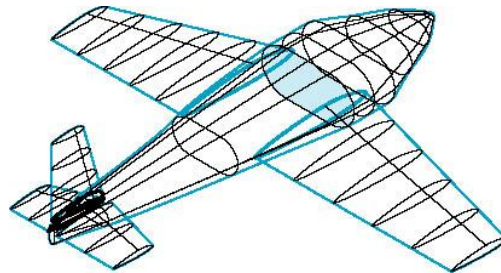


12. Curves and Surfaces



Overview

- Representation of Curves and Surfaces
- Designing Parametric Cubic Curves
- Bezier and Spline Curves and Surfaces
- Rendering Curves and Surfaces
- Curves and Surfaces in OpenGL
- Source programs
- Reading: ANG Ch.12

Representation of Curves and Surfaces

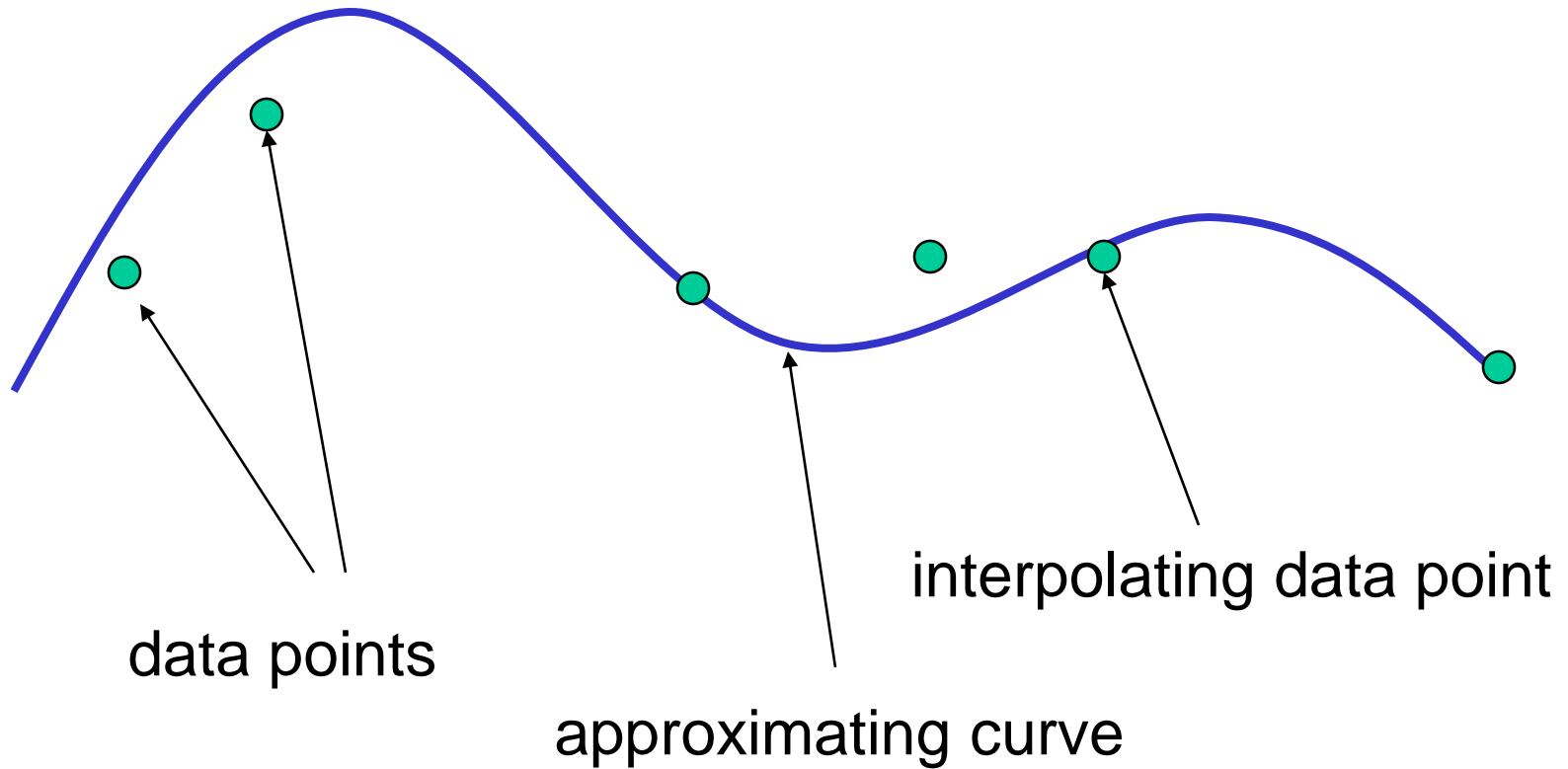
Objectives

- Introduce types of curves and surfaces
 - Explicit
 - Implicit
 - **Parametric**
 - Strengths and weaknesses
- Discuss Modeling and Approximations
 - Conditions
 - Stability

