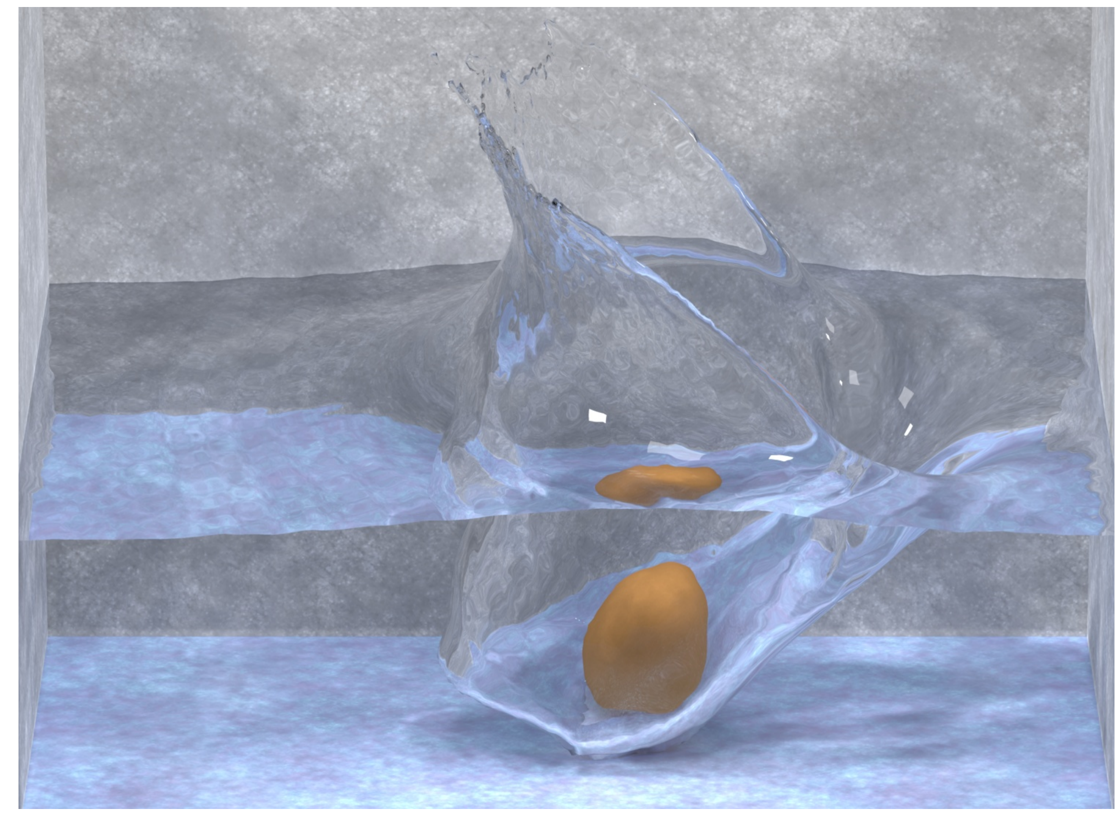
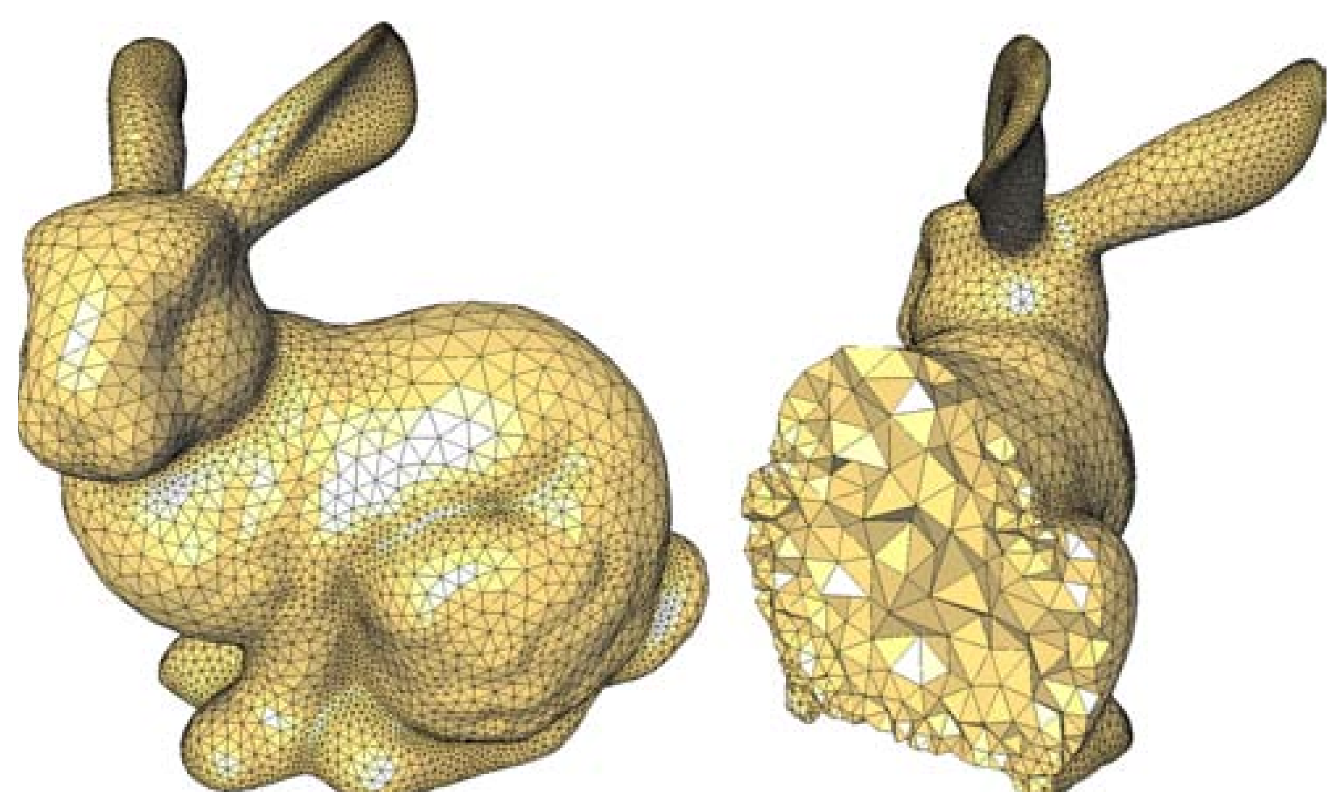
# *Discrete representations of geometric objects: Features, data structures and adequacy for dynamic simulation.*

