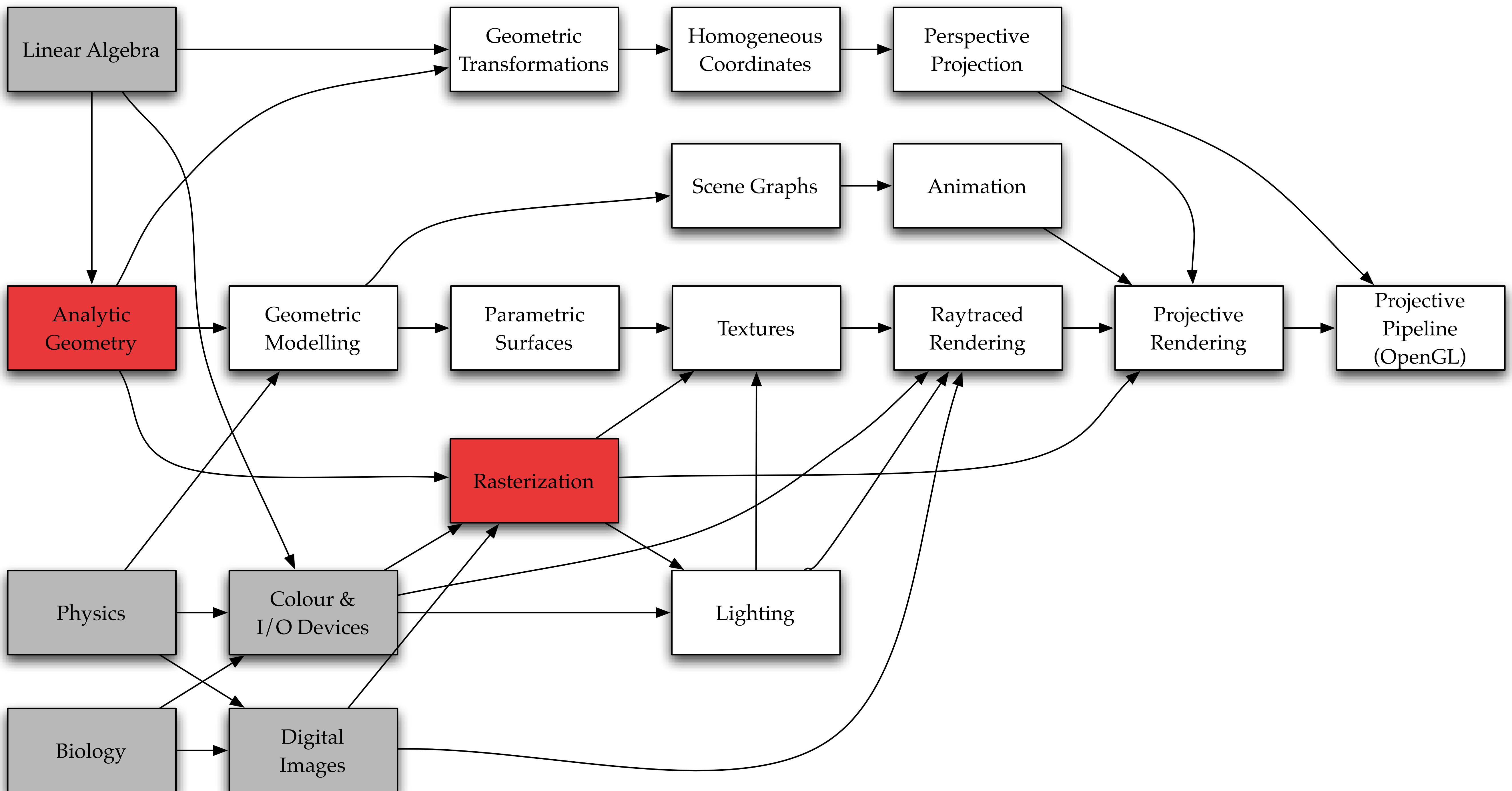
Curves & Circles

COMP 30020

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

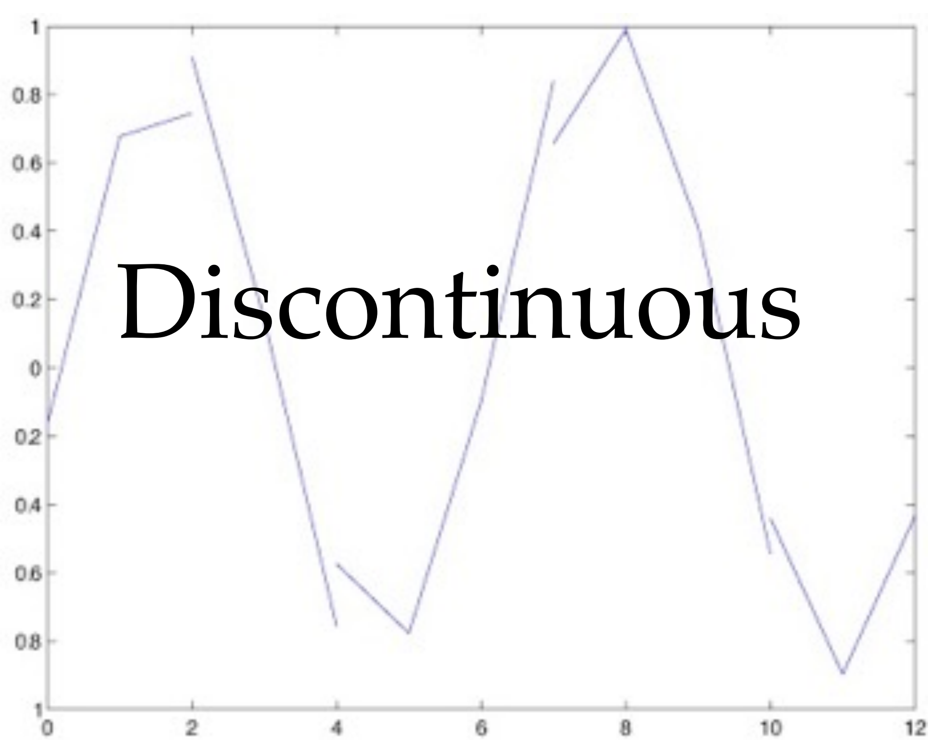
- Nature doesn't use straight lines (much)
  - let alone triangles
- Humans often use curves as well
  - how do we *represent* them?
  - how do we *rasterize* them?
- What is the difference between curves and lines?

