Fall 2014

CSCI 420: **Computer Graphics**

4.2 Splines

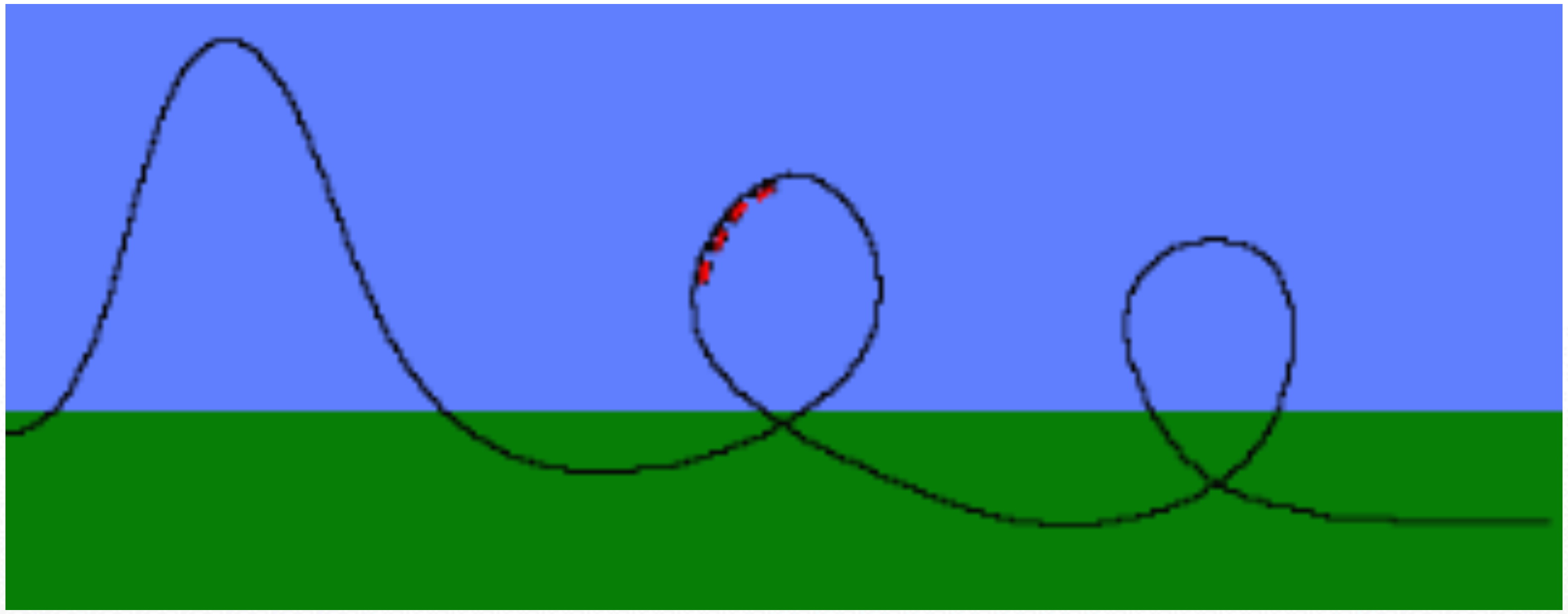
Hao Li

http://cs420.hao-li.com

Roller coaster

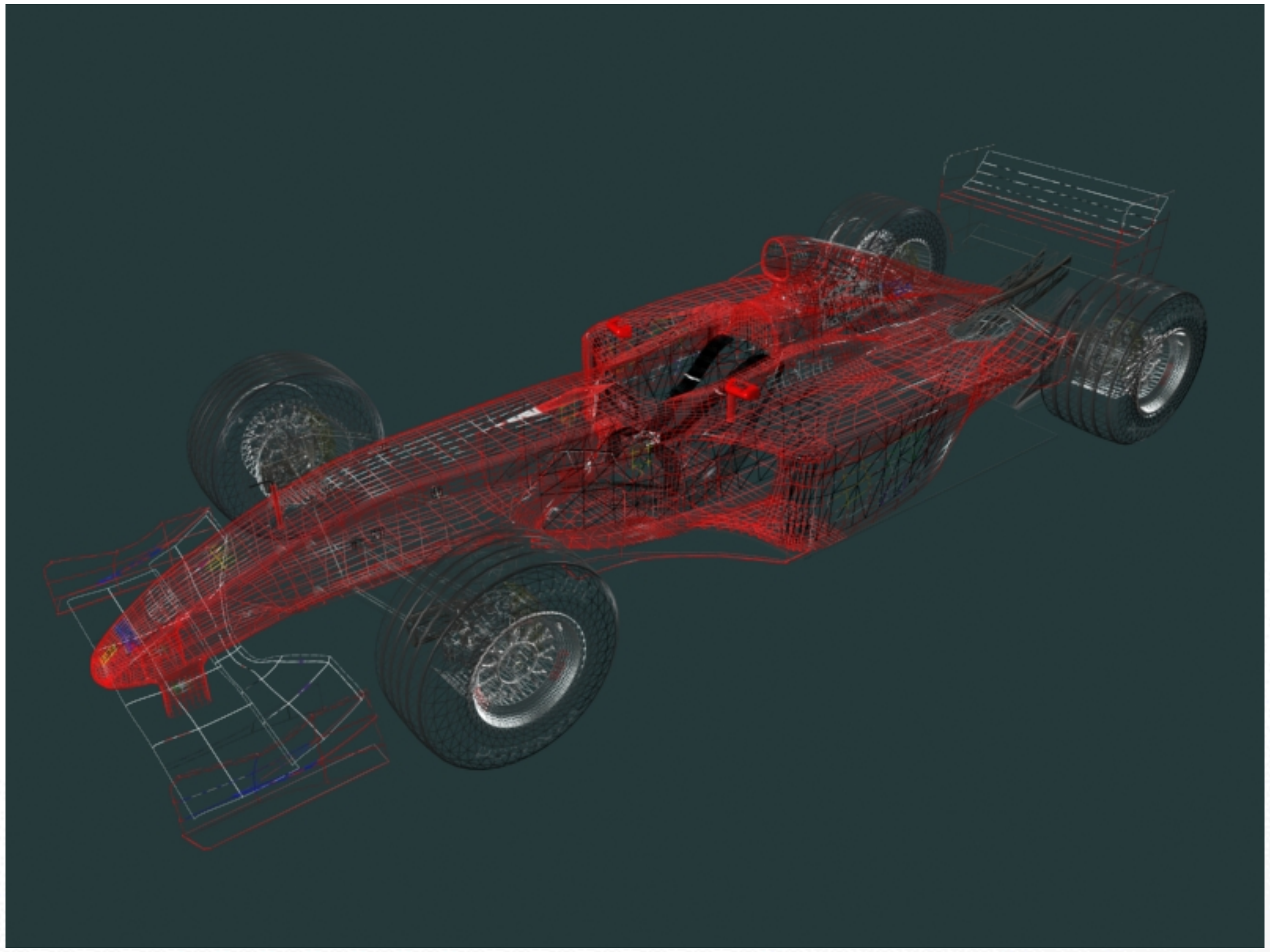
- Next programming assignment involves creating a 3D roller coaster animation

- We must model the 3D curve describing the roller coaster, but how?



Modeling Complex Shapes

- We want to build models of very complicated objects
- Complexity is achieved using simple pieces
  - polygons,
  - parametric curves and surfaces, or
  - implicit curves and surfaces



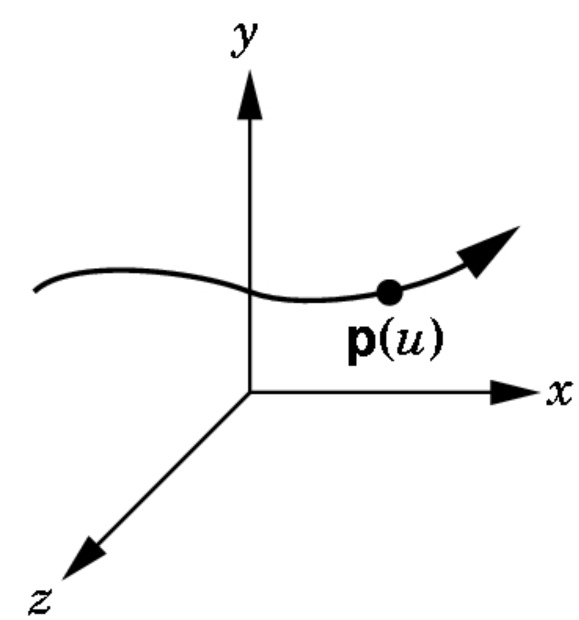
- This lecture: parametric curves

What Do We Need From Curves in Computer Graphics?