## *Part I : Solid geometry*





# Upcoming topics

- Describe a number of discrete representations used to encode geometric objects for modeling and simulation purposes
    - Meshes
    - Implicit surfaces
    - Point clouds
- Discuss the features of these representations that are specific to simulation, as opposed to general geometry processing and rendering
    - Objects need to support *dynamic deformation*
    - Volumetric objects need internal structure
    - Discrete geometry needs to be *simulation-quality (well-conditioned)*

# Upcoming topics

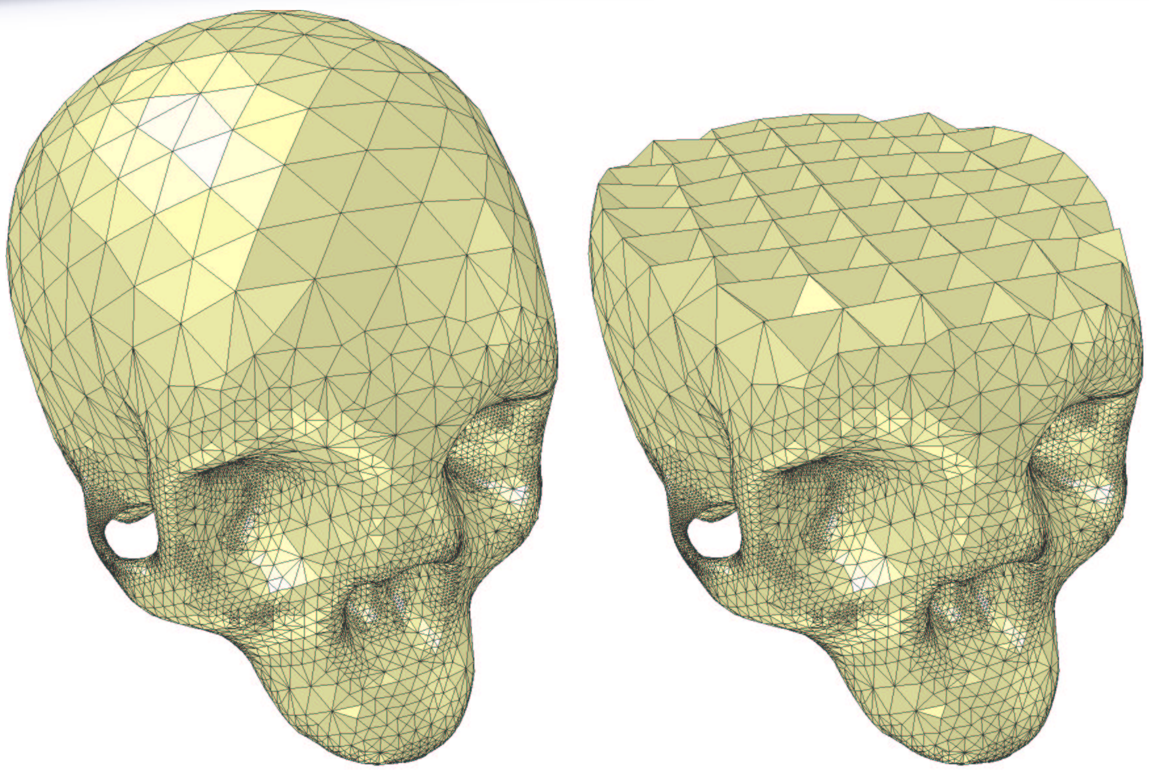
- Explain the features that make one representation better than another for certain tasks (e.g. meshes vs. implicit surfaces)
  - Static vs. dynamic topology (connectivity)
  - "Shape memory" and deformation drift
  - Regular, structured storage
  - Efficiency of geometric queries

