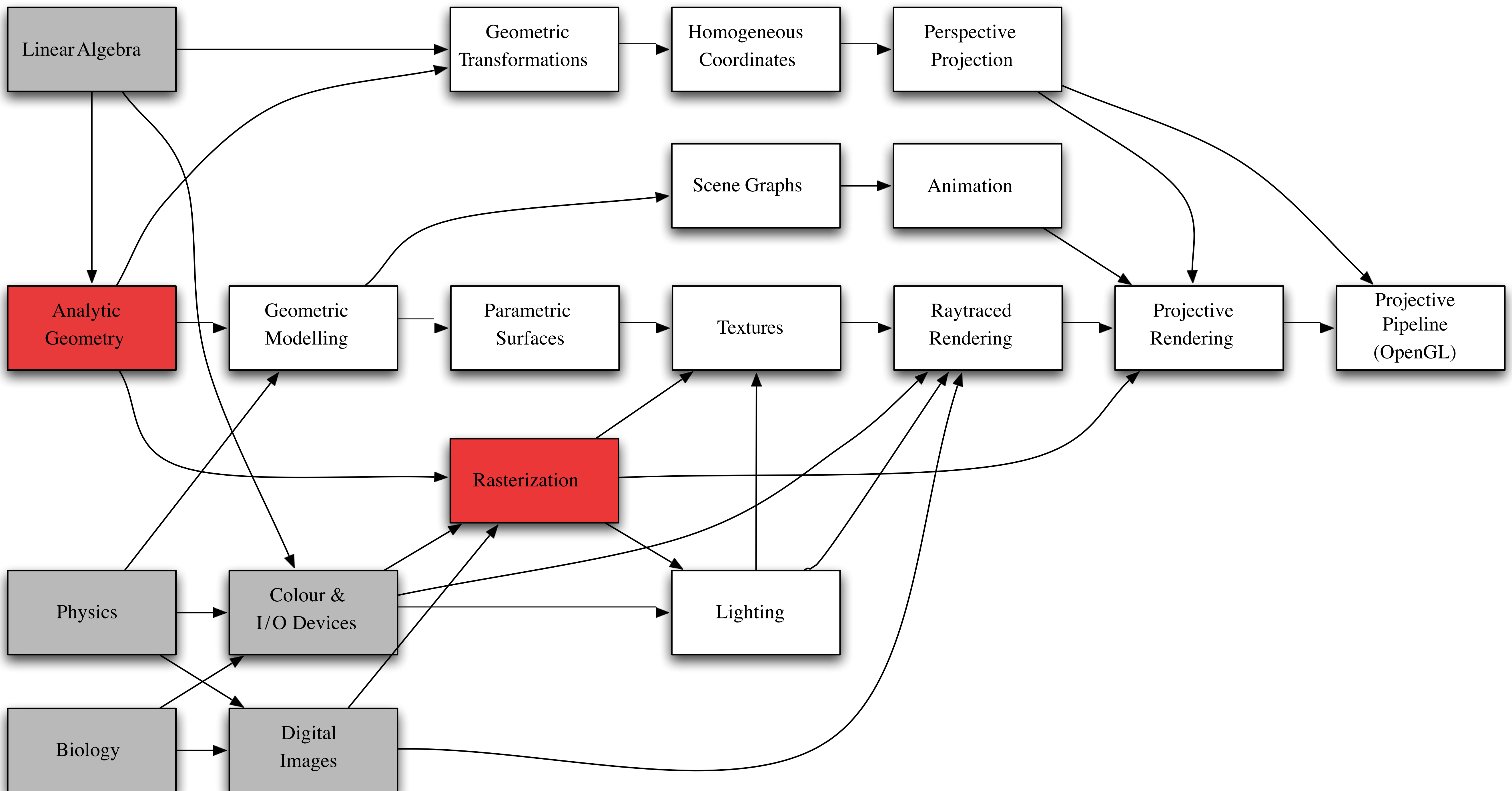
Curves & Circles

COMP 3011J

Where we Are


Linear Algebra
Geometric Transformations
Homogeneous Coordinates
Perspective Projection
Scene Graphs
Animation
Analytic Geometry
Geometric Modelling
Parametric Surfaces
Textures
Raytraced Rendering
Projective Rendering
Projective Pipeline (OpenGL)
Rasterization
Physics
Colour & I/O Devices
Lighting
Biology
Digital Images

Observations

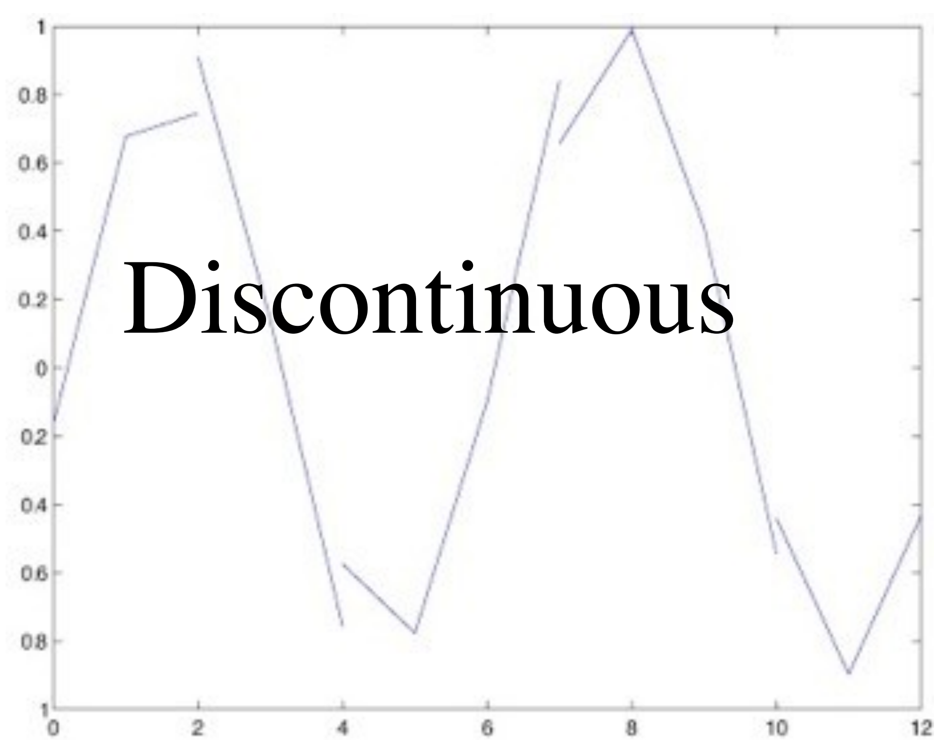
- Nature doesn't use straight lines (much)
  - let alone triangles
- Humans often use curves as well
  - how do we *represent* them?
  - how do we *rasterize* them?
- What is the difference between curves and lines?

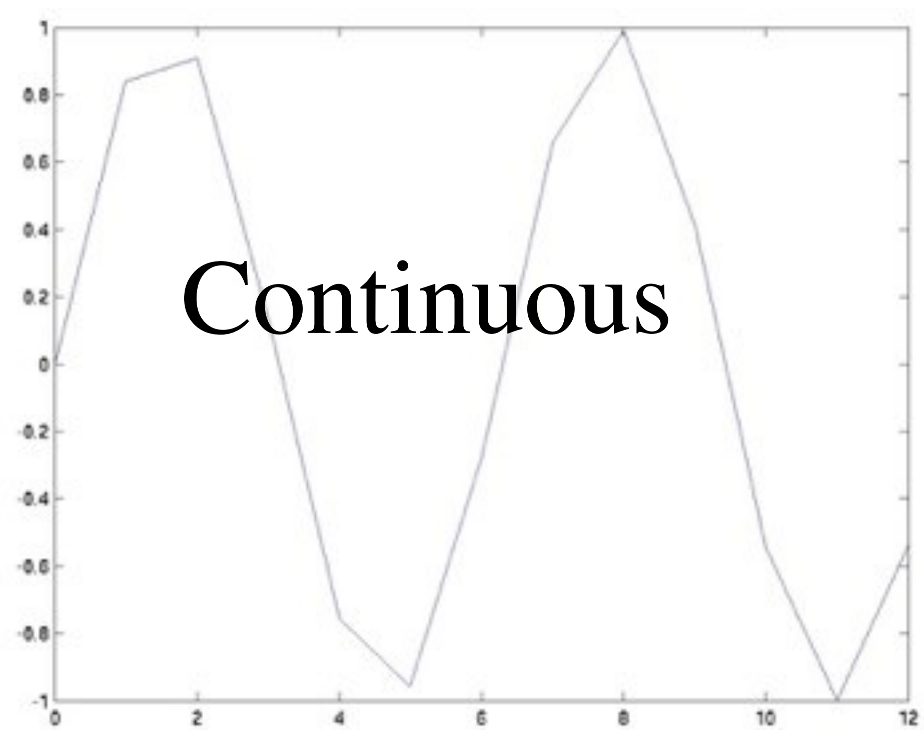
Continuity

- A *continuous function f(x)* satisfies:

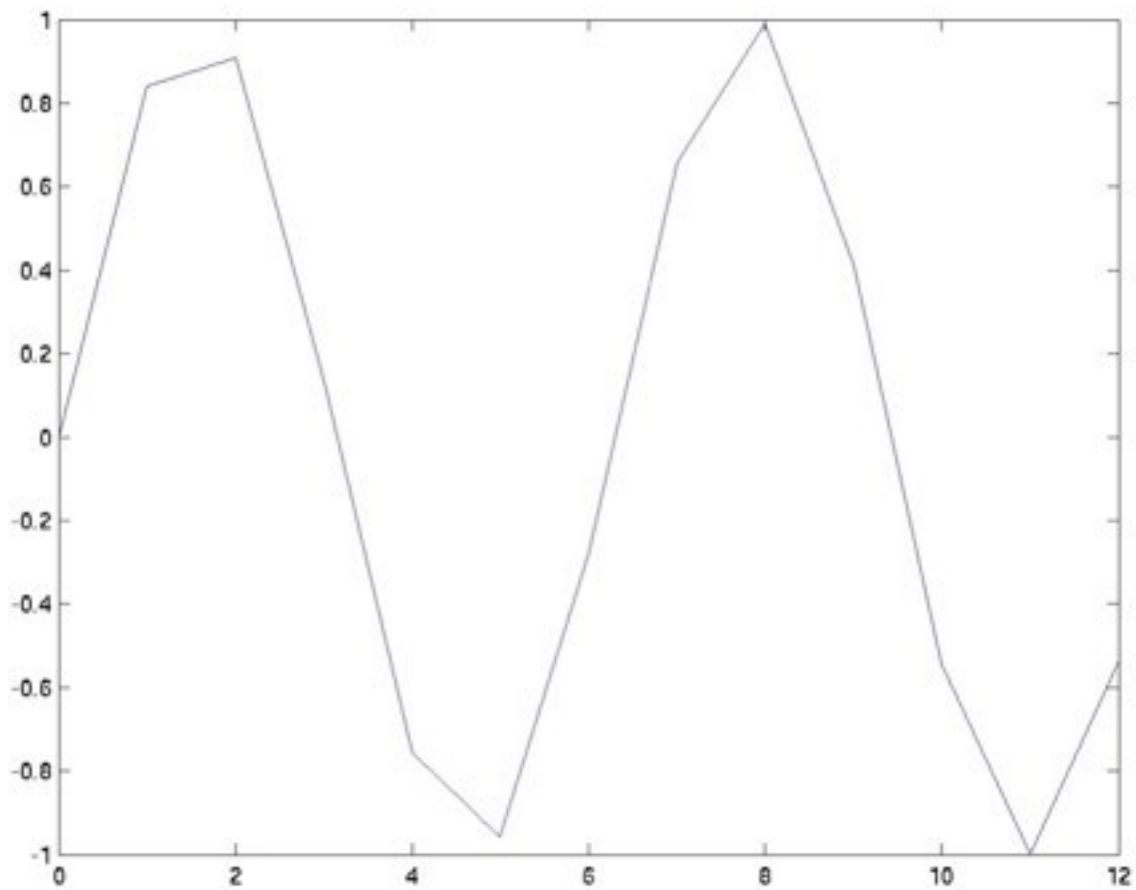
$$\lim_{x \to a^-} f(x) = f(a) = \lim_{x \to a^+} f(x)$$

