Designing Parametric Cubic Curves

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

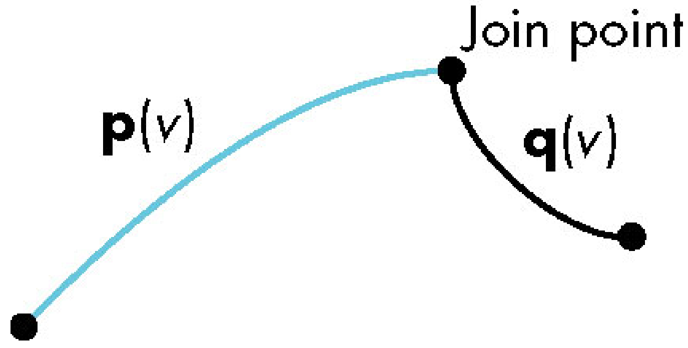
- Introduce the types of curves
    - Interpolating
    - Hermite
    - Bezier
    - B-Spline
- Analyze their performance

Design Criteria

- Why we prefer parametric polynomials of low degree:
  - Local control of shape,
  - Smoothness and continuity,
  - Ability to evaluate derivatives,
  - Stability,
  - Ease of rendering.

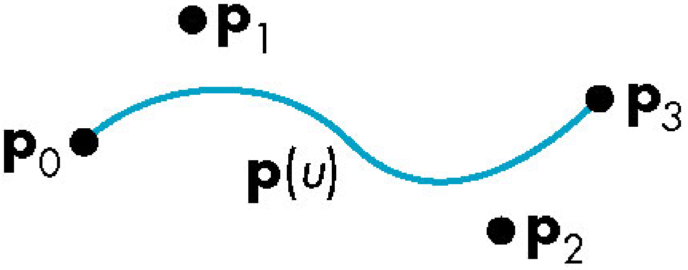
Smoothness

- Smoothness guaranteed because our polynomial equations are differentiable.
- Difficulties arise at the **join points**.



Control Points

- We prefer local control for stability.
  - The most common interface is a group of **control points**.



  - In this example, the curve passes through, or **interpolates**, some of the control points, but only comes close to, or **approximates**, others.

Parametric Cubic Polynomial Curves

- Choosing the degree:
  - High degree allows many control points, but computation is expensive.
  - Low degree may mean low level of control.
- The compromise: use low-degree curves over short intervals.
  - Most designers work with cubic polynomial curves.

Matrix Notation

$$\mathbf{p}(u) = \sum_{k=0}^{3} \mathbf{c}_k u^k = \mathbf{u}^T \mathbf{c},$$

the coefficient matrix to be determined

where

control points

$$\mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}, \quad \mathbf{c}_k = \begin{bmatrix} c_{kx} \\ c_{ky} \\ c_{kz} \end{bmatrix}.$$

Interpolation

- An **interpolating polynomial** passes through its control points.
  - Suppose we have four controls points

$$\mathbf{p}_k = \begin{bmatrix} x_k \\ y_k \\ z_k \end{bmatrix}, \text{ for } 0 \le k \le 3.$$

  - We let u vary over the interval [0,1], giving us four equally spaced values: 0, 1/3, 2/3, 1.

Evaluating the Control Points

- We seek coefficients $\mathbf{c}_0$, $\mathbf{c}_1$, $\mathbf{c}_2$, $\mathbf{c}_3$ satisfying the four conditions:

$$\mathbf{p}_0 = \mathbf{p}(0) = \mathbf{c}_0,$$

$$\mathbf{p}_1 = \mathbf{p}(1/3) = \mathbf{c}_0 + 1/3\,\mathbf{c}_1 + (1/3)^2\mathbf{c}_2 + (1/3)^3\mathbf{c}_3,$$

$$\mathbf{p}_2 = \mathbf{p}(2/3) = \mathbf{c}_0 + 2/3\,\mathbf{c}_1 + (2/3)^2\mathbf{c}_2 + (2/3)^3\mathbf{c}_3,$$

$$\mathbf{p}_3 = \mathbf{p}(1) = \mathbf{c}_0 + \mathbf{c}_1 + \mathbf{c}_2 + \mathbf{c}_3.$$

Matrix Notation

- In matrix notation $\mathbf{p} = \mathbf{A}\mathbf{c}$, where

$$\mathbf{p} = \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \quad \text{and} \quad \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1/3 & (1/3)^2 & (1/3)^3 \\ 1 & 2/3 & (2/3)^2 & (2/3)^3 \\ 1 & 1 & 1 & 1 \end{bmatrix}.$$

a column vector of row vectors

nonsingular: we will use its inverse

Interpolating Geometry Matrix

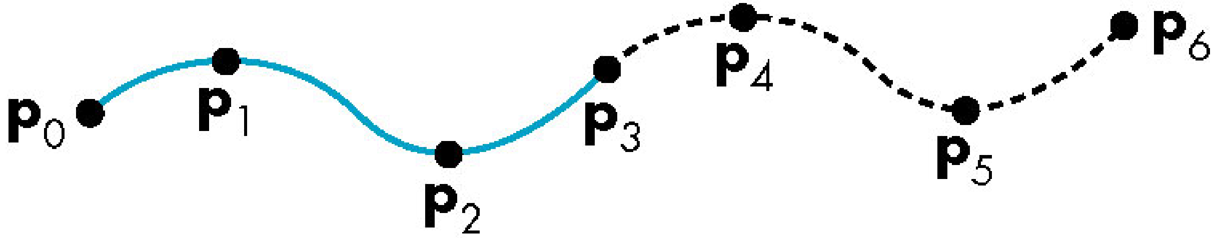
$$\mathbf{M}_I = \mathbf{A}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{bmatrix}$$

- The desired coefficients are

$$\mathbf{c} = \mathbf{M}_I \mathbf{p}.$$

Interpolating Multiple Segments

- Use the last control point of one segment as the first control point of the next segment.



  - To achieve smoothness in addition to continuity, we will need additional constraints on the derivatives.

Blending Functions

- Substituting the interpolating coefficients into our polynomial:

$$\mathbf{p}(u) = \mathbf{u}^T \mathbf{c} = \mathbf{u}^T \mathbf{M}_I \mathbf{p}.$$

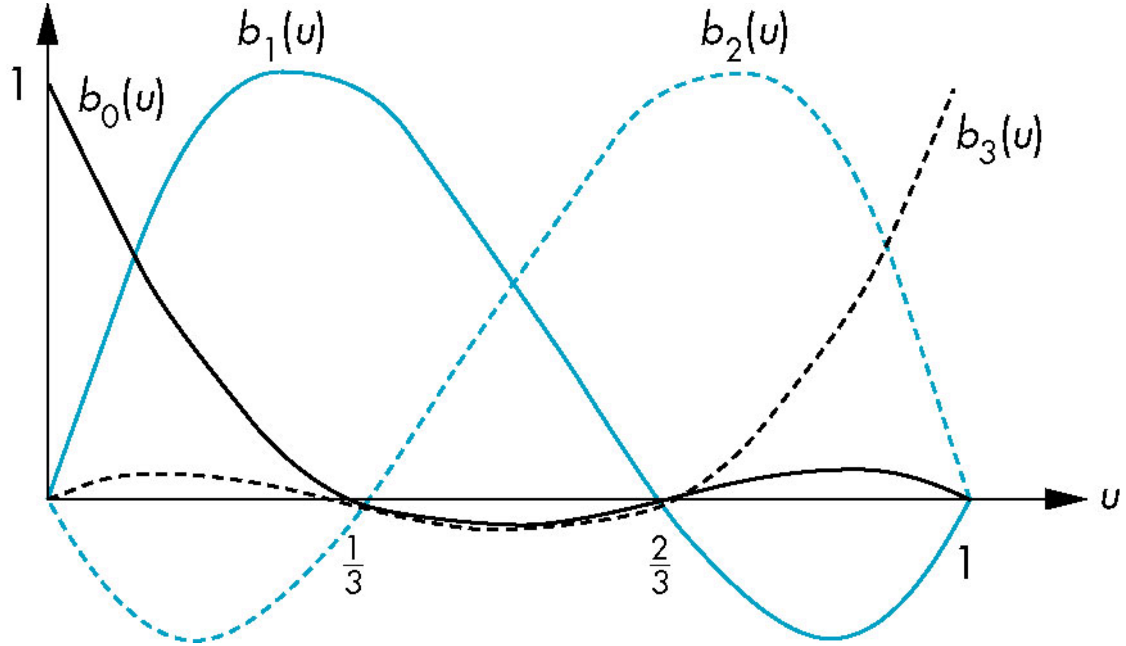
- Let

$$\mathbf{p}(u) = \mathbf{b}(u)^T \mathbf{p}, \text{ where } \mathbf{b}(u) = \mathbf{M}_I^T \mathbf{u}.$$

- The **b**(*u*) are the **blending polynomials**.