Escaping Flatland

- Until now we have worked with flat entities such as lines and flat polygons
 - Fit well with graphics hardware
 - Mathematically simple
- But the world is not composed of flat entities
 - Need curves and curved surfaces
 - May only have need at the application level
 - Implementation can render them approximately with flat primitives

Modeling with Curves



What Makes a Good Representation?

- There are many ways to represent curves and surfaces
- Want a representation that is
 - Stable
 - Smooth
 - Easy to evaluate
 - Must we interpolate or can we just come close to data?
 - Do we need derivatives?

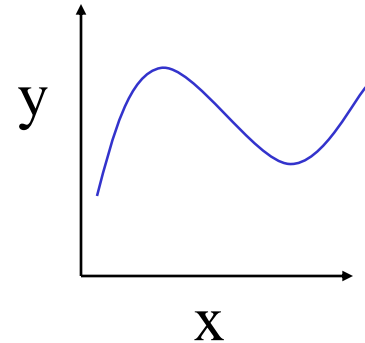
Explicit Representation

- Most familiar form of curve in 2D

$$y=f(x)$$

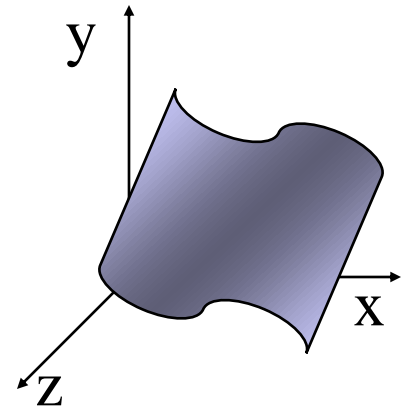
- **Cannot** represent all curves

- Vertical lines
- Circles



- Extension to 3D

- $y=f(x)$, $z=g(x)$
- The form $z = f(x,y)$ defines a surface



Implicit Representation

- Two dimensional curve(s)

$$g(x,y)=0$$

- Much more robust

- All lines $ax+by+c=0$

- Circles $x^2+y^2-r^2=0$

- Three dimensions $g(x,y,z)=0$ defines a surface

- Intersect two surface to get a curve

- In general, we *cannot solve for points that satisfy*

Algebraic Surface

$$\sum_i \sum_j \sum_k x^i y^j z^k = 0$$

- Quadric surface $2 \geq i+j+k$
- At most 10 terms
- Can solve **intersection with a ray** by reducing problem to **solving quadratic equation**

Parametric Curves

- Separate equation for each spatial variable

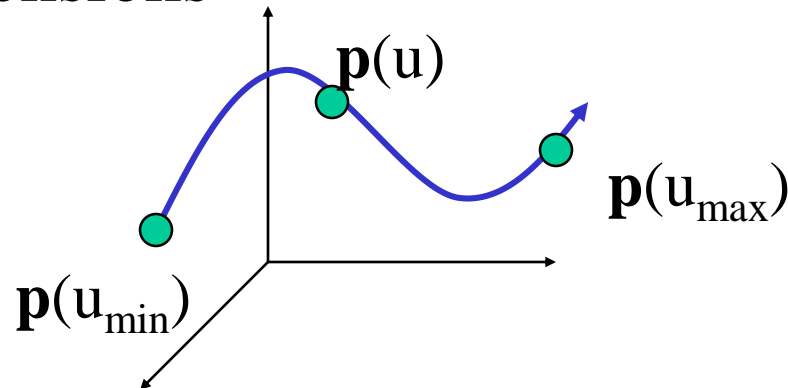
$$x=x(u)$$

$$y=y(u)$$

$$z=z(u)$$

$$\mathbf{p}(u)=[x(u), y(u), z(u)]^T$$

- For $u_{\max} \geq u \geq u_{\min}$ we trace out a curve in **two or three** dimensions



