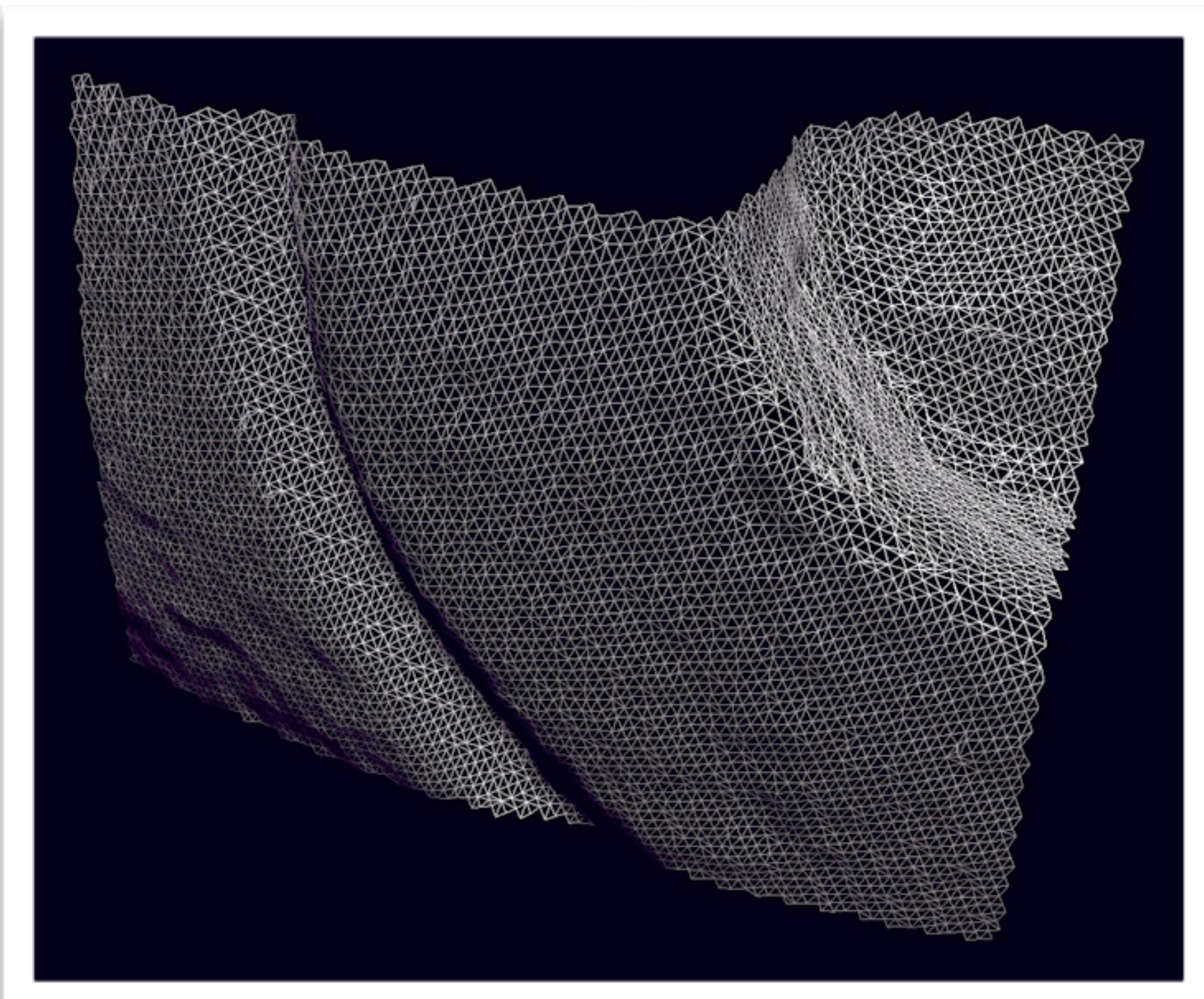
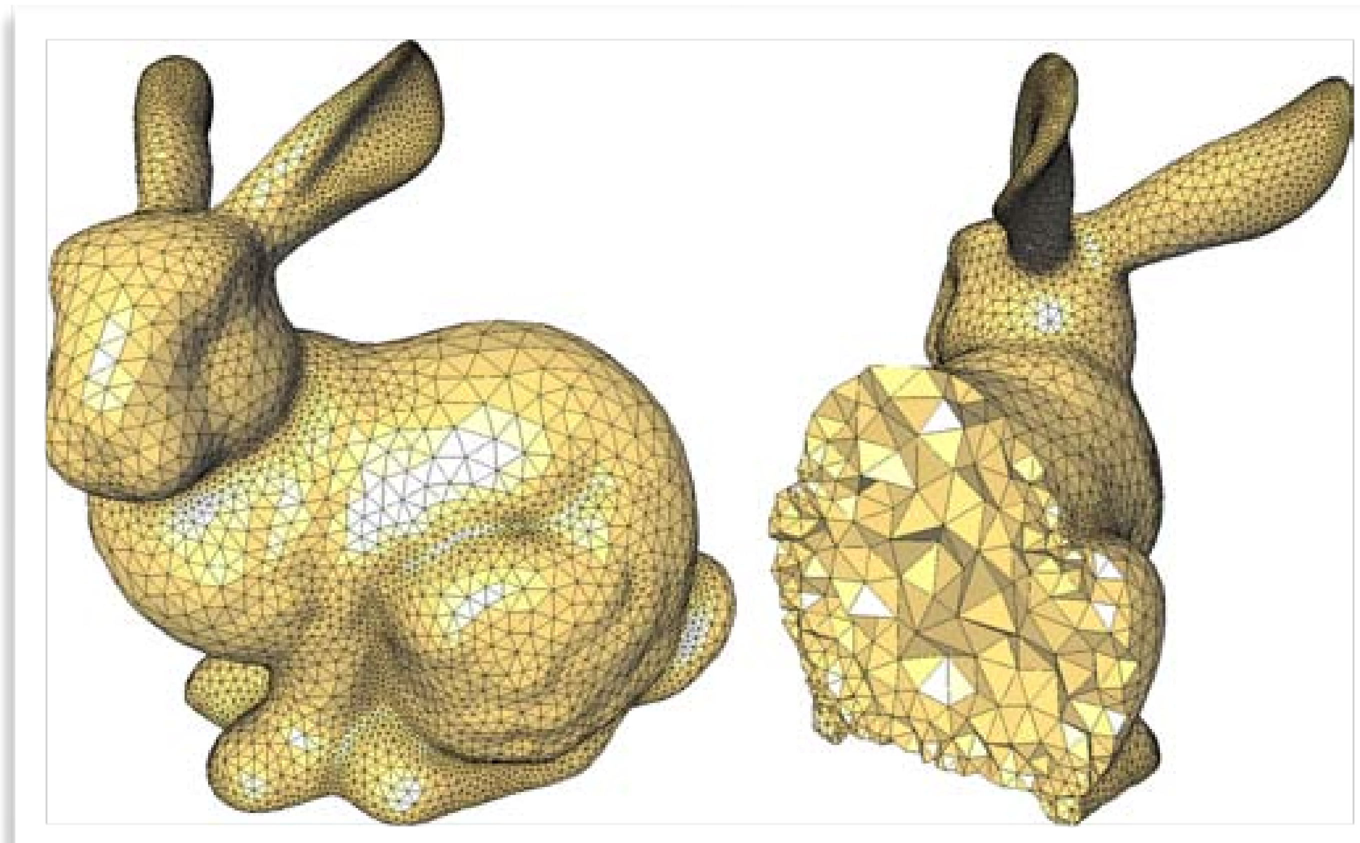
Representations of discrete solid geometry.
Dynamic meshes and introduction to Pixar USD