- Local control of shape
  (so that easy to build and modify)
- Stability
- Smoothness and continuity
- Ability to evaluate derivatives
- Ease of rendering

Curve Representations

- Explicit: $y = f(x)$
  - Must be a function (single-valued)
  - Big limitation—vertical lines?
- Parametric: $(x, y) = (f(u), g(u))$
  - Easy to specify, modify, control
  - Extra “hidden” variable u, the *parameter*

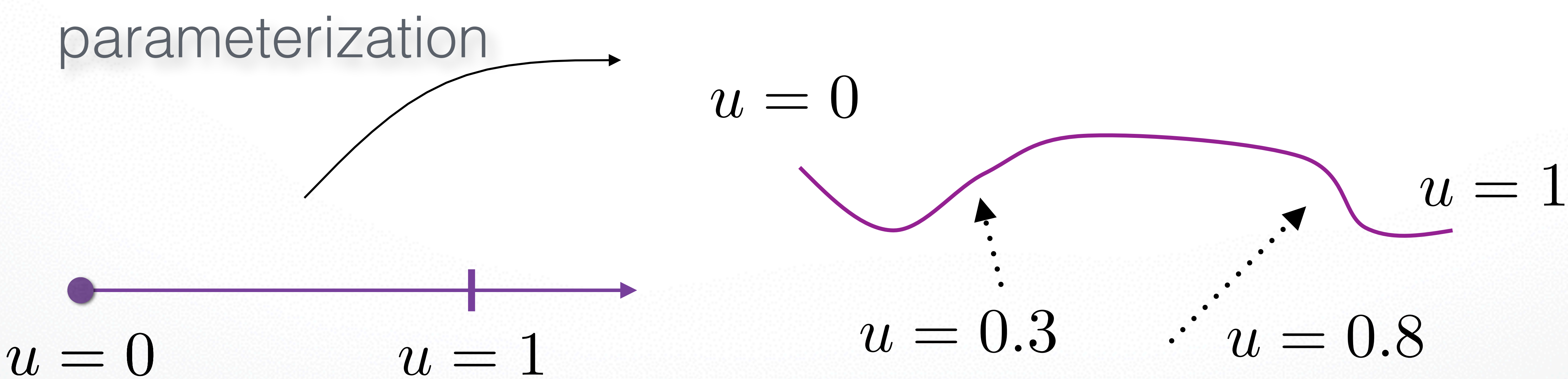


- Implicit: $f(x, y) = 0$
  - y can be a multiple valued function of x
  - Hard to specify, modify, control

Parameterization of a Curve

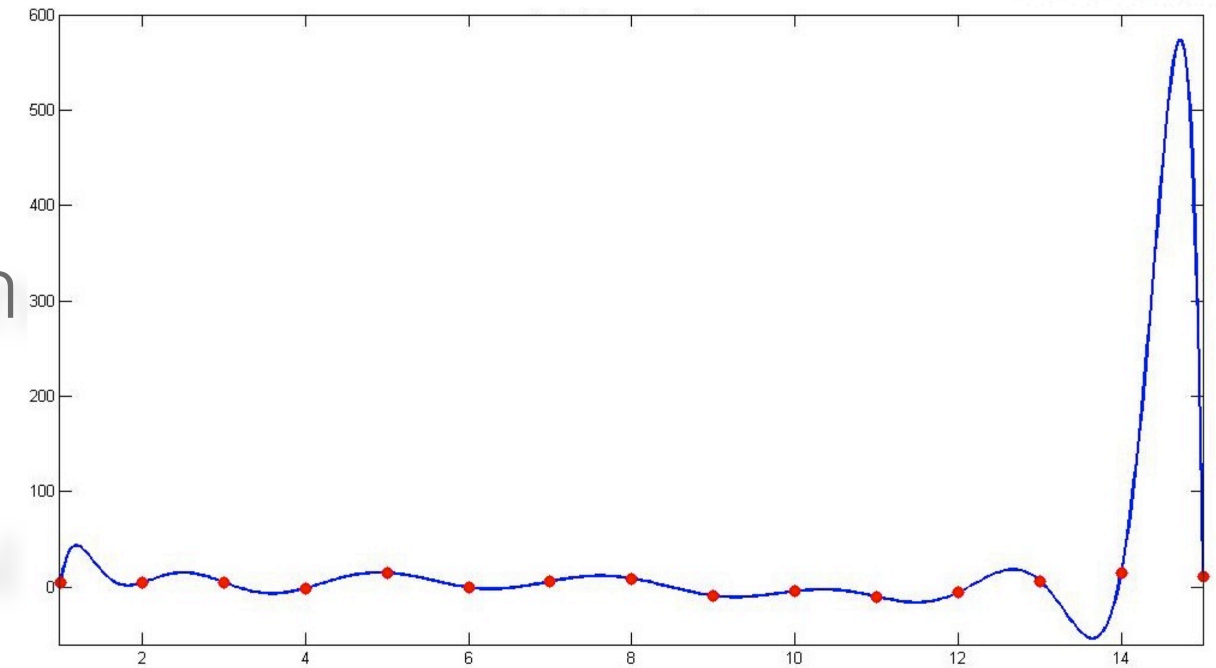
- *Parameterization* of a curve: how a change in u moves you along a given curve in xyz space.

- Parameterization is not unique. It can be slow, fast, with continuous / discontinuous speed, clockwise (CW) or CCW…



Polynomial Interpolation

- An n-th degree polynomial fits a curve to n+1 points
  - called Lagrange Interpolation
  - result is a curve that is too wiggly, change to any control point affects entire curve (non-local)
  - *this method is poor*



source:Wikipedia

Lagrange interpolation, degree=15

- We usually want the curve to be as smooth as possible
  - minimize the wiggles
  - high-degree polynomials are bad

Polynomial Approximation

Polynomials are computable functions

$$f(t) \;=\; \sum_{i=0}^{p} c_i\, t^i \;=\; \sum_{i=0}^{p} \tilde{c}_i\, \phi_i(t)$$

Taylor expansion up to degree p

$$g(h) \;=\; \sum_{i=0}^{p} \frac{1}{i!}\, g^{(i)}(0)\; h^i \;+\; O\big(h^{p+1}\big)$$

Error for approximation g by polynomial f

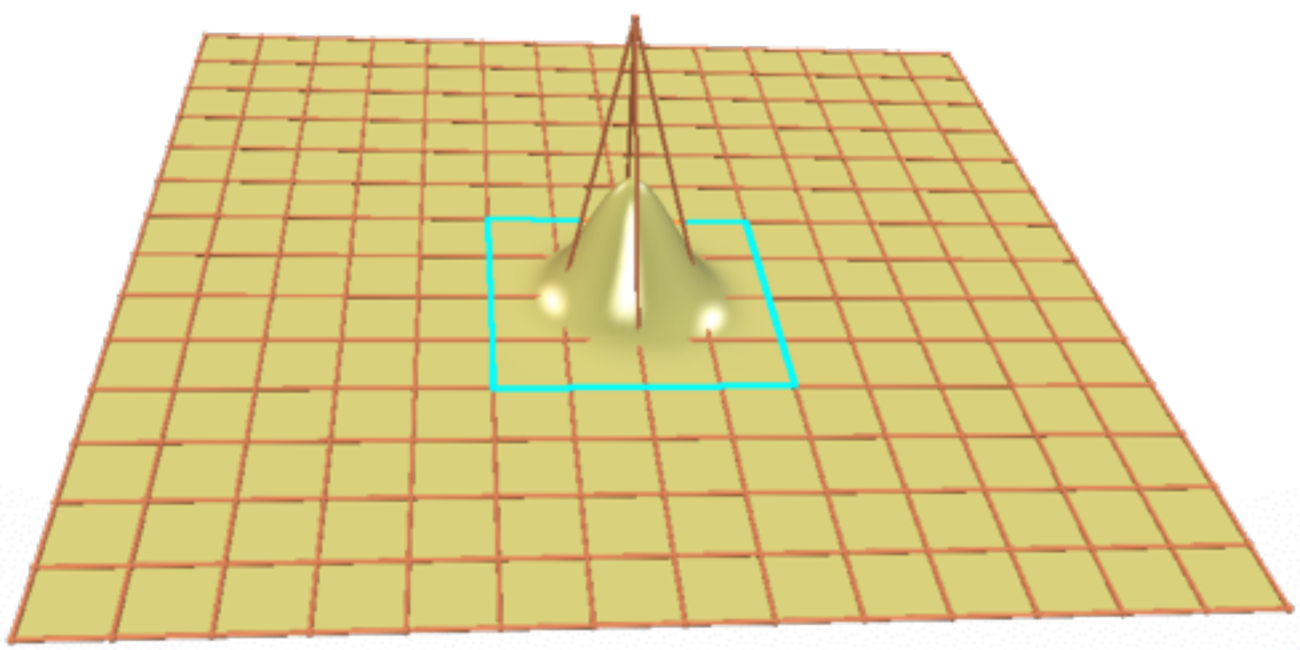
$$f(t_i) = g(t_i)\,, \quad 0 \le t_0 < \cdots < t_p \le h$$