Selecting Functions

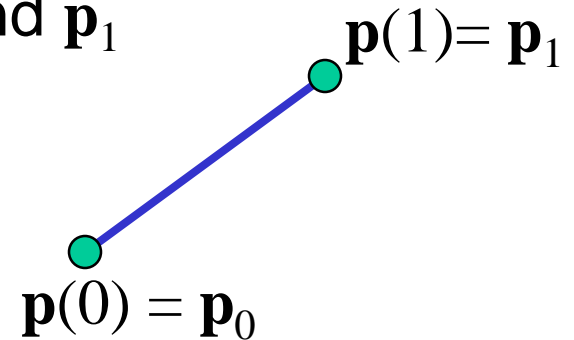
- Usually we can select “good” functions
 - not unique for a given spatial curve
 - Approximate or interpolate known data
 - Want functions which are easy to evaluate
 - Want functions which are easy to differentiate
 - Computation of normals
 - Connecting pieces (segments)
 - Want functions which are smooth

Parametric Lines

We can normalize u to be over the interval $(0,1)$

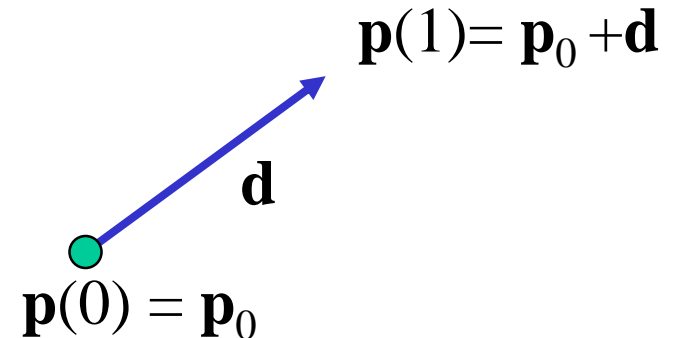
Line connecting two points \mathbf{p}_0 and \mathbf{p}_1

$$\mathbf{p}(u) = (1-u)\mathbf{p}_0 + u\mathbf{p}_1$$



Ray from \mathbf{p}_0 in the direction \mathbf{d}

$$\mathbf{p}(u) = \mathbf{p}_0 + u\mathbf{d}$$



Parametric Surfaces

- Surfaces require 2 parameters

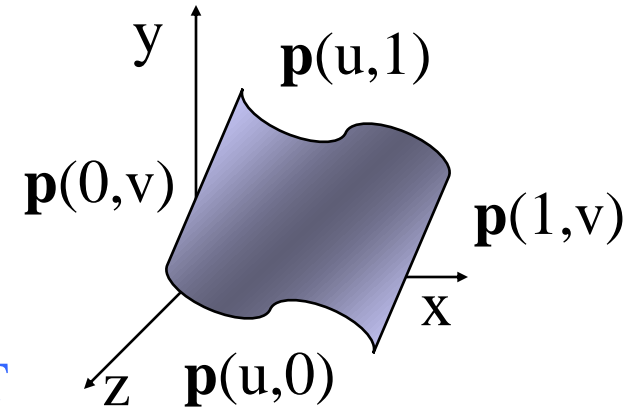
$$x=x(u,v)$$

$$y=y(u,v)$$

$$z=z(u,v)$$

$$\mathbf{p}(u,v) = [x(u,v), y(u,v), z(u,v)]^T$$

- Want same properties as curves:
 - Smoothness
 - Differentiability
 - Ease of evaluation



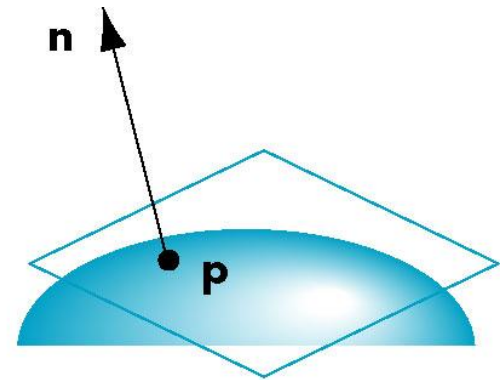
Normals

We can differentiate with respect to u and v to obtain the normal at any point \mathbf{p}

$$\frac{\partial \mathbf{p}(u, v)}{\partial u} = \begin{bmatrix} \partial x(u, v) / \partial u \\ \partial y(u, v) / \partial u \\ \partial z(u, v) / \partial u \end{bmatrix}$$

$$\frac{\partial \mathbf{p}(u, v)}{\partial v} = \begin{bmatrix} \partial x(u, v) / \partial v \\ \partial y(u, v) / \partial v \\ \partial z(u, v) / \partial v \end{bmatrix}$$

