Computer Graphics I

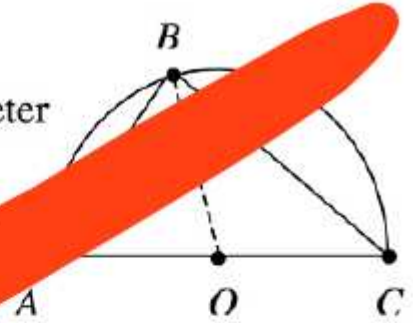
Lecture 4: Geometric representation & triangulation

Xiaopei LIU

School of Information Science and Technology
ShanghaiTech University

What is geometry?

TH 9.5. Let ΔABC be inscribed in a semicircle with diameter $\overline{A}\ \overline{C}$.
Then $\angle ABC$ ngle.

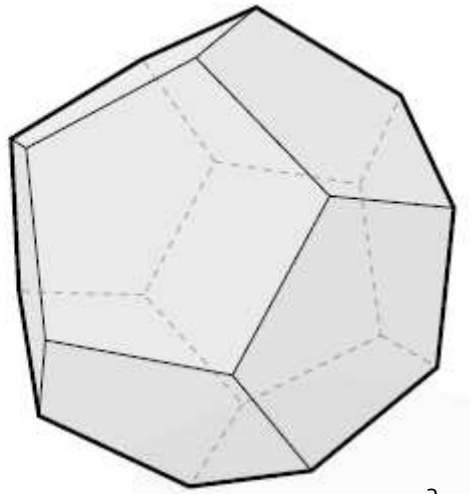


Proof:

Statement	
1. Draw radius OB. Then $OB = OC =$	Given
2. $m\angle OBC = m\angle BCA$ $m\angle OBA = m\angle BAC$	sceles Triangle Theorem
3. $m\angle ABC = m\angle OBA +$	3. Angl ostulate
4. $m\angle ABC + m\angle BC$ $\angle BAC = 180$	4. The sum gles of a triangle is 180
5. $m\angle ABC +$ $+ m\angle OBA = 180$	5. Substitution (li
6. $2\,m\angle$ 180	6. Substitution (line 3)
7. m $= 90$	7. Division Property of Equali
8. $\angle ABC$ is a right angle	8. Definition of Right Angle

What is geometry?

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



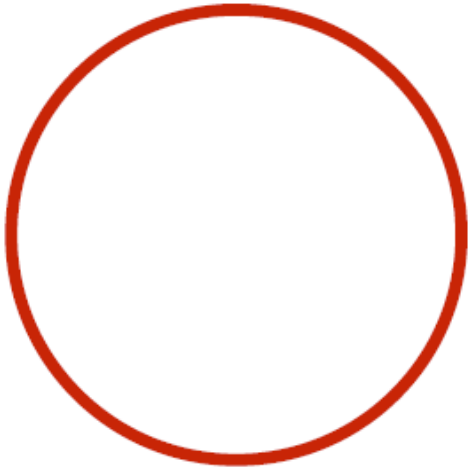
How can we describe geometry?

IMPLICIT

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

LINGUISTIC

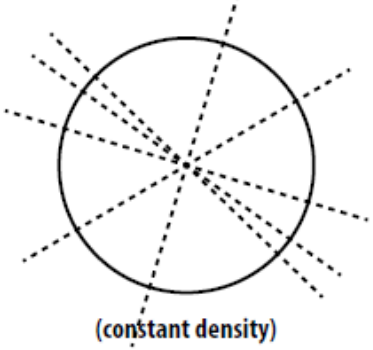
"unit circle"



EXPLICIT

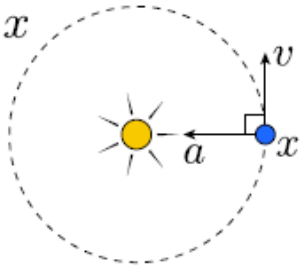
$$(\underbrace{\cos\theta}_{x}, \underbrace{\sin\theta}_{y})$$

TOMOGRAPHIC



DYNAMIC

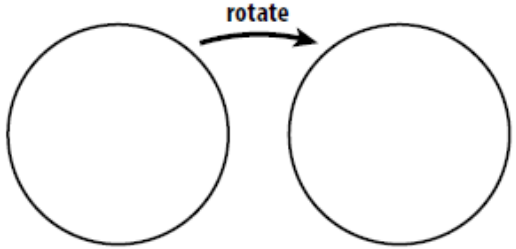
$$\frac{d^2}{dt^2}x = -x$$



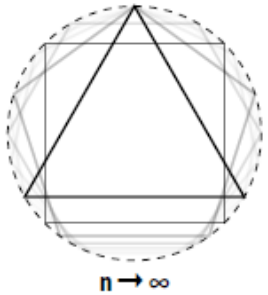
CURVATURE

$$\kappa = 1$$

SYMMETRIC



DISCRETE

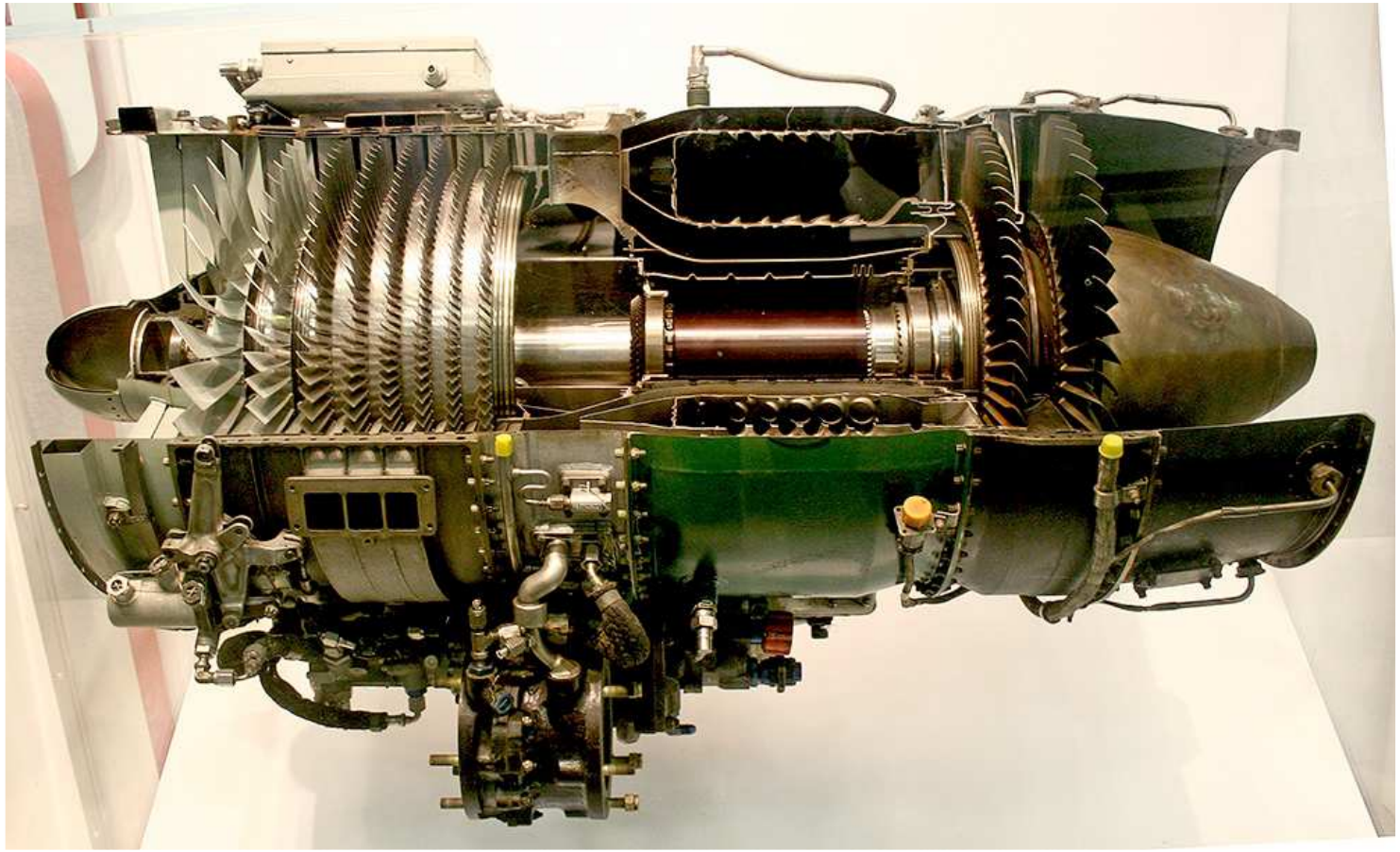


The best way to encode geometry on computer?

- **Examples of geometry**

The best way to encode geometry on computer?

- **Examples of geometry**



The best way to encode geometry on computer?

- **Examples of geometry**

The best way to encode geometry on computer?

- **Examples of geometry**

The best way to encode geometry on computer?

- **Examples of geometry**

The best way to encode geometry on computer?

- **Examples of geometry**

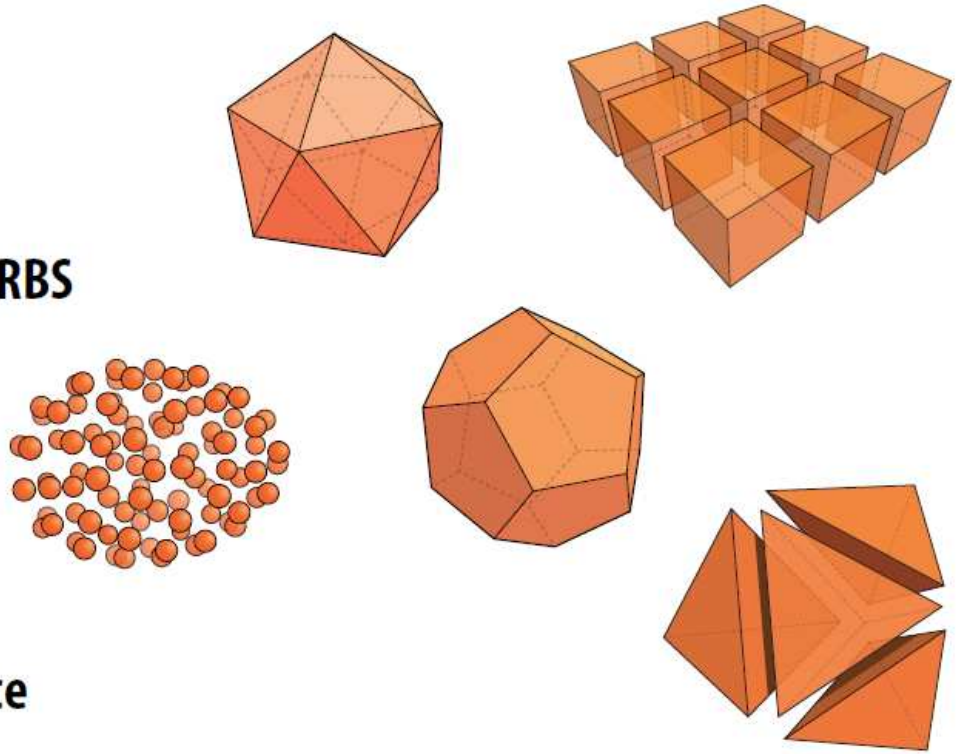
The best way to encode geometry on computer?

- **No "best" choice—geometry is hard!**

> ***"I hate meshes.***
> ***I cannot believe how hard this is.***
> ***Geometry is hard."***
>
> **—David Baraff**
> **Senior Research Scientist**
> **Pixar Animation Studios**

Many ways to digitally encode geometry

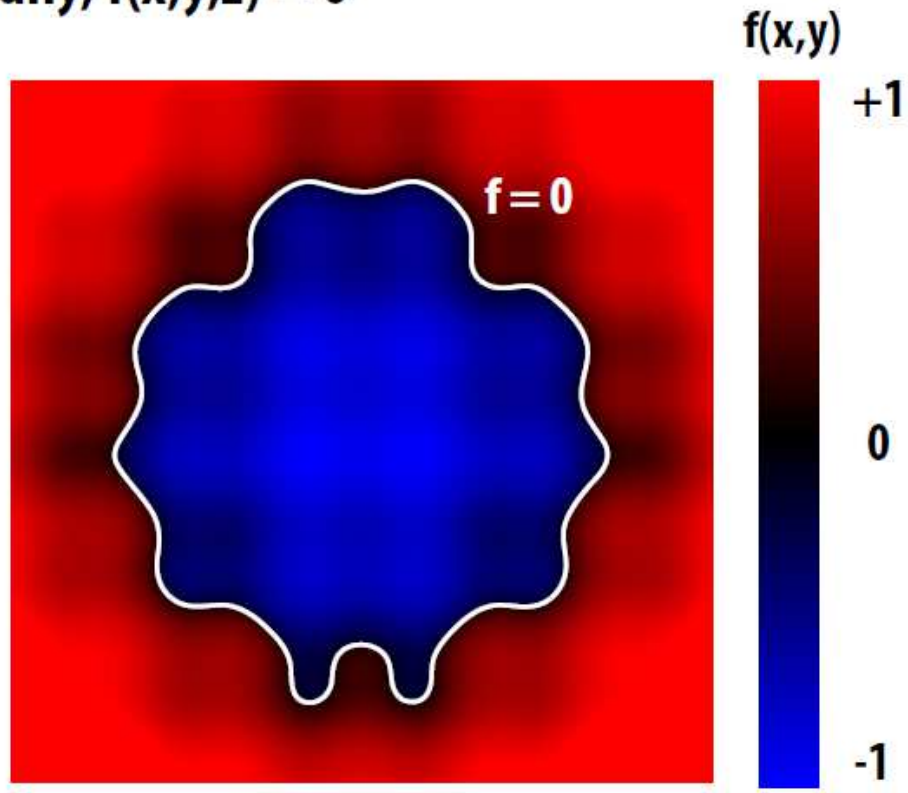
- **Many ways to digitally encode geometry**
  - **EXPLICIT**
    - **point cloud**
    - **polygon mesh**
    - **subdivision, NURBS**
    - **L-systems**
    - **…**
  - **IMPLICIT**
    - **level set**
    - **algebraic surface**
    - **…**
  - **Each choice best suited to a different task/type of geometry**



1. Different geometric representations

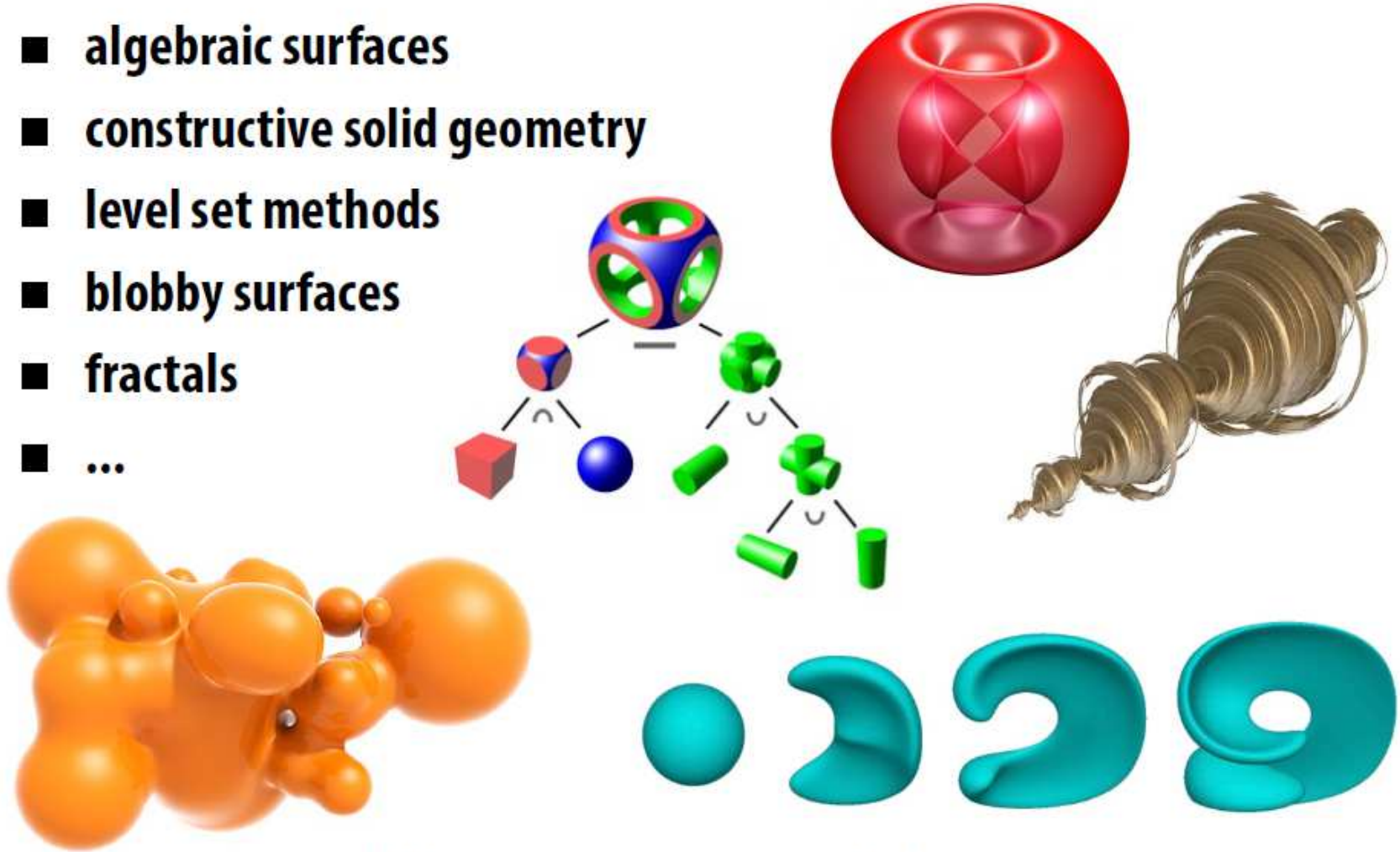
"Implicit" representations of geometry

- Points aren't known directly, but satisfy some relationship
- E.g., unit sphere is all points x such that $x^2+y^2+z^2=1$
- More generally, $f(x,y,z) = 0$



Many implicit representations in graphics

- algebraic surfaces
- constructive solid geometry
- level set methods
- blobby surfaces
- fractals
- ...

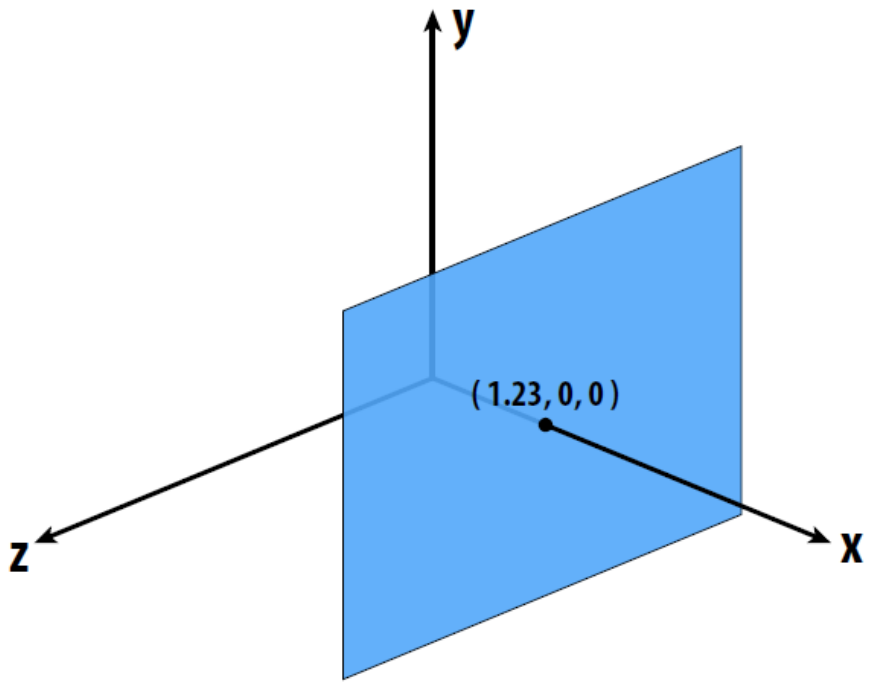


(Will see some of these a bit later.)

Prons. & cons. of implicit surfaces

- I'm thinking of an implicit surface f(x,y,z)=0. Find *any point on it.*

My function was f(x,y,z) = -1.23 (a plane): (a special case)



Implicit surfaces make some tasks hard (like sampling).

