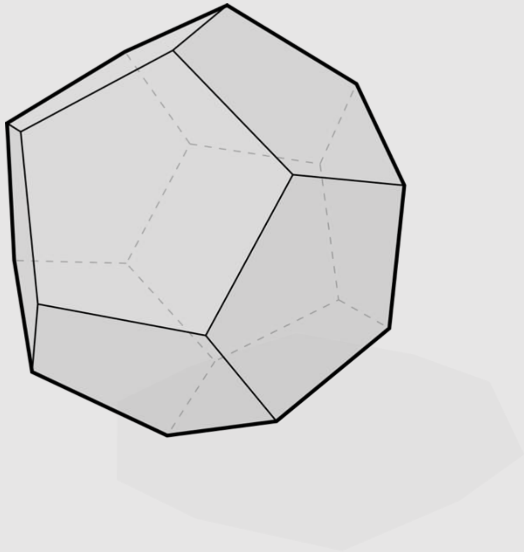
- Implicit & Explicit Geometry
- Manifold Geometry
- Local Geometric Operations

What Is Geometry?

"Earth" "measure"

g e • o m • e t • r y /jēˈämətrē/ *n.*

1. The study of shapes, sizes, patterns, and positions.
2. The study of spaces where some quantity (lengths, angles, etc.) can be *measured*.



Creating and processing geometry are key components of Computer Graphics.
First of all, we need to digitally represent the geometry!

How To Represent Geometry

[LINGUISTIC]

“unit circle”

[IMPLICIT]

$$x^2+y^2=1$$

[EXPLICIT]

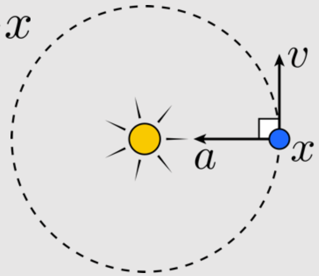
$$(\underbrace{\cos\theta}_{x}, \underbrace{\sin\theta}_{y})$$

[CURVATURE]

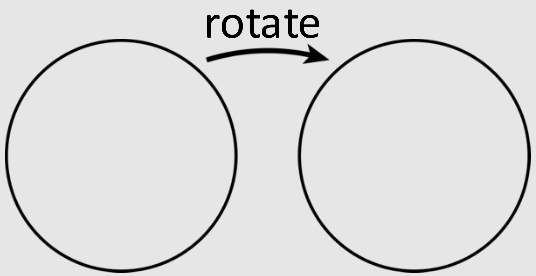
$$\kappa = 1$$

[DYNAMIC]

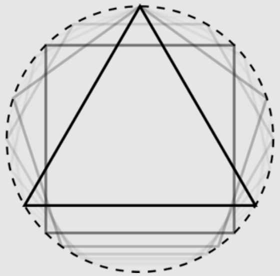
$$\frac{d^2}{dt^2}x = -x$$



[SYMMETRIC]



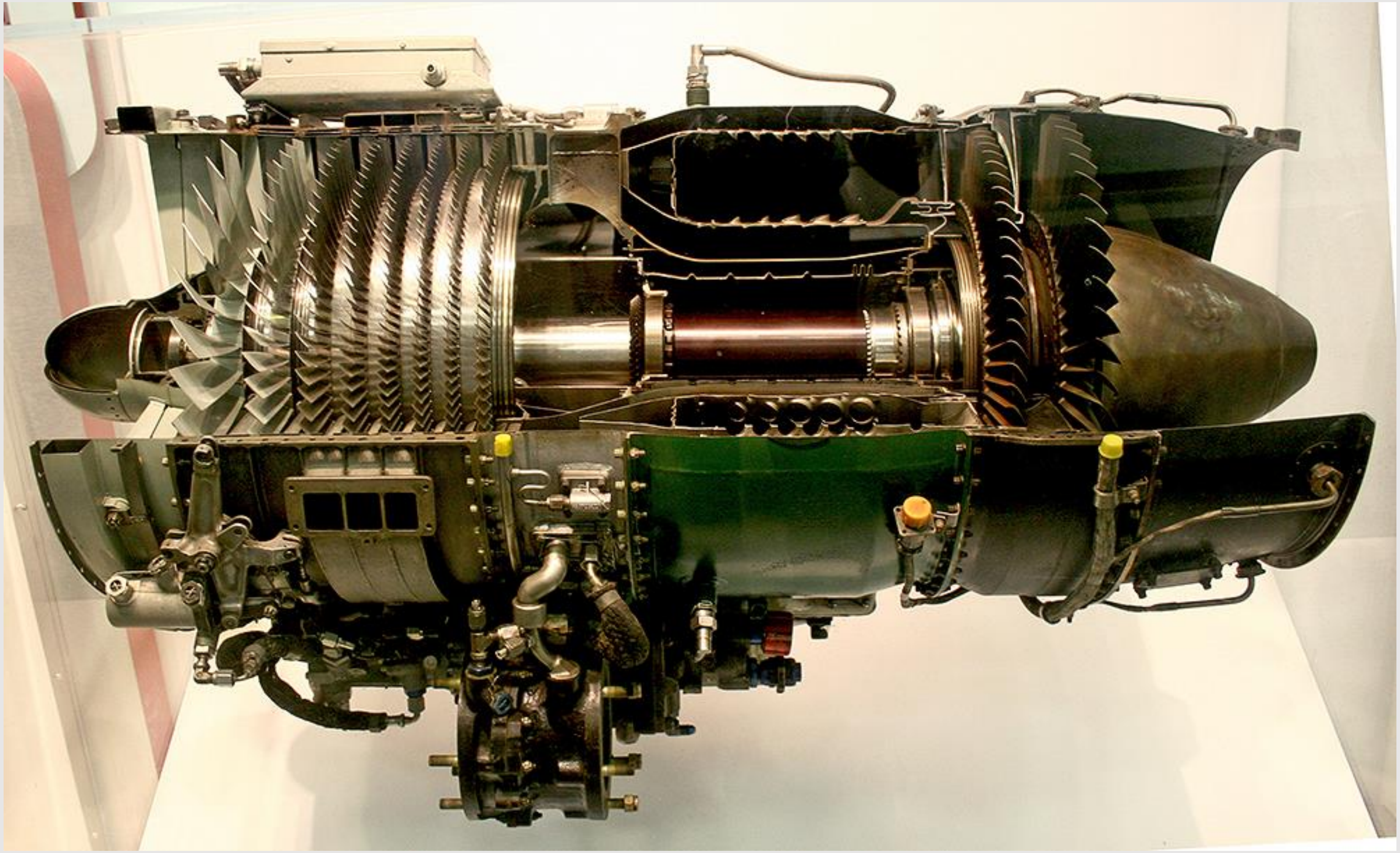
[DISCRETE]



n ➞ ∞

How To Represent Humans

How To Represent Machines



How To Represent Cloth

How To Represent Water

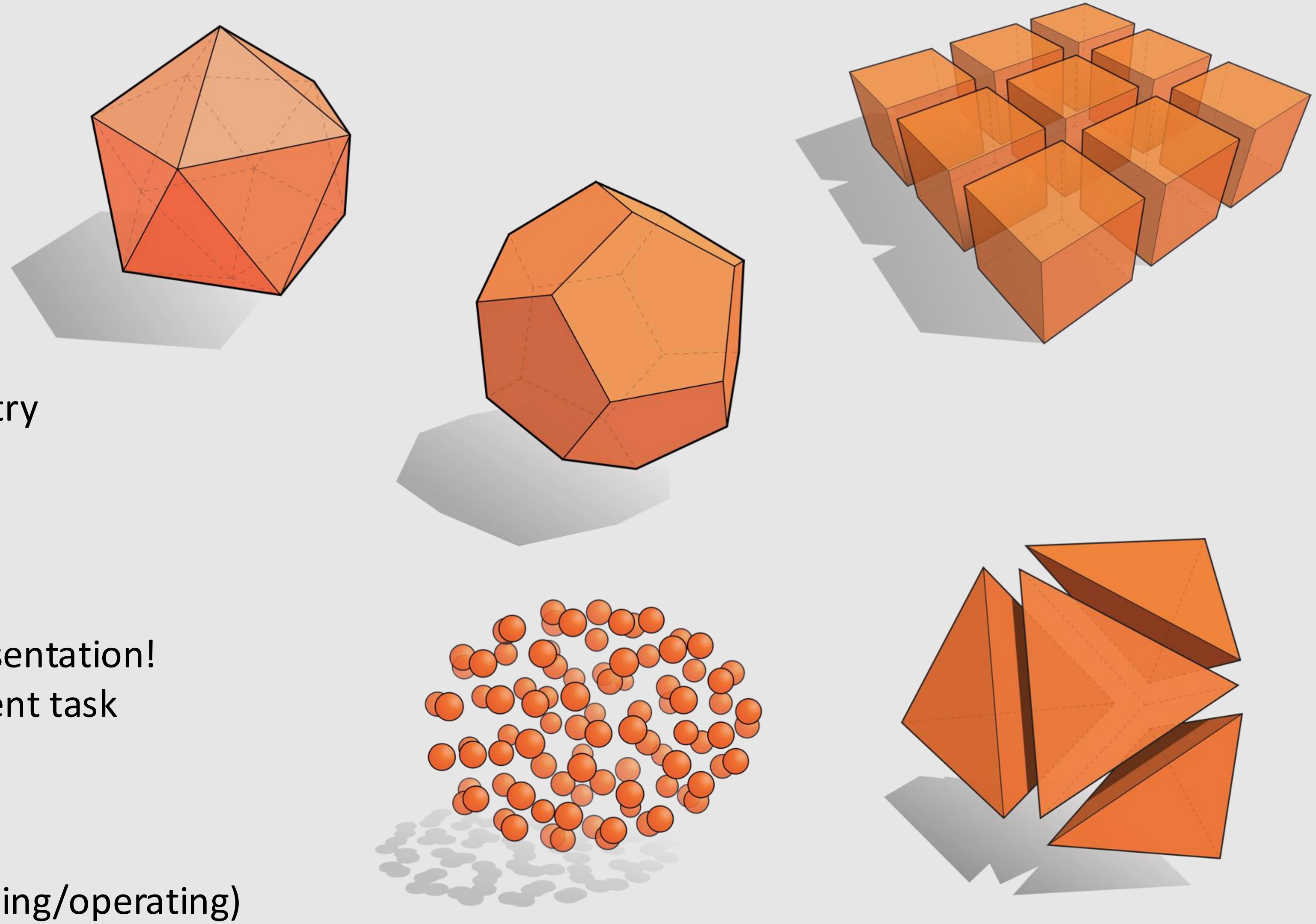
How To Represent This Thing

Many Ways To Encode Geometry

- Explicit:
  - point cloud
  - polygon meshes
  - subdivision surfaces
  - NURBS

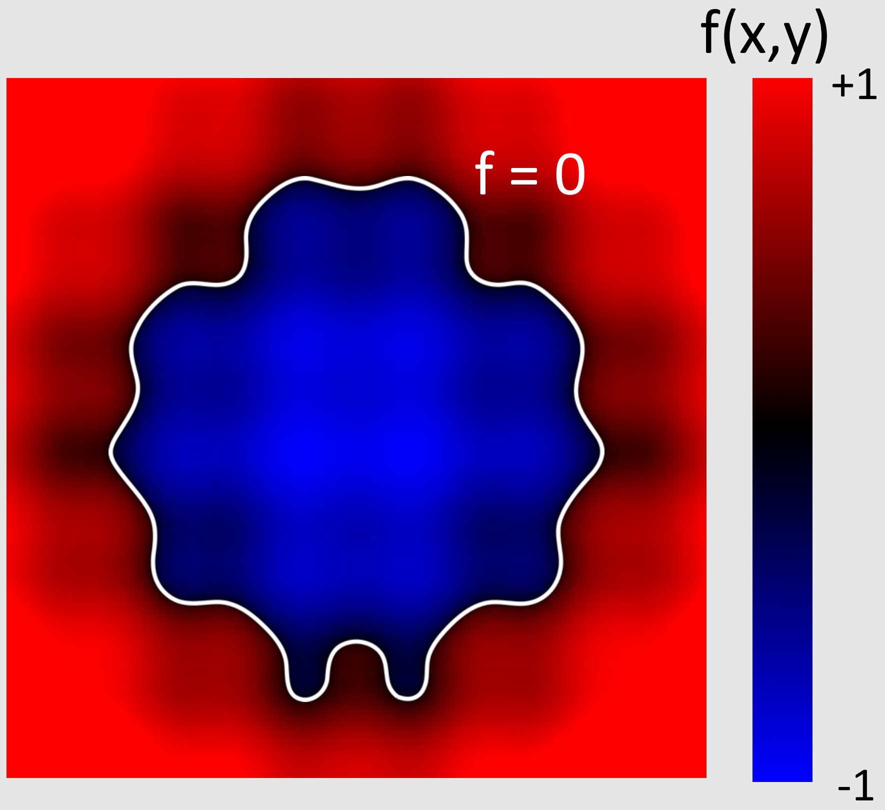
- Implicit:
  - level set
  - constructive solid geometry
  - algebraic surface
  - L-systems
  - Fractals

