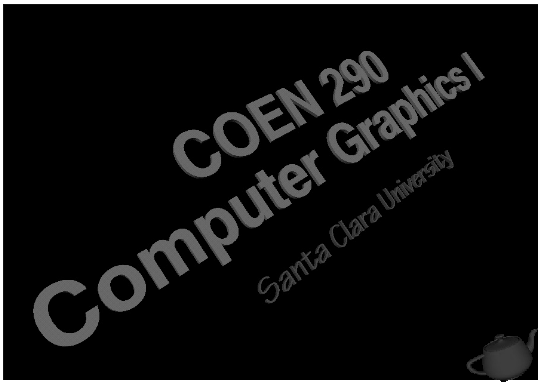
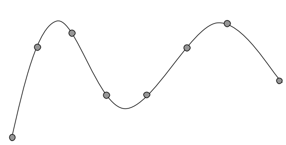


Evening's Goals

- Discuss types of algebraic curves and surfaces
- Develop an understanding of curve basis and blending functions
- Introduce Non-Uniform Rational B-Splines
  - also known as *NURBS*

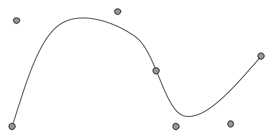
Problem #1

- We want to create a curve (surface) which passes through as set of data points



Problem #2

- We want to represent a curve (surface) for modeling objects
  - compact form
  - easy to share and manipulate
  - doesn't necessarily pass through points



Types of Algebraic Curves (Surfaces)

- *Interpolating*
  - curve passes through points
  - useful of scientific visualization and data analysis
- *Approximating*
  - curve's shape controlled by points
  - useful for geometric modeling

Representing Curves

- We'll generally use *parametric* cubic polynomials
  - easy to work with
  - nice continuity properties

$$p(u) = c_0 + c_1 u + c_2 u^2 + c_3 u^3$$
$$= \sum_{i=0}^{3} c_i u^i$$

Parametric Equations

- Compute values based on a *parameter*
  - parameter generally defined over a limited space
  - for our examples, let $u \in [0,1]$
- For example

$$\vec{f}(u) = \left(\cos(2\pi u) \quad \sin(2\pi u)\right)$$

What's Required for Determining a Curve

- Enough data points to determine coefficients
  - four points required for a cubic curve
- For a smooth curve, want to match
  - continuity
  - derivatives
- Good news is that this has all been figured out

Geometry Matrices

- We can identify curve types by their *geometry matrix*
  - another 4x4 matrix
- Used to define the polynomial coefficients for a curves *blending functions*

Interpolating Curves

- We can use the *interpolating geometry matrix* to determine coefficients
- Given four points in n-dimensional space $\vec{p}_i$, compute $\vec{c}_i$

$$\begin{pmatrix} \vec{c}_0 \\ \vec{c}_1 \\ \vec{c}_2 \\ \vec{c}_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{pmatrix} \begin{pmatrix} \vec{p}_0 \\ \vec{p}_1 \\ \vec{p}_2 \\ \vec{p}_3 \end{pmatrix}$$

Recasting the Matrix Form as Polynomials

- We can rewrite things a little

$$p(u) = \vec{u} \bullet \vec{c} \qquad \vec{u} = \begin{pmatrix} 1 & u & u^2 & u^3 \end{pmatrix}$$

$$\vec{c} = M_g \vec{p}$$

$$p(u) = \vec{u} \bullet \left(M_g \vec{p} \right)$$

$$= \vec{b}(u) \bullet \vec{p}$$

Blending Functions

- Computing static coefficients is alright, but we'd like a more general approach
- Recast the problem to finding simple polynomials which can be used as coefficients

$$p(u) = \sum_{i=0}^{3} b_i(u) p_i$$

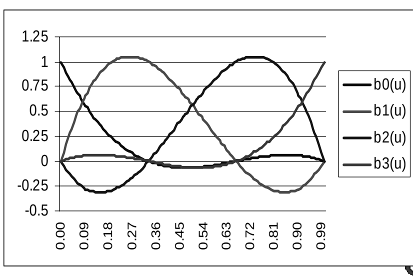
Blending Functions (cont.)

