Modeling Complex Shapes

Bezier Curves

Modeling Complex Shapes

- Can use line/polygon primitives to approximate
- But complex objects-->huge number of primitives
- Better to use more complex primitives
- Use curves (2-D) or surfaces (3-D)

Curves in Space

- Three forms:
  - Explicit
  - Implicit
  - Parametric

Explicit Form (2-D)

- y = f(x)
- example--line:

  y = m*x + b

  But this is not a finite line segment
- Not all curves can be put into this form

Implicit Form (2-D)

- f(x,y)=0
- Example--circle:

  $(x-h)^2 + (y-k)^2 - R^2 = 0$
- Indicates if a point x,y is on the curve
- Can be difficult to plot
  - May need to use approximation methods
    - Marching Squares
- In some cases can be cast into explicit form

Parametric Form

- x and y expressed as explicit functions of a parameter, t

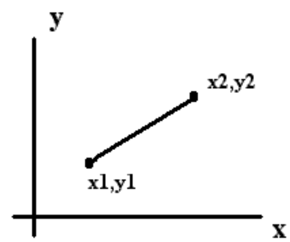
  x = f(t)

  y = g(t)
- Range of parameter is also given
  - Delimits the extent of the curve
- To plot, let t vary over its range
  - Points on curve are generated
- Easily extended to curves in 3-D

  z = h(t)

Parametric Equations for a Line Segment in 2-D

- Given endpoints P1(x1,y1), P2(x2,y2)
- Assume:
  - t=0: endpoint P1
  - t=1: endpoint P2
- Linear equation ==>
  - x = a*t + b
  - y = c*t + d
- Need to get constants a,b,c,d



x = a*t + b, y = c*t + d

- Apply boundary conditions:
  - t=0 ==> x=x1, y=y1
    - x1 = a*0 + b, so b=x1
    - y1 = c*0 + d, so d=y1
  - t=1 ==> x=x2, y=y2
    - x2 = a*1 + b, so a = x2 - b, or a = x2 - x1
    - y2 = c*1 + d, so c = y2 - d, or c = y2 - y1
- Resulting Parametric equations:
  - x = (x2-x1)*t + x1
  - y = (y2-y1)*t + y1

  0<=t<=1
- Easy to extend to 3-D
  - Z = (z2-z1)*t + z1

Polynomials

- Explicit Form of n-degree polynomial:

  $y = a_0 + a_1*x + a_2*x^2 + ... a_n*x^n$
- Assume we have a set of n+1 known control points: (xi,yi)
- Get polynomial coefficients a_i from the control points
- Two Methods:
  - Interpolation
  - Approximation

Interpolating Polynomial, degree n

- Curve passes through all n+1 control points (x_i,y_i)
- Given $(x_0,y_0), (x_1,y_1), (x_2,y_2) ... (x_n,y_n)$:

  $y0 = a_0 + a_1*x_0 + a_2*x_0^2 \ldots a_n*x_0^n$

  $y1 = a_0 + a_1*x_1 + a_2*x_1^2 \ldots a_n*x_1^n$

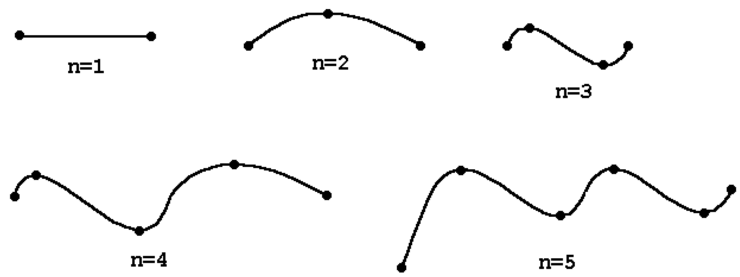
  …

  $yn = a_0 + a_1*x_n + a_2* x_n^2 \ldots a_n* x_n^n$
- n+1 equations in n+1 unknown constants:

  $a_0, a_1, a_2, ... a_n$

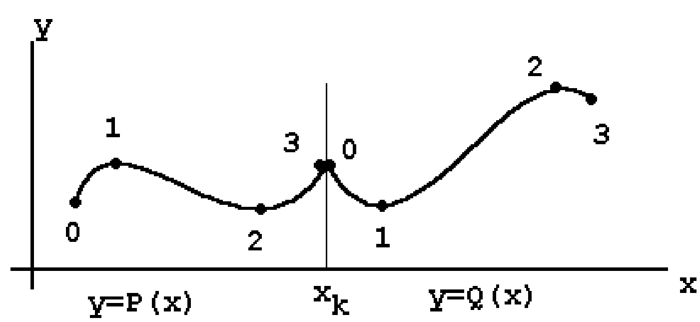
May not be good in graphics

- Many control points==>high degree polynomial
- Many calculations
- Polynomial “wiggle”



Segmented Interpolating Polynomials

- Break curve into segments
- Each with different low-degree polynomial
- Easier computations



Joining Segmented Curves

- Join points called knots
- kth knot at $x=x_k$
- Level-0 continuity: $P(x_k)=Q(x_k)$
  – Continuous, but not smooth (kinks)
- Level-1 continuity: $P'(x_k)=Q'(x_k)$
  – First derivative-->smoother curve
- Level-2 continuity: $P''(x_k)=Q''(x_k)$
  – Second derivative-->still smoother

Approximating Polynomials

- Curve determined by control points
- But does NOT go through all of them
- Control Points act as magnets
- Better for many graphics applications
- Most commonly used:

  Bezier curves

  B-spline curves

Bezier Curves

- Bezier Curves
  – See CS-460/560 Notes:
  – Bezier Polynomial Curves
  http://www.cs.binghamton.edu/~reckert/460/bezier.htm
- B-Spline Curves
  – See CS-460/560 Notes:
  – B-spline Polynomial Curves
  http://www.cs.binghamton.edu/~reckert/460/bspline.htm

Bezier Polynomial Curves

- Parametric equations for a 2-D cubic polynomial curve:

  $x = ax*t^3 + bx*t^2 + cx*t + dx$

  $y = ay*t^3 + by*t^2 + cy*t + dy$

  $0<=t<=1$

- Shape of curve determined by constant polynomial coefficients:
  - (ax,bx,cx,dx, ay,by,cy,dy)

Easily extended to 3-D

- Just add a third parametric equation:

  $z = az*t^3 + bz*t^2 + cz*t + dz$

Control Points

- Want to easily determine shape of curve
- Specify four control points:

  P0 (x0,y0,z0), P1(x1,y1,z1), P2(x2,y2,z2), P3(x3,y3,z3)
- Could use interpolating polynomial
- More useful: approximating polynomial
  - Doesn't interpolate all control points
  - Many ways to do the approximating

Uniform Cubic Bezier Polynomial

- Important kind of approximating polynomial
- Assume a generic parametric cubic polynomial:

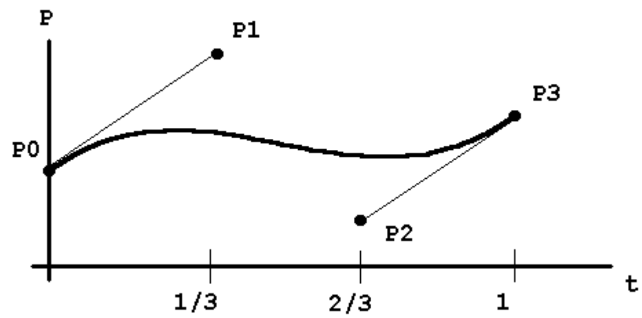
  $P = a*t^3 + b*t^2 + c*t + d, \quad 0 <= t <= 1$
- Determined by control points P0, P1, P2, P3
  - P could be x, y, or z
  - a could be ax, ay, or az
    - same with b, c, d
  - P0 could be x0, y0, z0
    - same with P1, P2, P3

<u>Uniform</u> Bezier Polynomial

$P = a*t^3 + b*t^2 + c*t + d, \quad 0 <= t <= 1$

- Control points uniformly separated in t

  P0 at t=0, P1 at t=1/3, P2 at t=2/3, P3 at t=1



Boundary conditions:

$P = a*t^3 + b*t^2 + c*t + d, \quad 0 <= t <= 1$

1. Curve must interpolate control point P0

   P=P0 when t=0

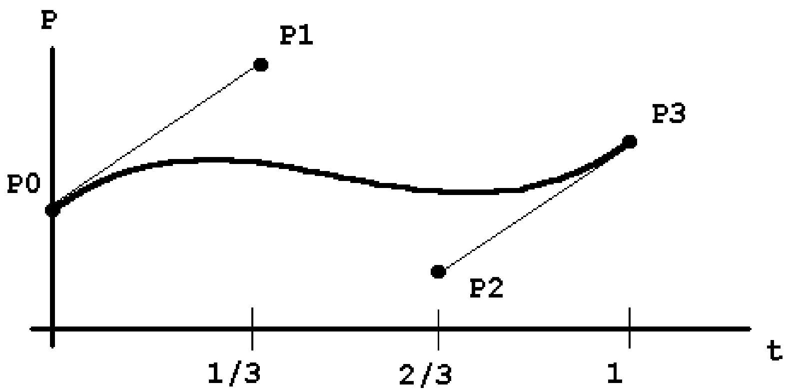
   So P0 = d

2. Curve must interpolate control point P3

   P=P3 when t=1

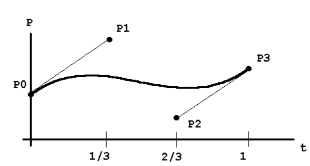
   so P3 = a + b + c + d

Uniform Cubic Bezier Curve



$P = a*t^3 + b*t^2 + c*t + d, \quad 0 <= t <= 1$

3. Slope of curve at t=0 must be equal to that of the line that joins control points P0 and P1
   - dP/dt(at t=0) = slope of P0-P1
   - $dP/dt = 3*a*t^2 + 2*b*t + c$
   - slope of P0-P1 = (P1-P0)/(1/3-0)
   - So: c = 3*(P1-P0)



4. Slope of curve at t=1 must be equal to that of the line that joins control points P2 and P3
   - dP/dt(at t=1) = slope of P2-P3
   - 3*a + 2*b + c = (P3-P2)/(1 - 2/3)
   - 3*a + 2*b + c = 3*(P3-P2)

Solving for Polynomial Coefficients

- Equations:

```
0   + 0   + 0   +   d   = P0
a   + b   + c   +   d   = P3
0   + 0   + c   +   0   = 3*(P1-P0)
3*a + 2*b + c   +   0   = 3*(P3-P2)
```

This can be expressed in matrix form:

```
 _          _      _ _      _          _
| 0  0  0  1 |    | a |    | P0         |
| 1  1  1  1 | *  | b | =  | P3         |
| 0  0  1  0 |    | c |    | 3*(P1-P0)  |
|_3  2  1  0_|    |_d_|    |_3*(P3-P2)_|
```

A * C = V

In other words:

A * C = V

C = [a, b, c, d], the coefficient vector – the unknowns

V = [P0, P3, 3*(P1-P0), 3*(P3-P2)]

A = the above 4X4 matrix

- To solve, multiply by A-inverse

  $A^{-1} * A * C = A^{-1} * V$

  $C = A^{-1} * V$

- Use Gauss-Jordan elimination or other techniques to get A-inverse
- Result:

$$A^{-1} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

So: $C = A^{-1} * V$

```
     _ _      _            _      _        _
    | a |    |  2 -2  1  1 |    | P0        |
C = | b | = | -3  3 -2 -1 | * | P3        |
    | c |    |  0  0  1  0 |    | 3(P1-P0) |
    |_d_|    |_ 1  0  0  0_|    |_3(P3-P2)_|
```

Final Result (after rearranging):

```
 _ _      _                _      _  _
| a |    |-1   3  -3   1 |    | P0 |
| b | =  | 3  -6   3   0 | *  | P1 |
| c |    |-3   3   0   0 |    | P2 |
|_d_|    |_1   0   0   0_|    |_P3_|
```

Uniform Cubic Bezier Result

- Polynomial Coefficients:
  - A constant 4 X 4 matrix multiplied by a vector whose components are the control points
  - Constant matrix called the Bezier geometry matrix
  - Other kinds of polynomial curves will have their polynomial coefficients given by a similar equation
    - Matrix elements of the constant 4 X 4 geometry matrix will change