# Upcoming topics

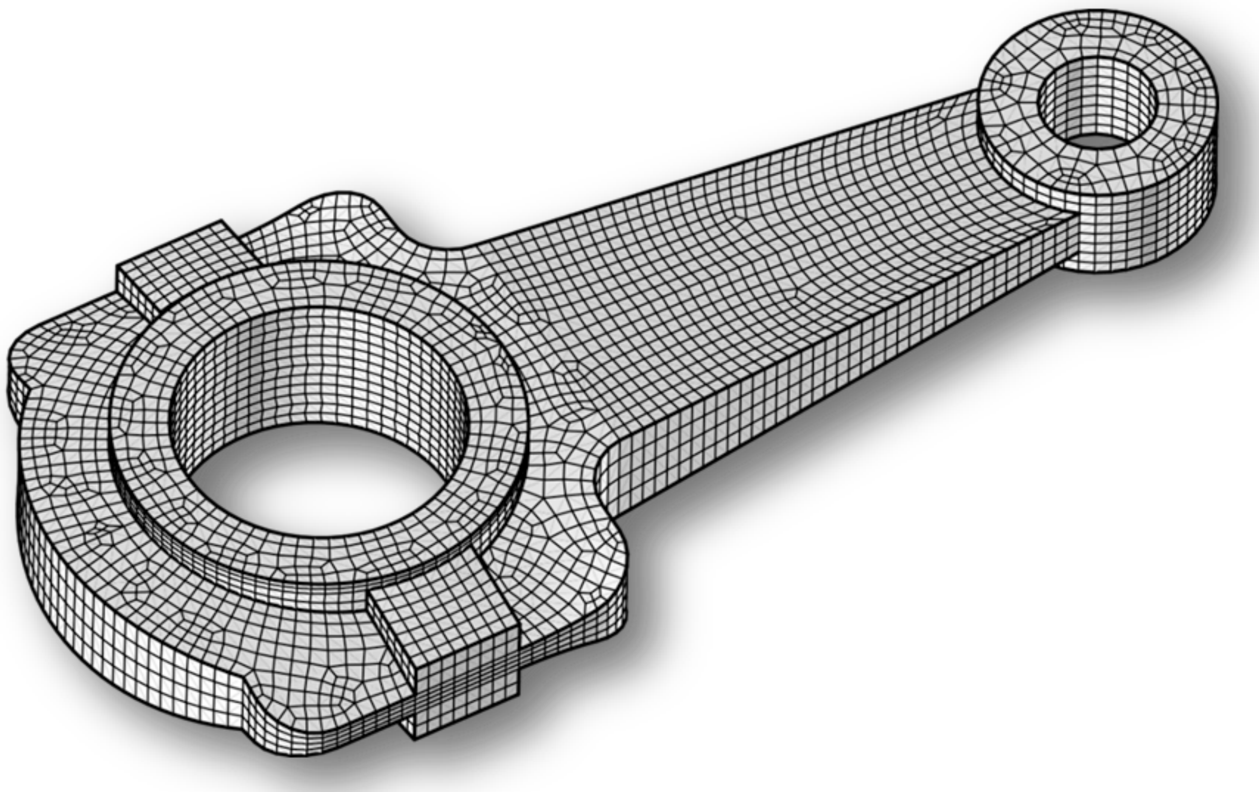
- Outline conversion methods between different geometric representations, e.g.
    - Tetrahedral meshing
    - Marching cubes, marching tetrahedra
    - MLS surface reconstruction, etc.

*Next topic : Introduction to PhysBAM data structures and scene layout*

# Discrete representation of solid geometry



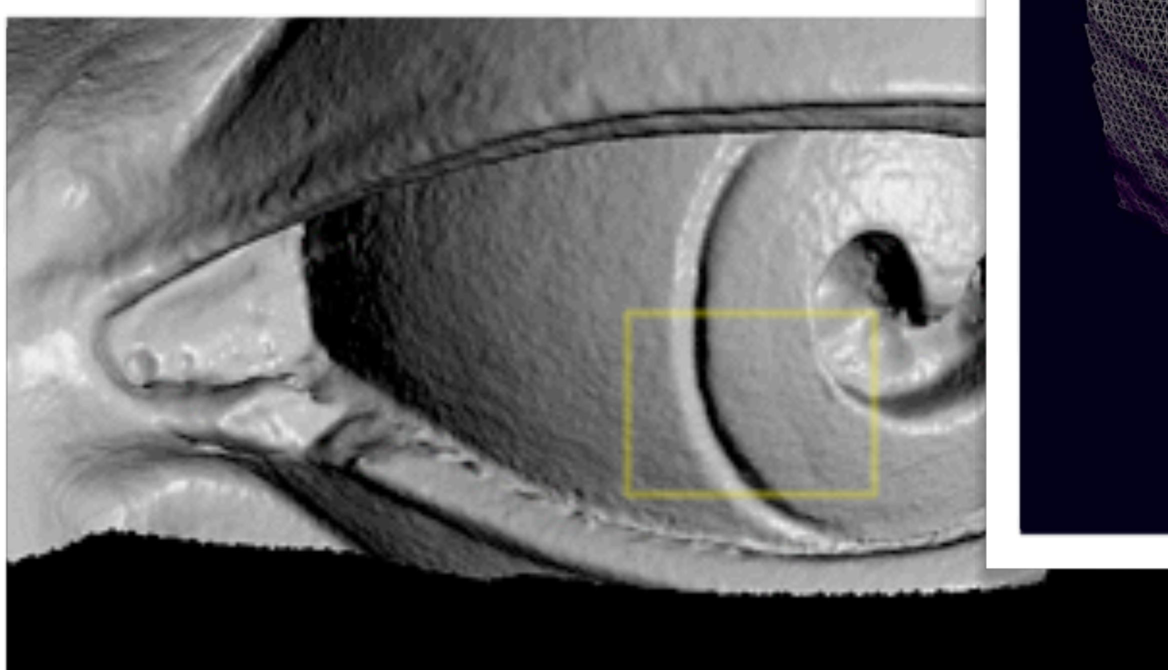
Tetrahedral meshes (volumetric)