- Here are the blending functions for the interpolation curve

$$b_0(u) = 1 - 5.5u + 9u^2 - 4.5u^3$$
$$b_1(u) = 9u - 22.5u^2 + 13.5u^3$$
$$b_2(u) = -4.5u + 18u^2 - 13.5u^3$$
$$b_3(u) = 1u - 4.5u^2 + 4.5u^3$$

Blending Functions (cont.)

Blending Functions for the Interpolation case



That's nice … but

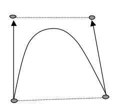
- Interpolating curves have their problems
  - need both data values and coefficients to share
  - hard to control curvature (derivatives)
- Like a less data dependent solution

Approximating Curves

- Control the shape of the curve by positioning *control points*
- Multiples types available
  - Bezier
  - B-Splines
  - NURBS (Non-Uniform Rational B-Splines)
- Also available for surfaces

Bezier Curves

- Developed by a car designer at Renault
- Advantages
  - curve contained to *convex hull*
  - easy to manipulate
  - easy to render



- Disadvantages
  - bad continuity at endpoints
    - tough to join multiple Bezier splines

Bezier Curves (cont.)

- Bezier geometry matrix

$$M_g = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix}$$

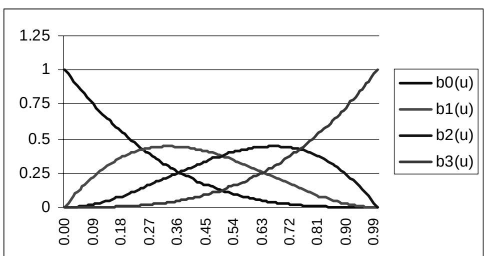
Bernstein Polynomials

- Bezier blending functions are a special set of called the *Bernstein Polynomials*
  - basis of OpenGL's curve evaluators

$$b_k^n(u) = \binom{n}{k} u^k (1-u)^{n-k}$$

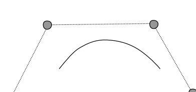
$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Bezier Blending Functions



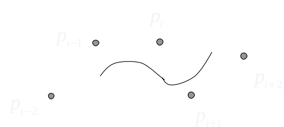
Cubic B-Splines

- B-Splines provide the one thing which Bezier curves don't -- continuity of derivatives
- However, they're more difficult to model with
  - curve doesn't intersect any of the control vertices



Curve Continuity

- Like all the splines we've seen, the curve is only defined over four control points
- How do we match the curve's derivatives?



COEN 290 - Computer Graphics I 22

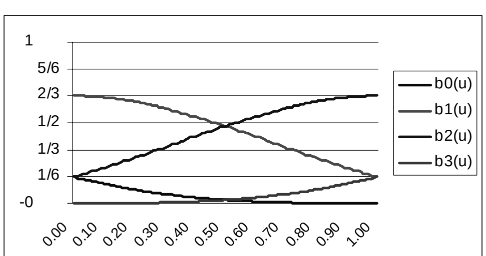
Cubic B-Splines (cont.)

- B-Spline Geometry Matrix

$$M_g = \frac{1}{6}\begin{pmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix}$$

COEN 290 - Computer Graphics I 23

B-Spline Blending Functions



COEN 290 - Computer Graphics I 24

B-Spline Blending Functions (cont.)

