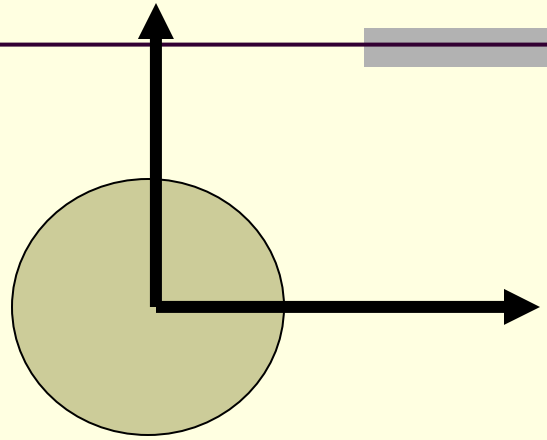
CS335
Fall 2007

2D Drawings: Circles and Curves

Representing A Circle



Explicit representation:

$$y = \pm\sqrt{r^2 - x^2}$$

Implicit representation:

$$F(x, y) = x^2 + y^2 - r^2$$

$$F(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2$$

Parametric representation:

$$x = r\cos\theta$$

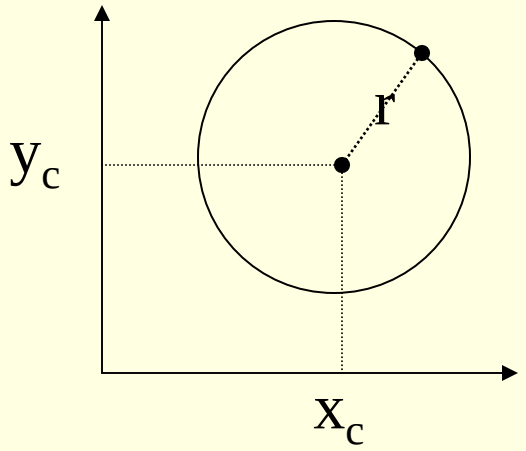
$$y = r\sin\theta$$

Drawing Circles

Equation for a circle:
Given: center = (x_c, y_c) and radius ***r***

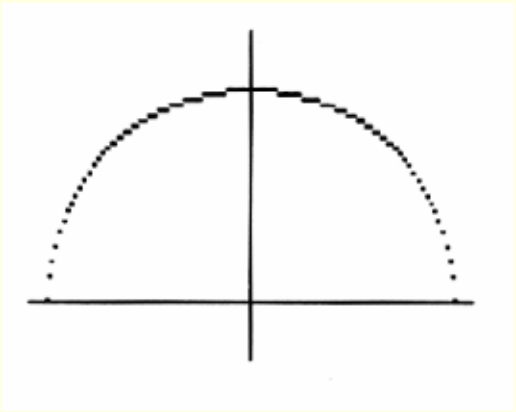
$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2}$$



Not an ideal approach:

Non-uniform spacing during evaluation causes noticeable artifacts

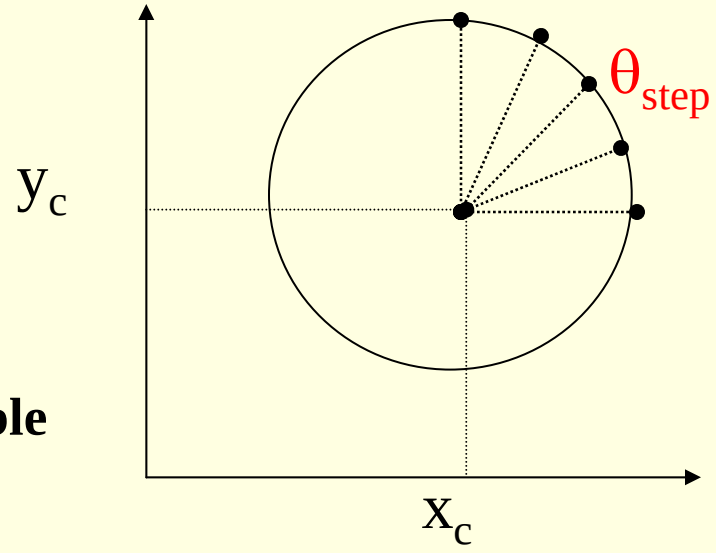


Also, we have to evaluate ^2, and sqrt()

Drawing Circles

Parametric Polar Representation for circle:

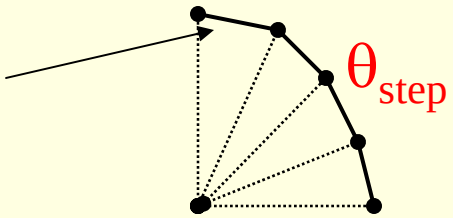
$$x = x_c + r\cos\theta$$
$$y = y_c + r\sin\theta$$



Allows a uniform spacing via the θ variable

θstep size is often set to be 1/r.

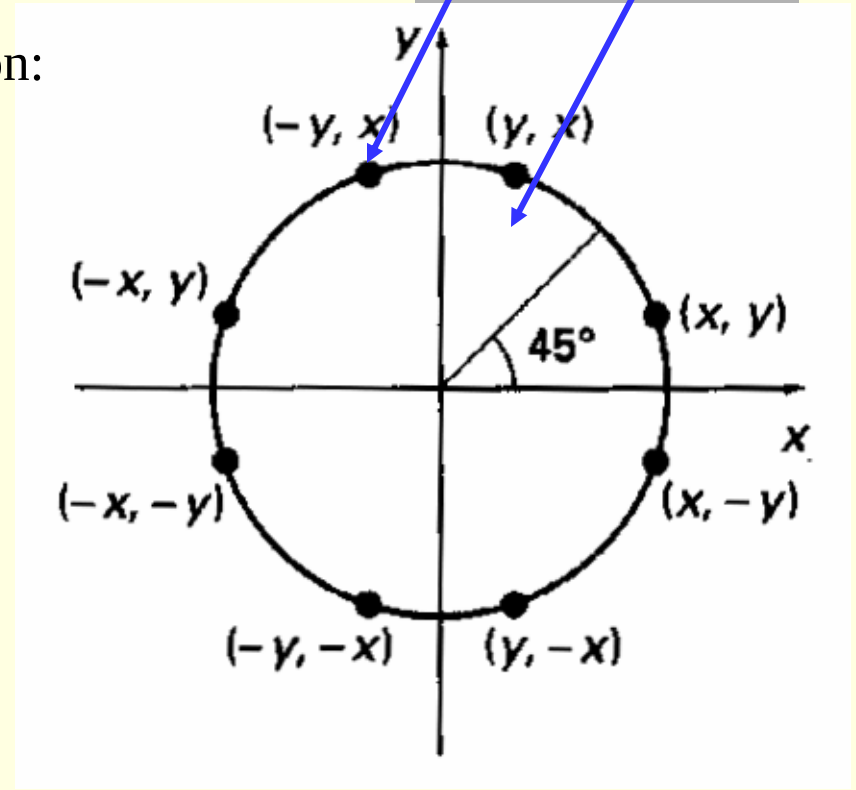
We can render adjacent points via lines (to avoid gaps)



Drawing Circles

Only need to compute 1 octant

We can reduce the number of calculation:

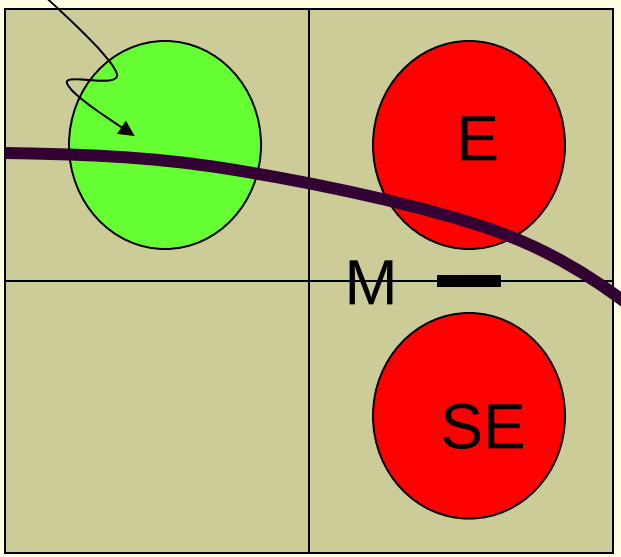


Exploit Symmetry of Circle

Need only to compute $\theta \in (0,45°)$ or $(90°,45°)$

The Circle Midpoint Algorithm

(x_p, y_p)



Key idea: at each step, decide which pixel (E or SE) is closest to the circle. Choose that pixel and draw it.

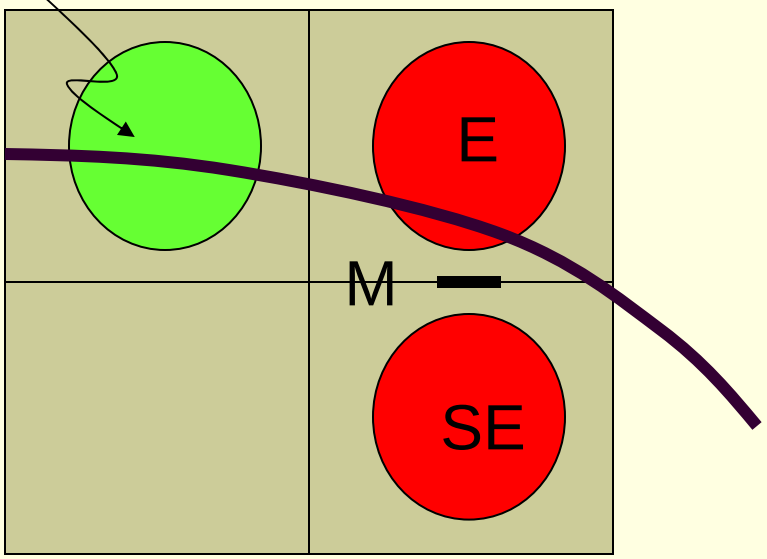
Goals:

Implicit formulation
Finite differences (incremental)
Integer arithmetic

The Implicit Circle Equation

If M lies inside the circle, draw pixel E
If M lies outside the circle, draw pixel SE

(x_p, y_p)