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



Parametric Planes

point-vector form

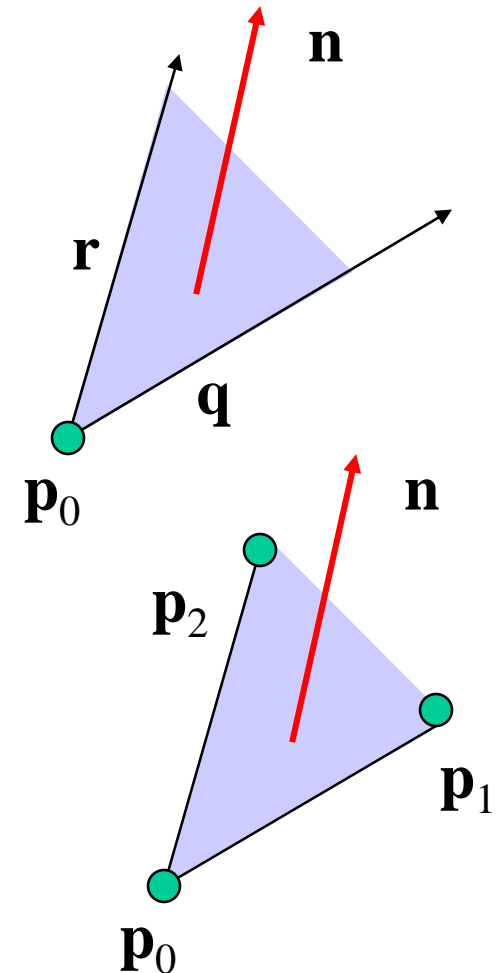
$$\mathbf{p}(u,v) = \mathbf{p}_0 + u\mathbf{q} + v\mathbf{r}$$

$$\mathbf{n} = \mathbf{q} \times \mathbf{r}$$

three-point form

$$\mathbf{q} = \mathbf{p}_1 - \mathbf{p}_0$$

$$\mathbf{r} = \mathbf{p}_2 - \mathbf{p}_0$$



Parametric Sphere

$$x(\theta, \varphi) = r \cos \theta \sin \varphi$$

$$y(\theta, \varphi) = r \sin \theta \sin \varphi$$

$$z(\theta, \varphi) = r \cos \varphi$$

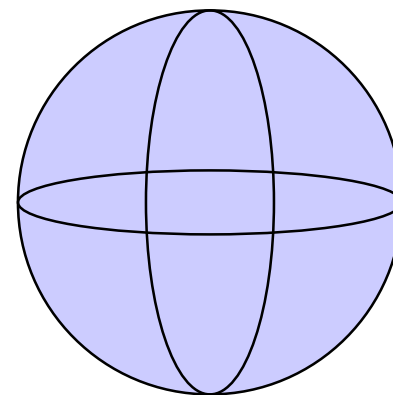
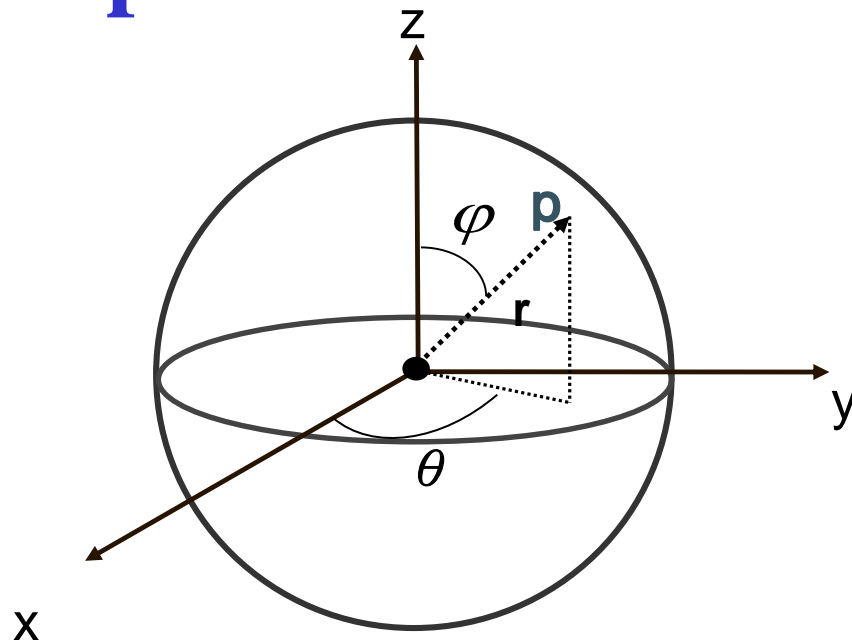
$$360 \geq \theta \geq 0$$