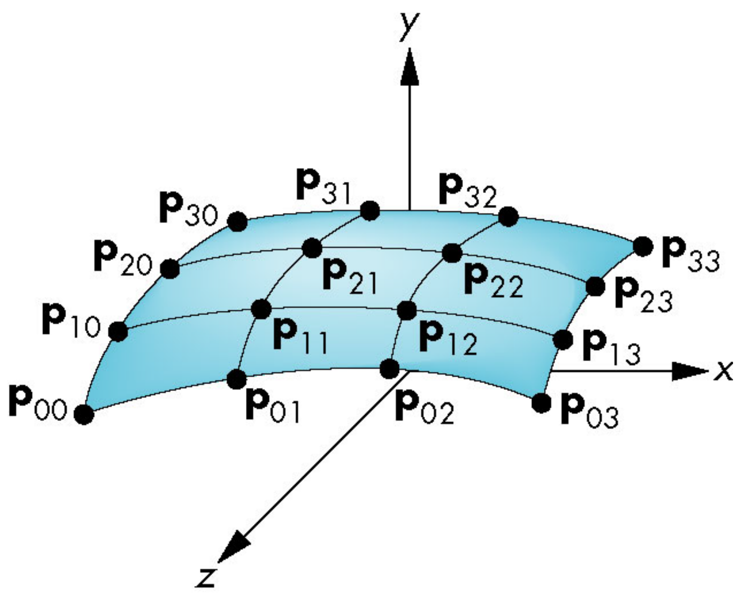
Visualizing the Curve Using Blending Functions

- The effect on the curve of an individual control point is easier to see by studying its blending function.



The Cubic Interpolating Patch

- A **bicubic surface patch**:



$$\mathbf{p}(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} u^i v^j \mathbf{c}_{ij}.$$

Matrix Notation

- In matrix form, the patch is defined by

$$\mathbf{p}(u,v) = \mathbf{u}^T \mathbf{C}\mathbf{v},$$

  - The column vector $\mathbf{v} = [1\ v\ \ v^2\ v^3]^T$.
  - $\mathbf{C}$ is a 4 x 4 matrix of column vectors.
- 16 equations in 16 unknowns.

Solving the Surface Equations

- By setting *v* = 0, 1/3, 2/3, 1 we can sample the surface using curves in *u*:

$$\mathbf{u}^T \mathbf{M}_I \mathbf{P} = \mathbf{u}^T \mathbf{C} \mathbf{A}^T .$$

- The coefficient matrix **C** is computed by

$$\mathbf{C} = \mathbf{M}_I \mathbf{P} \mathbf{M}_I^T .$$

- The equation for the surface becomes

$$\mathbf{p}(u,v) = \mathbf{u}^T \mathbf{M}_I \mathbf{P} \mathbf{M}_I^T \mathbf{v}.$$

Blending Patches

- Extending our use of blending polynomials to surfaces:

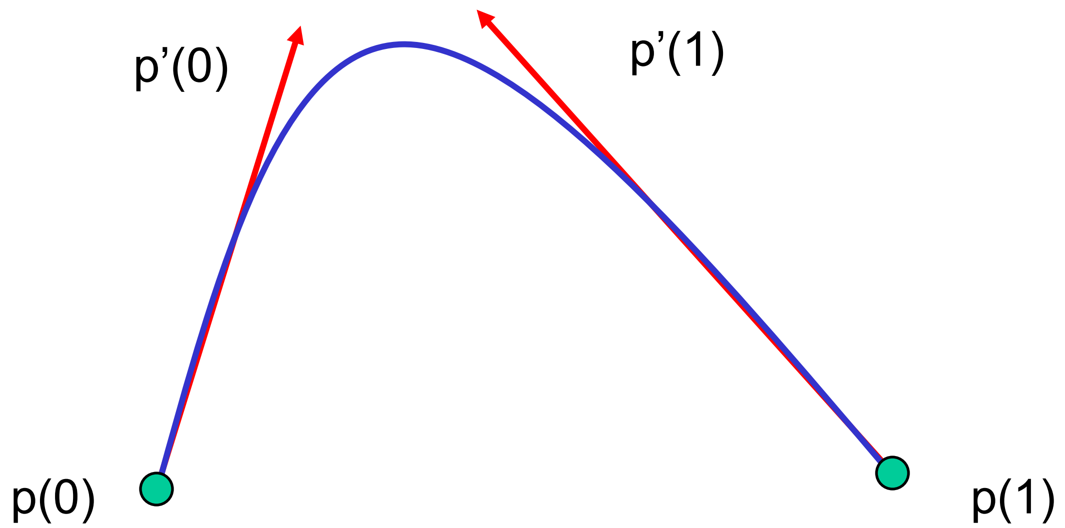
$$\mathbf{p}(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u) b_j(v) \mathbf{p}_{ij}.$$

  - 16 simple patches form a surface.
  - Also known as **tensor-product surfaces**.
  - These surfaces are not very smooth.
    - But they are **separable**, meaning they allow us to work with functions in *u* and *v* independently.

Other Types of Curves and Surfaces

- How can we get around the limitations of the interpolating form
  - Lack of smoothness
  - Discontinuous derivatives at join points
- We have four conditions (for cubics) that we can apply to each segment
  - Use them other than for interpolation
  - Need only come close to the data

Hermite Form

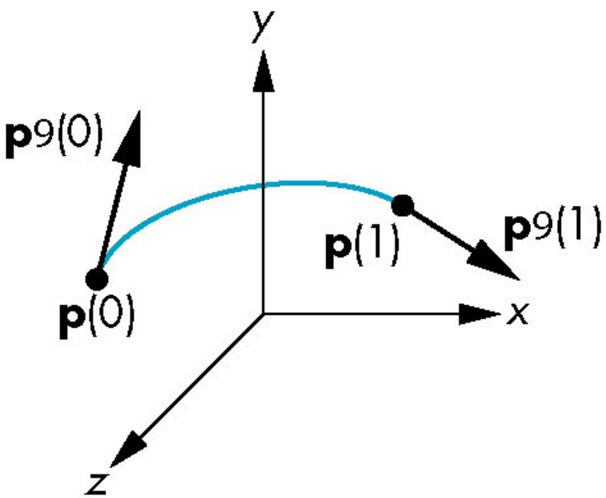


Use two interpolating conditions and two derivative conditions per segment

Ensures continuity and first derivative continuity between segments

Hermite Curves and Surfaces

- Use the data at control points differently in an attempt to get smoother results.
  - We insist that the curve interpolate the control points only at the two ends, $\mathbf{p}_0$ and $\mathbf{p}_3$.



$$\mathbf{p}(0) = \mathbf{p}_0 = \mathbf{c}_0,$$

$$\mathbf{p}(1) = \mathbf{p}_3 = \mathbf{c}_0 + \mathbf{c}_1 + \mathbf{c}_2 + \mathbf{c}_3.$$

Additional Conditions

- The derivative is a quadratic polynomial:

$$\mathbf{p}'(u) = \begin{bmatrix} dx/du \\ dy/du \\ dz/du \end{bmatrix} = \mathbf{c}_1 + 2u\mathbf{c}_2 + 3u^2\mathbf{c}_3.$$

- We now can derive two additional conditions:

$$\mathbf{p}'_0 = \mathbf{p}'(0) = \mathbf{c}_1,$$

$$\mathbf{p}'_3 = \mathbf{p}'(1) = \mathbf{c}_1 + 2\mathbf{c}_2 + 3\mathbf{c}_3.$$

Matrix Form

$$\begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_3 \\ \mathbf{p}'_0 \\ \mathbf{p}'_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \mathbf{c}.$$

call this $\mathbf{q}$

- The desired coefficient matrix is

$$\mathbf{c} = \mathbf{M}_H \mathbf{q}.$$

  - $\mathbf{M}_H$ is the **Hermite geometry** matrix.