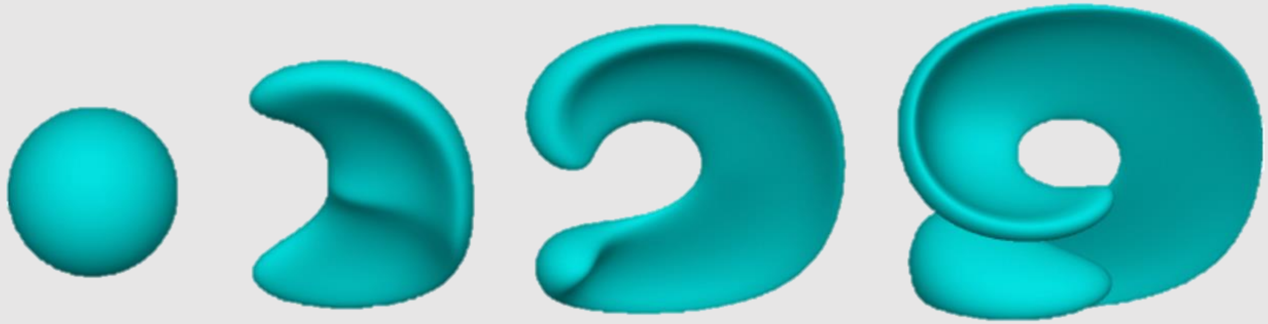
- Not one best geometric representation!
  - Each is suited for a different task
  - Tradeoffs between:
    - Accuracy
    - Memory
    - Performance (searching/operating)



Implicit Geometry

- Points aren't known directly, but satisfy some relationship
  - Example: unit sphere is all points such that $x^2+y^2+z^2=1$
- More generally, in the form $f(x,y,z) = 0$
- Finding example points is **hard**
  - Requires solving equation
- Checking if points are inside/outside is **easy**
  - Just evaluate the function with a given point
    - E.g. (2, 2, 2) is not on the unit sphere



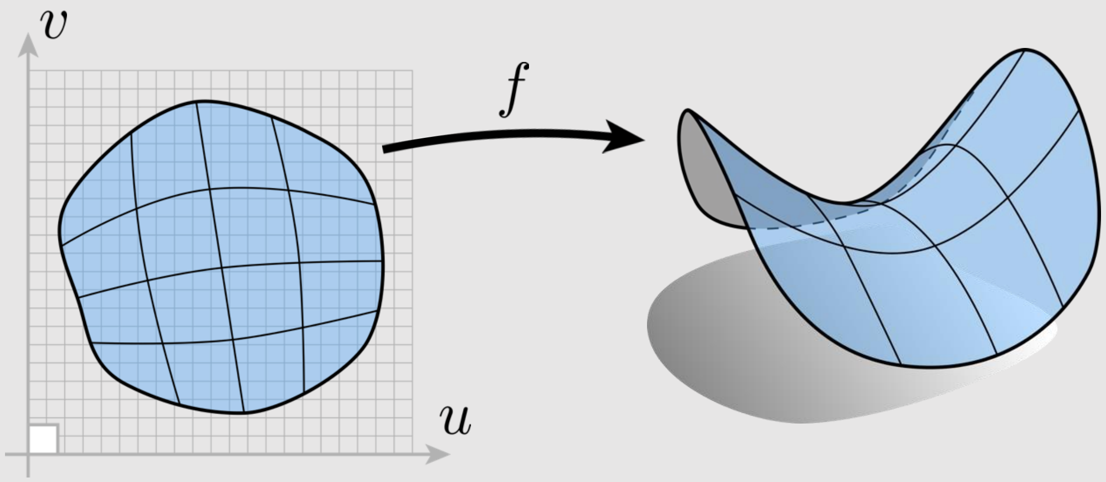


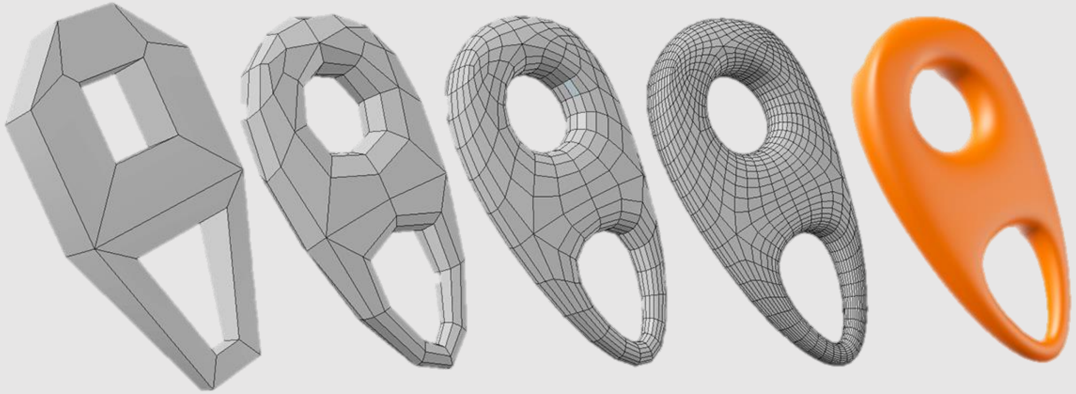
Explicit Geometry

- All points are given directly
- More generally:

$$f : \mathbb{R}^2 \to \mathbb{R}^3; (u, v) \mapsto (x, y, z)$$

  - Given any (u, v), we can find a point on the surface
  - Can limit (u, v) to some range
    - **Example:** triangle with barycentric coordinates
- Finding example points is **easy**
  - We are given them for free
- Checking if points are inside/outside is **hard**
  - We are given the output values and need to find input values that satisfy the geometry





More concrete examples on **easy** and **hard:**

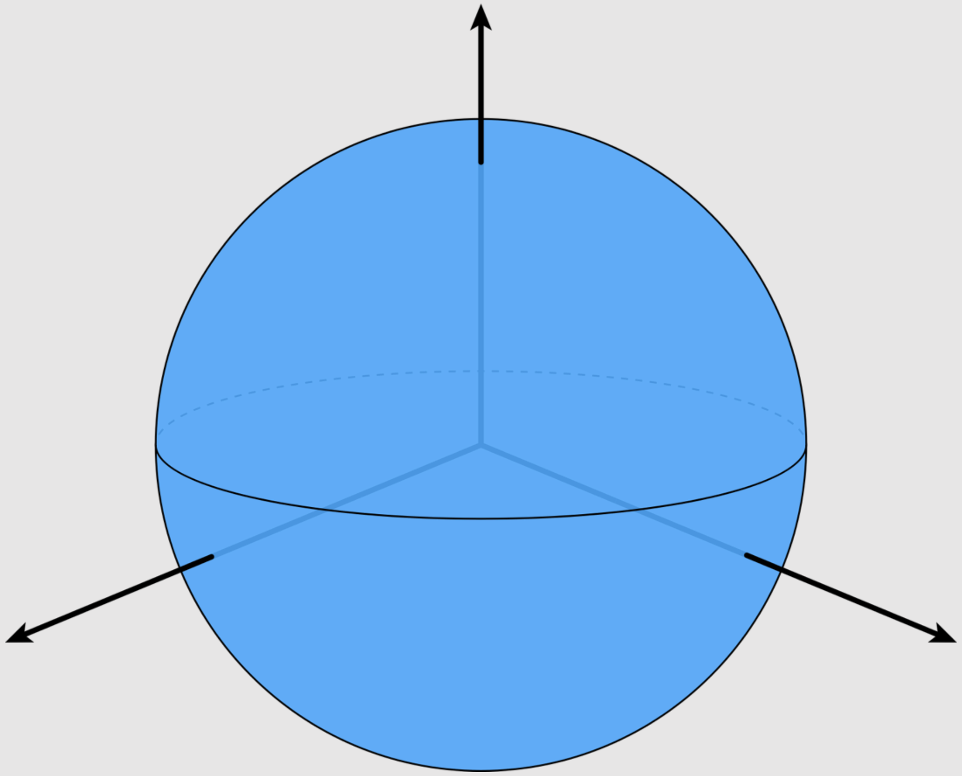
Implicit Geometry [Hard]

- Given the unit sphere:

$$f(x, y, z) = x^2 + y^2 + z^2 = 1$$

- Find a point that exists on it.

- **Answer:** (1,0,0)
  - Not so difficult, but how did you arrive at the answer?
  - We are given an equation, and need to find coordinates (x, y, z) that satisfy the equation
    - Keep guessing and checking



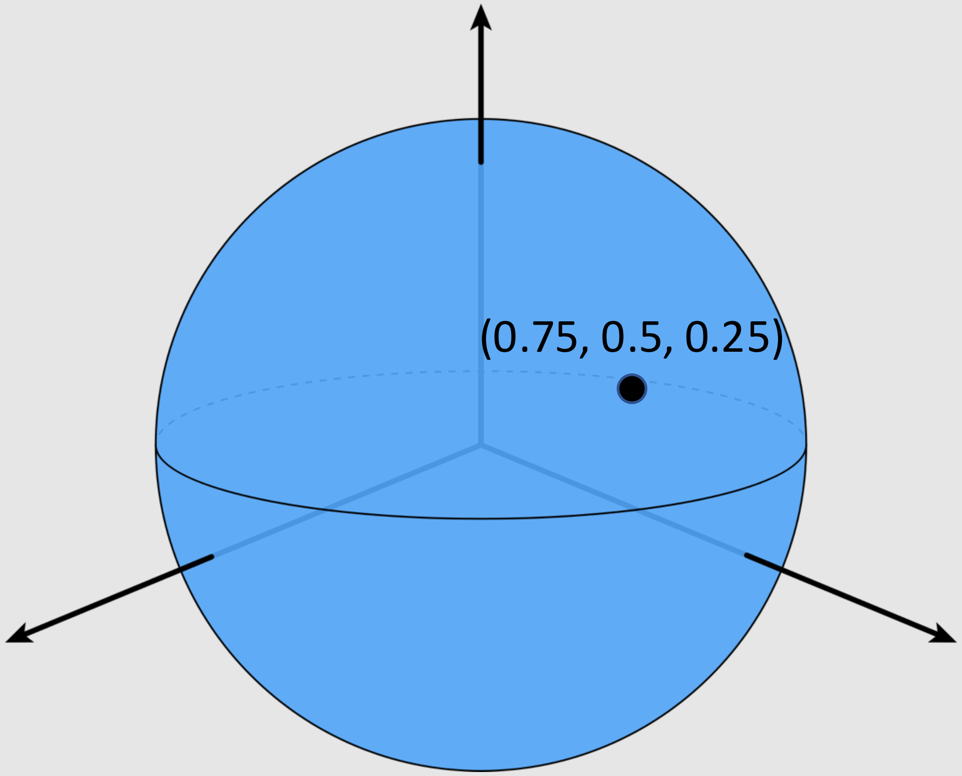
Implicit Geometry [Easy]

- Given the unit sphere:

$$f(x, y, z) = x^2 + y^2 + z^2 = 1$$

- Find if the point (0.75, 0.5, 0.25) lives inside it.