$$180 \geq \varphi \geq 0$$

θ constant: circles of constant **longitude**

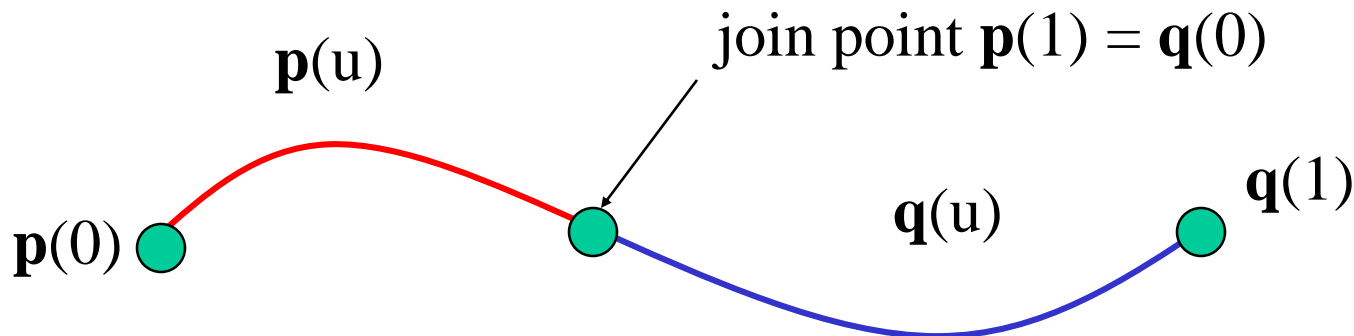
φ constant: circles of constant **latitude**

differentiate to show $\mathbf{n} = \mathbf{p}$

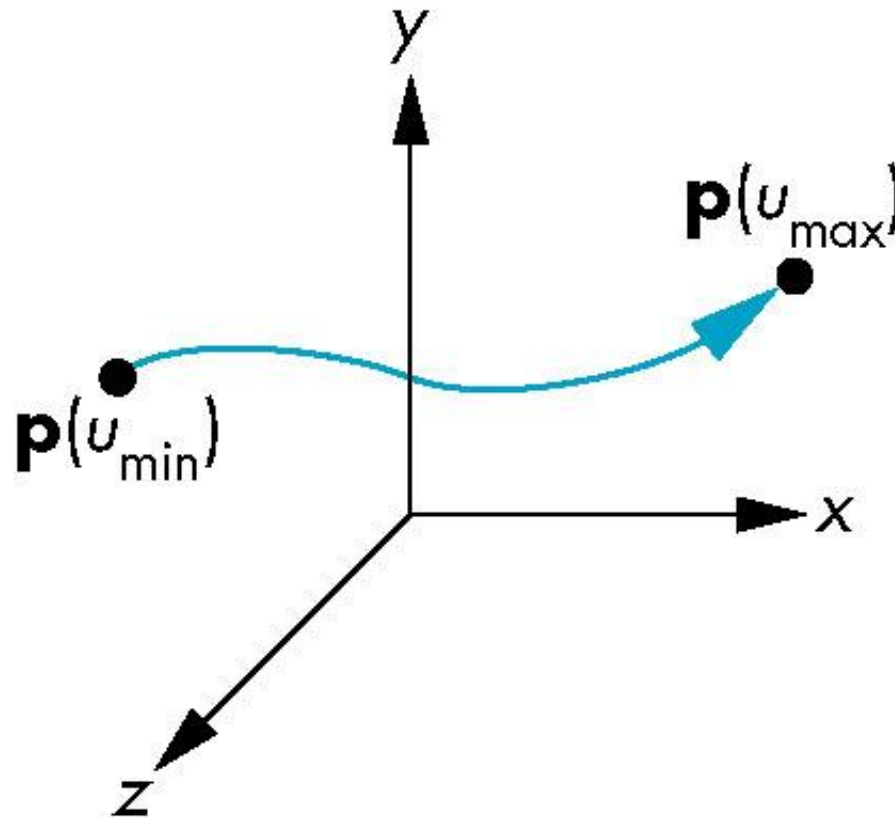


Curve Segments

- After normalizing u , each curve is written
 $\mathbf{p}(u)=[x(u), y(u), z(u)]^T, \quad 1 \geq u \geq 0$
- In classical numerical methods, we design a single global curve
- In computer graphics and CAD, it is **better** to design **small connected curve segments**