Prons. & cons. of implicit surfaces

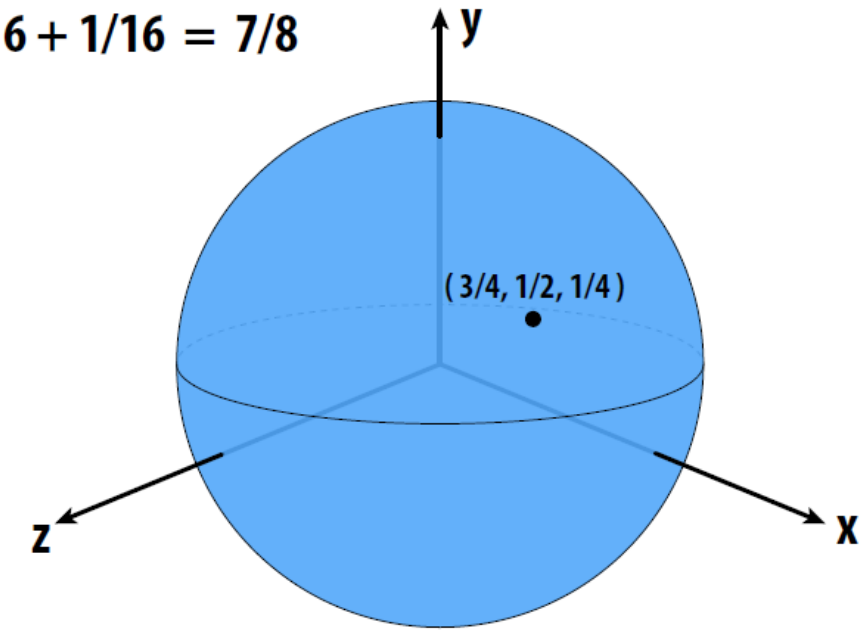
- I have a new surface $f(x,y,z) = x^2 + y^2 + z^2 - 1$.
  I want to see if a point is *inside it*.

How about the point (3/4, 1/2, 1/4)?

$9/16 + 4/16 + 1/16 = 7/8$

$7/8 < 1$

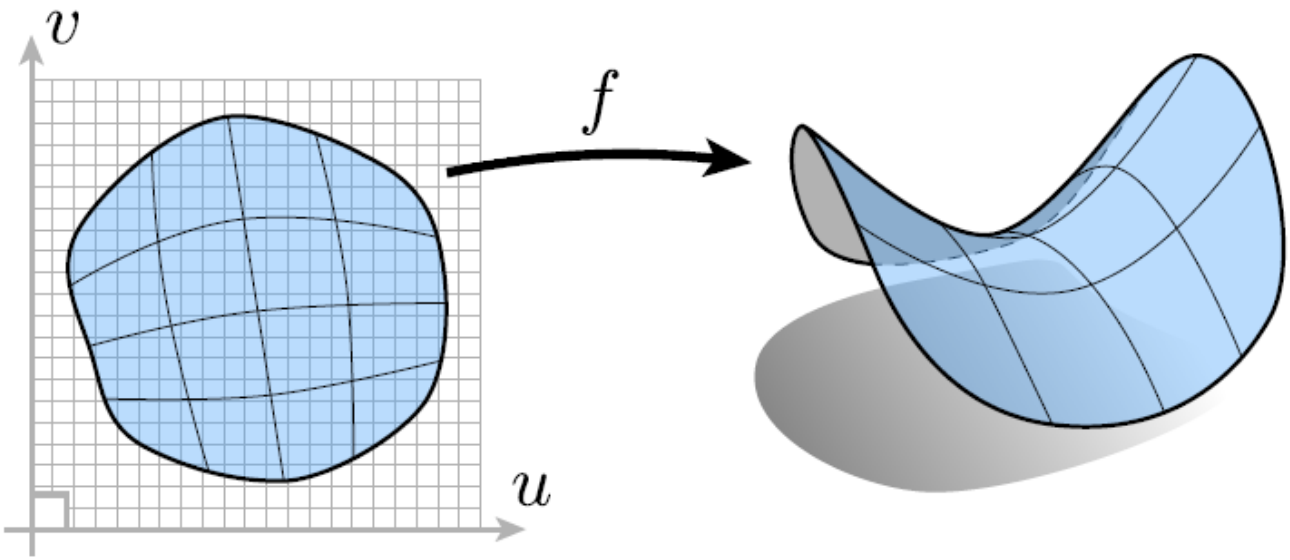
YES.



Implicit surfaces make other tasks easy (like inside/outside tests).

"Explicit" representations of geometry

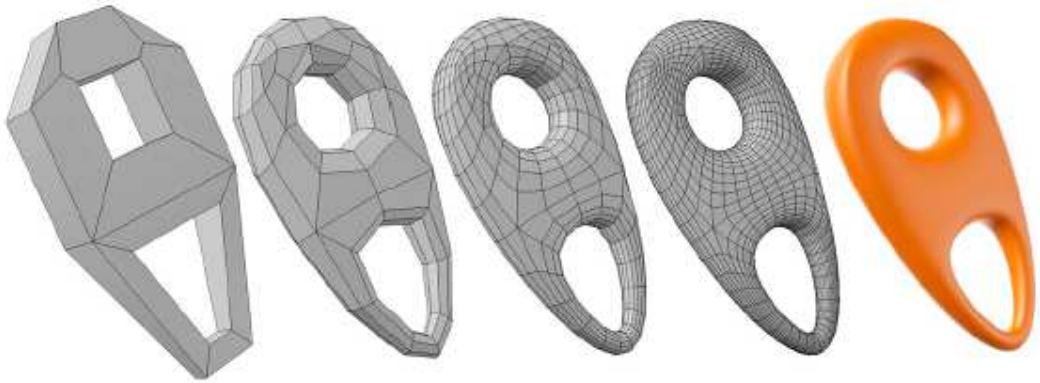
- **All points are given directly**
- **E.g., points on sphere are** $(\cos(u)\sin(v), \sin(u)\sin(v), \cos(v)),$
  $\text{for } 0 \leq u < 2\pi \text{ and } 0 \leq v \leq \pi$
- **More generally:** $f : \mathbb{R}^2 \to \mathbb{R}^3; (u, v) \mapsto (x, y, z)$

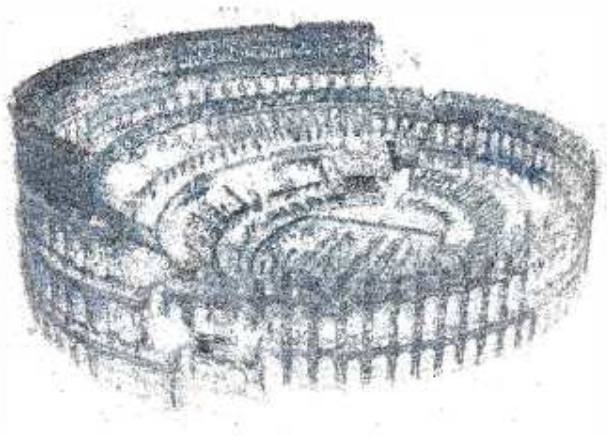


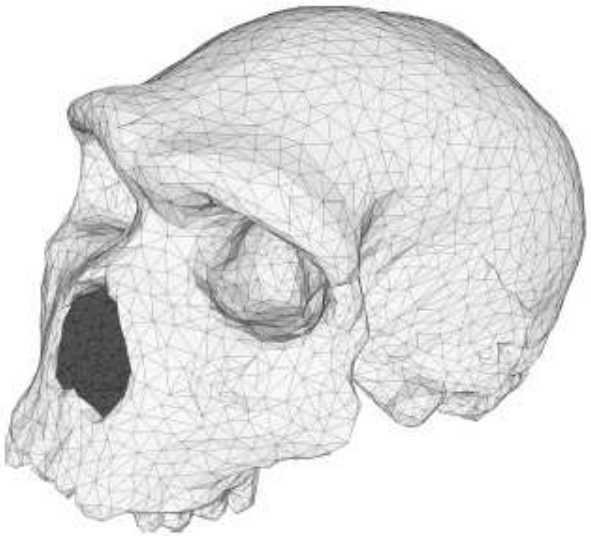
- **(Might have a bunch of these maps, e.g., one per triangle.)**

Many explicit representations in graphics

- triangle meshes
- polygon meshes
- subdivision surfaces
- NURBS
- point clouds
- ...







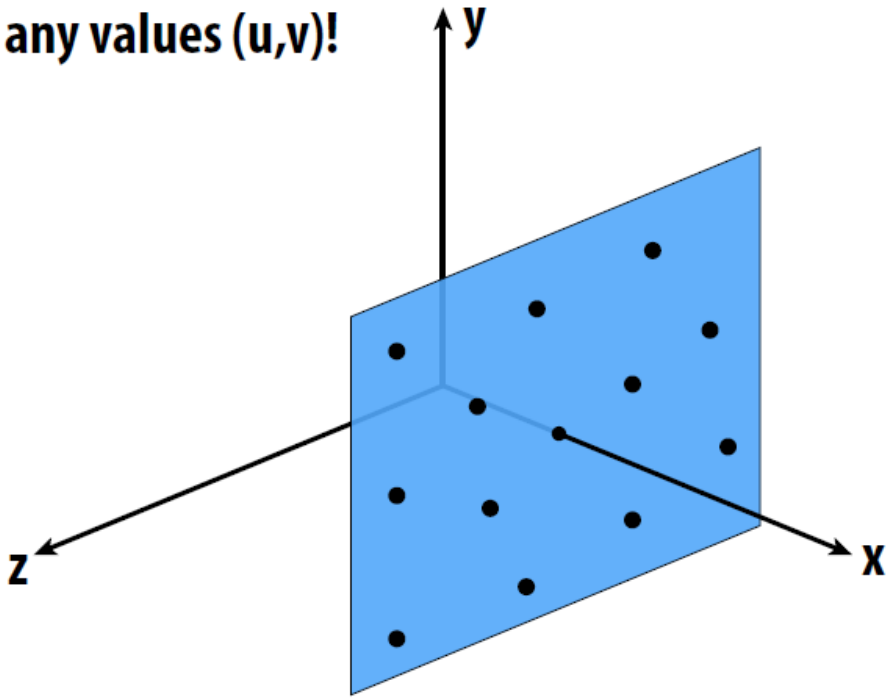
(Will see some of these a bit later.)

Prons. & cons. of explicit surfaces

- Sampling an explicit surface

My surface is f(u, v) = (1.23, u, v).

Just plug in any values (u,v)!



Explicit surfaces make some tasks easy (like sampling).

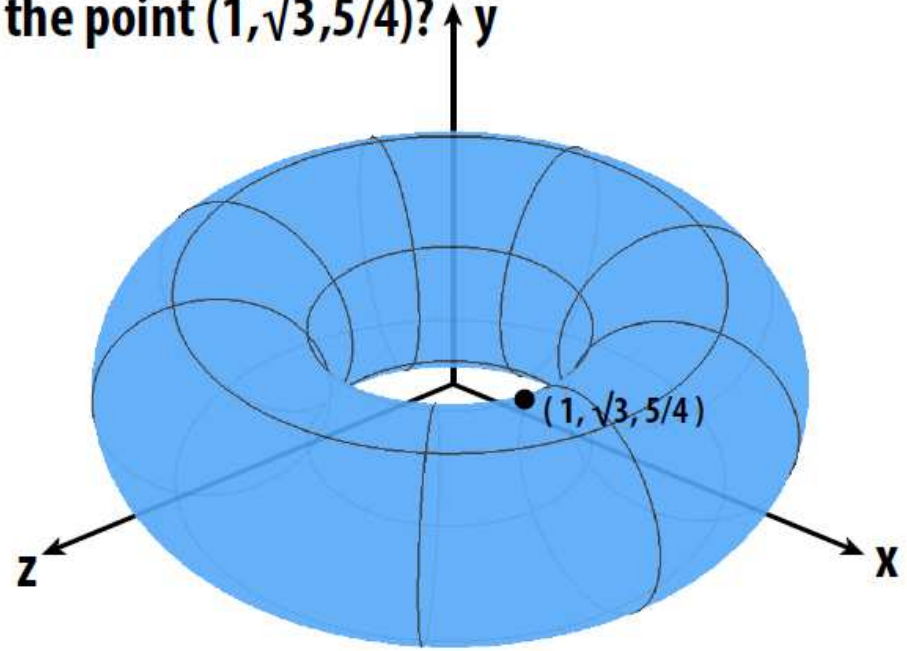
Prons. & cons. of explicit surfaces

- **Check if this point is inside the torus**

My surface is f(u,v) = (2+cos(u))cos(v), 2+cos(u))sin(v), sin(u))

How about the point (1,√3,5/4)?

...NO!



Explicit surfaces make other tasks hard (like inside/outside tests).

Conclusion

- Some representations work better than others—depends on the task!
- Different representations will also be better suited to different types of geometry.
- Let's take a look at some common representations used in computer graphics.

Different representations

- **Algebraic surfaces (implicit)**
  - **Surface is zero set of a polynomial in x, y, z ("algebraic variety")**
  - **Examples:**

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

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

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

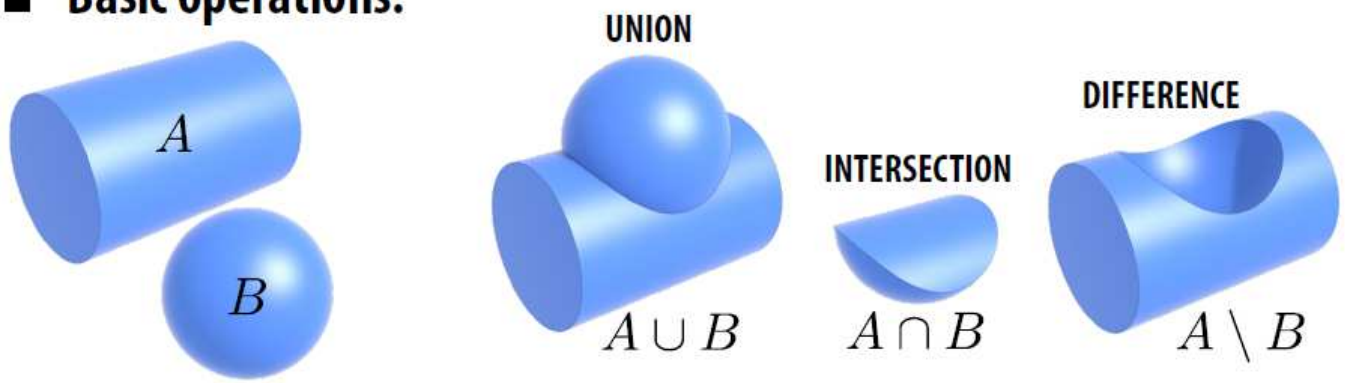
$x^2z^3+\frac{9y^2z^3}{80}$

  - **What about more complicated shapes?**

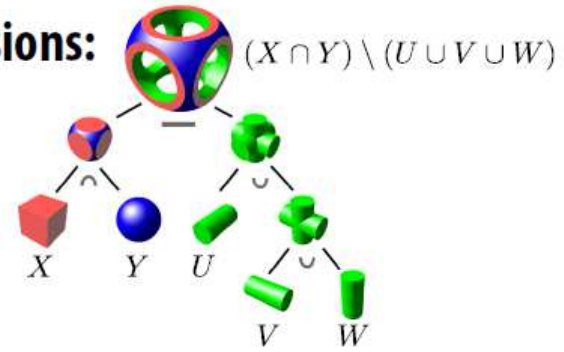
  - **Very hard to come up with polynomials!**

Different representations

- **Constructive solid geometry (implicit)**
  - **Build more complicated shapes via Boolean operations**
  - **Basic operations:**

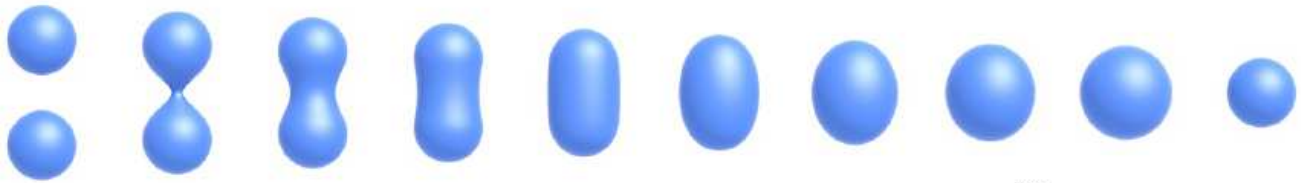


  - **Then chain together expressions:**



Different representations

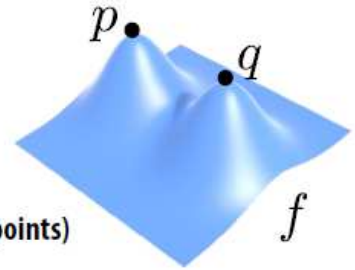
- **Blobby surfaces (implicit)**
  - **Instead of Booleans, gradually blend surfaces together:**

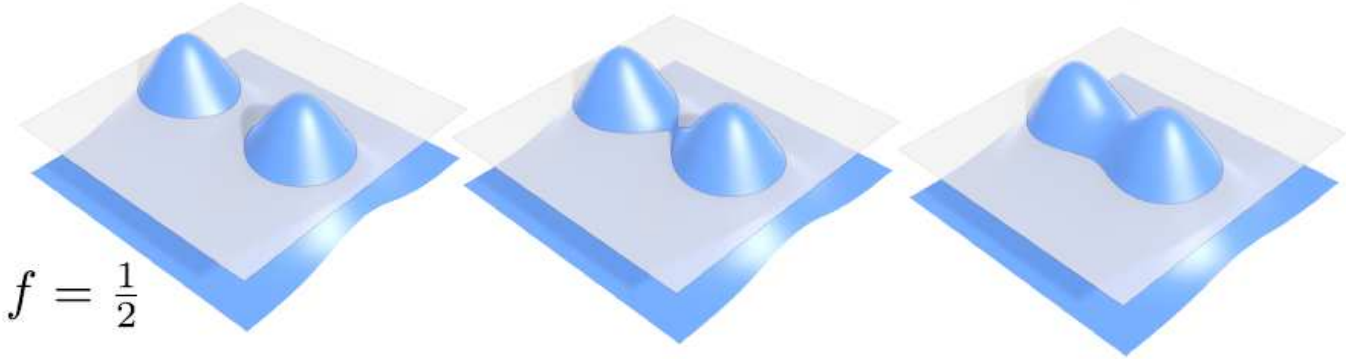


  - **Easier to understand in 2D:**

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

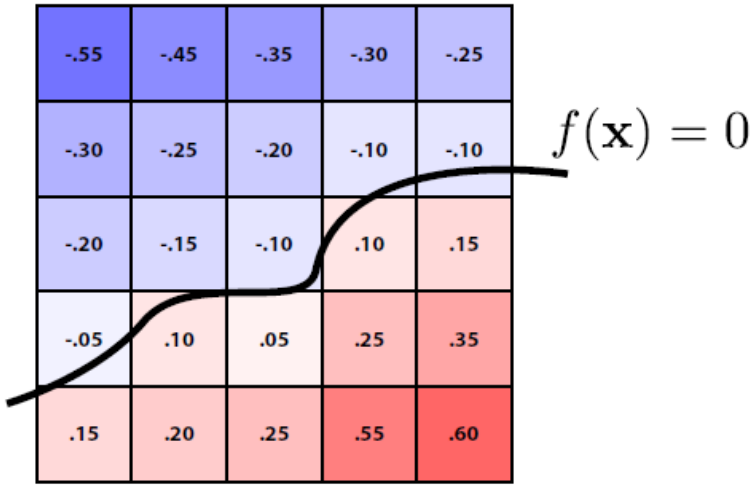
$f := \phi_p + \phi_q$ (Sum of Gaussians centered at different points)