- **Answer:** yes!
  - $f(0.75, 0.5, 0.25) = 0.75^2 + 0.5^2 + 0.25^2 = 0.875 < 1$
  - Easy to check! Just evaluate the sign of the function at the desired point



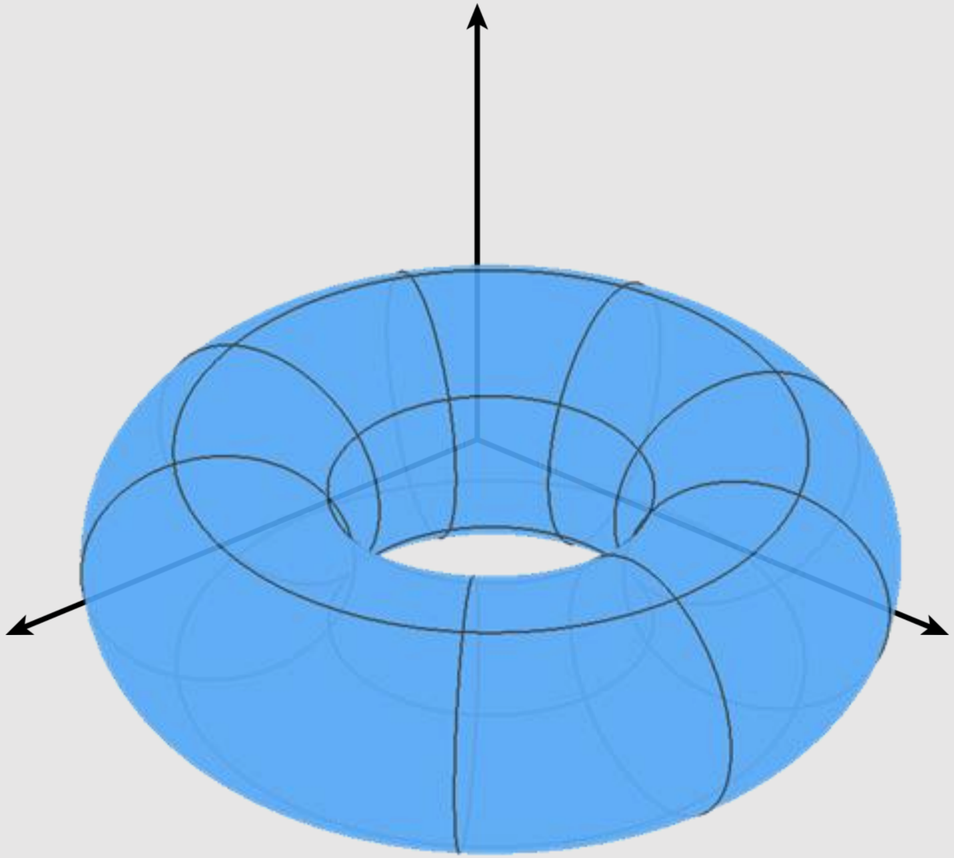
Explicit Geometry [Easy]

- Given the torus:

$$f(u, v) = ((2 + \cos u) \cos v \, , (2 + \cos u) \sin v \, , \sin u)$$

- Find a point that exists on it.

- **Answer:** (3,0,0)
  - Just plug in any value of (u, v)!
    - We plugged in $(u, v) = (0{,}0)$



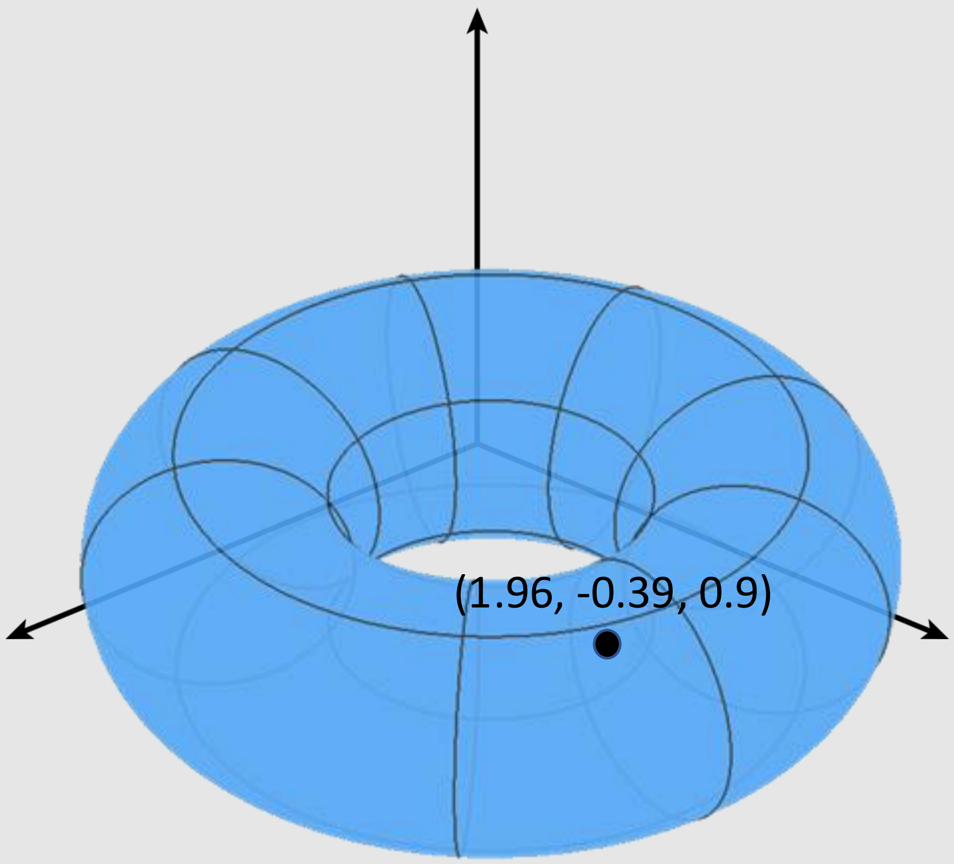
Explicit Geometry [Hard]

- Given the torus:

$$f(u, v) = ((2 + \cos u)\cos v, (2 + \cos u)\sin v, \sin u)$$

- Find if the point (1.96, -0.39, 0.9) lives inside it.

- **Answer:** no, I'm not computing that
  - Formally speaking, if any rays from the point towards infinity intersect with the surface an odd number of times, then the point is inside; otherwise, no.

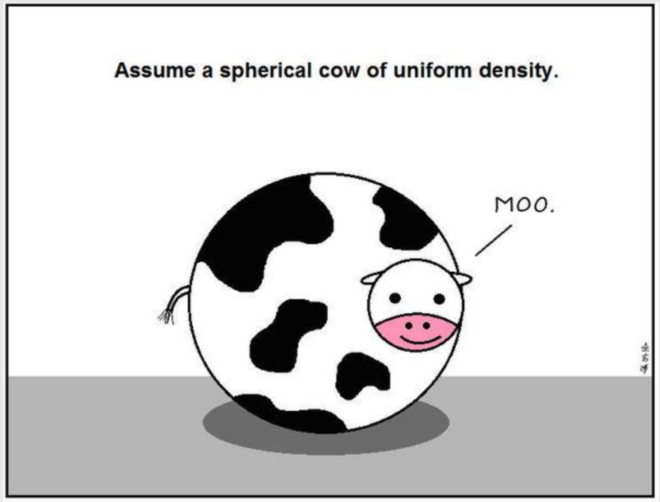


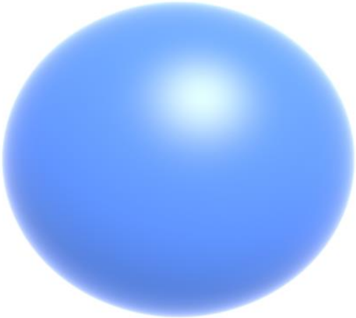
Let’s look at some implicit examples…

Algebraic Surfaces [Implicit]

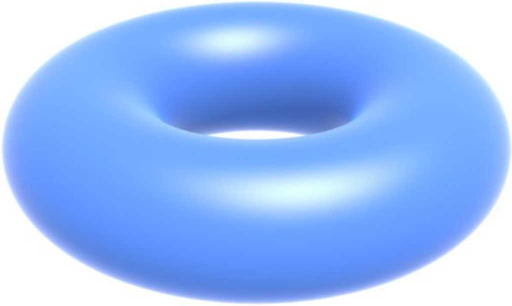
- A surface built with algebra
  - Generally thought of as a surface where points are some radius r away from another point/line/surface

- [+] Generates smooth/symmetric surfaces
- [-] Challenging to represent complex shapes





$$x^2+y^2+z^2=1$$

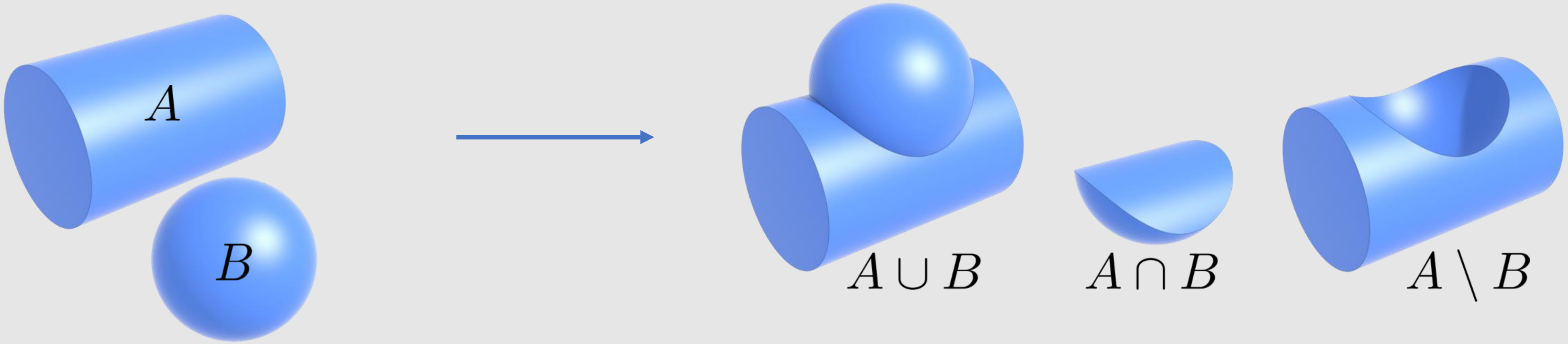


$$(R-\sqrt{x^2+y^2})^2+z^2=r^2$$

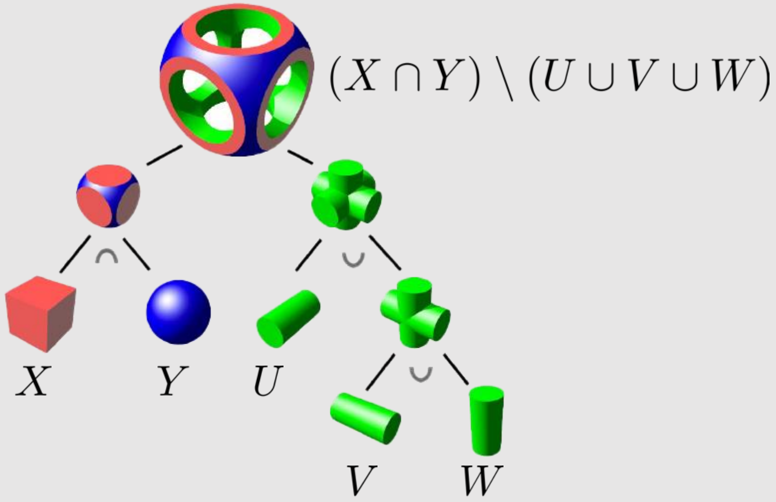
$$(x^2+\frac{9y^2}{4}+z^2-1)^3 = x^2z^3+\frac{9y^2z^3}{80}$$

Constructive Solid Geometry [Implicit]

- Build more complicated shapes via Boolean operations
  - Basic operations:

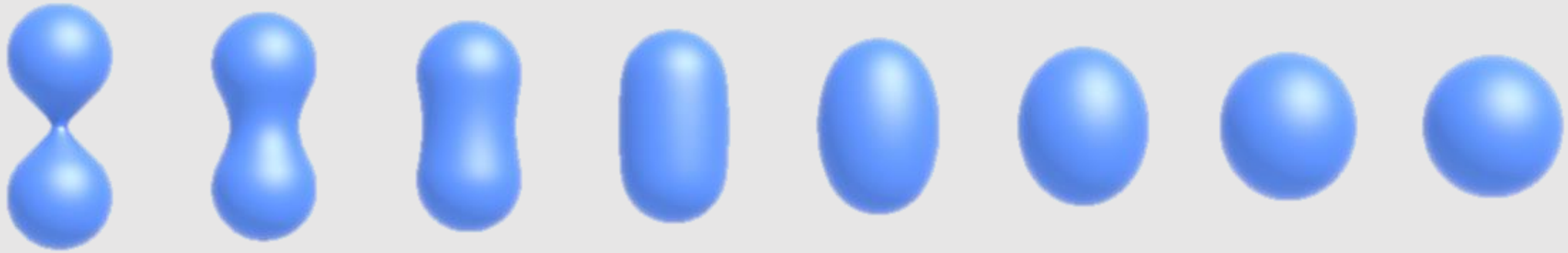


- Can be used to form complex shapes!