$$|f(t) - g(t)| \;\le\; \frac{1}{(p+1)!}\; \max f^{(p+1)} \prod_{i=0}^{p} (t - t_i) \;=\; O\Big(h^{(p+1)}\Big)$$

Spline Surfaces

Piecewise polynomial approximation

$$\mathbf{f}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{c}_{ij} N_i^n(u) N_j^m(v)$$

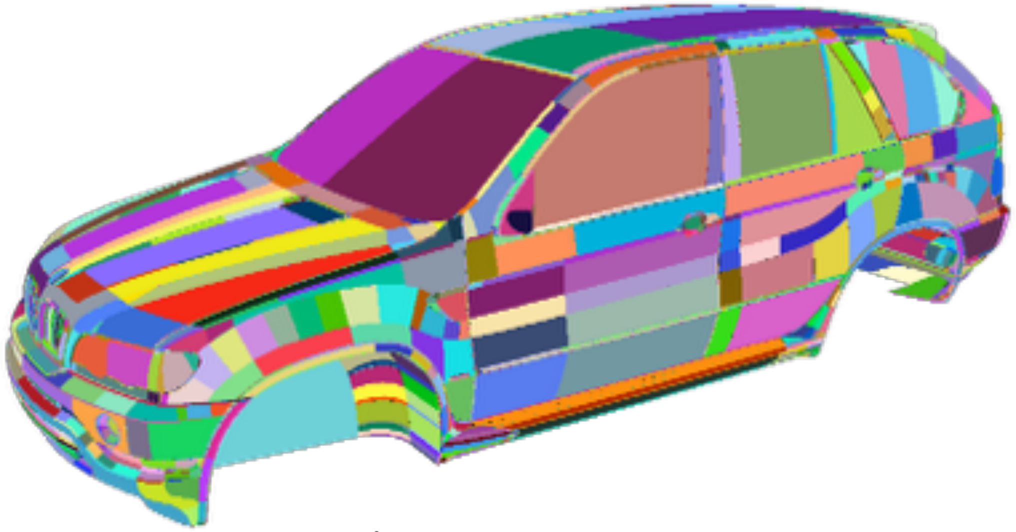


Spline Surfaces

Piecewise polynomial approximation

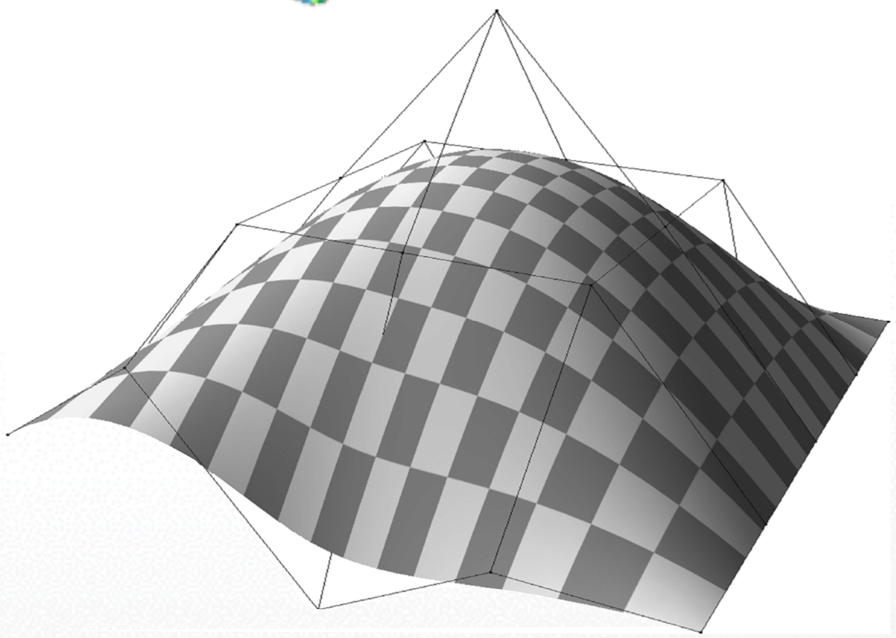
Geometric constraints

- Large number of patches
- Continuity between patches
- Trimming

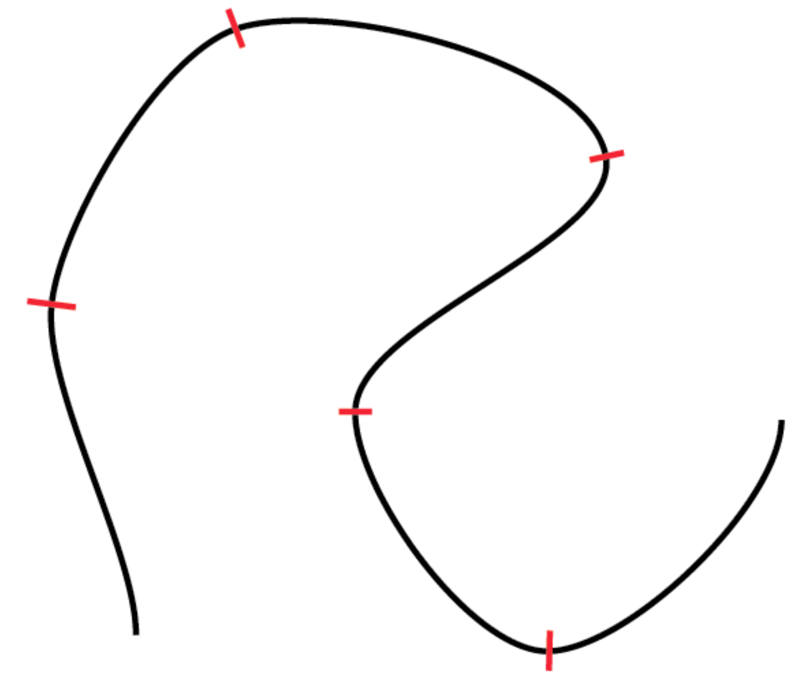


Topological constraints

- Rectangular patches
- Regular control mesh



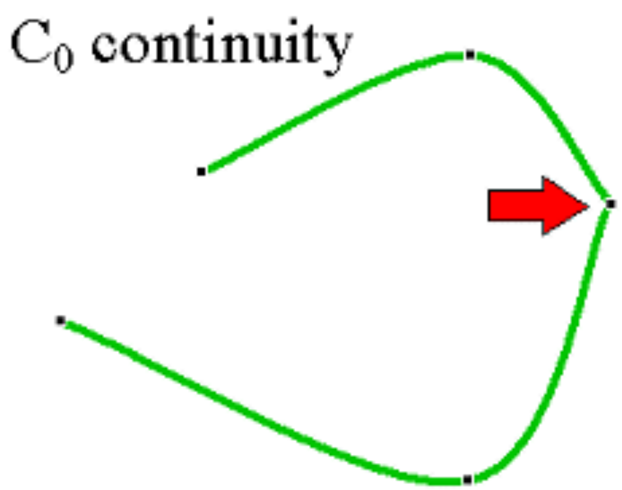
Splines: Piecewise Polynomials



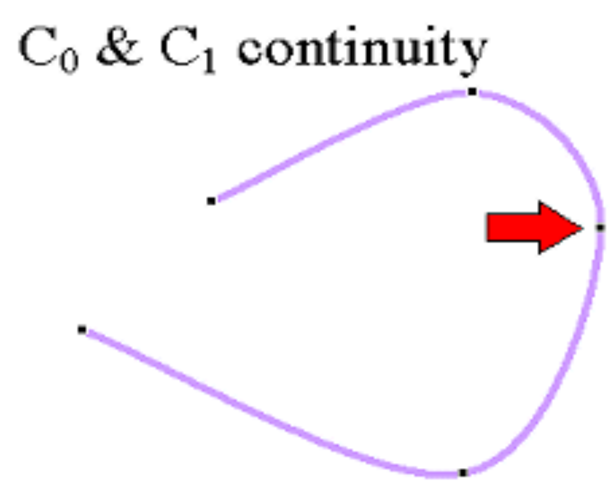
- A spline is a *piecewise polynomial:* Curve is broken into consecutive segments, each of which is a low-degree polynomial interpolating (passing through) the control points
- *Cubic* piecewise polynomials are the most common:
  - They are the lowest order polynomials that
    1. interpolate two points and
    2. allow the gradient at each point to be defined (C^1 continuity is possible)
  - Piecewise definition gives local control
  - Higher or lower degrees are possible, of course