- Also called C^0 continuous



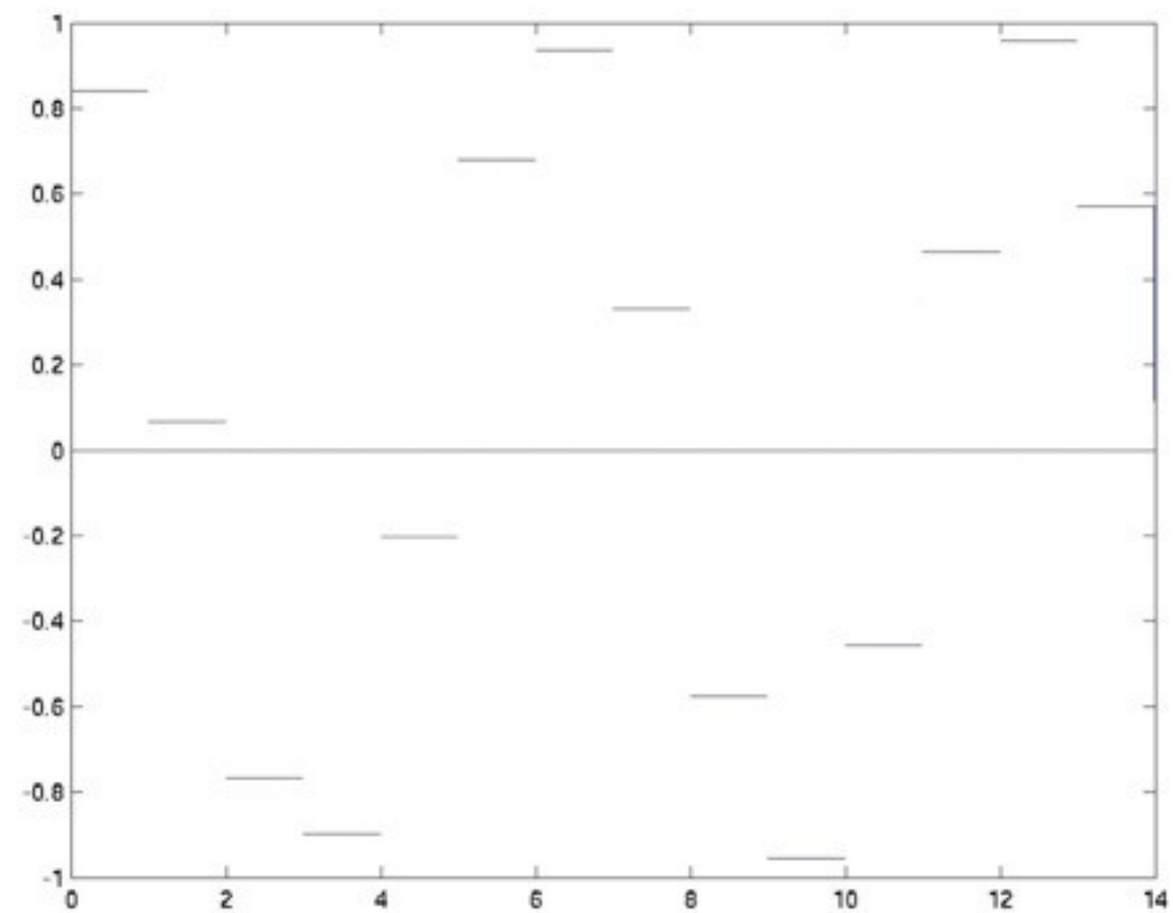


Continuous ≠ Smooth



Not *smooth*

Why not?



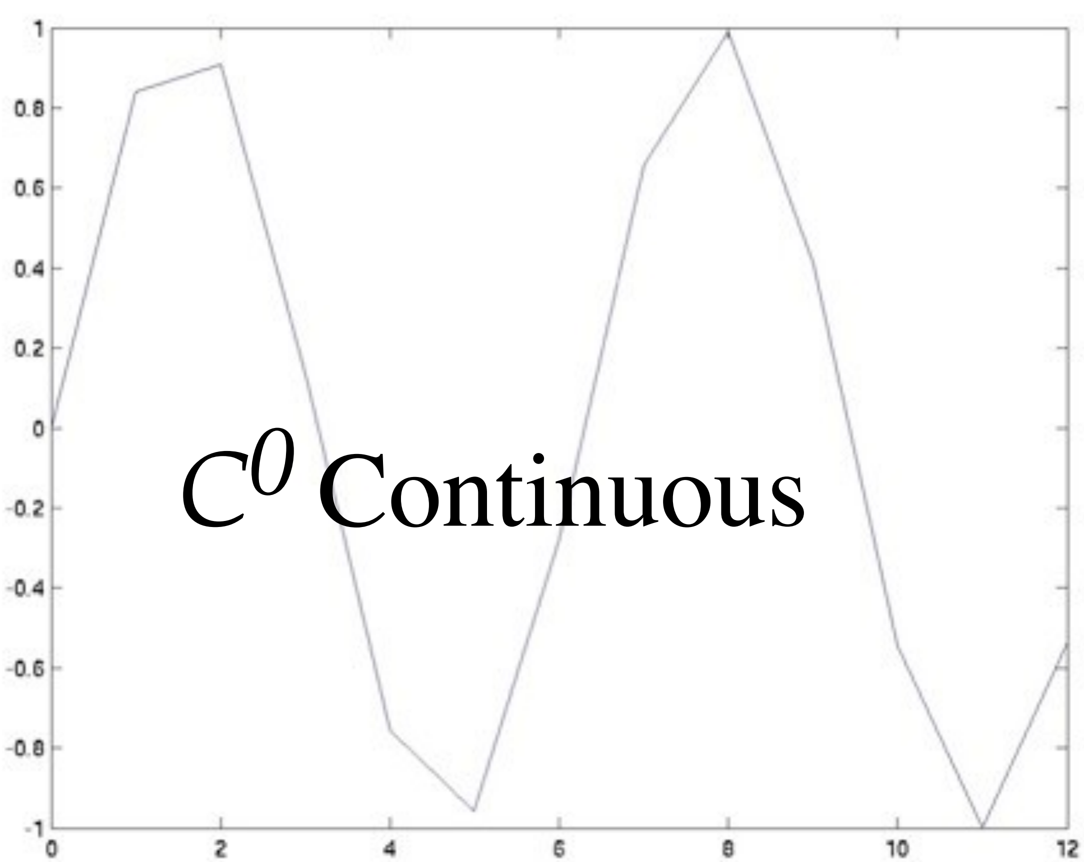
Slope (derivative)

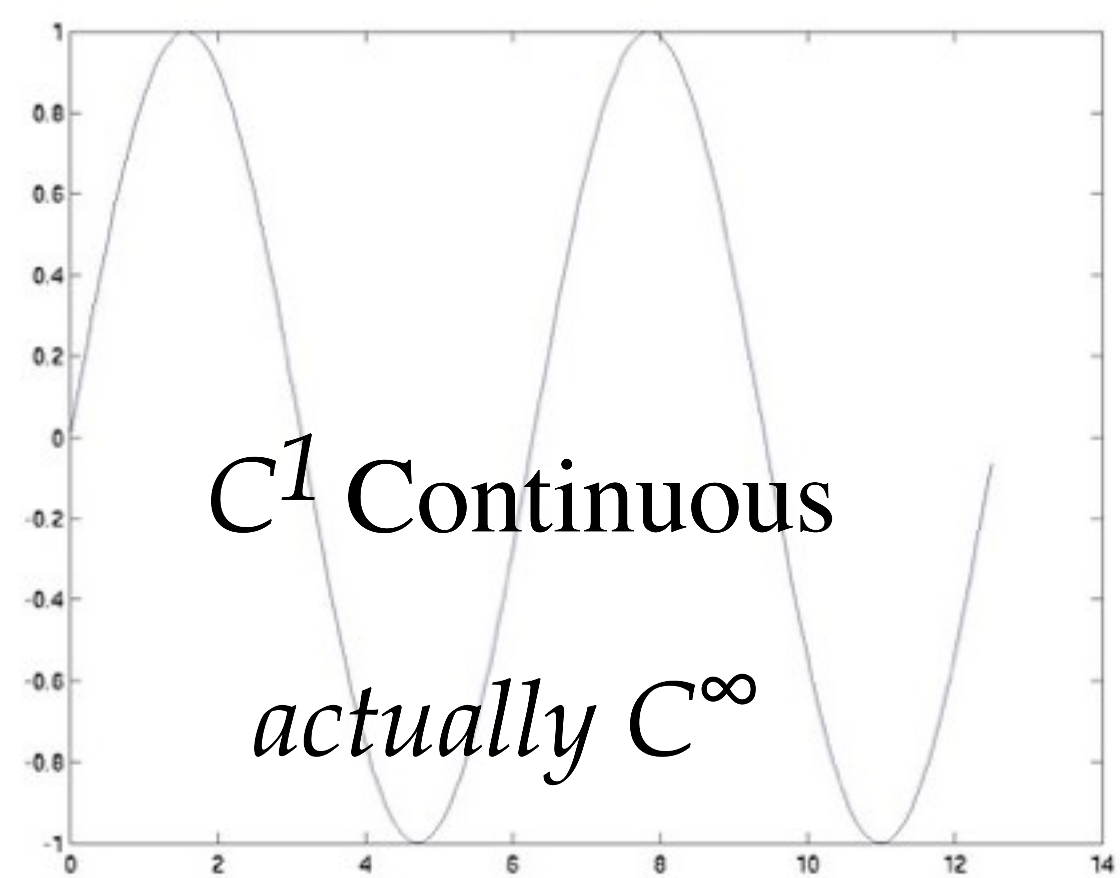
slope is discontinuous

C^n Continuity

- A function *f(x)* is C^n *continuous* if:

$$\lim_{x \to a^-} f^{(n)}(x) = f^{(n)}(a) = \lim_{x \to a^+} f^{(n)}(x)$$



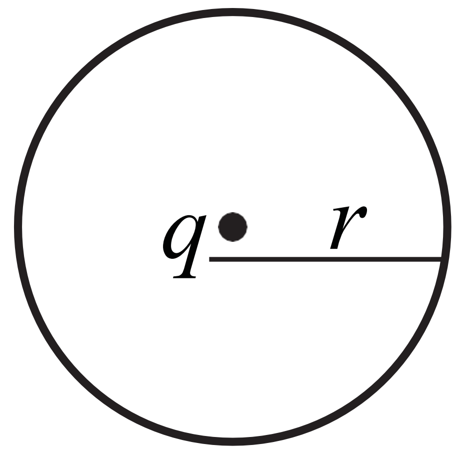


Smoothness

- Smoothness is C^1 continuity
- i.e. continuous derivatives
- But we'll start with something simple
  - a circle

A Circle

- Set of points at distance *r* from point *q*



$$
\begin{aligned}
Circle(q,r) &= \{p=(x,y) : dist(p,q)=r\} \\
&= \left\{p=(x,y) : \sqrt{(x-q_x)^2+(y-q_y)^2}=r\right\} \\
&= \left\{p=(x,y) : (x-q_x)^2+(y-q_y)^2=r^2\right\} \\
&= \left\{p=(x,y) : (p-q)\bullet(p-q)=r^2\right\}
\end{aligned}
$$

Explicit Form

Implicit form:

$$(x-q_x)^2+(y-q_y)^2=r^2$$

Explicit form:

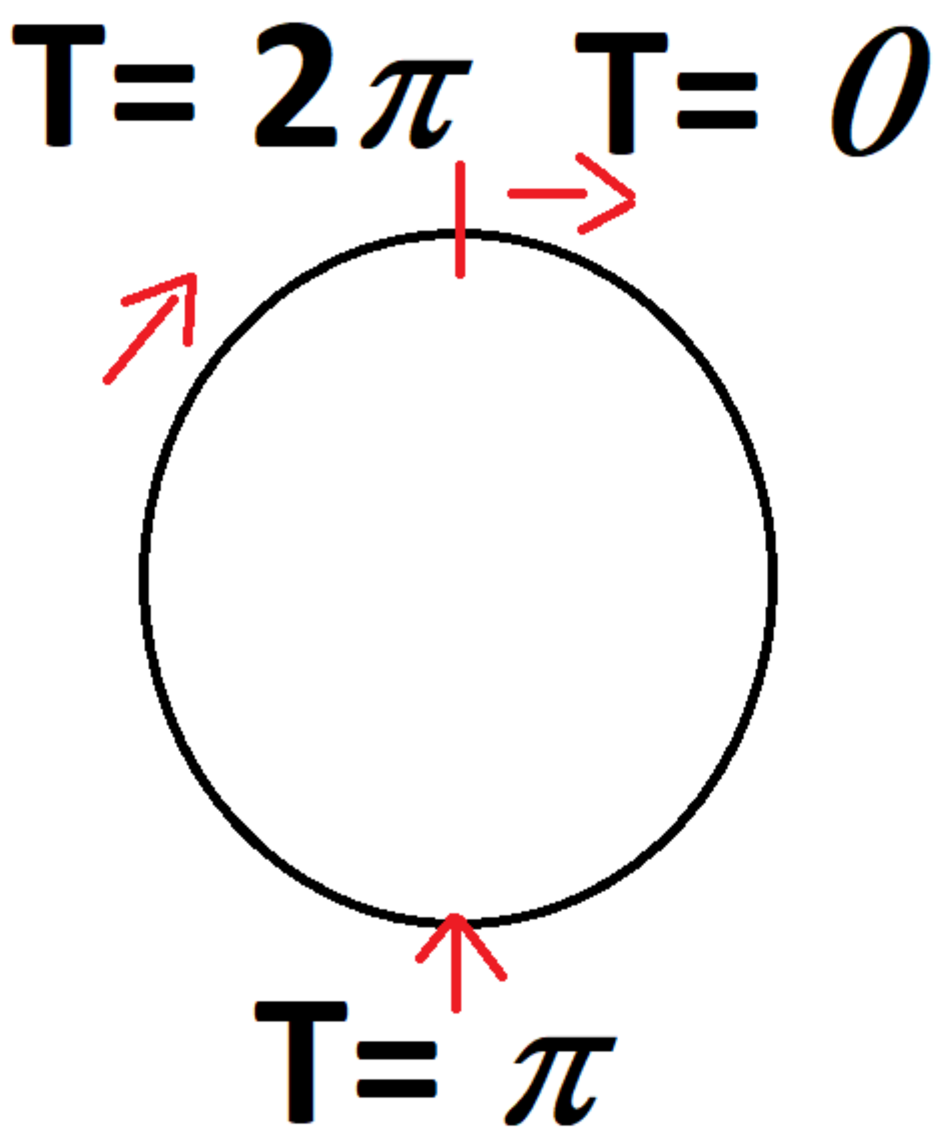
$$(y-q_y)^2=r^2-(x-q_x)^2$$

$$y-q_y=\sqrt{r^2-(x-q_x)^2}$$

$$y=q_y+\sqrt{r^2-(x-q_x)^2}$$

Parametric Circle

$$Circle(q,r) = \left\{ \left(q_x + r\sin t, q_y + r\cos t \right) : 0 \le t \le 2\pi \right\}$$



Rasterization

- Explicit Form:

```
for (dx = -r; dx <= r; dx++)
  {
  p.x =  q.x + dx;
  p.y =  q.y + sqrt(r*r-dx*dx);
  setPixel(p.x,p.y);
  p.y = q.y - sqrt(r*r-dx*dx);
  setPixel(p.x,p.y);
  }
```

Implicit Rasterization

- Convenient, but inefficient:
- Checks all pixels' distance from q
- Sets them if distance < 0.5

```
for (dx = -r; dx <= r; dx++)
  for (dy = -r; dy <= r; dy++)
    {
    dVec = Vector(dx, dy);
    dvLength = dVec.Length();
    if ((dvLength > r - 0.5) && (dvLength < r + 0.5)
      {
      p = q + dVec;
      setPixel(p.x,p.y);
      }
    }
```

Parametric Form

- Simple (as usual)

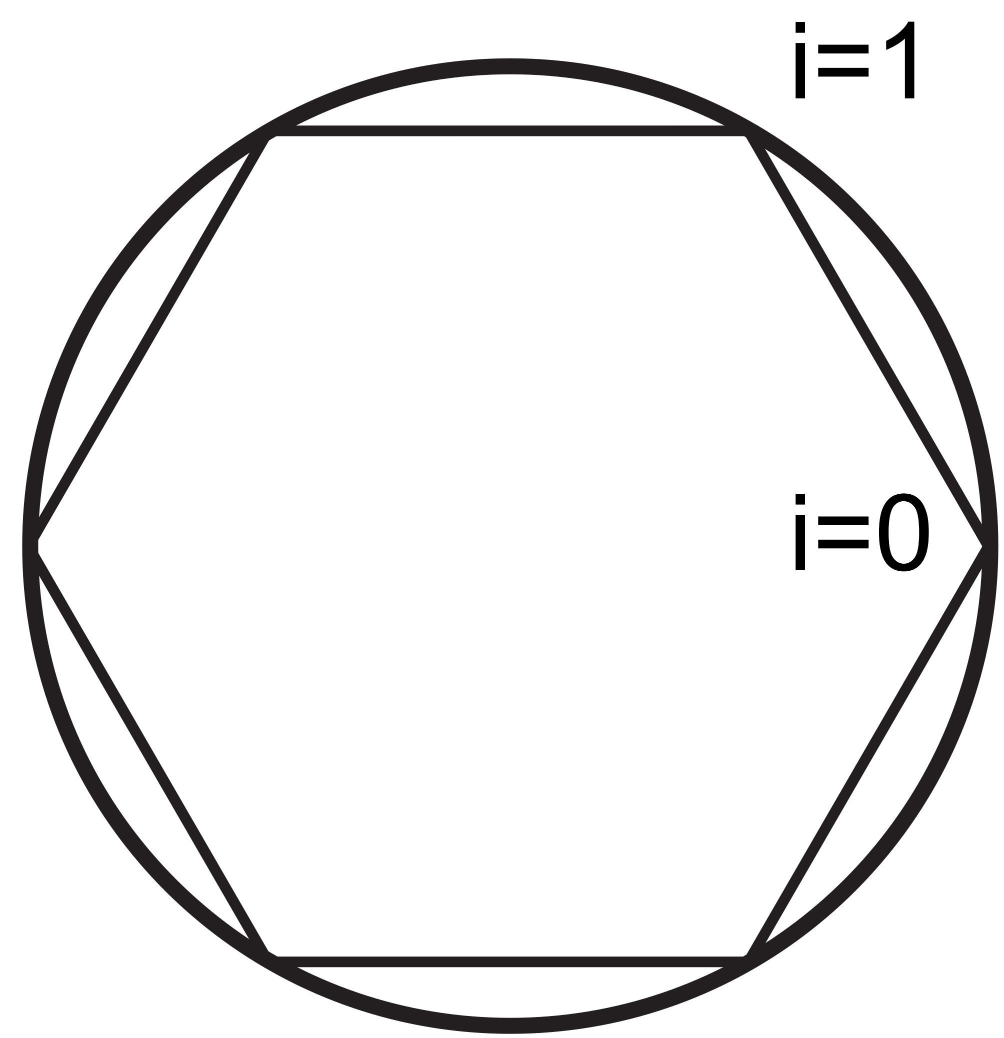
```
for (t = 0.0; t <= 2.0*PI; t+=0.01)
  {
  p = q + r*Vector(sin(t), cos(t));
  setPixel(p.x,p.y);
  }
```

- But slow = sin & cos are *expensive*
- But we can speed this up
  - by treating circle as a set of *lines*