Announcements

- If you have trouble enrolling in Piazza:
  - Try : http://piazza.com/wisc/fall2019/cs839 (signup link)
  - If still not working, email the instructor
- Do try the USD installation, as soon as possible!
  - Just first of 3 stages of getting class software set-up
  - Seek help early!
  - Some up-front pain, to save you later headaches!
- Decisions of enrollment authorization by next Wednesday

Announcements

- Big THANK YOU to troubleshooting contributors on Piazza!
    - Great showing of class service, immensely appreciated
    - Please keep it up :)

Today's topics

- Describe established discrete representations used to encode solid bodies for modeling and simulation purposes
  - Surface (polygon) meshes
  - Volumetric meshes
  - Introduction to Pixar USD - Demo & API walkthrough
- Discuss the features of these representations specific to simulation, as opposed to general geometry processing or rendering
  - Objects need to support *dynamic deformation*
  - Volumetric objects need internal structure
  - Discrete geometry needs to be *simulation-quality (well-conditioned)*

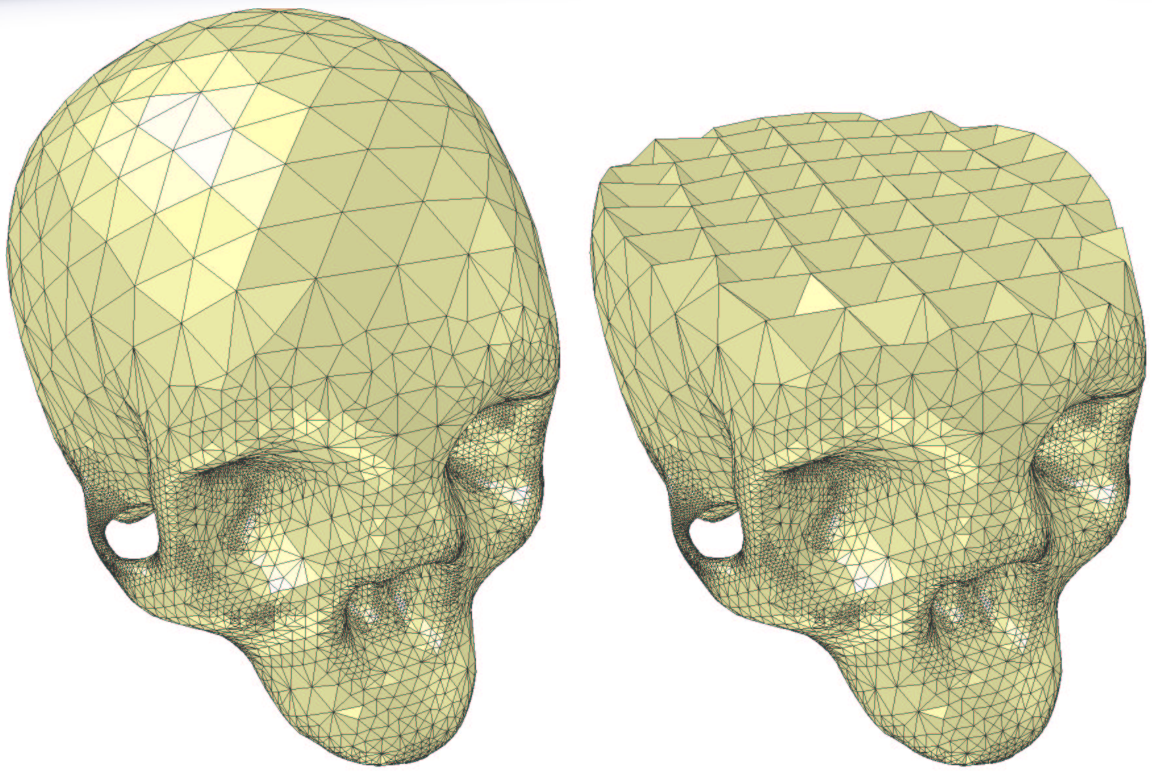
Your TODO list

- Previous lecture : Installing USD (Part I of infrastructure setup)
- Today : Download, compile and test simple demos that use the USD API (Part II of setup)
  - Obtain from : https://github.com/uwgraphics/PhysicsBasedModeling-Demos
  - Should provide 3 tools (helloWorld, animateAttribute, USDtoOBJ)
  - (demonstration of results)
  - As always, report build issues on Piazza.
- Deliverable by Sunday Sep 15th (end-of-day)
  - Create your own, animated USD scene (however simple), created programmatically via a program using the USD API
  - Don't worry about it being "simulated" (we'll do that next)
  - Suggested : Try a nontrivial deformation (Stretch, scale, twist, pinch, etc)
  - Optional : Import a custom model from the Web (e.g. as OBJ file)