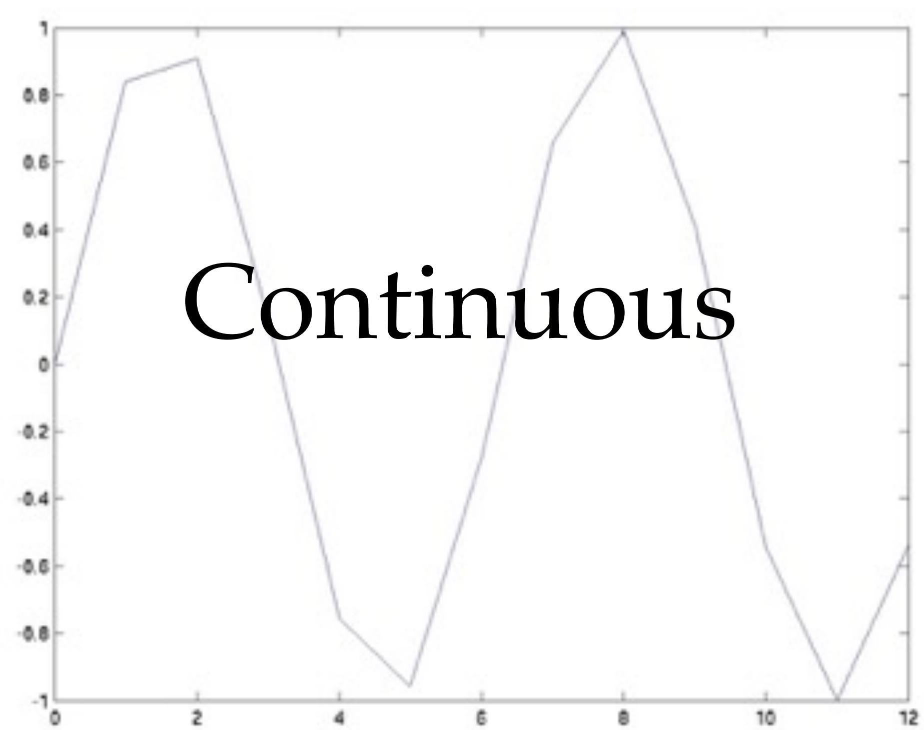
Continuity

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

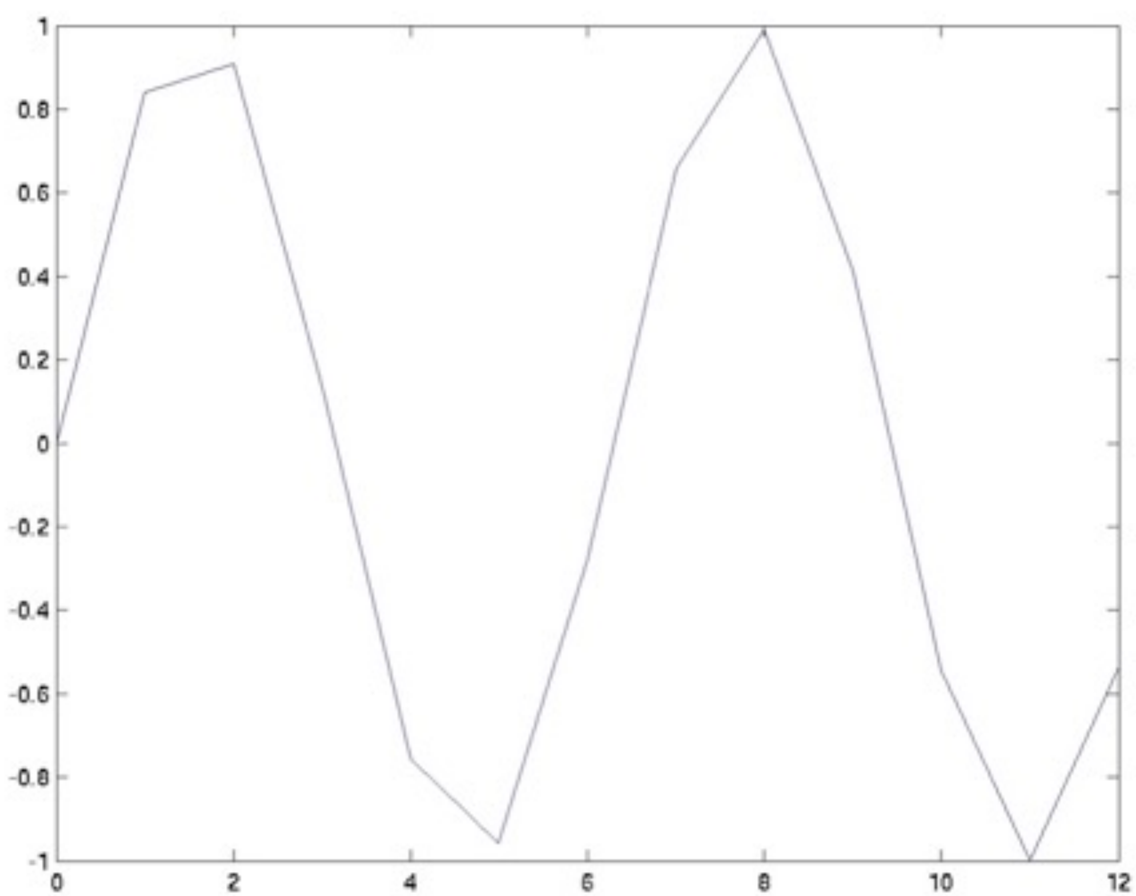
$$\lim_{x \to a^-} f(x) = f(a) = \lim_{x \to a^+} f(x)$$

