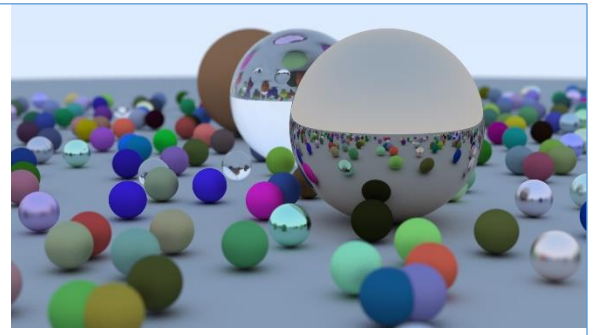
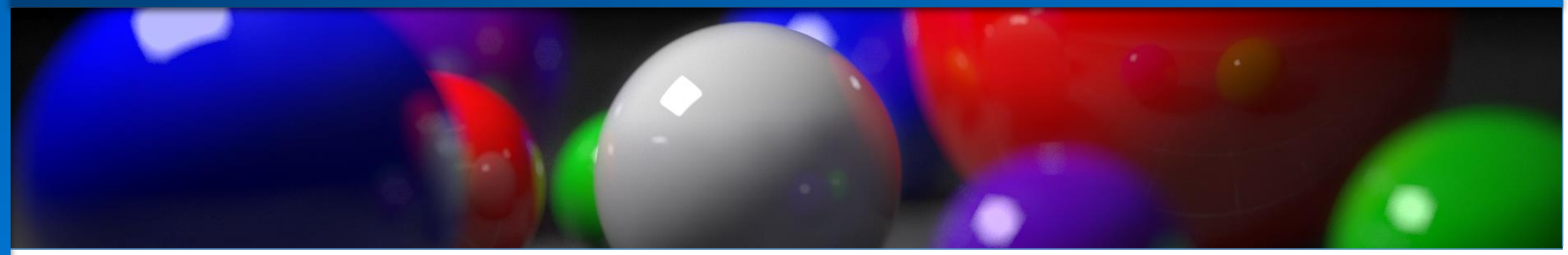
Comp4422



Computer Graphics

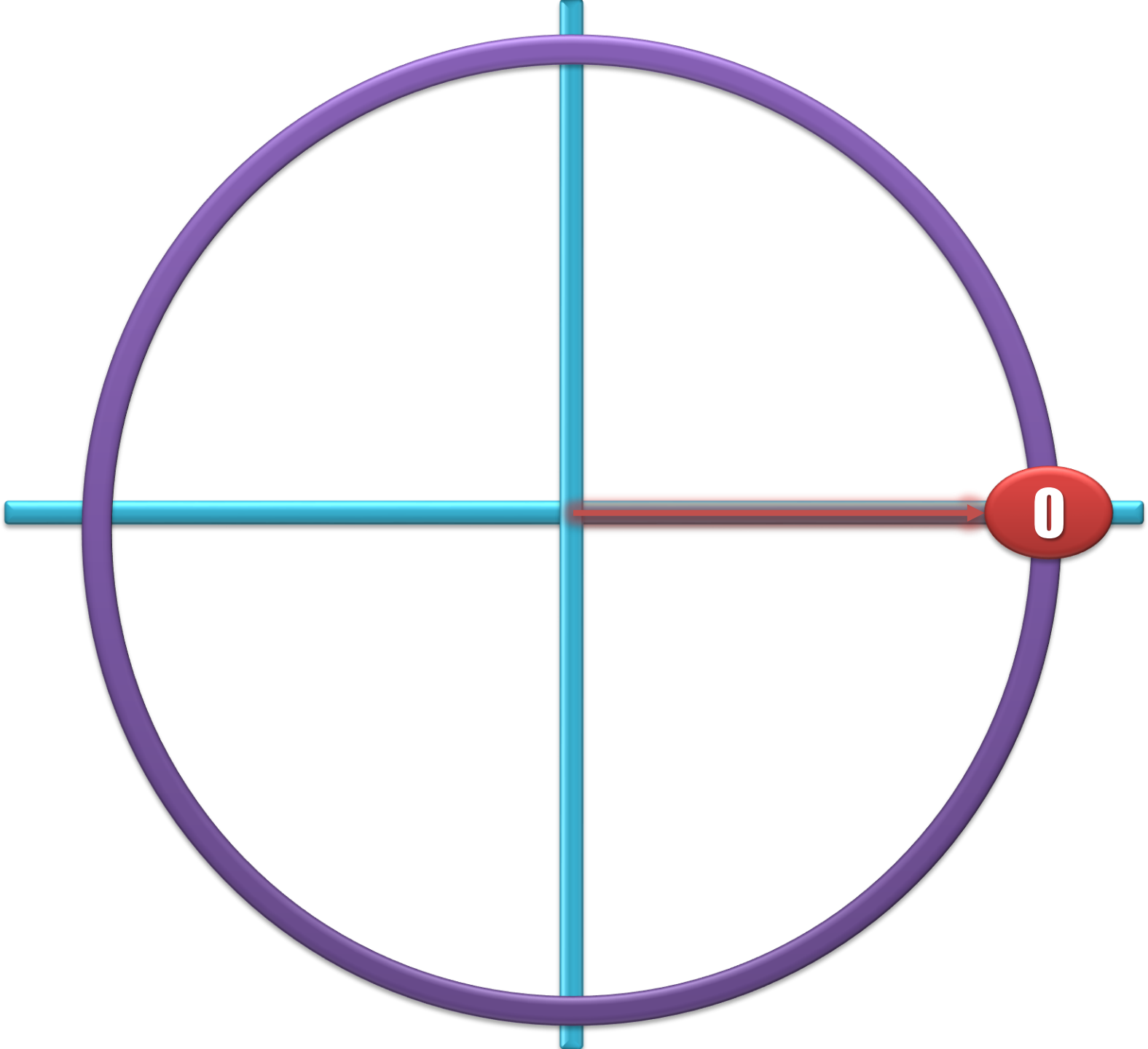
Lecture 10:

Modeling

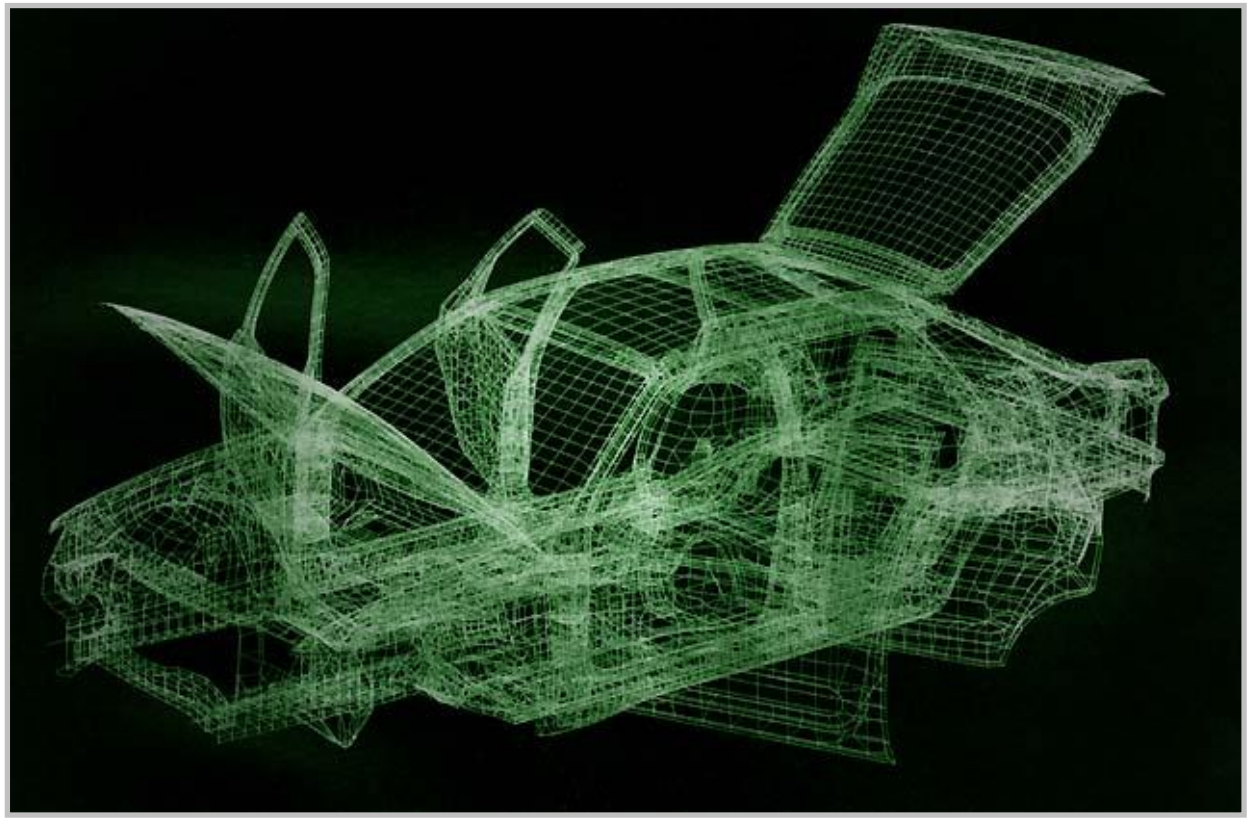


What you will learn…

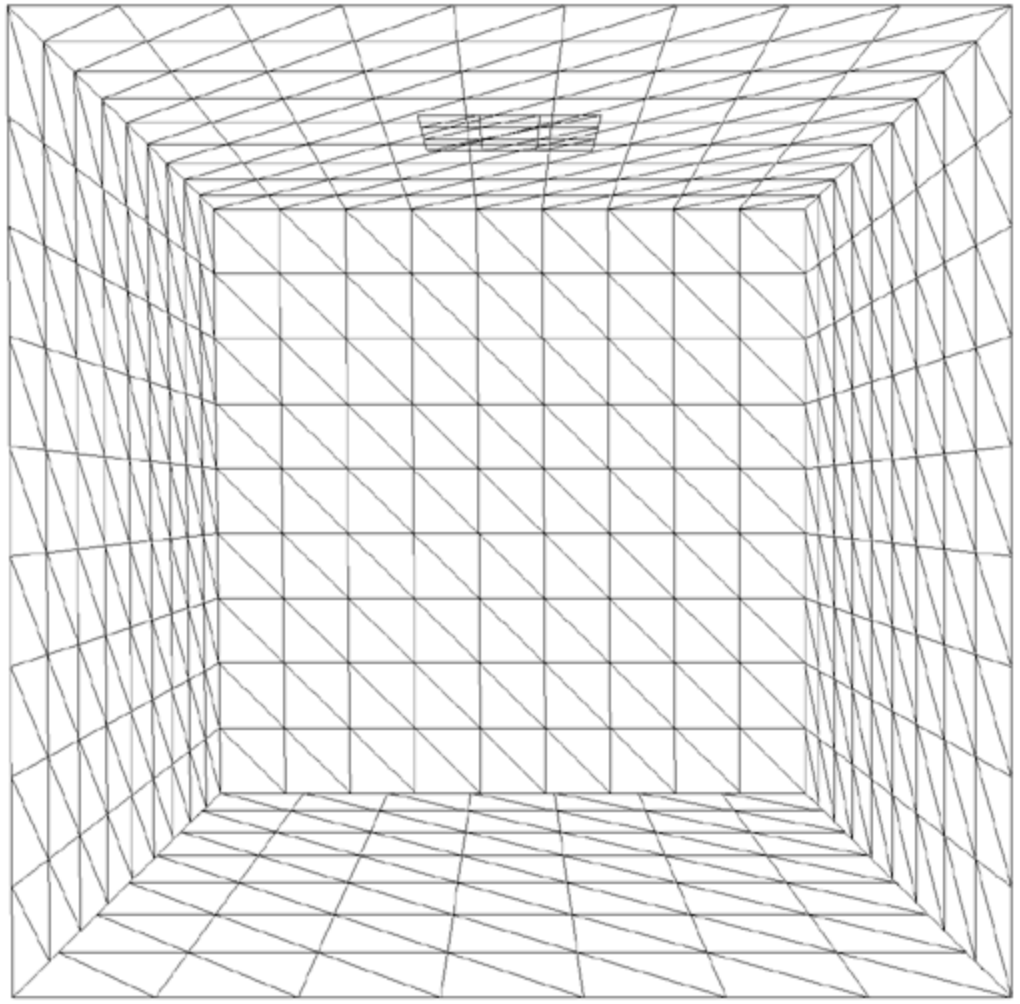
- Geometric Modeling
- Meshes
- Piecewise Continuous Curves
- Surfaces