Different representations

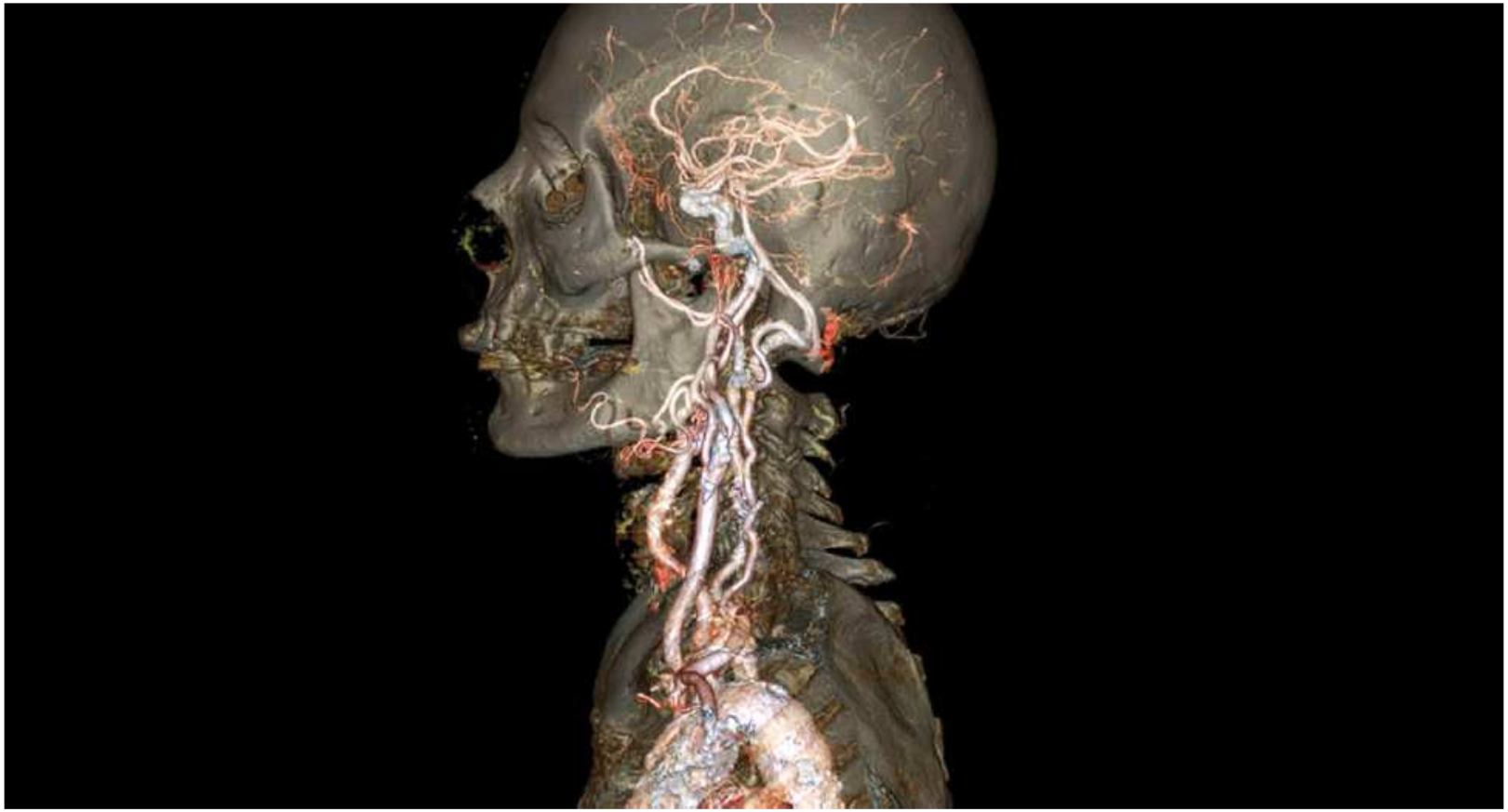
- **Level set methods (implicit)**
  - Implicit surfaces have some nice features (e.g., merging/splitting)
  - But, hard to describe complex shapes in closed form
  - Alternative: store a grid of values approximating function



  - Surface is found where *interpolated* values equal zero
  - Provides much more explicit control over shape (like a texture)

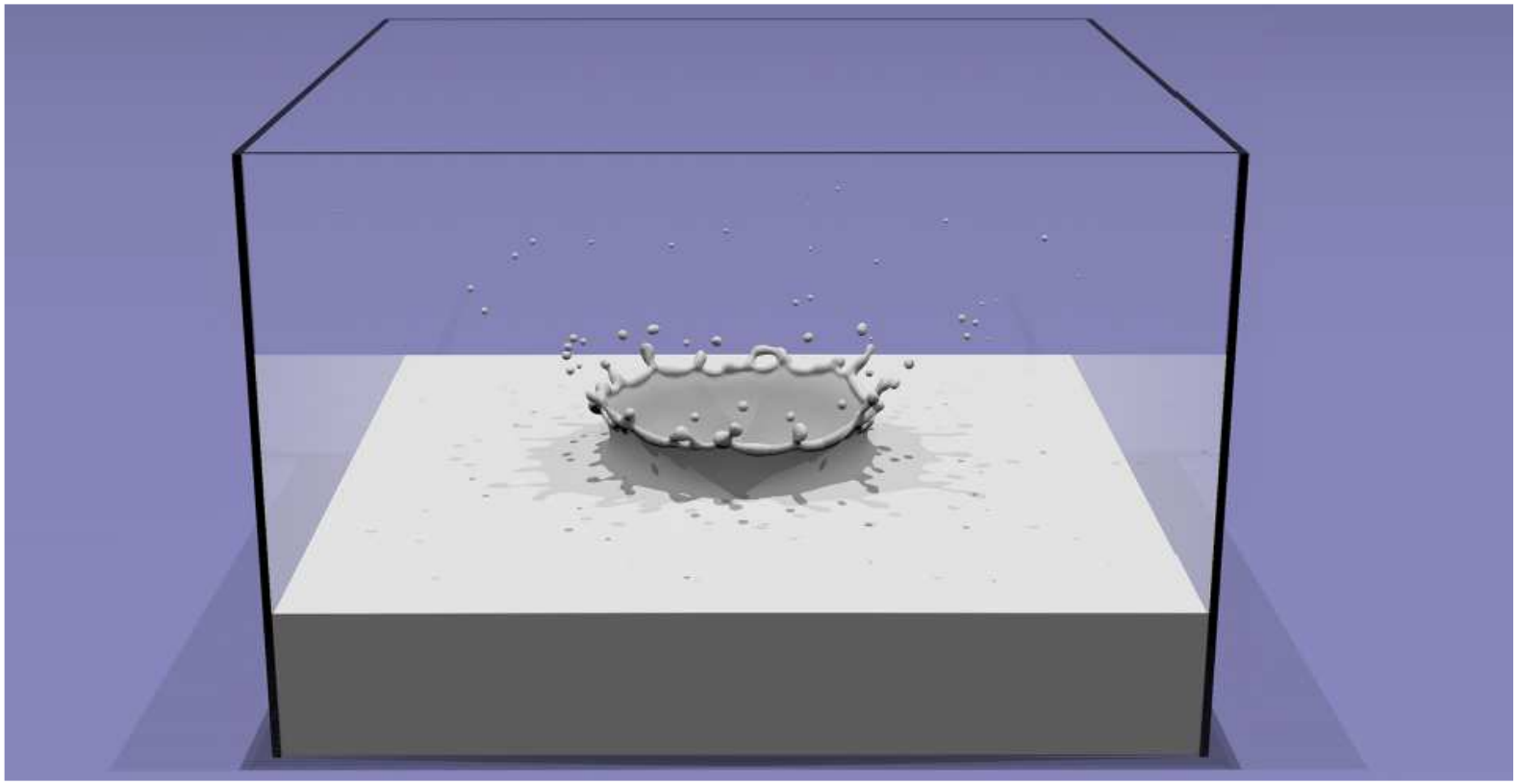
Different representations

- **Level sets from medical data (CT, MRI, etc.)**
  - Level sets encode, e.g., constant tissue density



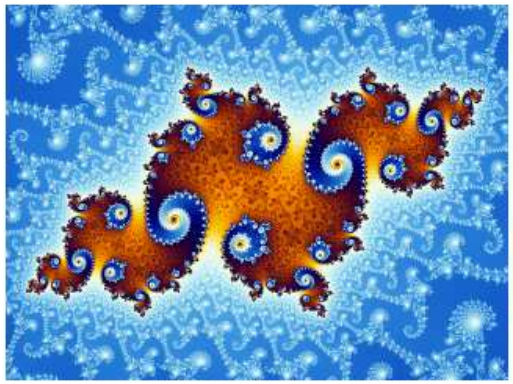
Different representations

- **Level sets in physical simulation**
  - Level set encodes distance to air-liquid boundary



Different representations

- **Fractals (implicit)**
  - No precise definition; exhibit self-similarity, detail at all scales
  - New "language" for describing natural phenomena
  - Hard to control shape!



Different representations

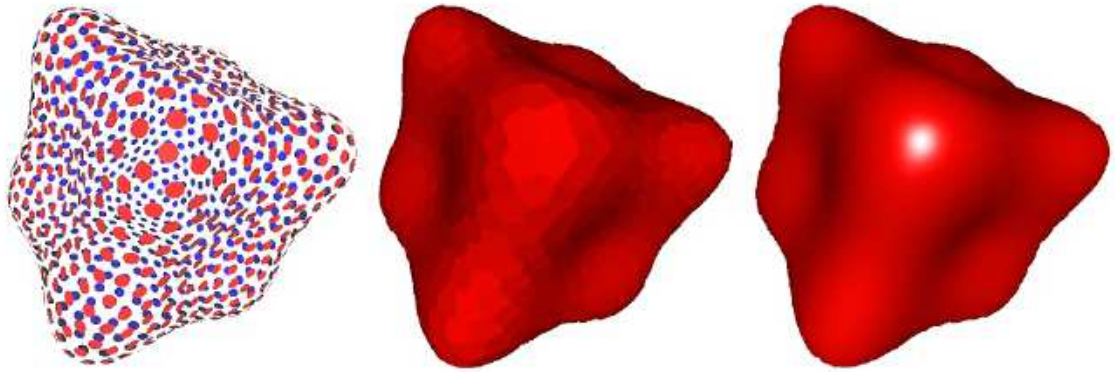
- **Iterated function systems**

Different representations

- Implicit representations - pros & cons
  - Pros:
    - description can be very compact (e.g., a polynomial)
    - easy to determine if a point is in our shape (just plug it in!)
    - other queries may also be easy (e.g., distance to surface)
    - for simple shapes, exact description/no sampling error
    - easy to handle changes in topology (e.g., fluid)
  - Cons:
    - expensive to find all points in the shape (e.g., for drawing)
    - *very difficult to model complex shapes*

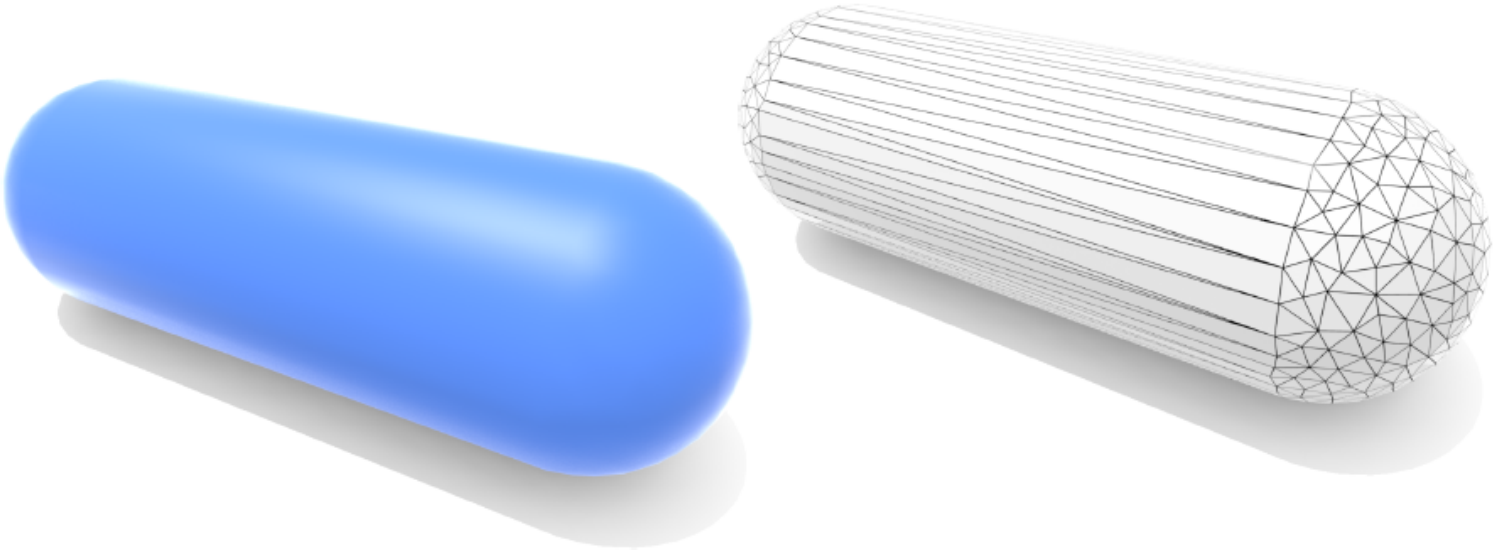
Different representations

- **Point cloud (explicit)**
  - Easiest representation: list of points (x,y,z)
  - Often augmented with *normals*
  - Easily represent any kind of geometry
  - Useful for LARGE datasets (>>1 point/pixel)
  - Difficult to draw in undersampled regions
  - Hard to do processing / simulation



Different representations

- Polygon mesh (explicit)
  - Store vertices *and* polygons (most often triangles or quads)
  - Easier to do processing/simulation, adaptive sampling
  - More complicated data structures
  - Perhaps most common representation in graphics



Different representations

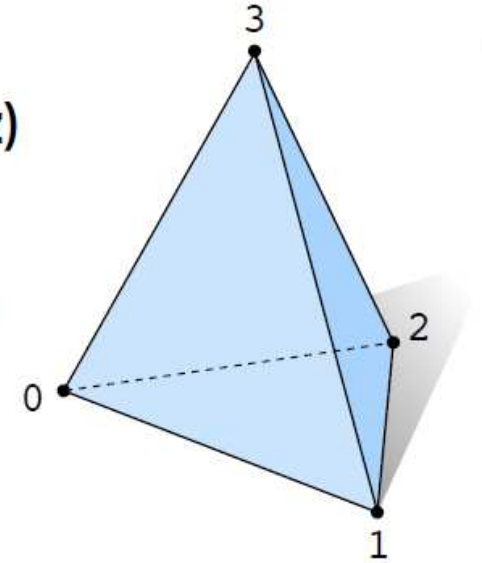
- **Triangle mesh (explicit)**
  - Store vertices as triples of coordinates (x,y,z)
  - Store triangles as triples of indices (i,j,k)
  - E.g., tetrahedron:

VERTICES

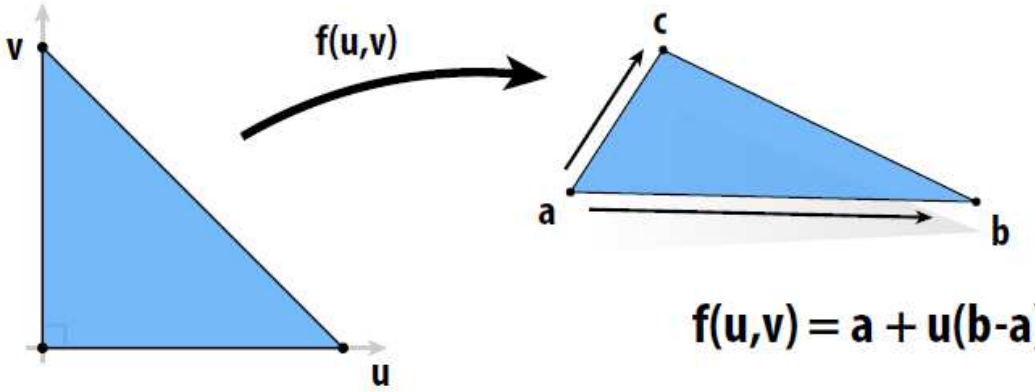
	x	y	z
0:	-1	-1	-1
1:	1	-1	1
2:	1	1	-1
3:	-1	1	1

TRIANGLES

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



  - Can think of triangle as *affine* map from plane into space:



$$f(u,v) = a + u(b-a) + v(c-a)$$

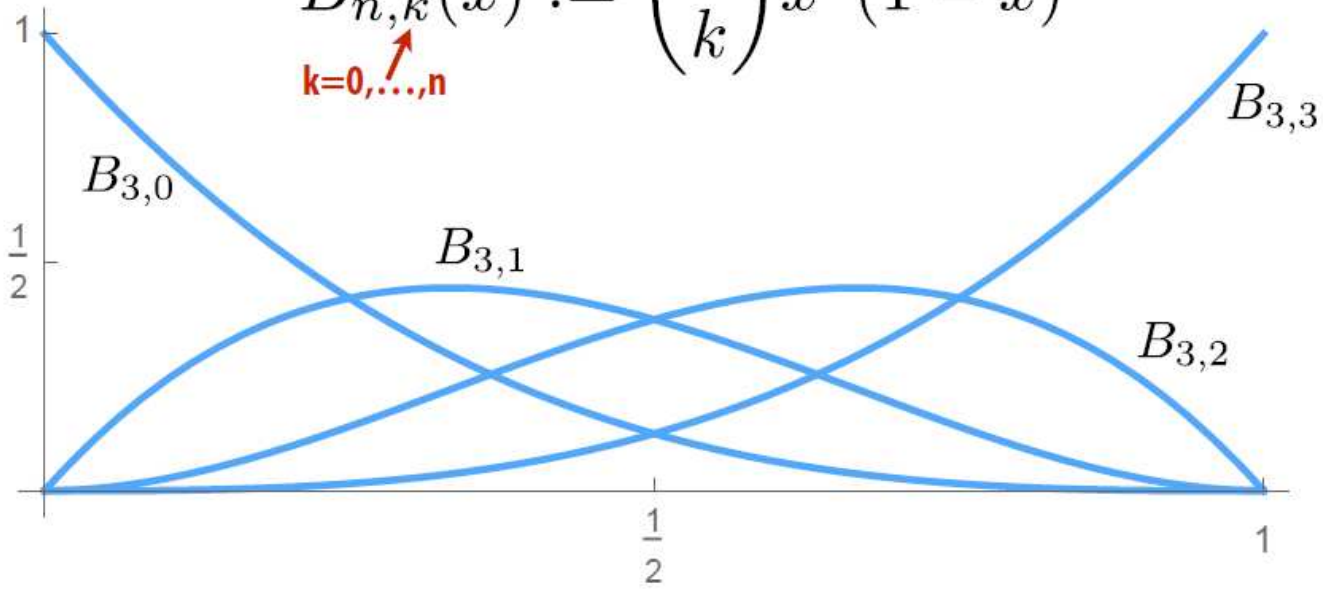
Different representations

- **Bernstein basis**
  - Why limit ourselves to just affine functions?
  - More flexibility by using higher-order polynomials
  - Instead of usual basis (1, x, x^2, x^3, ...), use Bernstein basis:

degree $0 \le x \le 1$ "n choose k"

$$B_{n,k}(x) := \binom{n}{k} x^k (1-x)^{n-k}$$

$k = 0, \ldots, n$



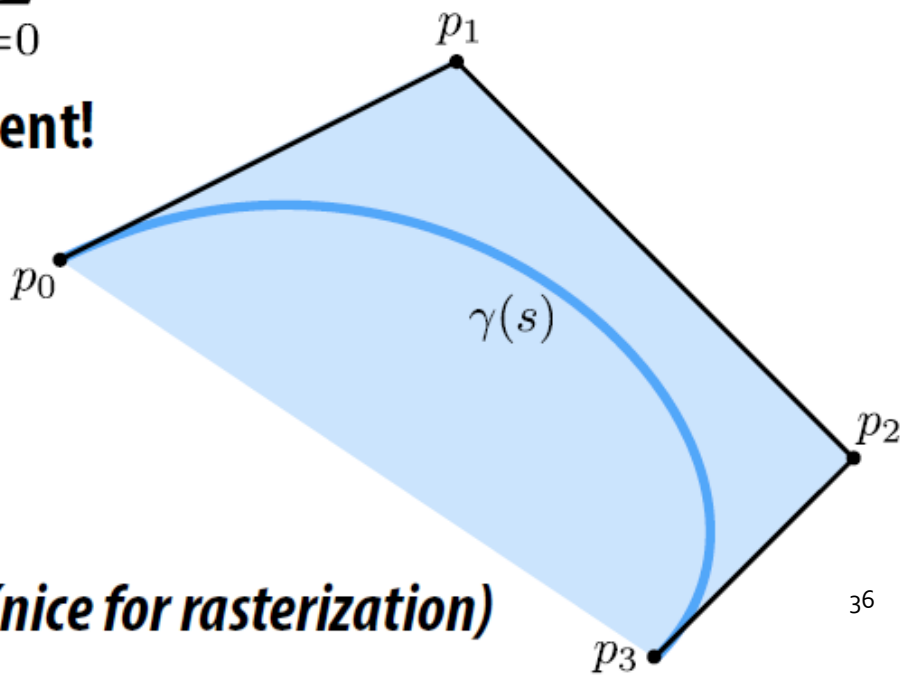
Different representations

- **Bézier curves (explicit)**
  - **A Bézier curve is a curve expressed in the Bernstein basis:**

$$\gamma(s) := \sum_{k=0}^{n} B_{n,k}(s) p_k$$

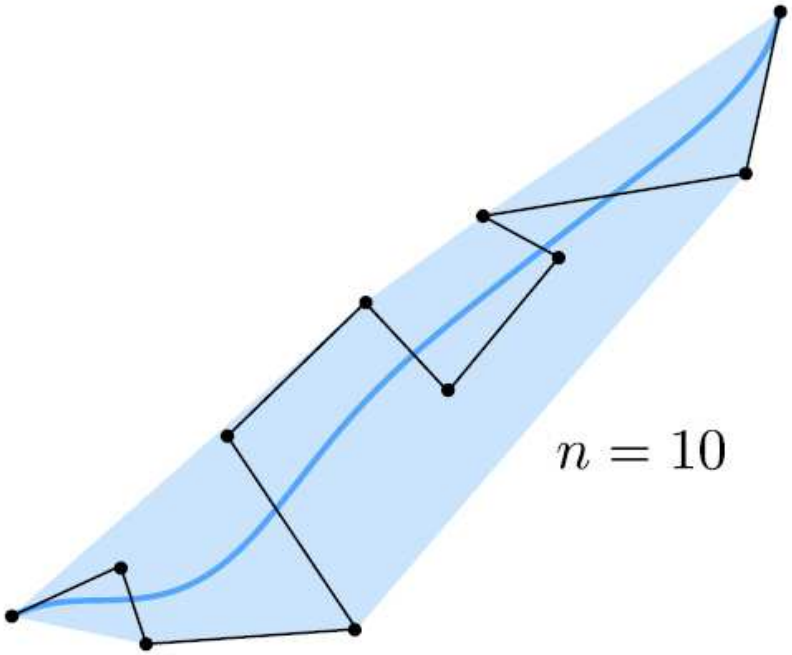
control points

  - **For n=1, just get a line segment!**
  - **For n=3, get "cubic Bézier":**
  - **Important features:**
    1. **interpolates endpoints**
    2. **tangent to end segments**
    3. **contained in convex hull *(nice for rasterization)***



Different representations

- **Higher-order Bézier curves?**
  - **What if we want a more interesting curve?**
  - **High-degree Bernstein polynomials don't interpolate well:**



Very hard to control!