Line Approximation

```
for (i = 0; i < nLines; i++)
  {
  t1 = 2.0 * PI * i / nLines;
  t2 = 2.0 * PI * (i+1) / nLines;

  p1 = q + Vector(r*sin(t1), r*cos(t1));
  p2 = q + Vector(r*sin(t2), r*cos(t2));
  drawLine(p1,p2);
  }
```



Observations

- Parametric form is always easy
- and it handles complex shapes
- circles, other types of curves
  - but it can be expensive
- Approximation with lines is cheaper

Filling Circles

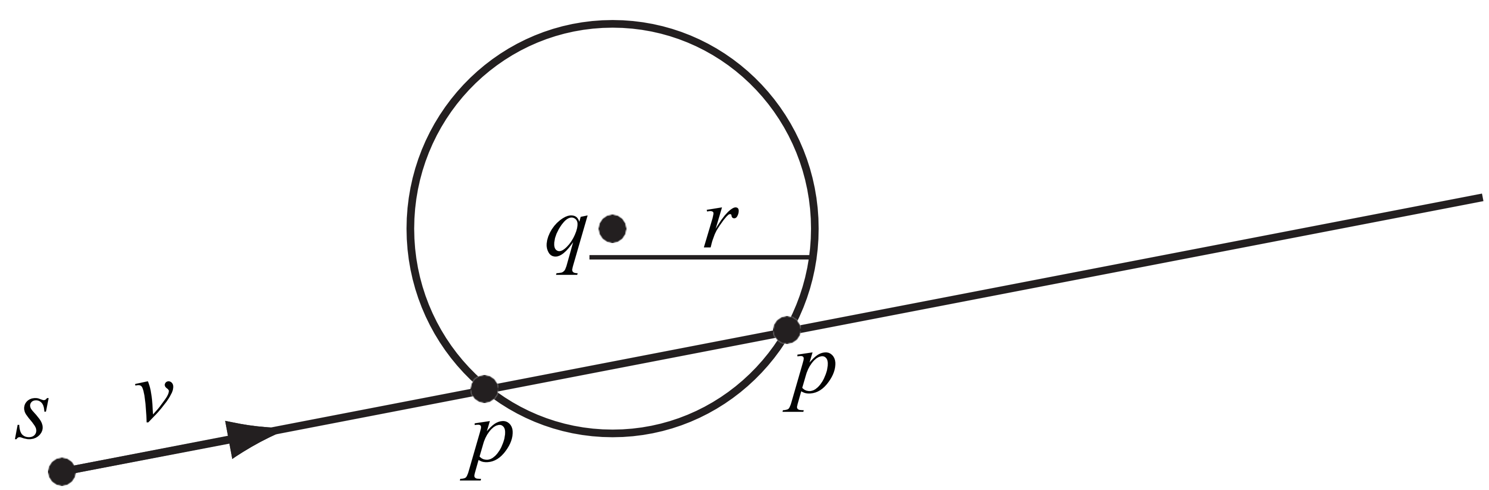
- Explicit: Raster Scan still works
- Implicit: Use <= r, not == r
- Parametric: use r as second parameter
- Lines: draw triangle *(p1, p2, q)*

Lines & Circles

- We can intersect two lines
- What about two circles?
  - We won't need to do this
- Or a line and a circle?
  - We will need to do this

Line-Circle Intersection

- Given a circle $Circle(q,r)$
- And a line $l = s + vt$
- Find point p at intersection
  - i.e. find t



Step 1

We know that:

$$p = s + \vec{v}t$$

and that:

$$(p - q) \bullet (p - q) = r^2$$

So we plug one into the other and get:

$$(s + \vec{v}t - q) \bullet (s + \vec{v}t - q) = r^2$$

We will simplify this by letting:

$$\vec{u} = s - q$$

And we get:

$$(\vec{u} + \vec{v}t) \bullet (\vec{u} + \vec{v}t) = r^2$$

Step 2

$$(\vec{u} + \vec{v}t)\bullet(\vec{u} + \vec{v}t) = r^2$$

$$\vec{u}\bullet\vec{u} + 2\vec{u}\bullet\vec{v}t + \vec{v}\bullet\vec{v}t^2 = r^2$$

$$(\vec{v}\bullet\vec{v})t^2 + (2\vec{u}\bullet\vec{v})t + (\vec{u}\bullet\vec{u} - r^2) = 0$$

But this is a quadratic equation, so we solve:

$$A = \vec{v}\bullet\vec{v}$$

$$B = 2\vec{u}\bullet\vec{v}$$

$$C = \vec{u}\bullet\vec{u} - r^2$$

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Code

```
bool Intersect(Line l, Circle C, Point &p)
  { // passes closest intersection back in p
  // l.point is the point that the line starts from (i.e. s)
  Vector v = l.vector;
  Vector u = l.point - C.centre;
  float A = v.Dot(v);
  float B = 2*u.Dot(v);
  float C = u.Dot(u) - C.radius*C.radius;
  float discriminant = B*B - 4*A*C;
  // can't take square root of -ve numbers: i.e. no point p
  if (discriminant < 0) return false;
  float t1 = (-B - sqrt(discriminant))/2*A;
  float t2 = (-B + sqrt(discriminant))/2*A;
  // now take closest +ve result (-ve is *behind* point s)
 if (t1 > 0) {
    p = l.point + v*t1
    return      } ;
    true;
 if (t2 > 0) {
    p = l.point + v*t2
                ;
  else return false;
    return      }
  } // end of Intersect()
    true;
```

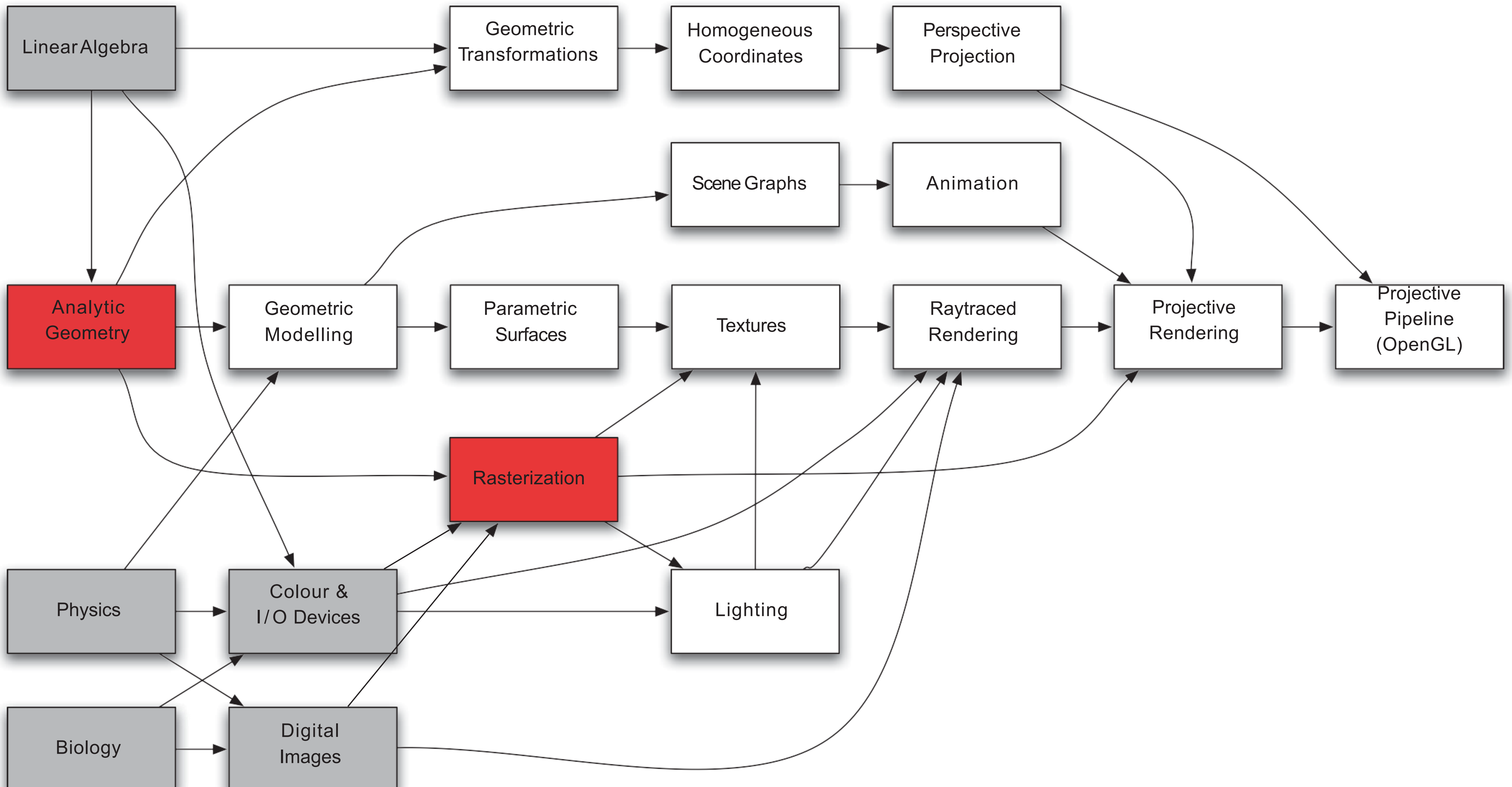
Other Curves

- We *could* do
  - ellipses
  - parabolae
  - hyperbolae
- But we want something more general

Hermite & Bézier Curves

COMP 3011J

Where we Are



Linear Algebra
Geometric Transformations
Homogeneous Coordinates
Perspective Projection
Scene Graphs
Animation
Analytic Geometry
Geometric Modelling
Parametric Surfaces
Textures
Raytraced Rendering
Projective Rendering
Projective Pipeline (OpenGL)
Rasterization
Physics
Colour & I/O Devices
Lighting
Biology
Digital Images