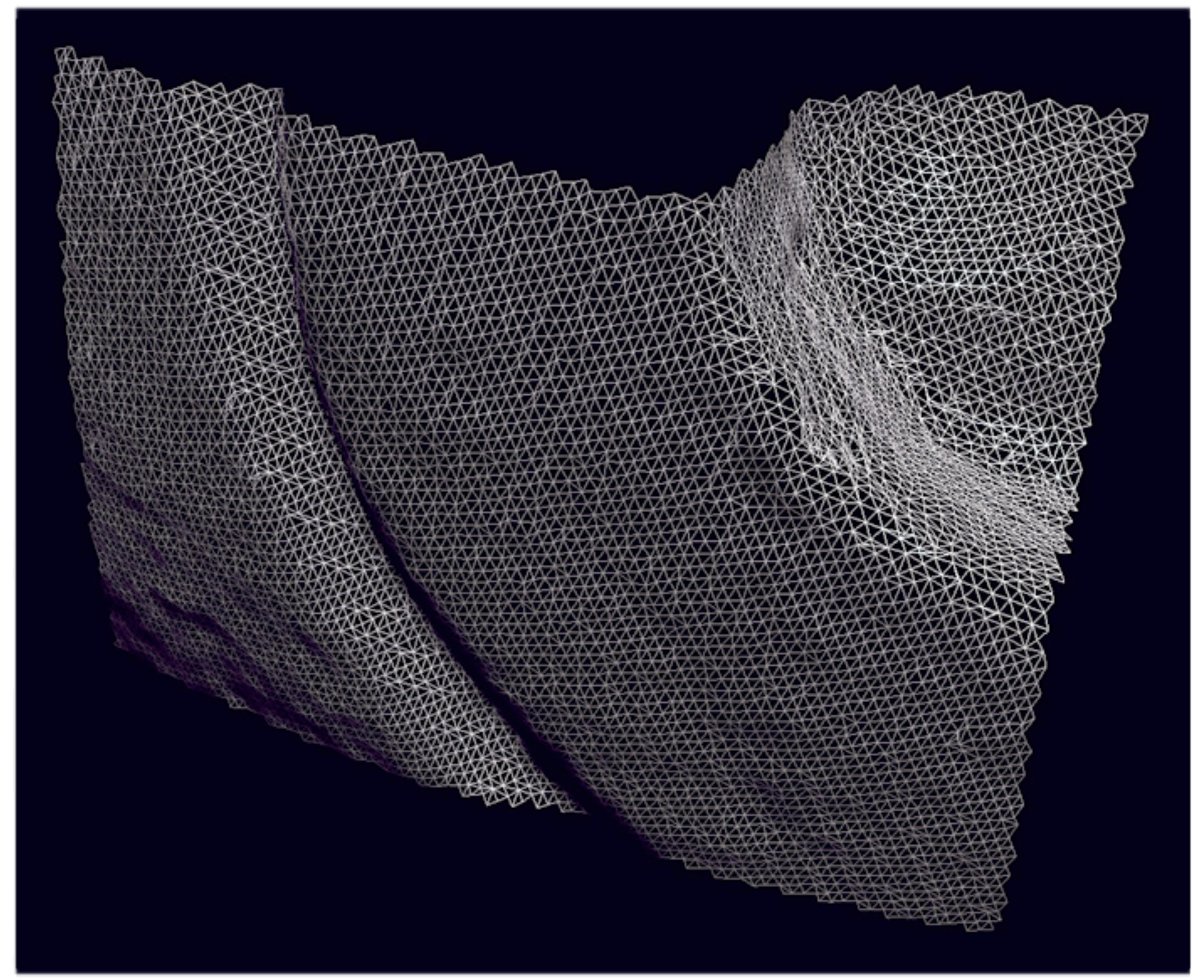


Hexahedral meshes (volumetric)

# Discrete representation of solid geometry

Triangular *surface* meshes
(not volumetric)