Up next …

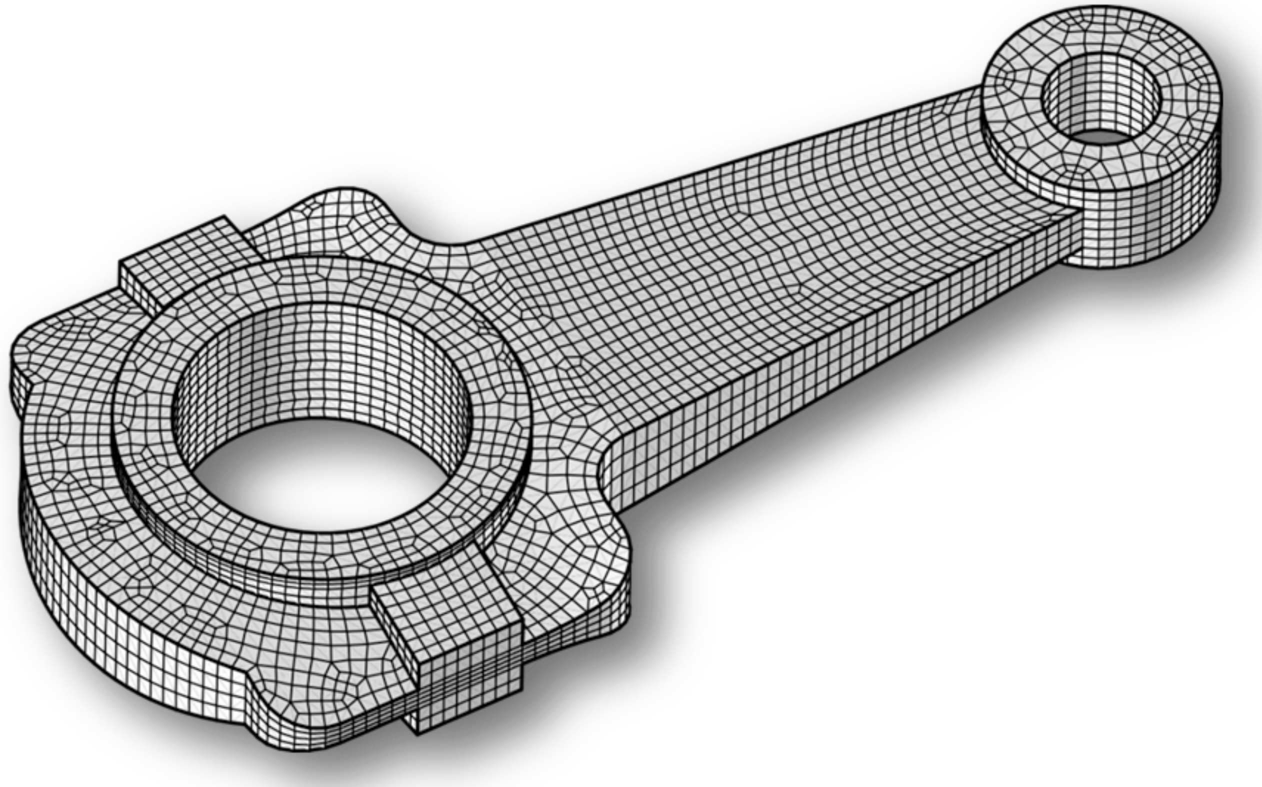
- Explain the features that make one representation better than another for certain tasks (e.g. meshes vs. implicit surfaces)
  - Static vs. dynamic topology (connectivity)
  - “Shape memory” and deformation drift
  - Regular, structured storage
  - Efficiency of geometric queries

Discrete representation of solid geometry

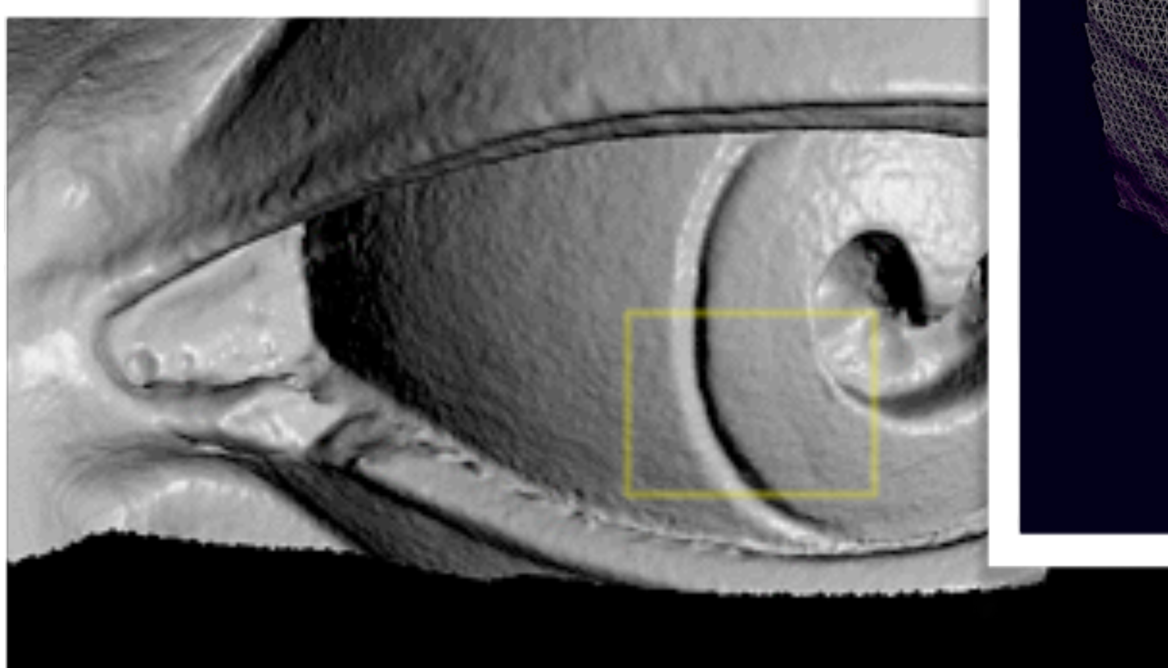


Tetrahedral meshes
(volumetric)