Implicit Formulation

$$F(x, y) = x^2 + y^2 - r^2 = 0$$

Notice that $(x^2 + y^2) > r^2$ means $F(x, y) > r^2$

$$(x^2 + y^2) > r^2$$

$$y^2 > r^2 - x^2$$

$$y > \sqrt{r^2 - x^2}$$

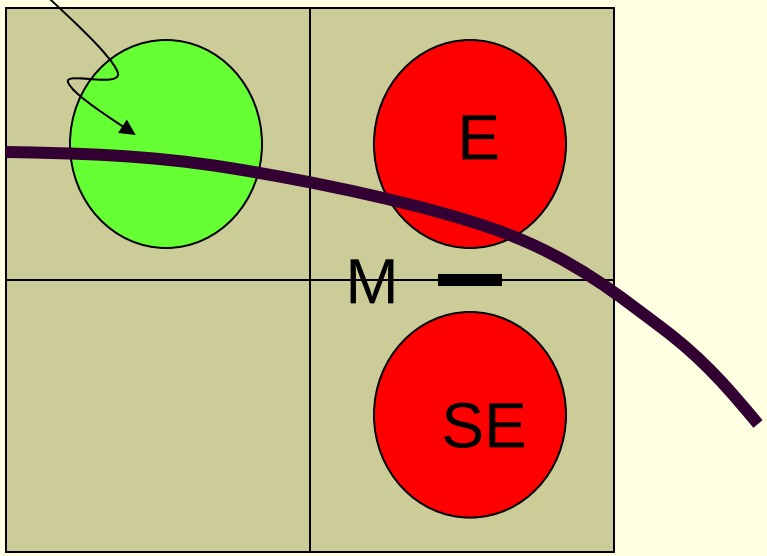
$F(x, y) > 0$ points outside circle

$F(x, y) < 0$ points inside circle

The Midpoint Test

(x_p, y_p)

$$M = (x_p + 1, y_p - \tfrac{1}{2})$$



The sign of F(M) gives the answer as to which pixel, E or SE, to draw

Formulating an Algorithm

Let d be a decision variable which makes the midpoint test. Then the test to decide which pixel to draw is just

```
if (d < 0)
then
     //  the midpoint is inside the circle
     //  the circle passes closer to the upper pixel
     draw the E pixel
else
     //  the midpoint is outside the circle
     //  the circle passes closer to the lower pixel
     draw the SE pixel
```

Formulating an Algorithm

```
MidpointCircle (radius, x_center, y_center)

x = 0;  y = radius;
CirclePoints (x_center, y_center, x, y);
while (y > x) {
   if (d < 0)  //  the midpoint is inside the circle
               //  the circle is closer to E pixel
       x = x + 1;
       CirclePoints (x_center, y_center, x, y);
   else        //  the midpoint is outside the circle
               //  the circle is closer to SE pixel
       x = x + 1;  y = y - 1;
       CirclePoints (x_center, y_center, x, y);
end
```

Formulating an Algorithm

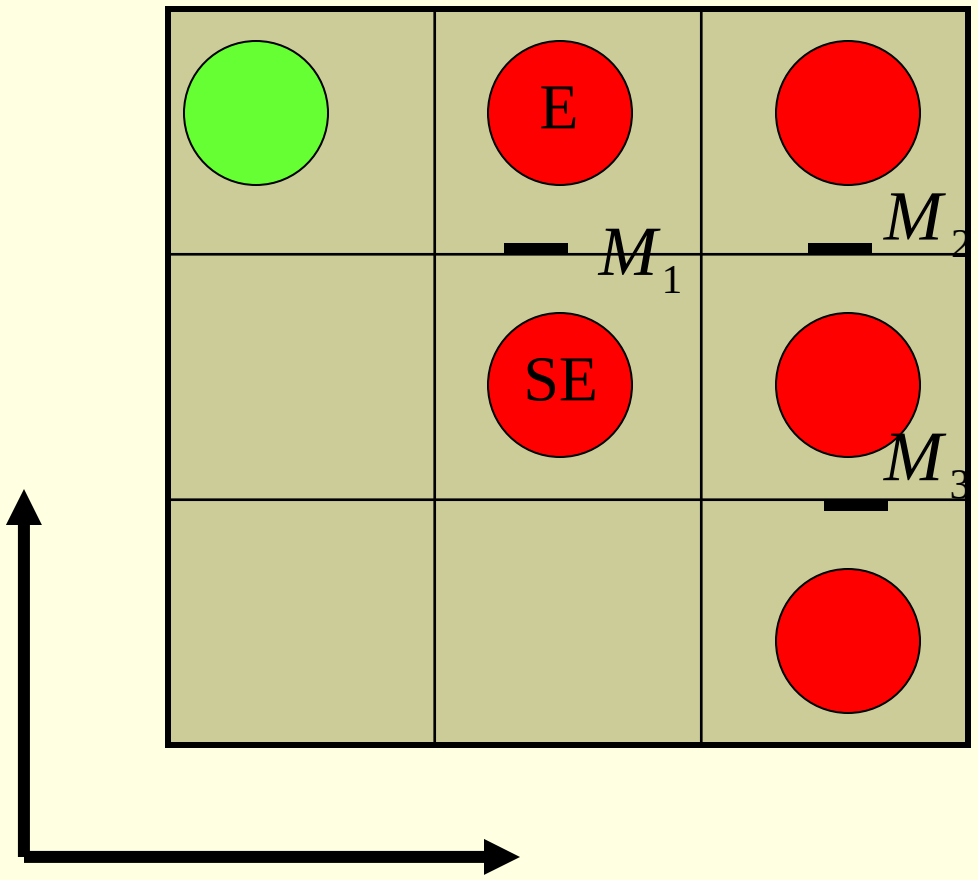
```
CirclePoints (a, b, x, y)

   DrawPoint (a + x, b + y);
   DrawPoint (a + x, b - y);
   DrawPoint (a - x, b + y);
   DrawPoint (a - x, b - y);
   DrawPoint (a + y, b + x);
   DrawPoint (a + y, b - x);
   DrawPoint (a - y, b + x);
   DrawPoint (a - y, b - x);

end
```

Making the Algorithm Incremental

(x_p, y_p) $(x_p + 1, y_p)$ $(x_p + 2, y_p)$



$$M_1 = (x_p + 1, y_p - \tfrac{1}{2})$$

$$M_2 = (x_p + 2, y_p - \tfrac{1}{2})$$

$$M_3 = (x_p + 2, y_p - \tfrac{3}{2})$$

Incremental Formulation

$$d_{first} = F(M_1) = F(x_p + 1, y_p - \tfrac{1}{2})$$
$$= (x_p + 1)^2 + (y_p - \tfrac{1}{2})^2 - r^2$$

If we choose E after this computation, we must compute $F(M_2)$ next.

What is the relationship between

$$d_{first} = F(M_1) \text{ and } F(M_2)$$

Incremental Formulation

Notice that if we consider moving East:

$$d_{first} = (x_p + 1)^2 + (y_p - \tfrac{1}{2})^2 - r^2$$
$$d_{next} = (x_p + 2)^2 + (y_p - \tfrac{1}{2})^2 - r^2$$

(since $d_{next} = F(M_2))$ so that we can compute

$$d_{next} - d_{first} = 2x_p + 3$$

Or just

$$d_{next} = d_{first} + (2x_p + 3)$$

Incremental Formulation

Notice that if we consider moving Southeast:

$$d_{first} = (x_p + 1)^2 + (y_p - \tfrac{1}{2})^2 - r^2$$
$$d_{next} = (x_p + 2)^2 + (y_p - \tfrac{3}{2})^2 - r^2$$

(since $d_{next} = F(M_3)$) so that we can compute

$$d_{next} - d_{first} = 2x_p - 2y_p + 5$$

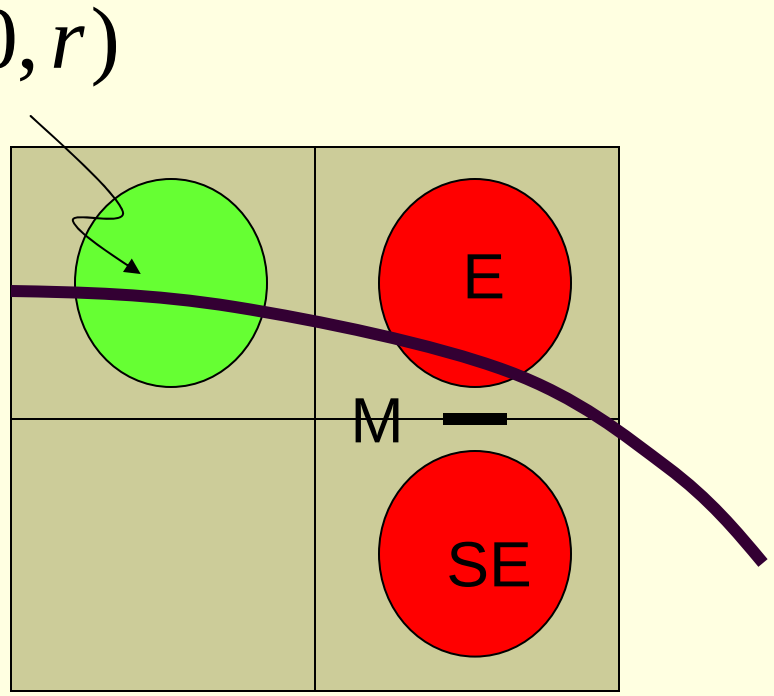
Or just

$$d_{next} = d_{first} + (2x_p - 2y_p + 5)$$

Initial Condition

The first point lies on the circle at point (0, r). The first midpoint is thus

$$F(1, r - \tfrac{1}{2})$$



The starting value for the test variable is

$$d_{start} = (1)^2 + (r - \tfrac{1}{2})^2 - r^2 = (\tfrac{5}{4} - r)$$

The Algorithm

```
MidpointCircle (r, xc, yc)
   x = 0;  y = r;  d = 5/4 - r;
   CirclePoints (xc, yc, x, y);
   while (y > x) do
      if (d < 0)
         d = d + 2*x + 3;
         x++;
      else
         d = d + 2*(x - y) + 5;
         x++;  y--;
      CirclePoints (xc, yc, x, y);
   end
end
```

Hearn/Baker errata:
Algorithm on page 102 should be:

p += 2*x +3
  else
p += 2*(x-y)+5 (see eq above)

Second Order Differences

Notice that the increments are functions of the point of evaluation (x, y) rather than simple constants:

East: $d_{next} = d_{first} + (2x_p + 3)$

Southeast: $d_{next} = d_{first} + (2x_p - 2y_p + 5)$

Key idea: incremental approach can be applied **again,** to the increments themselves

Second Order Differences (Four cases)

East (current iteration/point of evaluation):

- East increment

$$\Delta_{E\ \text{first}} \quad (x_p, y_p) = 2x_p + 3$$

$$\Delta_{E\ \text{next}} \quad (x_p + 1, y_p) = 2(x_p + 1) + 3$$

$$\boxed{\Delta_{E\ \text{next}} = \Delta_{E\ \text{first}} + 2}$$

- Southeast increment:

$$\Delta_{SE\ \text{first}} \quad (x_p, y_p) = 2x_p - 2y_p + 5$$

$$\Delta_{SE\ \text{next}} \quad (x_p + 1, y_p) = 2(x_p + 1) - 2y_p + 5$$

$$\boxed{\Delta_{SE\ \text{next}} = \Delta_{SE\ \text{first}} + 2}$$

Second Order Differences

Southeast (current iteration/point of evaluation):