Writing Bezier Result in Compact Form

- Points P on curve are given by:

  $P = a*t^3 + b*t^2 + c*t + d, \quad 0 <= t <= 1$
- Can be written in a more compact form:

  P = T * Bg * Pc

  T: row vector of parameter powers $[\ t^3 \ \ t^2 \ \ t \ \ 1\ ]$

  Bg: the constant 4 X 4 Bezier Geometry matrix

  Pc: column vector of the control points

Blending Function Representation

- Multiply matrix equation & rearrange:

$$P = (1-t)^3 * P0 + 3t(1-t)^2 * P1 + 3t^2(1-t) * P2 + t^3 * P3$$

$$P = \sum_{i=0}^{3} P_i * B_i(t)$$

- P_i: the control points (P0, P1, P2, P3)
- $B_i(t)$: "Bernstein Blending Functions"

- Blending Function form:
  - A weighted sum of the control points
  - Weighting factors: the Blending Functions
  - Value of Blending function gives "pull" of corresponding control point on curve at any point t
- The blending functions are given by:

  $B_i(t) = C(3,i) * t^i * (1-t)^{(3-i)}$
    - C(3,i) is the number of combinations of 3 things taken i at a time:
    - $C(3,i) = 3! / (i! * (3-i)!)$

The Berstein Blending Functions

- For the cubic Bezier polynomial:

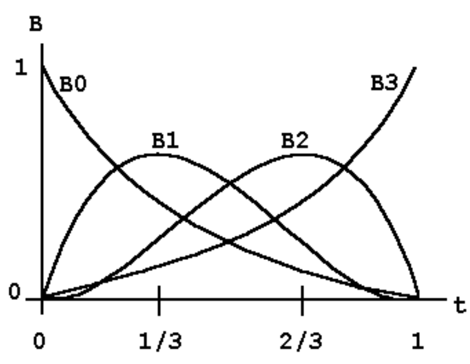
  $B_0(t) = (1-t)^3$

  $B_1(t) = 3 * t * (1-t)^2$

  $B_2(t) = 3 * t^2 * (1-t)$

  $B_3(t) = t^3$

The Berstein Blending Functions



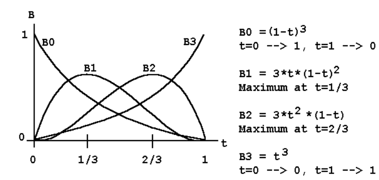
$B0 = (1-t)^3$
t=0 --> 1, t=1 --> 0

$B1 = 3*t*(1-t)^2$
Maximum at t=1/3

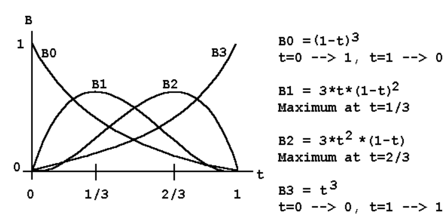
$B2 = 3*t^2*(1-t)$
Maximum at t=2/3

$B3 = t^3$
t=0 --> 0, t=1 --> 1

- B0 has maximum value of 1 (100%) at t=0
  - All other blending functions give 0 there
    - Control point P0 pulls with 100% "force" at t=0
    - None of the other control points pulls at all
    - So curve must go through P0 (as we know)
- B3 has maximum value of 1 (100%) at t=1
  - All other blending functions give 0 there
  - So curve must go through P3

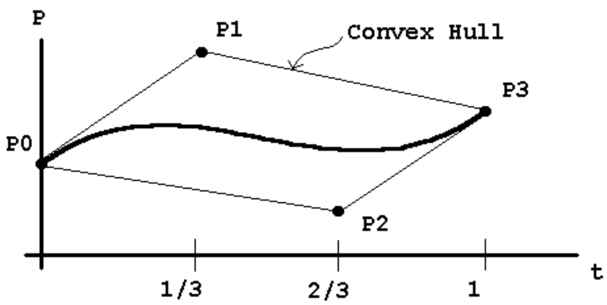


- B1 has its maximum value at t=1/3
  - Value is less than 1 (<100% pull)
  - Other Blending functions are non-zero but with values < B1
  - So curve cannot pass through P1
  - Curve pulled hardest by P1
- Similarly, at t=2/3, P2 pulls hardest