# Discrete representation of solid geometry

***"Meshed" geometry***
*(or just "geometry")*

→

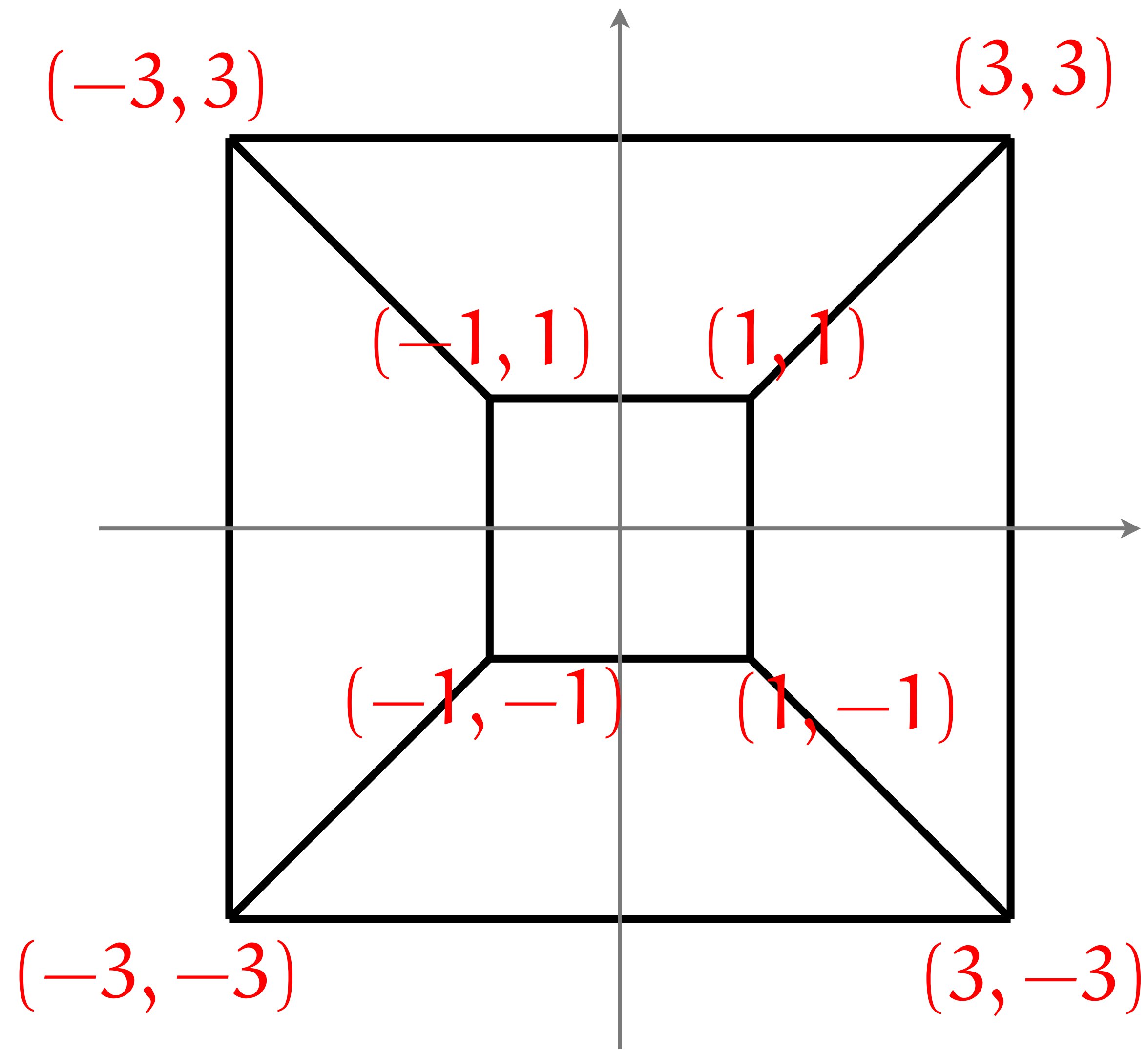
***Particles***

+

***Mesh***
*(topology/connectivity)*

# Discrete representation of solid geometry

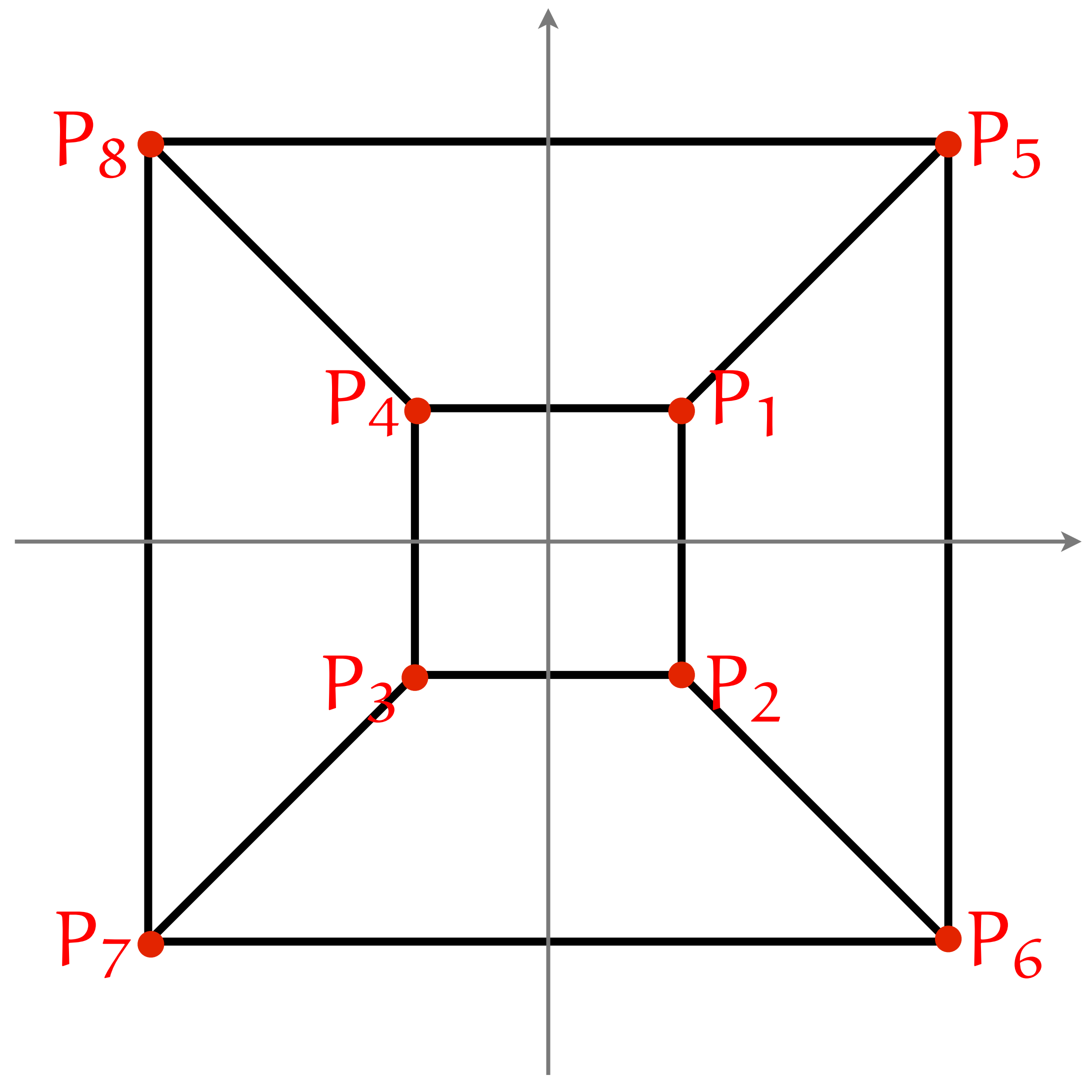
*Example:*
**A quadrilateral mesh**



# Discrete representation of solid geometry

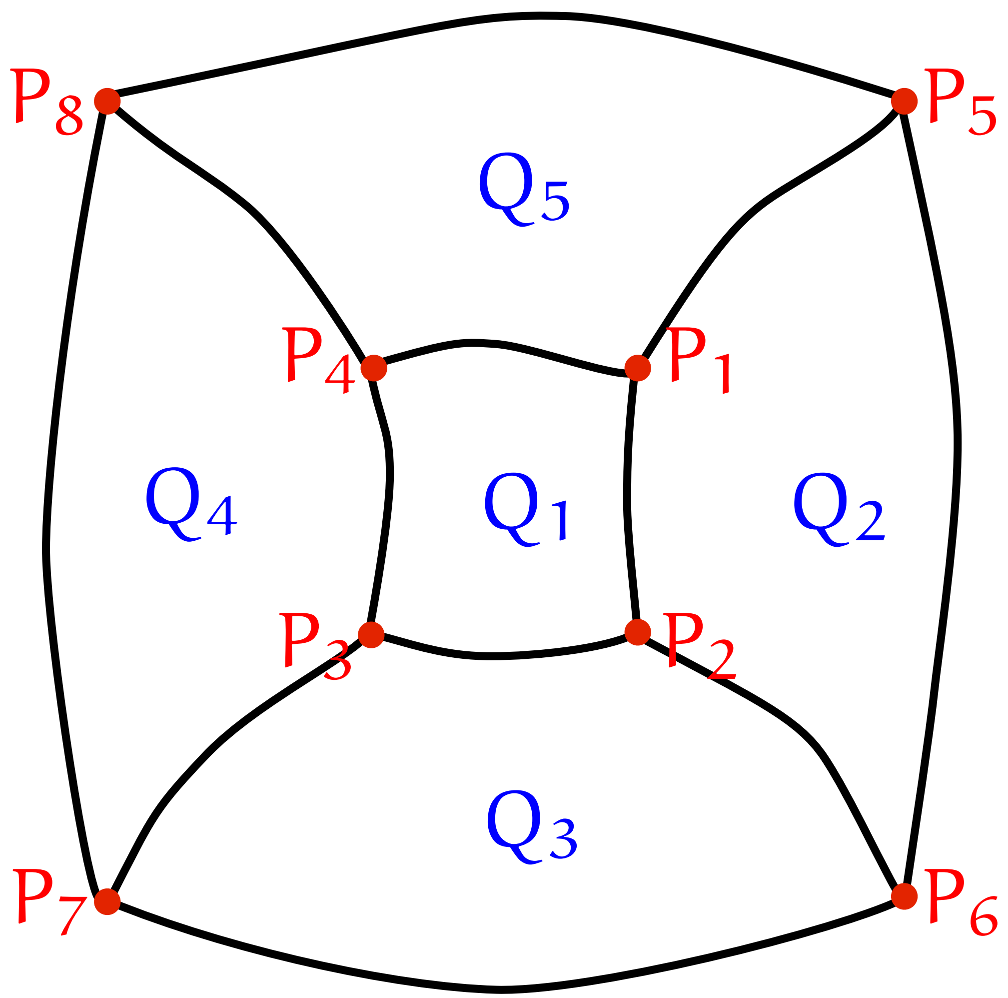