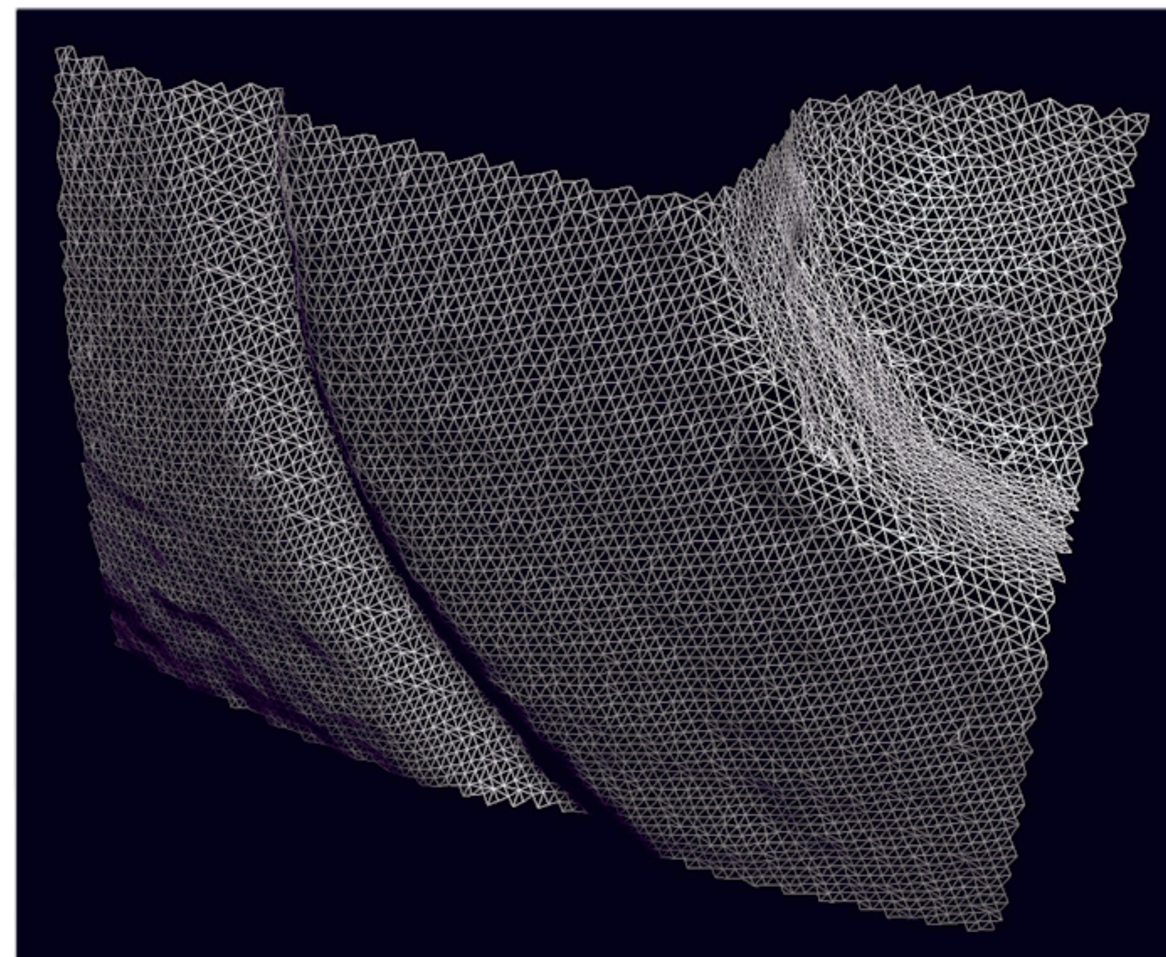
Hexahedral meshes
(volumetric)



Discrete representation of solid geometry



Triangular *surface* meshes
(not volumetric)



Discrete representation of solid geometry

"Meshed" geometry
(or just "geometry")

→

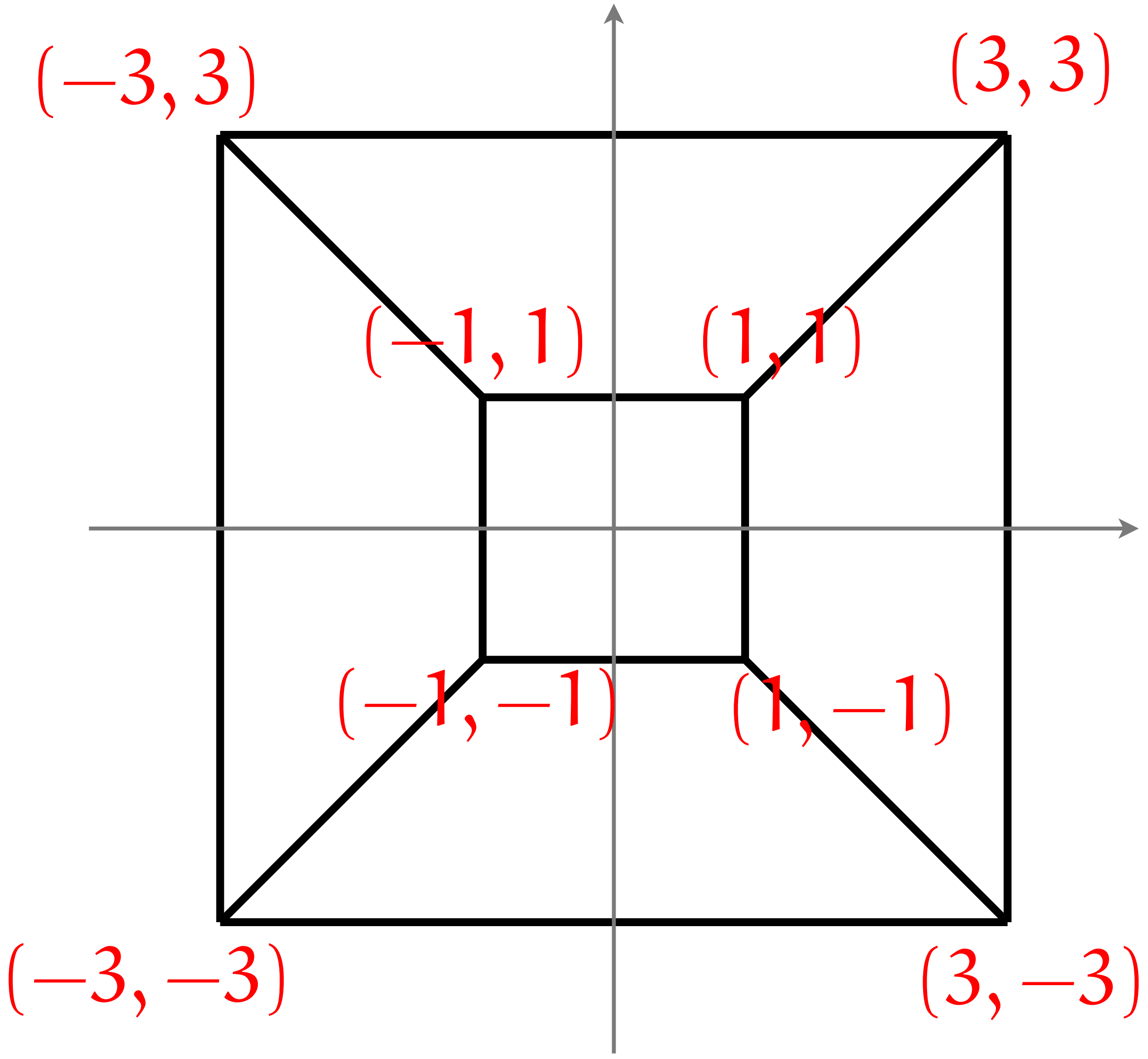
Particles

+

Mesh
(topology/connectivity)