Piecewise Polynomials

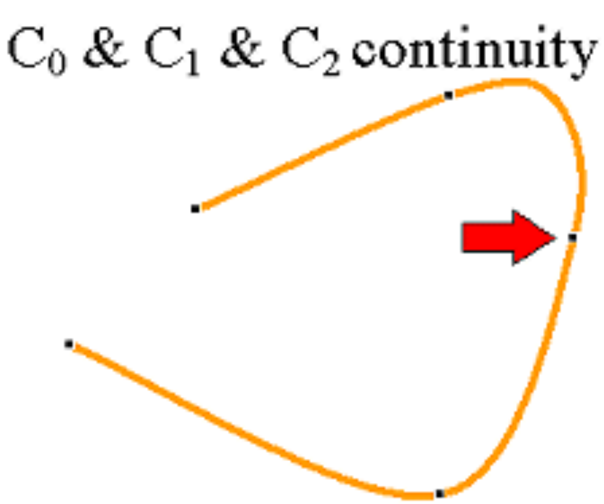
- Spline: many polynomials pieced together
- Want to make sure they fit together nicely



Continuous in position



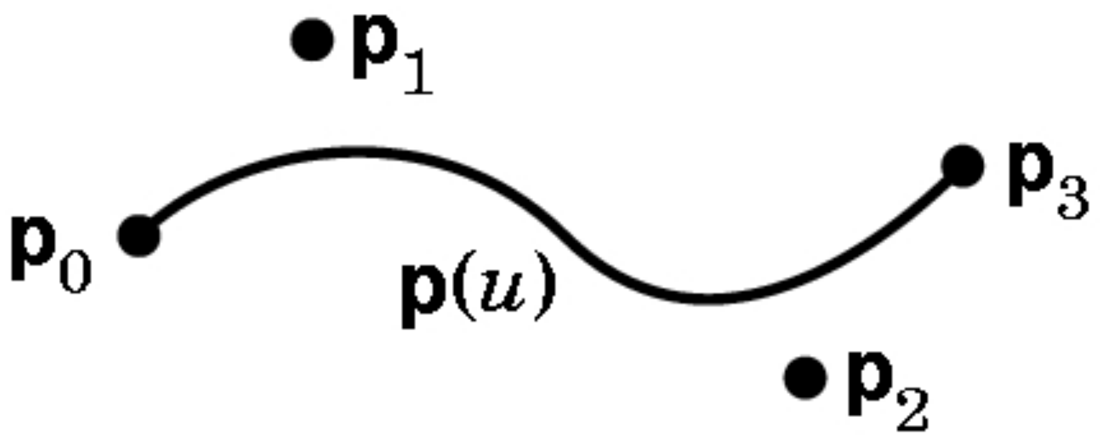
Continuous in position and tangent vector



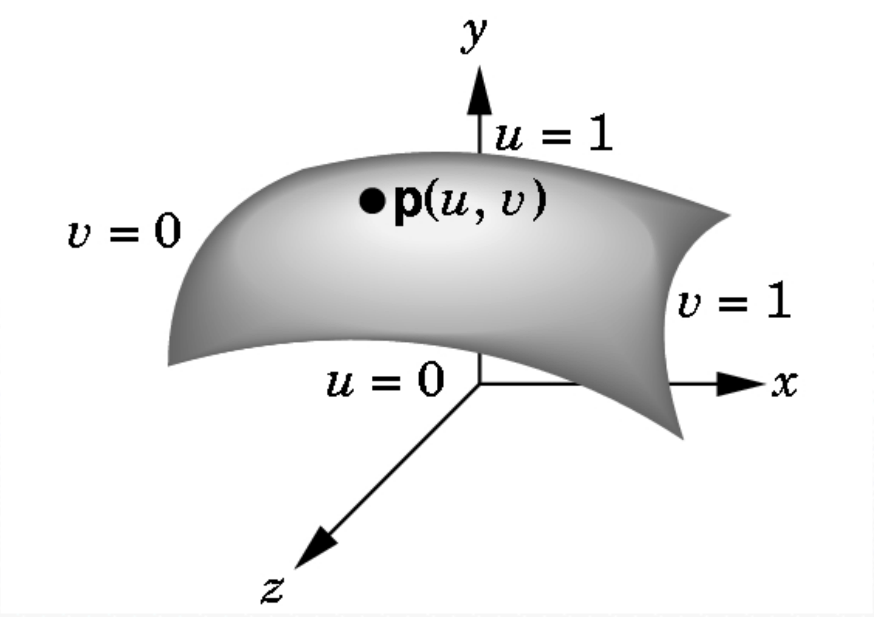
Continuous in position, tangent, and curvature

Splines

- Types of splines:
  - Hermite Splines
  - Bezier Splines
  - Catmull-Rom Splines
  - Natural Cubic Splines
  - B-Splines
  - NURBS



- Splines can be used to model both curves and surfaces



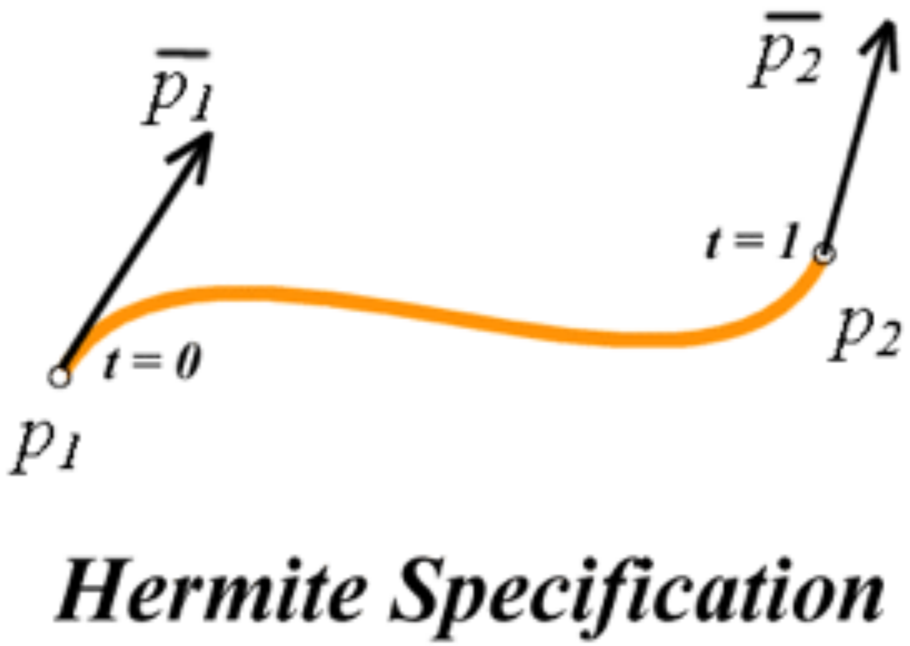
Cubic Curves in 3D

- Cubic polynomial:
  - $p(u) = au^3 + bu^2 + cu + d$
    $= \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} a & b & c & d \end{bmatrix}$
  - a, b, c, d are 3-vectors, u is a scalar
- Three cubic polynomials, one for each coordinate:
  - $x(u) = a_x u^3 + b_x u^2 + c_x u + d_x$
  - $y(u) = a_y u^3 + b_y u^2 + c_y u + d_y$
  - $z(u) = a_z u^3 + b_z u^2 + c_z u + d_z$
- In matrix notation:

$$\begin{bmatrix} x(u) & y(u) & z(u) \end{bmatrix} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}$$

- Or simply: $p = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} A$

Cubic Hermite Splines



We want a way to specify the end points and the slope at the end points!