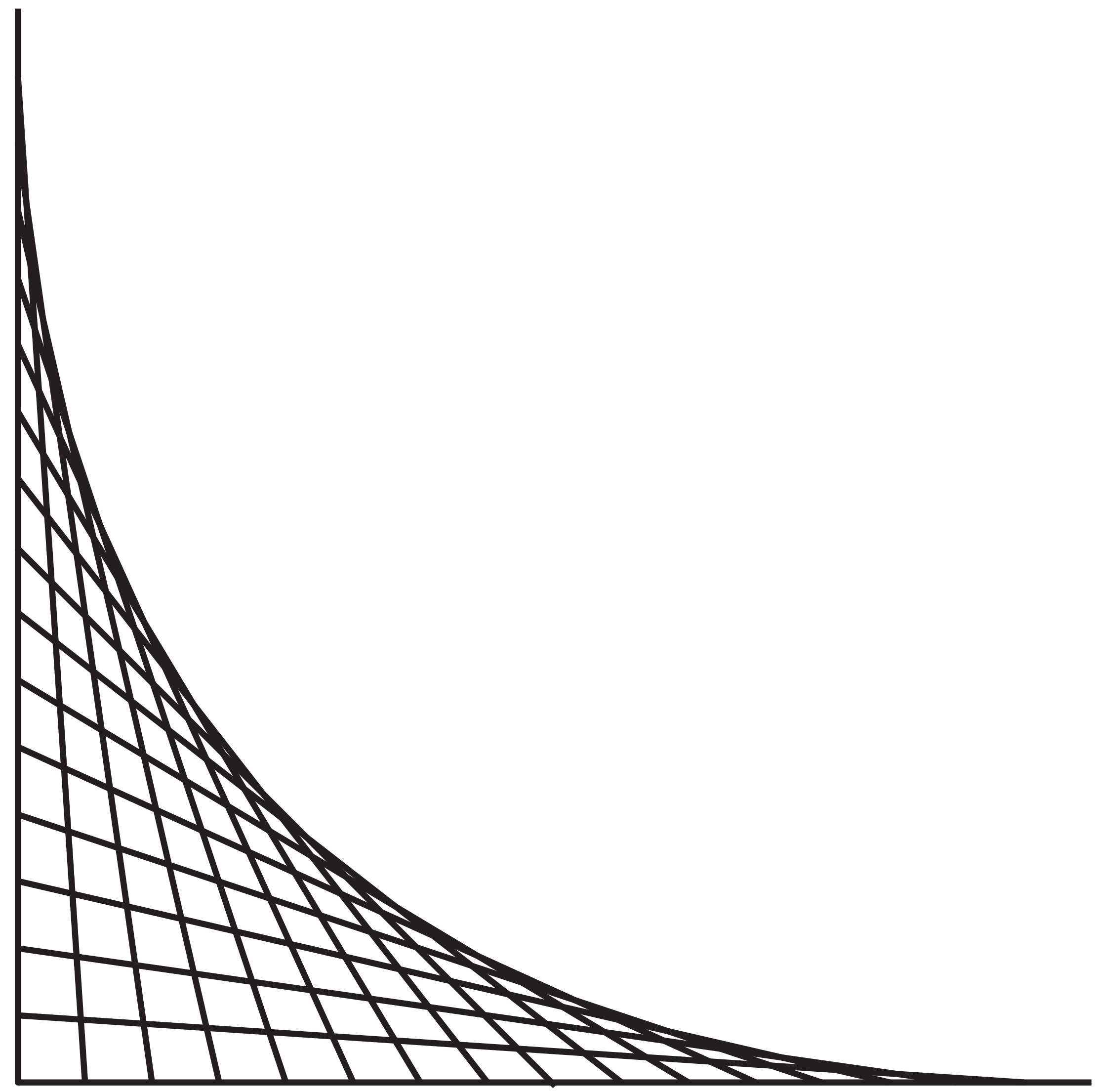
Continuity

- We want *smooth* curves (& surfaces)
- I.e. we need *C^1 continuity*
  - and we want to build them from *lines*
  - repeated linear interpolation

String Art

from www.stringart.eu

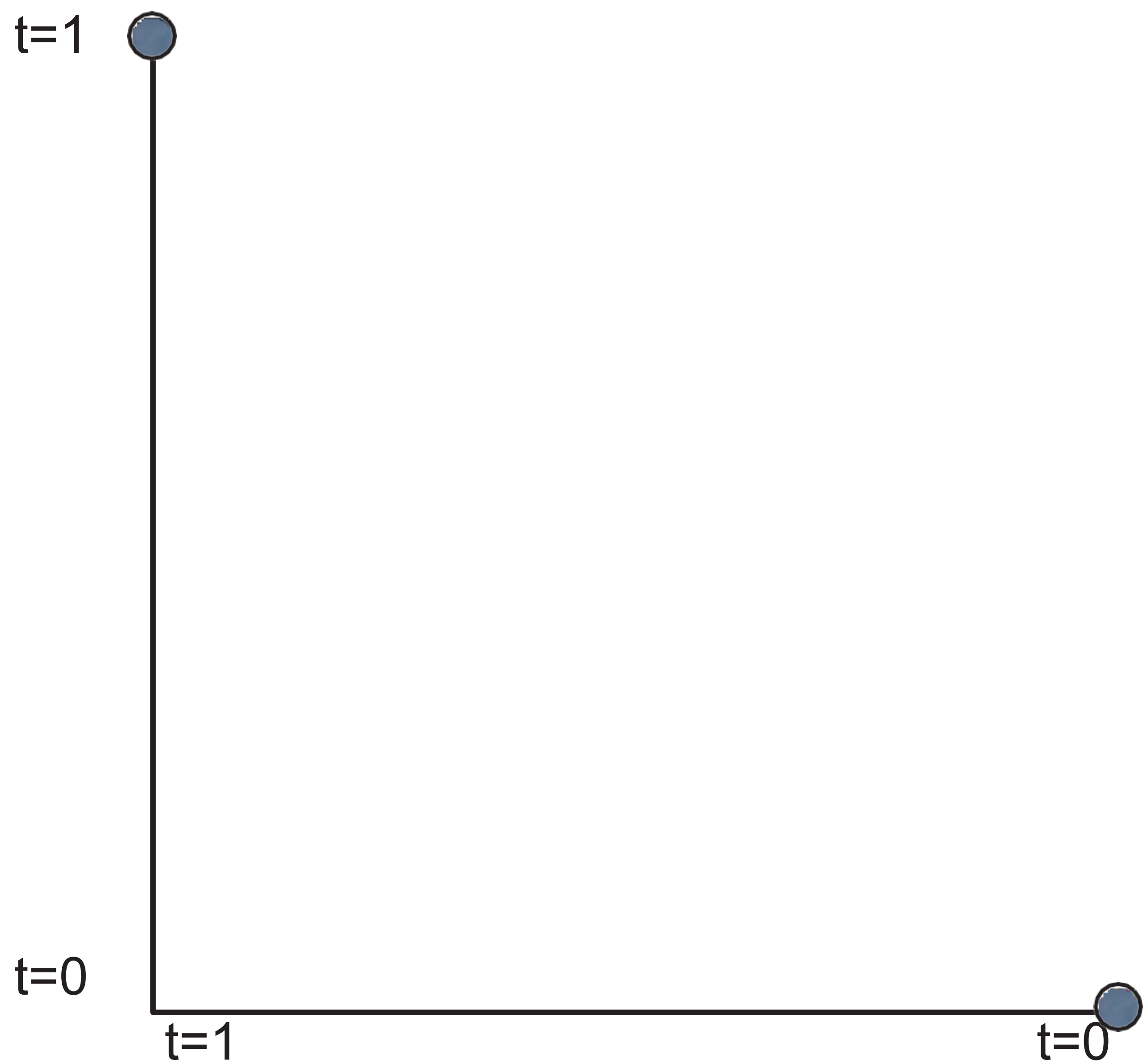
Curves from Lines



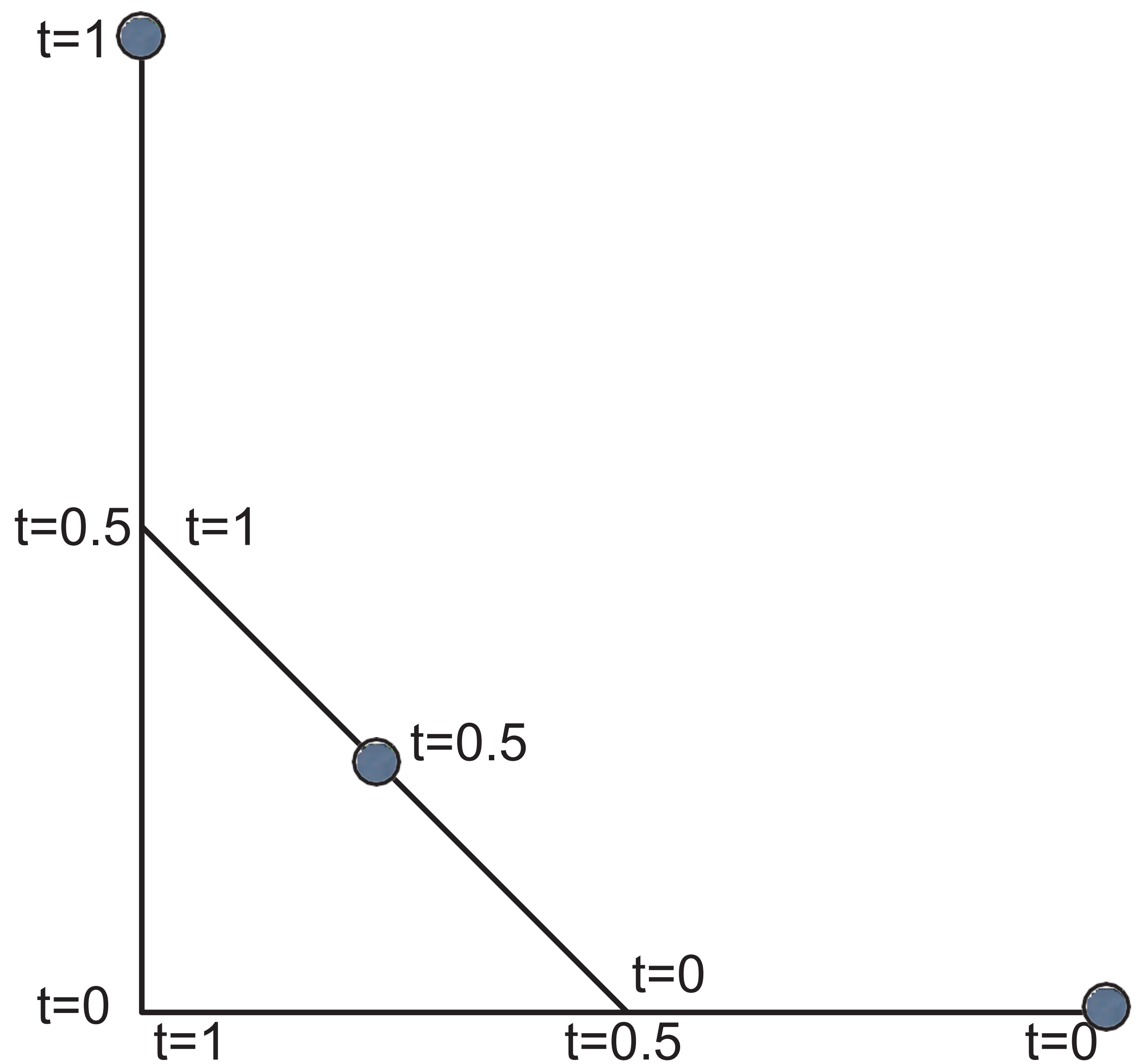
Properties

- All we need is linear interpolation
- Curve is *contained* by original points
- Curve *built* up of small segments
  - in the limit, of individual points
- But lines underneath are not needed
- And we want to parameterize it in *t*