Discrete representation of solid geometry

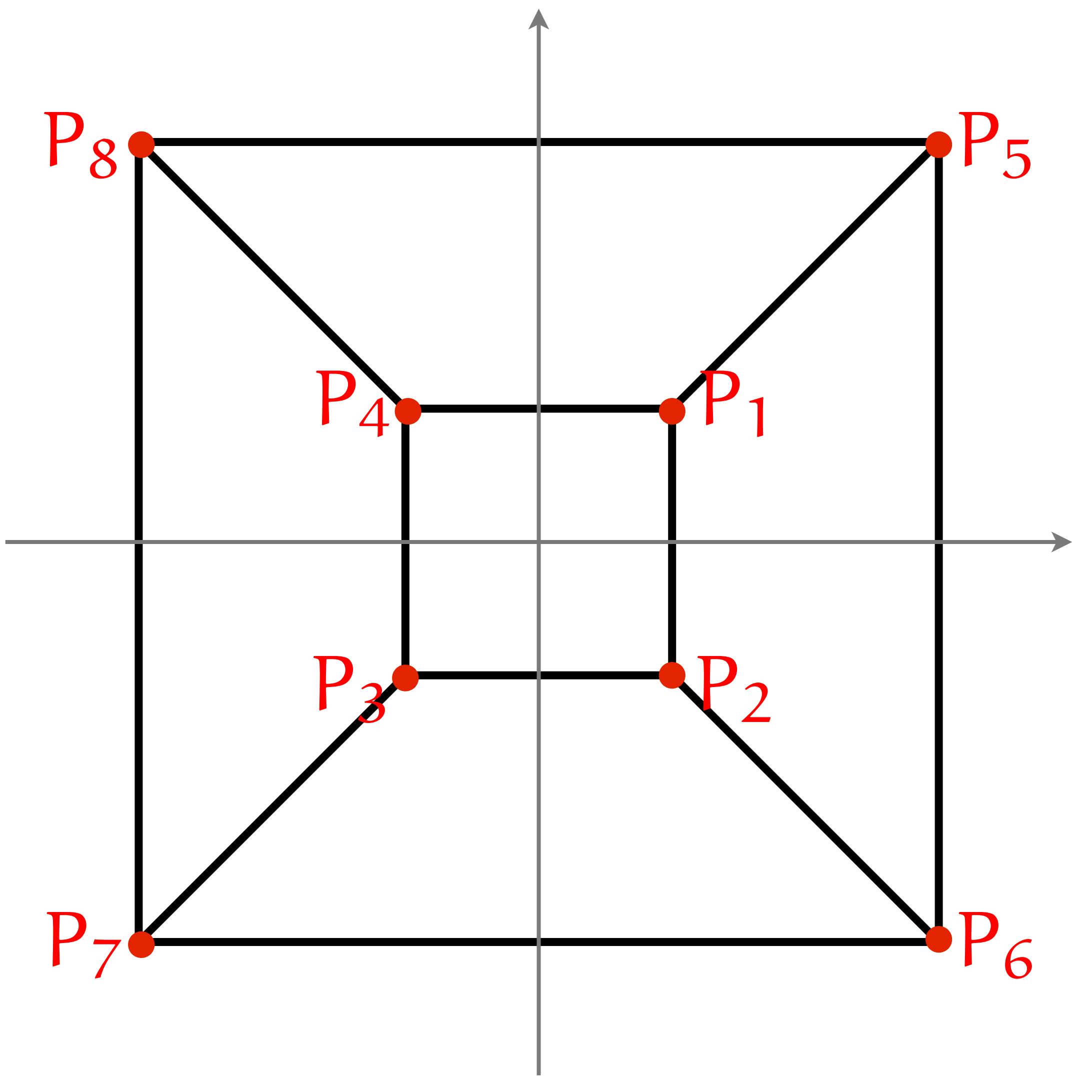
Example:
A quadrilateral mesh



Discrete representation of solid geometry

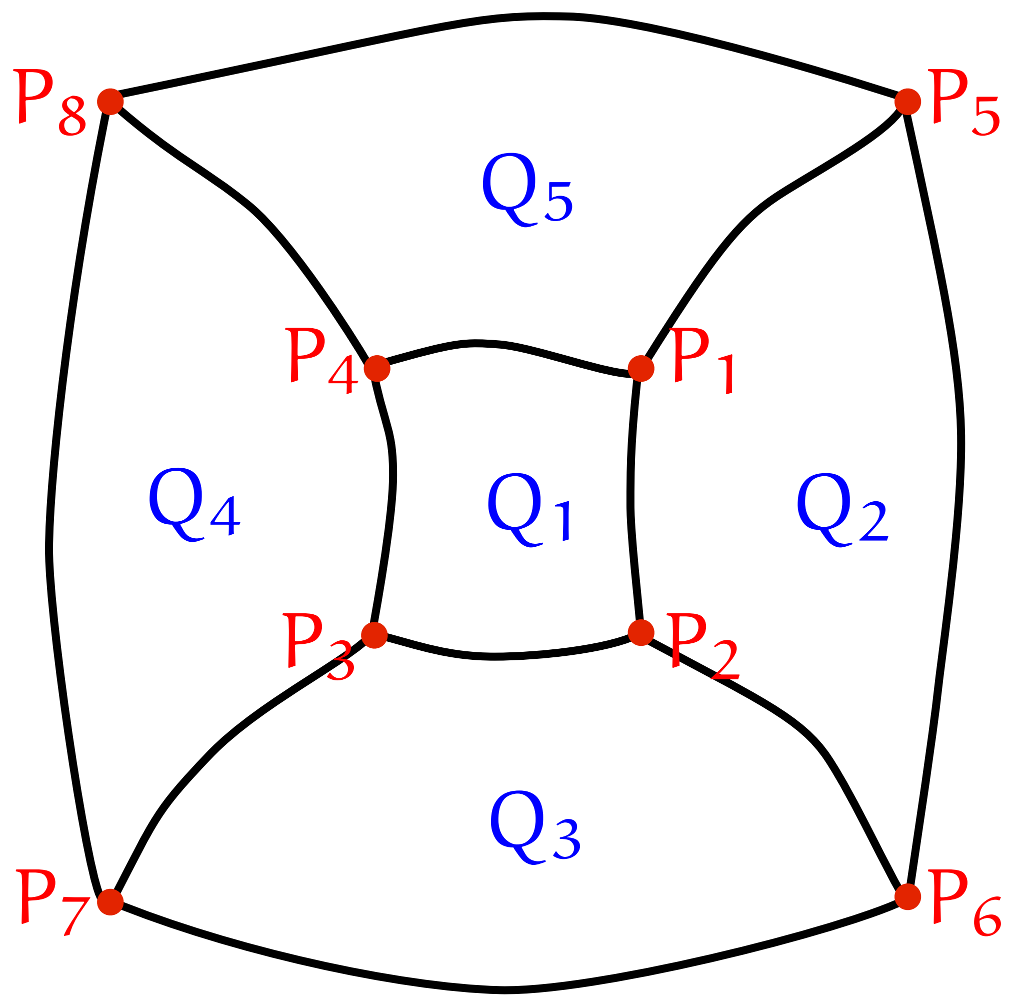
Particle data structure

Particle ID	Position
P_1	(1, 1)
P_2	(1, -1)
P_3	(-1, -1)
P_4	(-1, 1)
P_5	(3, 3)
P_6	(3, -3)
P_7	(-3, -3)
P_8	(-3, 3)