- East increment

$$\Delta_{E\,\text{first}} \quad (x_p, y_p) = 2x_p + 3$$

$$\Delta_{E\,\text{next}} \quad (x_p + 1, y_p - 1) = 2(x_p + 1) + 3$$

$$\boxed{\Delta_{E\,\text{next}} = \Delta_{E\,\text{first}} + 2}$$

- Southeast increment:

$$\Delta_{SE\,\text{first}} \quad (x_p, y_p) = 2x_p - 2y_p + 5$$

$$\Delta_{SE\,\text{next}} \quad (x_p + 1, y_p - 1) = 2(x_p + 1) - 2(y_p - 1) + 5$$

$$\boxed{\Delta_{SE\,\text{next}} = \Delta_{SE\,\text{first}} + 4}$$

An Integer Algorithm

Let $h = d - \frac{1}{4}$ so that $d = h + \frac{1}{4}$

Since $d_{start} = \frac{5}{4} - r$ by substitution $h_{start} + \frac{1}{4} = \frac{5}{4} - r$

and we get $h_{start} = 1 - r$

Now, if the original midpoint test is $(d < 0)$

The new midpoint test becomes $(h < -\frac{1}{4})$

But since h starts as an integer and is only incremented by integers, the same test for h will hold:

$$(h < 0)$$

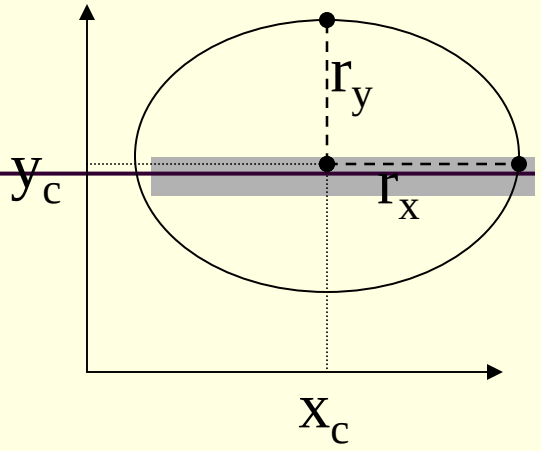
The Final Algorithm

```
MidpointCircle (r, xc, yc)
    x = 0;  y = r;  h = 1 - r;
    deltaE = 3;  deltaSE = 2*r + 5;
    CirclePoints (xc, yc, x, y);
    while (y > x) do
        if (h < 0)
            h = h + deltaE;
            deltaE = deltaE + 2;
            deltaSE = deltaSE + 2;
            x++;
```

The Final Algorithm

```
        else
            h = h + deltaSE;
            deltaE = deltaE + 2;
            deltaSE = deltaSE + 4;
            x++;  y++;
        CirclePoints (xc, yc, x, y);
    end
end
```

Drawing Ellipses



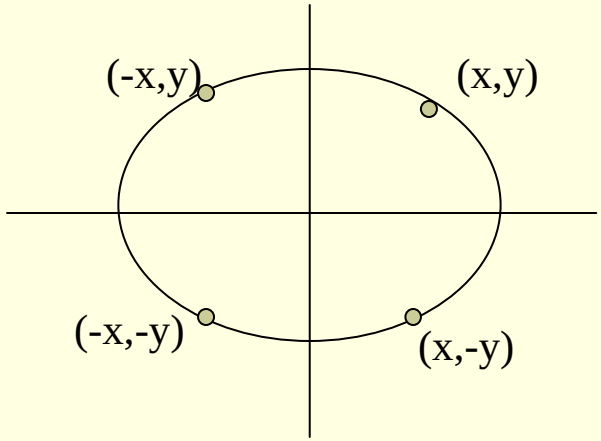
Using Polar Coordinates

$$x = x_c + r_x \cos\theta$$
$$y = y_c + r_y \sin\theta$$

Implicit

$$f(x,y)_{ellipse} = r_x^2 (x - x_c)^2 + r_y^2 (y - y_c)^2 - r_x^2 r_y^2$$

Exploit Symmetry



Midpoint Ellipse Algorithm
Hearn Page 109

Other Algebraic Surfaces

Bresenham approach is typically used for low-level scan conversion of Lines, Circles & Ellipses.

In principle, any algebraic curve can be expressed in the Bresenham style.

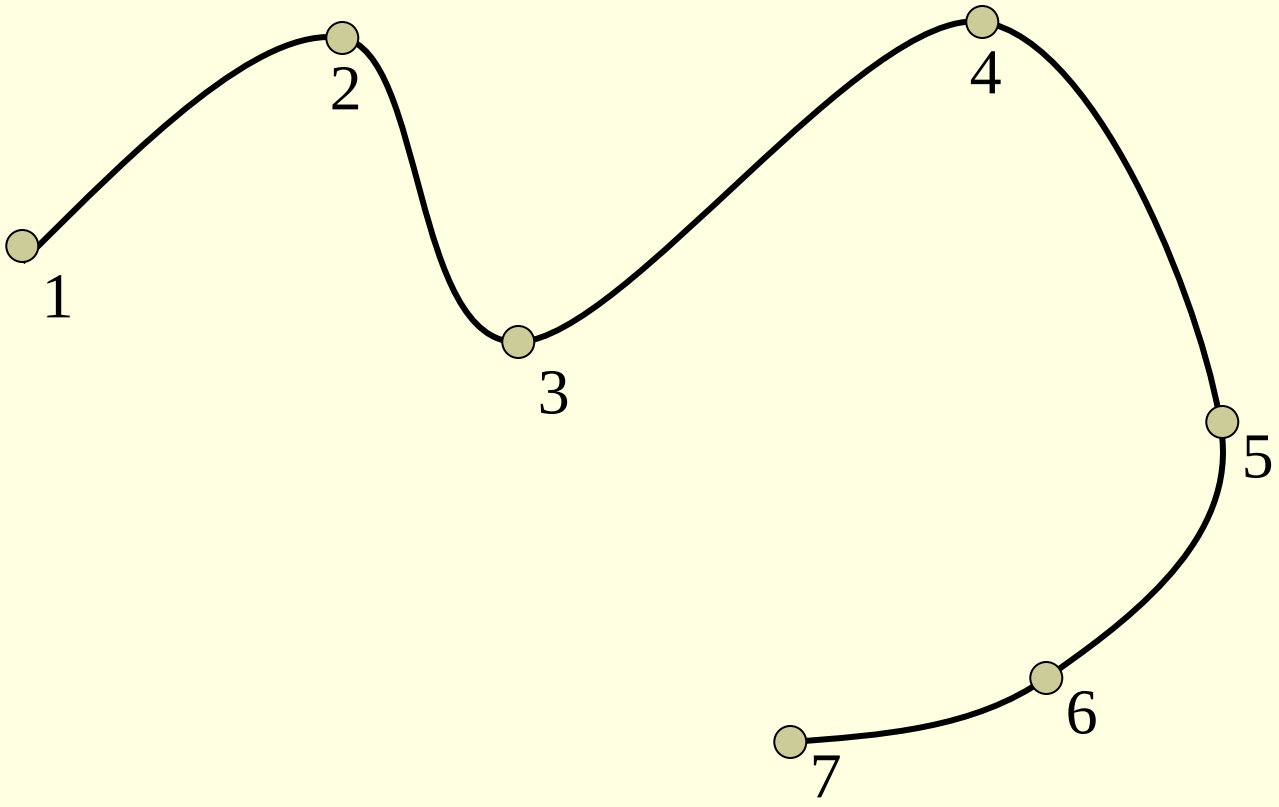
1. Measure error to actual curve (make decision)
2. Find integer form of error
3. Find incremental update of error criteria

In practice, curve beyond conics are drawn as a series of short Bresenham straight lines.

CS335
Graphics and Multimedia

2D Drawings: Splines

2D Freeform Curves or Splines



Splines

A typical free form curve designed to pass through or near a sequence of points.

Splines are parameterized curves that generalize linear interpolation of straight lines to higher powers of interpolation parameter.

Linear interpolation (1st order Spline)

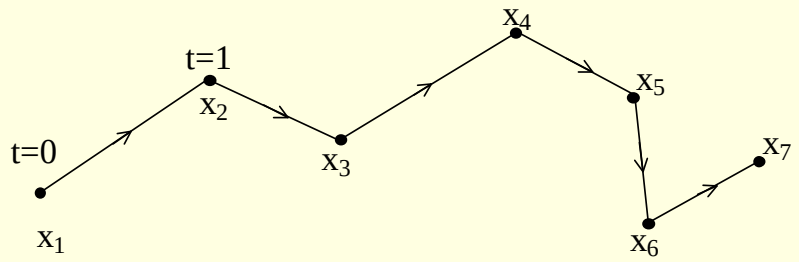
start : $\vec{x}_1 = (x_1 , y_1 , z_1 , \dots)$

end : $\vec{x}_2 = (x_2 , y_2 , z_2 , \dots)$

$$x(t) = x_1 + t(x_2 - x_1)$$
$$= (1-t)x_1 + tx_2$$

where : $x(0) = x_1$ and $x(1) = x_2$

Jagged Lines :



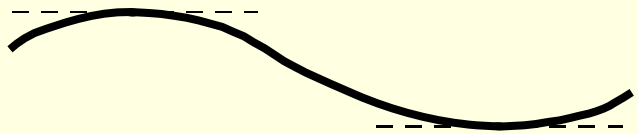
G^0: geometric continuity: the endpoints coincide

Quadratic interpolation

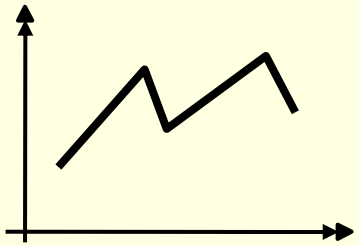
$$x(t) = a + bt + ct^2$$

$$= A(1-t)^2 + Bt(1-t) + Ct^2$$

matches derivatives



but 2*nd* derivative is choppy

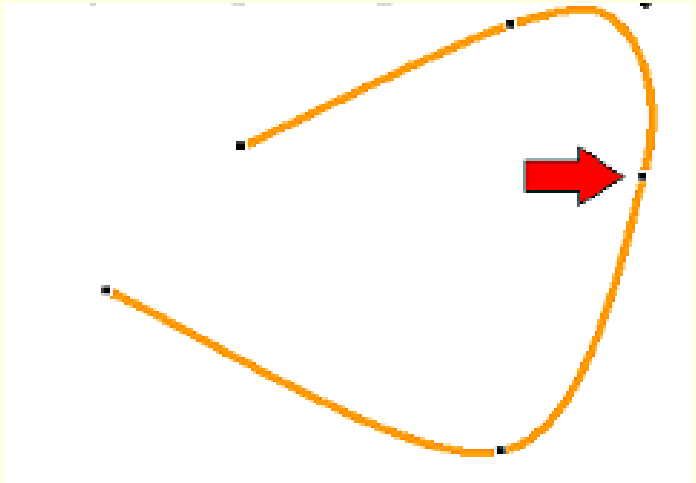


G^1: tangents have the same slope

Cubic Spline & Interpolation

$$\vec{x}(t) = \vec{A}(1-t)^3 + \vec{B}t(t-1)^2 + \vec{C}t^2(t-1) + \vec{D}t^3$$
$$= \vec{\alpha} + \vec{\beta}\,t + \vec{\gamma}\,t^2 + \vec{\delta}\,t^3$$