Deriving Hermite Splines

- Four constraints: value and slope
  (in 3-D, position and tangent vector)
  at beginning and end of interval [0,1] :

$$p(0) = p_1 = (x_1, y_1, z_1)$$
$$p(1) = p_2 = (x_2, y_2, z_2)$$
$$p'(0) = \overline{p_1} = (\overline{x_1}, \overline{y_1}, \overline{z_1})$$
$$p'(1) = \overline{p_2} = (\overline{x_2}, \overline{y_2}, \overline{z_2})$$

the user constraints

- Assume cubic form: $p(u) = au^3 + bu^2 + cu + d$

- Four unknowns: a, b, c, d

Deriving Hermite Splines

- Assume cubic form: $p(u) = au^3 + bu^2 + cu + d$

$$
\begin{aligned}
p_1 &= p(0) = d \\
p_2 &= p(1) = a + b + c + d \\
\overline{p_1} &= p'(0) = c \\
\overline{p_2} &= p'(1) = 3a + 2b + c
\end{aligned}
$$

- Linear system: 12 equations for 12 unknowns
(however, can be simplified to 4 equations for 4 unknowns)

- Unknowns: a, b, c, d (each of a, b, c, d is a 3-vector)

Deriving Hermite Splines

$$\begin{aligned} d &= p_1 \\ a + b + c + d &= p_2 \\ c &= \overline{p_1} \\ 3a + 2b + c &= \overline{p_2} \end{aligned}$$

Rewrite this 12x12 system as a 4x4 system:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \overline{x_1} & \overline{y_1} & \overline{z_1} \\ \overline{x_2} & \overline{y_2} & \overline{z_2} \end{bmatrix}$$

The Cubic Hermite Spline Equation

- After inverting the 4x4 matrix, we obtain:

$$\underbrace{\begin{bmatrix} x & y & z \end{bmatrix}}_{\text{point on the spline}} = \underbrace{\begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}}_{\text{parameter vector}} \underbrace{\begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}}_{\text{basis}} \underbrace{\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \bar{x}_1 & \bar{y}_1 & \bar{z}_1 \\ \bar{x}_2 & \bar{y}_2 & \bar{z}_2 \end{bmatrix}}_{\text{control matrix (what the user gets to pick)}}$$

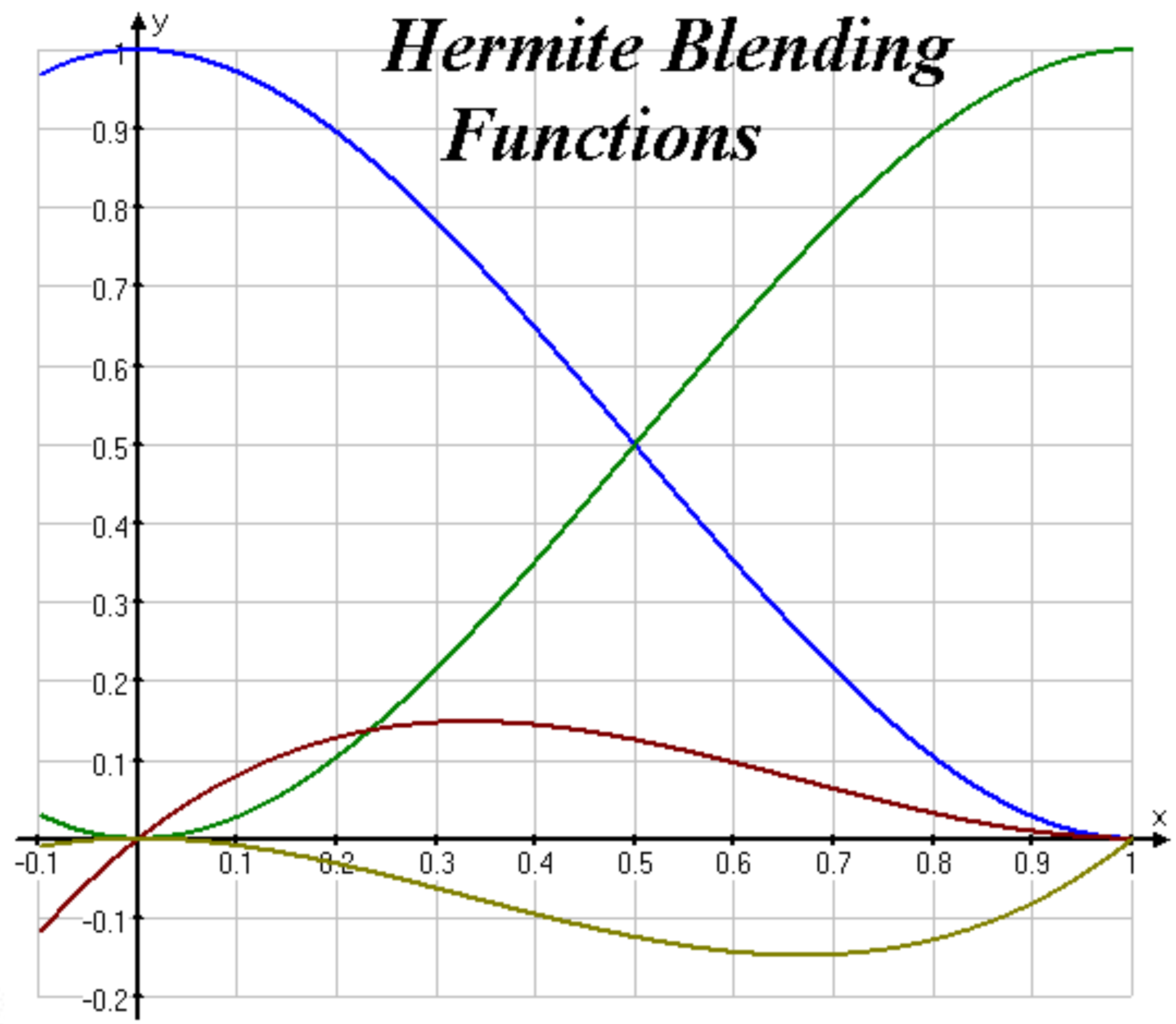
- This form is typical for splines
  - basis matrix and meaning of control matrix change with the spline type

Four Basis Functions for Hermite Splines

transpose

$$p(u) = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix}^T \begin{bmatrix} p_1 \\ p_2 \\ \overline{p}_1 \\ \overline{p}_2 \end{bmatrix}$$