The Hermite Geometry Matrix

$$\mathbf{M}_H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix}.$$

- The resulting polynomial is

$$\mathbf{p}(u) = \mathbf{u}^T \mathbf{M}_H \mathbf{q}.$$

Blending polynomials

- Using blending functions $\mathbf{p}(u)=\mathbf{b}(u)^T\mathbf{q}$,

$$\mathbf{b}(u) = \mathbf{M}_H^T \mathbf{u} = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix}.$$

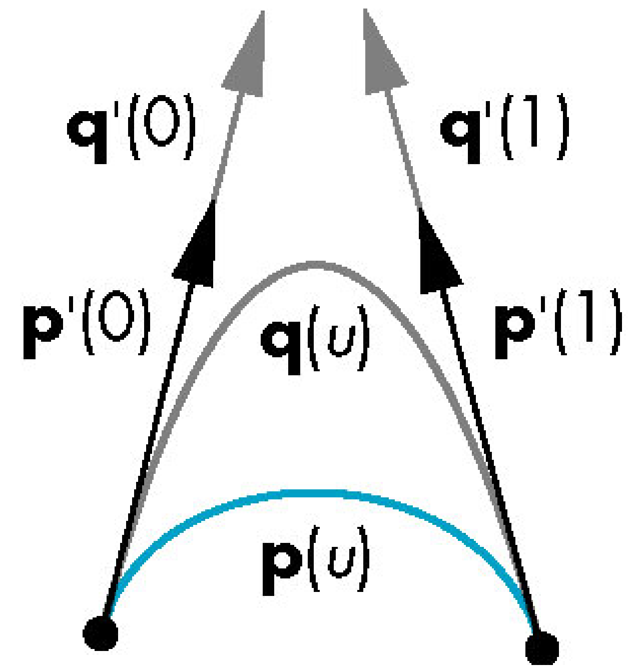
- Although these functions are smooth, the Hermite form is not used directly in Computer Graphics and CAD because we usually have control points but not derivatives
- However, the Hermite form is the basis of the Bezier form

Parametric and Geometric Continuity

- We can require the derivatives of x, y,and z to each be continuous at join points (*parametric continuity*)
- Alternately, we can only require that the tangents of the resulting curve be continuous (*geometry continuity*)
- The latter gives more flexibility as we have need satisfy only two conditions rather than three at each join point

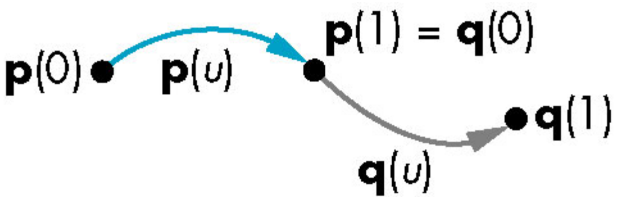
Example

- Here the p and q have the same tangents at the ends of the segment but different derivatives
- Generate different Hermite curves
- This techniques is used in drawing applications



Parametric Continuity

- Continuity is enforced by matching polynomials at join points.



- C^0 parametric continuity:

$$\mathbf{p}(1) = \begin{bmatrix} p_x(1) \\ p_y(1) \\ p_z(1) \end{bmatrix} = \mathbf{q}(0) = \begin{bmatrix} q_x(0) \\ q_y(0) \\ q_z(0) \end{bmatrix}.$$

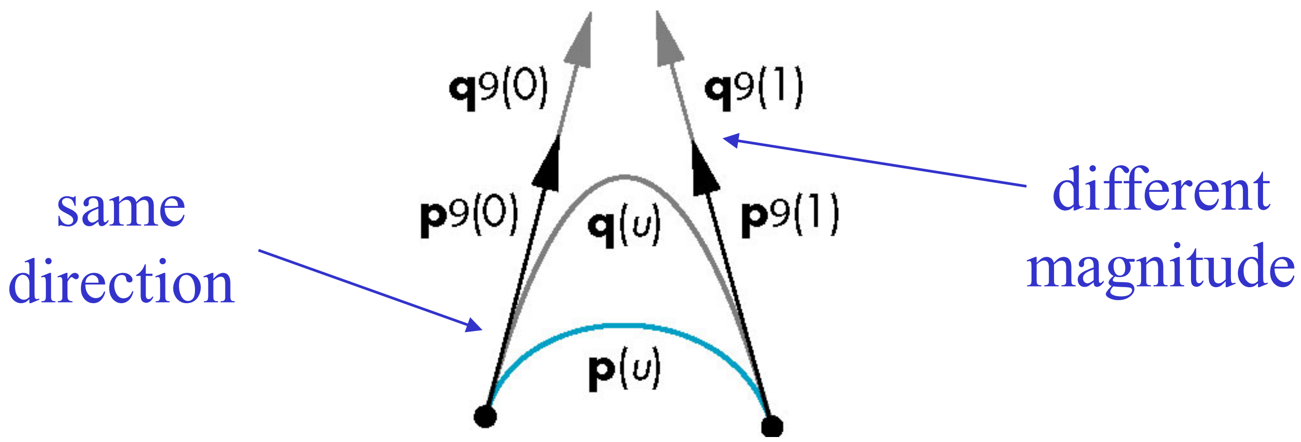
C^1 Parametric Continuity

- Matching derivatives at the join points gives us C^1 continuity:

$$\mathbf{p}'(1) = \begin{bmatrix} p_x'(1) \\ p_y'(1) \\ p_z'(1) \end{bmatrix} = \mathbf{q}'(0) = \begin{bmatrix} q_x'(0) \\ q_y'(0) \\ q_z'(0) \end{bmatrix}.$$

Another Approach: Geometric Continuity

- If the derivatives are proportional, then we have **geometric continuity**.



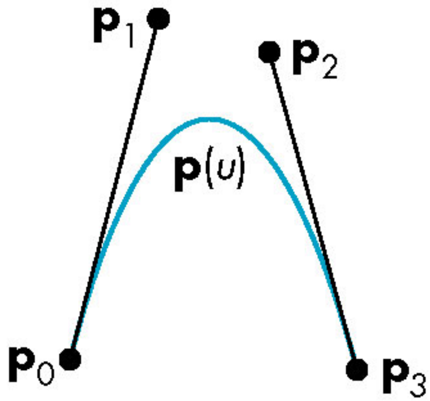
- One extra degree of freedom.
- Extends to higher dimensions.

Bezier Curves: Basic Idea

- In graphics and CAD, we usually don't have derivative data
- Bezier suggested using the same 4 data points as with the cubic interpolating curve to approximate the derivatives in the Hermite form

Bezier Curves and Surfaces

- Bezier added control points to manipulate derivatives.



  - The two derivative conditions become

$$3\mathbf{p}_1 - 3\mathbf{p}_0 = \mathbf{c}_1,$$

$$3\mathbf{p}_3 - 3\mathbf{p}_2 = \mathbf{c}_1 + 2\mathbf{c}_2 + 3\mathbf{c}_3.$$

Bezier Geometry Matrix

- We solve $\mathbf{c}=\mathbf{M}_B\mathbf{p}$, where

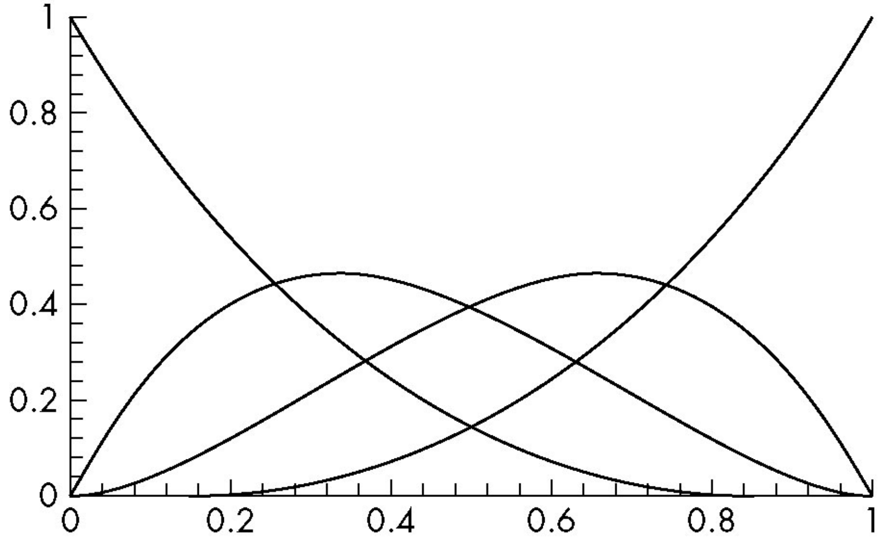
$$\mathbf{M}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & 3 & 1 \end{bmatrix}.$$

- The cubic Bezier polynomial is thus

$$\mathbf{p}(u) = \mathbf{u}^T \mathbf{M}_B \mathbf{p}.$$

Bezier Blending Functions

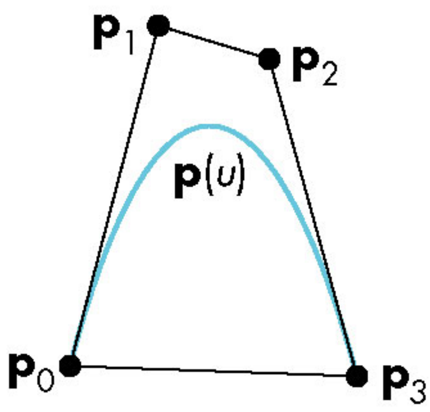
- These functions are **Bernstein polynomials**:



$$b_{kd}(u) = \frac{d!}{k!(d-k)!} u^k (1-u)^{d-k}.$$

Properties of Bernstein Polynomials

- All zeros are either at $u = 0$ or $u = 1$.
  - Therefore, the curve must be smooth over (0,1)
- The value of u never exceeds 1.
  - $\mathbf{p}(u)$ is a convex sum, so the curve lies inside the convex hull of the control points.