0

Mesh Models

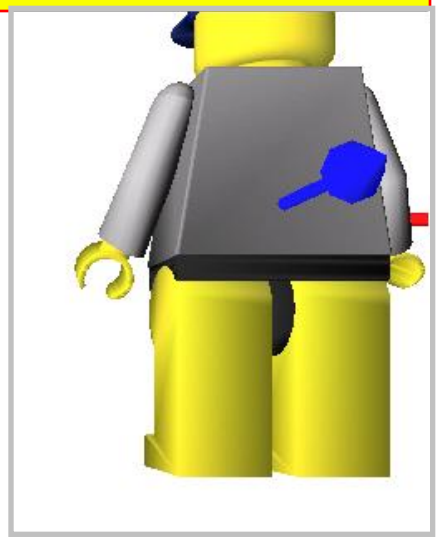


Mesh Models: Coarse

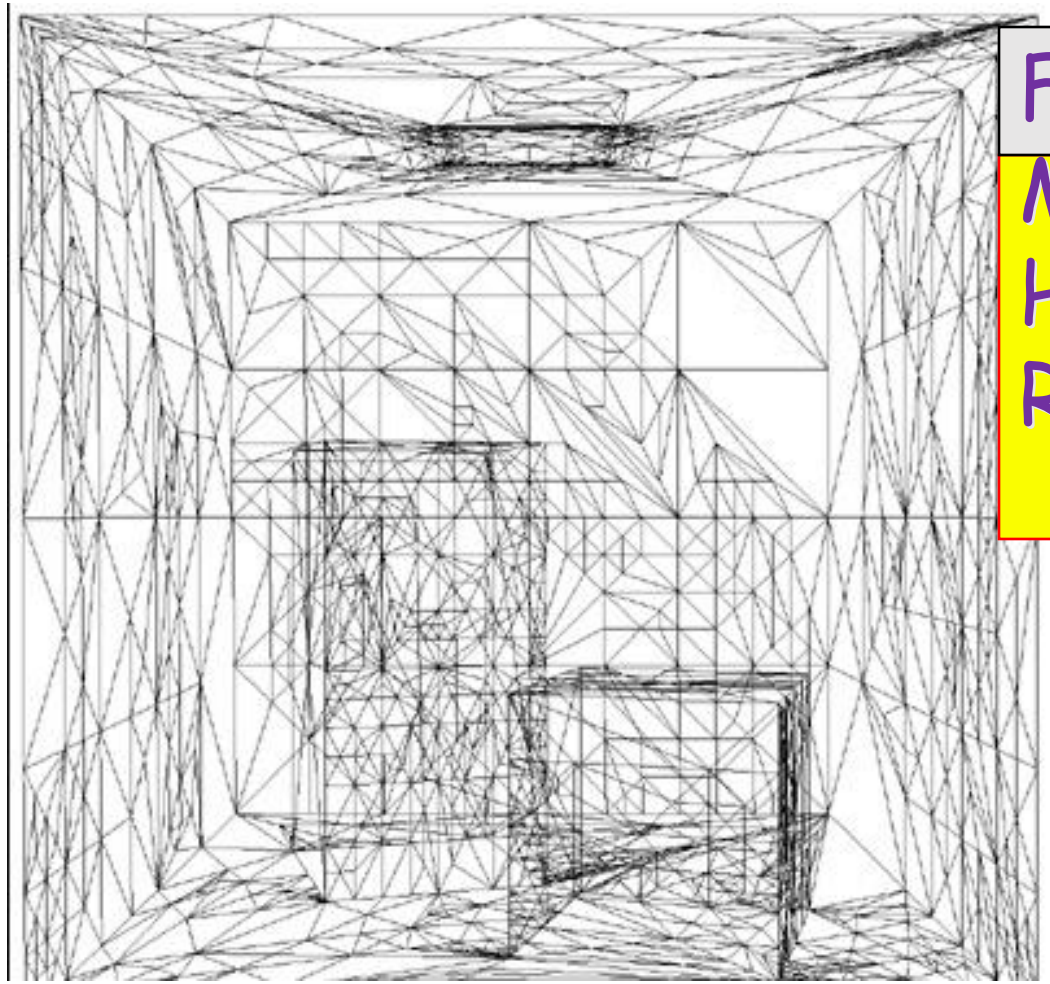


Coarse Meshes

Fewer elements
Low triangle count
Coarse model



Mesh Models: Fine

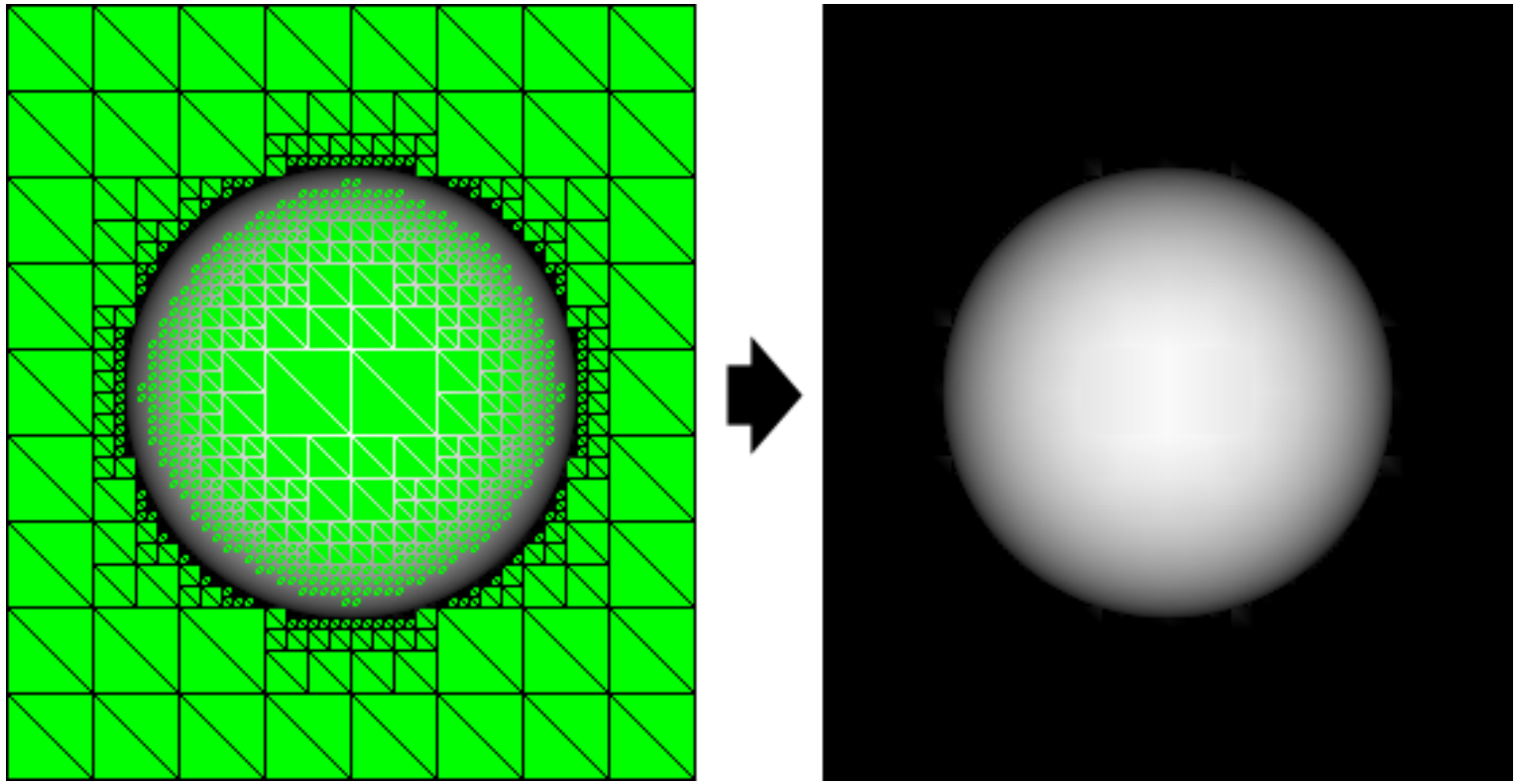


Fine Meshes

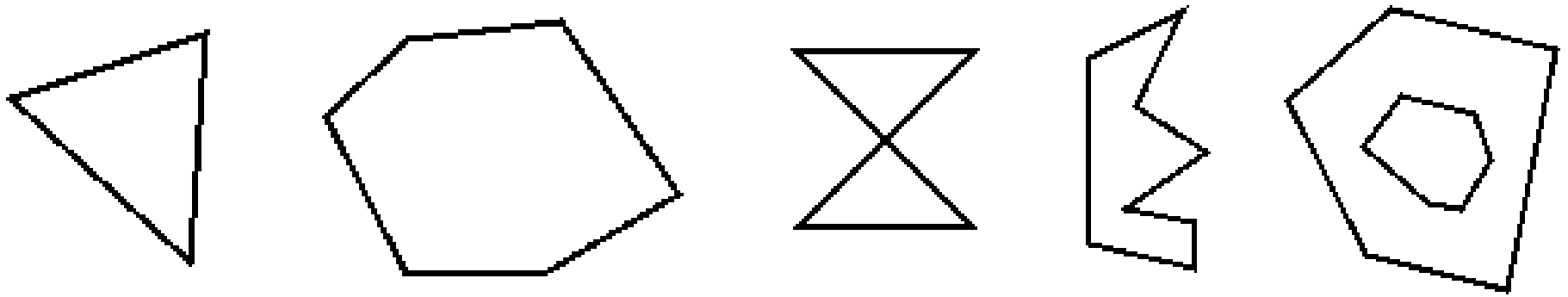
More elements
High triangle count
Refined model

Adaptive Subdivision

- Subdivide along the high edge or curvature density:



Valid and invalid polygons



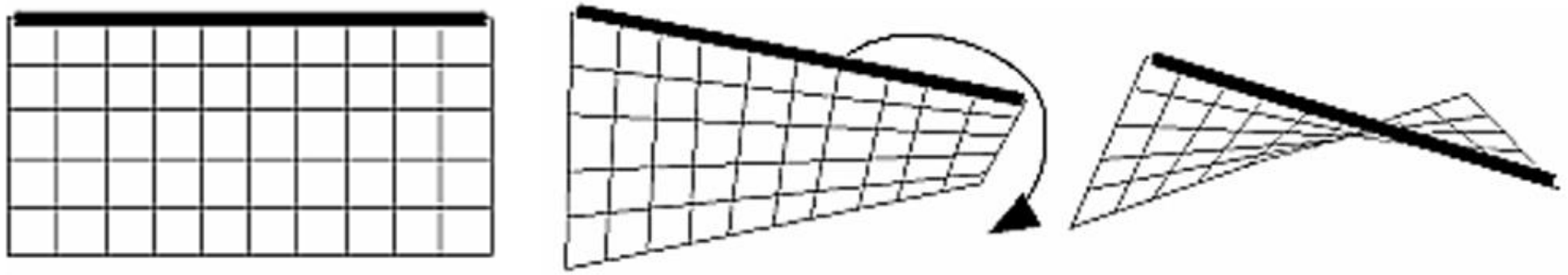
Valid polygons

For rendering

Invalid polygons

For rendering

Non-planar Polygon Elements

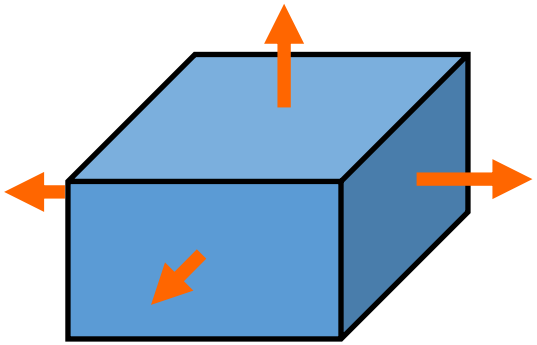


May lead to ruled surfaces with more than one normals

Face Normals

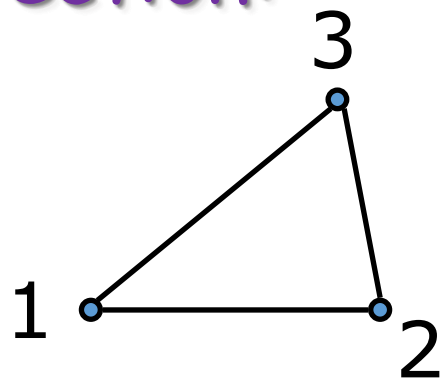
Normal vector to a face:

a vector perpendicular to the face, pointing toward the front.



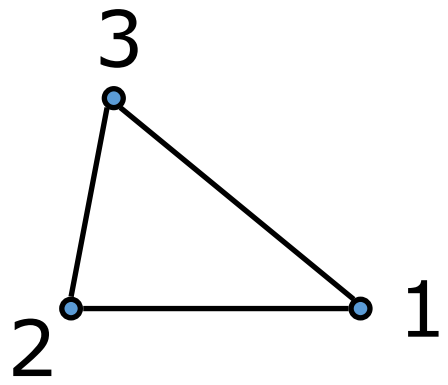
Triangle “Face”?

Face Direction:



CCW: “front” faces toward you

Counter-ClockWise

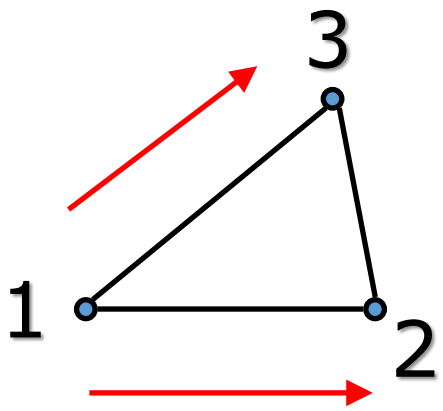


CW: “front” faces away from you

ClockWise

For General Polygons

Calculation of the positive or negative polygon face values.



Newell's Formula

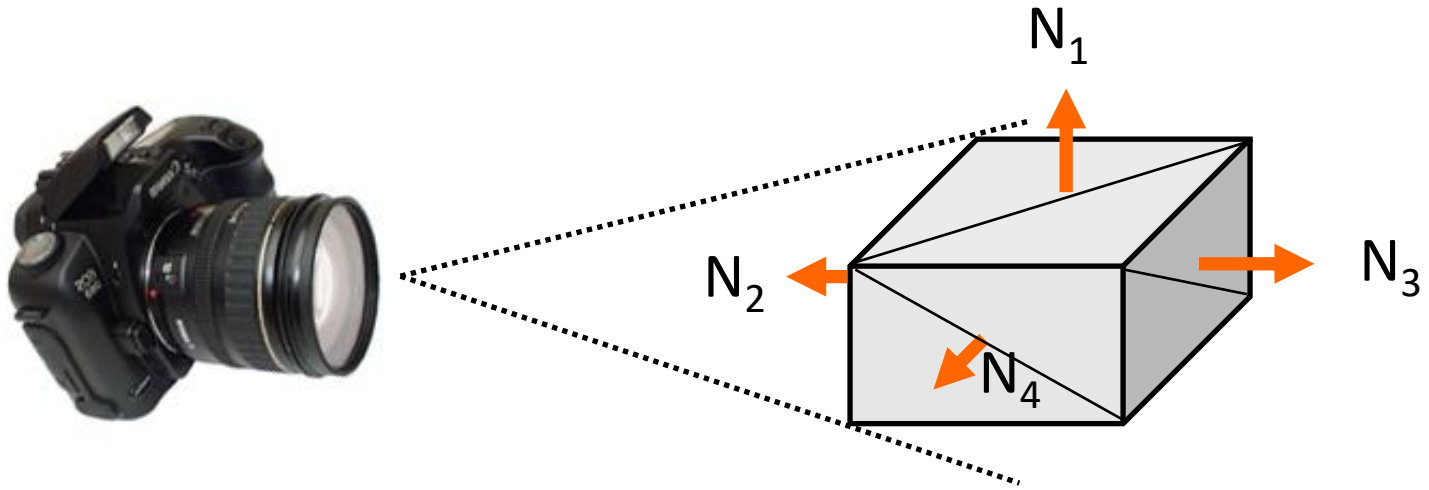
$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_i y_{i \oplus 1} - x_{i \oplus 1} y_i$$

where

$$i \oplus 1 = (i+1) \bmod n$$

Viewing Faces

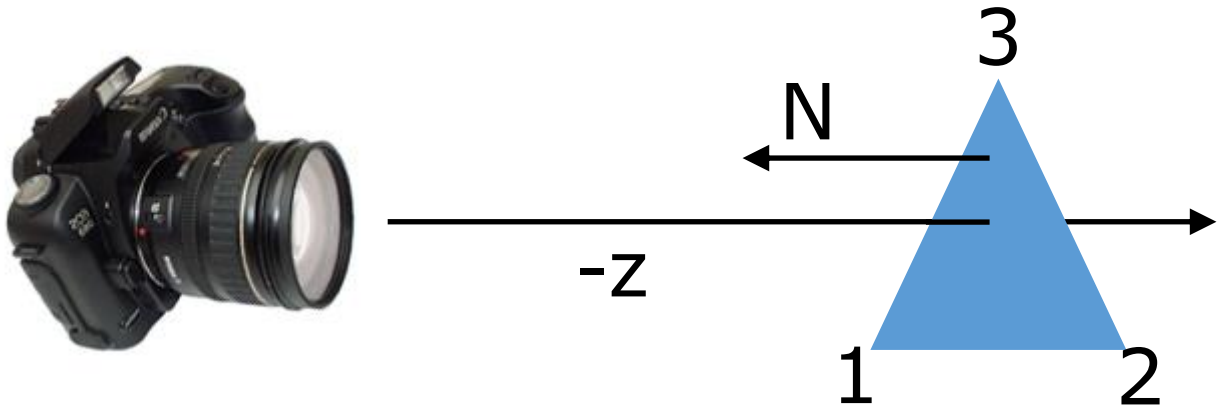
Viewing Direction



Which direction does a triangle "face"?

Which triangles “face the viewer”?