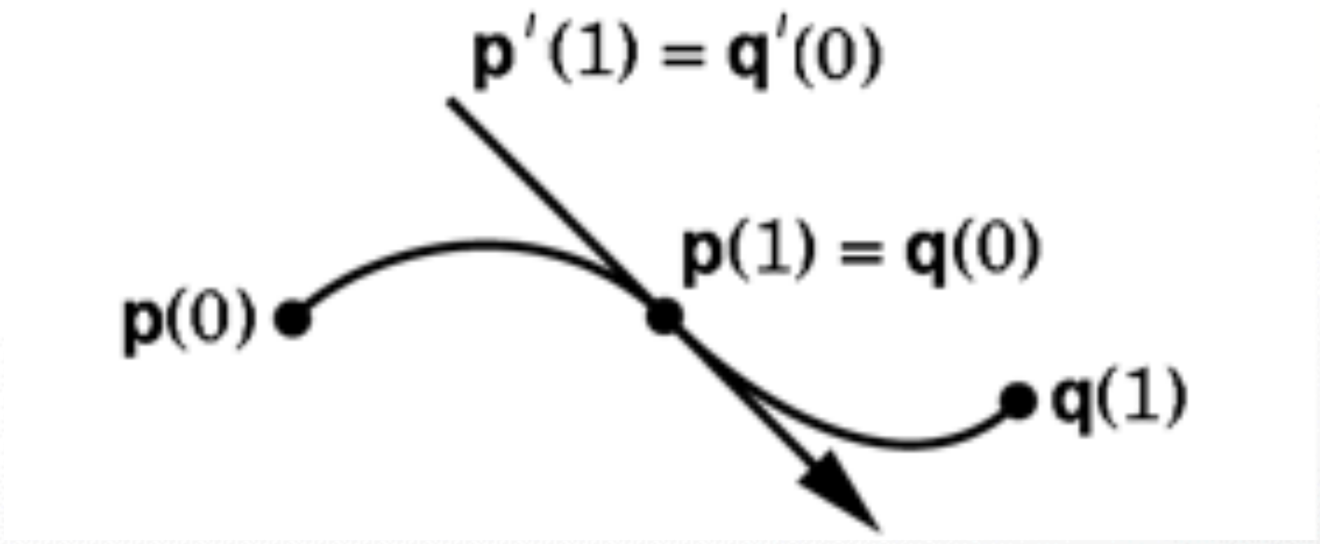
4 Basis Functions



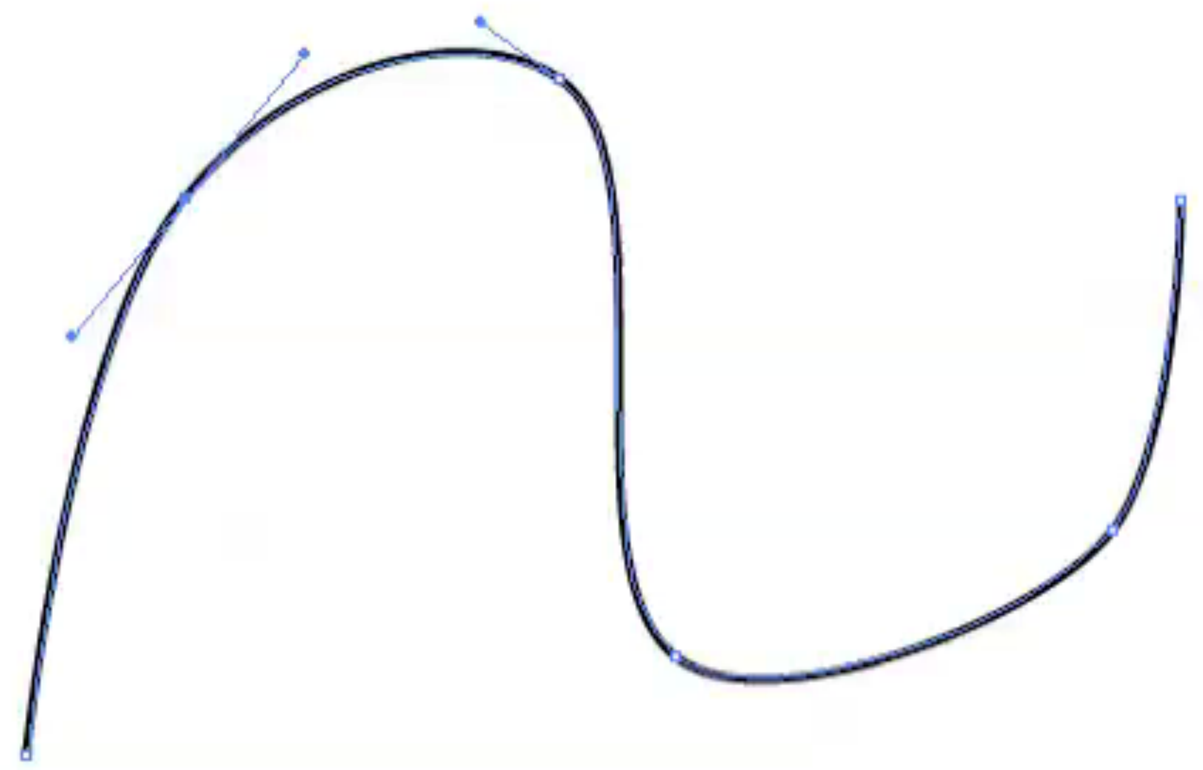
Every cubic Hermite spline is a linear combination (blend) of these 4 functions.

Piecing together Hermite Splines

- It's easy to make a multi-segment Hermite spline:
  - each segment is specified by a cubic Hermite curve
  - just specify the position and tangent at each “joint” (called knot)
  - the pieces fit together with matched positions and first derivatives
  - gives C1 continuity

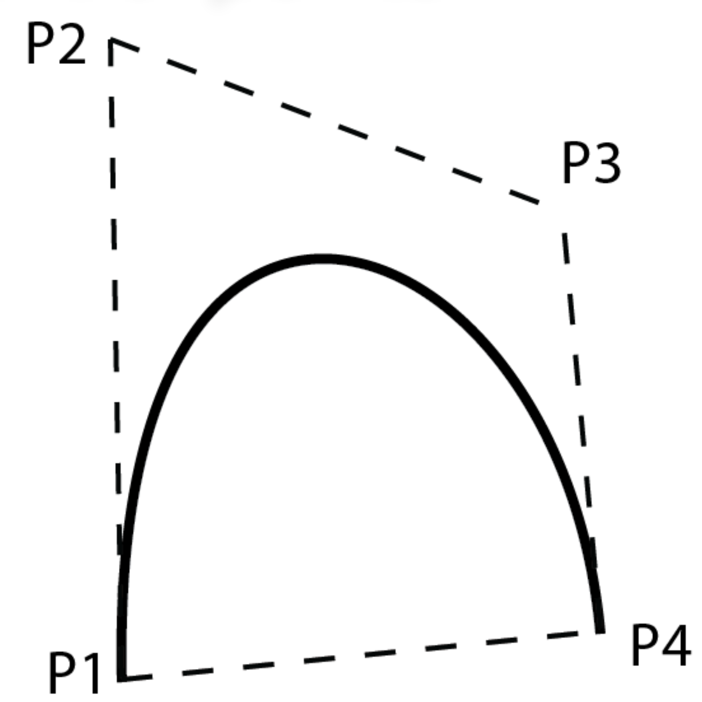


Hermite Splines in Adobe Illustrator



Bezier Splines

- Variant of the Hermite spline
- Instead of endpoints and tangents, four control points
  - points $P1$ and $P4$ are on the curve
  - points $P2$ and $P3$ are off the curve
  - $p(0) = P1, p(1) = P4$
  - $p'(0) = 3(P2 - P1),\ p'(1) = 3(P4 - P3)$



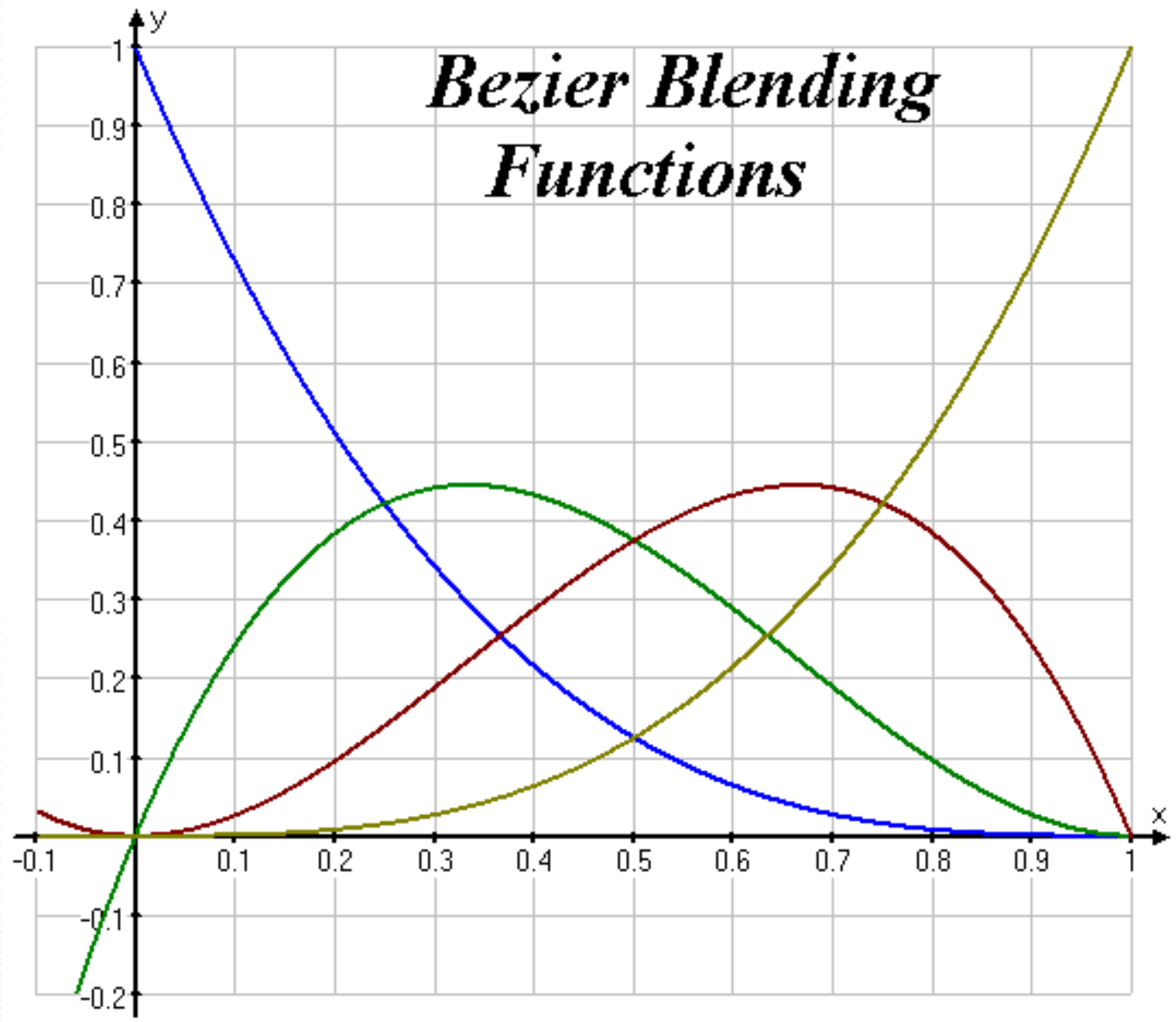
- Basis matrix is derived from the Hermite basis (or from scratch)
- Convex Hull property: curve contained within the convex hull of control points
- Scale factor "3" is chosen to make "velocity" approximately constant