Blobby Surfaces [Implicit]

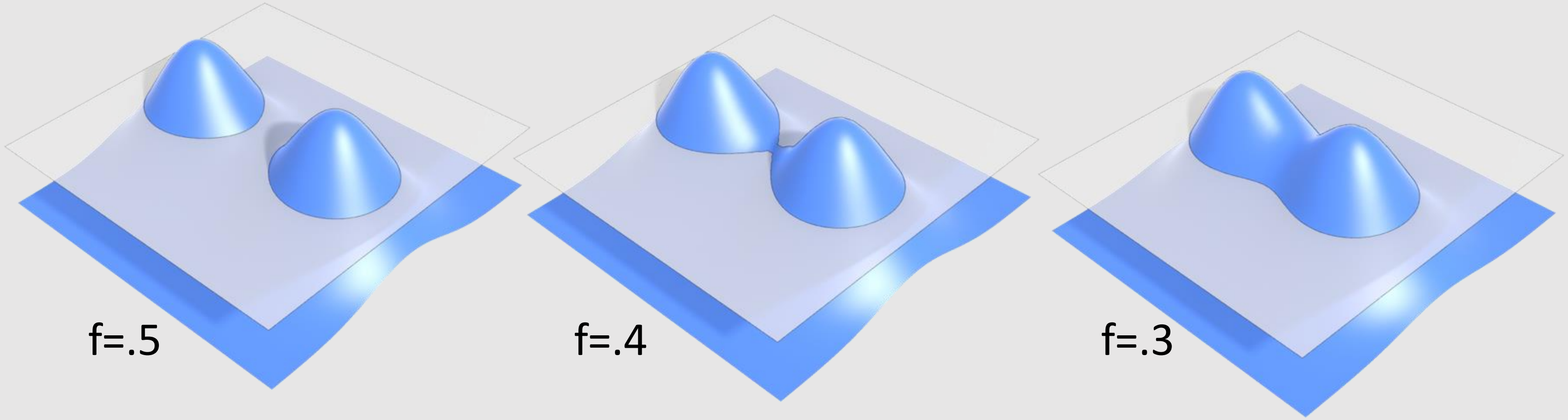
- Instead of Booleans, gradually blend surfaces together:



- Easier to understand in 2D:

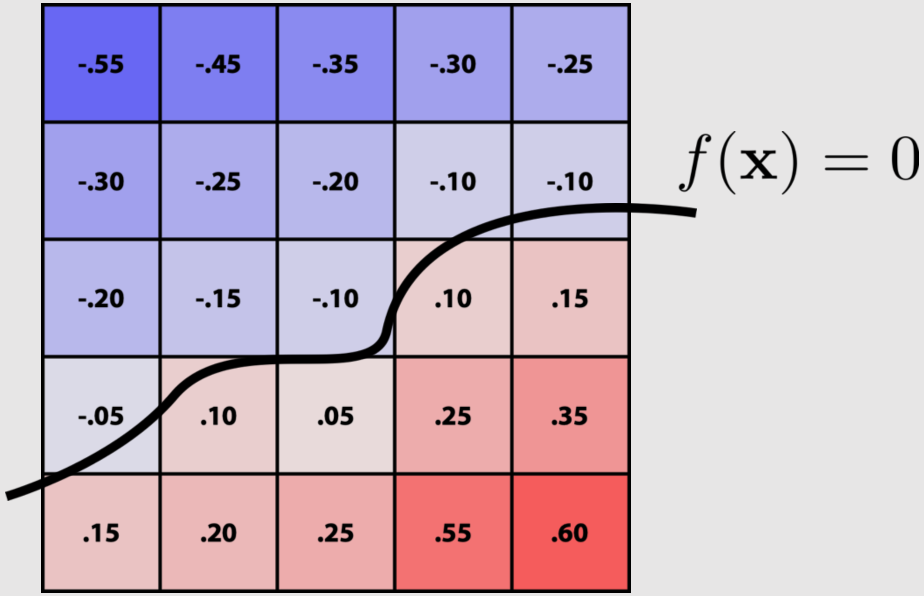
$$\phi_p(x) := e^{-|x-p|^2}$$ (Gaussian centered at p)

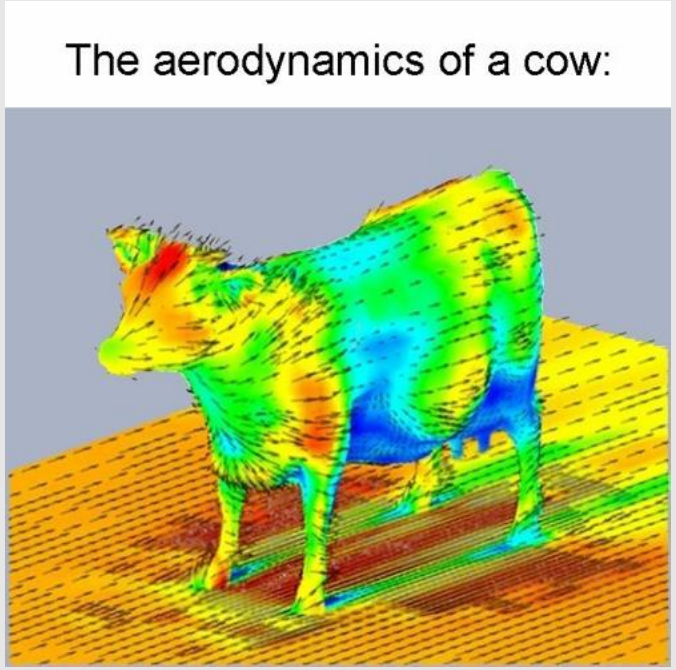
$$f := \phi_p + \phi_q$$ (Sum of Gaussians centered at different points)



Level Set Methods [Implicit]

- Store a grid of values approximating function





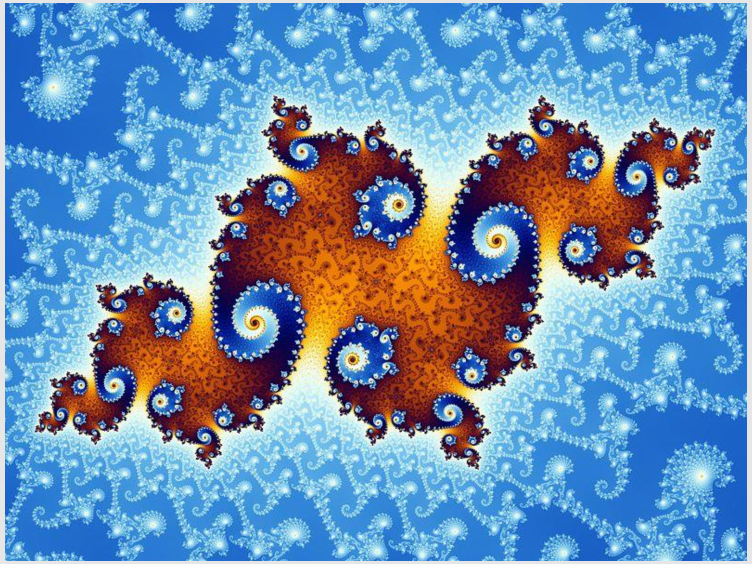
- Surface is found where interpolated values equal zero

- [+] Provides much more explicit control over shape
- [-] Runs into problems of <u>aliasing!</u>
- [-] storage can be expensive! -> can alleviate this issue by only storing a narrow band of distances near the surface

Fractals [Implicit]

- Defined recursively; exhibit self-similarity, detail at all scales

- [+] New “language” for describing natural phenomena
- [-] Hard to control shape!

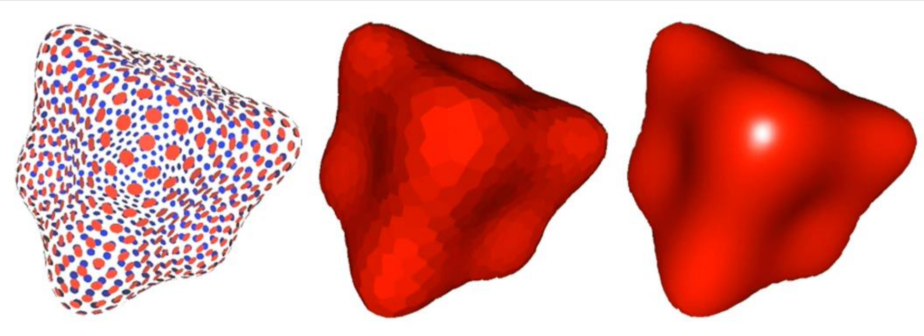


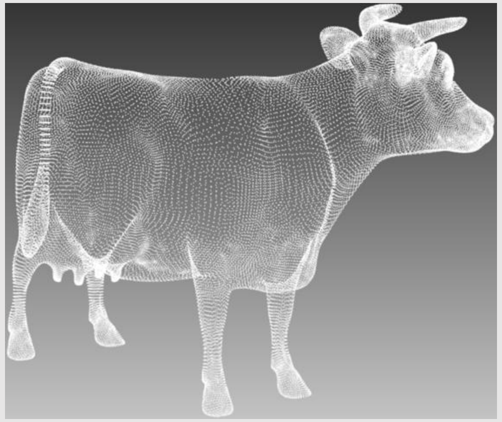
Let’s look at some explicit examples…

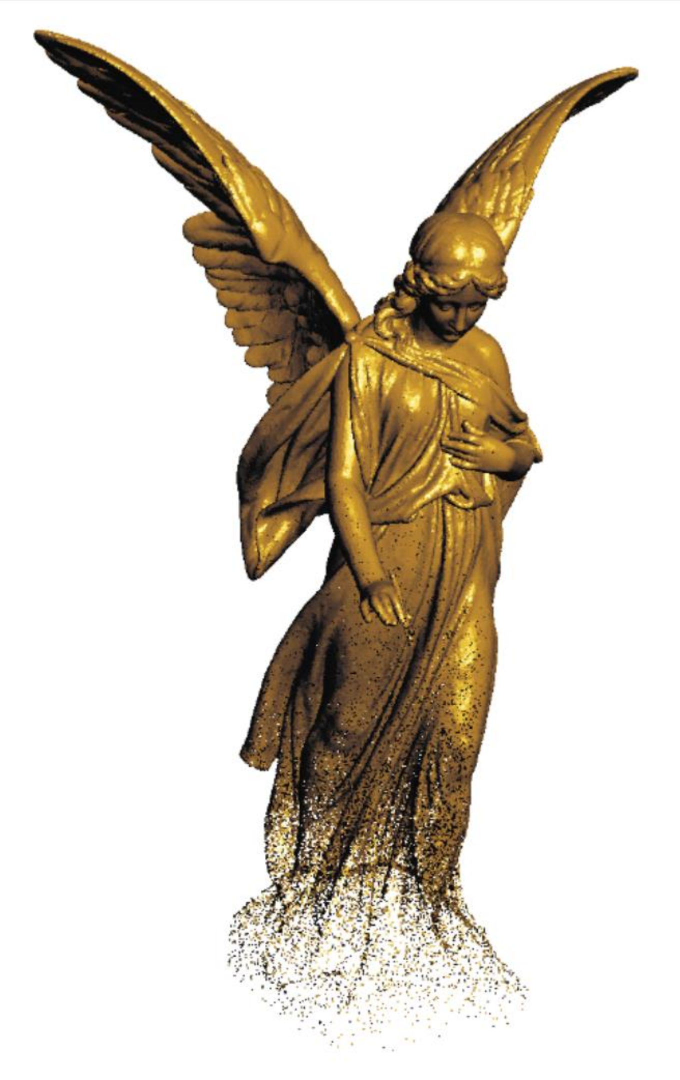
Point Cloud [Explicit]

- A list of points (x, y, z)
  - Often augmented with normal, colors, etc.