Different representations

- **B-Spline curves (Explicit)**
  - Instead, use many low-order Bézier curve (B-spline)
  - Widely-used technique in software (Illustrator, Inkscape, etc.)

  - Formally, piecewise Bézier curve:

B-spline

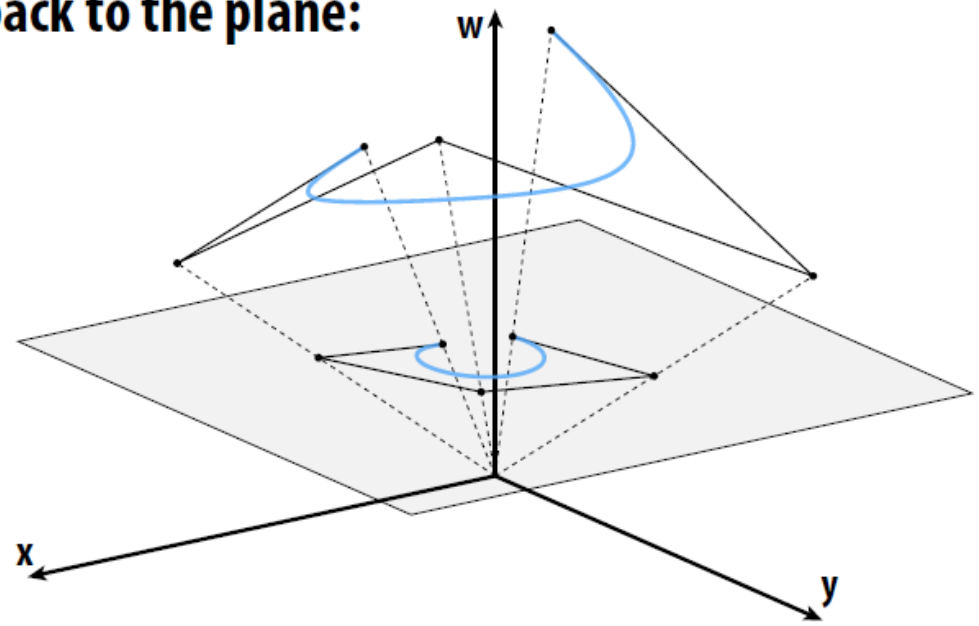
$$\gamma(u) := \gamma_i \left(\frac{u - u_i}{u_{i+1} - u_i} \right), \qquad u_i \leq u < u_{i+1}$$

Bézier

  - Location of u_i parameters are called "knots"

Different representations

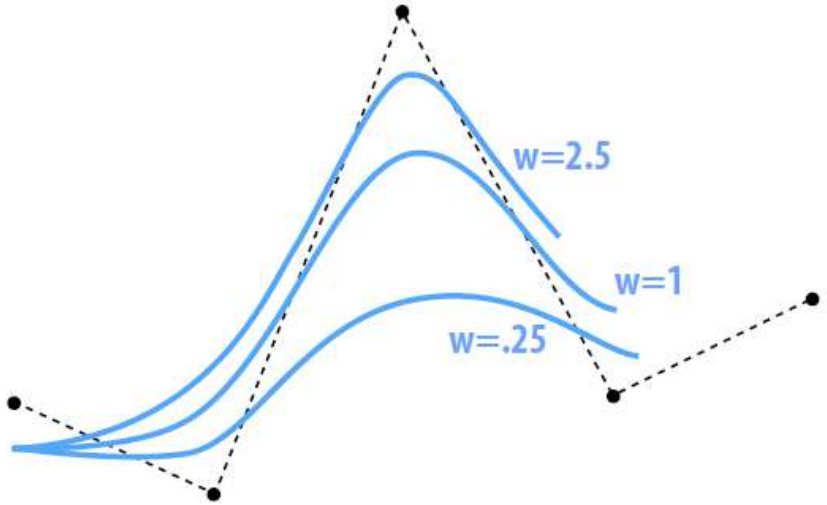
- **Rational B-splines (explicit)**
  - **B-Splines can't exactly represent *conics*—not even the circle!**
  - **Solution: interpolate in homogeneous coordinates, then project back to the plane:**



Result is called a *rational* B-spline.

Different Representations

- **NURBS (explicit)**
  - (N)on-(U)niform (R)ational (B)-(S)pline
    - knots at arbitrary locations (non-uniform)
    - expressed in homogeneous coordinates (rational)
    - piecewise polynomial curve (B-Spline)
  - Homogeneous coordinate w controls "strength" of a vertex:

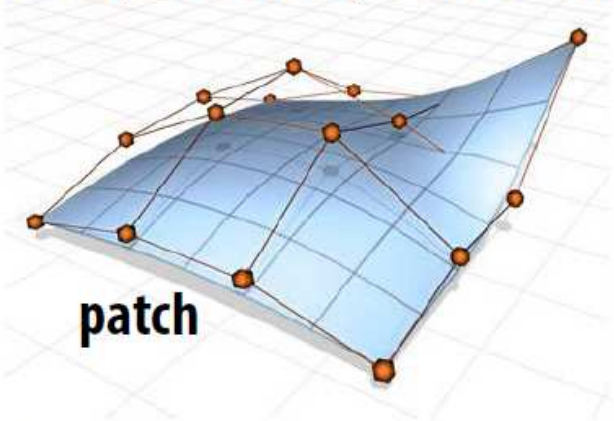


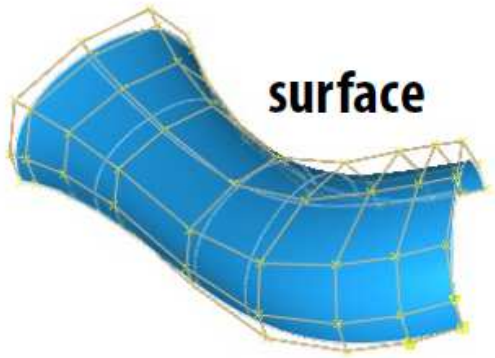
Different representations

- **NURBS Surface (explicit)**
  - Much more common than using NURBS for curves
  - Use *tensor product* of scalar NURBS curves to get a patch:

$$S(u,v) := N_i(u)N_j(v)p_{ij}$$

  - Multiple NURBS patches form a surface

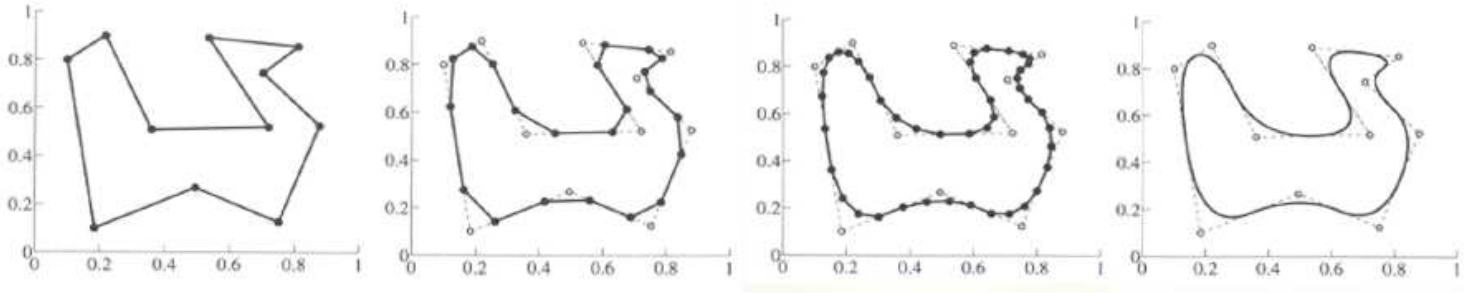




  - Pros: easy to evaluate, exact conics, high degree of continuity
  - Cons: Hard to piece together patches, hard to edit (many DOFs)

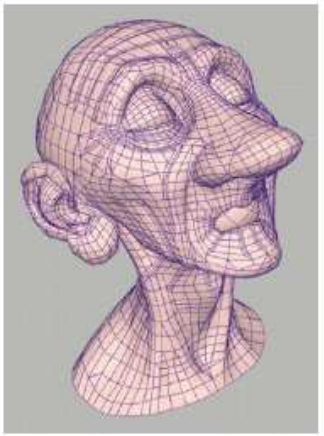
Different representations

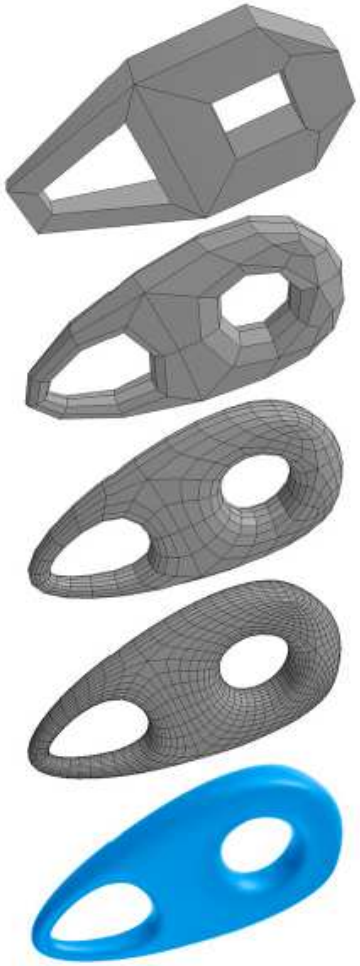
- **Subdivision (explicit)**
  - **Alternative starting point for B-spline curves: *subdivision***
  - **Start with control curve**
  - **Insert new vertex at each edge midpoint**
  - **Update vertex positions according to fixed rule**
  - **For careful choice of averaging rule, yields smooth curve**
    - **Average with "next" neighbor (Chaikin): quadratic B-spline**
    - **Lane-Riesenfeld: B-spline curve of any degree**



Different representations

- Subdivision Surfaces (Explicit)
  - Start with coarse polygon mesh ("control cage")
  - Subdivide each element
  - Update vertices via local averaging
  - Many possible rule:
    - Catmull-Clark (quads)
    - Loop (triangles)
    - ...
  - Common issues:
    - interpolating or approximating?
    - continuity at vertices?
  - Easier than NURBS for modeling; harder to guarantee continuity

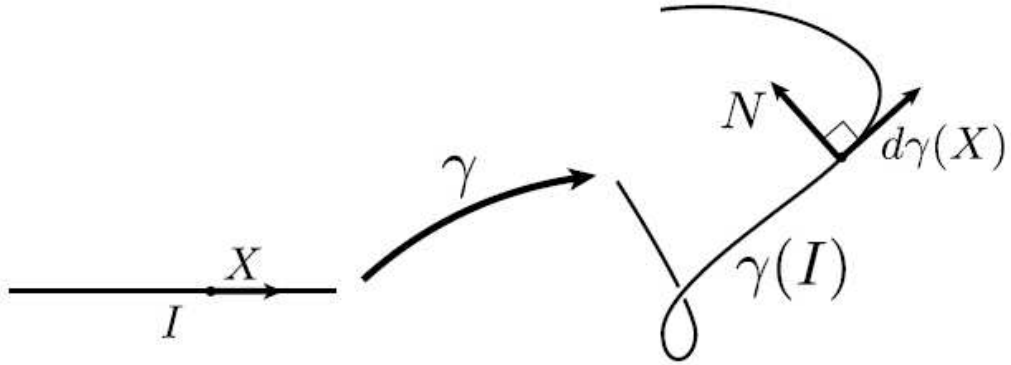


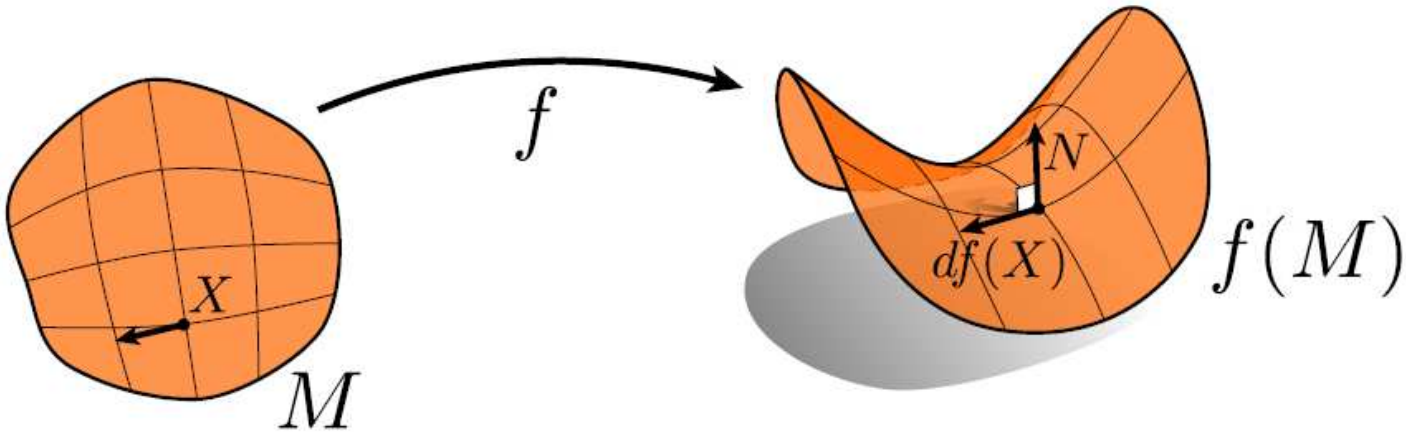


2. Geometric surface

Tangent vector of a surface

- **How to evaluate the surface tangent?**



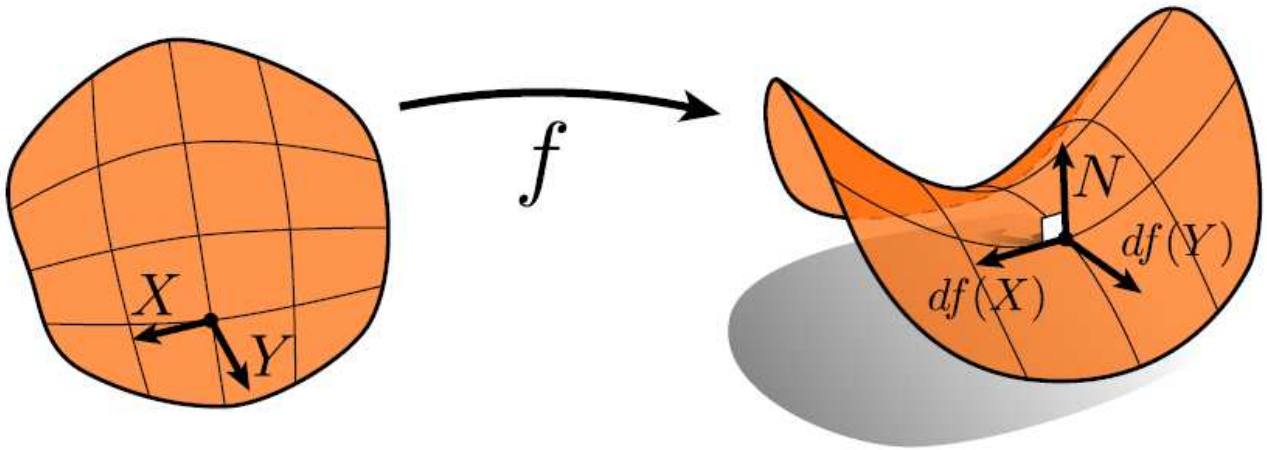


Surface normal

- **How to evaluate the surface normal?**
  - A normal is a vector orthogonal to all tangents