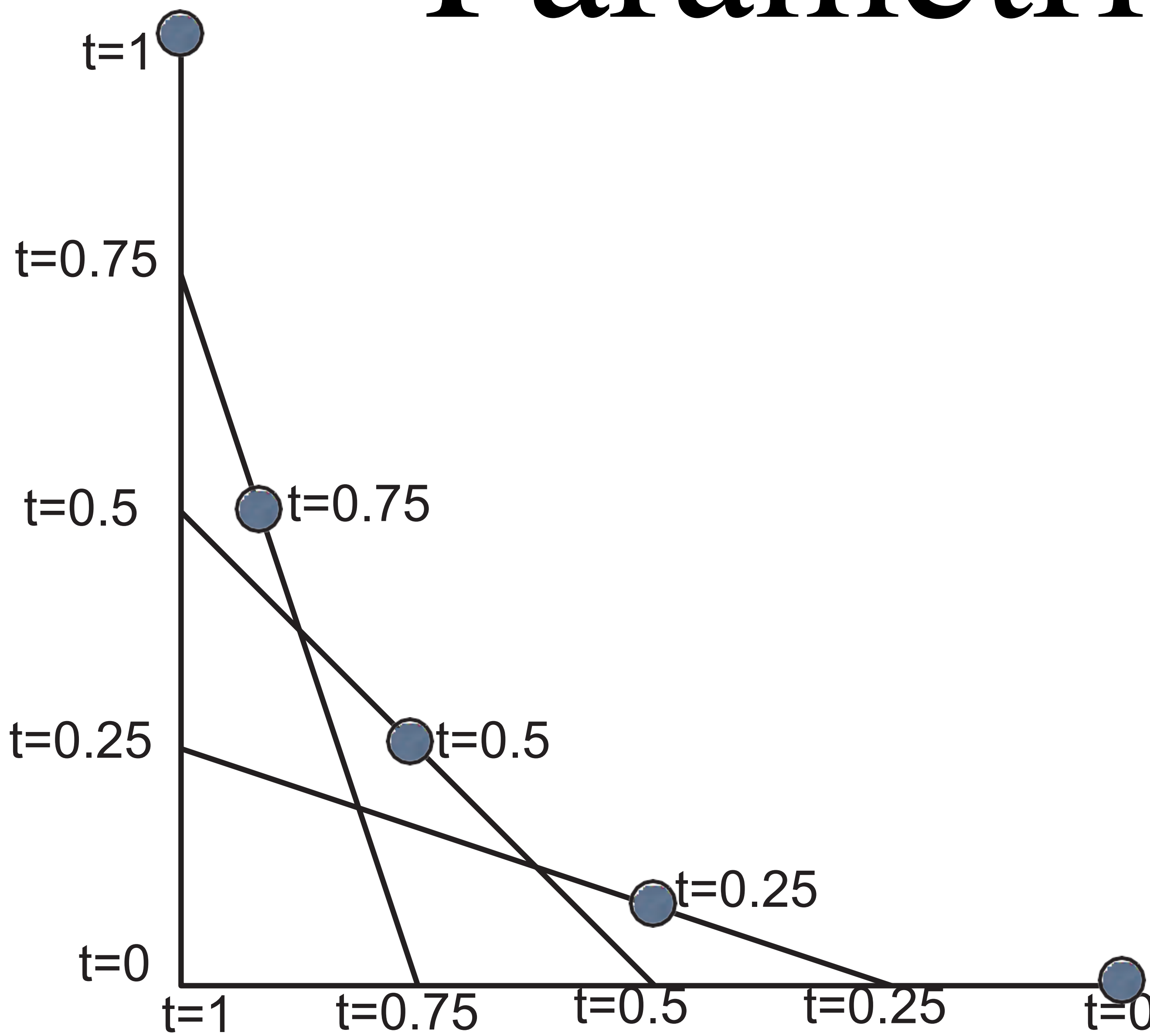
Parametrization



Parametrization



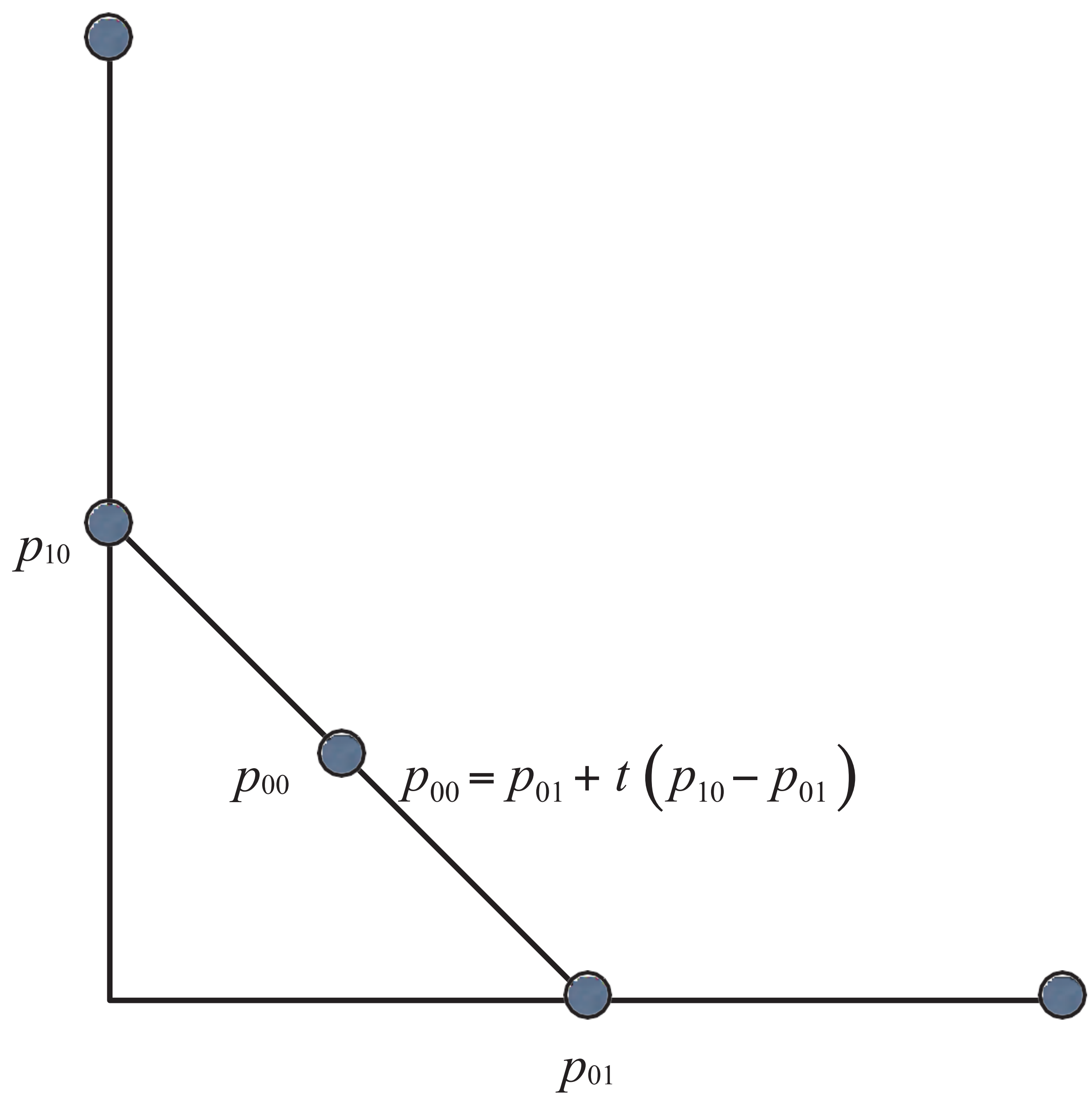
Parametrization



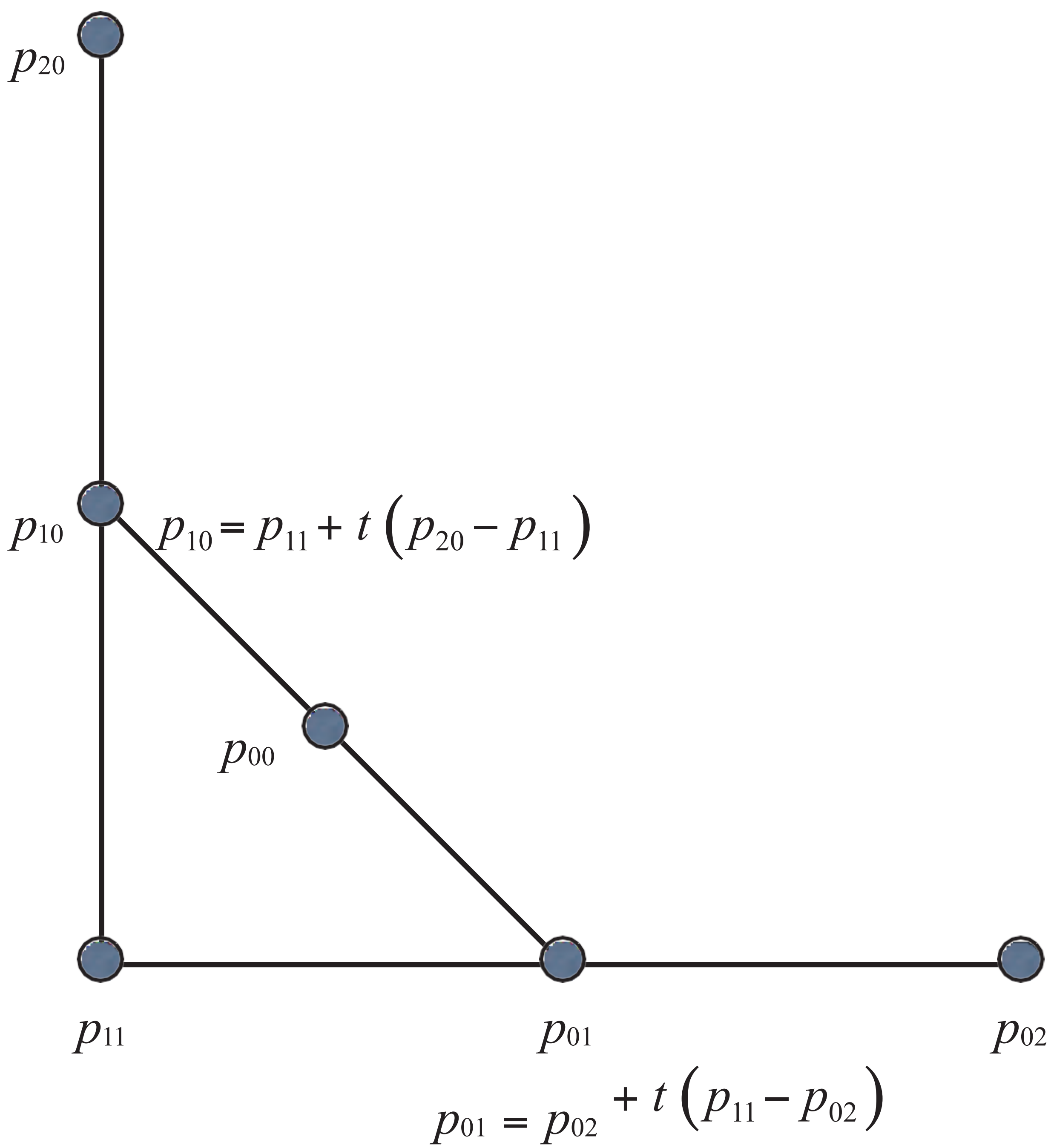
In the limit

- We take *one* point from each line
- For a given *t*
  - Interpolate along original edges
  - Then along the next edge
  - Repeat until we have a single point