- [+] Easily represent any kind of geometry
- [+] Easy to draw dense cloud (>>1 point/pixel)
- [+] Useful for simulating topology changes (e.g., fluids, fractures, etc.)
- [-] Large lookup time
- [-] Hard to interpolate under-sampled regions
- [-] Hard to do processing (need to figure out connectivity)







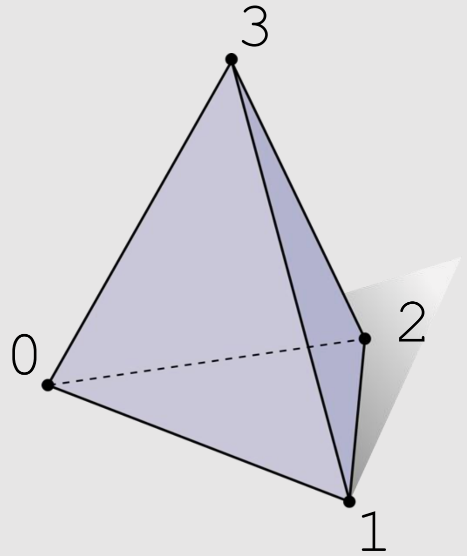
Triangle Mesh [Explicit]

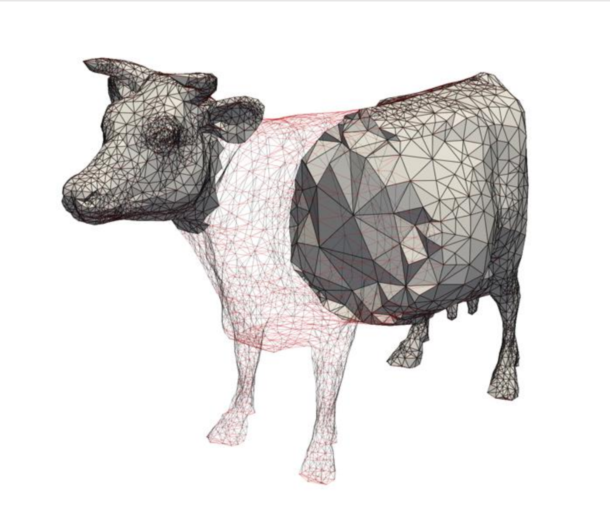
- Store vertices as triples of coordinates *(x,y,z)*
- Store triangles as triples of indices *(i,j,k)*

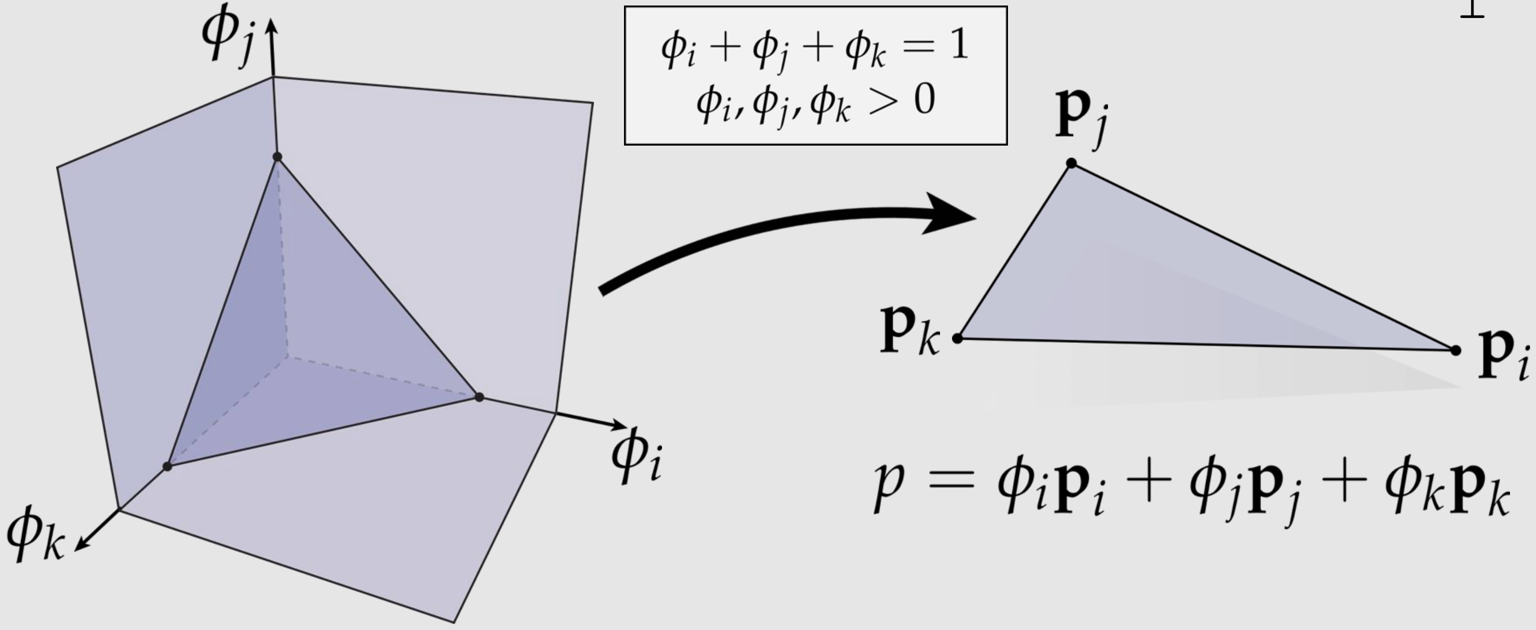
- [+] Easy interpolation with good approximation
  - Use barycentric interpolation to define points inside triangles
- [-] irregular neighborhood

- Polygonal Mesh: shapes do not need to be triangles
  - E.g., quads

	[VERTICES]			[TRIANGLES]		
	x	**y**	**z**	**i**	**j**	**k**
0:	-1	-1	-1	0	2	1
1:	1	-1	1	0	3	2
2:	1	1	-1	3	0	1
3:	-1	1	1	3	1	2







$$\boxed{\begin{array}{c}\phi_i + \phi_j + \phi_k = 1 \\ \phi_i, \phi_j, \phi_k > 0\end{array}}$$

$$p = \phi_i \mathbf{p}_i + \phi_j \mathbf{p}_j + \phi_k \mathbf{p}_k$$

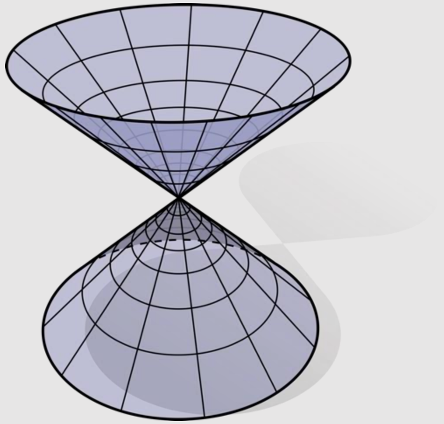
- ~~Implicit & Explicit Geometry~~
- Manifold Geometry
- Local Geometric Operations

Manifold Assumption

- A mesh is manifold if and only if it can exist in real life
  - Important for simulation/3D printing

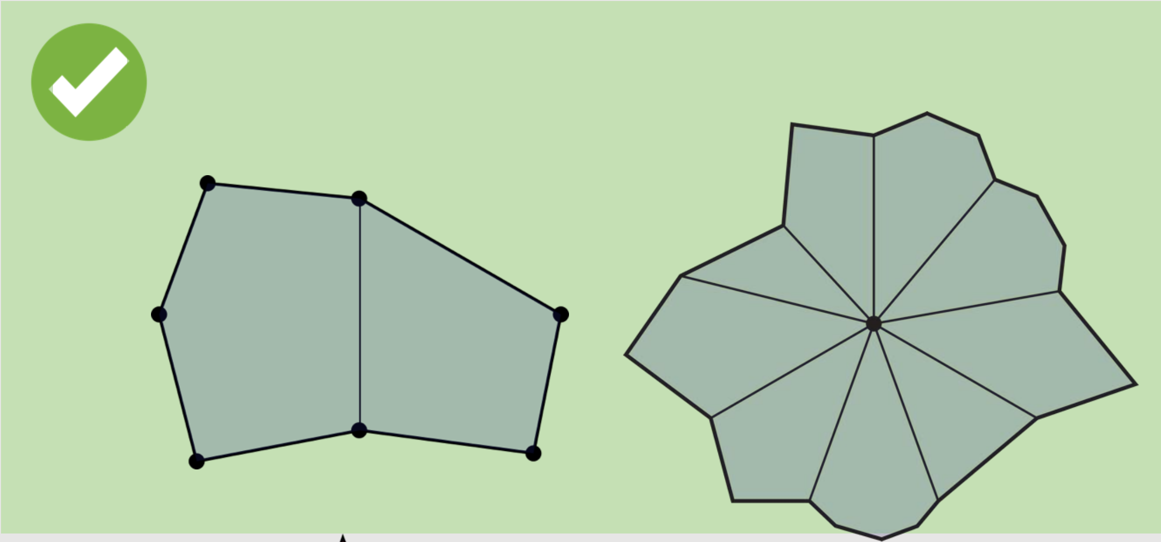
- Everything in real life has volume to it
  - Likewise, every manifold surface has some volume it encases
  - Allows us to think of manifold surfaces as ‘shells’ to an inner volume
    - **Example:** M&Ms

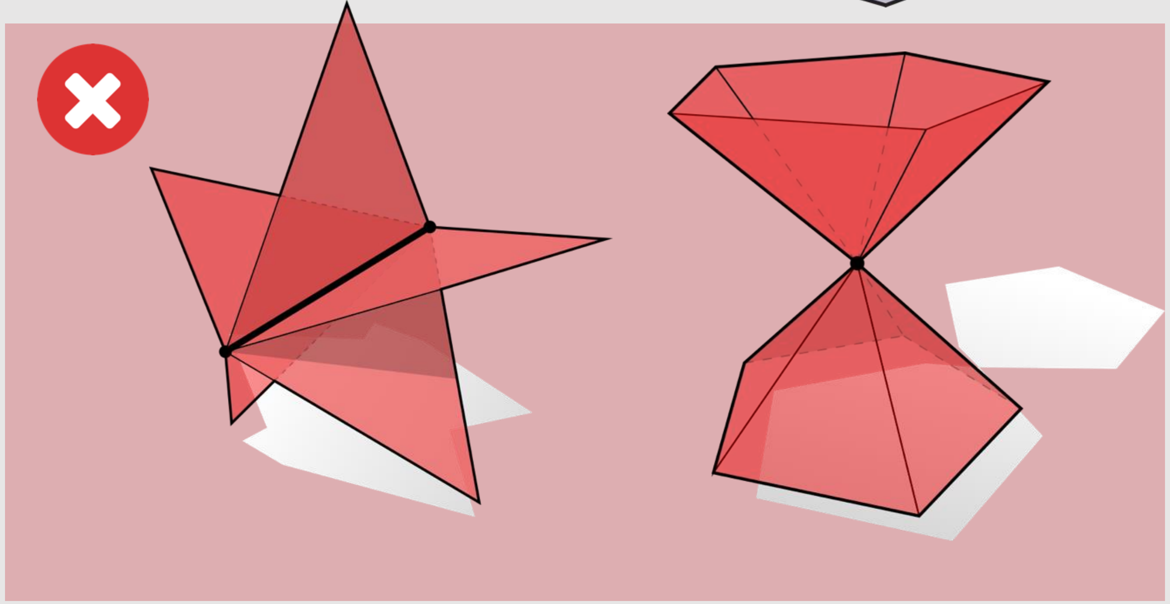
- Everything in real life, when zoomed in close enough, should be able to have a rectangular coordinate grid
- Likewise, every manifold surface should be planar when zoomed out far enough
  - **Example:** Planet Earth