C^1: first derivative on both curves match at join point



Examples:

- Natural Cubic Spline
- Hermite
- Bezier
- B-spline

The "speed" is the same before and after

Representing Curves

- Cubic curve (to maintain C^1 continuity)

$$x = a_x t^3 + b_x t^2 + c_x t + d_x$$

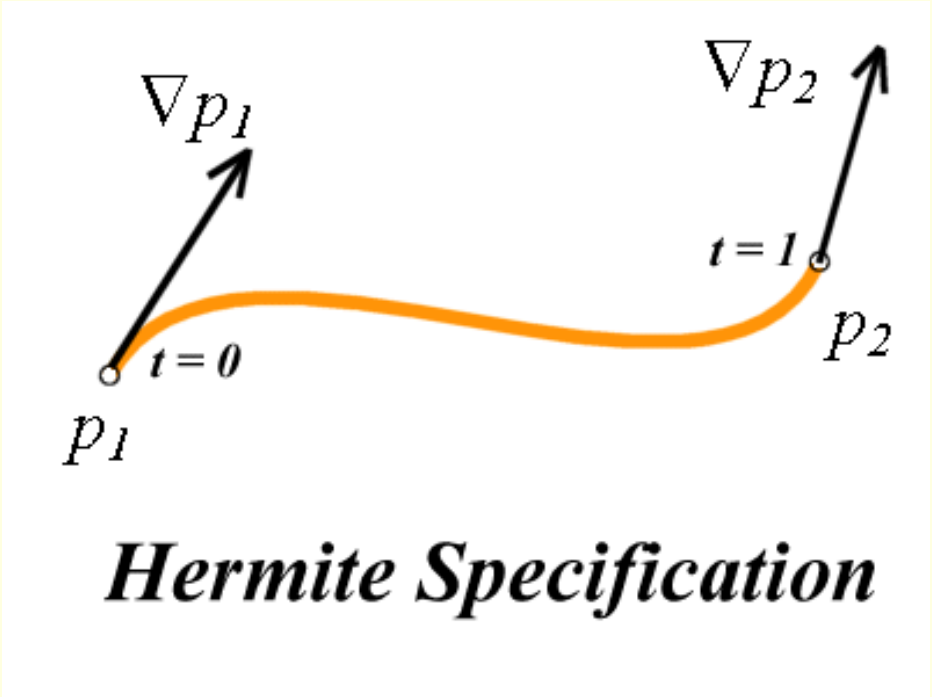
$$y = a_y t^3 + b_y t^2 + c_y t + d_y$$

- Matrix Representation

$$[x \quad y] = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ d_x & d_y \end{bmatrix}$$

Solving for Coefficients

- Continuity – C^0, C^1, etc.



The Gradient of a Cubic Curve

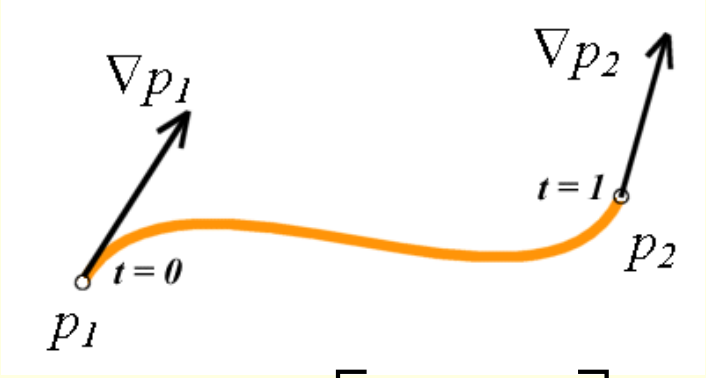
$$\frac{dx}{dt} = 3a_x t^2 + 2b_x t + c_x + 0$$

$$\frac{dy}{dt} = 3a_y t^2 + 2b_y t + c_y + 0$$

Matrix notion:

$$\left[\frac{dx}{dt} \quad \frac{dy}{dt}\right] = \begin{bmatrix} 3t^2 & 2t^1 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ d_x & d_y \end{bmatrix}$$

The Hermite Specification as a Matrix Equation



$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \frac{dx_1}{dt} & \frac{dy_1}{dt} \\ \frac{dx_2}{dt} & \frac{dy_2}{dt} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ d_x & d_y \end{bmatrix}$$

Solving the Hermite Coefficients

$$\begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \frac{dx_1}{dt} & \frac{dy_1}{dt} \\ \frac{dx_2}{dt} & \frac{dy_2}{dt} \end{bmatrix} = \begin{bmatrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ d_x & d_y \end{bmatrix}$$

M_{Hermite} G_{Hermite}

Cubic Hermite Spline Equation:

$$[x \quad y] = [\mathrm{t}^3 \quad \mathrm{t}^2 \quad \mathrm{t} \quad 1] M_{\text{Hermite}} G_{\text{Hermite}}$$

Another Way to Think About Splines

- Cubic Hermite Spline Equation

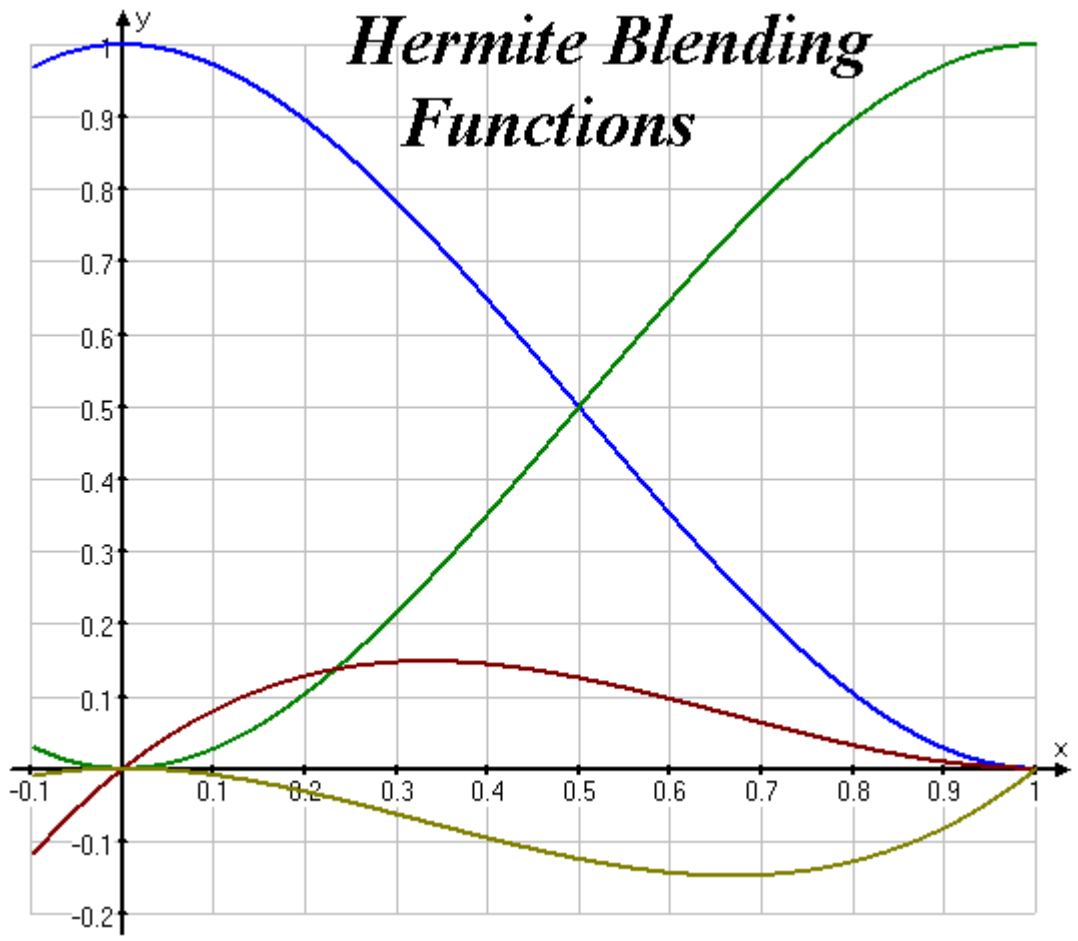
$$[\,x \quad y\,] = [\,\mathrm{t}^3 \quad \mathrm{t}^2 \quad \mathrm{t} \quad 1\,] M_{\text{Hermite}} \; G_{\text{Hermite}}$$