***Particle data structure***

| 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

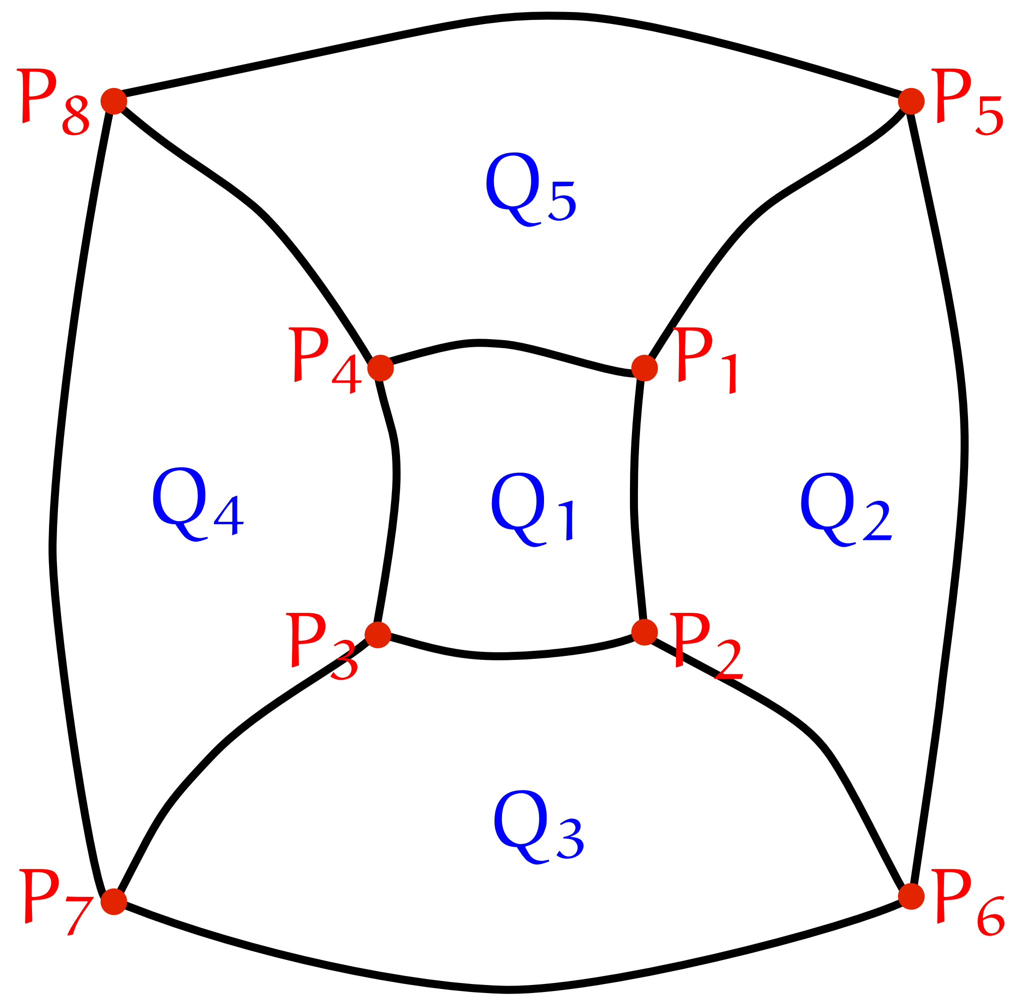
***Mesh data structure***



# 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

**_Why "particles"?_**
*(and not "points", "vertices", ...)*

| | |
|---|---|
| Physical attributes | ✓ Position<br>✓ Velocity<br>✓ Acceleration<br>✓ Force<br>✓ Mass, etc .... |
| Secondary attributes | ✓ Texture coordinates<br>✓ Color<br>✓ Translucency, etc ... |