Properties of Bezier Curves

- Bk <=1, so:
  - Control points lie outside curve
    - curve lies inside "Convex Hull" of control points
  - Important for clipping



More Bezier Curve Properties

- "Pull" of a control point is proportional to "distance" (in t) from the control point
- Bezier Curves are invariant under affine transformations
  - So to transform a Bezier curve, just transform the control points and redraw the curve

Plotting Bezier Curves

- Brute Force Method:
  1. Get control points P0=(x0,y0), P1=(x1,y1), P2=(x2,y2), P3=(x3,y3).
     - Could use interactive locator device (mouse)
  2. Compute values of a, b, c, d from control points
     - Really ax,bx,cx,dx and ay,by,cy,dy
     - Use matrix equations
     - (Alternative: use blending functions)

```
3. for (t=0; t<=1; t+=delta)
       Compute P (x & y) from polynomial equations
       if (t==0)
         MoveTo(x,y)
       else
         LineTo(x,y)
```

- delta: a small increment (e.g. 0.05)
- Would give an approximation to the curve consisting of straight-line segments

Improving Performance

- Brute force is much too much work (too slow)

  P = a*t^3 + b*t^2 + c*t + d

  – Each iteration: 5 floating point multiplies

    c*t, t*t, b*(t*t), t*(t*t), a*(t*(t*t))

  – and 3 floating point adds

- Using Horner's rule for polynomial evaluation:

  P = ((a*t+b)*t+c)*t

  – 3 multiplies and 3 adds

- Can do much better
  - Use technique of Forward Differences
  - Will improve performance
    - only 3 floating point adds during each iteration!

Forward Differences

- Get new x,y values from old while stepping

  $x_{i+1} = x_i + \Delta x$

- Look at x equation:

  $x = at^3 + bt^2 + ct + d$

- Assume equal increments in t, $\delta t = \delta$

  $t_{i+1} = t_i + \delta, \quad \Delta x = x_{i+1} - x_i$

  $\Delta x = a(t+\delta)^3 + b(t+\delta)^2 + c(t+\delta) + d \;-\; (at^3 + bt^2 + ct + d)$

- Result (first forward difference):

  $\Delta x = 3a\delta t^2 + (3a\delta^2 + 2b\delta)t + a\delta^3 + b\delta^2 + c\delta$

  Reduced to quadratic in t

- Do again to simplify Δx

  $\Delta x = \Delta x + \Delta(\Delta x) = \Delta x + \Delta^2 x$

  $\Delta^2 x = \Delta x(t+\delta) - \Delta x(t)$

  $\Delta^2 x = 3a\delta(t+\delta)^2 + (3a\delta^2 + 2b\delta)(t+\delta) + k$

  $\quad -3a\delta t^2 - (3a\delta^2 + 2b\delta)t - k$

  where $k = a\delta^3 + b\delta^2 + c\delta$

- Result (second forward difference):

  $\Delta^2 x = 6a\delta^2 t + 6a\delta^3 + 2b\delta^2$

  Reduced to linear equation in t

  For next step let $k1 = 6a\delta^3 + 2b\delta^2$

- Do again to simplify $\Delta^2 x$

  $\Delta^2 x = \Delta^2 x + \Delta(\Delta^2 x) = \Delta^2 x + \Delta^3 x$

  $\Delta^3 x = \Delta^2 x(t+\delta) - \Delta^2 x(t)$

  $\Delta^3 x = 6a\delta^2(t+\delta) + k1 - 6a\delta^2 t - k1$

- Result (third forward difference):

  $\Delta^3 x = 6a\delta^3$

  Finally a constant result

- Final Results (recurrence relations):