Curve Segments



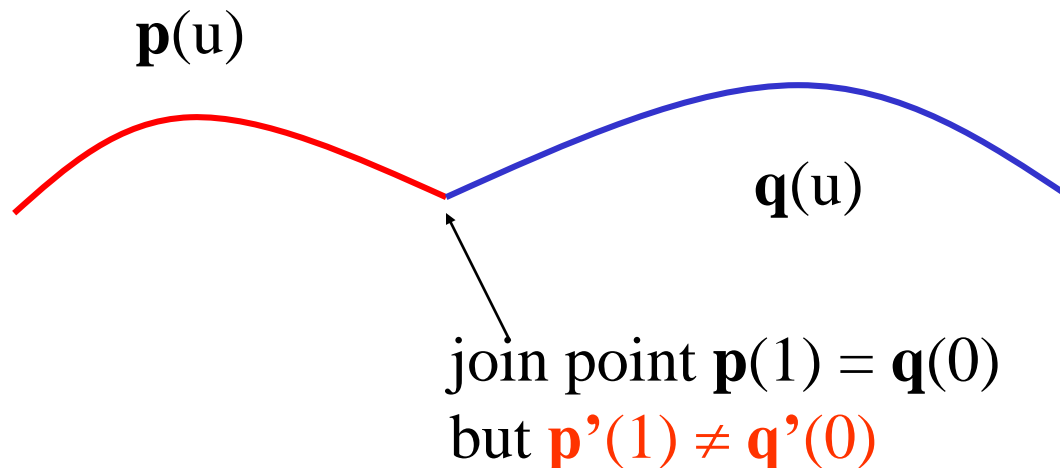
Parametric Polynomial Curves

$$x(u) = \sum_{i=0}^N c_{xi} u^i \quad y(u) = \sum_{j=0}^M c_{yj} u^j \quad z(u) = \sum_{k=0}^K c_{zk} u^k$$

- If $N=M=K$, we need to determine $3(N+1)$ coefficients
- Equivalently we need $3(N+1)$ independent conditions
- Noting that the curves for x, y and z are independent, we can define each independently in an identical manner
- We will use the form $p(u) = \sum_{k=0}^L c_k u^k$ where p can be any of x, y, z

Why Polynomials

- Easy to evaluate
- Continuous and differentiable everywhere
 - Must worry about continuity at join points including **continuity of derivatives**



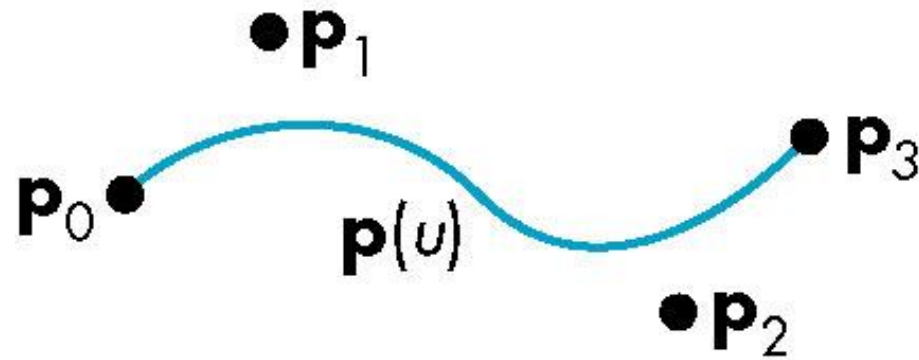
Cubic Parametric Polynomials

- $N=M=K=3$, gives balance between ease of evaluation and flexibility in design

$$p(u) = \sum_{k=0}^3 c_k u^k$$

- Four coefficients to determine for each of x, y and z
- Seek four independent conditions for various values of u resulting in 4 equations in 4 unknowns for each of x, y and z
 - Conditions are a mixture of continuity requirements at the join points and conditions for fitting the data