Discrete representation of solid geometry

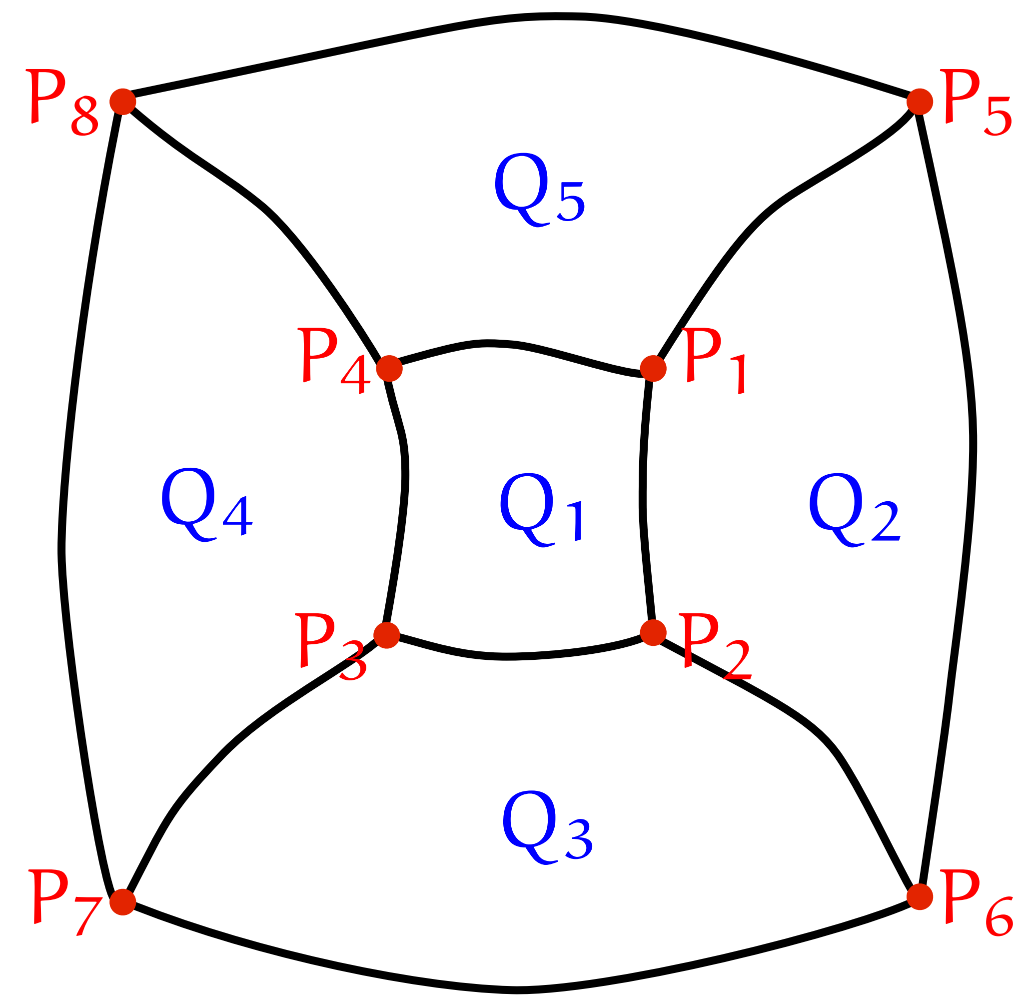
Mesh data structure



Discrete representation of solid geometry

Mesh data structure

Element (Quad) ID	Vertices
Q_1	(P_1, P_2, P_3, P_4)
Q_2	(P_1, P_5, P_6, P_2)
Q_3	(P_2, P_6, P_7, P_3)
Q_4	(P_3, P_7, P_8, P_4)
Q_5	(P_4, P_8, P_5, P_1)



Discrete representation of solid geometry

Physical attributes	✓ Position ✓ Velocity ✓ Acceleration ✓ Force ✓ Mass, etc ....
Secondary attributes	✓ Texture coordinates ✓ Color ✓ Translucency, etc ...

Why "particles"?
(and not "points", "vertices", ...)

Discrete representation of solid geometry

Particles : Implementation #1

```
struct Particle{
  float position[3];
  float velocity[3];
  float mass;
};

struct Particle particle_array[N];
```

Particles : Implementation #2

```
struct Particles{
  float positions[N][3];
  float velocities[N][3];
  float masses[N];
} particle_array;
```

Discrete representation of solid geometry

Particles : Implementation #1

```
struct Particle{
  float position[3];
  float velocity[3];
  float mass;
};

struct Particle particle_array[N];
```

Implementation #1 - BENEFITS

- Particles are self-contained
- Easy to construct subsets of particles
- Can extend to accommodate particles with different attributes, on the same array

Particles : Implementation #2

```
struct Particles{
  float positions[N][3];
  float velocities[N][3];
  float masses[N];
} particle_array;
```

Discrete representation of solid geometry

Particles : Implementation #1

```
struct Particle{
  float position[3];
  float velocity[3];
  float mass;
};

struct Particle particle_array
```

Particles : Implementation #2

```
struct Particles{
  float positions[N][3];
  float velocities[N][3];
  float masses[N];
} particle_array;
```

Implementation #2 - BENEFITS

- Simulation algorithms typically stream different properties during different passes - separation improves bandwidth
- Easy to construct **subsets of attributes** *(e.g. for visualization)*

Wavefront OBJ mesh format (.obj)

```
v 0.0625 0.125 0.25
v 0.0625 0.125 1.25
v 0.0625 1.125 0.25
v 0.0625 1.125 1.25
v 1.0625 0.125 0.25
v 1.0625 0.125 1.25
v 1.0625 1.125 0.25
v 1.0625 1.125 1.25
f 1 2 3
f 2 4 3
f 5 7 8
f 5 8 6
f 1 5 6
f 1 6 2
f 3 7 4
f 4 7 8
f 2 8 4
f 2 6 8
f 1 3 5
f 3 7 5
```