The Bezier Spline Matrix

$$
\begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}
\underbrace{\begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}}_{\text{Hermite basis}}
\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix}}_{\text{Bezier to Hermite}}
\underbrace{\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{bmatrix}}_{\text{Bezier control matrix}}
$$

$$
= \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}
\underbrace{\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}}_{\text{Bezier basis}}
\underbrace{\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{bmatrix}}_{\text{Bezier control matrix}}
$$

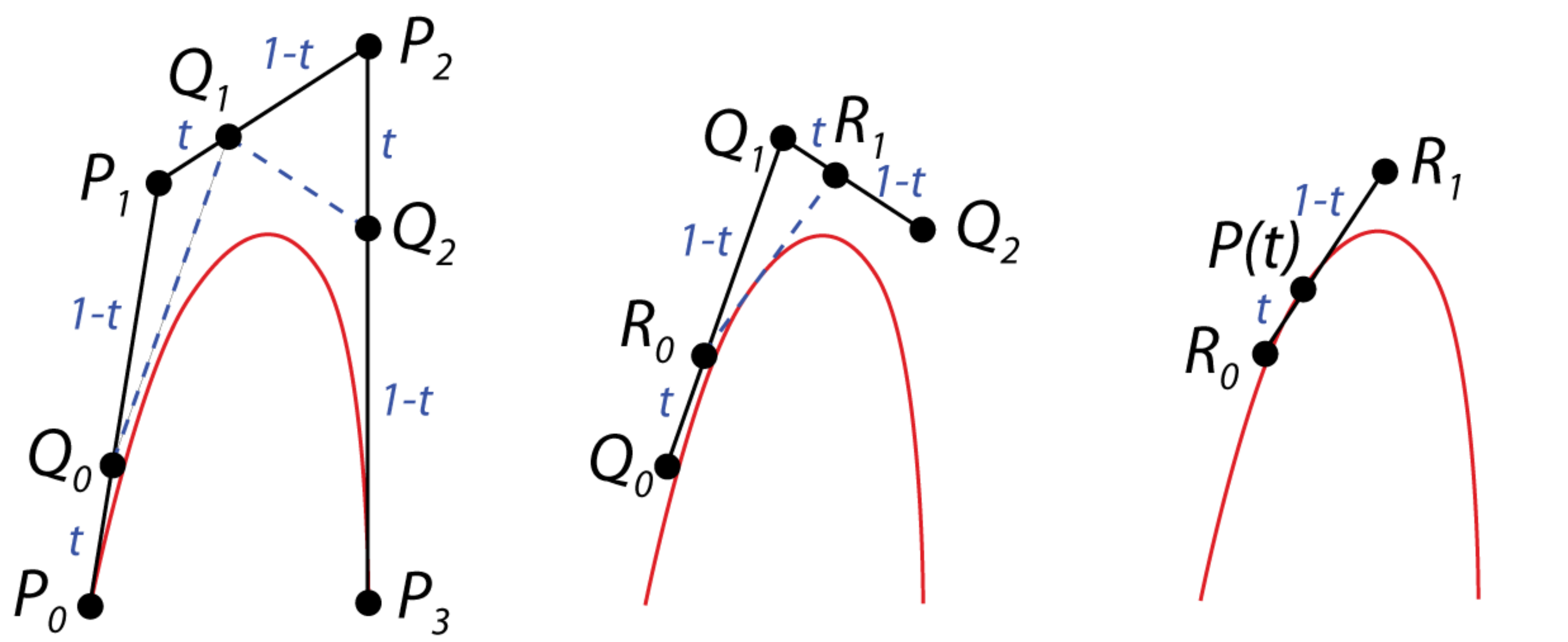
Bezier Blending Functions



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

- Also known as the order 4, degree 3 Bernstein polynomials
- Nonnegative, sum to 1
- The entire curve lies inside the polyhedron bounded by the control points

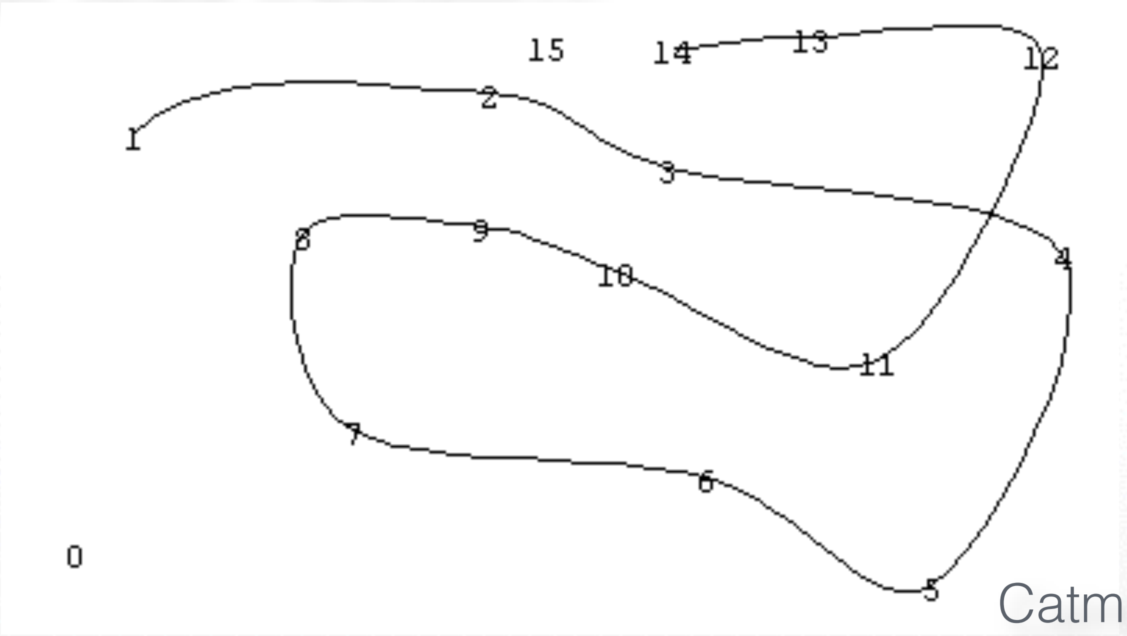
DeCasteljau Construction



Efficient algorithm to evaluate Bezier splines.
Similar to Horner rule for polynomials.
Can be extended to interpolations of 3D rotations.