Development



Development



Algebra

$$p_{10} = p_{11} + t\left(p_{20} - p_{11}\right) = \left(1-t\right)p_{11} + t\left(p_{20}\right)$$

$$p_{01} = p_{02} + t\left(p_{11} - p_{02}\right) = \left(1-t\right)p_{02} + t\left(p_{11}\right)$$

$$\begin{aligned}
p_{00} &= p_{01} + t\left(p_{10} - p_{01}\right) = \left(1-t\right)p_{01} + t\left(p_{10}\right) \\
&= \left(1-t\right)\left(\left(1-t\right)p_{02} + t\left(p_{11}\right)\right) + t\left(\left(1-t\right)p_{11} + t\left(p_{20}\right)\right) \\
&= p_{02} - 2p_{02}t + p_{02}t^2 + p_{11}t - p_{11}t^2 + p_{11}t - p_{11}t^2 + p_{20}t^2 \\
&= \left(p^{02} - 2p^{11} + p^{20}\right)t^2 \\
&\quad + \left(-2p_{02} + 2p_{11}\right)t \\
&\quad + p_{02} \\
&= \begin{bmatrix} p^{02} & p^{11} & p^{20} \end{bmatrix} \begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^2 \\ t \\ 1 \end{bmatrix}
\end{aligned}$$

Table method

$p_{00} = (1-t)p_{01} + t(p_{10})$ d=0	$\xleftarrow{1-t}$ $p_{01} = (1-t)p_{02} + t(p_{11})$ d=1	$\xleftarrow{1-t}$ p_{02} d=2
$\uparrow t$ $p_{10} = (1-t)p_{11} + t(p_{20})$ d=1	$\uparrow t$ $\xleftarrow{1-t}$ p_{11} d=2	
$\uparrow t$ p_{20} d=2		

In general

- Compute diagonals in *descending* order
- And each entry is found by:

$$p_{ij} = (1-t)p_{i,j+1} + t\left(p_{i+1,j}\right) \text{ where } i+j=d$$

- We stop when we reach p_{00}
- And draw it
- Repeat for different values of t

de Casteljau Algorithm

```
int N_PTS = 3;
Point bezPoints[N_PTS][N_PTS];

void DrawBezier()
  { // DrawBezier()
  for (float t = 0.0; t <= 1.0; t  += 0.01)
    { // parameter loop
    for (int diag = N_PTS-2; diag >= 0; diag--)
      { // diagonal loop
      for (int i = 0; i <= diag; i++)
         { // i loop
         int j = diag - i;
         bezPoints[i][j] = (1.0-t)*bezPoints[i][j+1] + t*bezPoints[i+1][j];
         } // i loop
      } // diagonal loop
      // set the pixel for this parameter value
      SetPixel(bezPoints[0][0];
    } // parameter loop
  } // DrawBezier()
```

Bézier Curves

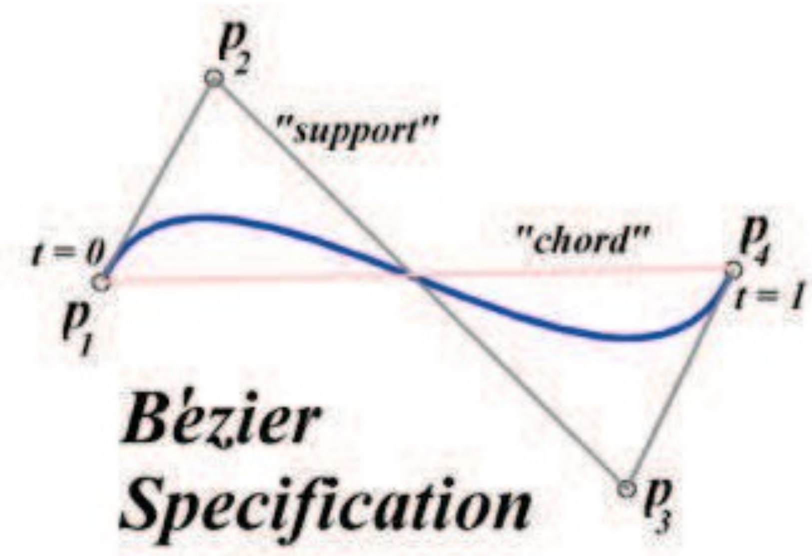
- Oldest form of computed curve
- Invented in automotive industry
- Can be computed for *any* degree
  - 2 points: line
  - 3 points: quadratic
  - 4 points: cubic

Smooth Curves

- For C^1 continuity, choose *slopes* at endpoints
- two *slopes* + two *points* = 4 *constraints*
  - So we need (4 - 1) = 3 degree polynomials
    - i.e. *cubic* curves
- There are several ways of defining them

Cubic Bézier Curves

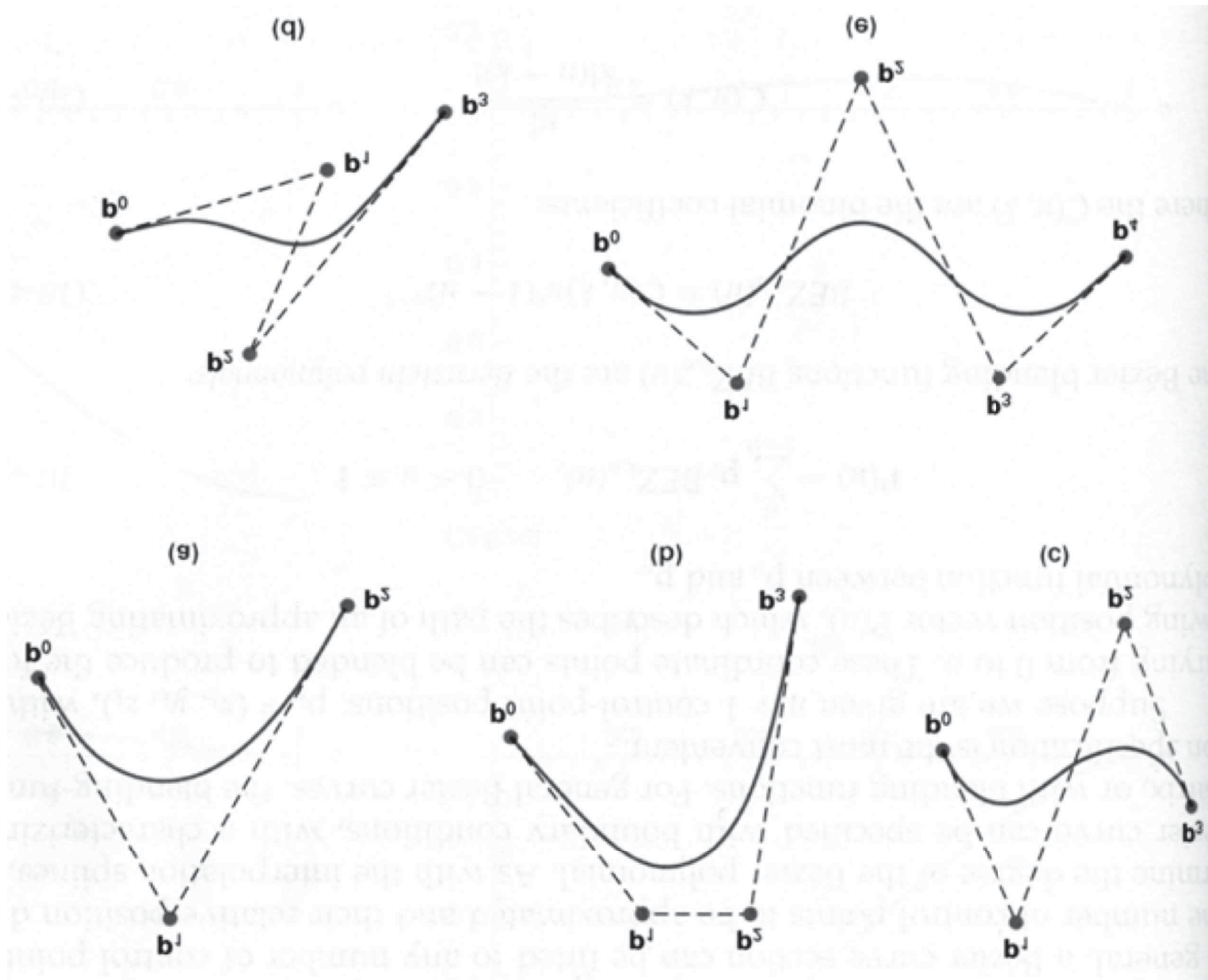
- Curves defined by *4* points
- Curve passes through two points
  - contained in *convex hull* of points



©T. Munzner, UBC

$$p_{00}(t) = \begin{bmatrix} p_{03} & p_{12} & p_{21} & p_{30} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