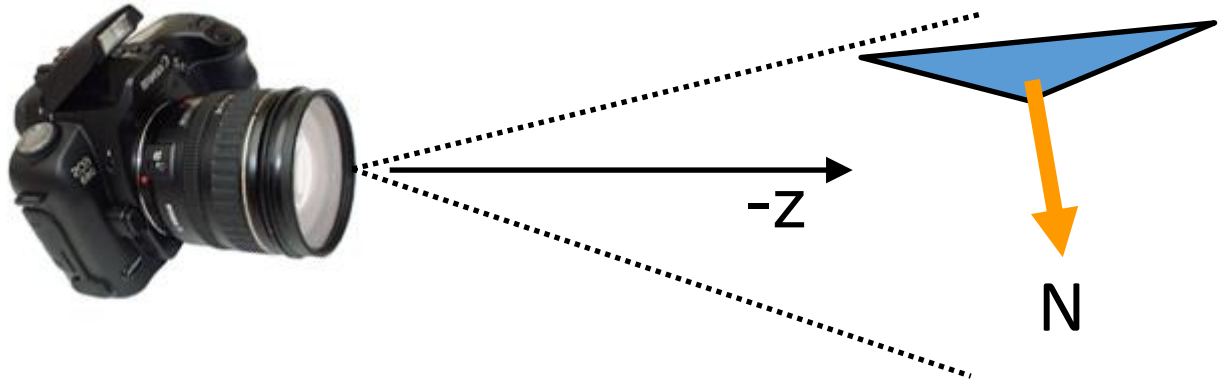
In the *eye coordinates*, how could I decide if a triangle faces the viewer?



What’s wrong with this approach?

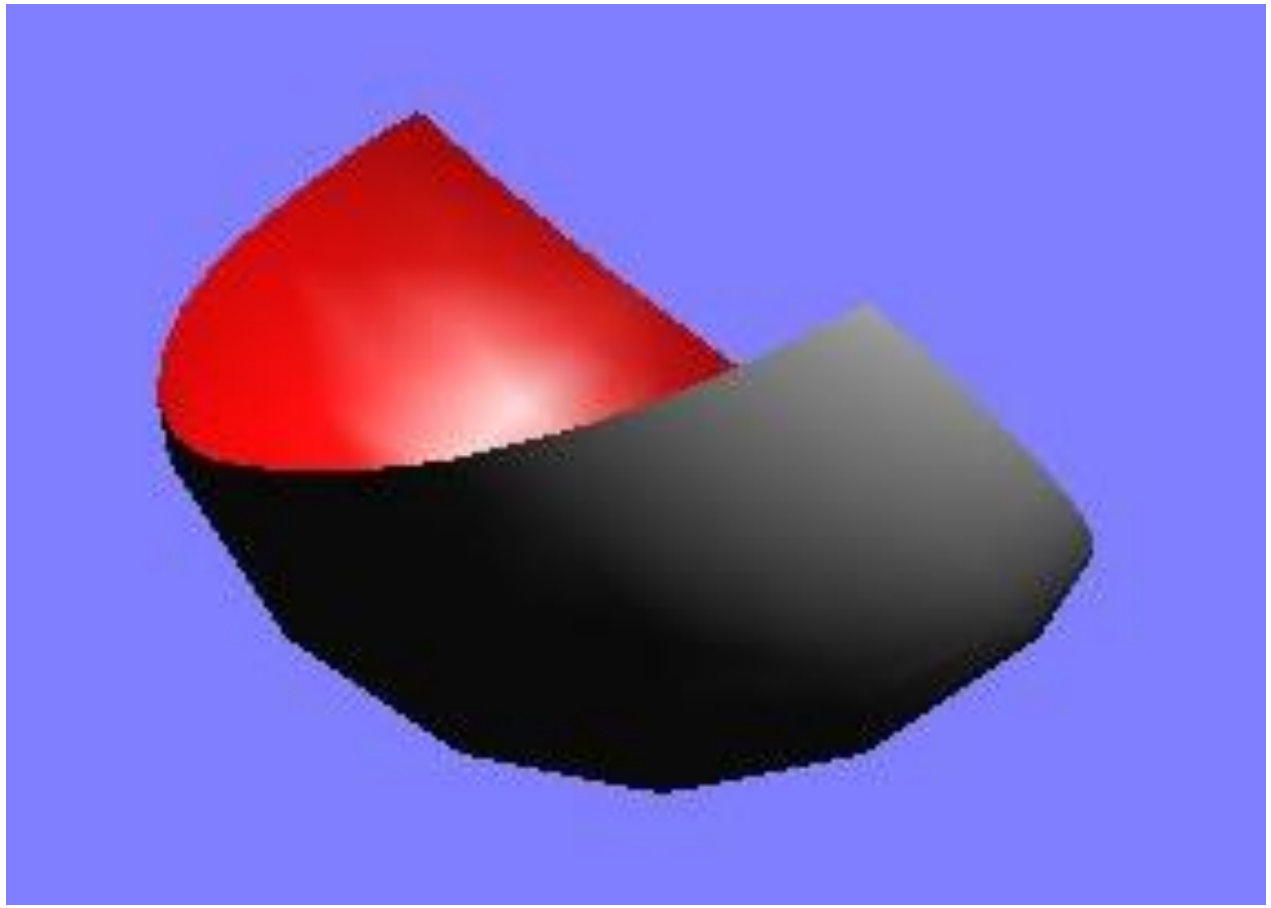
Normal Projection

Problem:



The polygon appears to face away from the eye in eye coordinates, but projection changes everything!

Front-back Faces



Culling in WebGL

Just before calling the gl.drawElements function

```
gl.enable(gl.CULL_FACE);
gl.cullFace(gl.FRONT);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_BYTE, 0);

gl.enable(gl.CULL_FACE);
gl.cullFace(gl.BACK);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_BYTE, 0);
```

Objects:
Data Structures

Representing Objects

A Mesh Data Structure

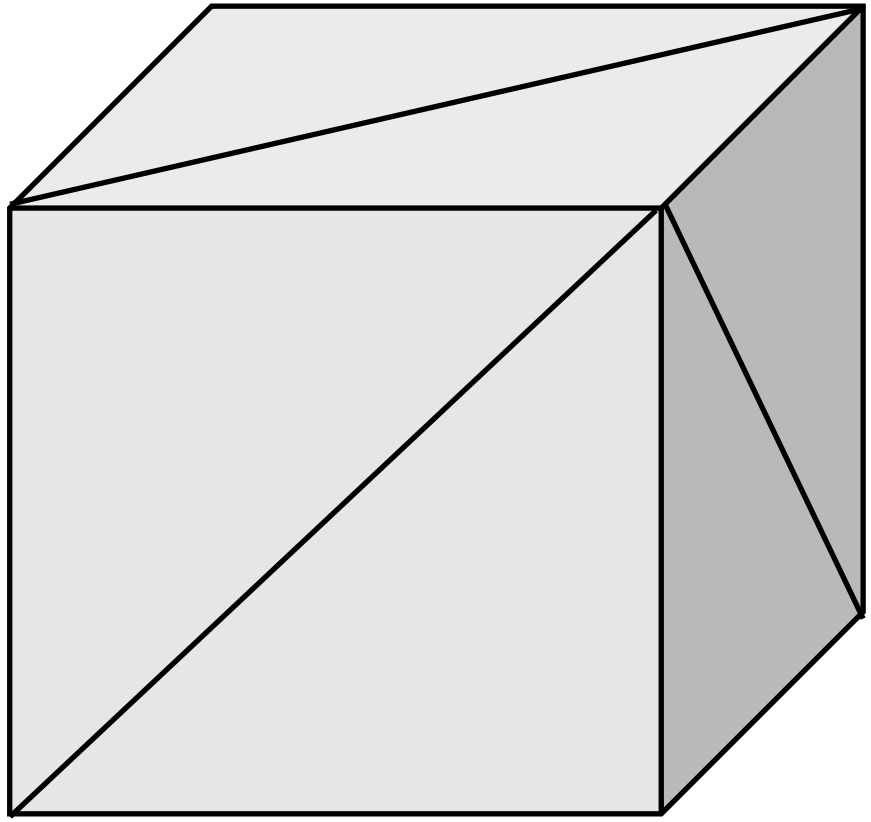
```
struct vertex {
  float x,y,z;
}
struct triangle {
  vertex a,b,c;
}
struct object {
  unsigned int nTriangles;
  triangle* triangles;
}
```

Duplications!

One triangle shares two vertices!

What's wrong with this approach?

How many vertices in a cube?



Redundant Vertices?

What are some disadvantages of having redundant vertices?

Coordinate Floating Point Mismatch:

v =[1.0, 1.0, 0.00000014]

v' =[1.0, 1.0, 0.00000132]

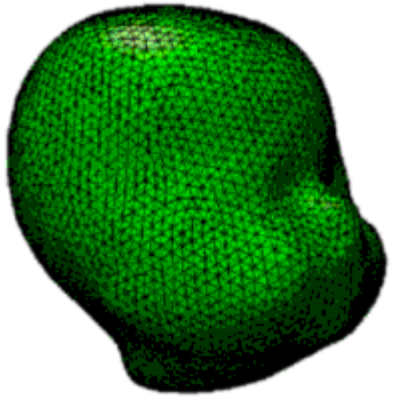
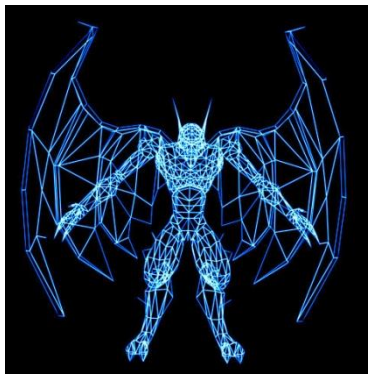
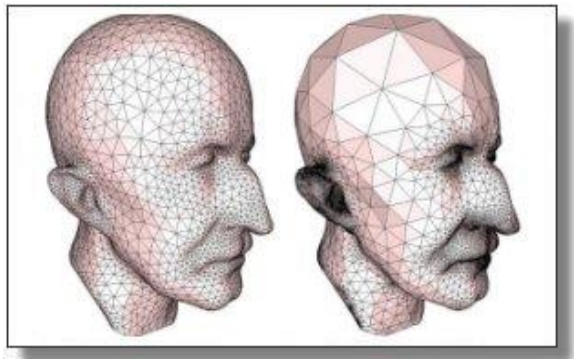
A better mesh data structure

```
struct vertex {
  float x,y,z;
}
struct triangle {
  unsigned int a,b,c;
}
struct object {
  unsigned int nVertices;
  vertex*      vertices;
  unsigned int nTriangles;
  triangle*    triangles;
}
```

3 integer indexes

Change to
a list of vertices

Mesh File Formats

Popular formats: { *.3ds, *.obj, *.stl }

Most formats add more data per vertex:

- Surface normals
- Color information
- Texture coordinates

When aren't meshes quite right?

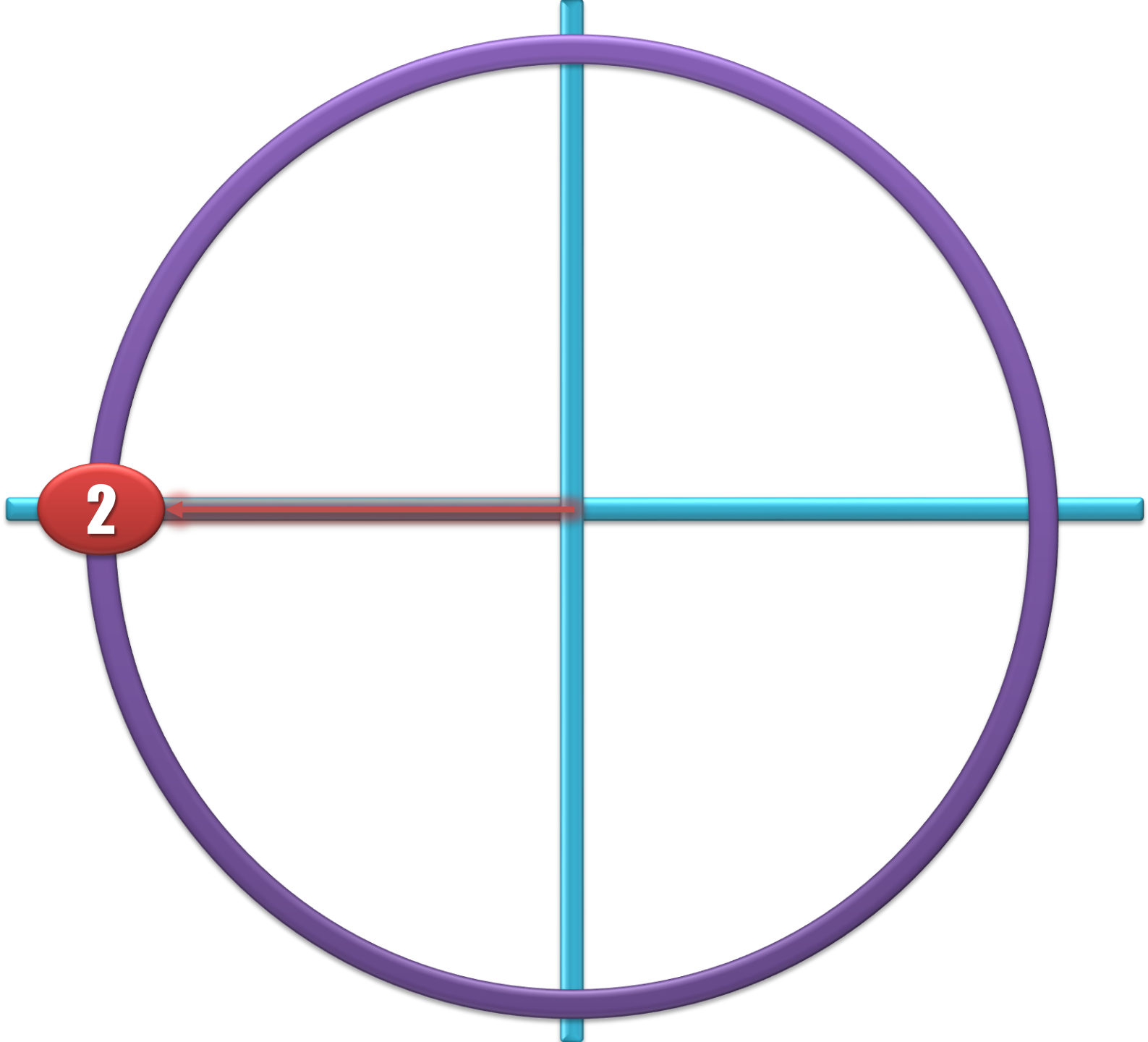
Which of these objects could be represented more efficiently with another approach?

Why not always store our objects as meshes?

- Because…

Regularity and symmetry play a very important role in object representations!

For Example:
Circles, Cylinders, Spheres


2

Curves

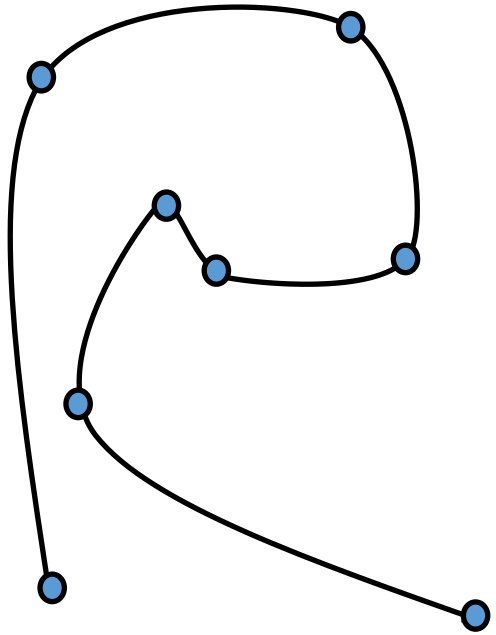
Why Smooth Curves?