$$N$$

$$N \cdot df(X) = 0 \quad \forall X$$

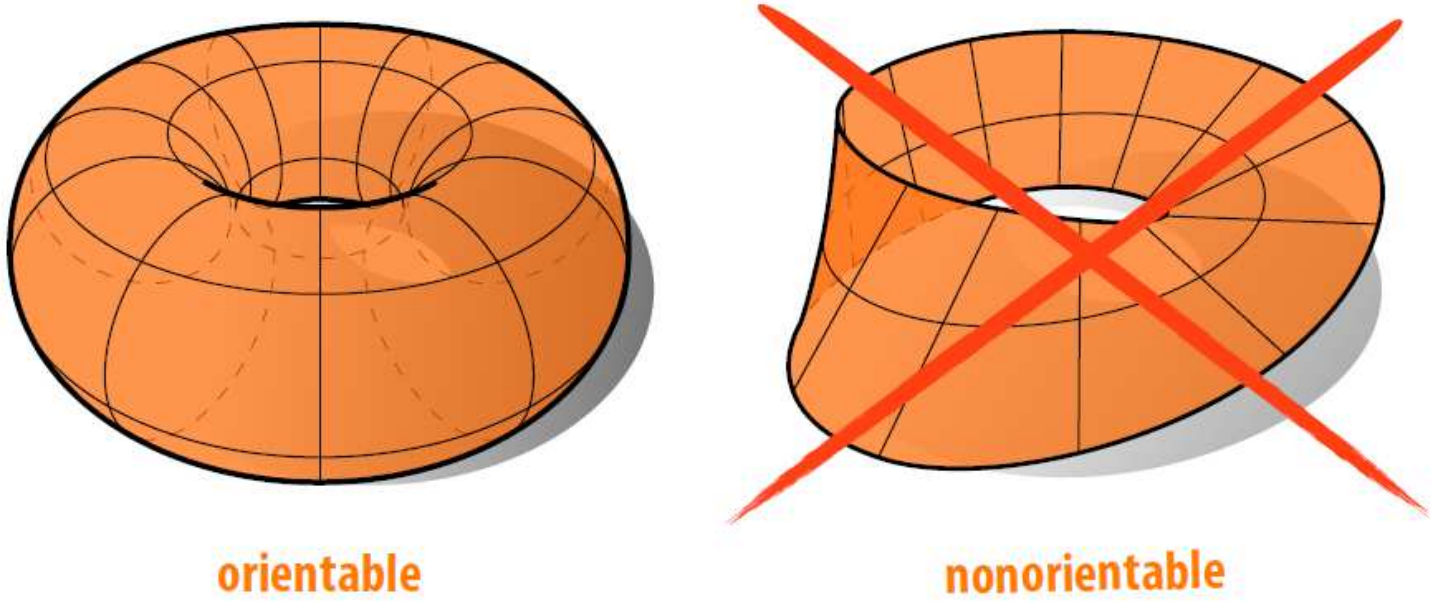


Surface normal

- **Which direction does the normal point?**

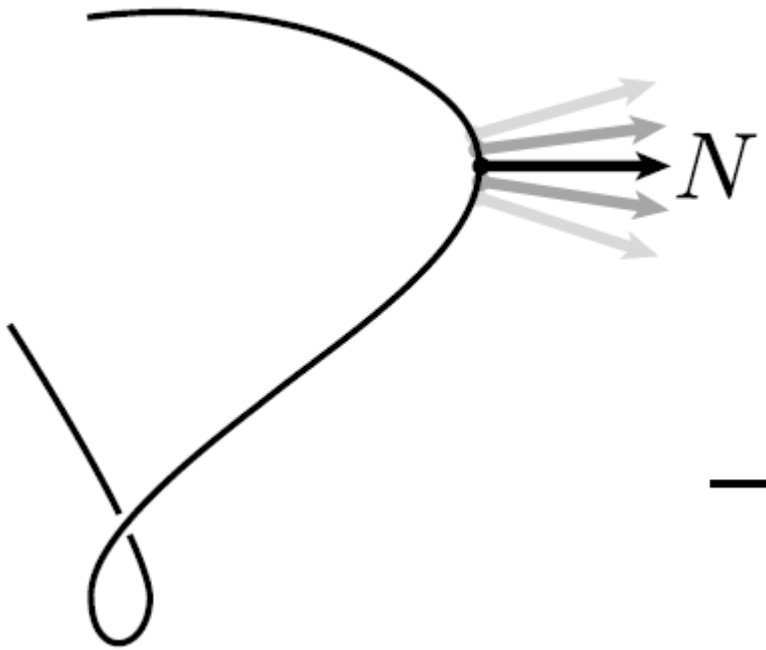
$$|N| = 1$$

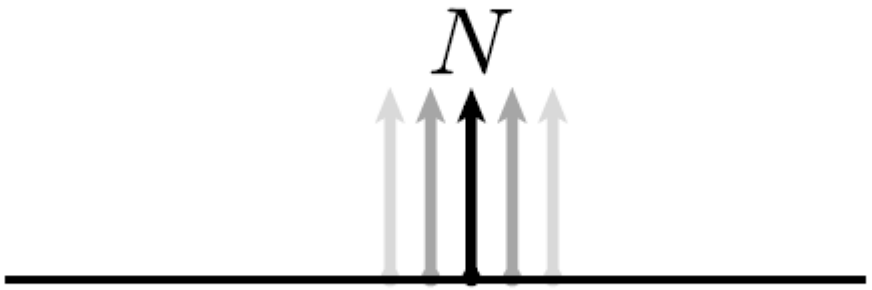
$$N \cdot df(X) = 0 \quad \forall X$$



Curvature

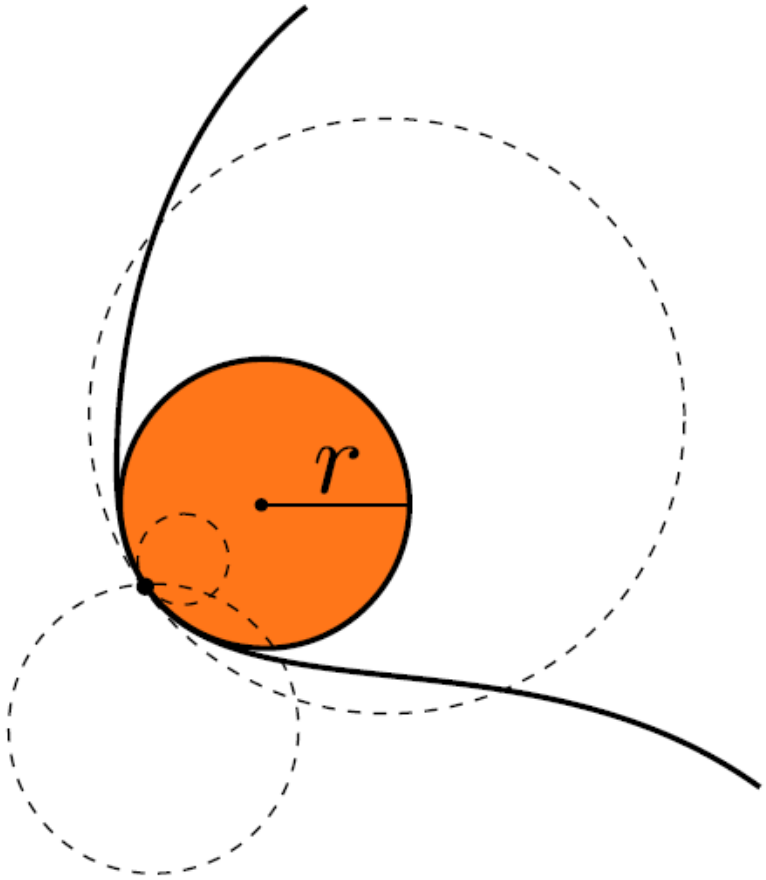
- **Curvature is the rate of *change in normal***





Curvature

- **Standard definition: radius of curvature**



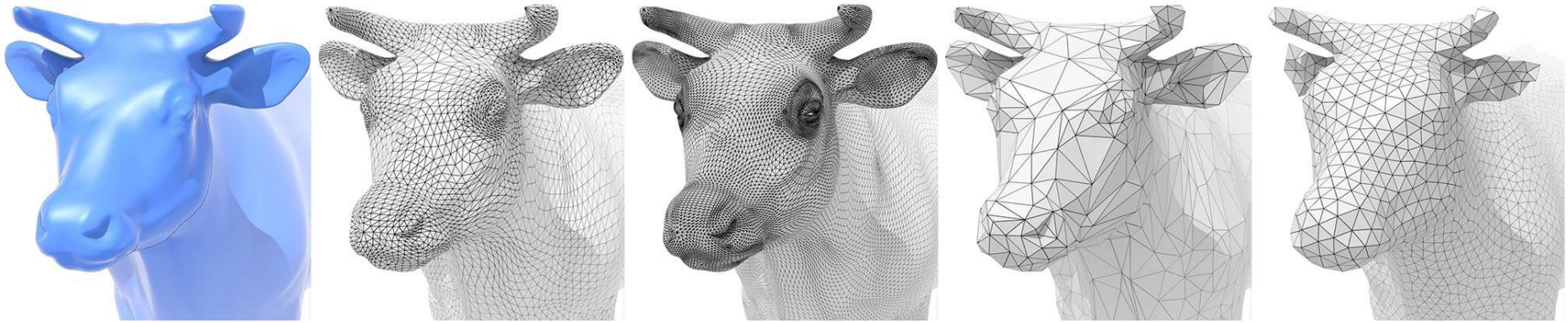
$$\kappa = \frac{1}{r}$$

curvature

3. Mesh representation

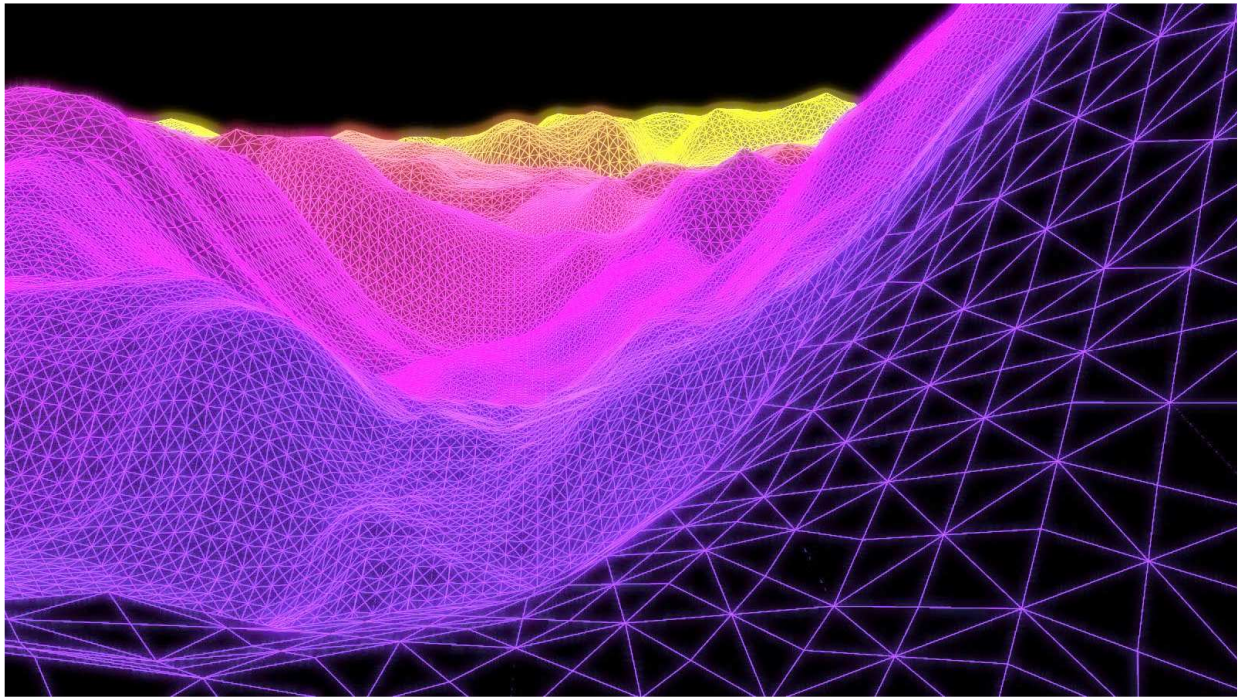
Discretizing geometric representations

- **Converting continuous surface representation into an assembly of discrete primitive elements**
  - Triangles/quadrilaterals
  - A certain organization of primitive elements



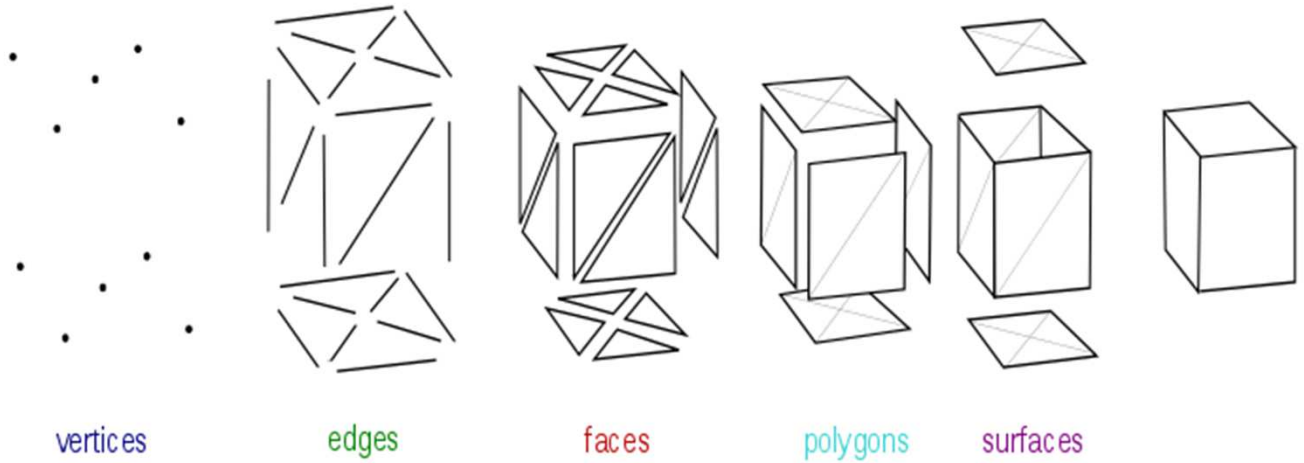
Mesh

- **Representation of shapes with a collection of geometrical primitives**
  - Triangles/quadrilaterals
  - Vertices, faces, normals, collectivity (topology)



Mesh organization

- **Polygon mesh**
  - A collection of vertices, edges and faces
    - Define the shape of a polyhedral object in 3D computer graphics and solid modeling
    - The faces usually consist of triangles, quadrilaterals, or other simple convex polygons

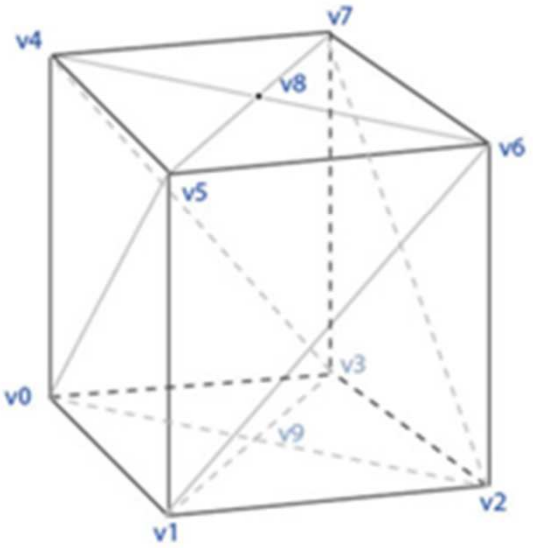


Mesh organization

- **Vertex-vertex representation**

Vertex-Vertex Meshes (VV)

	Vertex List	
v0	0,0,0	v1 v5 v4 v3 v9
v1	1,0,0	v2 v6 v5 v0 v9
v2	1,1,0	v3 v7 v6 v1 v9
v3	0,1,0	v2 v6 v7 v4 v9
v4	0,0,1	v5 v0 v3 v7 v8
v5	1,0,1	v6 v1 v0 v4 v8
v6	1,1,1	v7 v2 v1 v5 v8
v7	0,1,1	v4 v3 v2 v6 v8
v8	.5,.5,1	v4 v5 v6 v7
v9	.5,.5,0	v0 v1 v2 v3



Mesh organization

- **Face-vertex representation**

Face-Vertex Meshes

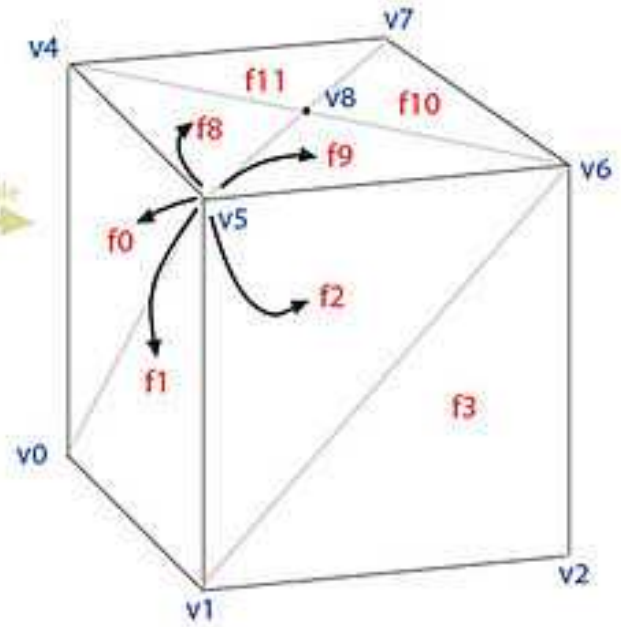
Face List

f0	v0 v4 v5
f1	v0 v5 v1
f2	v1 v5 v6
f3	v1 v6 v2
f4	v2 v6 v7
f5	v2 v7 v3
f6	v3 v7 v4
f7	v3 v4 v0
f8	v8 v5 v4
f9	v8 v6 v5
f10	v8 v7 v6
f11	v8 v4 v7
f12	v9 v5 v4
f13	v9 v6 v5
f14	v9 v7 v6
f15	v9 v4 v7

Vertex List

v0	0,0,0	f0 f1 f12 f15 f7
v1	1,0,0	f2 f3 f13 f12 f1
v2	1,1,0	f4 f5 f14 f13 f3
v3	0,1,0	f6 f7 f15 f14 f5
v4	0,0,1	f6 f7 f0 f8 f11
v5	1,0,1	f0 f1 f2 f9 f8
v6	1,1,1	f2 f3 f4 f10 f9
v7	0,1,1	f4 f5 f6 f11 f10
v8	.5,.5,0	f8 f9 f10 f11
v9	.5,.5,1	f12 13 14 15

example



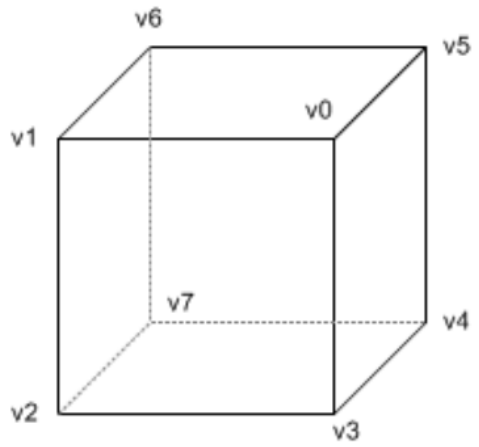
Draw meshes in OpenGL

- **Using vertex array:**
  - Vertex position
  - Vertex color
  - Vertex normal
  - Vertex texture coordinates
- **Using index array:**
  - Face index: triangle elements, quadrilateral elements, etc.

Draw meshes in OpenGL

- **Traditional way of drawing geometries**

```
glBegin(GL_TRIANGLES); // draw a cube with 12
triangles
// front face =================
glVertex3fv(v0); // v0-v1-v2
glVertex3fv(v1);
glVertex3fv(v2);

glVertex3fv(v2); // v2-v3-v0
glVertex3fv(v3);
glVertex3fv(v0);

// right face =================
glVertex3fv(v0); // v0-v3-v4
glVertex3fv(v3);
glVertex3fv(v4);

glVertex3fv(v4); // v4-v5-v0
glVertex3fv(v5);
glVertex3fv(v0);

// draw other 4 faces

glEnd();
```



Draw meshes in OpenGL

- **Using vertex array**
  - Transmit vertex array to GPU at once
  - Activate vertex array

```
glEnableClientState(GL_VERTEX_ARRAY);
```

  - Specify vertex data

```
glVertexPointer(3, GL_FLOAT, 0, vertices);
glColorPointer(3, GL_FLOAT, 0, colors);
glNormalPointer(3, GL_FLOAT, 0, normals);
…
```

  - Draw meshes based on face list index

```
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, indices);
```

  - Deactivate vertex array

```
glDisableClientState(GL_VERTEX_ARRAY);
```

Draw meshes in OpenGL

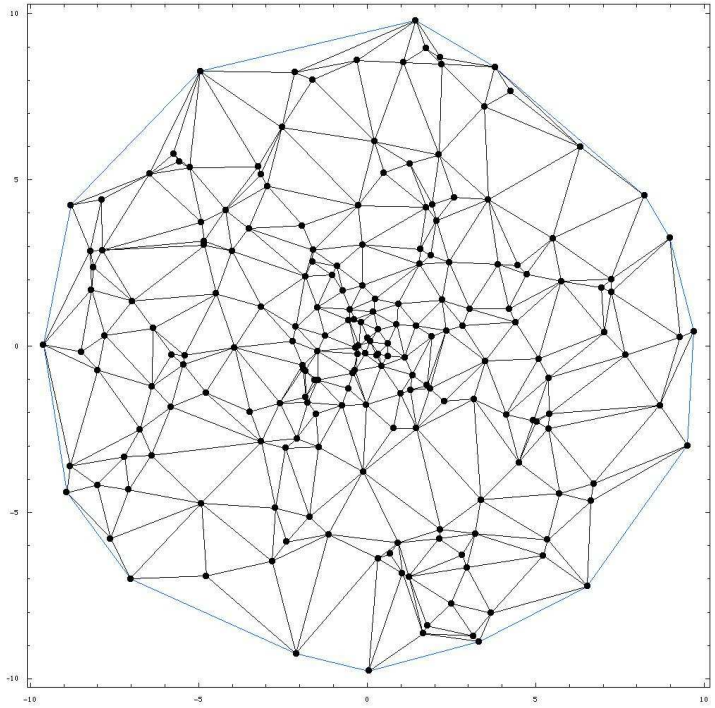
- **An example code in OpenGL**
  - http://www.songho.ca/opengl/gl_vertexarray.html

```
GLfloat vertices[] = {...};           // 8 of vertex coords
GLubyte indices[] = {0,1,2, 2,3,0,    // 36 of indices
                     0,3,4, 4,5,0,
                     0,5,6, 6,1,0,
                     1,6,7, 7,2,1,
                     7,4,3, 3,2,7,
                     4,7,6, 6,5,4};
...
// activate and specify pointer to vertex array
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

// draw a cube
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, indices);

// deactivate vertex arrays after drawing
glDisableClientState(GL_VERTEX_ARRAY);
```

4. Triangulation

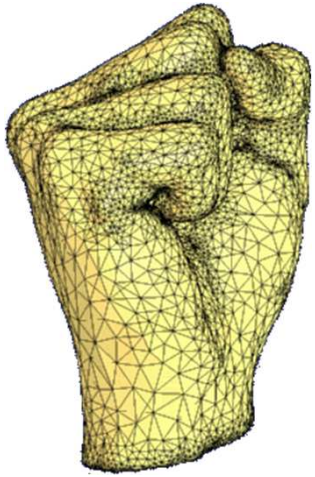
Triangulation

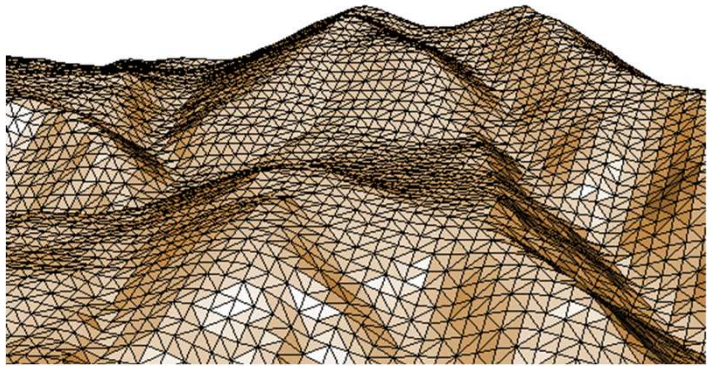
- **What is triangulation**
  - In trigonometry and geometry, triangulation is the process of determining the location of a point by forming triangles to it from known points

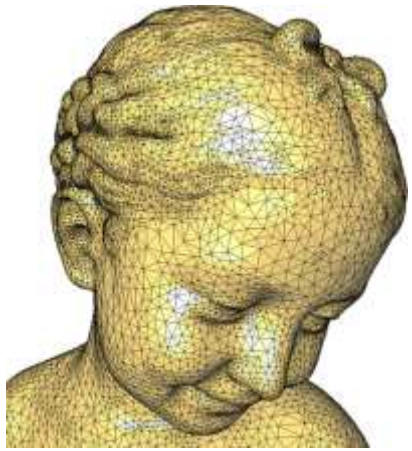


Triangulation

- **More examples**
  - Triangulations of 3D surfaces





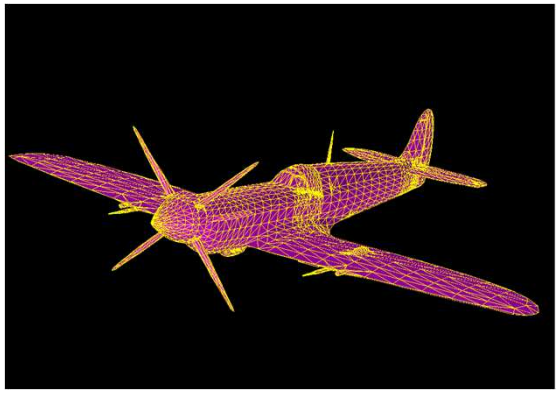


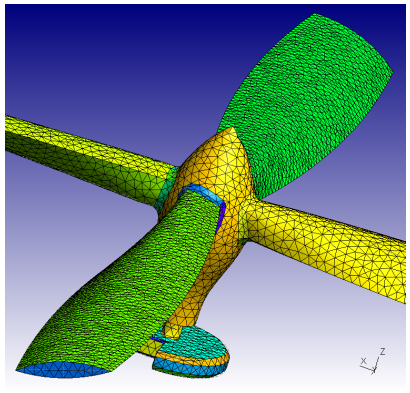
Why we need triangulation?