  - Also called C^0 continuous

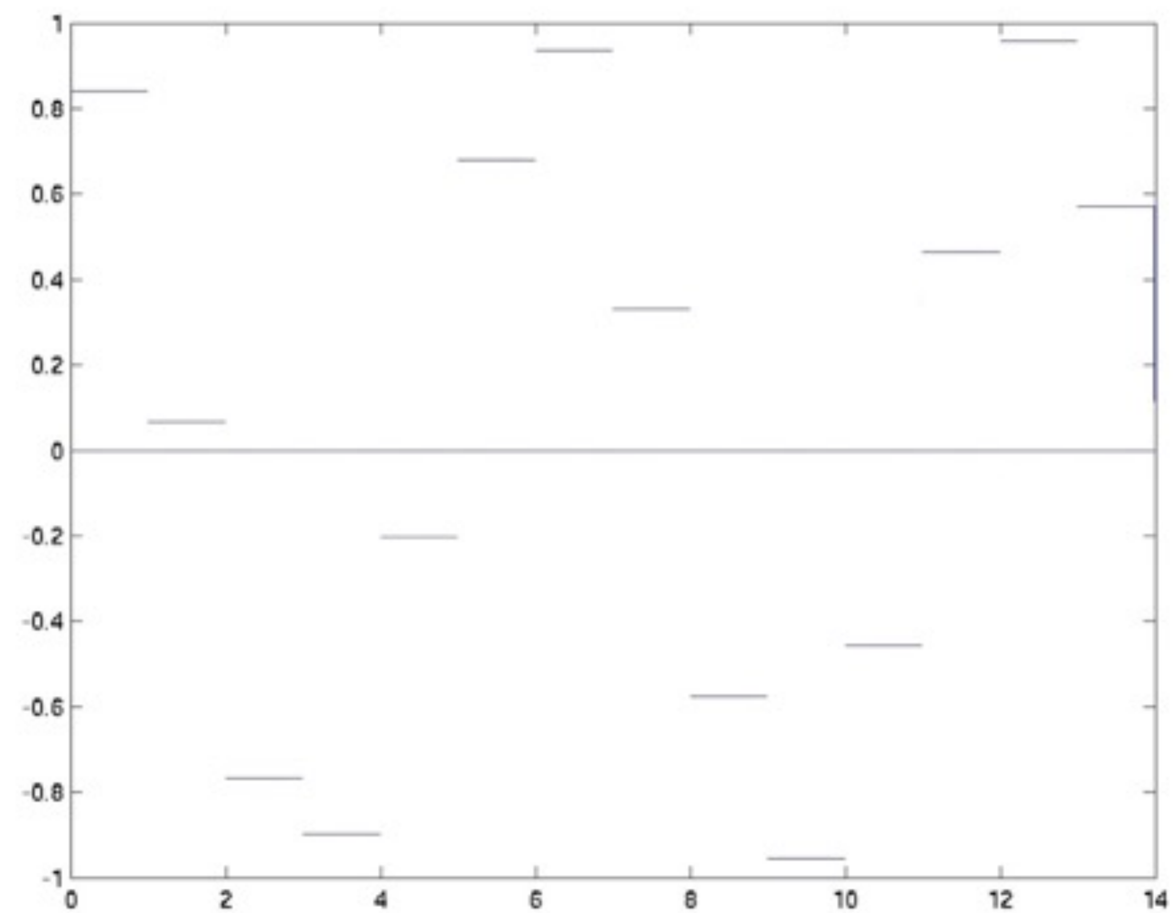




Continuous ≠ Smooth



Not *smooth*
Why not?