Note: 1-based indexing of vertices

A USD (static) scene with a triangle mesh *(helloWorld.usda)*

```
#usda 1.0

def Mesh "TriangulatedSurface"
{
    float3[] extent = [(0, 0, 0), (1, 1, 1)]
    int[] faceVertexCounts = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
    int[] faceVertexIndices = [0, 1, 2, 1, 3, 2, 4, 6, 7, 4, 7, 5, 0, 4, 5, 0, 5, 1,
                               2, 6, 3, 3, 6, 7, 1, 7, 3, 1, 5, 7, 0, 2, 4, 2, 6, 4]
    point3f[] points = [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),
                        (1, 0, 1), (1, 1, 0), (1, 1, 1)]
}
```

Note: 0-based indexing of vertices

A USD (dynamic) scene with a triangle mesh *(helloWorld.usda)*

```
#usda 1.0

def Mesh "TriangulatedSurface0"
{
    float3[] extent = [(0, 0, 0), (1.625, 2.25, 3.5)]
    int[] faceVertexCounts = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
    int[] faceVertexIndices = [0, 1, 2, 1, 3, 2, 4, 6, 7, 4, 7, 5, 0, 4, 5, 0, 5, 1,
                               2, 6, 3, 3, 6, 7, 1, 7, 3, 1, 5, 7, 0, 2, 4, 2, 6, 4]
    point3f[] points.timeSamples = {
        0: [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)],
        1: [(0.03125, 0.0625, 0.125), (0.03125, 0.0625, 1.125), (0.03125, 1.0625, 0.125),
            (0.03125, 1.0625, 1.125), (1.03125, 0.0625, 0.125), (1.03125, 0.0625, 1.125),
            (1.03125, 1.0625, 0.125), (1.03125, 1.0625, 1.125)],
        2: [(0.0625, 0.125, 0.25), (0.0625, 0.125, 1.25), (0.0625, 1.125, 0.25), (0.0625, 1.125, 1.25),
            (1.0625, 0.125, 0.25), (1.0625, 0.125, 1.25), (1.0625, 1.125, 0.25), (1.0625, 1.125, 1.25)],
        3: [(0.09375, 0.1875, 0.375), (0.09375, 0.1875, 1.375), (0.09375, 1.1875, 0.375),
            (0.09375, 1.1875, 1.375), (1.09375, 0.1875, 0.375), (1.09375, 0.1875, 1.375),
            (1.09375, 1.1875, 0.375), (1.09375, 1.1875, 1.375)],
       [...]
       20: [(0.625, 1.25, 2.5), (0.625, 1.25, 3.5), (0.625, 2.25, 2.5),
            (0.625, 2.25, 3.5), (1.625, 1.25, 2.5), (1.625, 1.25, 3.5),
            (1.625, 2.25, 2.5), (1.625, 2.25, 3.5)],
    }
}
```

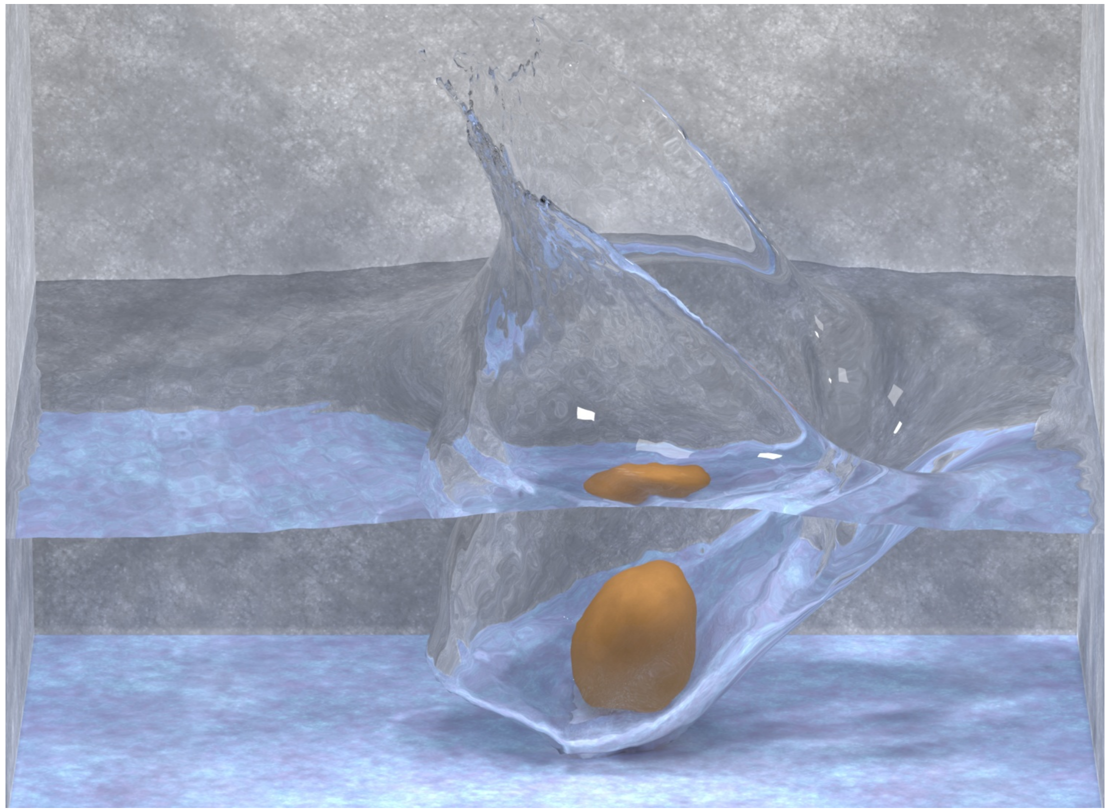
Summary

- *Meshed objects* are composed of 2 parts:
  - An array of *particles* (with "attributes" such as position, velocity, mass, etc)
  - A *mesh* data structure, encoded as an array of segments, triangles, tetrahedra, etc
    (whose vertices are the predefined particles)
- Topological queries & Derivative structures
  - ✓ Can be precomputed, do not need to store explicitly
- Geometrical queries (collisions, inside/outside tests)
  - ✓ Cannot be precomputed, since they depend on the particle attribute values
  - ✓ Potentially expensive to determine

(Sneak preview; more in 2nd half of class)

Alternative representations of volumetric geometry

Point Clouds, Levelsets & implicit surfaces



Implicit curves and surfaces (a.k.a. level-sets)

- Motivation
  - ✓ Accelerated geometric queries for problems such as:
    - ➡ Is a point (x^*,y^*) *inside* the object?
    - ➡ Is a point (x^*,y^*) *within a distance of* d^* from the object surface?
    - ➡ What is the point on the surface which is *closest* to the query point (x^*,y^*)?