Catmull-Rom Splines

- Roller-coaster (next programming assignment)
- With Hermite splines, the designer must arrange for consecutive tangents to be collinear, to get C^1 continuity. Similar for Bezier. This gets tedious.
- Catmull-Rom: an interpolating cubic spline with *built-in C^1 continuity*.
- Compared to Hermite/Bezier: fewer control points required, but less freedom.



Catmull-Rom spline

Constructing the Catmull-Rom Spline

- Suppose we are given n control points in 3-D: $p_1, p_2, ..., p_n$
- For a Catmull-Rom spline, we set the tangent at p_i to $s * (p_{i+1} - p_{i-1})$ for $i = 2, ..., n-1$ for some s (often $s = 0.5$)
- s is *tension parameter*: determines the magnitude (but not direction!) of the tangent vector at point p_i
- What about endpoint tangents? Use extra control points p_0, p_{n+1}
- Now we have positions and tangents at each knot. This is a Hermite specification. Now, just use Hermite formulas to derive the spline
- Note: curve between p_i and p_{i+1} is completely determined by $p_{i-1}, p_i, p_{i+1}, p_{i+2}$

Catmull-Rom Spline Matrix

$$\begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{bmatrix}$$

basis control matrix

- Derived in way similar to Hermite and Bezier
- Parameter s is typically set to s=1/2

Splines with More Continuity?

- So far, only C^1 continuity
- How could we get C^2 continuity at control points?
- Possible answers:

  - Use higher degree polynomials

    degree 4 = quartic, degree 5 = quintic, … but these get computationally expensive, and sometimes wiggly

  - Give up local control → natural cubic splines

    A change to any control point affects the entire curve

  - Give up interpolation → cubic B-splines

    Curve goes near, but not through, the control points

Comparison of Basic Cubic Splines

Type	Local Control	Continuity	Interpolation
Hermite	YES	C1	YES
Bezier	YES	C1	YES
Catmull-Rom	YES	C1	YES
Natural	NO	C2	YES
B-Splines	YES	C2	NO

Summary:

Cannot get C2, interpolation and local control with cubics

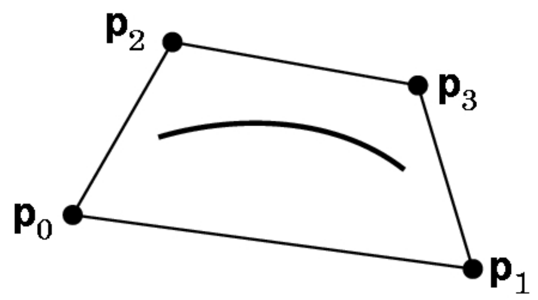
Natural Cubic Splines

- If you want 2nd derivatives at joints to match up, the resulting curves are called *natural cubic splines*
- It's a simple computation to solve for the cubics' coefficients. (See *Numerical Recipes in C* book for code.)
- Finding all the right weights is a *global* calculation (solve tridiagonal linear system)

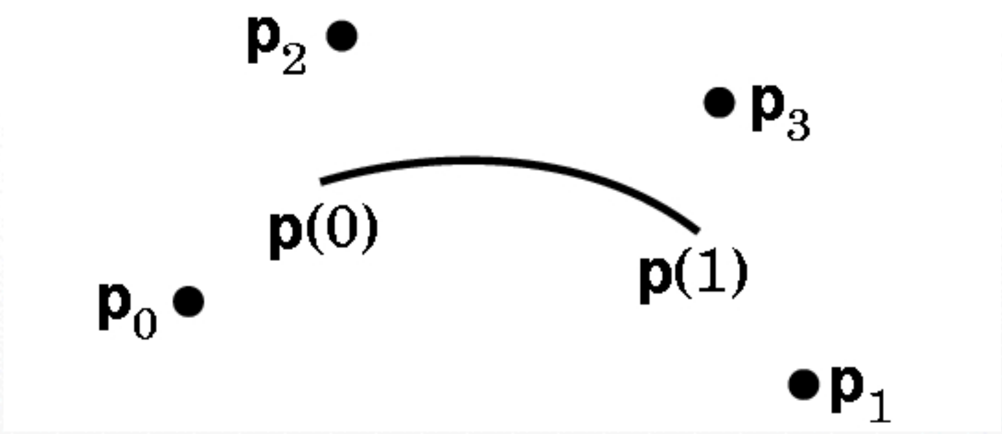
B-Splines

- Give up interpolation
  - the curve passes *near* the control points
  - best generated with interactive placement (because it's hard to guess where the curve will go)

- Curve obeys the convex hull property



- C2 continuity and local control are good compensation for loss of interpolation

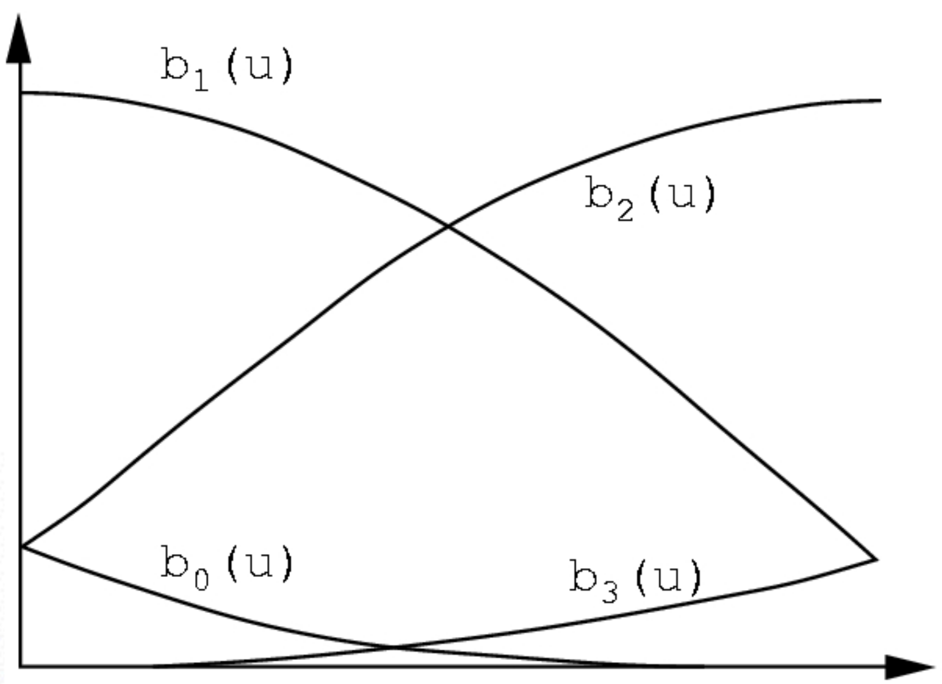


B-Spline Basis

- We always need 3 more control points than the number of spline segments

$$M_{Bs} = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

$$G_{Bs_i} = \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}$$



Other Common Types of Splines

- Non-Uniform Splines
- Non-Uniform Rational Cubic curves (NURBS)
- NURBS are very popular and used in many commercial packages

How to Draw Spline Curves

- Basis matrix equation allows same code to draw any spline type
- **Method 1**: brute force
  - Calculate the coefficients
  - For each cubic segment, vary u from 0 to 1 (fixed step size)
  - Plug in u value, matrix multiply to compute position on curve
  - Draw line segment from last position to current position
- What's wrong with this approach?
  - Draws in even steps of u
  - Even steps of u does not mean even steps of x
  - Line length will vary over the curve
  - Want to bound line length

    too long: curve looks jagged

    too short: curve is slow to draw

Drawing Splines, 2

- **Method 2**: recursive subdivision

  - vary step size to draw short lines

```
Subdivide(u0,u1,maxlinelength)
umid = (u0 + u1)/2
x0 = F(u0)
x1 = F(u1)
if |x1 - x0| > maxlinelength
    Subdivide(u0,umid,maxlinelength)
    Subdivide(umid,u1,maxlinelength)
else drawline(x0,x1)
```

- **Variant on Method 2** - subdivide based on curvature
  - replace condition in “if” statement with straightness criterion
  - draws fewer lines in flatter regions of the curve

Summary

- Piecewise cubic is generally sufficient
- Define conditions on the curves and their continuity
- Most important:
  - basic curve properties
    (what are the conditions, controls, and properties for each spline type)
  - generic matrix formula for uniform cubic splines

    $$p(u) = u\ B\ G$$

  - given a definition, derive a basis matrix
    (do not memorize the matrices themselves)

http://cs420.hao-li.com

Thanks!