- After reordering multiplications

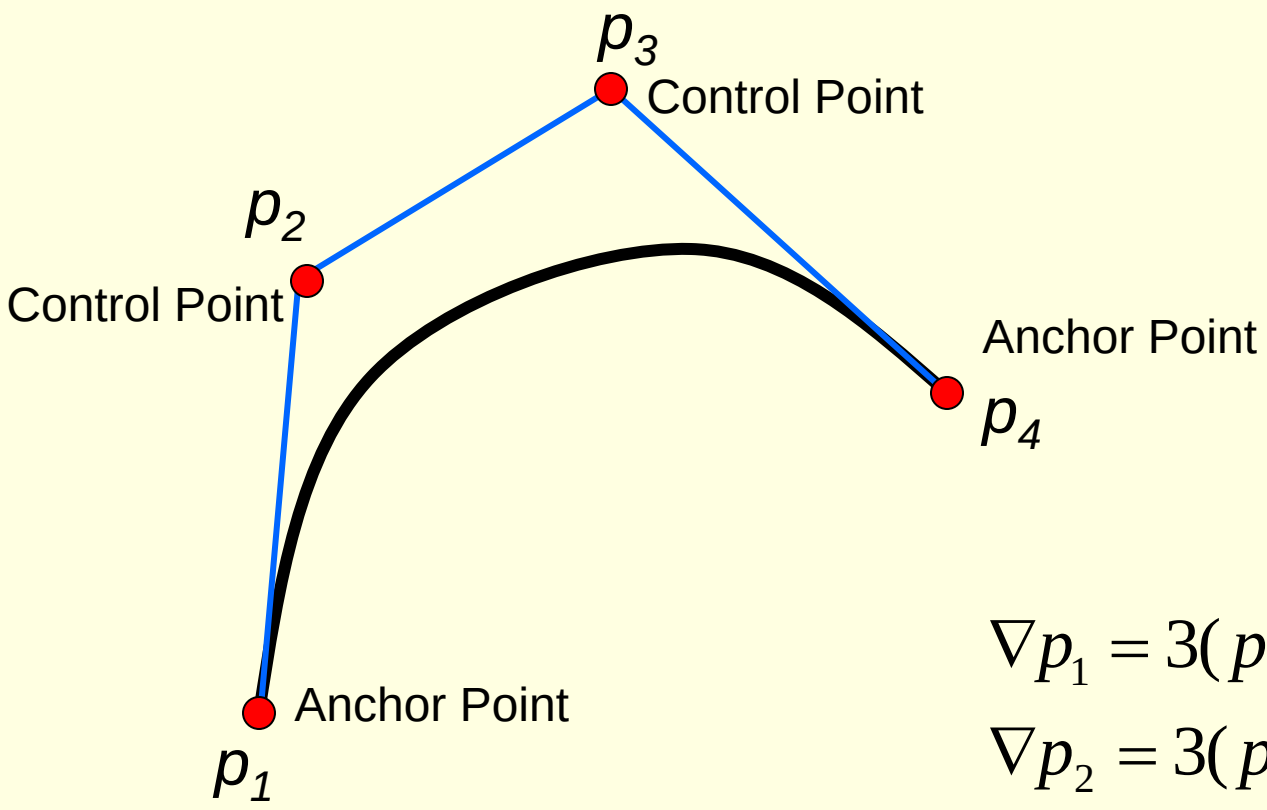
$$[x \quad y] = \begin{bmatrix} 2t^3 - 3t^2 + 1 \\ -2t^3 + 3t^2 \\ t^3 - 2t^2 + t \\ t^3 - t^2 \end{bmatrix}^T \begin{bmatrix} p_1 \\ p_2 \\ \nabla p_1 \\ \nabla p_2 \end{bmatrix}$$

$$= f_1(t)\,p_1 + f_2(t)\,p_2 + f_3(t)\,p_1' + f_4(t)\,p_2'$$

Hermite Blending Functions



Bezier Spline



$$\nabla p_1 = 3(p_2 - p_1)$$

$$\nabla p_2 = 3(p_3 - p_4)$$

Converting from Bezier to Hermite

Since

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \frac{dx_1}{dt} & \frac{dy_1}{dt} \\ \frac{dx_2}{dt} & \frac{dy_2}{dt} \end{bmatrix} = \overbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix}}^{T} \begin{bmatrix} x_1 & y_1 \\ x_x & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix}$$

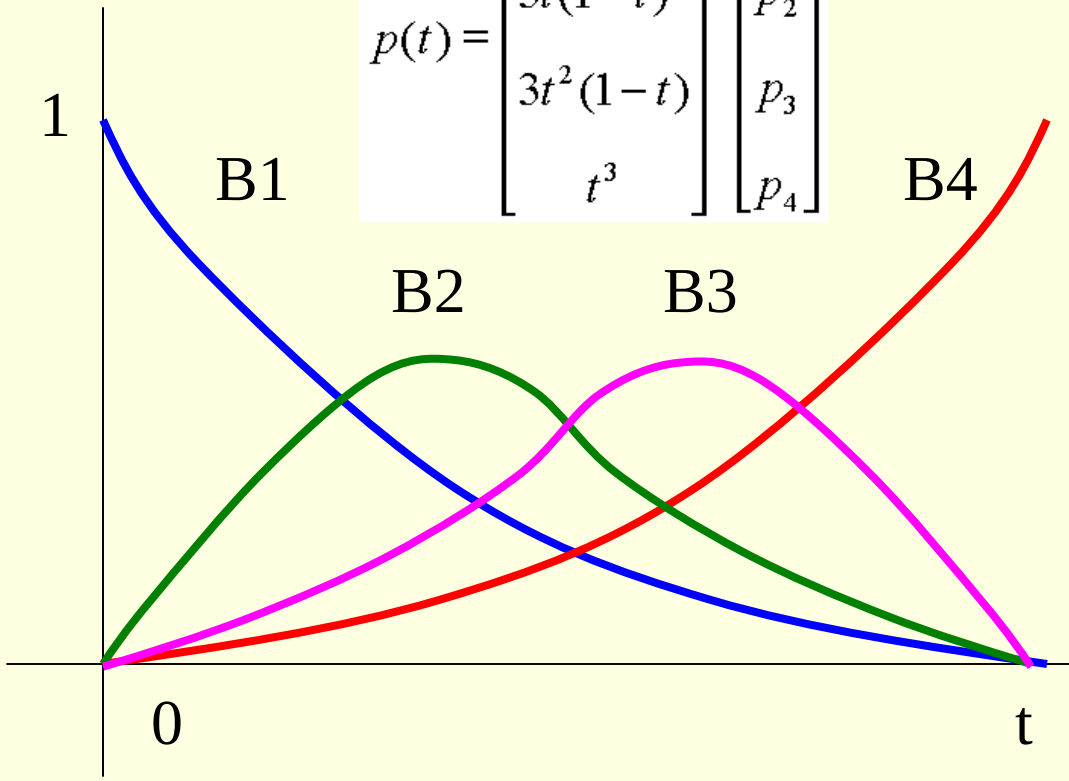
Substituting gives:

$$\begin{bmatrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ d_x & d_y \end{bmatrix} = M_{\text{Hermite}} T \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}}_{M_{\text{Bezier}}}$$

Blending Functions for Bezier Splines

$$p(t) = \begin{bmatrix} (1-t)^3 \\ 3t(1-t)^2 \\ 3t^2(1-t) \\ t^3 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$



$$B_1 \longrightarrow (1-t^3)$$
$$B_2 \longrightarrow (1-t)^2 3t$$
$$B_3 \longrightarrow (1-t)3t^2$$
$$B_4 \longrightarrow t^3$$

Bezier Curve Properties

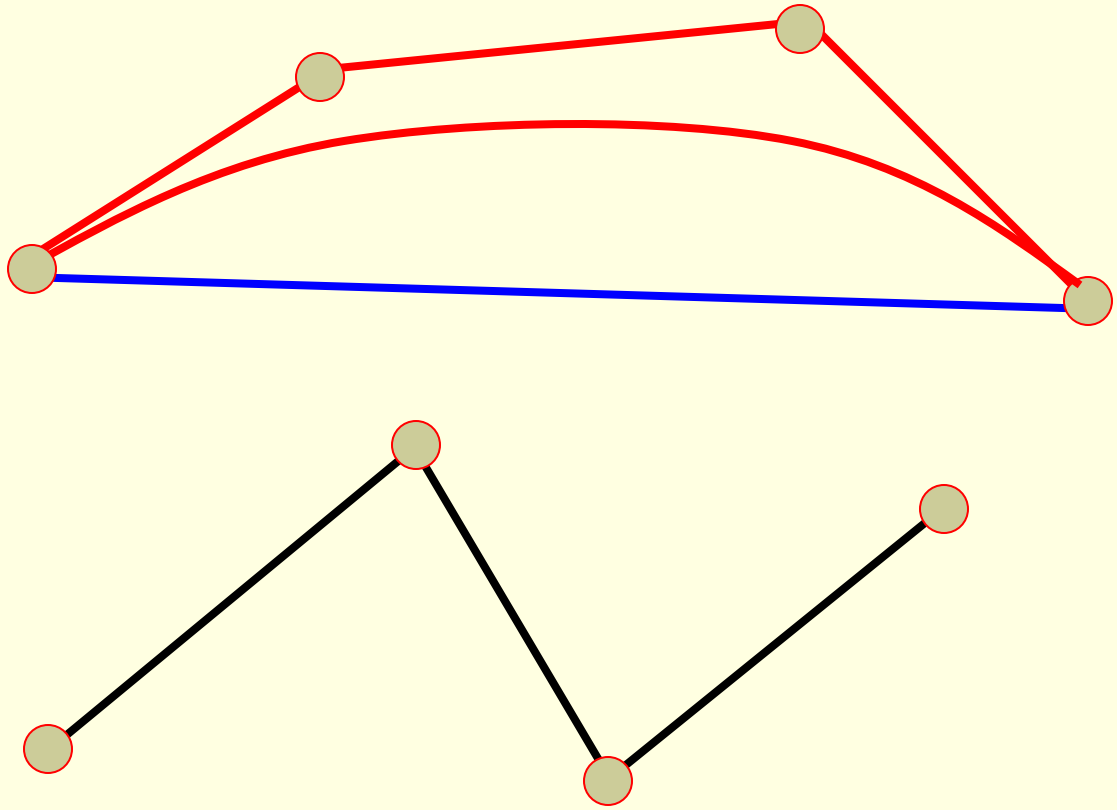
Blending functions always

- Sum to 1 (partition of unity)
- Are non-negative
- Satisfy $1 - B_2 - B_3 - B_4 = B_1$

Curve always

- Passes inside the convex hull of the control points
- Goes through the two endpoints
- Is tangent to the control polygon at the endpoints

Convex Hull Property

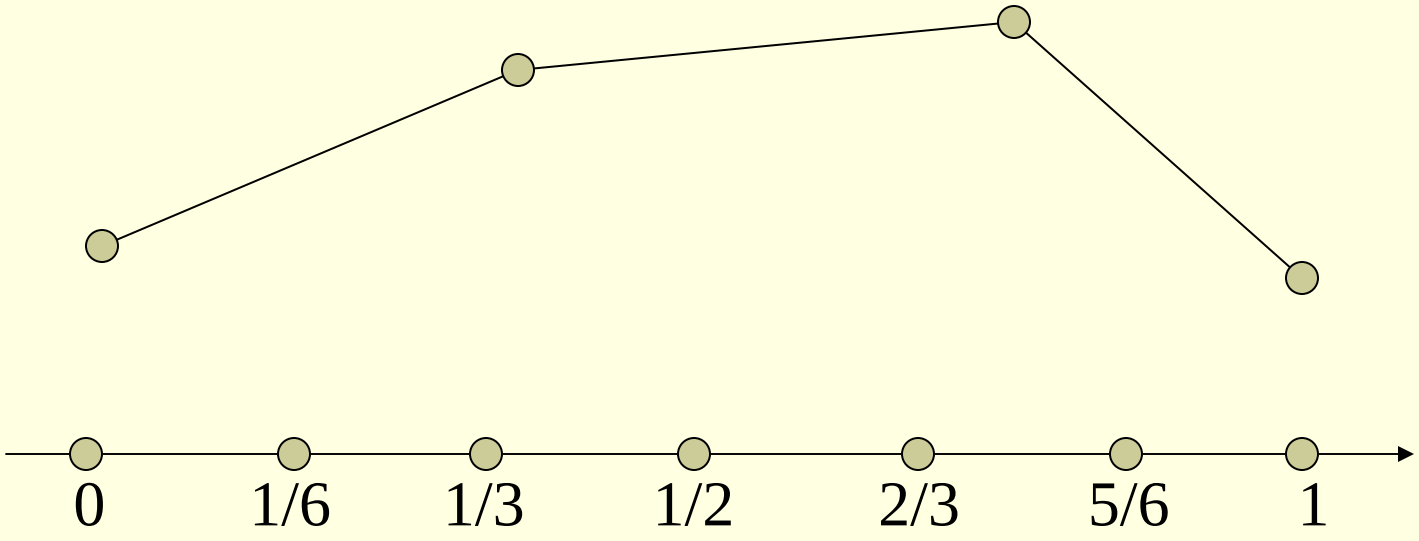


Drawing Spline Curves

$[x(t), y(t)]$ for t between 0 and 1 gives all the points on the curve:

$$x(\tfrac{1}{2}) = (1-\tfrac{1}{2}^3)P_{1x} + 3\tfrac{1}{2}(1-\tfrac{1}{2})^2 P_{2x} + 3\tfrac{1}{2}^2(1-\tfrac{1}{2})P_{3x} + \tfrac{1}{2}^3 P_{4x}$$

$$= \tfrac{7}{8}P_{1x} + \tfrac{3}{8}P_{2x} + \tfrac{3}{8}P_{3x} + \tfrac{1}{8}P_{4x}$$