Bezier Surface Patches

- Using a 4 x 4 array of control points **P**,

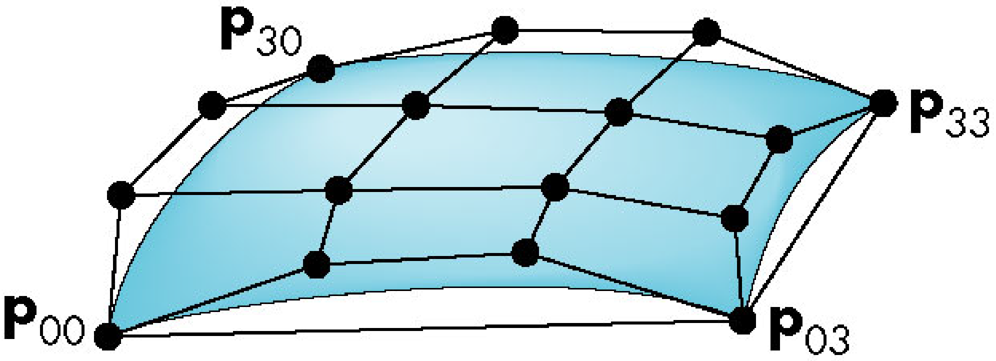
two blending functions

$$\mathbf{p}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} b_i(u)b_j(u)\mathbf{p}_{ij}$$

$$= \mathbf{u}^T \mathbf{M}_B \mathbf{P} \mathbf{M}_B^T \mathbf{v}.$$

Convex Hull Property in 3D

- The patch is inside the convex hull of the control points and interpolates the four corner points $\mathbf{p}_{00}$, $\mathbf{p}_{03}$, $\mathbf{p}_{30}$, $\mathbf{p}_{33}$.



Bezier Patch Edges

- Partial derivatives in the *u* and *v* directions treat the edges of the patch as 1D curves.

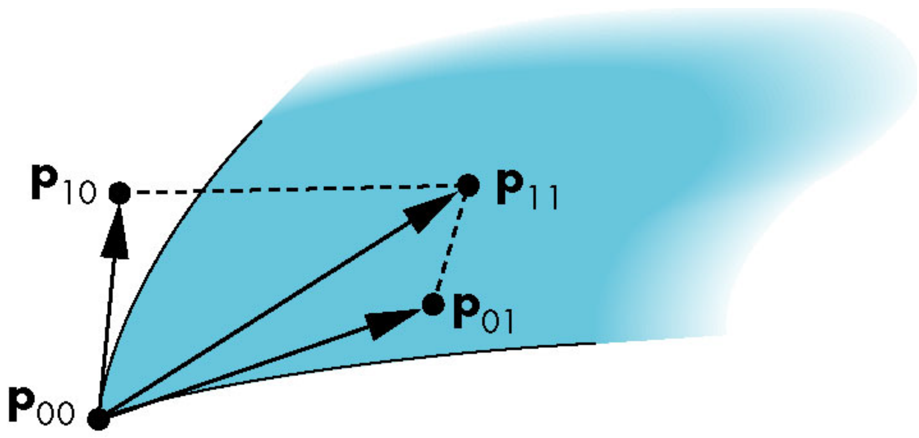
$$\frac{\partial \mathbf{p}}{\partial u}(0,0) = 3(\mathbf{p}_{10} - \mathbf{p}_{00}),$$

$$\frac{\partial \mathbf{p}}{\partial v}(0,0) = 3(\mathbf{p}_{01} - \mathbf{p}_{00}).$$

Bezier Patch Corners

- The **twist** vector draws the center of the patch away from the plane.

$$\frac{\partial^2 \mathbf{p}}{\partial u \partial v}(0,0) = 9(\mathbf{p}_{00} - \mathbf{p}_{01} + \mathbf{p}_{10} - \mathbf{p}_{11}).$$



Cubic B-Splines

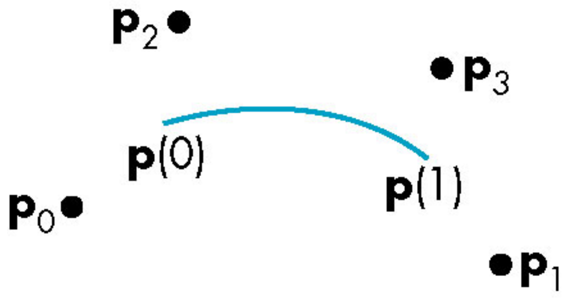
- Bezier curves and surfaces are widely used.
  - One limitation: C^0 continuity at the join points.
- **B-Splines** are not required to interpolate any control points.
  - Relaxing this requirement makes it possible to enforce greater smoothness at join points.

The Cubic B-Spline Curve

- The control points now reside in the middle of a sequence:

$$\{\mathbf{p}_{i-2}, \mathbf{p}_{i-1}, \mathbf{p}_{i}, \mathbf{p}_{i+1}\}.$$

  - The curve spans only the distance between the middle two control points.



Formulating the Geometry Matrix

- We are looking for a polynomial

$$\mathbf{p}(u) = \mathbf{u}^T \mathbf{M} \mathbf{p},$$

where **p** is the matrix of control points.

  - **M** can be made to enforce a number of conditions.
  - In particular, we can impose continuity requirements at the join points.

Join Point Continuity

- Construct **q** from the same matrix as **p**:

$$\mathbf{p} = \begin{bmatrix} \mathbf{p}_{i-2} \\ \mathbf{p}_{i-1} \\ \mathbf{p}_{i} \\ \mathbf{p}_{i+1} \end{bmatrix} \quad \text{and} \quad \mathbf{q} = \begin{bmatrix} \mathbf{p}_{i-3} \\ \mathbf{p}_{i-2} \\ \mathbf{p}_{i-1} \\ \mathbf{p}_{i} \end{bmatrix}.$$

  - Now let $\mathbf{q}(u) = \mathbf{u}^T\mathbf{M}\mathbf{q}$.
  - Constraints on derivates allow us to control smoothness.

Symmetric Approximations

- Enforcing symmetry at the join points is a popular choice for **M**.
- Two conditions that satisfy symmetry are

$$\mathbf{p}(0) = \mathbf{q}(1) = \frac{1}{6}(\mathbf{p}_{i-2} + 4\mathbf{p}_{i-1} + \mathbf{p}_i),$$

$$\mathbf{p}'(0) = \mathbf{q}'(1) = \frac{1}{2}(\mathbf{p}_i - \mathbf{p}_{i-2}),$$

Additional Conditions

- We apply the same symmetry conditions to **p**(1), the other endpoint.
  - We now have four equations in the four unknowns $\mathbf{c}_0$, $\mathbf{c}_1$, $\mathbf{c}_2$, $\mathbf{c}_3$:

$$\mathbf{p}(u) = \mathbf{u}^T \mathbf{c}.$$

The B-Spline Geometry Matrix

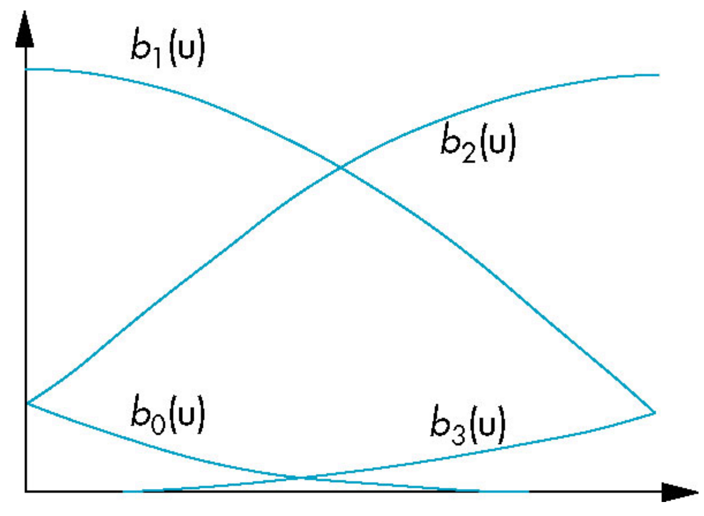
- Once we have the coefficient matrix, we can solve for the geometry matrix:

$$\mathbf{M}_S = \frac{1}{6}\begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}.$$

B-Spline Blending Functions

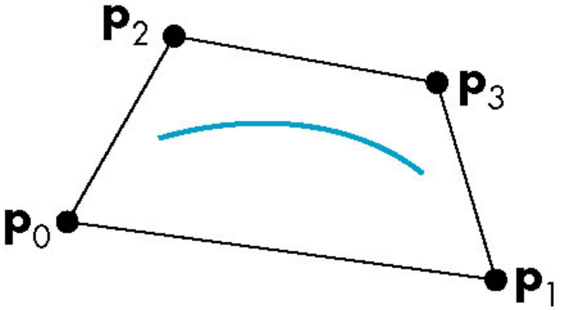
- The blending functions are

$$\frac{1}{6}\begin{bmatrix} (1-u)^3 \\ 4-6u^2+3u^3 \\ 1+3u+3u^2-3u^3 \\ u^3 \end{bmatrix}$$



Advantages of B-spline Curves

- In sequence, B-spline curve segments have C^2 continuity at the join points.
  - They are also confined to their convex hulls.