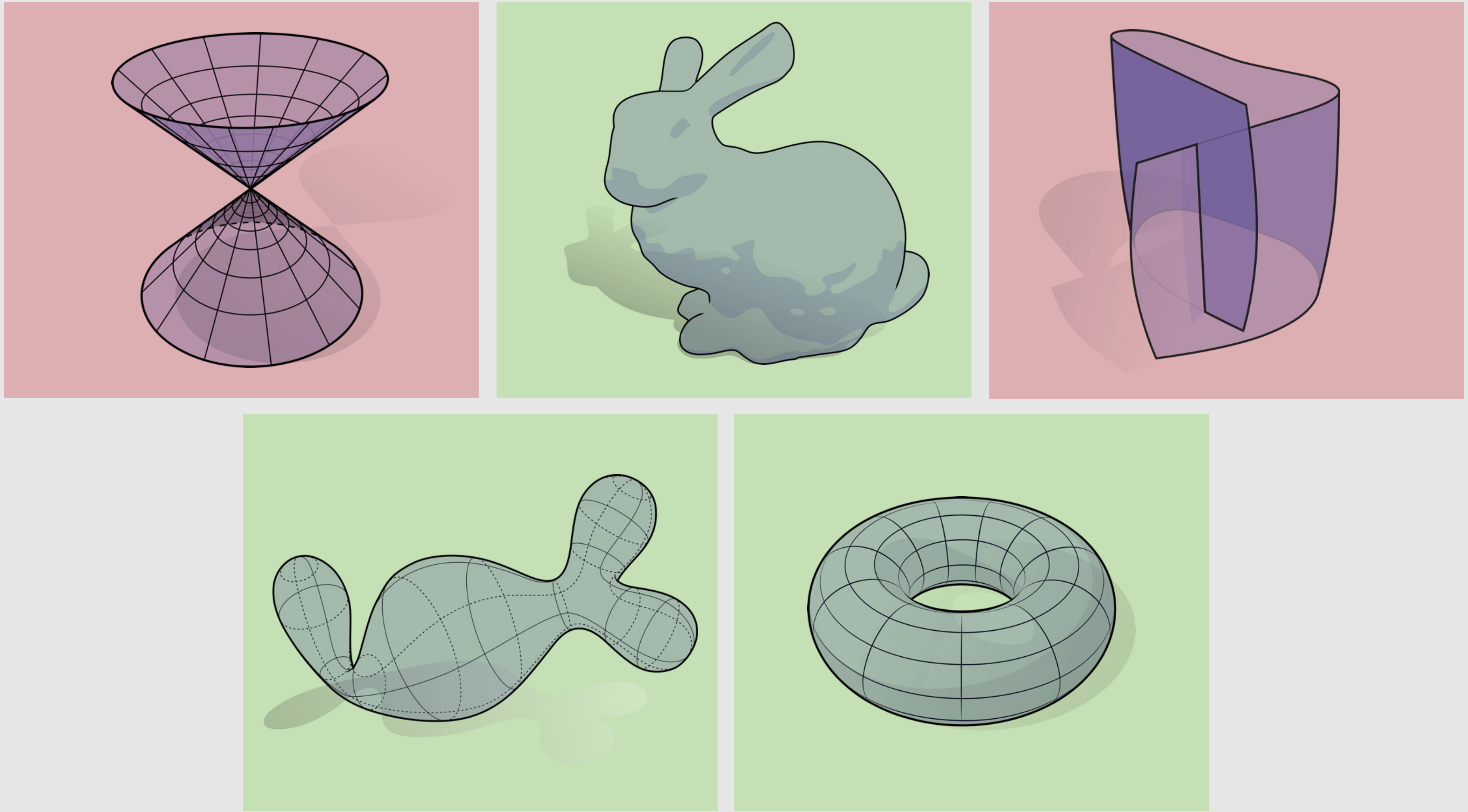
Manifold Properties on Polygon Meshes

- For polygonal surfaces, check for “**fins**” and ”**fans**”
- 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

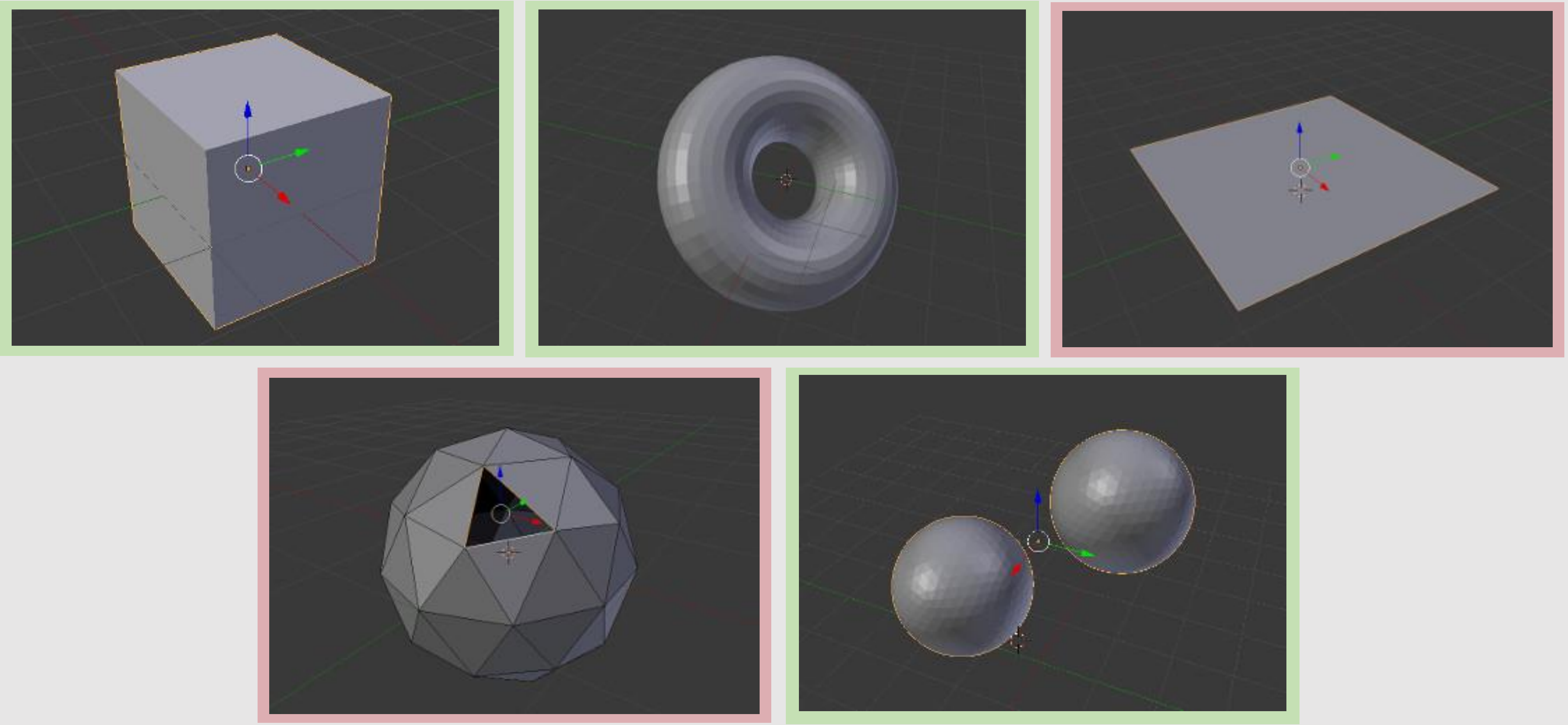




Manifold Check



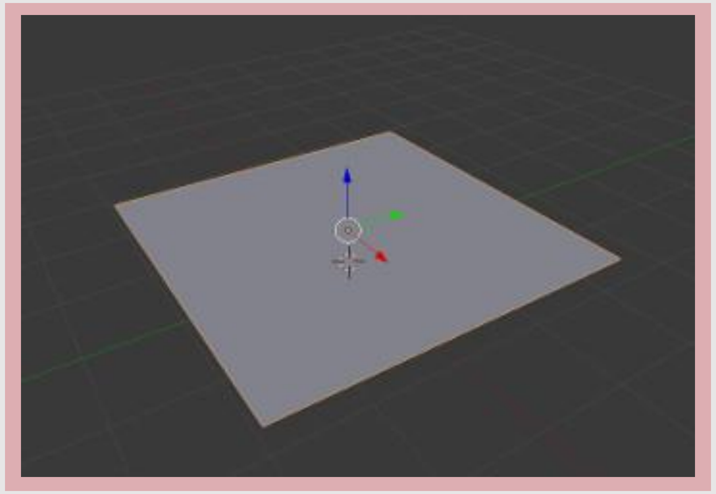
Manifold Check

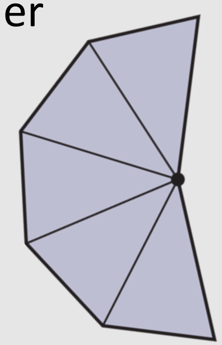


**https://github.com/rlguy/Blender-FLIP-Fluids/wiki/Manifold-Meshes

Planes Are Not Manifold

- Each edge of a plane only touches 1 polygon
  - Breaks the “fin” constraint
- More intuitively: **no notion of thickness!**
  - Does not exist in real life
  - Paper (best approximation of plane) still has thickness, about 0.1mm
- But it is efficient to represent such thin geometry using a single layer of surface mesh
- So we relax our assumption a bit, and allow the existence of **boundary edges**, which has 1 polygon per edge
  - -- a way to represent complex manifold geometry as simpler non-manifold geometry





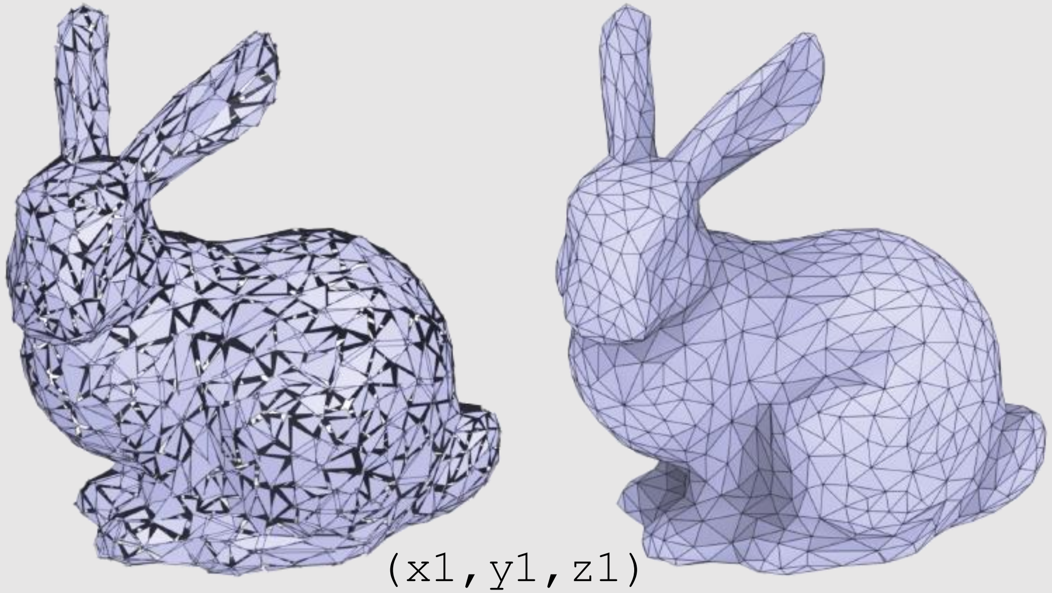
What are some ways to describe the connectivity of geometry?