Displaying Cubic Curves

- Divide interval [0, 1] into *n* even pieces
- Evaluate the curve at each value of t within the interval
- Draw line segments connecting the points

What is the cost of displaying curves this way?

$$x(t) = (1-t^3)P_{1x} + 3t(1-t)^2 P_{2x} + 3t^2(1-t)P_{3x} + t^3 P_{4x}$$

An **incremental** approach is possible…

Forward Differencing

The incremental approach used by the Midpoint algorithm for lines is a form of **forward differencing**

Consider the following linear equation:

$$\Delta_1 \left\langle \begin{array}{c} x(t) = at + b \\ x(t+\varepsilon) = a(t+\varepsilon) + b \end{array} \right.$$

$$\Delta_1 = x(t+\varepsilon) - x(t) = a(t+\varepsilon) + b - (at+b)$$

$$= a\varepsilon$$

Forward Differencing

$$x(t) + \Delta_1 = x(t + \varepsilon)$$

$$\Delta_1 = a\varepsilon$$

In this linear case (polynomial of degree 1), the forward difference applied once gives a constant term which can be used to compute successive values of x(t)

For a **cubic curve**, we have a general form which is a cubic equation, and calculating a single “difference” will *not* reduce the equation immediately to a constant

Forward Differencing

$$x(t) = at^3 + bt^2 + ct + d$$

$$x(t+\varepsilon) = a(t+\varepsilon)^3 + b(t+\varepsilon)^2 + c(t+\varepsilon) + d$$

$$\Delta_1 = x(t+\varepsilon) - x(t)$$

By expanding the difference equation and simplifying we get

$$\Delta_1 = (3a\varepsilon)t^2 + (3a\varepsilon^2 + 2b\varepsilon)t + (a\varepsilon^3 + b\varepsilon^2 + c\varepsilon)$$

Cubic Splines: Forward Differencing

$$\Delta_1 = (3a\varepsilon)t^2 + (3a\varepsilon^2 + 2b\varepsilon)t + (a\varepsilon^3 + b\varepsilon^2 + c\varepsilon)$$

This difference is the **first order** difference, which reduces the degree of the cubic equation by 1, i.e., it is now a quadratic

The second order difference can be computed by applying the same differencing technique to the equation for Δ_1

$$\Delta_1(t) = (3a\varepsilon)t^2 + (3a\varepsilon^2 + 2b\varepsilon)t + (a\varepsilon^3 + b\varepsilon^2 + c\varepsilon)$$

$$\Delta_1(t+\varepsilon) = (3a\varepsilon)(t+\varepsilon)^2 + (3a\varepsilon^2 + 2b\varepsilon)(t+\varepsilon) + (a\varepsilon^3 + b\varepsilon^2 + c\varepsilon)$$

Second-Order Differencing

$$\Delta_2 = \Delta_1(t+\varepsilon) - \Delta_1(t)$$
$$= 6a\varepsilon^2 t + (6a\varepsilon^3 + 2b\varepsilon^2)$$
$$\Delta_1 = (3a\varepsilon)t^2 + (3a\varepsilon^2 + 2b\varepsilon)t + (a\varepsilon^3 + b\varepsilon^2 + c\varepsilon)$$

We wish to evaluate points on the curve for these values of t:

$$\{0, \varepsilon, 2\varepsilon, 3\varepsilon, \ldots, 1\}$$

Using the current differencing scheme (first and second-order differencing), we get the following algorithm for computing points on the curve:

Computing Curve Points

$$\varepsilon = \tfrac{1}{n}; \quad t = 0$$

$$x(t_{=0}) = a_x t^3 + b_x t^2 + c_x t + d_x$$
$$= d_x$$

$$\Delta_1(t_{=0}) = (3a_x\varepsilon)t^2 + (3a_x\varepsilon^2 + 2b_x\varepsilon)t + (a_x\varepsilon^3 + b_x\varepsilon^2 + c_x\varepsilon)$$
$$= (a_x\varepsilon^3 + b_x\varepsilon^2 + c_x\varepsilon)$$

$$\Delta_2(t_{=0}) = 6a_x\varepsilon^2 t + (6a_x\varepsilon^3 + 2b_x\varepsilon^2)$$
$$= (6a_x\varepsilon^3 + 2b_x\varepsilon^2)$$

Computing Curve Points

```
DrawPoint((int)x,(int)y);
        // also must compute
        // the y-coordinate
for (i=0; i < n; i++)
```

$$t = t + \varepsilon$$

$$x = x + \Delta_1$$

$$\Delta_1 = \Delta_1 + \Delta_2$$

$$\Delta_2 = 6a\varepsilon^2 t + (6a\varepsilon^3 + 2b\varepsilon^2)$$

```
        DrawPoint((int)x,(int)y);
```

Third-Order Differencing

$$\Delta_2 = 6a\varepsilon^2 t + (6a\varepsilon^3 + 2b\varepsilon^2)$$

$$\Delta_2(t+\varepsilon) = 6a\varepsilon^2(t+\varepsilon) + (6a\varepsilon^3 + 2b\varepsilon^2)$$

$$\Delta_3 = \Delta_2(t+\varepsilon) - \Delta_2(t)$$

$$= 6a\varepsilon^3$$

Now we have an incremental formulation that uses only simple additions in the loop where curve points are computed:

Algorithm: 3rd Order Differencing

$\varepsilon = \frac{1}{n}; \quad t = 0; \qquad x = d_x; \; y = d_y$

$\Delta_{1x} = (a_x\varepsilon^3 + b_x\varepsilon^2 + c_x\varepsilon) \qquad \Delta_{1y} = (a_y\varepsilon^3 + b_y\varepsilon^2 + c_y\varepsilon)$

$\Delta_{2x} = (6a_x\varepsilon^3 + 2b_x\varepsilon^2) \qquad \Delta_{2y} = (6a_y\varepsilon^3 + 2b_y\varepsilon^2)$

$\Delta_{3x} = 6a_x\varepsilon^3 \qquad \Delta_{3y} = 6a_y\varepsilon^3$

```
DrawPoint((int)x,(int)y);
for (i=0; i < n; i++)
```

$x = x + \Delta_{1x} \qquad y = y + \Delta_{1y}$

$\Delta_{1x} = \Delta_{1x} + \Delta_{2x} \qquad \Delta_{1y} = \Delta_{1y} + \Delta_{2y}$

$\Delta_{2x} = \Delta_{2x} + \Delta_{3x} \qquad \Delta_{2y} = \Delta_{2y} + \Delta_{3y}$

```
DrawPoint((int)x,(int)y);
```

A Few Things to Consider

We have an incremental, fast algorithm for evaluating the points on a cubic spline, where the curve equation is of the form

$$\alpha(t) = at^3 + bt^2 + ct + d$$

But the Bezier curve as we have developed it is of the form

$$\alpha_B(t) = (1-t^3)\vec{P_1} + 3t(1-t)^2\vec{P_2} + 3t^2(1-t)\vec{P_3} + t^3\vec{P_4}$$

$$x(t) = (1-t^3)P_{1x} + 3t(1-t)^2 P_{2x} + 3t^2(1-t)P_{3x} + t^3 P_{4x}$$

$$y(t) = (1-t^3)P_{1y} + 3t(1-t)^2 P_{2y} + 3t^2(1-t)P_{3y} + t^3 P_{4y}$$

How can we formulate the incremental algorithm for Bezier curves?

Regrouping

We can take the original form of the Bezier curve

$$\alpha_B(t) = (1-t^3)\vec{P_1} + 3t(1-t)^2\vec{P_2} + 3t^2(1-t)\vec{P_3} + t^3\vec{P_4}$$

And regroup terms so that it is rewritten as a grouped cubic:

$$\alpha_B(t) = (-\vec{P_1} + 3\vec{P_2} - 3\vec{P_3} + \vec{P_4})t^3$$
$$+ (3\vec{P_1} - 6\vec{P_2} + 3\vec{P_3})t^2 + (-3\vec{P_1} + 3\vec{P_2})t + \vec{P_1}$$

In this form,

$$a = (-\vec{P_1} + 3\vec{P_2} - 3\vec{P_3} + \vec{P_4})$$
$$b = (3\vec{P_1} - 6\vec{P_2} + 3\vec{P_3})$$
$$c = (-3\vec{P_1} + 3\vec{P_2})$$
$$d = \vec{P_1}$$

Incremental Bezier

Using the mapping of the control points of the Bezier to the coefficients of the incremental cubic, we obtain the following equations:

$$a = (-\vec{P}_1 + 3\vec{P}_2 - 3\vec{P}_3 + \vec{P}_4)$$

$$b = (3\vec{P}_1 - 6\vec{P}_2 + 3\vec{P}_3)$$

$$c = (-3\vec{P}_1 + 3\vec{P}_2)$$

$$d = \vec{P}_1$$

$$x = a = \vec{P}_{1x} \qquad \Delta_{1x} = (a_x\varepsilon^3 + b_x\varepsilon^2 + c_x\varepsilon)$$

$$\Delta_{1x} = (-\vec{P}_{1x} + 3\vec{P}_{2x} - 3\vec{P}_{3x} + \vec{P}_{4x})\varepsilon^3 +$$

$$(3\vec{P}_{1x} - 6\vec{P}_{2x} + 3\vec{P}_{3x})\varepsilon^2 + (-3\vec{P}_{1x} + 3\vec{P}_{2x})\varepsilon$$