- On the other hand, we need more control points than we did for Bezier curves.

B-Splines and Bases

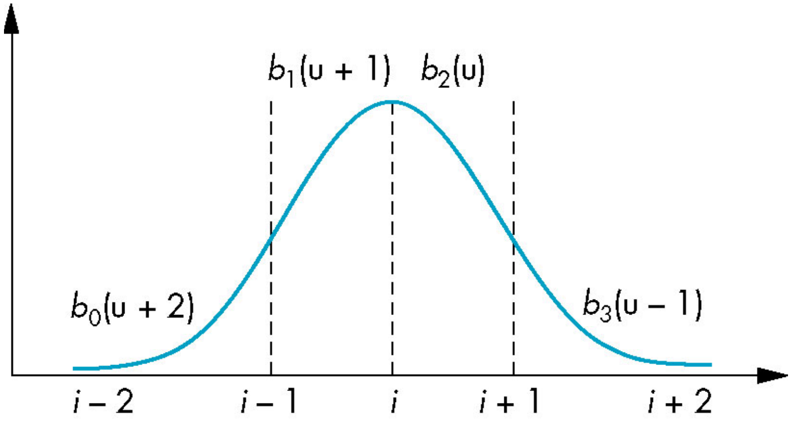
- Each control point affects four adjacent intervals.

$$B_i(u) = \begin{cases} 0 & u < i-2, \\ b_0(u+2) & i-2 \le u < i-1, \\ b_1(u+1) & i-1 \le u < i, \\ b_2(u) & i \le u < i+1, \\ b_3(u-1) & i+1 \le u < i+2, \\ 0 & u \ge i+2. \end{cases}$$

Spline Basis Function

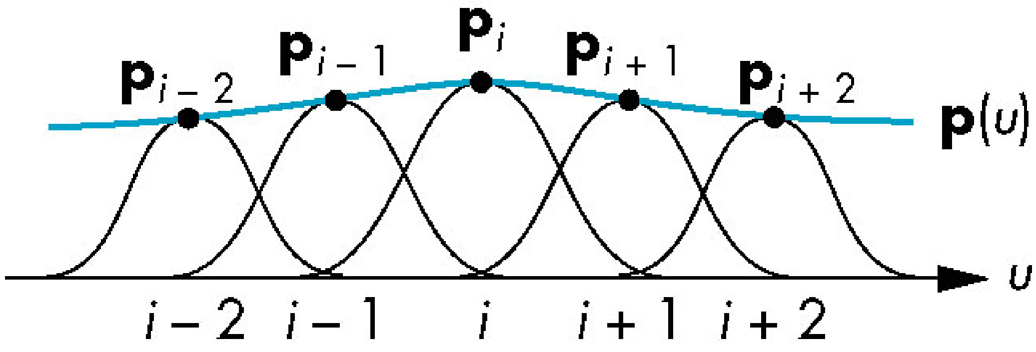
- A single expression for the spline curve using basis functions:

$$\mathbf{p}(u) = \sum_{i=1}^{m-1} B_i(u)\mathbf{p}_i .$$



Approximating Splines

- Each B_i is a shifted version of a single function.
  - Linear combinations of the B_i form a piecewise polynomial curve over the whole interval.

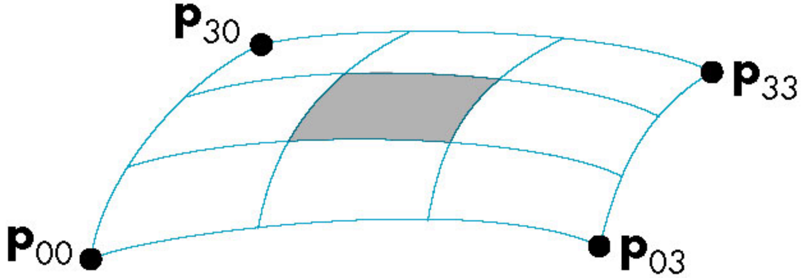


Spline Surfaces

- The same form as Bezier surfaces:

$$\mathbf{p}(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u) b_j(u) \mathbf{p}_{ij}.$$

  - But one segment per patch, instead of nine!



  - However, they are also much smoother.

General B-Splines

- Polynomials of degree *d* between *n* knots $u_0,\ldots,u_n$:

$$\mathbf{p}(u) = \sum_{j=0}^{d} \mathbf{c}_{jk} u^j, \quad u_k < u < u_{k+1}$$

  - If $d = 3$, then each interval contains a cubic polynomial: $4n$ equations in $4n$ unknowns.
  - A global solution that is not well-suited to computer graphics.

The Cox-deBoor Recursion

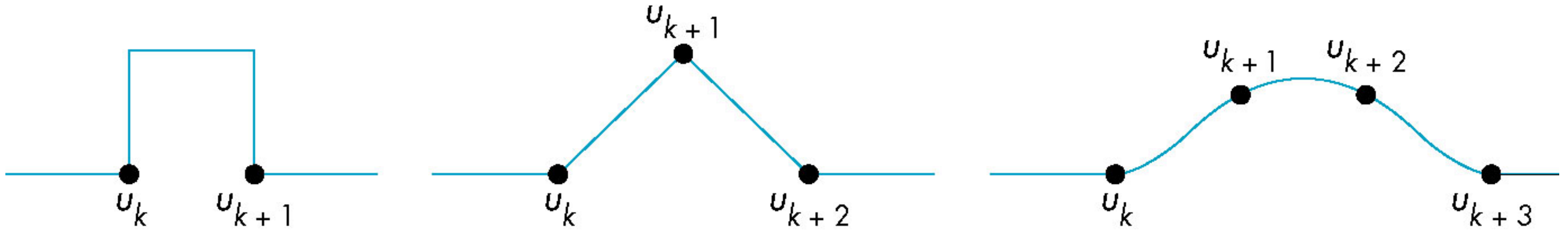
- A particular set of basis splines is defined by the **Cox-deBoor recursion**:

$$B_{k0} = \begin{cases} 1 & u_k \le u \le u_{k+1}, \\ 0 & \text{otherwise}; \end{cases}$$

$$B_{kd} = \frac{u - u_k}{u_{k+d} - u_k} B_{k,d-1}(u) + \frac{u_{k+d} - u}{u_{k+d+1} - u_{k+1}} B_{k+1,d-1}(u).$$

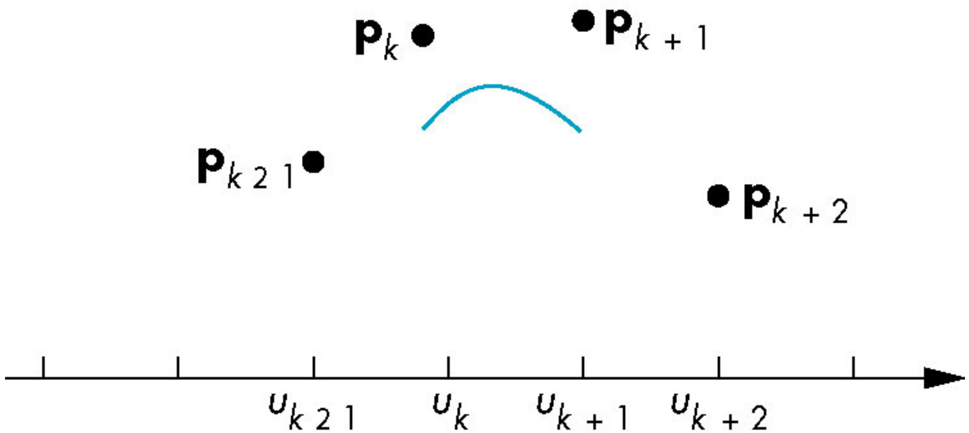
Recursively Defined B-Splines

- Linear interpolation of polynomials of degree *k* produces polynomials of degree *k* + 1.