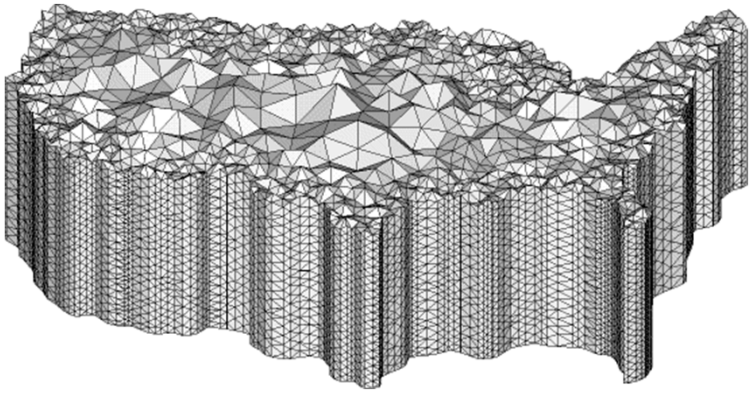
- **Interpolation**
  - Interpolate interior continuous values
  - Surface approximation
  - Solving partial differential equations
- **Representing objects based on sample points**
  - No topology among sample points
- **Surface reconstruction**
  - Fitting triangular meshes from dense/sparse points

Why we need triangulation?

- **FEM mesh illustration**
  - For CAD/CAE, simulation
  - Mostly triangulated mesh

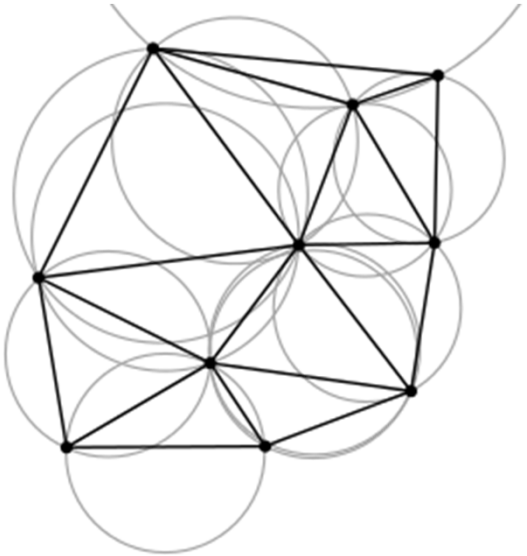






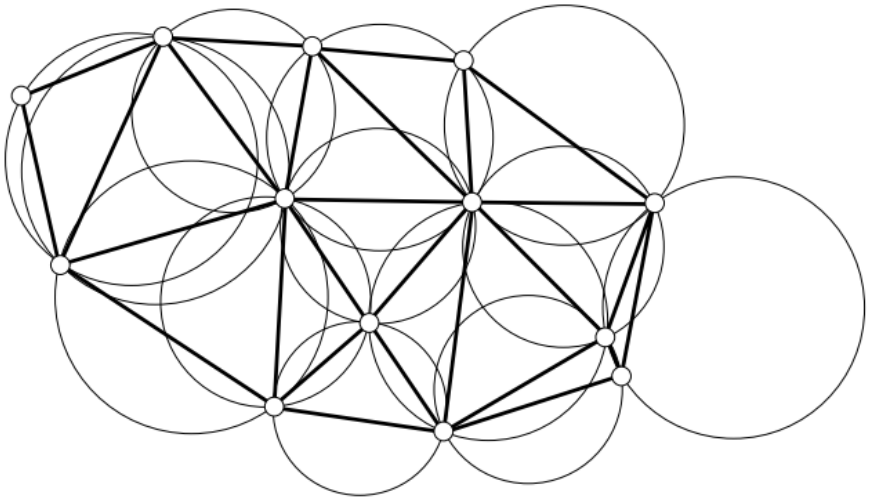
Delaunay triangulation

- **Delaunay triangulation for a point-set P:**
  - A triangulation DT(**P**) such that no point in **P** is inside the circumcircle of any triangle in DT(**P**)
  - Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation



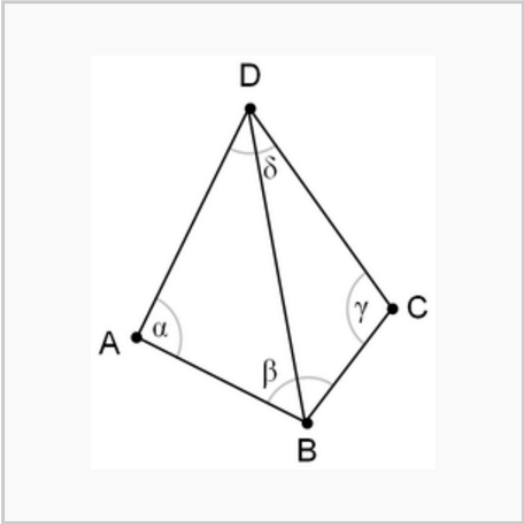
Delaunay triangulation

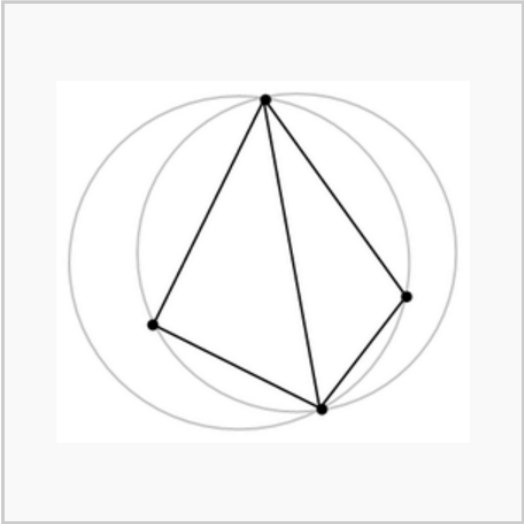
- **Property for Delaunay triangulation**
  - Every triangle in a Delaunay triangulation has an empty open circumdisk
  - Every point set has a Delaunay triangulation
  - The Delaunay triangulation is unique if and only if no four or more points lie on a common empty circle

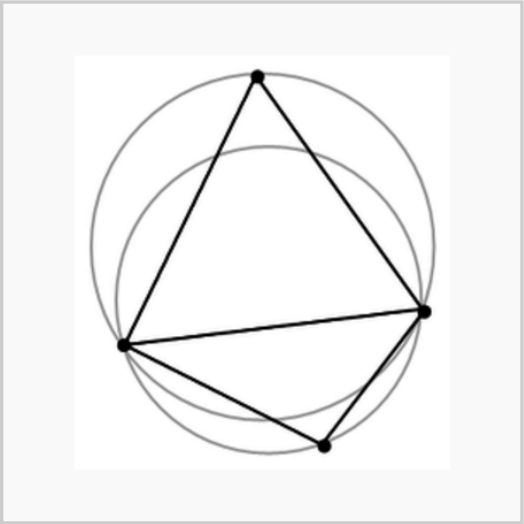


Delaunay triangulation

- **Flipping**
  - If two triangles do not meet the Delaunay condition, switching the connection produces two triangles with Delaunay condition





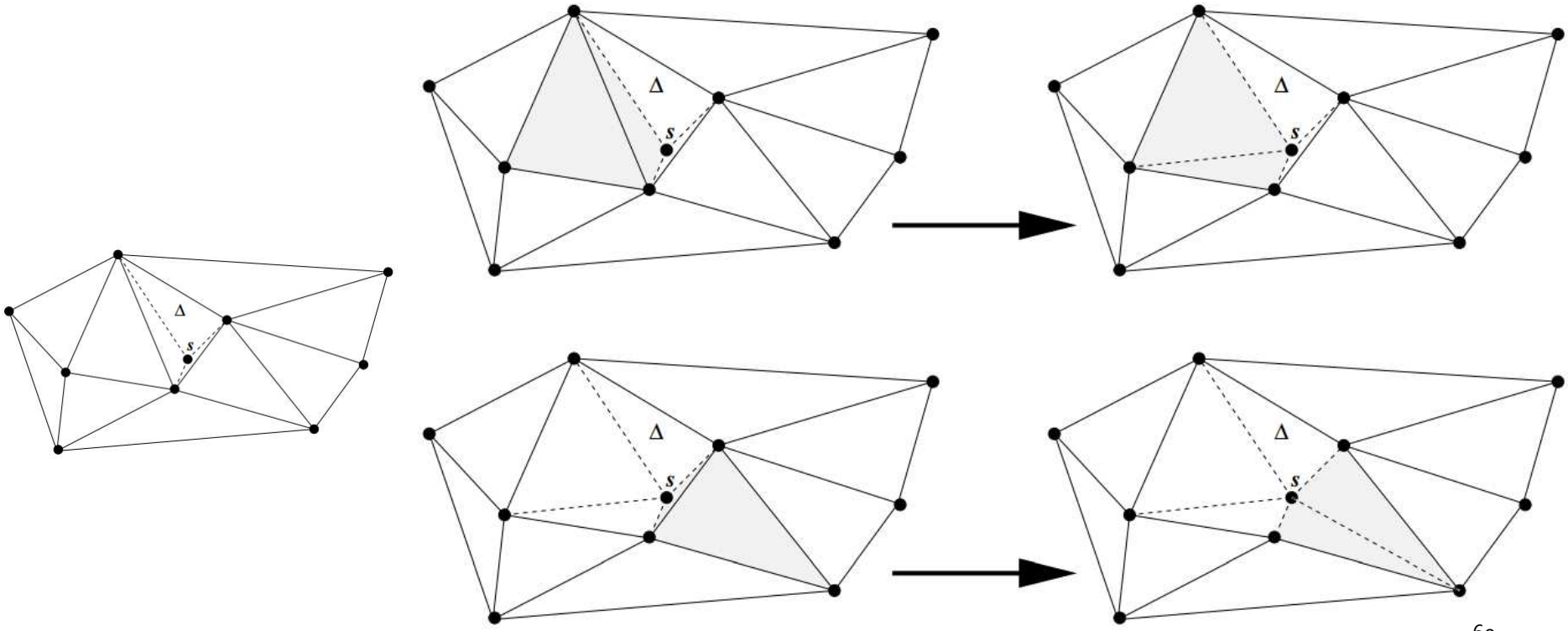


Delaunay triangulation

- **How to make a Delaunay triangulation**
  - Incremental construction
    - Incrementally create a triangulation by point insertion
    - Select a convex quadrilateral
    - Perform flipping
    - Scan until there is no non-Delaunay triangulation left