# 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

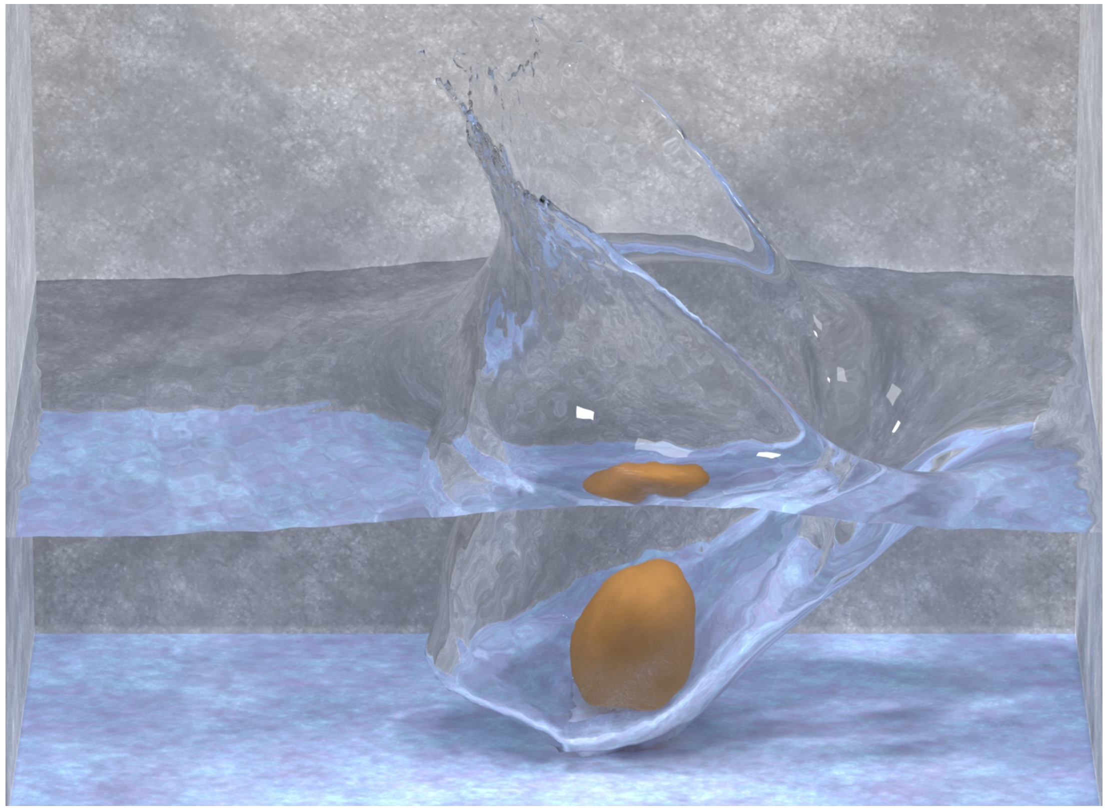
- Simulation algorithms typically stream different properties during different passes - separation improves bandwidth
- Easy to construct **subsets of attributes** *(e.g. for visualization)*