Cars!

Design!

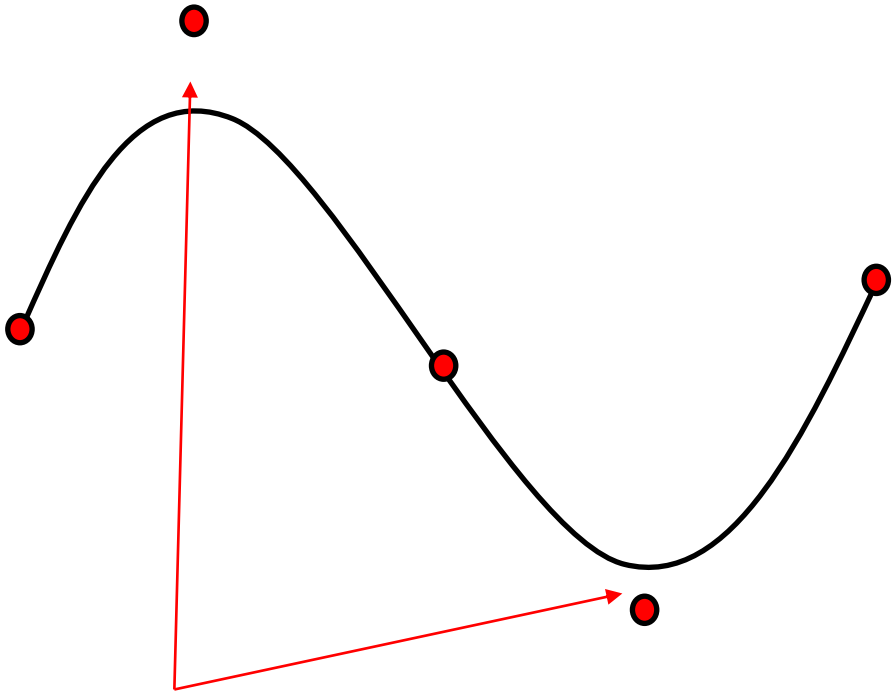
Engineering!

Control Points



Interpolating

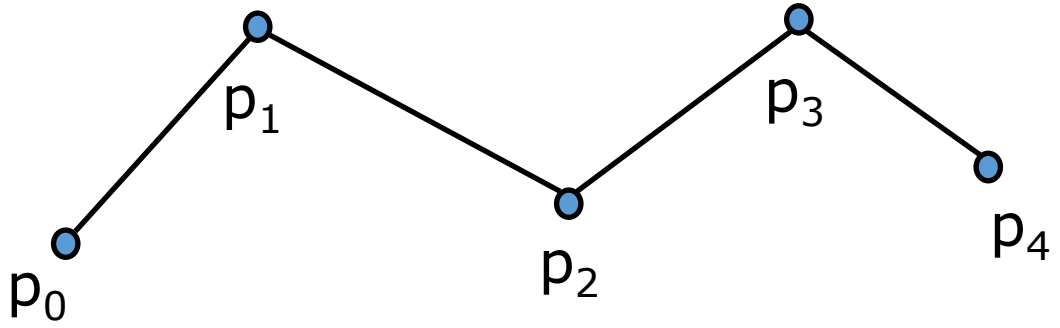
All points on the curve!



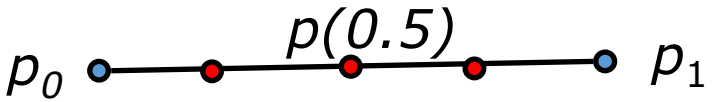
Approximating

Some points not on the curve!

Linear Interpolation

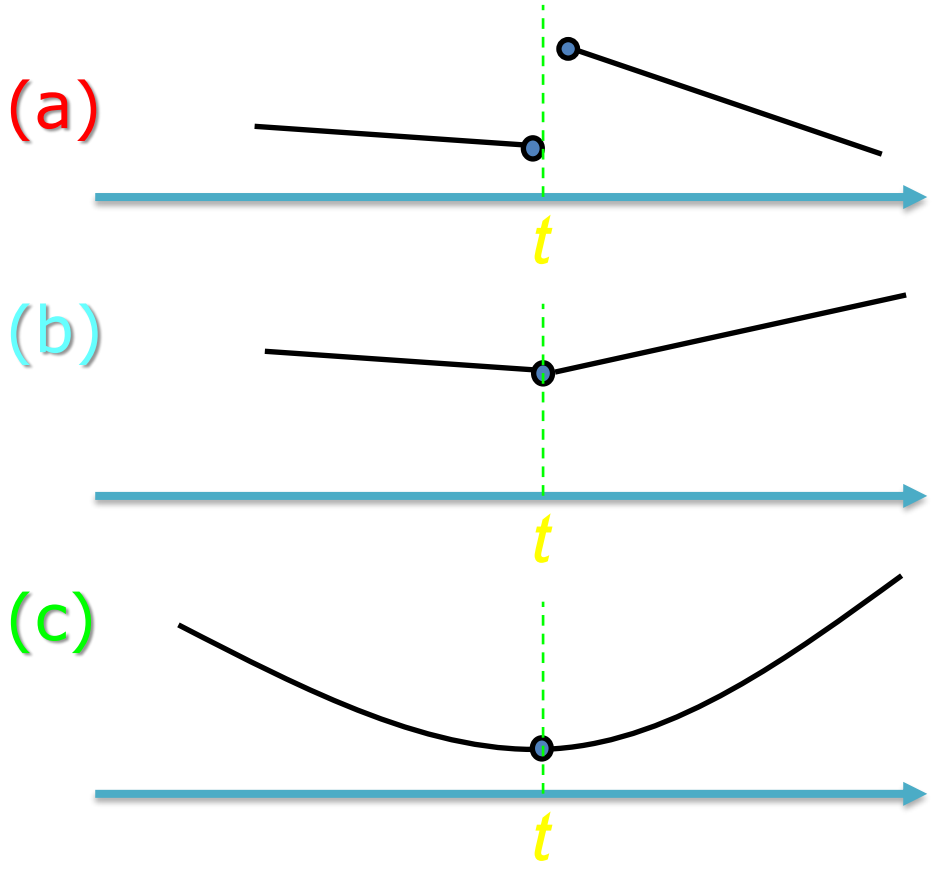


$$p(t) = (1-t)\cdot p_{i-1} + (t)\cdot p_i$$



Curve Continuity

A curve or surface is said to be C^n continuous at a point t if its n^{th} derivative at that point is continuous.

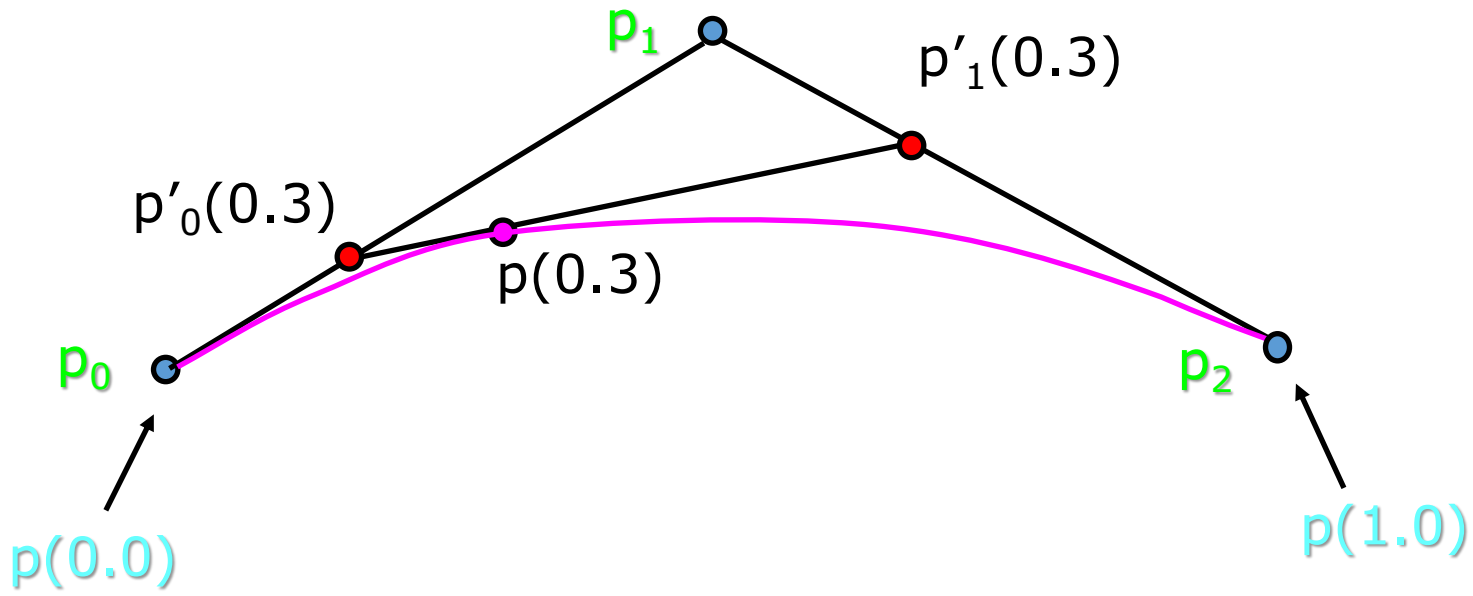


not c^0 continuous

c^0 continuous,
not c^1 continuous

c^0 and c^1 continuous,
hard to tell about
higher orders…

de Casteljau's Algorithm



Based on interpolating between interpolation points!

Bezier Curves

Finding the closed-form expression for **quadratic Bezier** curves:

$$p'_0(t) = (1-t)p_0 + (t)p_1$$
$$p'_1(t) = (1-t)p_1 + (t)p_2$$

$$p(t) = (1-t) \cdot p'_0(t) + t \cdot p'_1(t)$$

$$p(t) = (1-t)^2 \cdot \mathbf{p_0} + 2t(1-t) \cdot \mathbf{p_1} + t^2 \cdot \mathbf{p_2}$$

Bezier Curves

For a Bezier curve with L+1 control points p_0, p_1, p_2, ... p_L, the curve is:

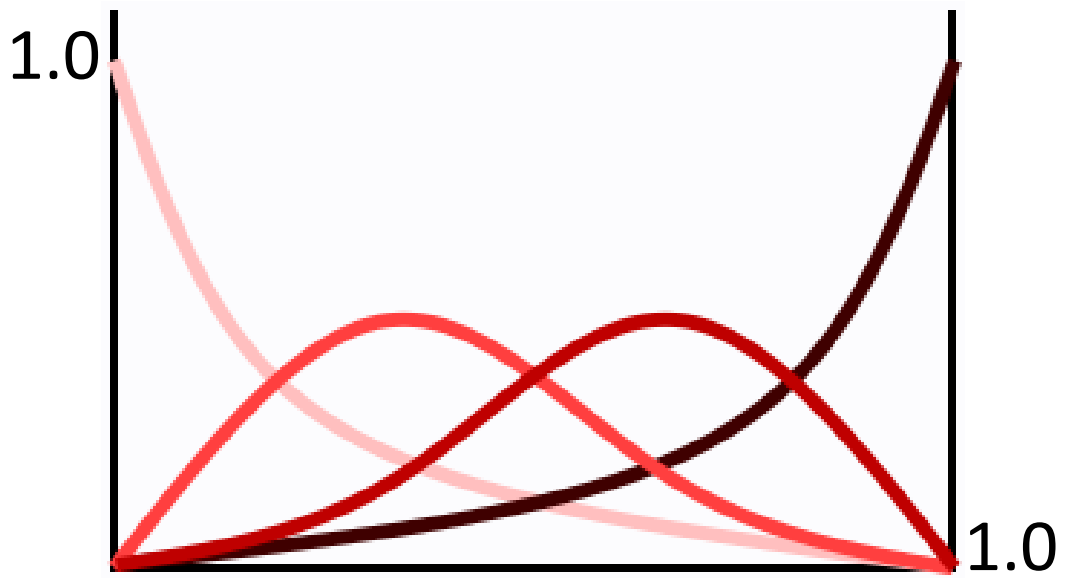
$$p(t) = \sum_{k=0}^{L} p_k B_k^{\,L}(t)$$

$B_k^{\,L}(t)$ is a ***Bernstein polynomial of order k***:

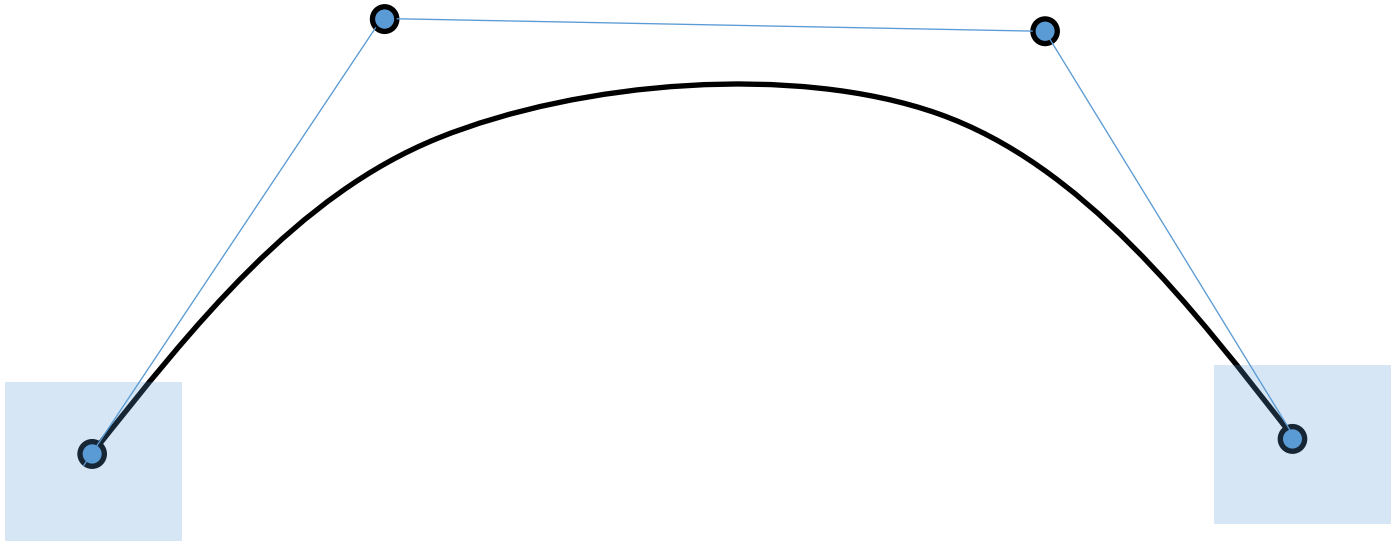
$$B_k^{\,L}(t) = \binom{L}{k}(1-t)^{L-k} \cdot t^k$$