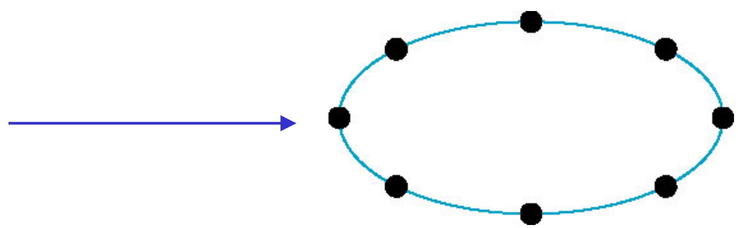


Uniform Splines

- Equally spaced knots.



periodic
uniform
B-spline



Nonuniform B-Splines

- Repeated knots pull the spline closer to the control point.
  - **Open splines** extend the curve by repeating the endpoints.
  - Knot sequences:

$$\{0,0,0,0,1,2,\ldots,n-1,n,n,n,n\} \longleftarrow \text{often used}$$

$$\{0,0,0,0,1,1,1,1\}. \longleftarrow \text{cubic Bezier curve}$$

  - Any spacing between the knots is allowed in the general case.

NURBS

- Use weights to increase or decrease the importance of a particular point.
  - The weighted homogeneous-coordinate representation of a control point $\mathbf{p}_i=[x_i\ y_i\ z_i]$ is

$$\mathbf{q}_i = w_i \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix}.$$

The NURBS Basis Functions

- A 4D B-spline

$$\mathbf{q}(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix} = \sum_{i=0}^{n} B_{i,d}(u) w_i \mathbf{p}_i .$$

- Derive the *w* component from the weights:

$$w(u) = \sum_{i=0}^{n} B_{i,d}(u) w_i .$$

Nonuniform Rational B-Splines

- Each component of **p**(*u*) is a rational function in *u*.
  - We use perspective division to recover the 3D points:

$$\mathbf{p}(u) = \frac{1}{w(u)}\mathbf{q}(u) = \frac{\sum_{i=0}^{n} B_{i,d}(u) w_i \mathbf{p}_i}{\sum_{i=0}^{n} B_{i,d}(u) w_i}.$$

  - These curves are invariant under perspective transformations.
  - They can approximate quadrics—one representation for all types of curves.

Rendering Curves and Surfaces

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Introduce methods to draw curves
  - Approximate with lines
  - Finite Differences
- Derive the recursive method for evaluation of Bezier curves and surfaces
- Learn how to convert all polynomial data to data for Bezier polynomials

Evaluating Polynomials

- Simplest method to render a polynomial curve is to evaluate the polynomial at many points and form an approximating polyline
- For surfaces we can form an approximating mesh of triangles or quadrilaterals
- Use Horner's method to evaluate polynomials

$$p(u)=c_0+u(c_1+u(c_2+uc_3))$$

- 3 multiplications/evaluation for cubic

Polynomial Evaluation Methods

- Our standard representation:

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{c}_i u^i, \quad 0 \le u \le 1$$

- Horner's method:

$$\mathbf{p}(u) = \mathbf{c}_0 + u(\mathbf{c}_1 + u(\mathbf{c}_2 + u(\ldots + \mathbf{c}_n u))).$$

  - If the points $\{u_i\}$ are spaced uniformly, we can use the method of **forward differences**.

The Method of Forward Differences

- Forward differences defined iteratively:

$$\Delta^{(0)}\mathbf{p}(u_k) = \mathbf{p}(u_k),$$

$$\Delta^{(1)}\mathbf{p}(u_k) = \mathbf{p}(u_{k+1}) - \mathbf{p}(u_k),$$

$$\Delta^{(m+1)}\mathbf{p}(u_k) = \Delta^{(m)}\mathbf{p}(u_{k+1}) - \Delta^{(m)}\mathbf{p}(u_k).$$

- If $u_{k+1} - u_k = h$ is constant, then $\Delta^{(n)}\mathbf{p}(u_k)$ is constant for all k.

Computing The Forward-Difference Table

- For the cubic polynomial

$$p(u) = 1 + 3u + 2u^2 + u^3,$$

we construct the table as follows:

t	0	1	2	3	4	5
p	1	7	23	55	109	191
$D^{(1)}$**p**	6	16	32	54	82	
$D^{(2)}$**p**	10	16	22	28		
$D^{(3)}$**p**	6	6	6			

← compute these

Using the Table

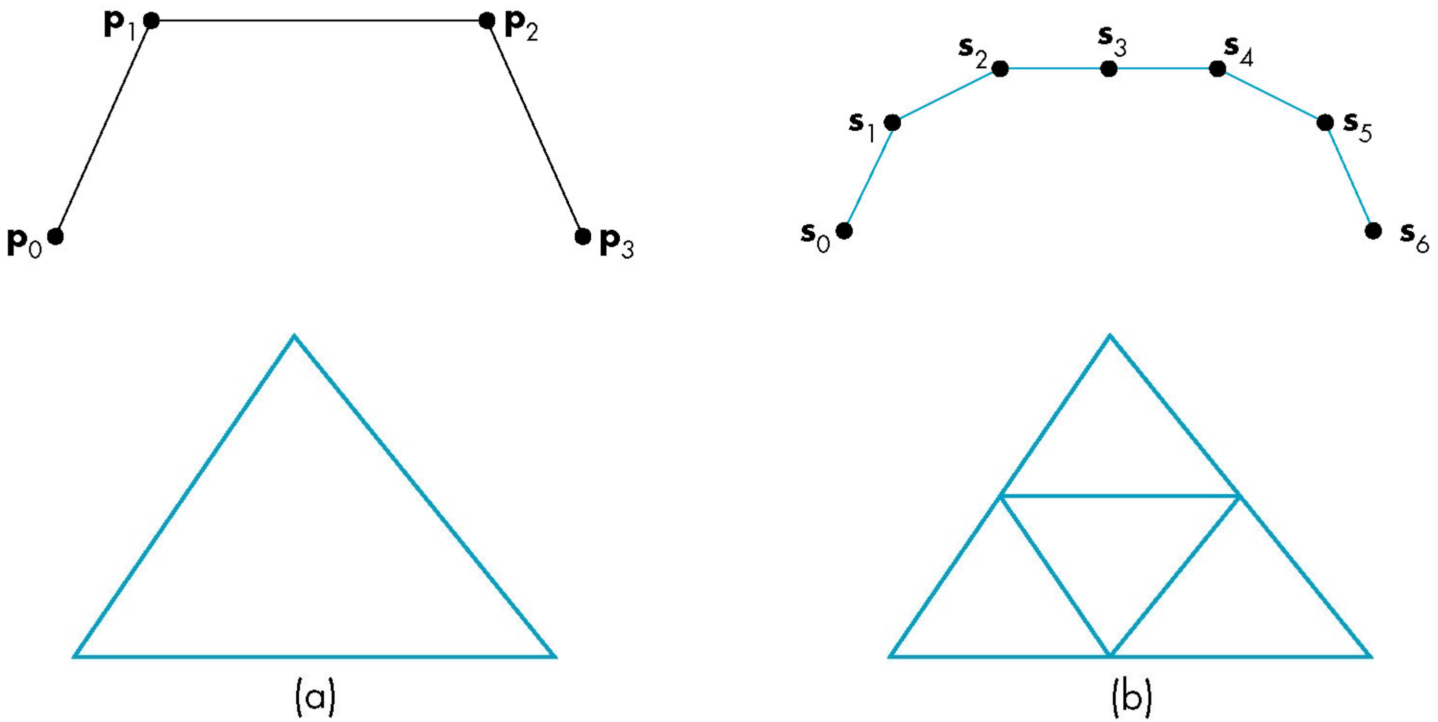
- Compute successive values of $p(u_k)$ starting from the bottom:

t	0	1	2	3	4	5
$\mathbf{p}$	1	7	23	55	109	191
$D^{(1)}\mathbf{p}$	6	16	32	54	82	
$D^{(2)}\mathbf{p}$	10	16	22	28		
$D^{(3)}\mathbf{p}$	6	6	6			

$$\Delta^{(m-1)}(p_{k+1}) = \Delta^{(m)} p(u_k) + \Delta^{(m-1)} p(u_k).$$

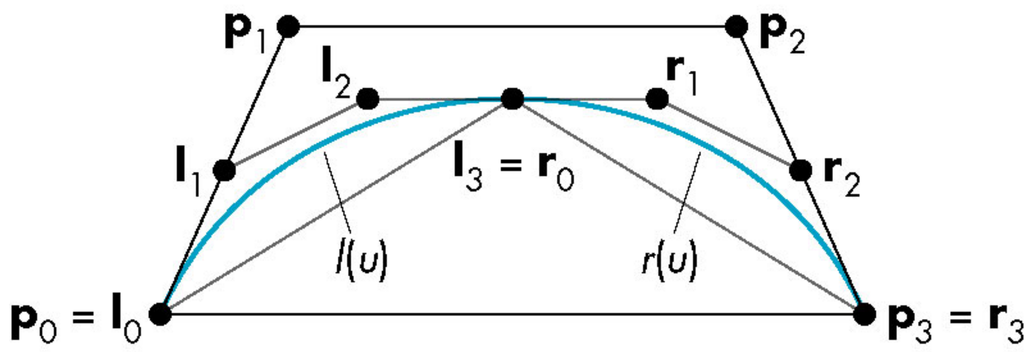
Subdivision Curves and Surfaces

- A process of iterative **refinement** that produces smooth curves and surfaces.