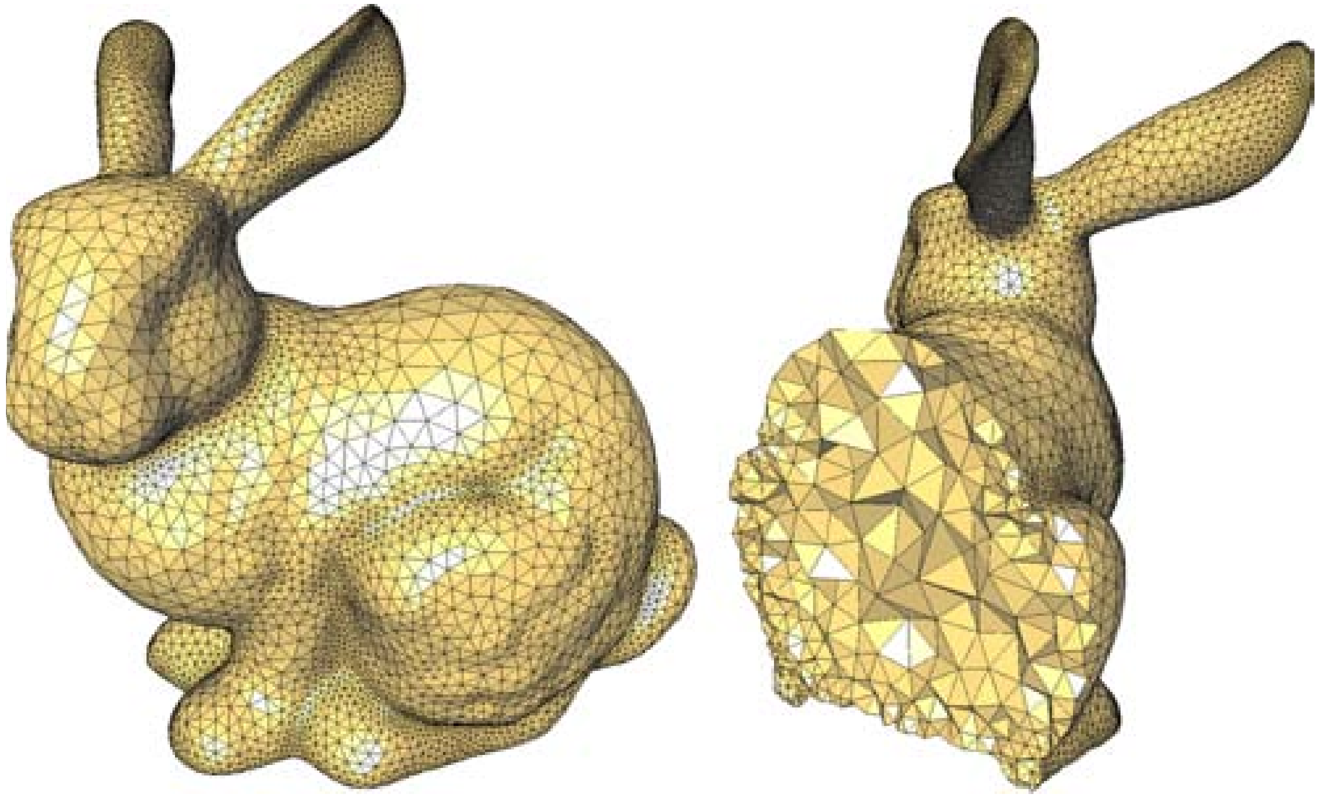
# Summary

- *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

# Discrete representations of geometric objects: Features, data structures and adequacy for dynamic simulation.

## Part II : 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

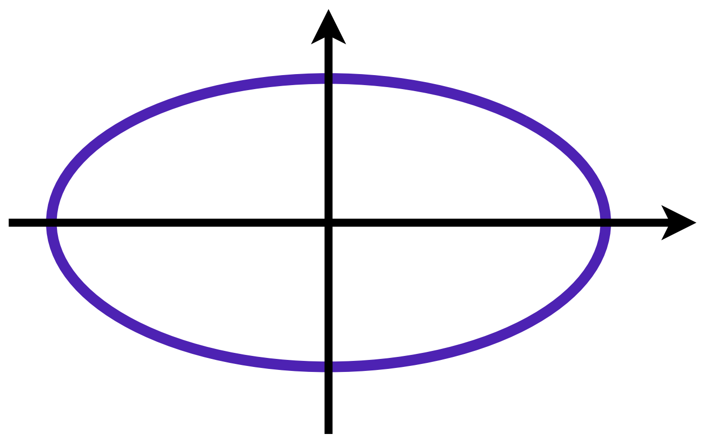
  - ✓ Such operations are difficult to encode with meshes, since they don't "split" or "merge" unless we force them to

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

- Familiar representations address *some* of these demands:

  ✓ e.g. Analytic equations

  ➡ For an ellipsis:



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

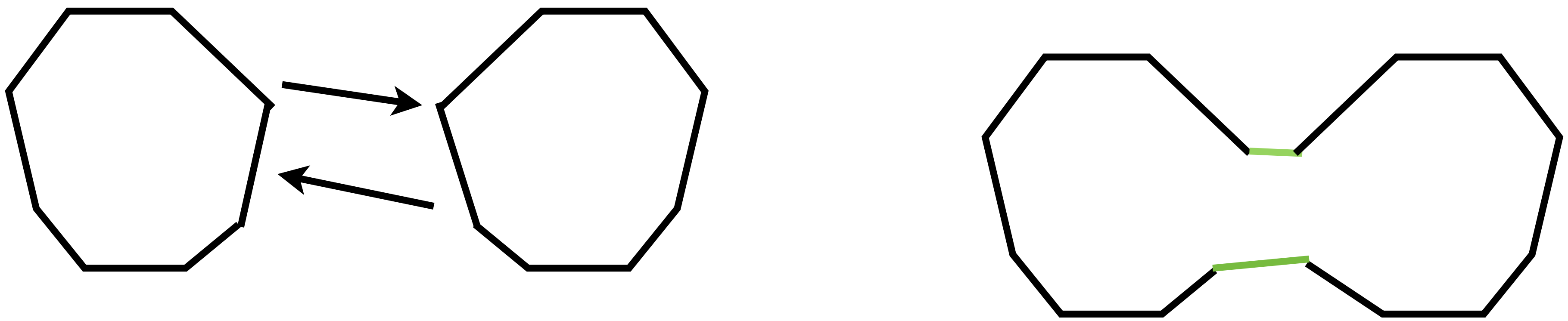
  ➡ Easy inside/outside tests

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

# 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

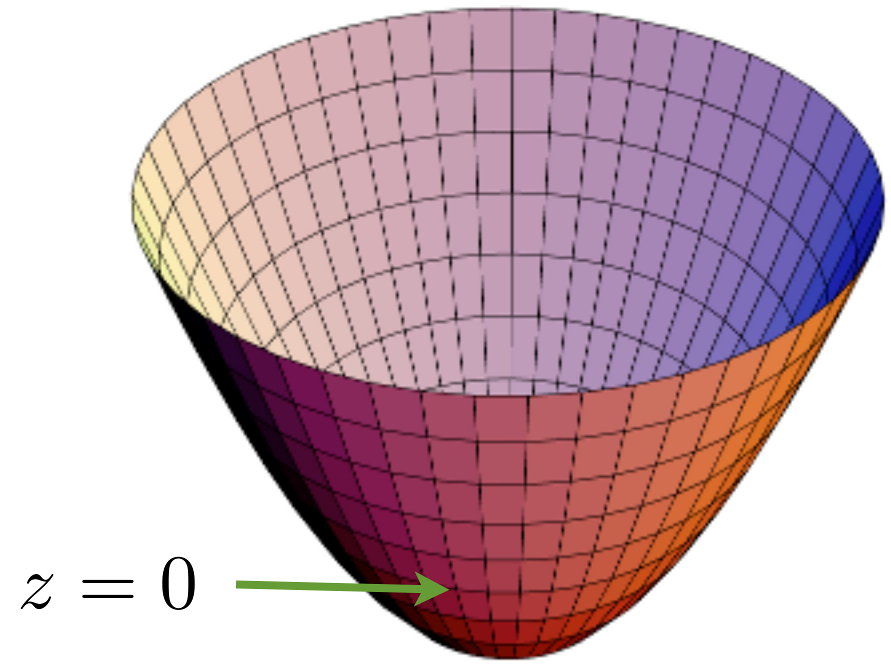
- Represent a curve in 2D (or, a surface in 3D) as the zero isocontour of a (continuous) function, i.e.

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

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