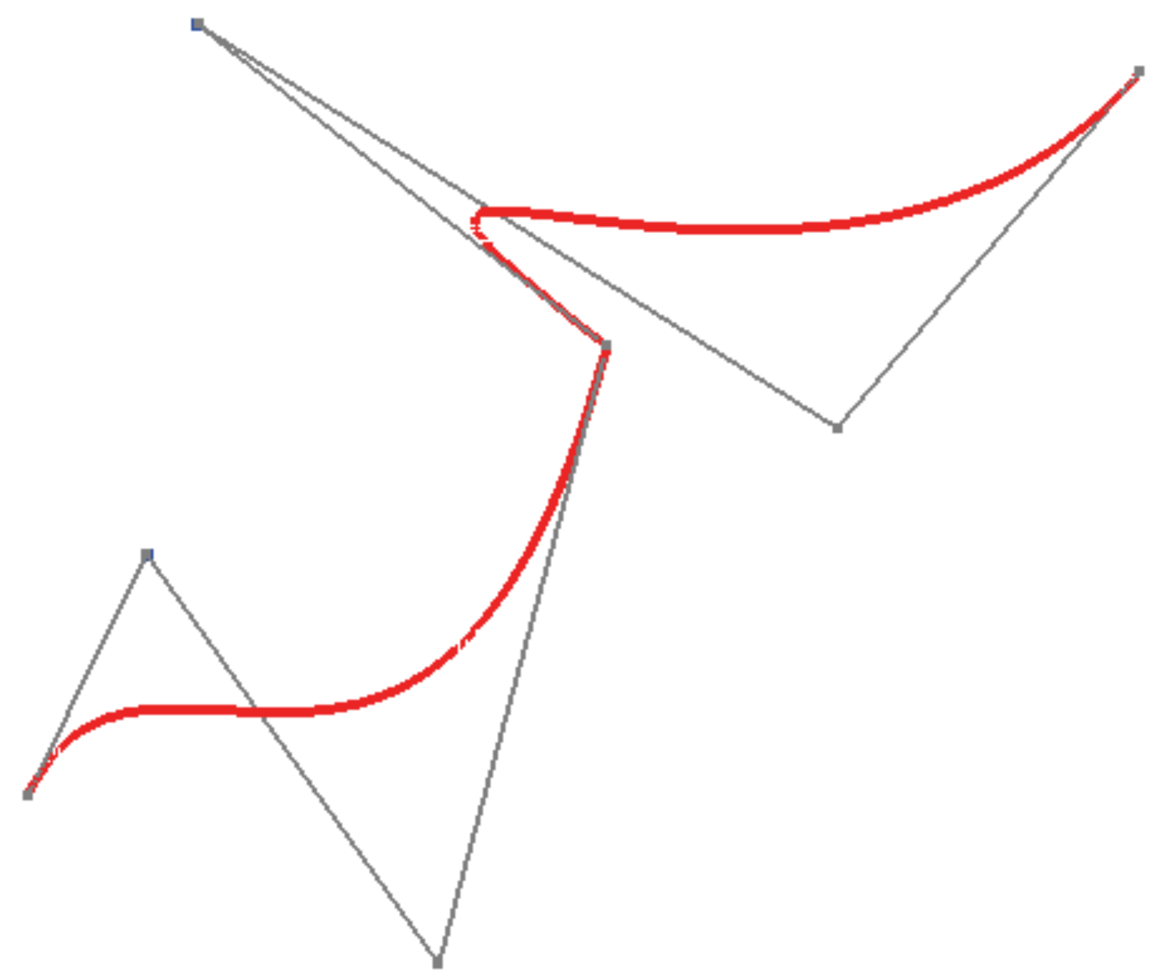
Some Examples



©T. Munzner, UBC

Piecewise Béziers

- Convenient, but
  - *not* C^1 continuous
  - *not* G^1 continuous
  - need 4 points /piece
  - we want *slopes* to match
    - rather like line strips



B-splines

- A *spline* is any piecewise-cubic curve
- B-splines use a different *matrix*:
    - identical to Béziers except last row

$$x(t) = \begin{bmatrix} x_{i-2} & x_{i-1} & x_i & x_{i+1} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} (t-i)^3 \\ (t-i)^2 \\ (t-i)^1 \\ 1 \end{bmatrix}$$

B-splines

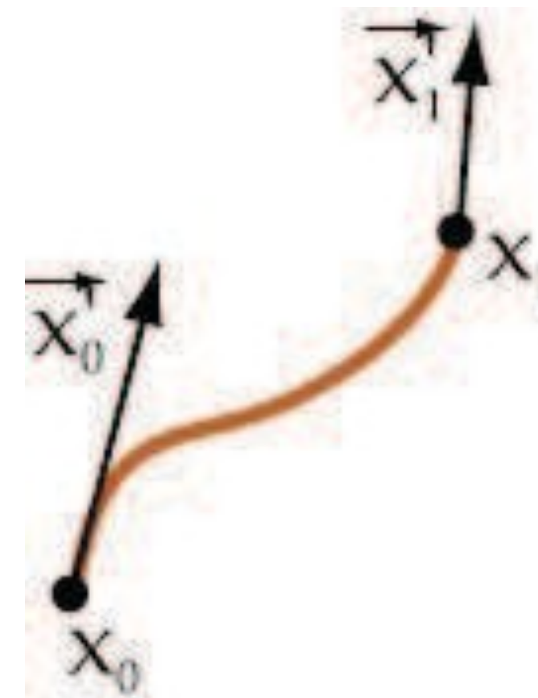
- Each control point is called a *knot*
- Only need *m+3* points for *m* pieces
- The pieces of the function are *uniform*
  - i.e. each piece is length 1 *(i .. i+1)*
- And they are G^1 continuous

Hermite Curves

- Used in *drawing* software
  - Adobe Illustrator, &c.
  - Vectors shown as *handles*
- Not always easy to get desired result

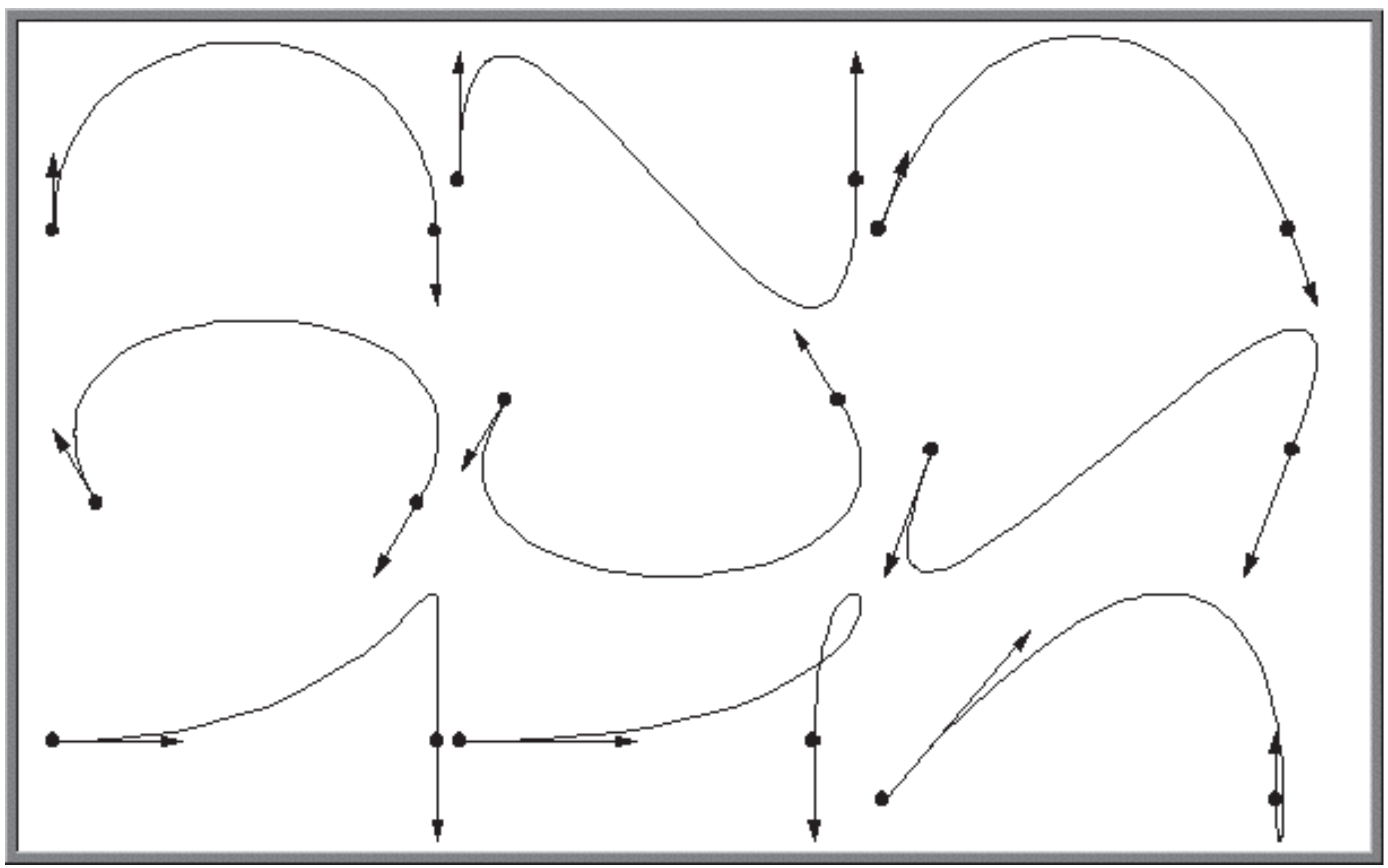
Hermite Curves

- A *Hermite* curve is given by:
  - 2 *endpoints* x_1, x_0
  - 2 *slopes* $\vec{x}_1{}', \vec{x}_0{}'$
    - Given by this equation:



$$x(t) = \begin{bmatrix} x_1 & x_0 & \vec{x}_1{}' & \vec{x}_0{}' \end{bmatrix} \begin{bmatrix} -2 & 3 & 0 & 0 \\ 2 & -3 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & -2 & 1 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t^1 \\ 1 \end{bmatrix}$$

Hermite Examples



©T. Munzner, UBC

Conversion

- Hermites can be converted to Béziers

$$\begin{bmatrix} p_{03} & p_{12} & p_{21} & p_{30} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t^1 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_0 & \vec{x}_1{}' & \vec{x}_0{}' \end{bmatrix} \begin{bmatrix} -2 & 3 & 0 & 0 \\ 2 & -3 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & -2 & 1 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t^1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} p_{03} & p_{12} & p_{21} & p_{30} \end{bmatrix} = \begin{bmatrix} x_1 & x_0 & \vec{x}_1{}' & \vec{x}_0{}' \end{bmatrix} \begin{bmatrix} -2 & 3 & 0 & 0 \\ 2 & -3 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & -2 & 1 & 0 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}^{-1}$$

$$\begin{bmatrix} p_{03} & p_{12} & p_{21} & p_{30} \end{bmatrix} = \begin{bmatrix} x_1 & x_0 & \vec{x}_1{}' & \vec{x}_0{}' \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & -3 & 0 \end{bmatrix}$$

Other Curves / Surfaces

- Other types of curves / surfaces include:
  - *higher-order: quadrics*, *multi-linear*, *Gaussian*
  - *limit surfaces:* defined by iterative refinement
    - *fractals*, *subdivision surfaces*
  - *geometric* surfaces: *spheres*, *hyperbolic surfaces*
  - *contours:* defined by $\left\{p \in \mathbb{R}^d : f\left(p\right) = h\right\}$
    - *contour lines*, *isosurfaces*, *soft (blobby) surfaces*