Implicit curves and surfaces (a.k.a. level-sets)

- Motivation

  ✓ Easy modeling of motions that involve topological change, e.g. shapes splitting or merging

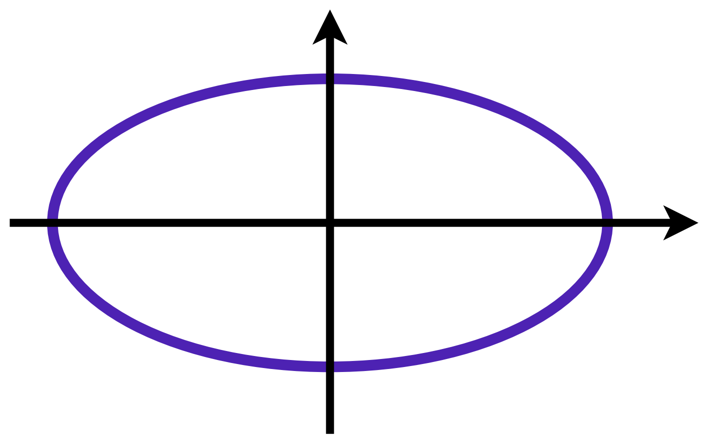
  ✓ Such operations are difficult to encode with meshes, since they don't "split" or "merge" unless we force them to

Implicit curves and surfaces (a.k.a. level-sets)

- Familiar representations address *some* of these demands:

✓ e.g. Analytic equations

➡ For an ellipsis:



$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

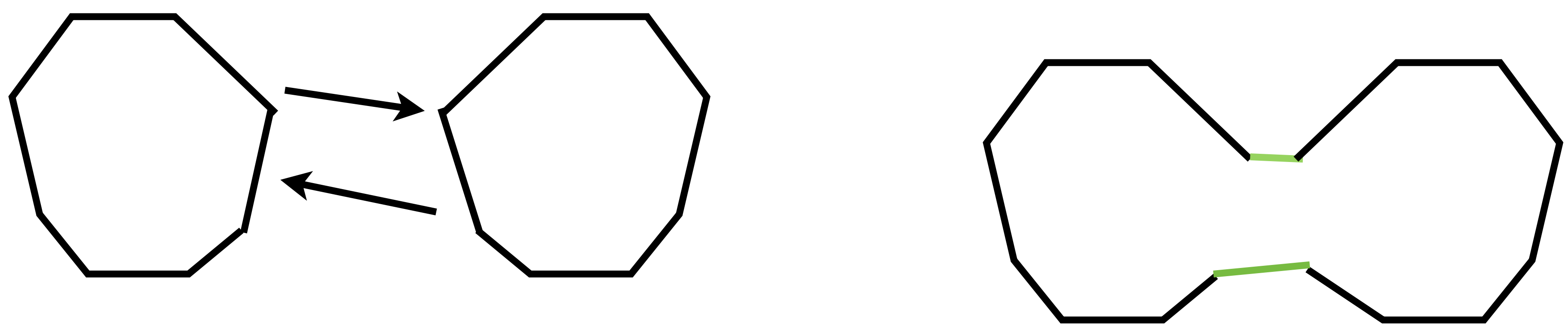
➡ Easy inside/outside tests

$$\frac{x_*^2}{a^2} + \frac{y_*^2}{b^2} < 1 \Leftrightarrow (x_*, y_*) \text{ is inside}$$

Implicit curves and surfaces (a.k.a. level-sets)

- Familiar representations address *some* of these demands:

  ✓ Describe a closed region via its boundary; split and reconnect when necessary



➡ This may be tractable in isolated cases, but very cumbersome and impractical for more complicated cases, and with 3-dimensional surfaces

The level-set concept

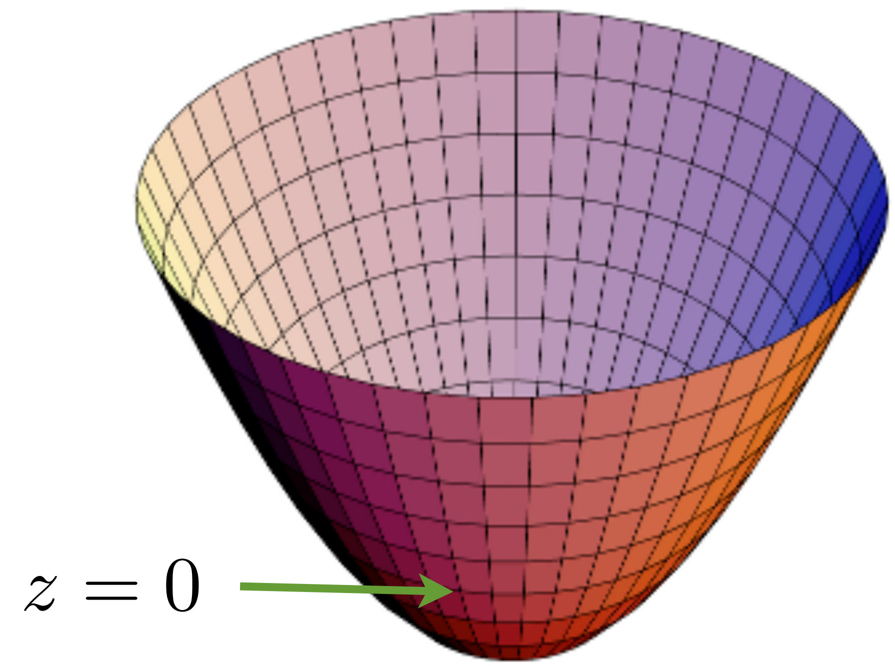
- Represent a curve in 2D (or, a surface in 3D) as the zero isocontour of a (continuous) function, i.e.

$$\mathcal{C} = \{(x.y) \in \mathbf{R}^2 : \phi(x,y) = 0\}$$

e.g.

$$\text{circle } x^2 + y^2 = R^2 \equiv \{(x,y) : \phi(x,y) = 0\}$$

$$where\ \phi(x,y) = x^2 + y^2 - R^2$$



The level-set concept

- This representation may seem redundant (we store information everywhere, just to capture a curve), but it conveys important benefits:
  - ➡ Containment queries

$$\text{Is } (x_*, y_*) \text{ inside } \mathcal{C}? \Leftrightarrow \phi(x_*, y_*) < 0$$

  - ➡ Composability

$$\left.\begin{array}{l}\phi_1(x,y) \text{ encodes } \Omega_1\\ \phi_2(x,y) \text{ encodes } \Omega_2\end{array}\right\} \Rightarrow \begin{array}{l}\max(\phi_1,\phi_2) \text{ encodes } \Omega_1 \cap \Omega_2\\ \max(\phi_1,\phi_2) \text{ encodes } \Omega_1 \cup \Omega_2\end{array}$$

  - ➡ We model both shape & topology change by simply varying the level set function