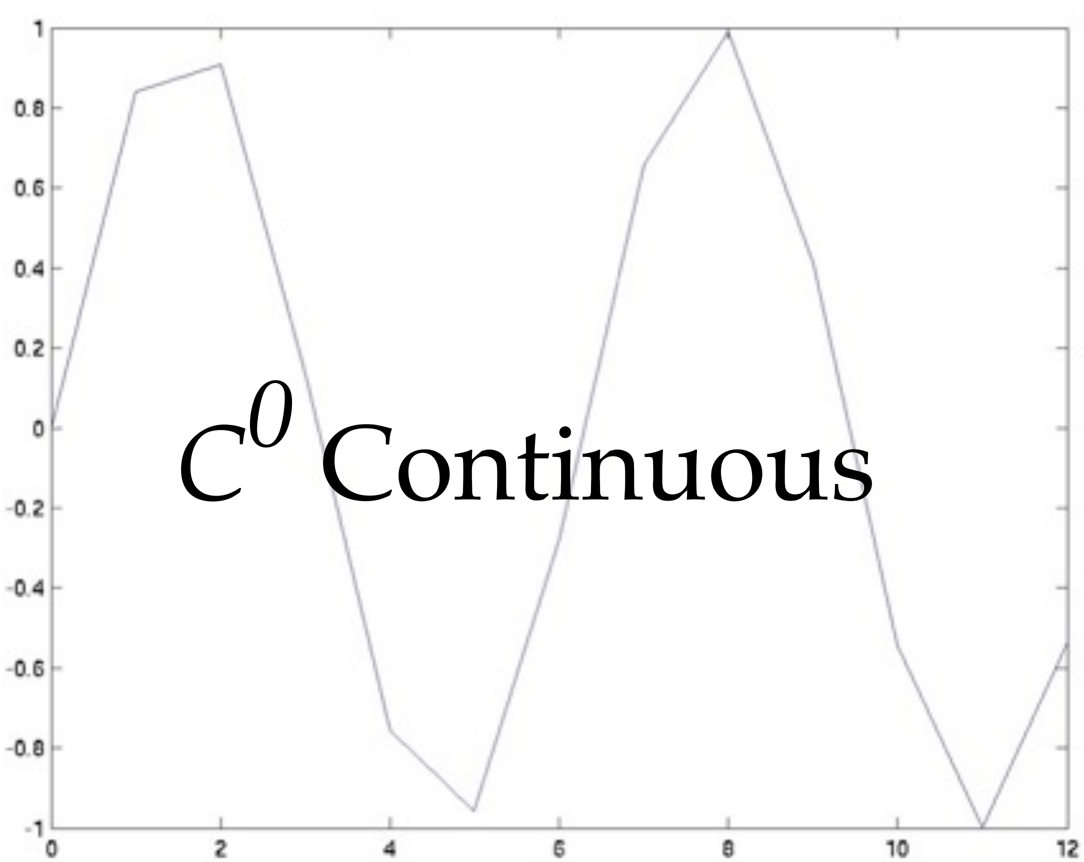


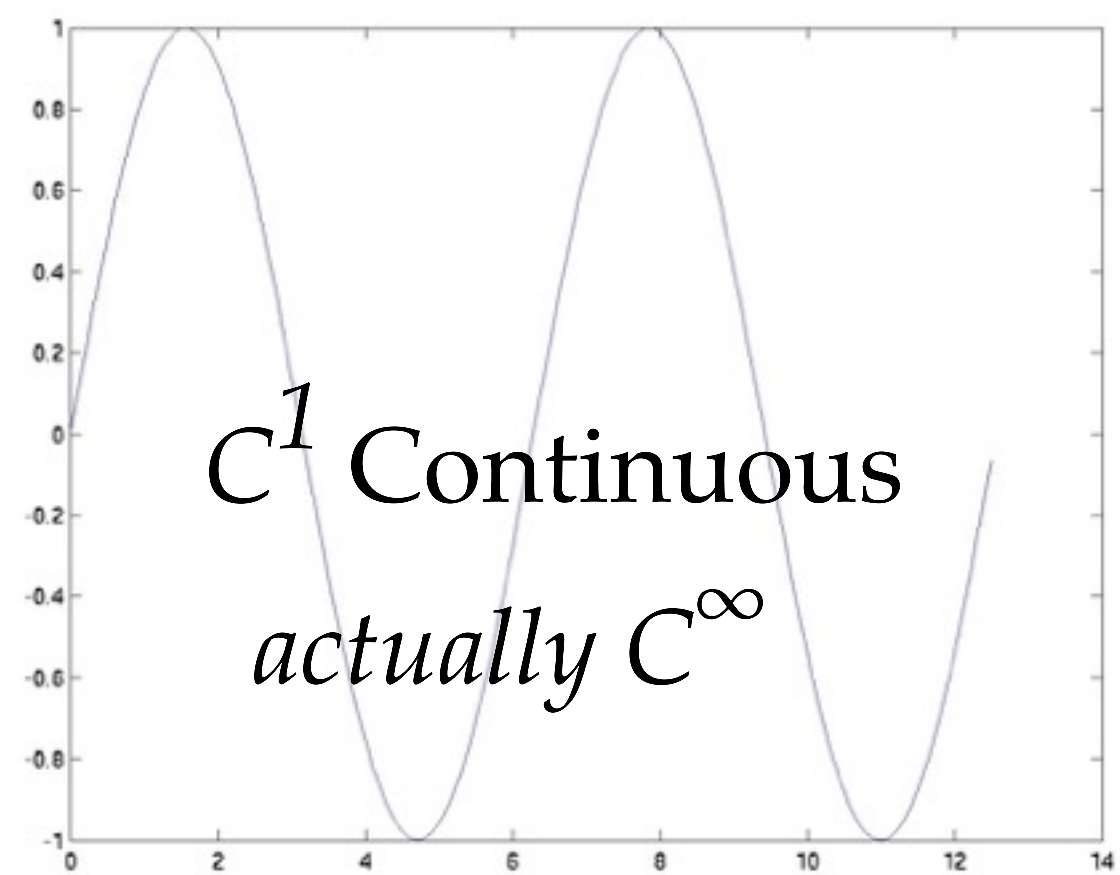
Slope (derivative)
slope is discontinuous