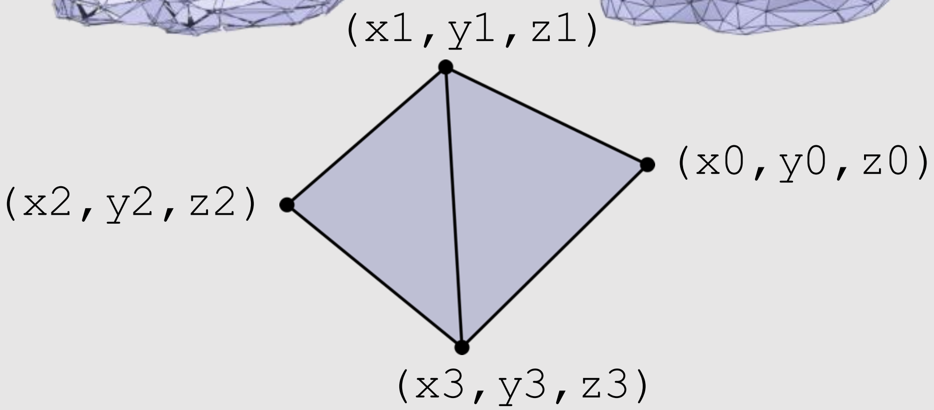
Polygon Soup

- 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

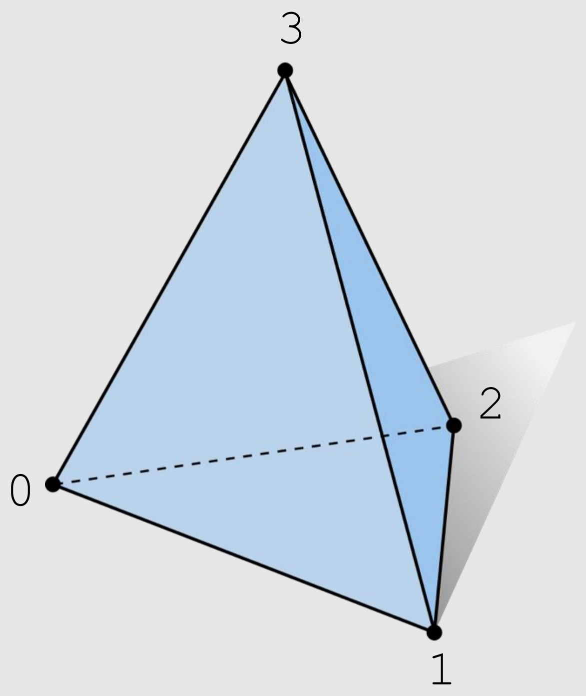
- **Cons:**
  - [-] Very difficult to find neighboring polygons
  - [-] Difficult to add/remove mesh elements

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

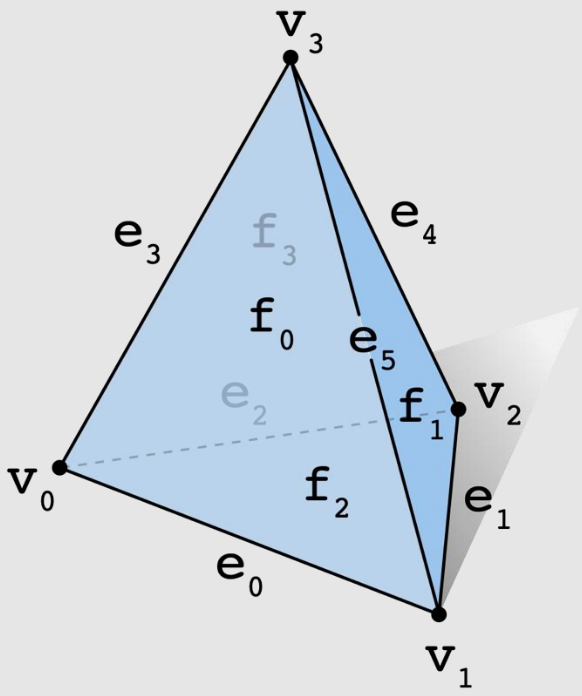
- **Cons:**
  - [-] Larger memory footprint
  - [-] Hard to change connectivity with fixed indices
  - [-] Difficult to add/remove mesh elements

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



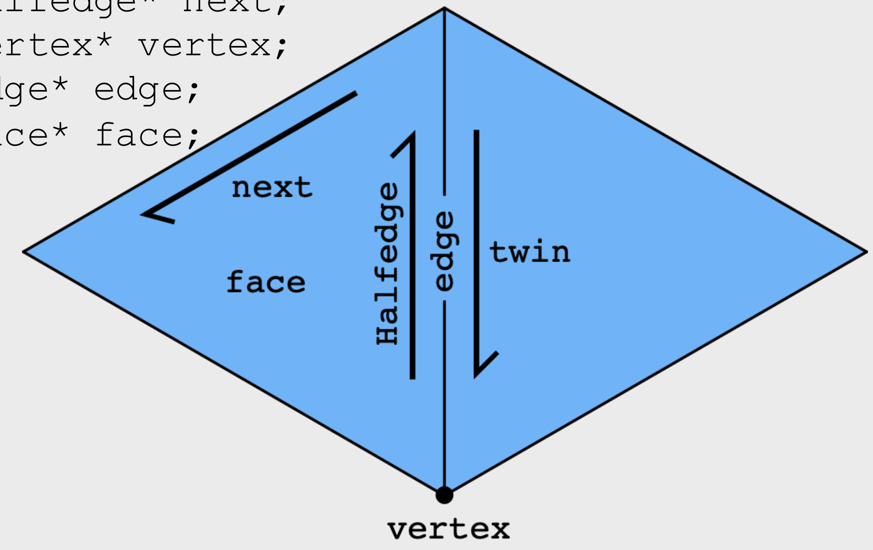
Halfedge Data Structure

- Let's store a little, but not a lot, about our neighbors:
  - Halfedge: edge with a direction (ordered)
    - E.g. edge formed by vertex v0 and v1 has 2 halfedges v0-v1 and v1-v0
    - 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;
};
```



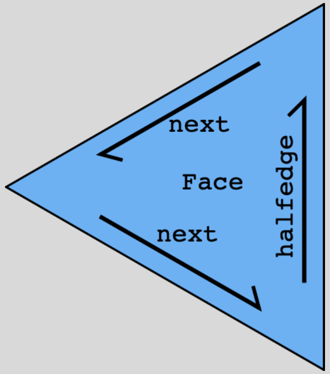
Halfedge Data Structure

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

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

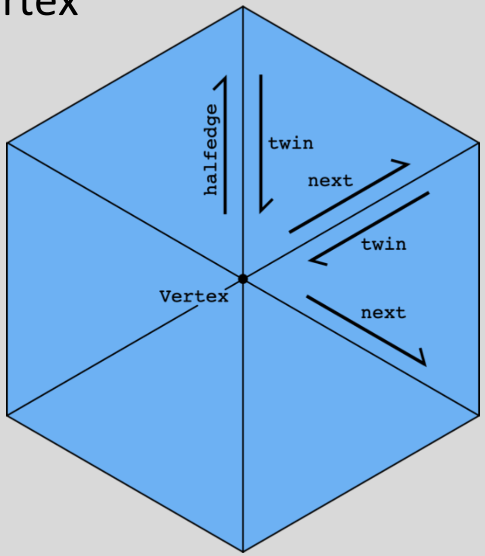
Example: visit all vertices in a face

```
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

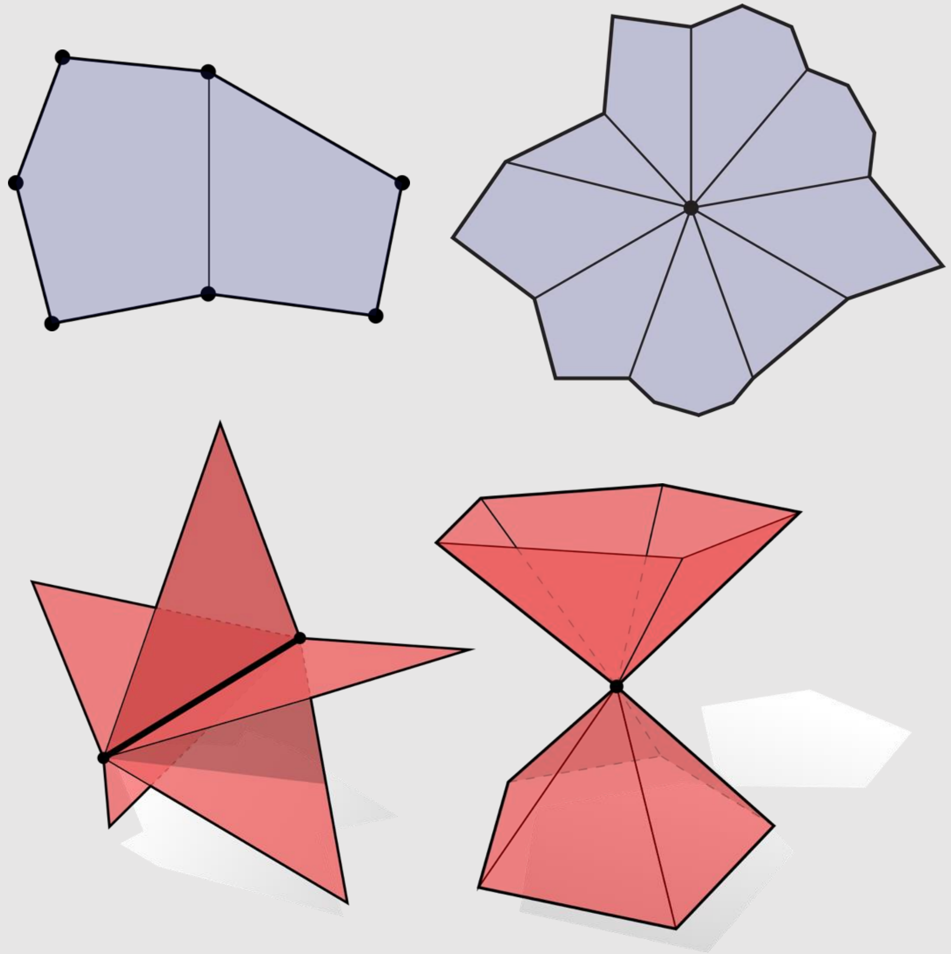
- Halfedge meshes are always manifold!