Blending Functions

$$p(t) = \sum_{k=0}^{L} p_k B_k{}^{L}(t)$$



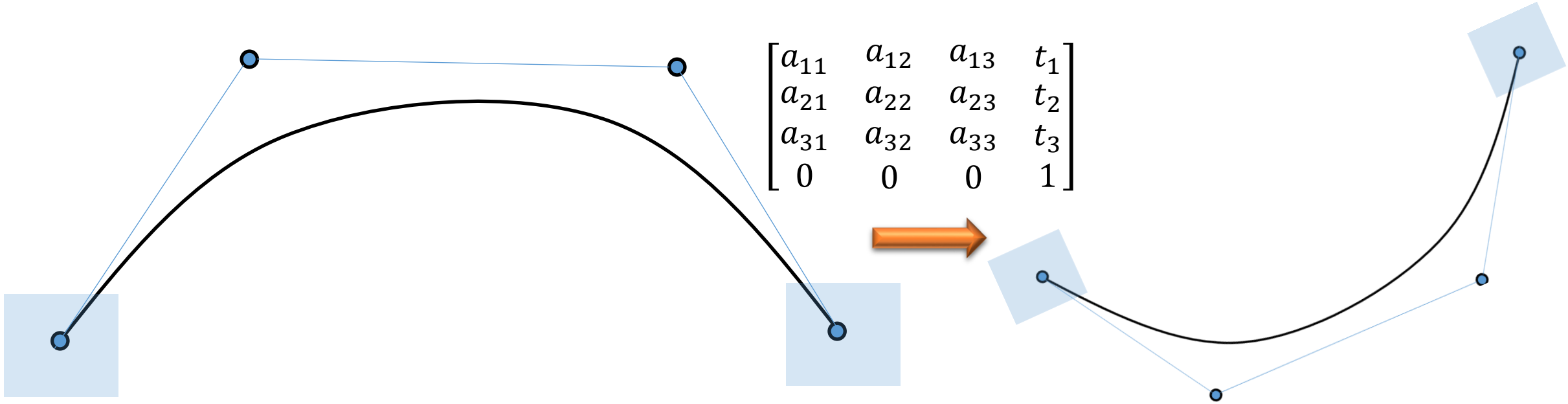
Bezier Curve Properties



1. Endpoint interpolation

A Bezier curve always passes through the first and last control points

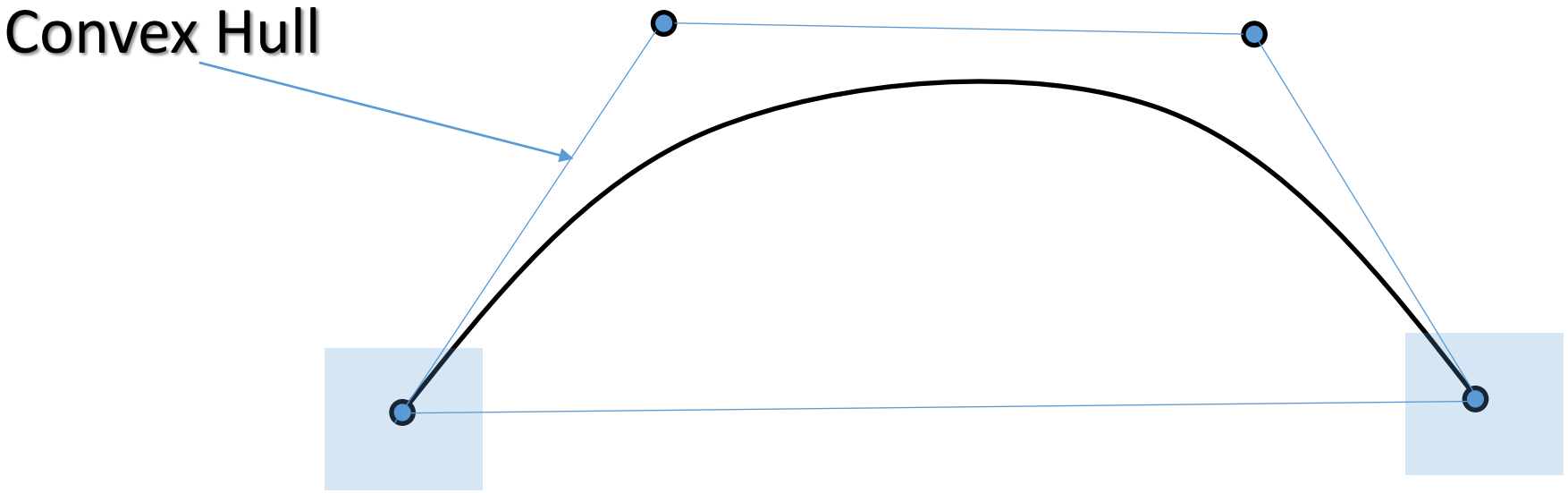
Bezier Curve Properties



2. Transformation Invariance

Transforming the control points transforms the curve "correctly"

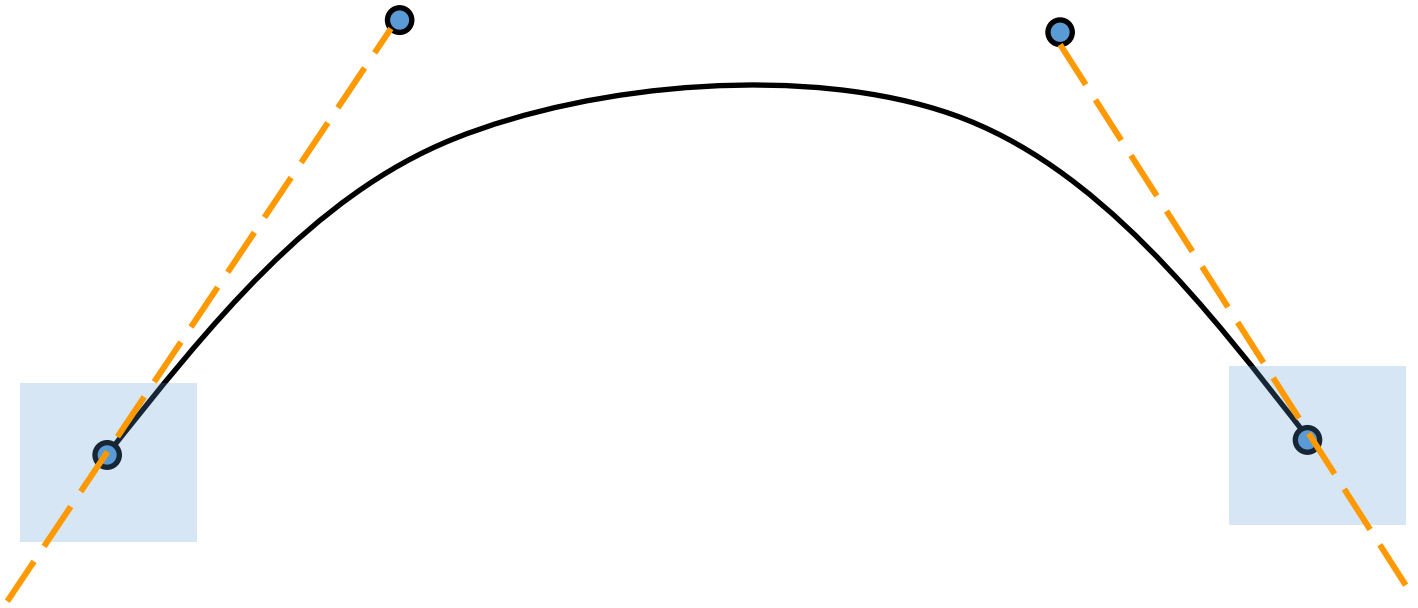
Bezier Curve Properties



3. Convex Hull Containment

A Bezier curve always stays “inside” the convex hull of the control points

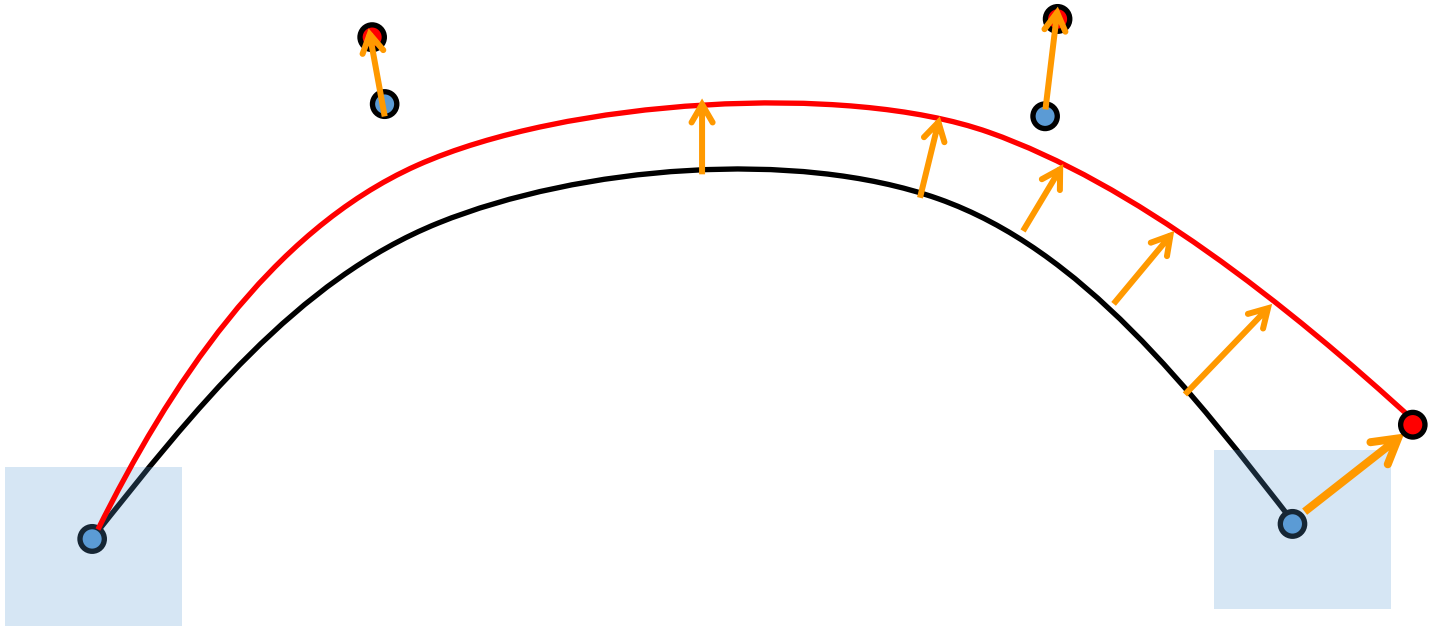
Bezier Curve Properties



4. Smoothness at End Points

A Bezier curve endpoint tangent equals the slope of the last “control segment”