- Notice something about the blending functions

$$\sum_{i=0}^{3} b_i(u) = 1.0$$

- Additionally, note that

$$b_0(0) = b_1(1)$$
$$b_1(0) = b_2(1)$$
$$b_2(0) = b_3(1)`$$

Basis Functions

- The blending functions for B-splines form a *basis* set.
  - The “B” in B-spline is for “basis”
- The basis functions for B-splines comes from the following recursion relation

$$B_k^d(u) = \frac{u-u_k}{u_{k+d-1}-u_k} B_k^{d-1}(u) + \frac{u_{k+d}-u}{u_{k+d}-u_{k+1}} B_{k+1}^{d-1}(u)$$

$$B_k^0(u) = \begin{cases} 1 & u_k \le u \le u_{k+1} \\ 0 & \text{otherwise} \end{cases}$$

Rendering Curves - Method 1

- Evaluate the polynomial explicitly

$$p(u) = c_0 + c_1 u + c_2 u^2 + c_3 u^3$$

```
float px( float u ) {
  float v = c[0].x;
  v += c[1].x * u;
  v += c[2].x * pow( u, 2 );
  v += c[3].x * pow( u, 3 );
  return v;
}
```

Evaluating Polynomials

- That's about the worst way you can compute a polynomial
  - very inefficient
    - `pow( u, n );`
- This is better, but still not great

```
float px( float u ) {
  float v = c[0].x;
  v += c[1].x * u;
  v += c[2].x * u*u;
  v += c[3].x * u*u*u;
  return v;
}
```


Horner's Method

- We do a lot more work than necessary
  - computing u^2 and u^3 is wasteful

$$
\begin{aligned}
u^2 &= u \cdot u \\
u^3 &= u \cdot u \cdot u \\
&= u \cdot (u \cdot u)
\end{aligned}
$$

Horner's Method (cont.)

- Rewrite the polynomial

$$
\begin{aligned}
p(u) &= c_0 + c_1 u + c_2 u^2 + c_3 u^3 \\
&= c_0 + u(c_1 + u(c_2 + c_3 u))
\end{aligned}
$$

```
float px( float u ) {
  return c[0].x +
    u*(c[1].x +
    u*(c[2].x +
    u*c[3].x));
}
```

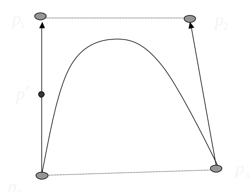
Rendering Curves - Method 1 (cont.)

```
glBegin( GL_LINE_STRIP );
for ( u = 0; u <= 1; u += du )
   glVertex2f( px(u), py(u) );
glEnd();
```

- Even with Horner’s Method, this isn’t the best way to render
  - equal sized steps in parameter doesn’t necessarily mean equal sized steps in world space

Rendering Curves - Method 2

- Use subdivision and recursion to render curves better
- Bezier curves subdivide easily



$$p' = \tfrac{1}{2}(p_0 + p_1)$$

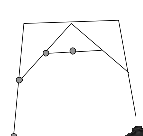
Rendering Curves - Method 2

- Three subdivision steps required

$$p_0' = \tfrac{1}{2}(p_0 + p_1) \quad p_1' = \tfrac{1}{2}(p_1 + p_2) \quad p_2' = \tfrac{1}{2}(p_2 + p_3)$$

$$p_0'' = \tfrac{1}{2}(p_0' + p_1') \quad p_1'' = \tfrac{1}{2}(p_1' + p_2')$$

$$p_0''' = \tfrac{1}{2}(p_0'' + p_1'') = \tfrac{1}{8}(p_0 + 3p_1 + 3p_2 + p_3)$$



Rendering Curves - Method 2

```
void drawCurve( Point p[] ) {
   if ( length( p ) < MAX_LENGTH )
      draw( p );
   Point p01  = 0.5*( p[0] + p[1] );
   Point p12  = 0.5*( p[1] + p[2] );
   Point p23  = 0.5*( p[2] + p[3] );
   Point p012 = 0.5*( p01 + p12 );
   Point p123 = 0.5*( p12 + p13 );
   Point m    = 0.5*( p012 + p123 );
   drawCurve( p[0], p01, p012, m );
   drawCurve( m, p123, p23, p[3] );
}
```