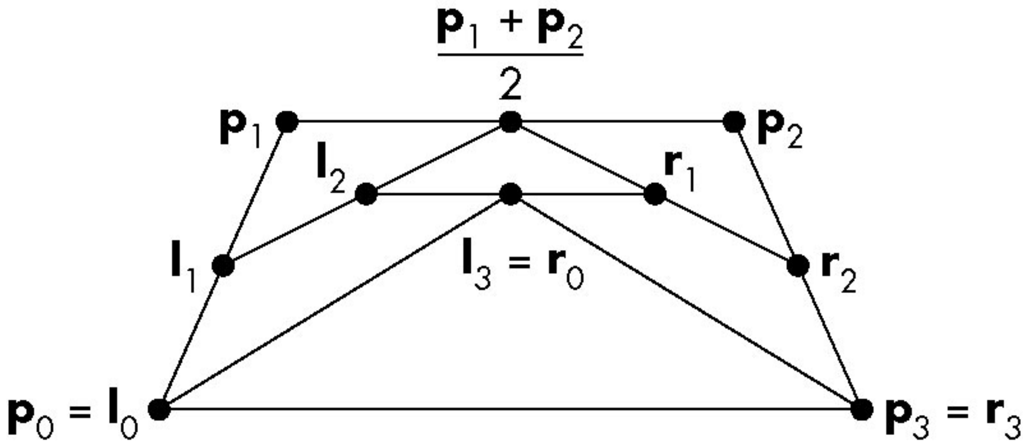
Recursive Subdivision of Bezier Polynomials: deCasteljau Algorithm

- Break the curve into two separate polynomials, **l**(u) and **r**(u).



  - The convex hulls for **l** and **r** must lie inside the convex hull for p: the **variation-diminishing property**:

Efficient Computation of the Subdivision



$$\mathbf{l}_0 = \mathbf{p}_0,$$

$$\mathbf{l}_1 = \frac{1}{2}(\mathbf{p}_0 + \mathbf{p}_1),$$

$$\mathbf{l}_2 = \frac{1}{2}\left(\mathbf{l}_1 + \frac{1}{2}(\mathbf{p}_1 + \mathbf{p}_2)\right),$$

$$\mathbf{l}_3 = \mathbf{r}_0 = \frac{1}{2}(\mathbf{l}_2 + r_1).$$

Requires only shifts and adds!

Every Curve is a Bezier Curve

- We can render a given polynomial using the recursive method if we find control points for its representation as a Bezier curve
- Suppose that p(u) is given as an interpolating curve with control points $\mathbf{q}$

$$p(u)=\mathbf{u}^T\mathbf{M}_I\mathbf{q}$$

- There exist Bezier control points $\mathbf{p}$ such that

$$p(u)=\mathbf{u}^T\mathbf{M}_B\mathbf{p}$$

- Equating and solving, we find $\mathbf{p}=\mathbf{M}_B^{-1}\mathbf{M}_I$

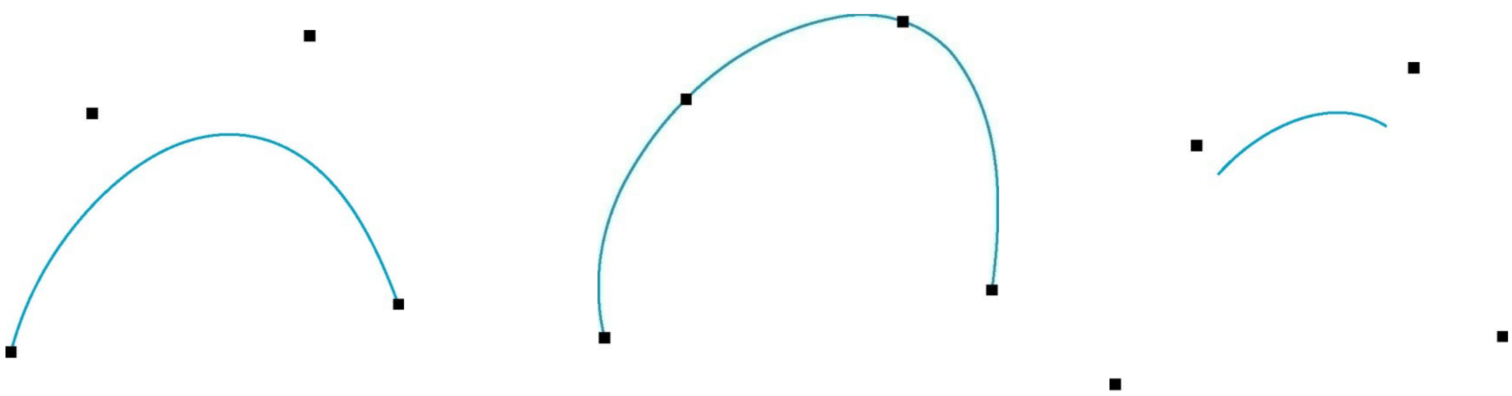
Matrices

Interpolating to Bezier $$\mathbf{M}_B^{-1}\mathbf{M}_I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{5}{6} & 3 & -\frac{3}{2} & \frac{1}{3} \\ \frac{1}{3} & -\frac{3}{2} & 3 & -\frac{5}{6} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

B-Spline to Bezier $$\mathbf{M}_B^{-1}\mathbf{M}_S = \begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix}$$

Example

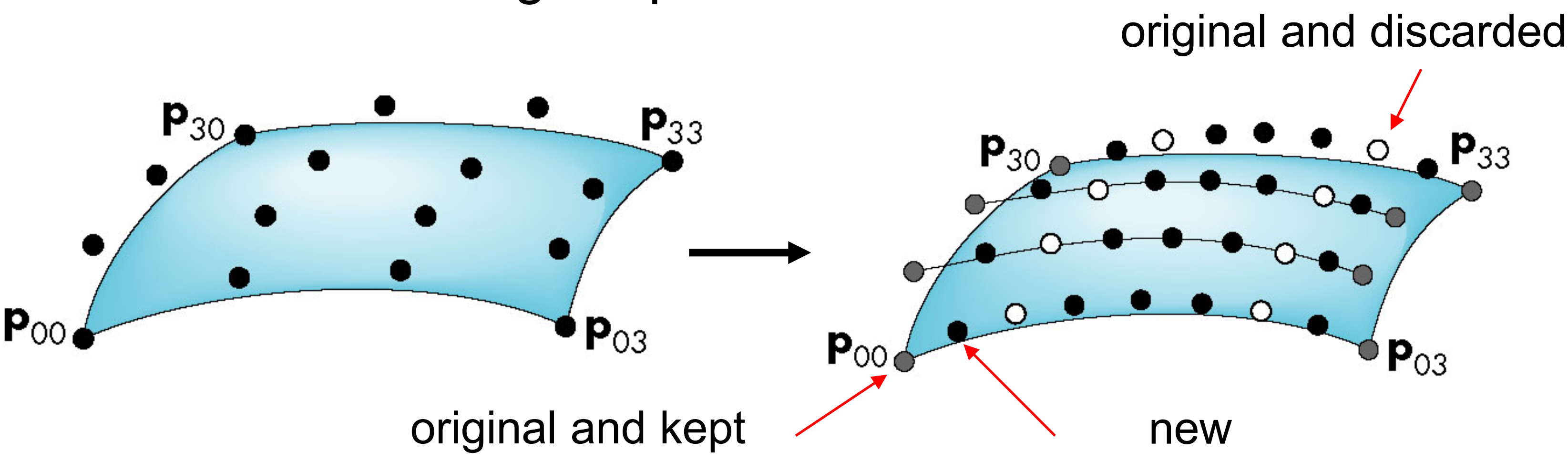
These three curves were all generated from the same original data using Bezier recursion by converting all control point data to Bezier control points