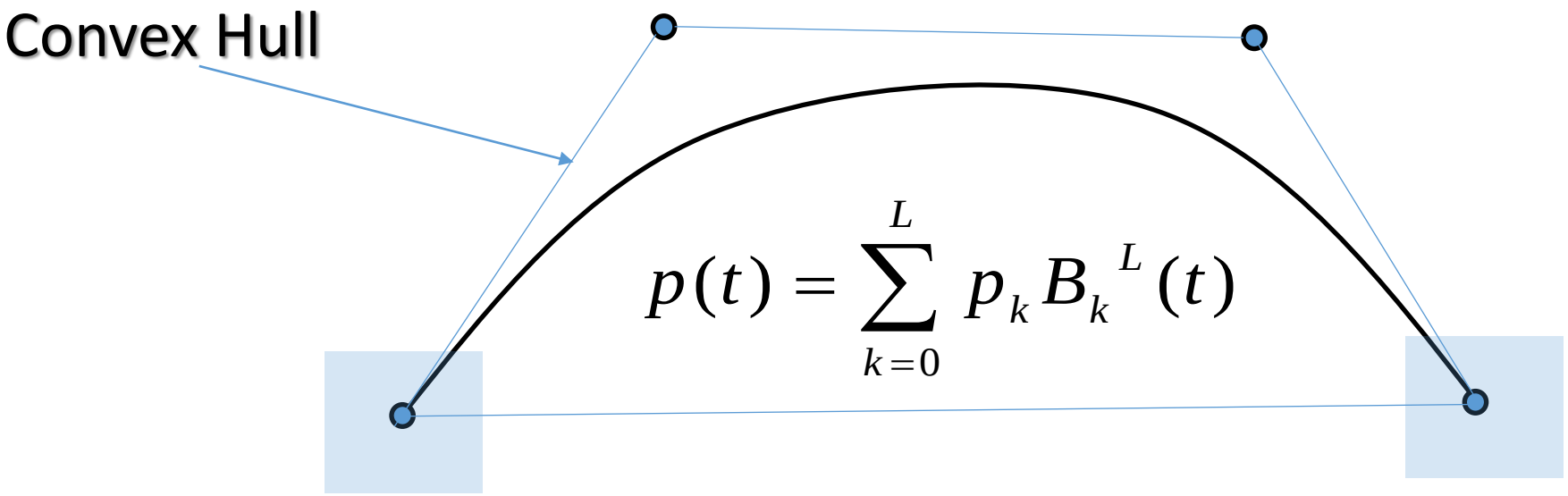
Bezier Curve Drawbacks



1. Lack of local control

Moving any point changes the entire curve

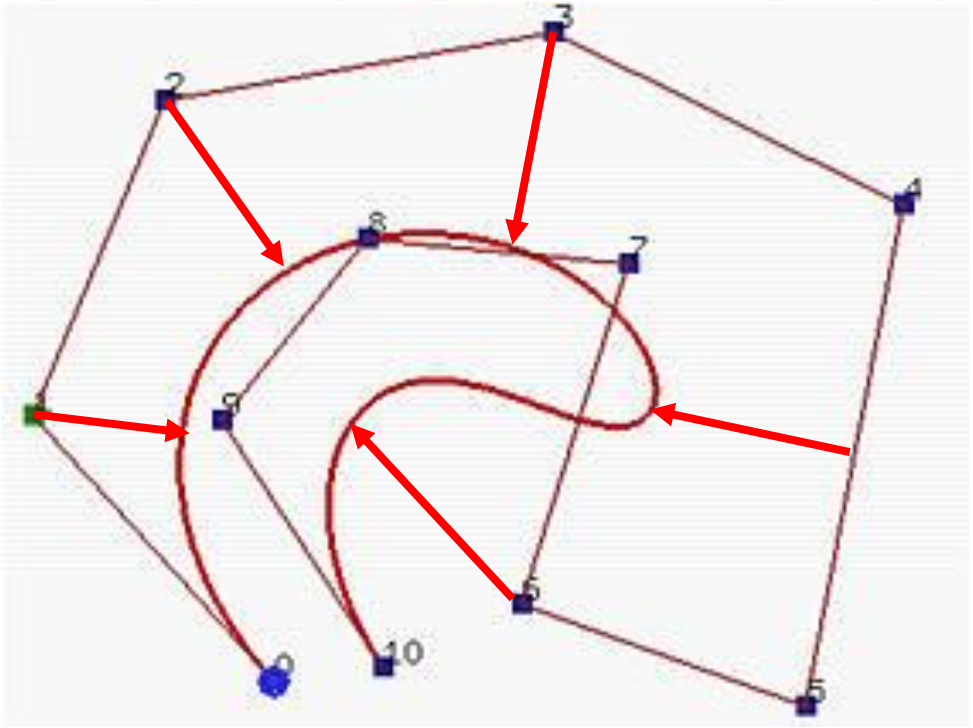
Bezier Curve Drawbacks



2. Computational Complexity

Evaluating high-degree polynomials is impractical

Bezier Curve Drawbacks

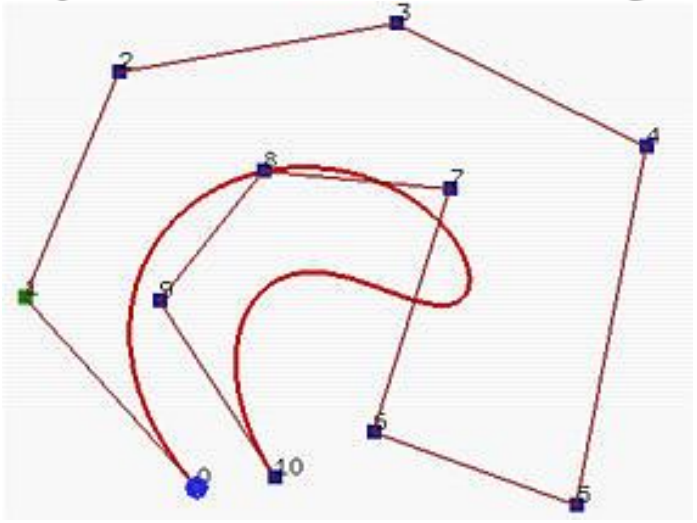


3. Poor Fitting

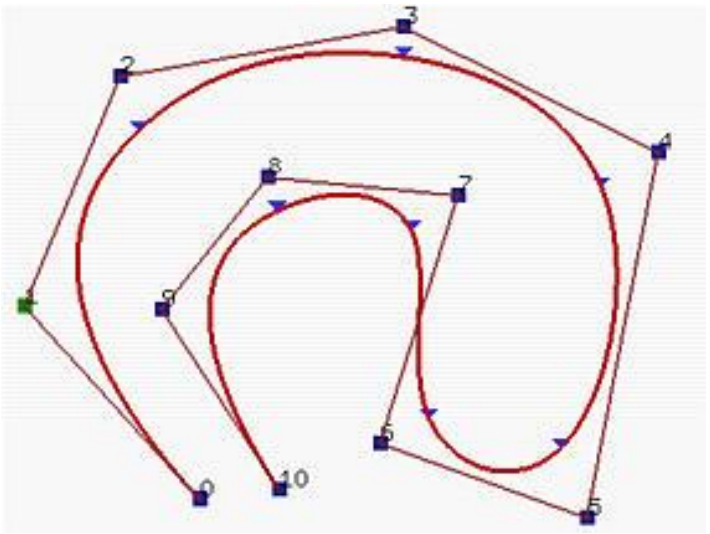
Bezier curves sometimes stray far from the control polygon

Solution: Cubic B-Splines

- Curves with C^0, C^1, and C^2 continuity, constructed from piecewise polynomial functions
- Translation: stitch together a bunch of cubics without funny artifacts at the joints



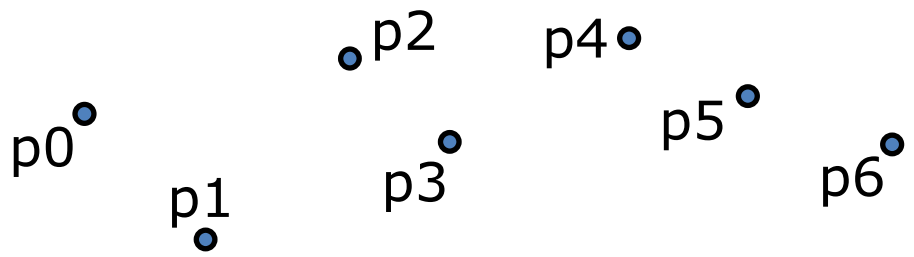
Bezier curve of degree 10



Cubic B-Spline curve (degree 3)

Cubic B-Splines

- Segments of the curve are each influenced by *four* control points

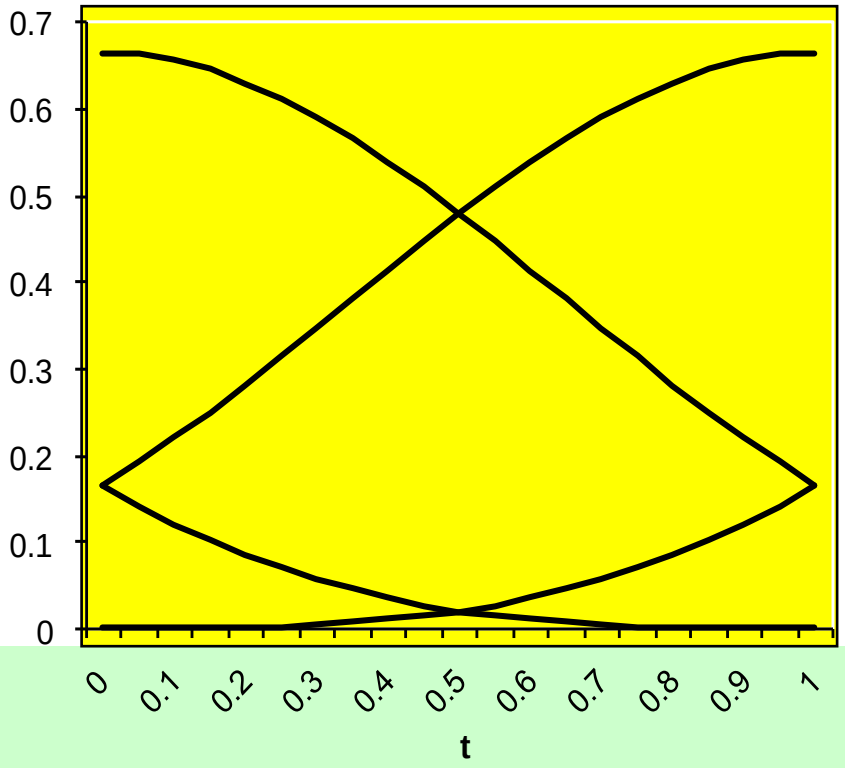


- First segment: cubic curve using p0, p1, p2, p3
- Next segment: cubic curve using p1, p2, p3, p4
- Next segment: cubic curve using p2, p3, p4, p5
- Next segment: cubic curve using p3, p4, p5, p6

Cubic B-splines Properties

- Local Control
  - Moving or adding a control point doesn't affect the whole curve
- Low degree
  - Cubic splines are easy to compute
  - Cubic splines are used to describe many complex curves
- All the nice properties of Bezier curves

Cubic B-Spline Bases

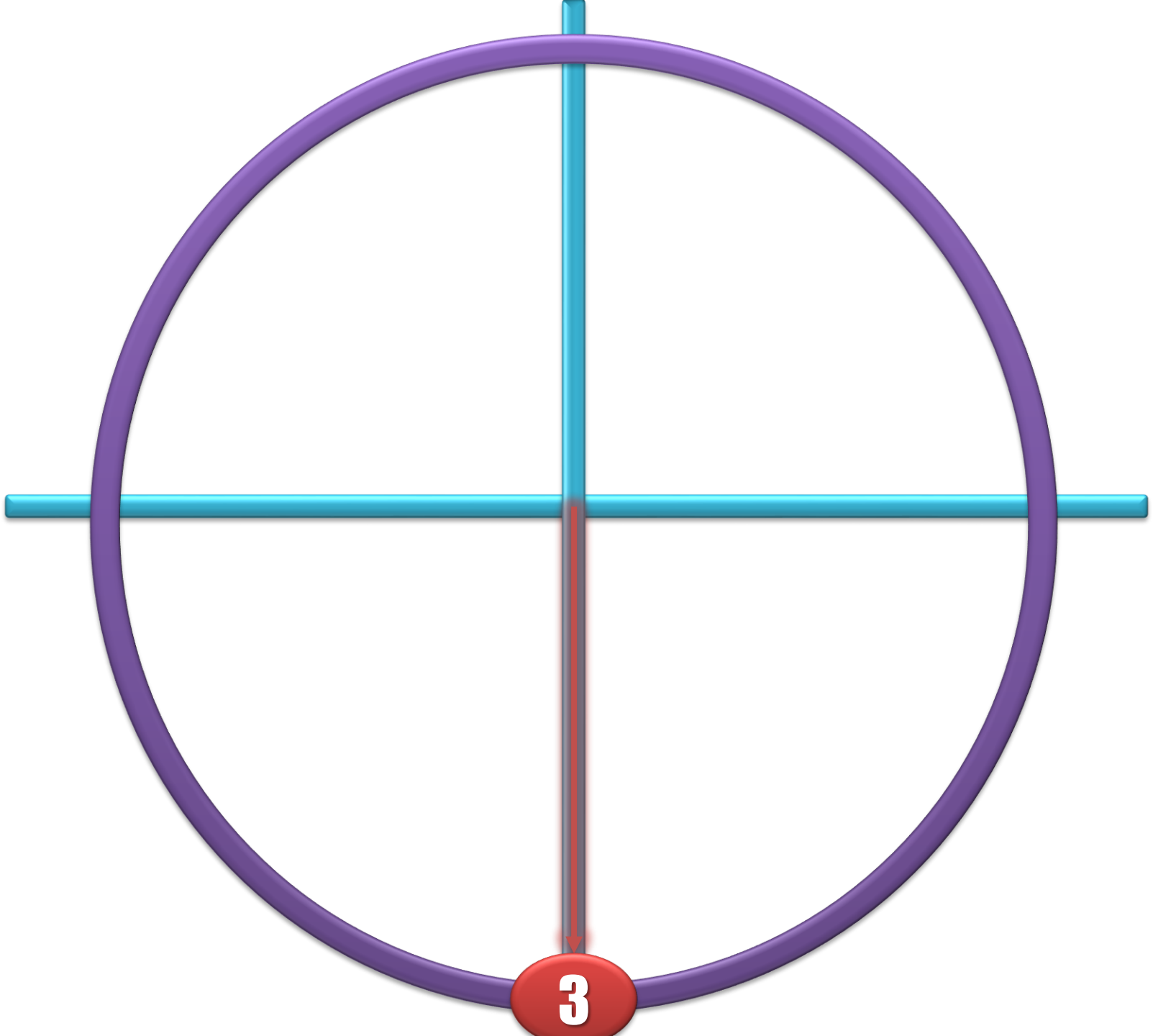


$$p(t) = \sum_{i=0}^{3} P_i B_{i,4}(t)$$

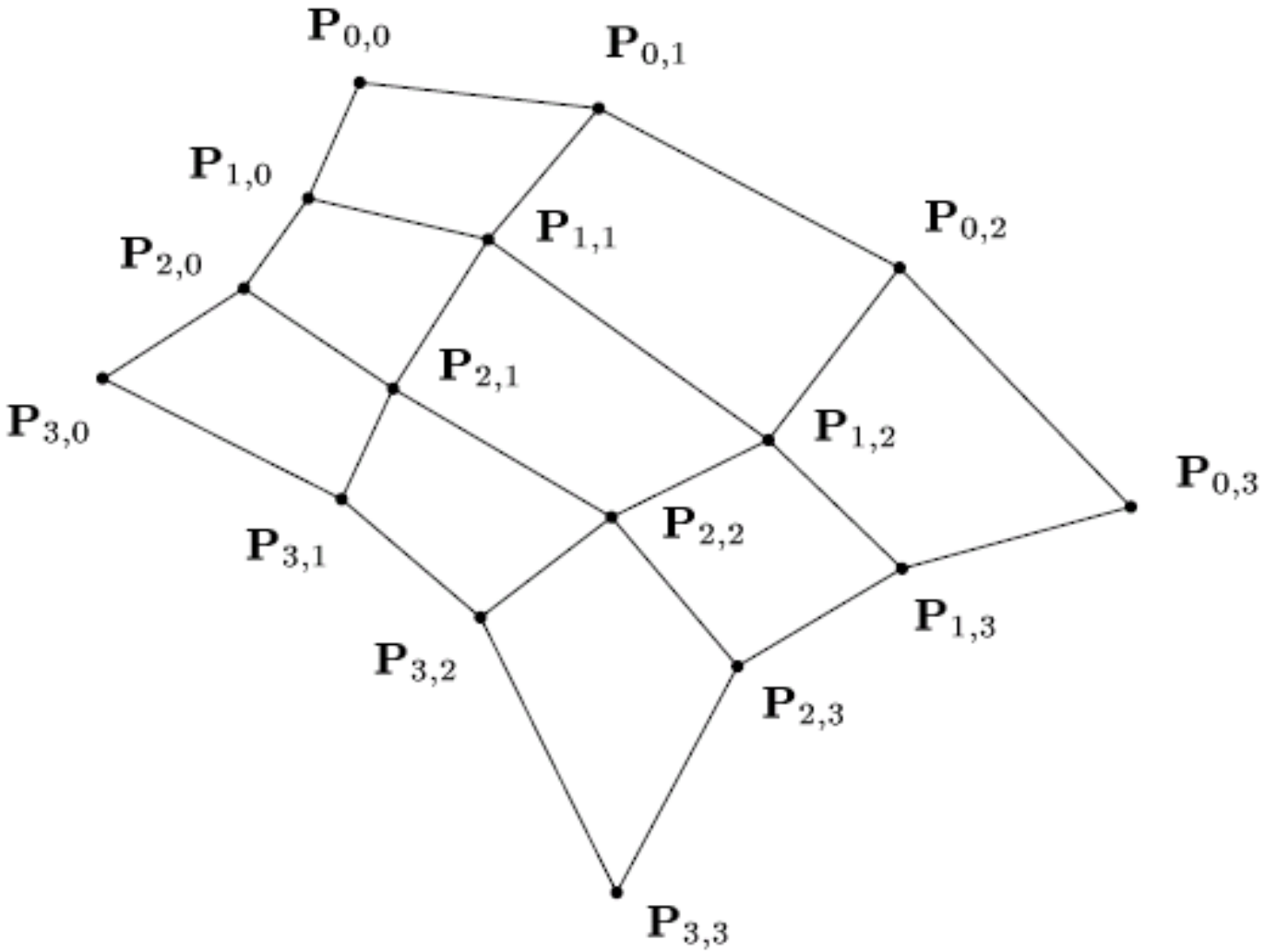
$$= P_0 \frac{1}{6}\left(1 - 3t + 3t^2 - t^3\right) + P_1 \frac{1}{6}\left(4 - 6t^2 + 3t^3\right) + P_2 \frac{1}{6}\left(1 + 3t + 3t^2 - 3t^3\right) + P_3 \frac{1}{6}\left(t^3\right)$$