C^n Continuity

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

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



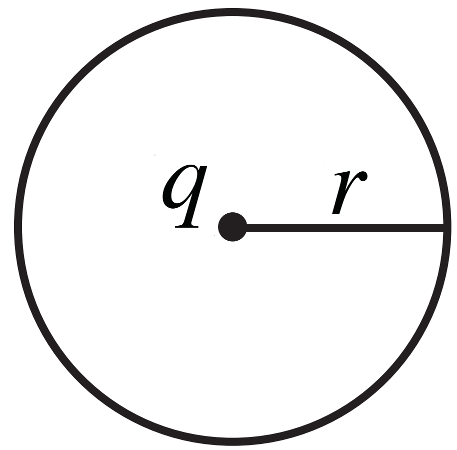


Smoothness

- 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*



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

Explicit Form

Implicit form:

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

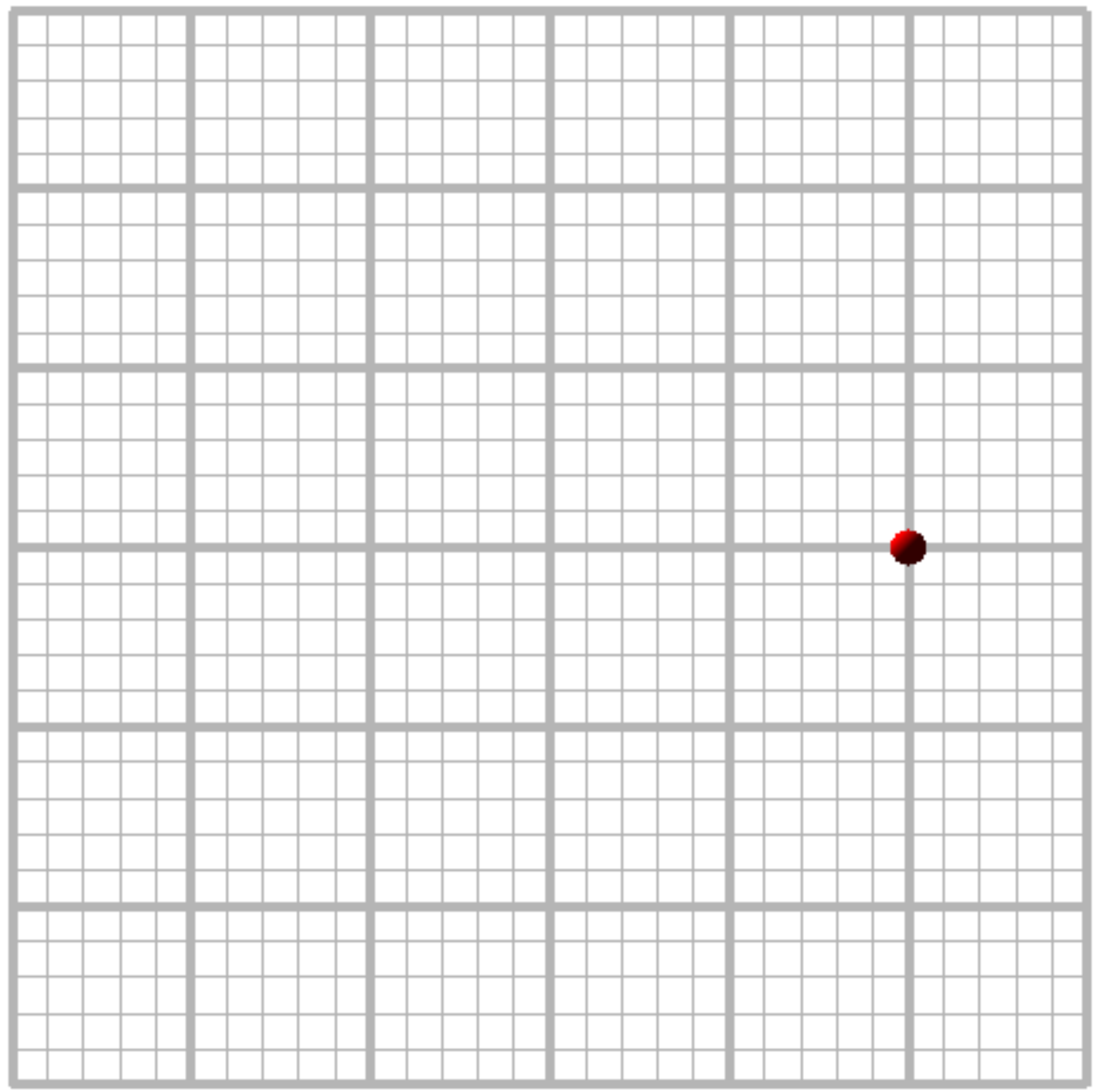