The other equations can be similarly converted

Java Code for Drawing a Bezier Curve

```
import java.awt.*;
import java.applet.Applet;

public class Bezier {
   private int xpoints[], ypoints[];
   private int limit=4;
   private int count;
   private int width=10;
   private int height=10;
   private int intervals=5;

   public Bezier ()
   {
      xpoints = new int[limit];
      ypoints = new int[limit];
      count = 0;
   }
```

Drawing the Curve

```
public void paintCurve ( Graphics g ) {
    double x, y, xold, yold, A, B, C;
    double t1, t2, t3;
    double deltax1, deltax2, deltax3;
    double deltay1, deltay2, deltay3;
    if (count != limit)
       return;

    // Initialize values for fast Bezier
    t1 = 1.0/intervals;
    t2 = t1*t1;
    t3 = t2*t1;
    x = xpoints[0];
    y = ypoints[0];
    xold = x;
    yold = y;
```

Drawing the Curve: Initializations

```
// set up deltas for the x-coords
A = (-xpoints[0] + 3*xpoints[1] -
    3*xpoints[2] + xpoints[3]);
B = (3*xpoints[0] - 6*xpoints[1] +
    3*xpoints[2]);
C = (-3*xpoints[0] + 3*xpoints[1]);
deltax1 = A*t3 + B*t2 + C*t1;
deltax2 = 6*A*t3 + 2*B*t2;
deltax3 = 6*A*t3;
```

Drawing the Curve: Initializations

```
// set up deltas for the y-coords
A = (-ypoints[0] + 3*ypoints[1] -
    3*ypoints[2] + ypoints[3]);
B = (3*ypoints[0] - 6*ypoints[1] +
    3*ypoints[2]);
C = (-3*ypoints[0] + 3*ypoints[1]);
deltay1 = A*t3 + B*t2 + C*t1;
deltay2 = 6*A*t3 + 2*B*t2;
deltay3 = 6*A*t3;
```

Drawing the Curve: Forward Differencing

```
  for (int i = 0; i < intervals; i++) {
      x += deltax1;
      deltax1 += deltax2;
      deltax2 += deltax3;

      y += deltay1;
      deltay1 += deltay2;
      deltay2 += deltay3;

      g.drawLine ((int)xold, (int)yold,
                  (int)x, (int)y);
      xold = x;
      yold = y;
    }  // end of for loop
} // end of paintCurve()
```

Managing the Control Points

```
public void drawPolyline ( Graphics g )
{
   for (int i = 0; i < (count-1); i++) {
       g.fillOval (xpoints[i]-(width/2),
              ypoints[i]-(height/2),
              width, height);
       g.drawLine (xpoints[i], ypoints[i],
              xpoints[i+1], ypoints[i+1]);
   }

   g.fillOval (xpoints[count-1]-(width/2),
              ypoints[count-1]-(height/2),
              width, height);
}
```

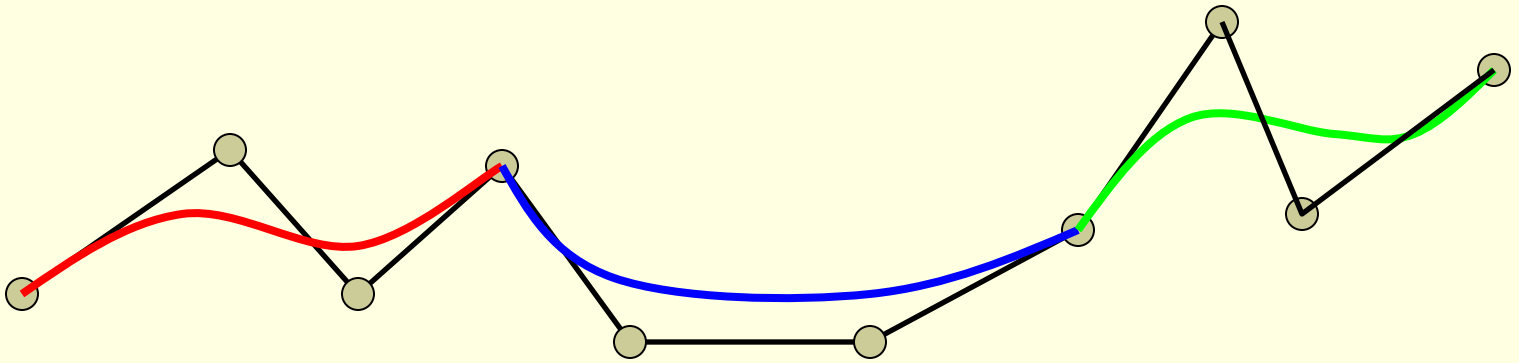
Managing the Control Points

```
  public void addPoint (int x, int y)
   {
      if (count >= limit)
         return;
      xpoints[count] = x;
      ypoints[count] = y;
      count++;
   }
  public void resetCurve ()
     { count = 0;}

  public void setInterval (int i)
     { intervals = i;}
}
```

Multiple Connected Curve Segments

One long curve can be formed from multiple connected curve segments



Principal of locality: change in the position of a single control point will affect at most 2 curve segments. This is important for efficient interactive manipulation of the curves

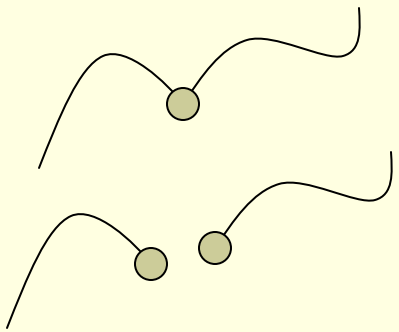
Smoothness at Join Points

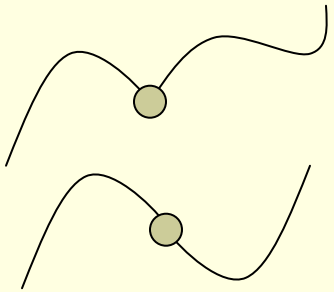
G^0 geometric continuity: the endpoints coincide

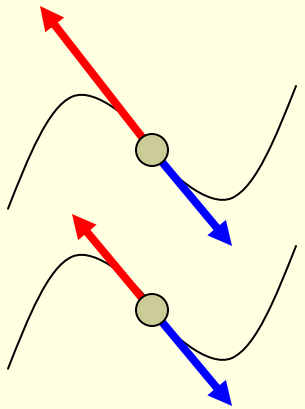
G^1 tangents have the same slope

C^1 first derivative on both curves match at join point

C^n nth derivative on both curves match at join point







Smoother Joins

Bezier splines can be joined to form chains, but the join points guarantee only geometric (***G***) continuity, not derivative (***C***) continuity. Thus they are not as smooth in the derivative sense as we might like.

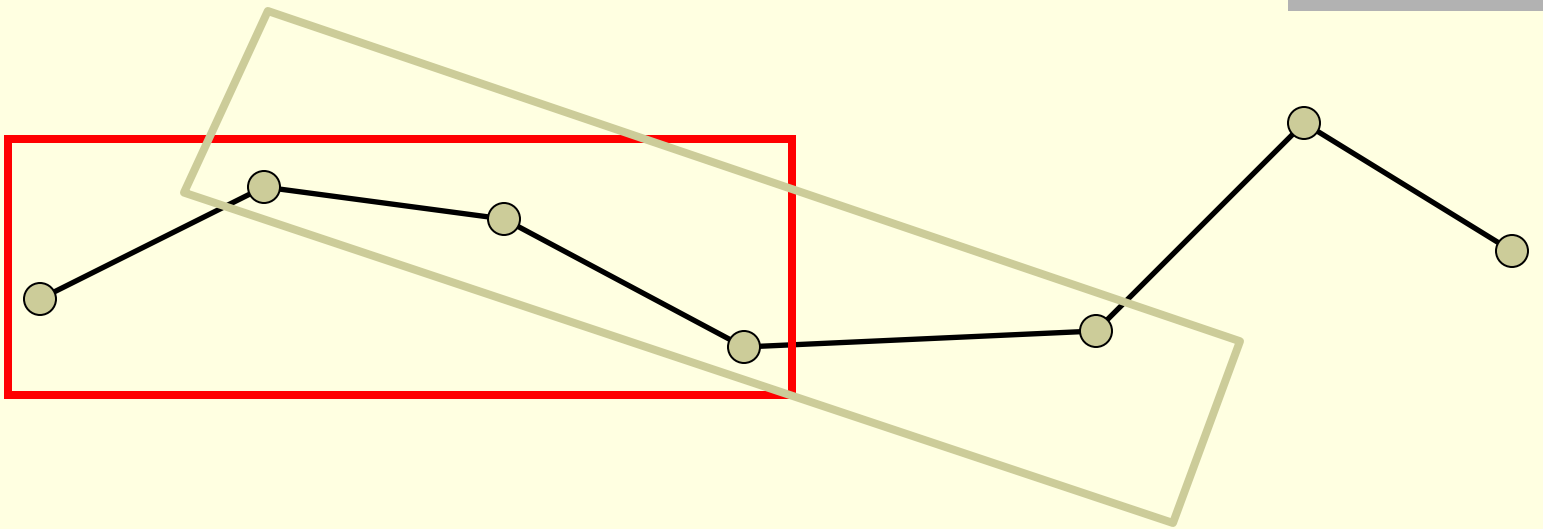
Other Choices for Curves:

- Natural cubic splines
- Hermite curves
- Non-rational parametric cubic B-splines

B-Splines

- B-spline: Basis-spline - curve is represented with basis functions
- Uniform: curve joins are equally spaced in parameter space
- Non-rational: basis functions are not constrained to be rational
- Parametric cubic: basis functions are cubic functions of a parameter

B-Splines



Curve segments overlap by sharing 3 control points (out of the four that are required to define a curve segment). This guarantees very smooth join points, but complicates the display process slightly

Principle of Locality: how many curve segments are affected (at most) when a control point is moved?