B-Splines Limitations

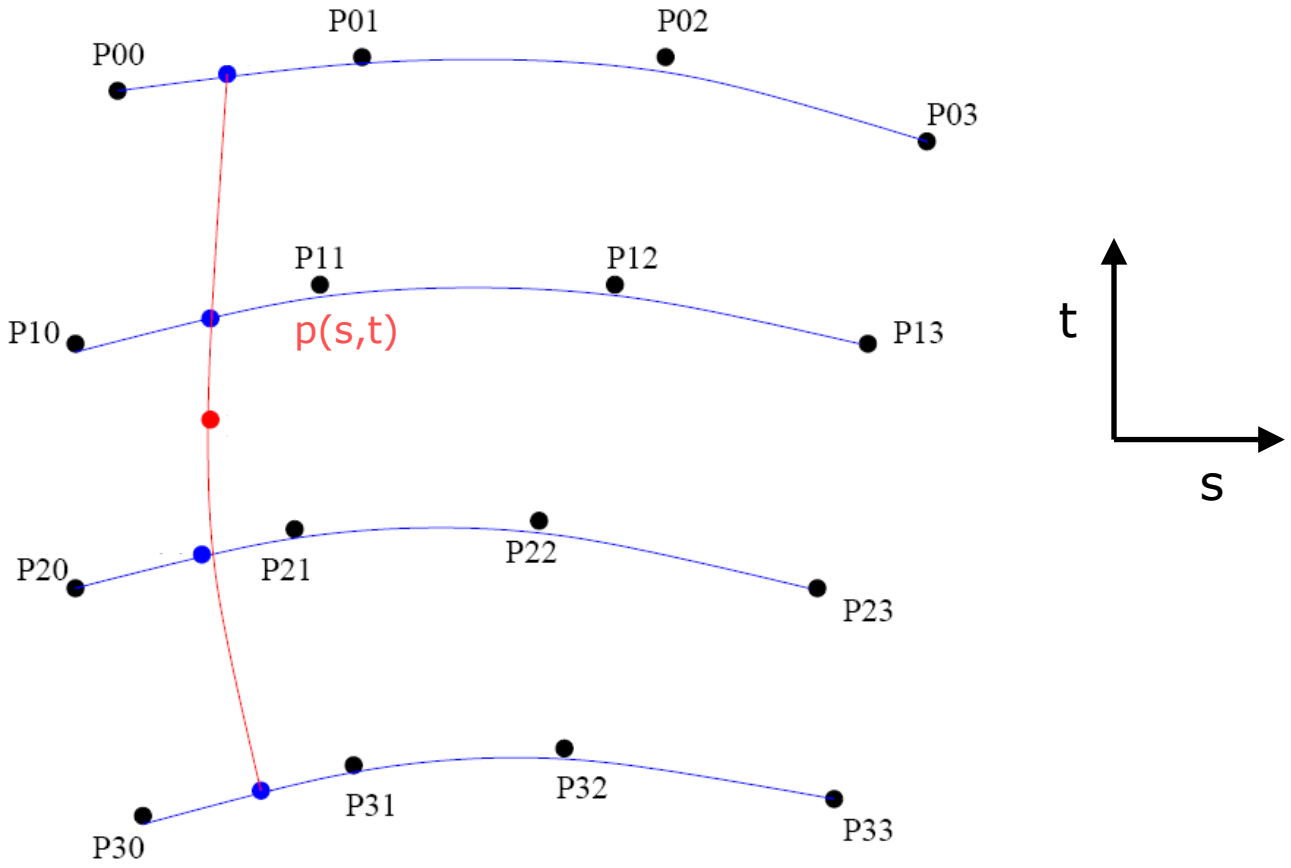
- Non-interpolating: Not guaranteed to pass through any of the control points (even the first and the last)
- Still B-splines can’t express all shapes (e.g. circles)
- Solution: Use NURBS (Non-Uniform Rational B-Splines)


3

Surface Control Mesh



Bezier Surface Patch



Bezier Surface Patch

For each Bezier curve:

$$p(t) = \sum_{k=0}^{L} p_k B_k^{\,L}(t)$$

Let the control points themselves be functions of another variable:

$$p(s,t) = \sum_{k=0}^{L} p_k(s) B_k^{\,L}(t)$$

Specifically, let those functions be Bezier curves:

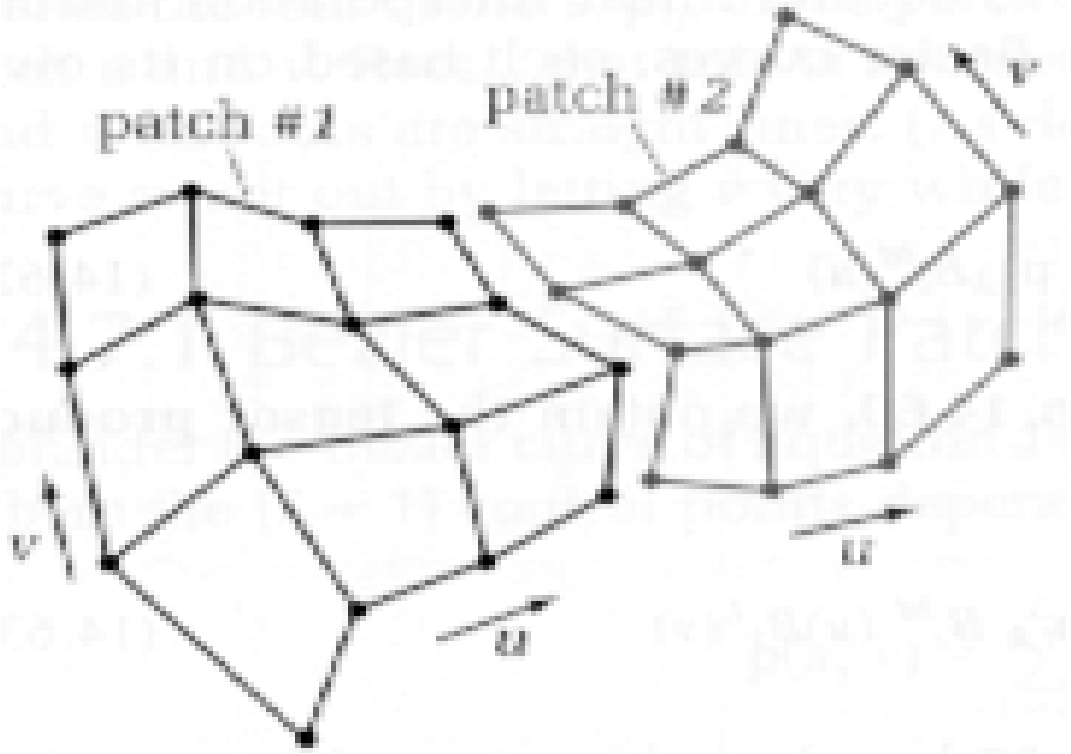
$$p_k(s) = \sum_{j=0}^{M} p_{j,k} B_j^{\,M}(s)$$

This gives us the function for a Bezier patch:

$$p(s,t) = \sum_{j=0}^{M} \sum_{k=0}^{L} p_{j,k} B_j^{\,M}(s) B_k^{\,L}(t)$$

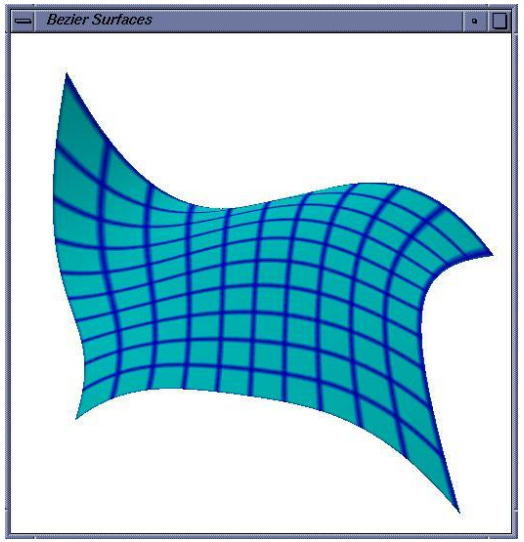
Stitching Surface Patches

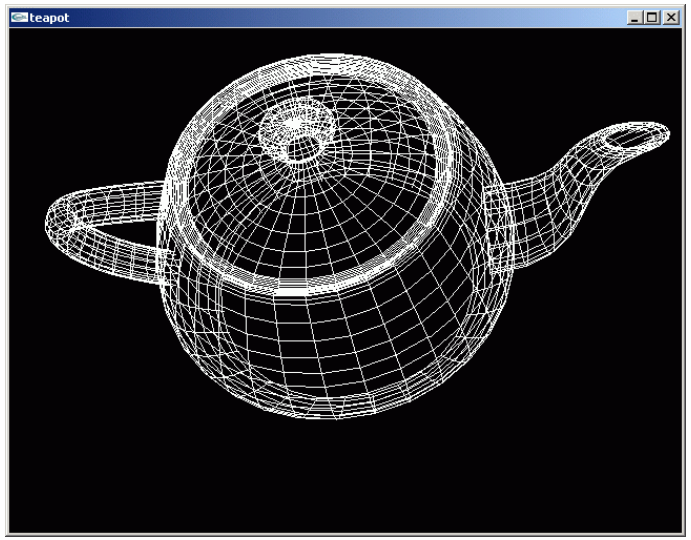
Building complex surfaces by putting together Bezier patches or B-spline patches:

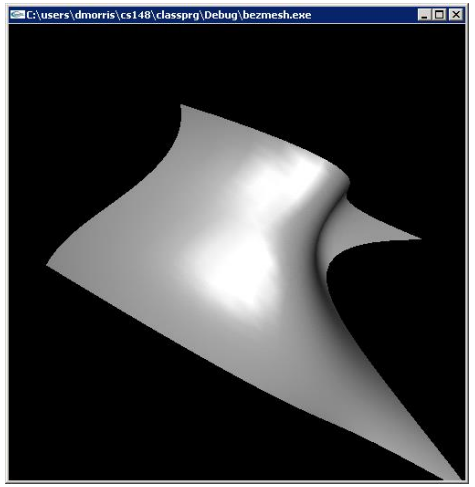


What can you do with surfaces?

You can model correctly
all kinds of shapes

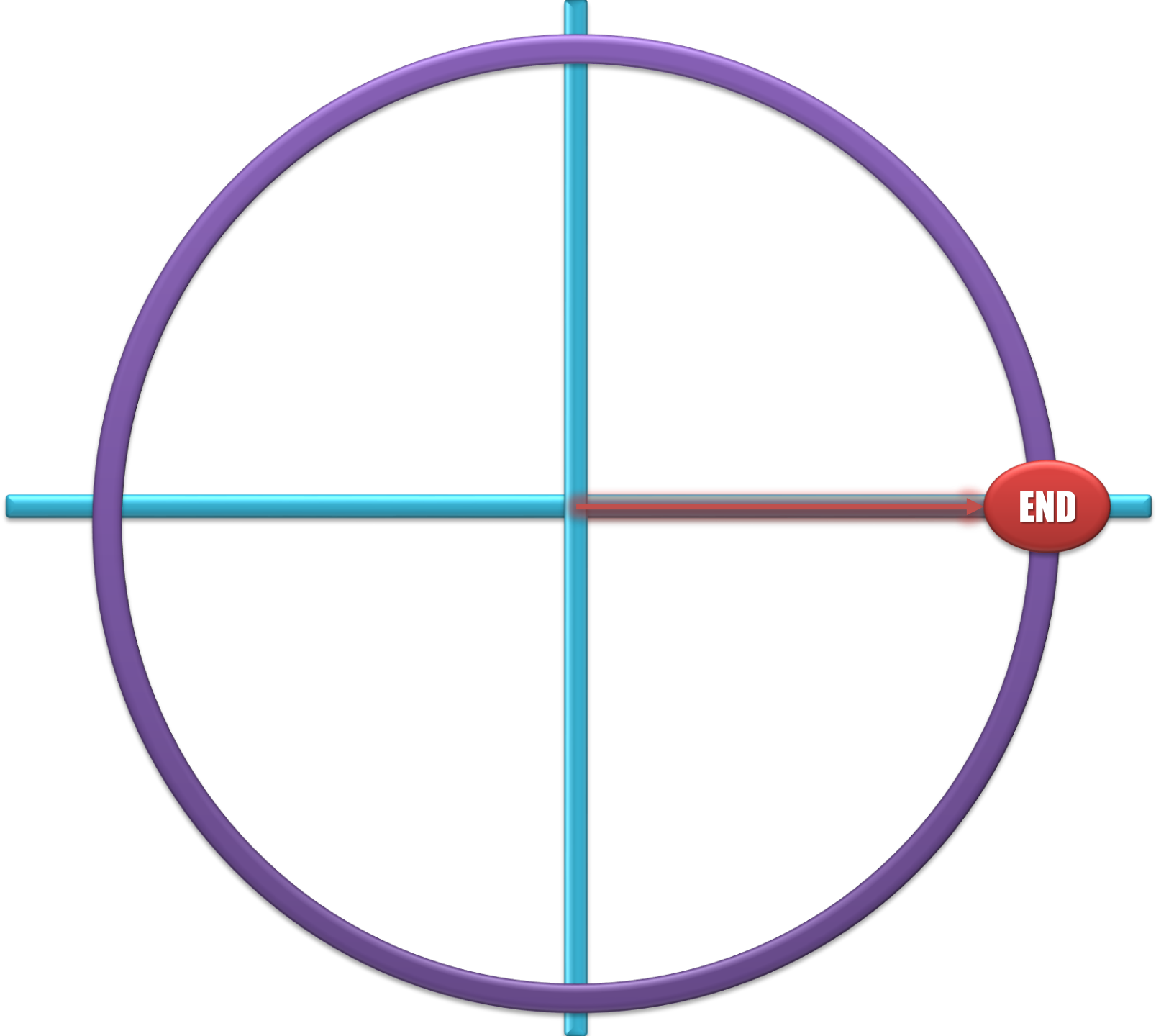


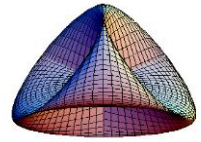




What can you do with surfaces?

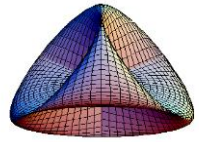
You can build cars!


END



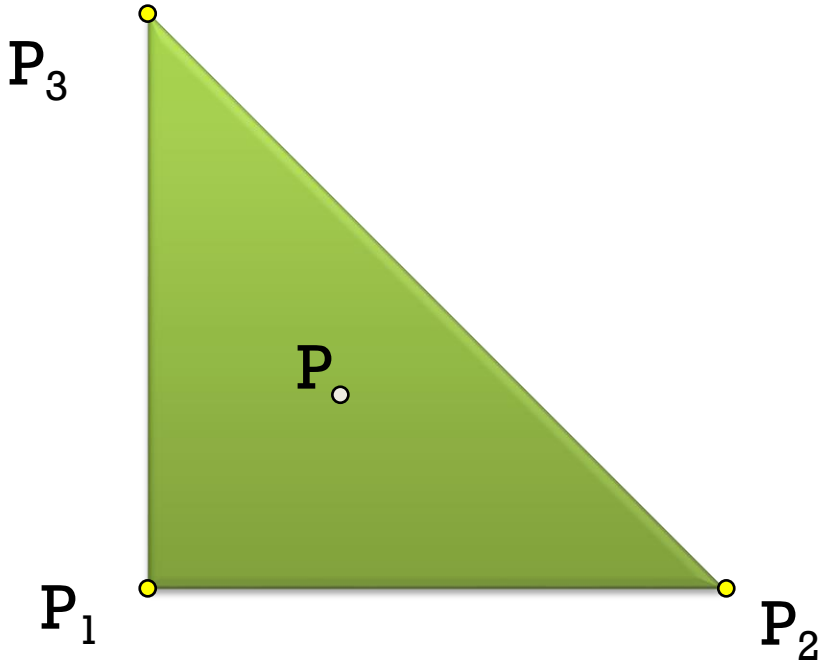
Objectives

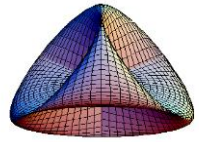
1. Convex Hulls
2. Barycentric Coordinates
3. Weighted Poins



Problem

How to express the interior point of a polygon in terms of its vertices?