Explicit form:

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

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

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

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);
  }
```

- Same problems as for lines

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

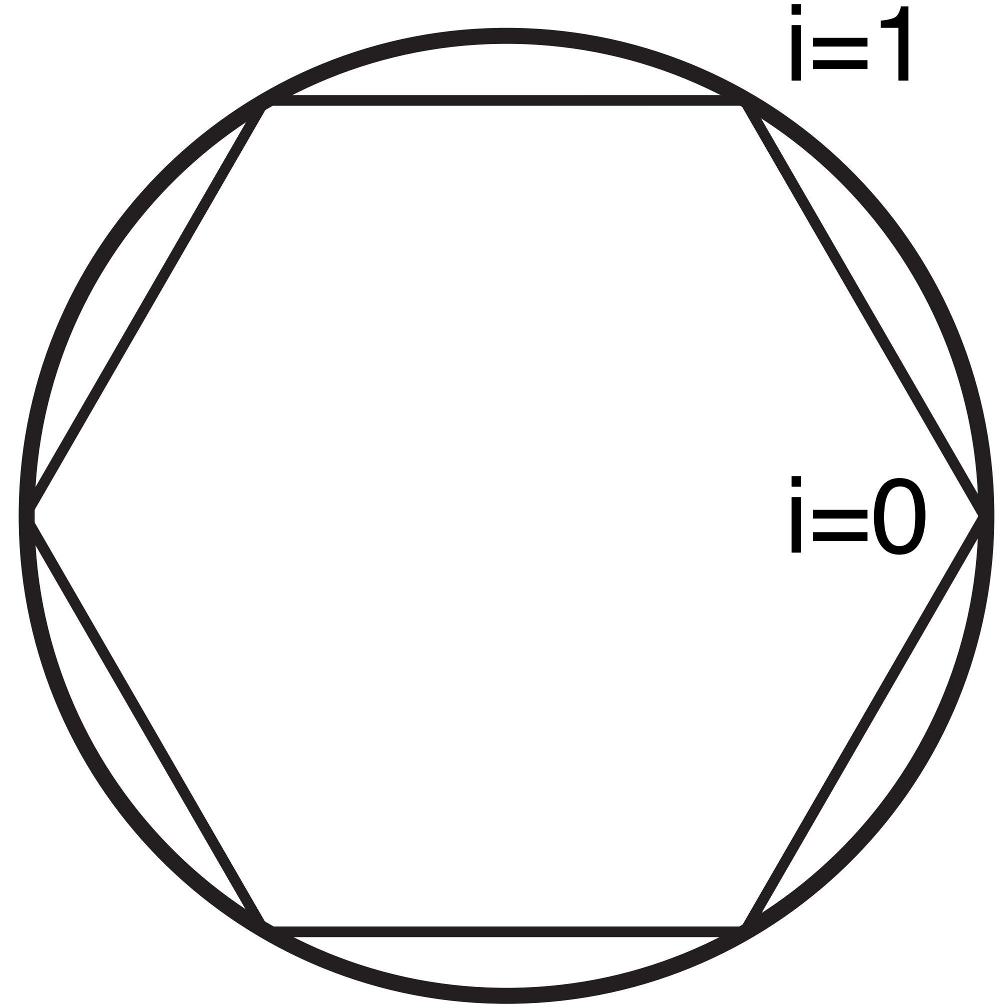
- Simple (as usual)