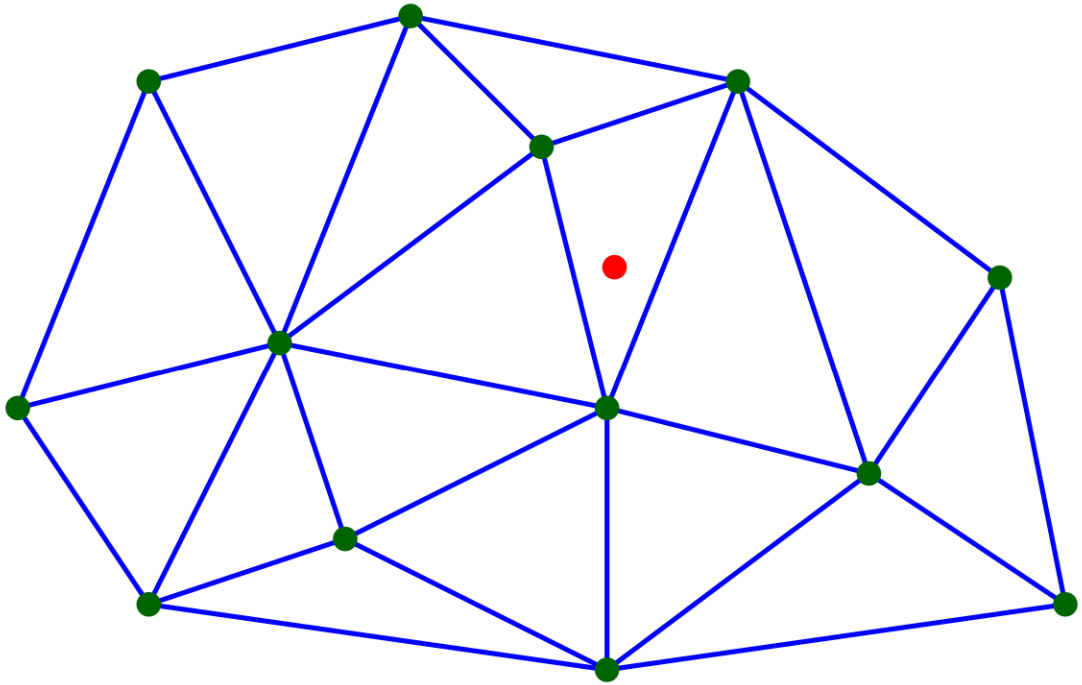
Delaunay triangulation

- **How to make a Delaunay triangulation**
  - Incremental construction



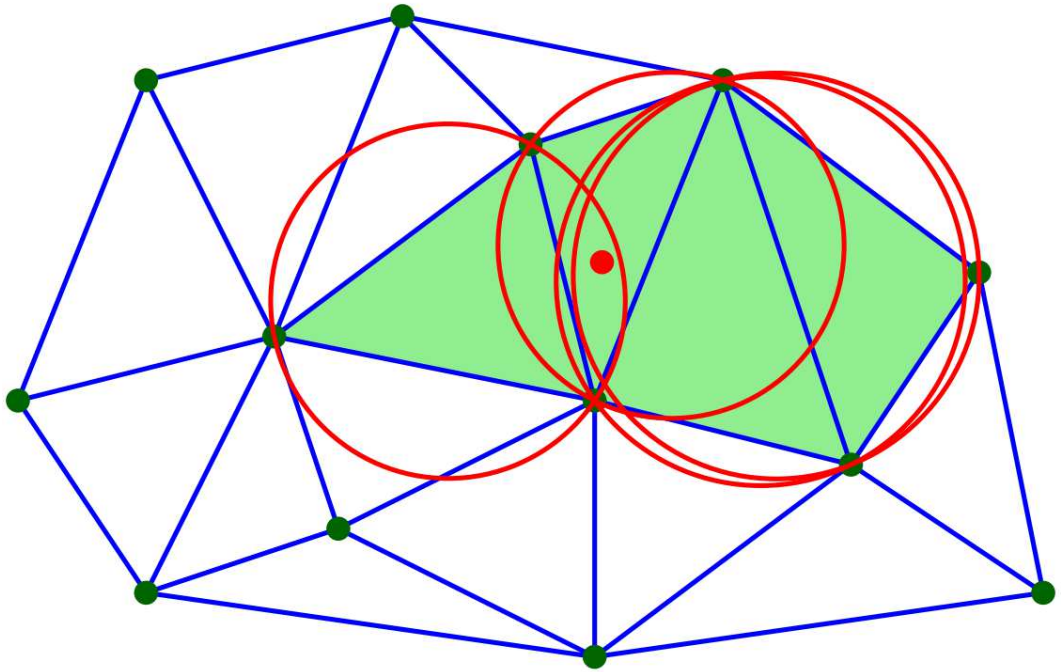
Delaunay triangulation

- **Incremental Delaunay**



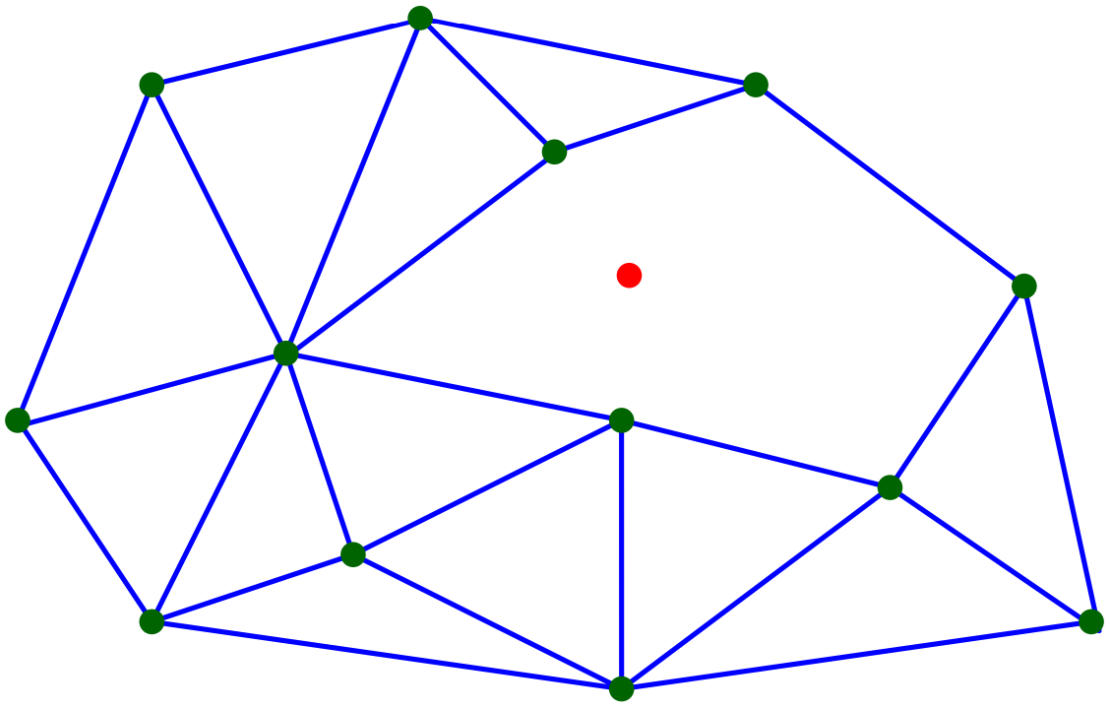
Delaunay triangulation

- **Incremental Delaunay**
  - Find triangles in conflict



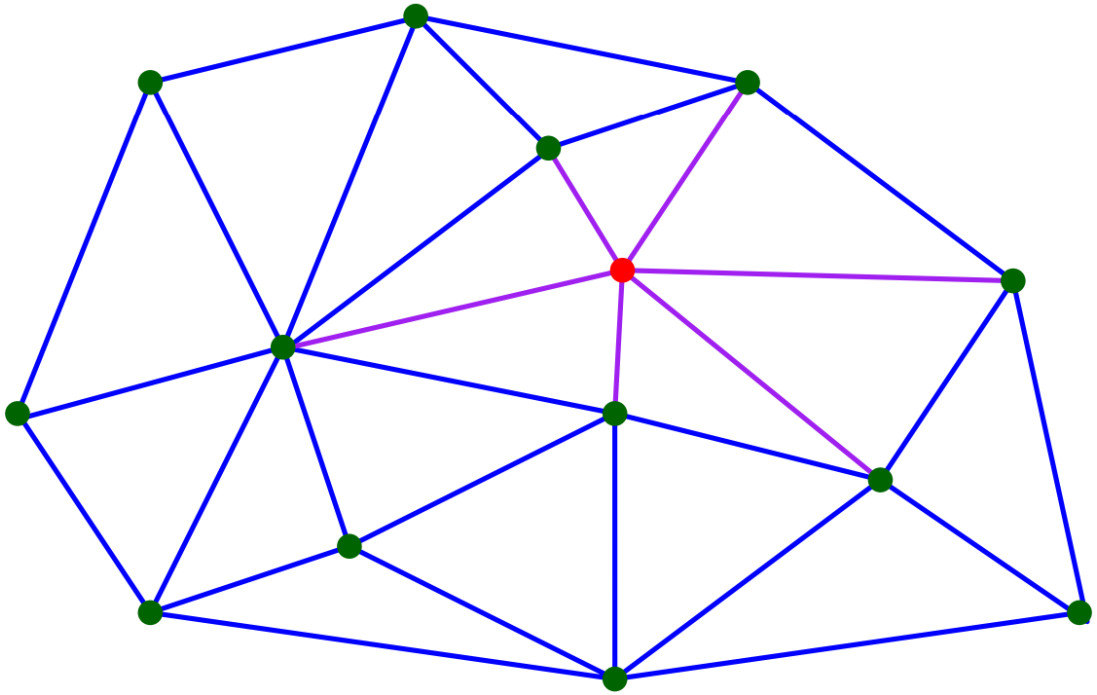
Delaunay triangulation

- **Incremental Delaunay**
  - Delete triangles in conflict



Delaunay triangulation

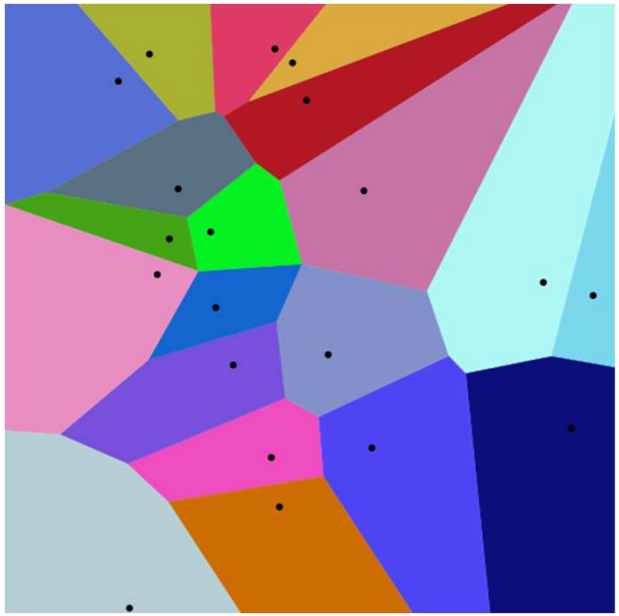
- **Incremental Delaunay**
  - Triangulate the hole



Voronoi diagram

- **A Voronoi diagram is**
  - a partitioning of a plane into regions based on distance to points
  - for each seed there is a corresponding region consisting of all points closer to that seed than to any other

- **These regions are called *Voronoi cells***

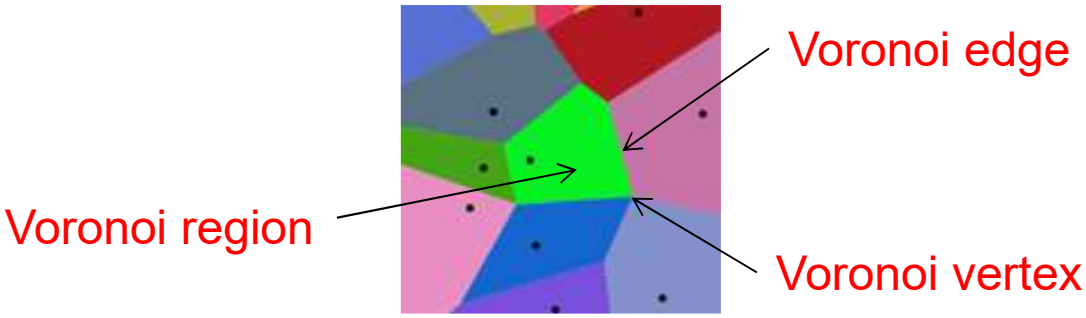


Voronoi diagram

- **Definition**
  - Let Ω be a connected region in R^2 , Let $\mathbf{S} = \{\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_n\}$ be a set of n points in Ω
  - We use $\|\mathbf{p} - \mathbf{q}\|$ to denote the Euclidean distance between two points **p** and **q**
  - For $\mathbf{s}_i \in \mathbf{S}$, we define: $R(\mathbf{S}; \mathbf{s}_i) = \{\mathbf{p} \in \Omega : \|\mathbf{p} - \mathbf{s}_i\| < \|\mathbf{p} - \mathbf{s}_j\|, \mathbf{s}_j \in \Omega, \mathbf{s}_j \mathrel{!}= \mathbf{s}_i\}$
  - In other words, $R(\mathbf{S}; \mathbf{s}_i)$ is the set of points nearer to $\mathbf{s}_i$ than to any other point in **S**
  - The region Ω is partitioned into $R(\mathbf{S}; \mathbf{s}_1)$, $R(\mathbf{S}; \mathbf{s}_2)$, . . . , $R(\mathbf{S}; \mathbf{s}_n)$ and their boundaries
  - This partition is called the Voronoi diagram of **S**, denoted by $V(\mathbf{S})$

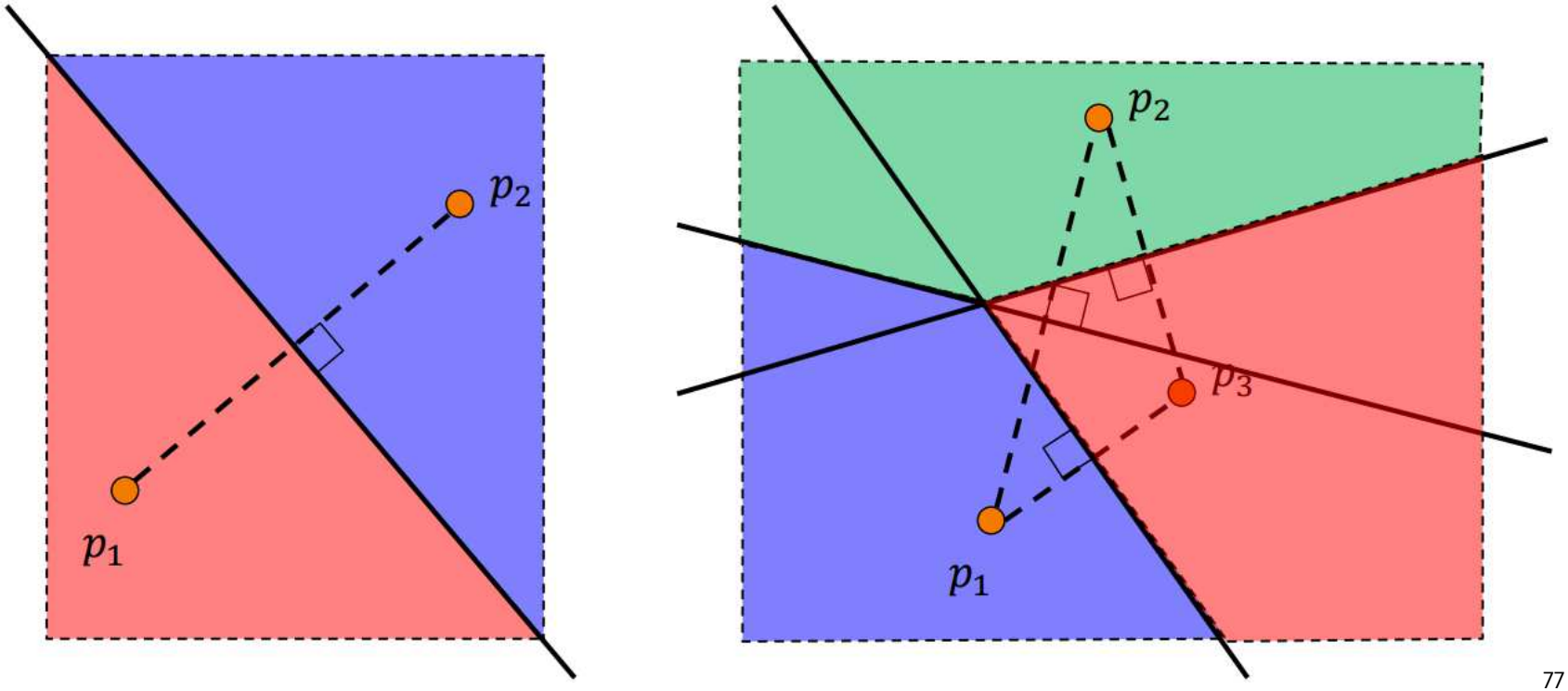
Voronoi diagram

- **Voronoi region**
  - R(**S**; $\mathbf{s}_i$) is called the Voronoi region of $\mathbf{s}_i$
- **Voronoi edge**
  - The line segments shared by the boundaries of two Voronoi regions are called Voronoi edges
- **Voronoi vertex**
  - The points shared by the boundaries of three or more Voronoi regions



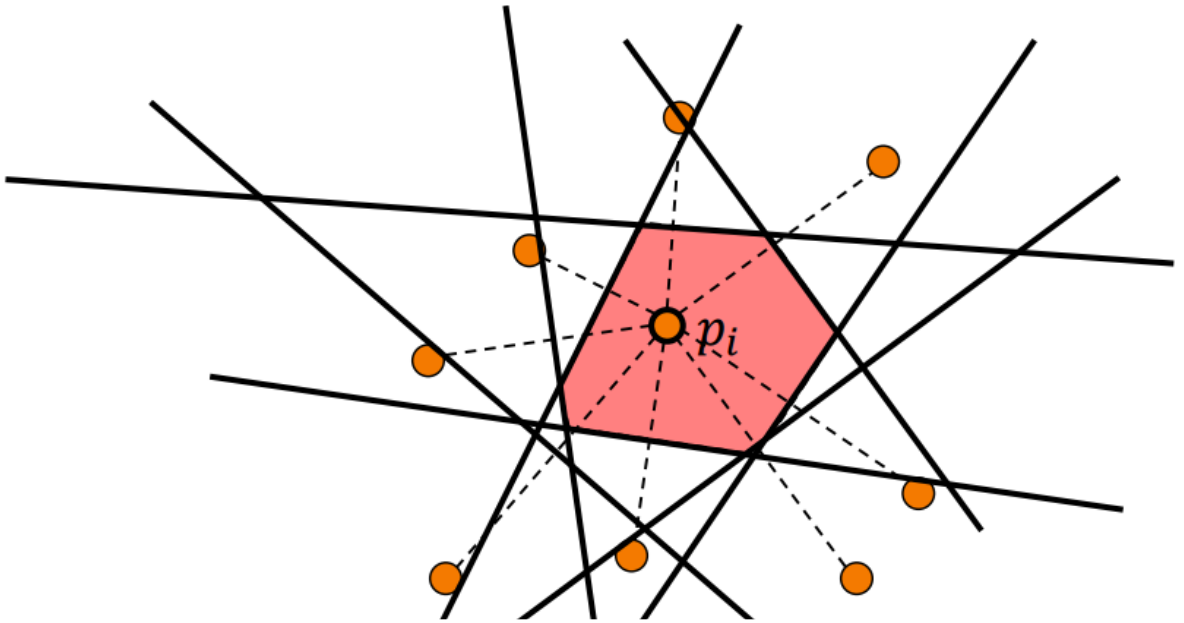
Voronoi diagram

- **How to compute Voronoi Diagram?**
  - Half-space partition
  - For example, 2 points and 3 points partition



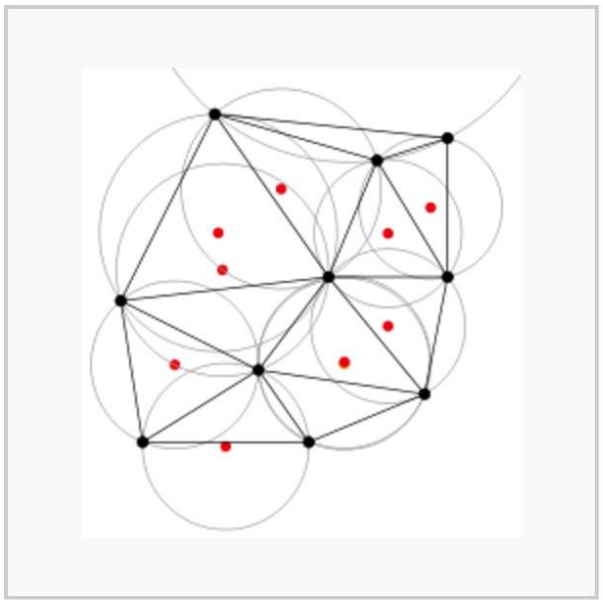
Voronoi diagram

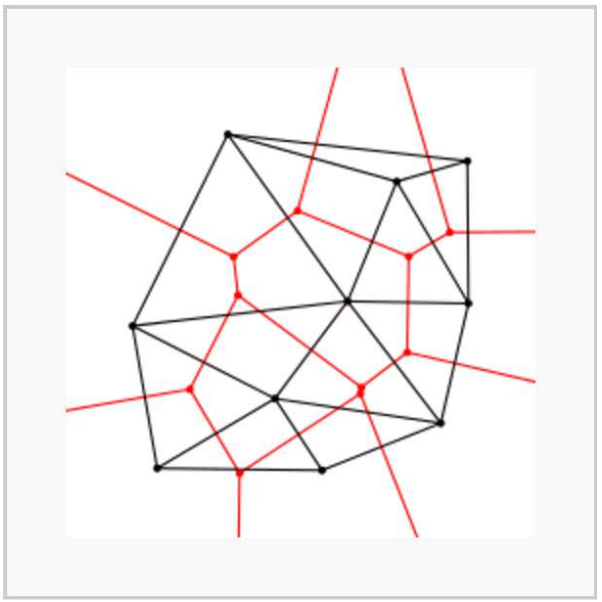
- **How to compute Voronoi Diagram?**
  - The Voronoi region associated to point p_i is the intersection of the half-spaces defined by the perpendicular bisectors



Voronoi diagram

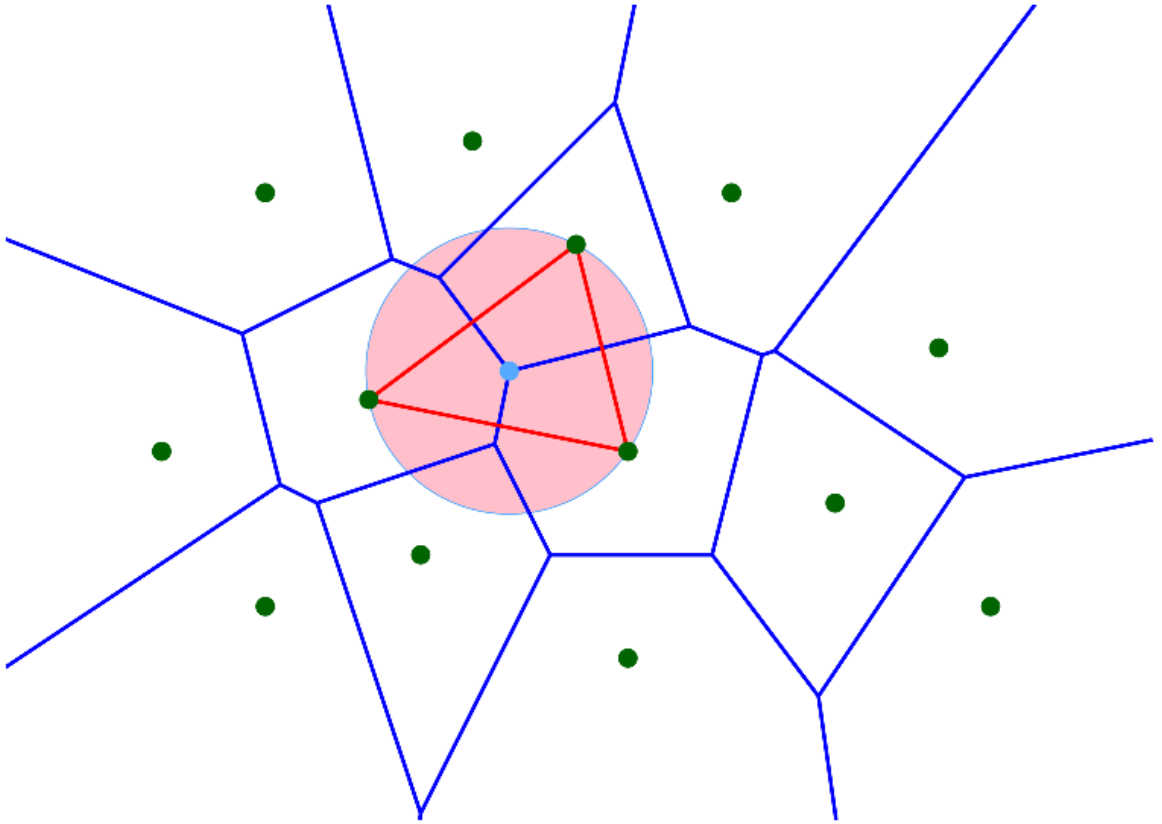
- **Why we need Voronoi diagram?**
  - Voronoi diagram of a set of points is dual to its Delaunay triangulation
  - Connecting the centers of the circumcircles produces the Voronoi diagram