- 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's is someone's "next"
```

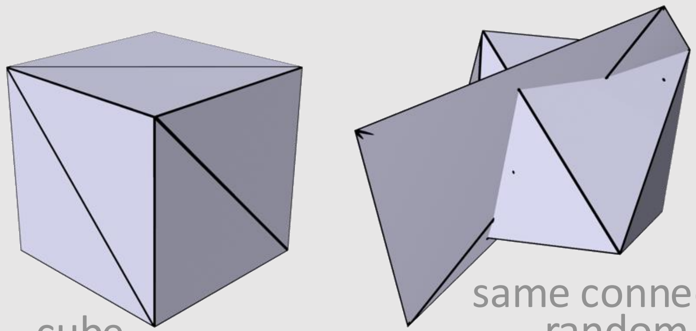
- Keep following **next** and you'll traverse a face
- Keep following **twin** and you'll traverse an edge
- Keep following **next->twin** and you'll traverse a vertex

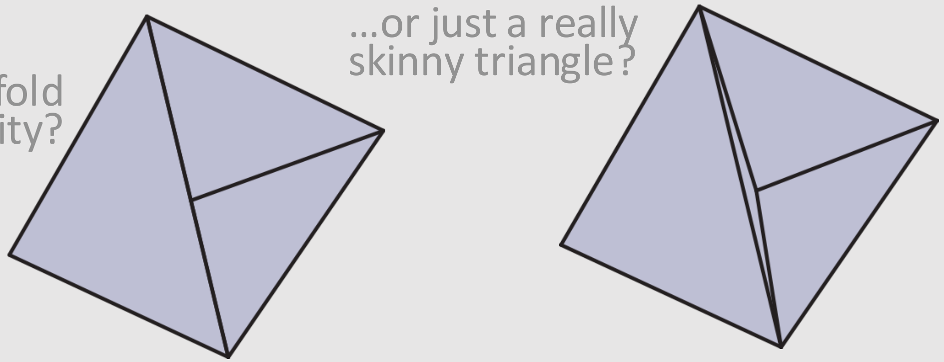
- **Q: Why, therefore, is it impossible to encode the red figures?**
  - Not enough variable to store the full connectivity
  - The traversal will not work either.



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
- Leads to confusion when debugging:
  - Mesh looks “bad”, even though connectivity is fine

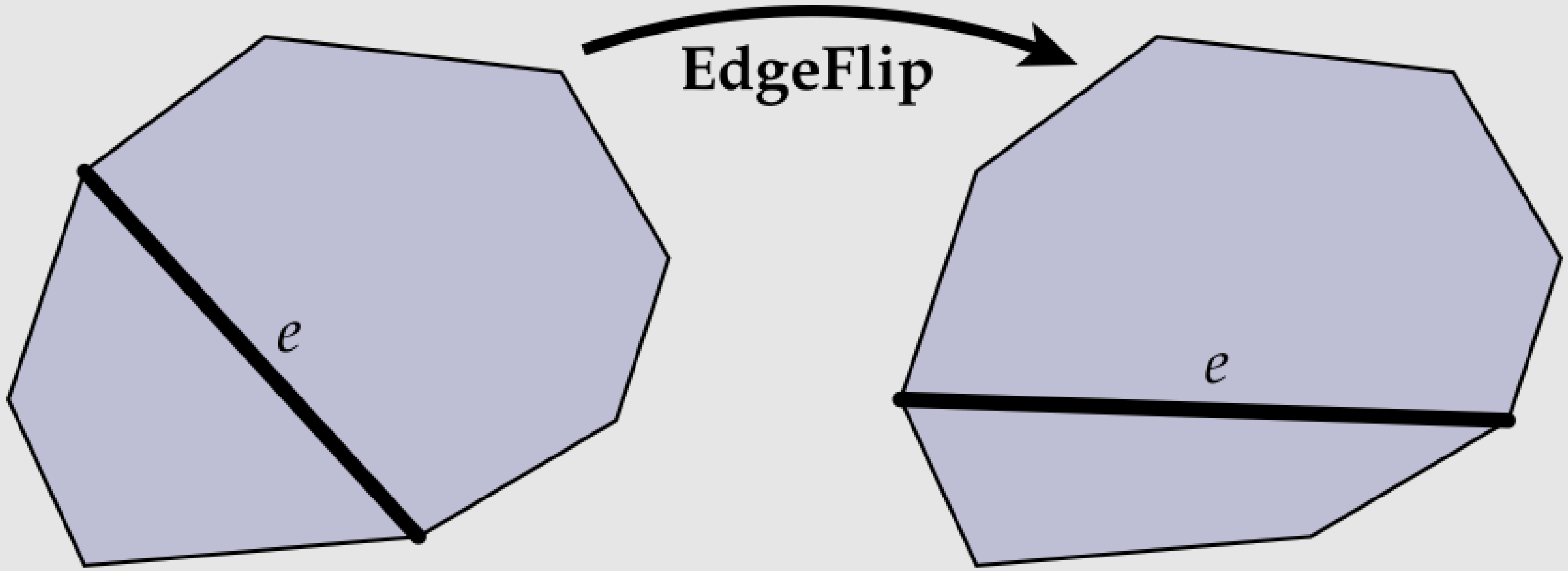




- ~~Implicit & Explicit Geometry~~
- ~~Manifold Geometry~~
- Local Geometric Operations

Edge Flip

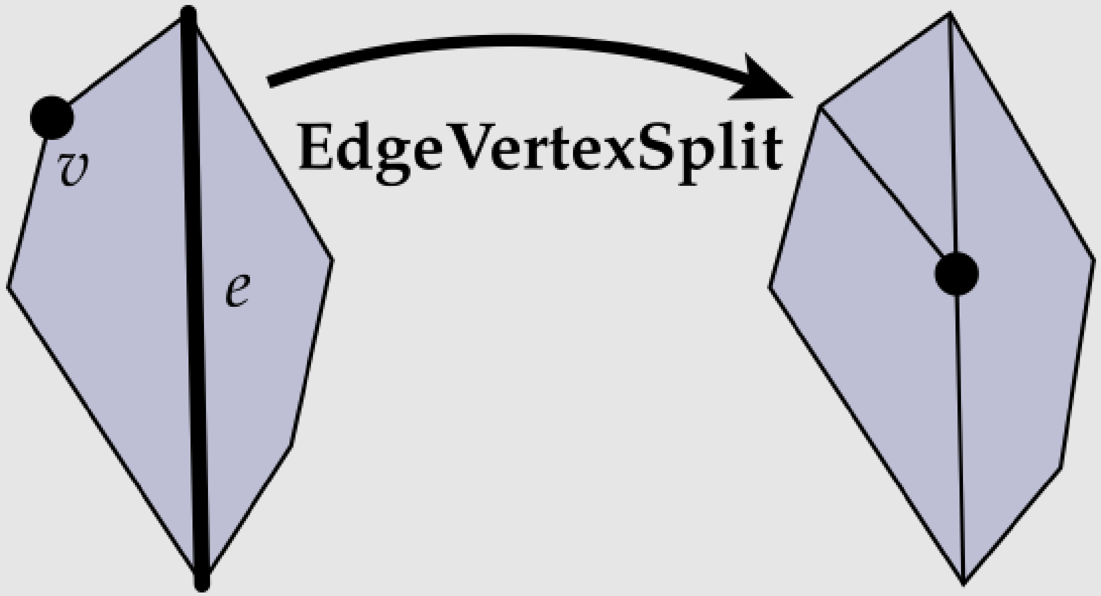
Goal: Move edge e around faces adjacent to it:



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

Edge Vertex Split

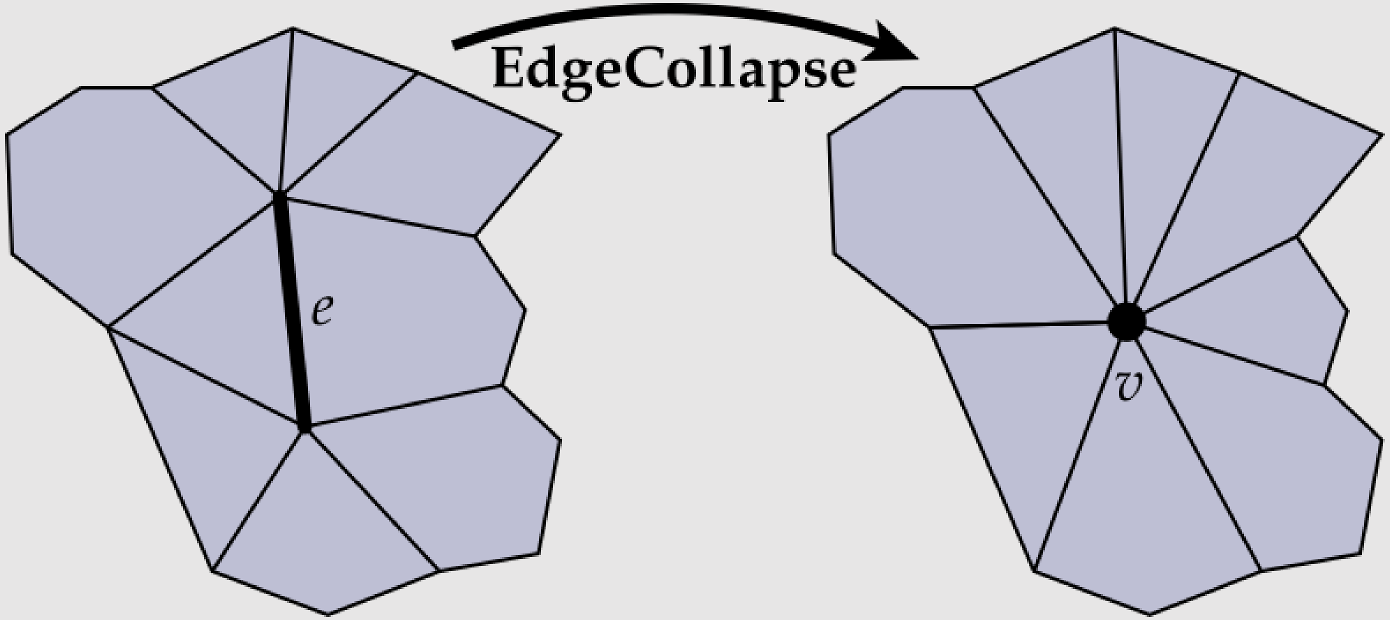
Goal: Insert edge between vertex v and midpoint of edge e:



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

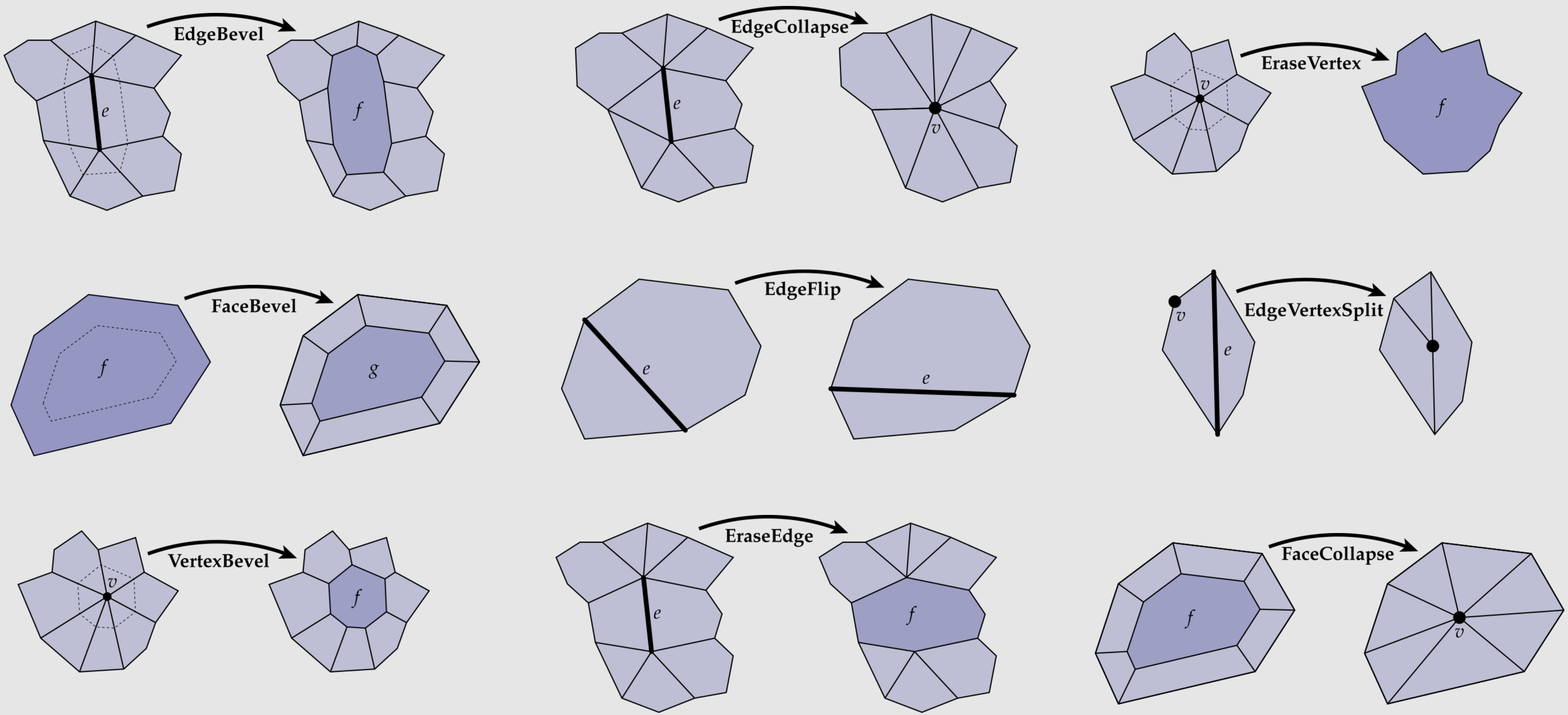
Edge Collapse

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

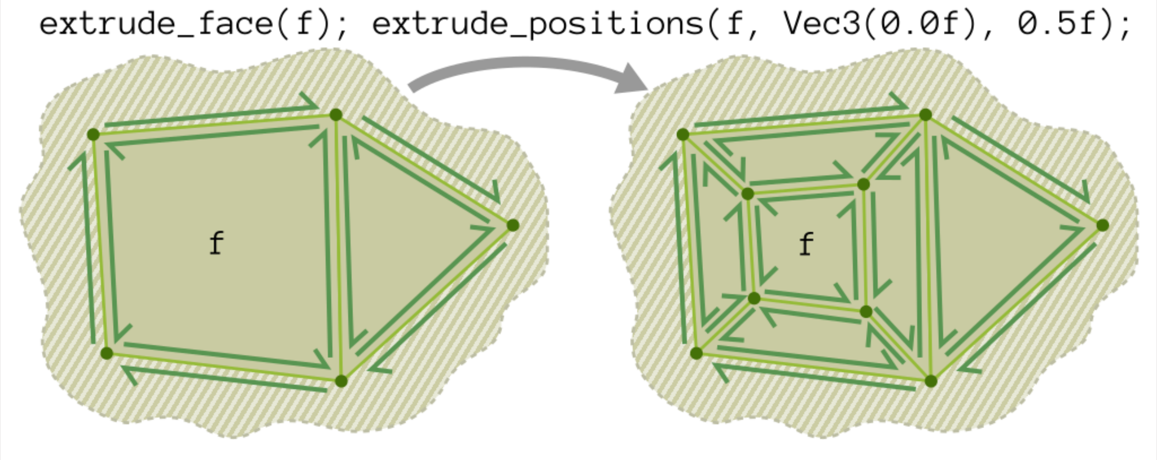


Many other local operations you will explore in your homework…

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.