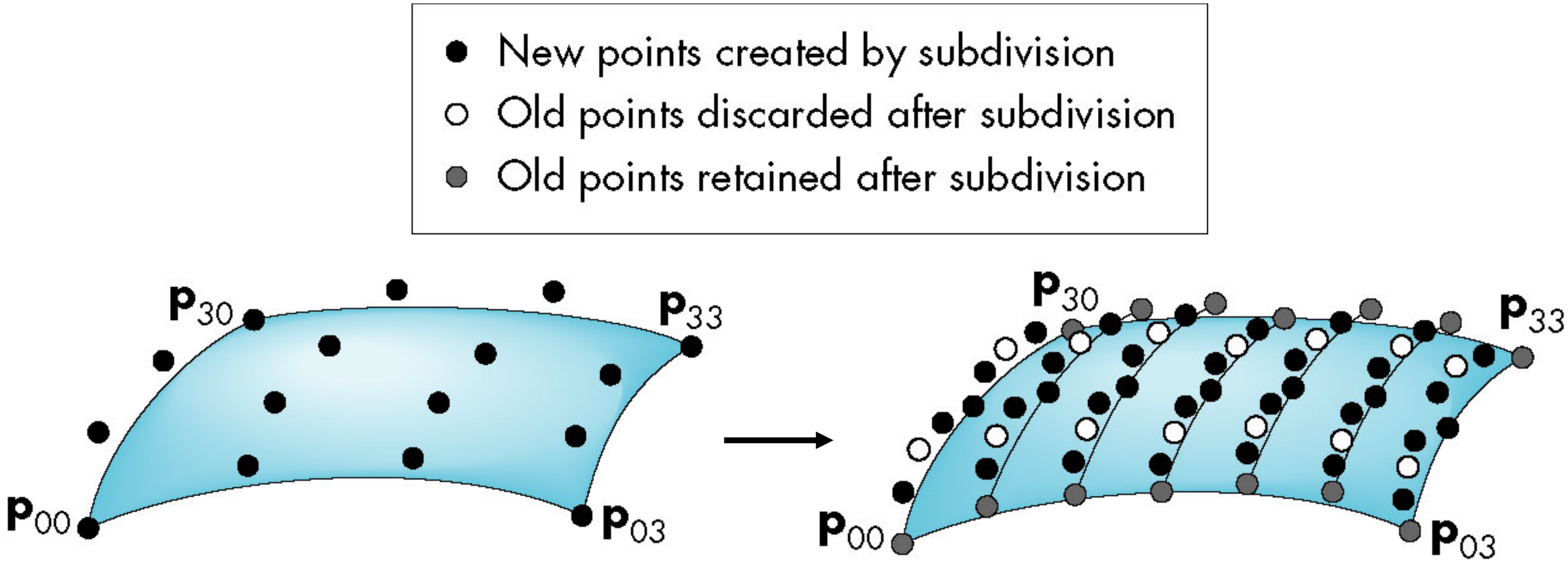
Bezier Interpolating B Spline

Surfaces

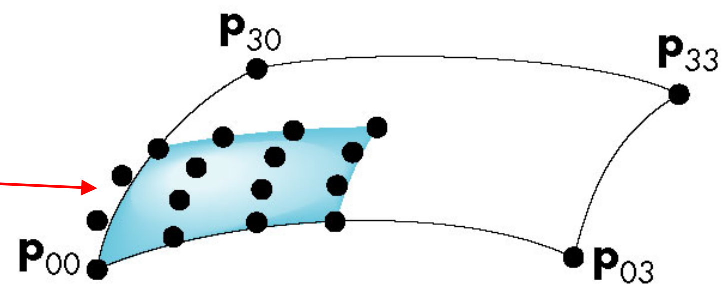
- Can apply the recursive method to surfaces if we recall that for a Bezier patch curves of constant u (or v) are Bezier curves in u (or v)
- First subdivide in u
  - Process creates new points
  - Some of the original points are discarded



Second Subdivision



16 final points for
1 of 4 patches created



Normals

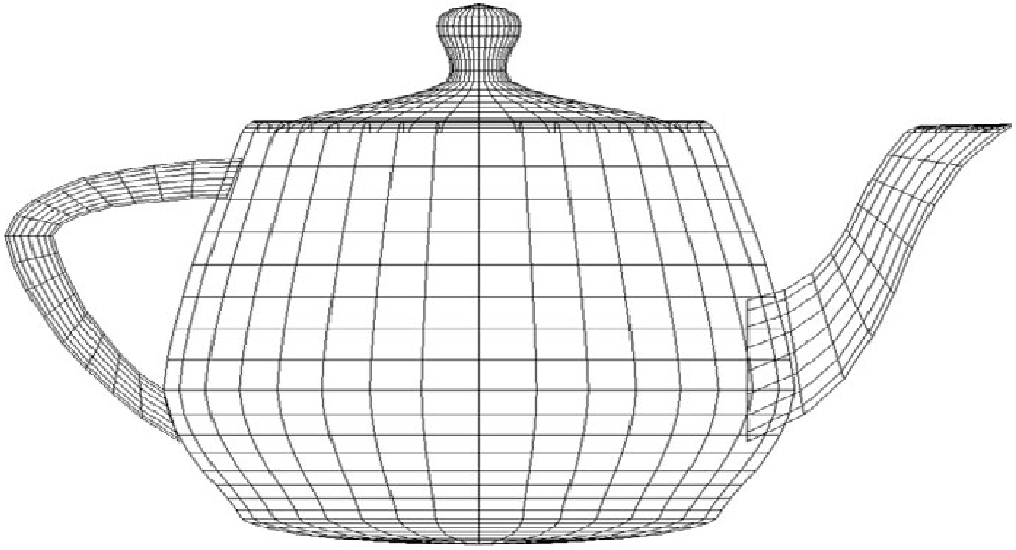
- For rendering we need the normals if we want to shade
  - Can compute from parametric equations

$$\mathbf{n} = \frac{\partial \mathbf{p}(u,v)}{\partial u} \times \frac{\partial \mathbf{p}(u,v)}{\partial v}$$

  - Can use vertices of corner points to determine
  - OpenGL can compute automatically

Utah Teapot

- Most famous data set in computer graphics
- Widely available as a list of 306 3D vertices and the indices that define 32 Bezier patches



Algebraic Surfaces

- **Quadric surfaces** are described by implicit equations of the form

$$\mathbf{p}^T \mathbf{A}\mathbf{p} + \mathbf{b}^T \mathbf{p} + c = 0.$$

  - 10 independent coefficients **A**, **b**, and *c* determine the quadric.
  - Ellipsoids, paraboloids, and hyperboloids can be created by different groups of coefficients.
  - Equations for quadric surfaces can be reduced to standard form by affine transformation.

Rendering Quadric Surfaces

- Finding the intersection of a quadric with a ray involves solving a scalar quadratic equation.
  - We substitute ray $\mathbf{p} = \mathbf{p}_0 + \alpha\mathbf{d}$ and use the quadratic formula.
  - Derivatives determine the normal at a given point.

Quadric Objects in OpenGL

- OpenGL supports disks, cylinders and spheres with quadric objects.

```
GLUquadricObj *qobj;
qobj = gluNewQuadric();
```

  - Choose wire frame rendering with

```
gluQuadricDrawStyle(qobj, GLU_LINE);
```

  - To draw an object, pass the reference:

```
gluSphere(qobj, RADIUS, SLICES, STACKS);
```

Bezier Curves in OpenGL

- Creating a 1D evaluator:

```
glMap1f(type, u_min, u_max, stride,
        order, point_array);
```

  - **`type`**: points, colors, normals, textures, etc.
  - **`u_min`**, **`u_max`**: range.
  - **`stride`**: points per curve segment.
  - **`order`**: degree + 1.
  - **`point_array`**: control points.

Drawing the Curve

- One evaluator call takes the place of vertex, color, and normal calls.
  - The user enables them with **`glEnable`**.

```
typedef float point[3];
point data[] = {...};
glMap1f(GL_MAP_VERTEX_3, 0.0, 1.0, 3, 4, data);
glEnable(GL_MAP_VERTEX_3);

glBegin(GL_LINE_STRIP)
for(i=0; i<100; i++) glEvalCoord1f(i/100.);
glEnd();
```

Bezier Surfaces in OpenGL

- Using a 2D evaluator:

```
glMap2f(GL_MAP_VERTEX_3,0,1,3,4,0,1,12,4,data);
...
for(j=0; j<99; j++) {
      glBegin(GL_QUAD_STRIP);
      for(i=0; i<=100; i++) {
            glEvalCoord2f(i/100., j/100.);
            glEvalCoord2f((i+1)/100., j/100.);
      }
      glEnd();
}
```

Example: Bezier Teapot

- Vertex information goes in an array:

```
GLfloat data[32][4][4];
```

- Initialize the grid for wireframe rendering:

```
void myInit() {
     glEnable(GL_MAP2_VERTEX_3);
     glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
}
```

Drawing the Teapot

```
for(k=0; k<32; k++) {
      glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
              0, 1, 12, 4, &data[k][0][0][0]);
      for (j=0; j<=8; j++) {
            glBegin(GL_LINE_STRIP);
            for (i=0; i<=30; i++)
                 glEvalCoord2f((GLfloat)i/30.0,
                               (GLfloat)j/8.0);
            glEnd();
            glBegin(GL_LINE_STRIP);
            for (i=0; i<=30; i++)
               glEvalCoord2f((GLfloat)j/8.0,
                             (GLfloat)i/30.0);
            glEnd();
      }
   }
```