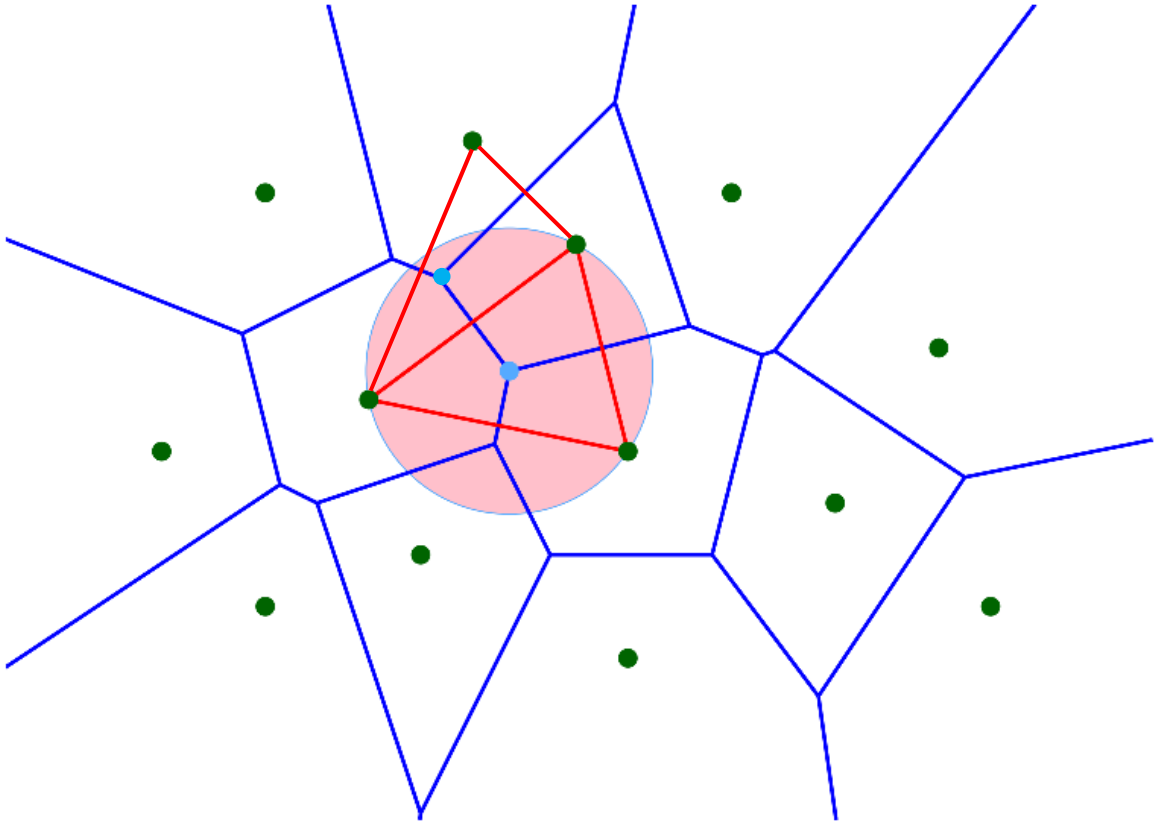
Voronoi diagram

- **From Voronoi diagram to Delaunay triangulation**



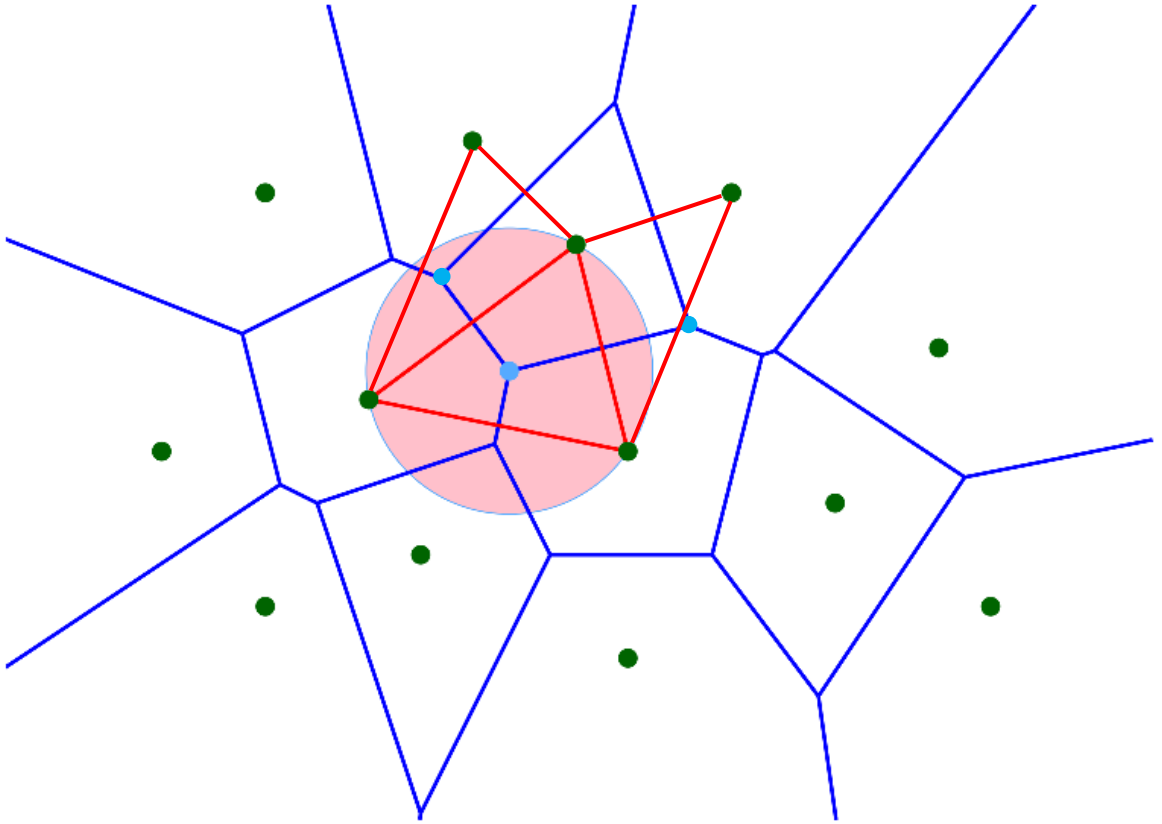
Voronoi diagram

- **From Voronoi diagram to Delaunay triangulation**



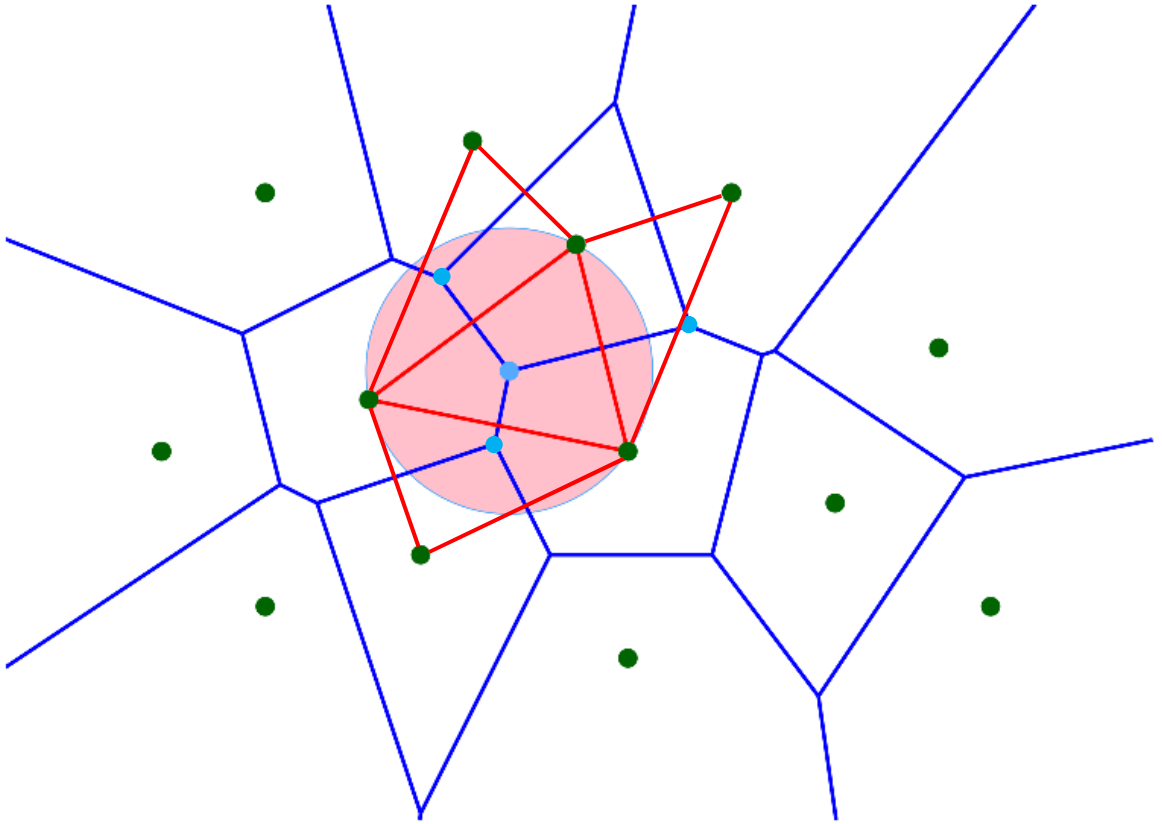
Voronoi diagram

- **From Voronoi diagram to Delaunay triangulation**



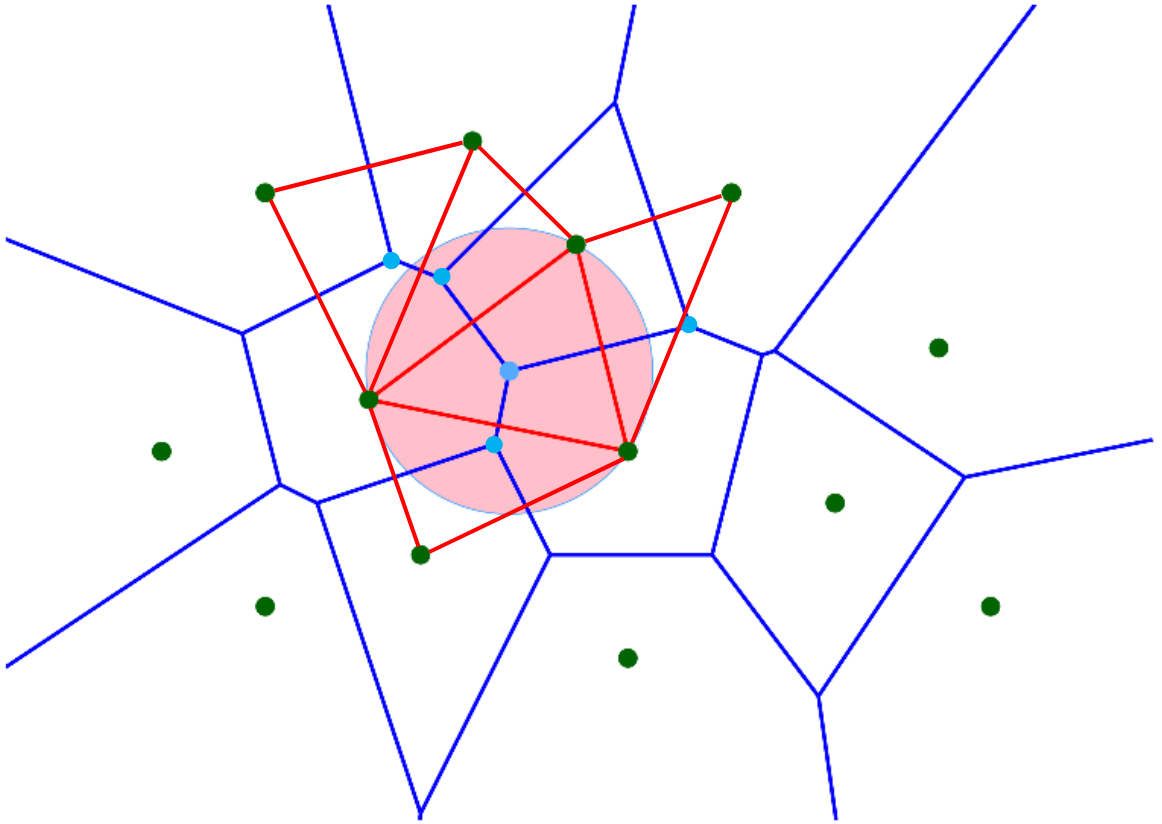
Voronoi diagram

- **From Voronoi diagram to Delaunay triangulation**



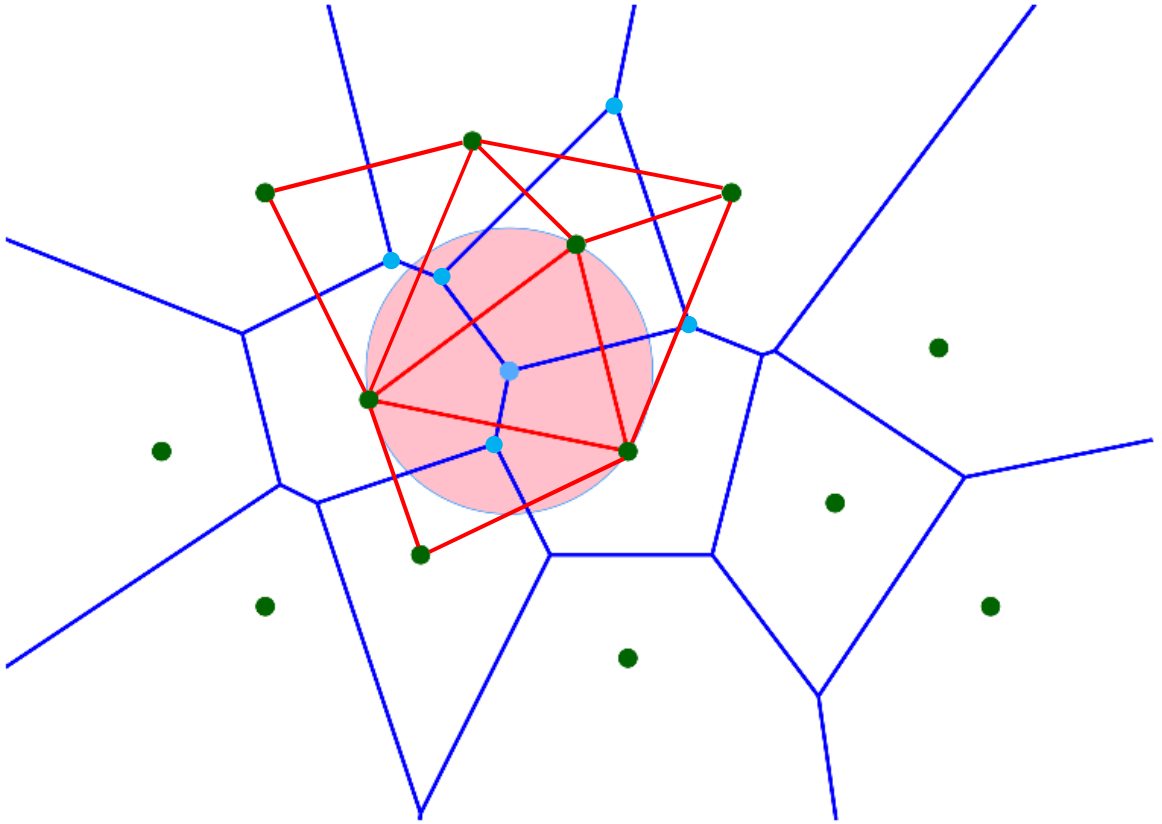
Voronoi diagram

- **From Voronoi diagram to Delaunay triangulation**



Voronoi diagram

- **From Voronoi diagram to Delaunay triangulation**

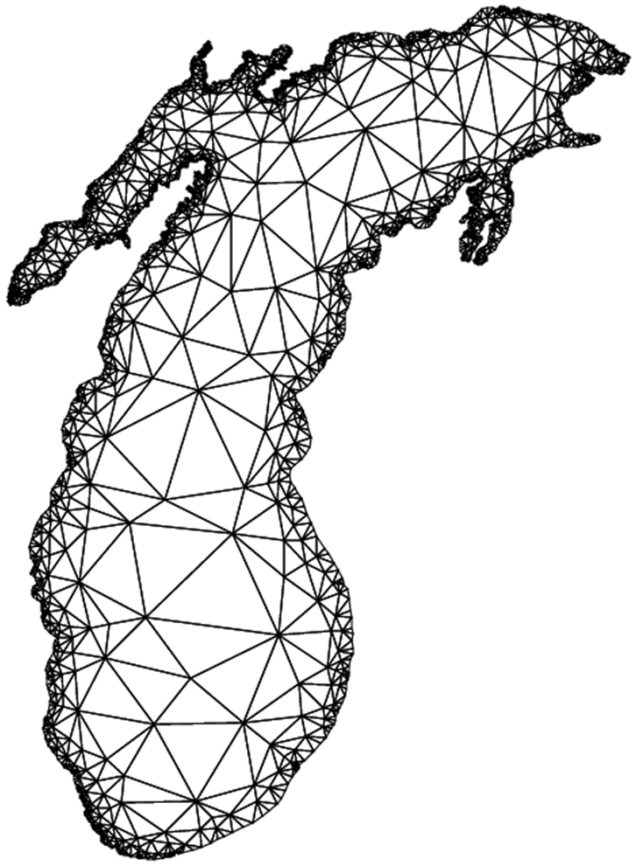


Constrained Delaunay triangulation

- **A *constrained Delaunay triangulation* is a generalization of the Delaunay triangulation**
  - Force certain required segments into the triangulation
  - Often a constrained Delaunay triangulation contains edges that do not satisfy the Delaunay condition

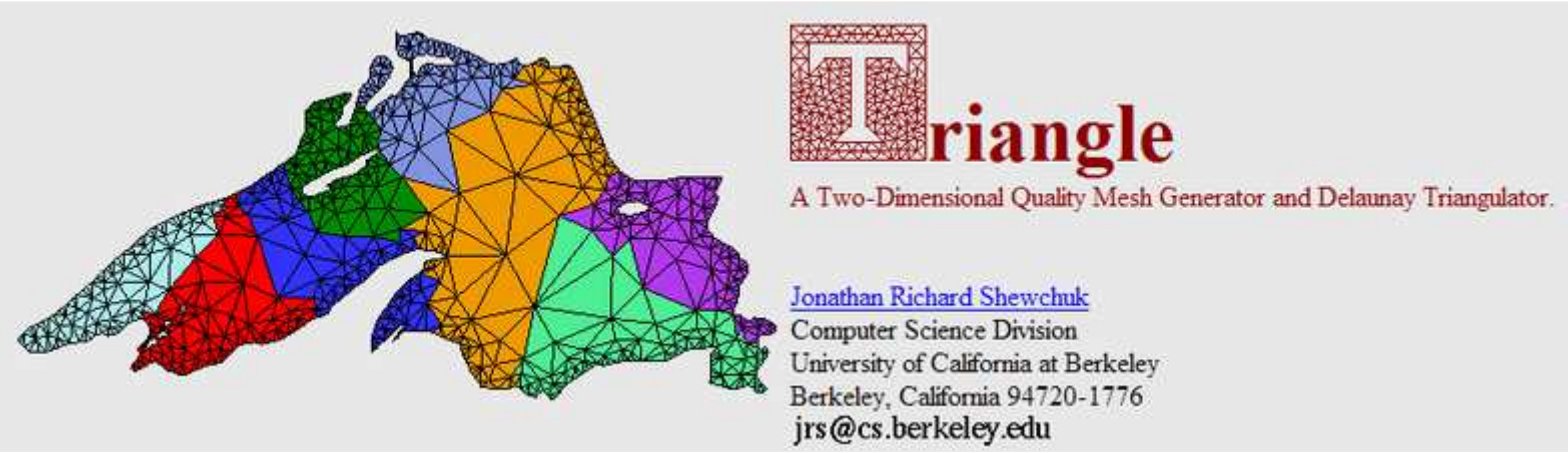
Constrained Delaunay triangulation

- **Chew's second algorithm**
  - A Delaunay refinement algorithm for creating quality constrained Delaunay triangulations
  - Chew's second algorithm has been adopted as a two-dimensional mesh generator due to practical advantages in certain cases



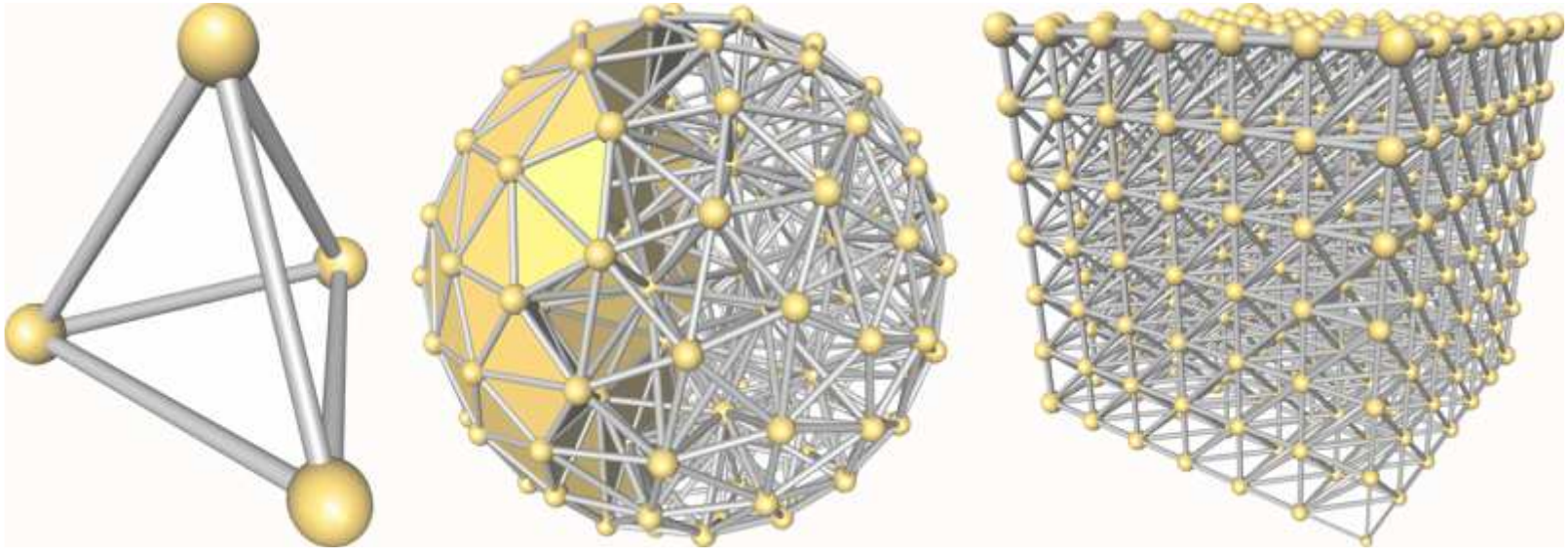
Triangulation toolbox

- **Triangle (CMU)**
  - A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator
  - https://www.cs.cmu.edu/~quake/triangle.html



CGAL

- **Computational geometric algorithms library**
  - A software project that provides easy access to efficient and reliable geometric algorithms in the form of a C++ library
  - https://www.cgal.org/



Next Lecture :

Geometric modeling 1