B-Splines: Single Curve Segment

$$\alpha_{B-spline}(t) = \mathbf{T}\mathbf{M}_{B-spline}\mathbf{G}$$

$$= \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} \overrightarrow{P}_{1x} & \overrightarrow{P}_{1y} \\ \overrightarrow{P}_{2x} & \overrightarrow{P}_{2y} \\ \overrightarrow{P}_{3x} & \overrightarrow{P}_{3y} \\ \overrightarrow{P}_{4x} & \overrightarrow{P}_{4y} \end{bmatrix}$$

$$\alpha_{B-spline}(t) = (1-t)^3\overrightarrow{P}_1 + (3t^3 - 6t^2 + 4)\overrightarrow{P}_2 +$$

$$(-3t^3 + 3t^2 + 3t + 1)\overrightarrow{P}_3 + t^3\overrightarrow{P}_4$$

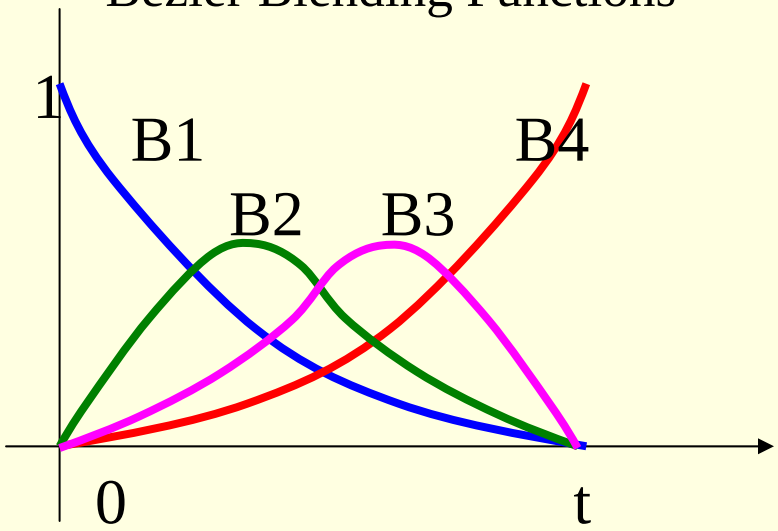
Bezier vs. B-spline Curve Definitions

$$\alpha_{B-spline}(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} \vec{P}_{1x} & \vec{P}_{1y} \\ \vec{P}_{2x} & \vec{P}_{2y} \\ \vec{P}_{3x} & \vec{P}_{3y} \\ \vec{P}_{4x} & \vec{P}_{4y} \end{bmatrix}$$

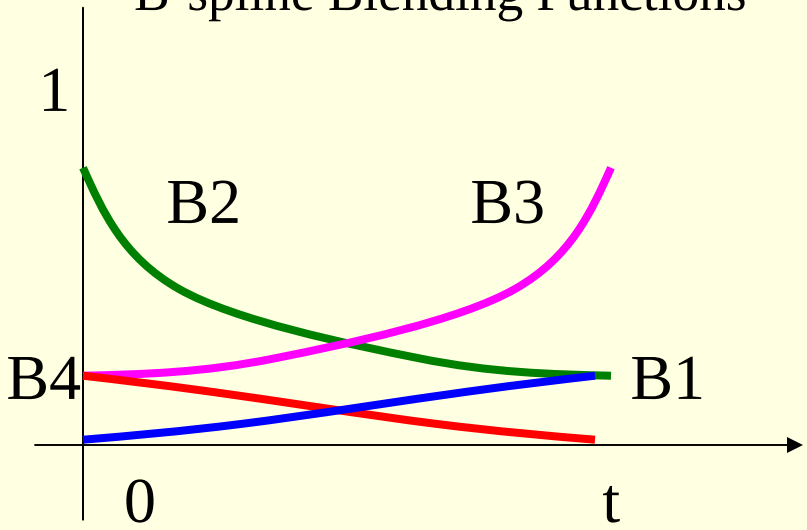
$$\alpha_{Bezier}(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \vec{P}_{1x} & \vec{P}_{1y} \\ \vec{P}_{2x} & \vec{P}_{2y} \\ \vec{P}_{3x} & \vec{P}_{3y} \\ \vec{P}_{4x} & \vec{P}_{4y} \end{bmatrix}$$

Blending Functions

Bezier Blending Functions



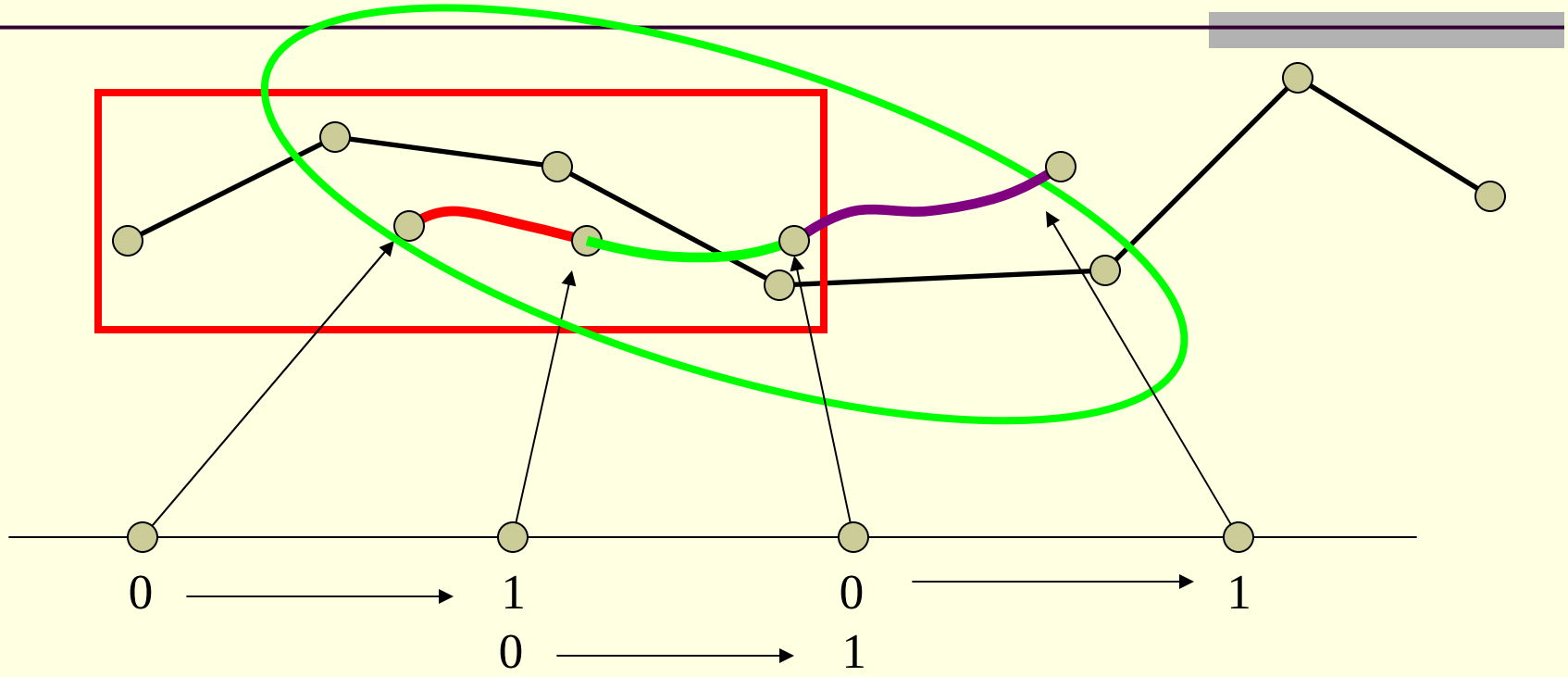
B-spline Blending Functions



$$\left.\begin{aligned} B_1 &\longrightarrow (1-t^3) \\ B_2 &\longrightarrow (1-t)^2 3t \\ B_3 &\longrightarrow (1-t)3t^2 \\ B_4 &\longrightarrow t^3 \end{aligned}\right\}$$

$$\left.\begin{aligned} B_1 &\longrightarrow (1-t^3) \\ B_2 &\longrightarrow 3t^3 - 6t^2 + 4 \\ B_3 &\longrightarrow -3t^3 + 3t^2 + 3t + 1 \\ B_4 &\longrightarrow t^3 \end{aligned}\right\}$$

Control Points and the Parameter



$$Q_0 = P_0, P_1, P_2, P_3$$

$$Q_1 = \quad P_1, P_2, P_3, P_4$$

$$Q_2 = \quad\quad P_2, P_3, P_4, P_5$$

Gains and Losses with B-Splines

Gains:

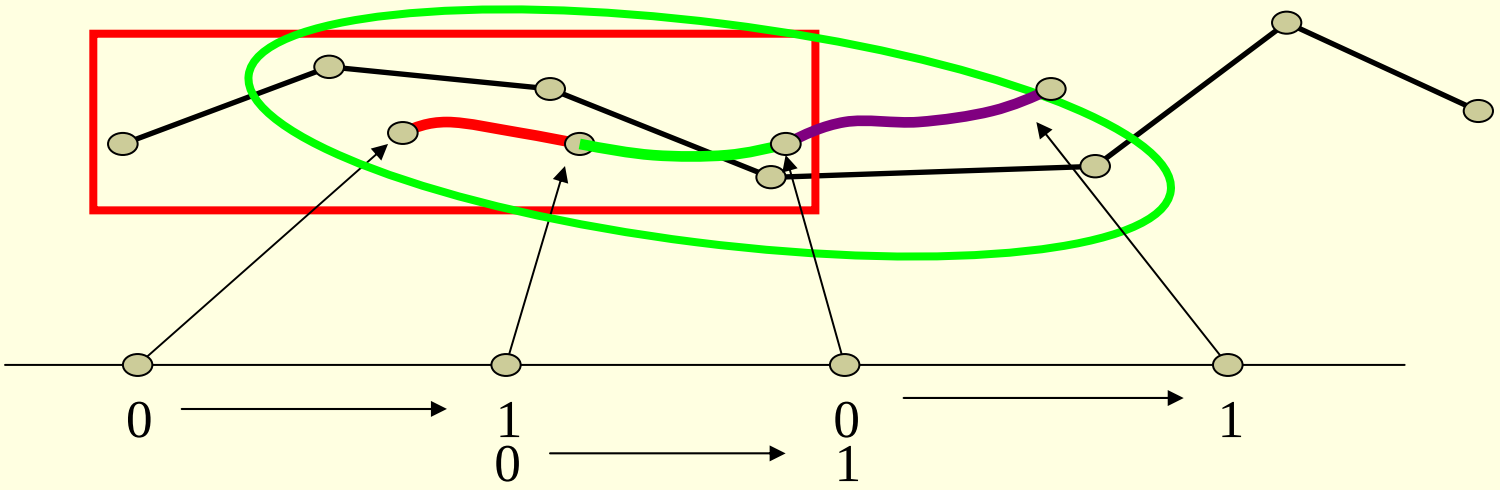
- Locality
- Compact form for multiple segments
- C_2 continuity
- Blending function/convex hull properties

Losses:

- Control over exact position of curve (interpolation vs. approximation)

Additional Topics to Study

- Knot position and control
- Approximation vs. Interpolation
- Inserting/deleting control points
- Other blending functions



Summary

- Continuity of free-form curves (G^0, G^1, C^0, C^1, C^2…)
- Cubic Curves (Splines)
  - Hermite
    - Matrix representation
    - Blending
  - Bezier
  - B-Spline