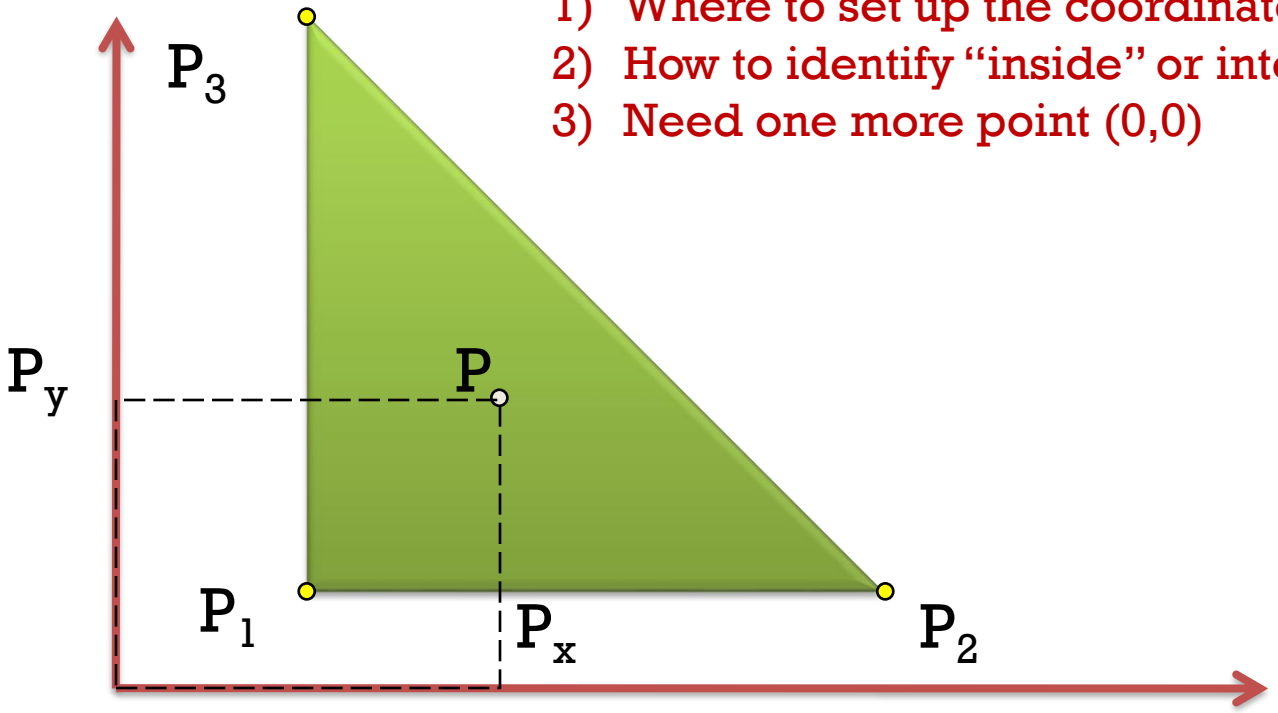


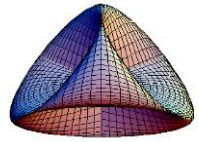
Solution 1

Use a coordinate system

Problems:

1) Where to set up the coordinate system?
2) How to identify "inside" or internal points?
3) Need one more point (0,0)



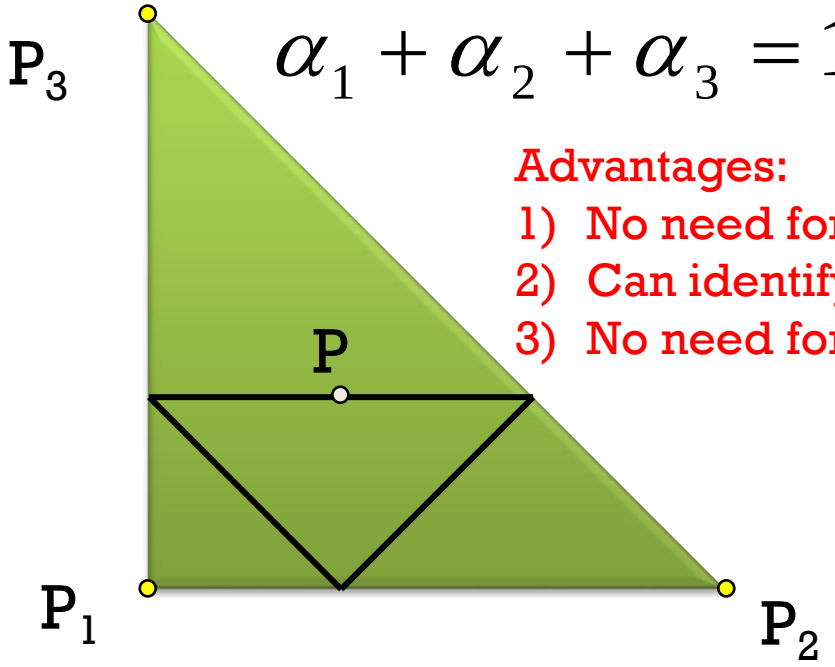


Solution 2

Barycentric coordinate system

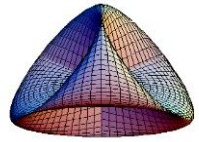
$$P = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3$$

$$\alpha_1 + \alpha_2 + \alpha_3 = 1$$



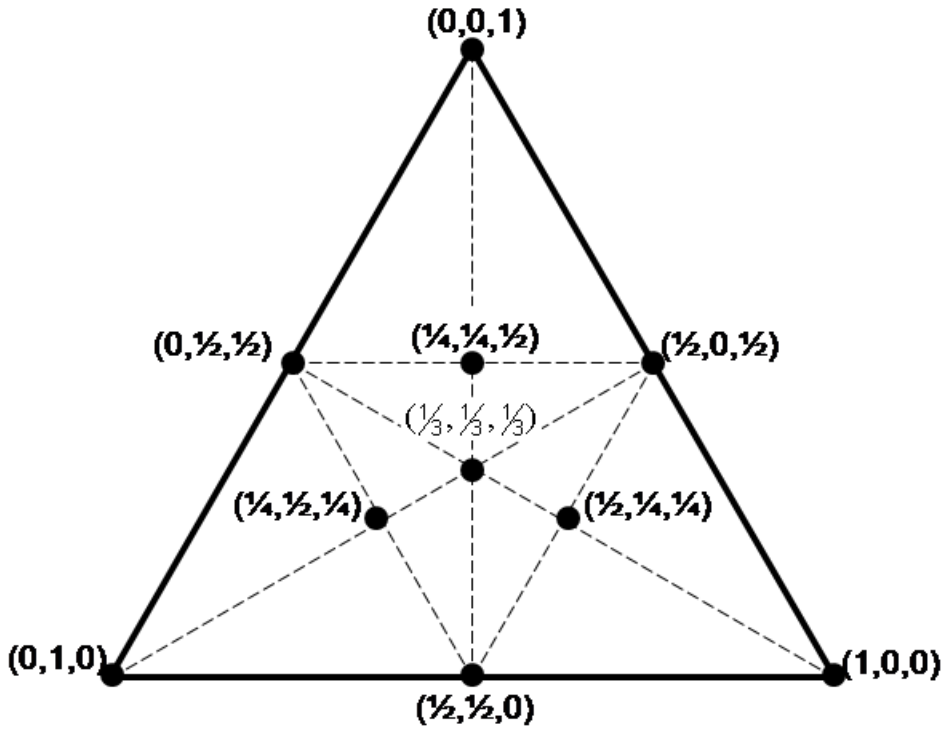
Advantages:

1) No need for a coordinate system
2) Can identify "inside" or internal points
3) No need for one more point (0,0)

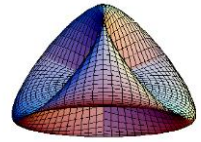


Barycentric Coord.

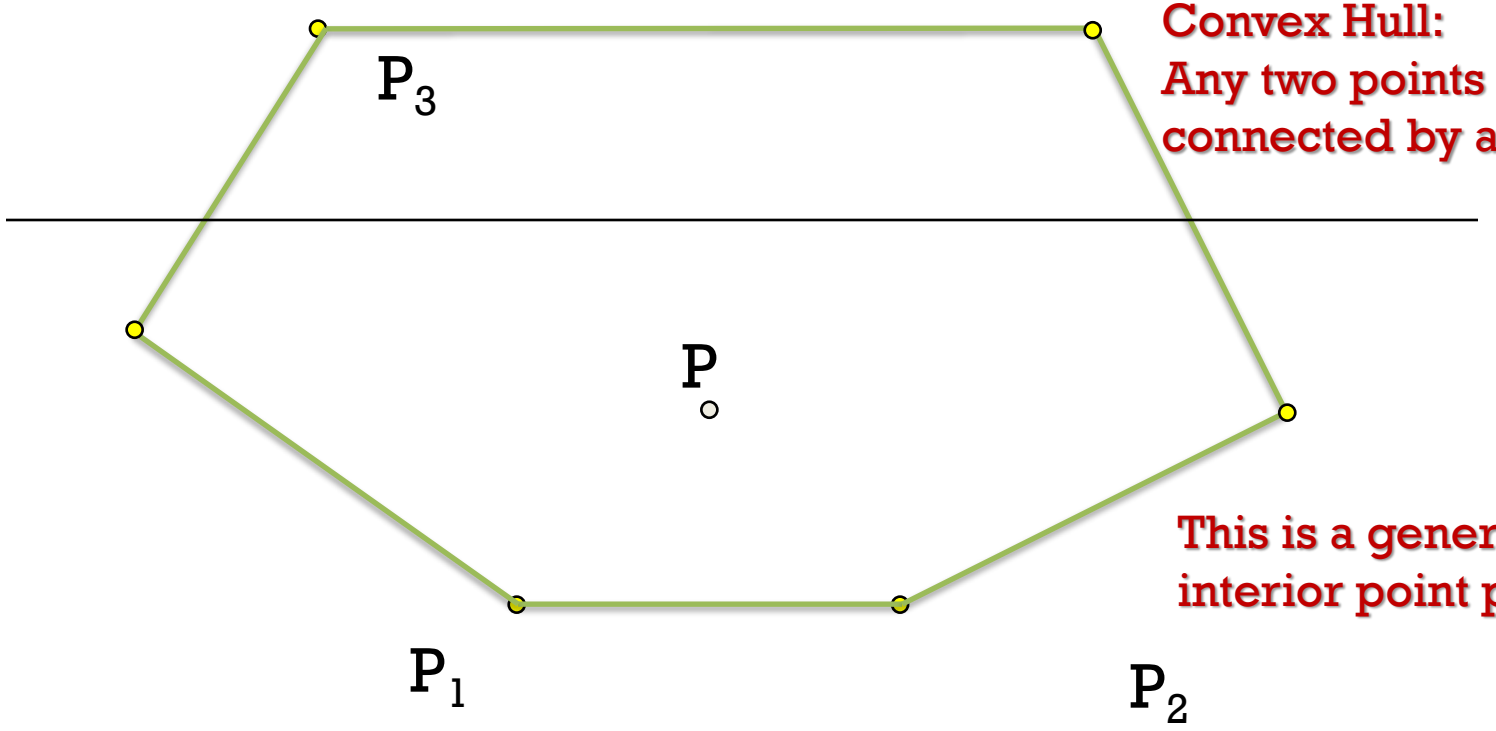
Another way:
Barycentric Coordinates



Convex Hull

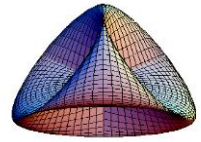


$$P = \sum_{i=1}^{n} \alpha_i P_i \qquad \text{such that} \qquad \sum_{i=1}^{n} \alpha_i = 1$$



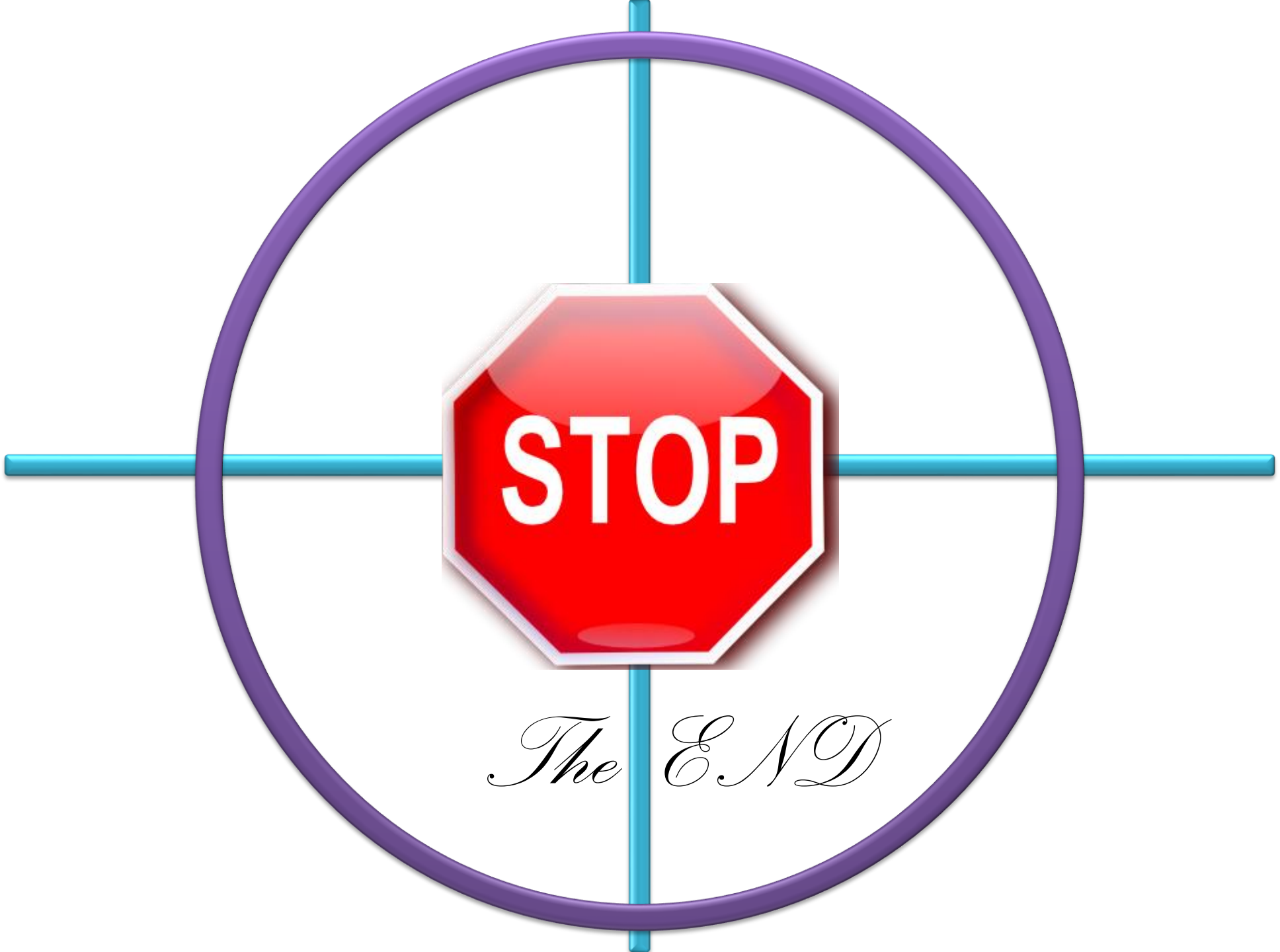
Convex Hull:
Any two points inside can be connected by a straight line.

This is a generalization of the interior point problem



We learnt…

1. Mesh Modeling
2. Building Objects
3. Curves and Control Points
4. Smooth Surfaces



The END