  $x = x + \Delta x$

  $\Delta x = \Delta x + \Delta^2 x$

  $\Delta^2 x = \Delta^2 x + \Delta^3 x$

  Three adds on each iteration

Initial Values

- Need to calculate only once

$x0 = a*t0^3 + b*t0^2 + c*t0 + d$

$\Delta x0 = 3a\delta * t0^2 + (3a\delta^2 + 2b\delta)*t0 + a\delta^3 + b\delta^2 + c\delta$

$\Delta^2 x0 = 6a\delta^{2*}t0 + 6a\delta^3 + 2b\delta^2$

$\Delta^3 x0 = 6a\delta^3$

Example Using Forward Difference Calculations

$x = t^3 + 2t^2 + 3t + 1, \ 0<=t<=10$

To illustrate, take $\delta = 1$

And a=1, b=2, c=3, d=1

t0=0

x0 = 1

$\Delta x0 = 1 + 2 + 3 = 6$

$\Delta^2 x0 = 6 + 2*2 = 10$

$\Delta^3 x0 = 6$

$x0 = a*t0^3 + b*t0^2 + c*t0 + d$

$\Delta x0 = 3a\delta * t0^2 + (3a\delta^2 + 2b\delta)*t0 + a\delta^3 + b\delta^2 + c\delta$

$\Delta^2 x0 = 6a\delta^2 * t0 + 6a\delta^3 + 2b\delta^2$

$\Delta^3 x0 = 6a\delta^3$

Example Forward Difference Calculations

t	0	1	2	3	4	5
x	1	7	23	55	109	191
Δx	6	16	32	54	82	116
$\Delta^2 x$	10	16	22	28	34	40
$\Delta^3 x$	6	6	6	6	6	6

Higher Degree Bezier Curves

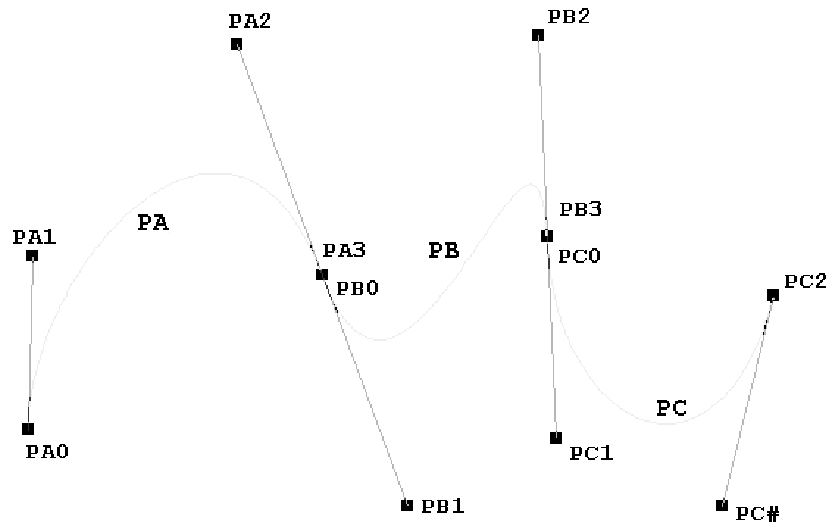
- Cubic (n=3) ==> 4 control points
- 4th degree ==> 5 control points
- nth degree ==> n+1 control points
- In general:

$$P(t) = \sum_{i=0}^{n} B^{n}_{i}(t) * P_i$$

$$B^{n}_{i}(t) = C(n,i) * t^{i} * (1-t)^{n-i}$$

- Higher Degree Bezier curves:
  - Can represent complex shapes
  - But moving any control point affects <u>entire</u> curve
  - Want <u>local control</u>
    - Moving a control point affects only one section of the curve
  - One way: use segmented Bezier curves

A Segmented Cubic Bezier Curve



Conditions at Knots

- Curves PA and PB
  - Determined by Control Points
    - PA0, PA1, PA2, PA3; PB0, PB1, PB2, PB3
- Level-0 continuity at knot:

  PA(at t=1) = PB(at t=0), i.e. at knot

  So PA3 = PB0 (Same control point)
- Level-1 continuity:

  dPA/dt(at t=1) = dPB/dt(at t=0)
  - So segments PA2-PA3 and PB0-PB1 must be colinear
  - (Recall Bezier Boundary Conditions)