```
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

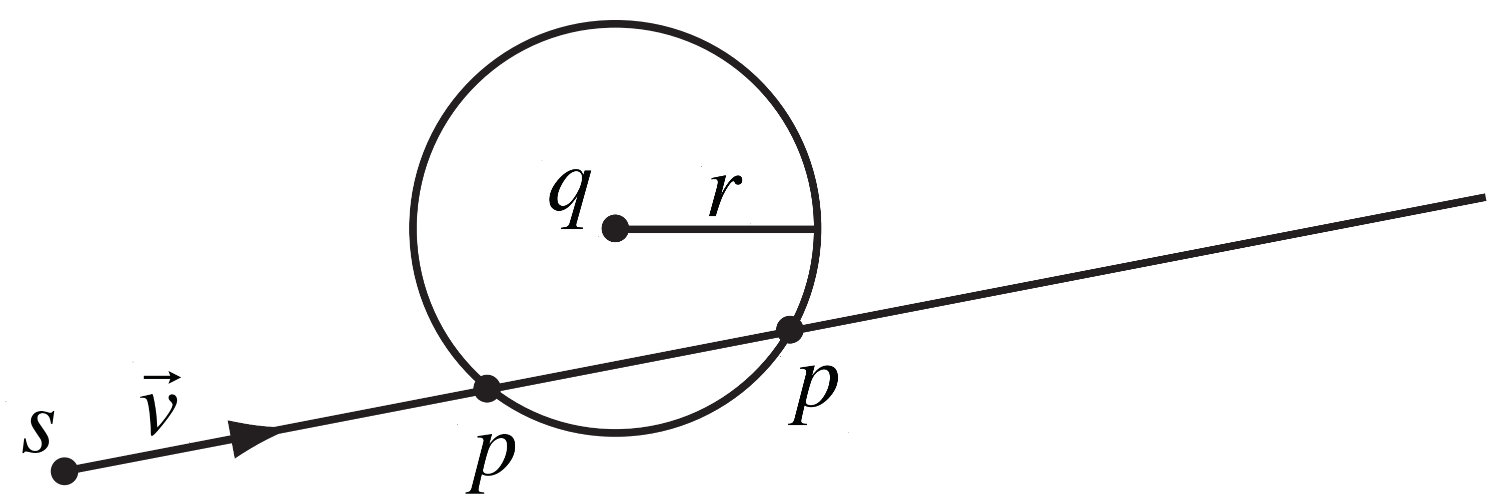
- Explicit: Raster Scan still works
- Implicit: Use $<= r$, not $== r$
- Parametric: use r as second parameter
- Lines: draw triangle $(p1, p2, q)$
- What about interpolation?

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 $\overleftrightarrow{l} = s + \vec{v}t$
- 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

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

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

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

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;
    return true; }
  else return false;
  } // end of Intersect()
```

Other Curves

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