Curve Segments and Control Points



Cubic Polynomial Surfaces

$$\mathbf{p}(u,v)=[x(u,v), y(u,v), z(u,v)]^T$$

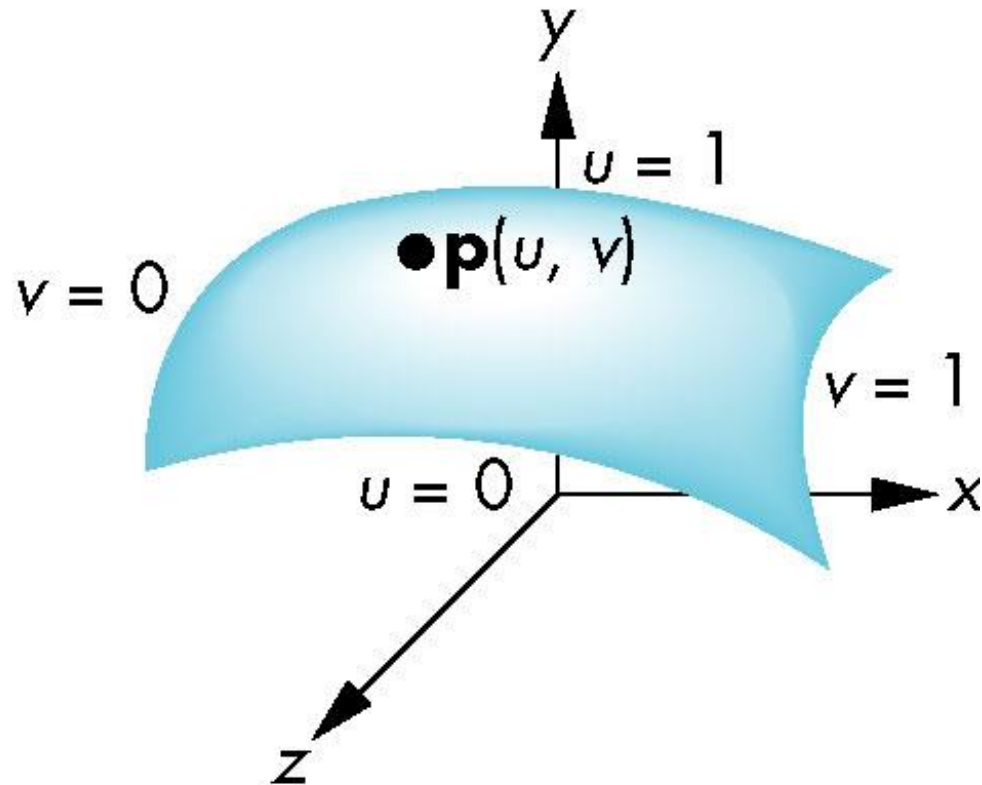
where

$$p(u,v) = \sum_{i=0}^3 \sum_{j=0}^3 c_{ij} u^i v^j$$

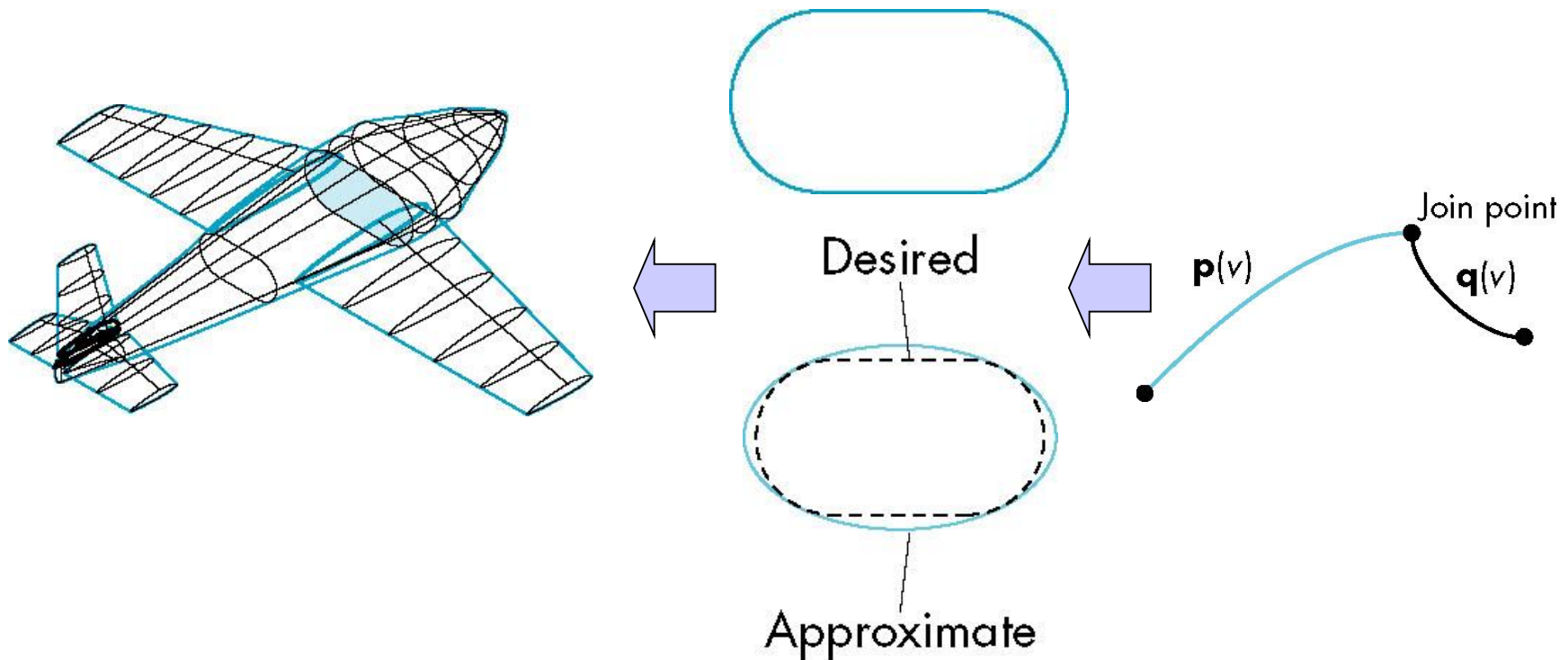
p is any of x, y or z

Need 48 coefficients (3 independent sets of 16) to determine a surface patch

Surface Patch



Designing Parametric Cubic Curves



Objectives

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

Matrix-Vector Form

$$p(u) = \sum_{k=0}^3 c_k u^k$$

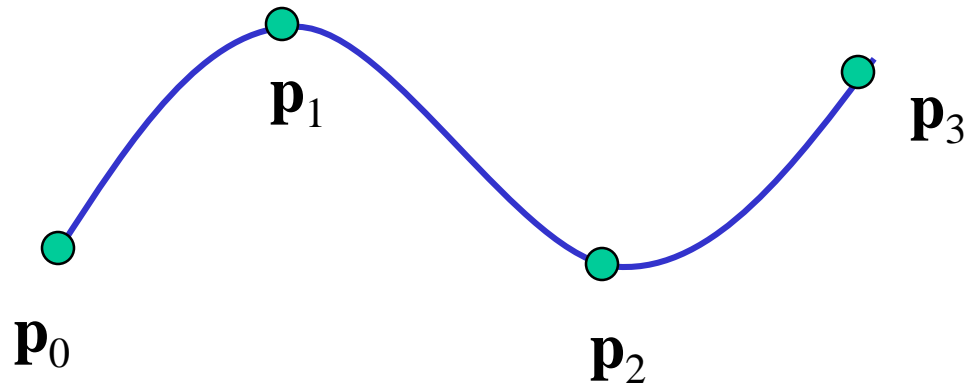
define

$$\mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}$$

then

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

Interpolating Curve



Given four data (control) points p_0 , p_1 , p_2 , p_3
determine cubic $p(u)$ which **passes through** them

Must find c_0 , c_1 , c_2 , c_3

Interpolation Equations

apply the interpolating conditions at $u=0, 1/3, 2/3, 1$

$$p_0 = p(0) = c_0$$

$$p_1 = p(1/3) = c_0 + (1/3)c_1 + (1/3)^2 c_2 + (1/3)^3 c_3$$

$$p_2 = p(2/3) = c_0 + (2/3)c_1 + (2/3)^2 c_2 + (2/3)^3 c_3$$

$$p_3 = p(1) = c_0 + c_1 + c_2 + c_3$$

or in matrix form with $\mathbf{p} = [p_0 \ p_1 \ p_2 \ p_3]^T$

$$\mathbf{p} = \mathbf{A}\mathbf{c} \quad \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & \left(\frac{1}{3}\right) & \left(\frac{1}{3}\right)^2 & \left(\frac{1}{3}\right)^3 \\ 1 & \left(\frac{2}{3}\right) & \left(\frac{2}{3}\right)^2 & \left(\frac{2}{3}\right)^3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Interpolation Matrix

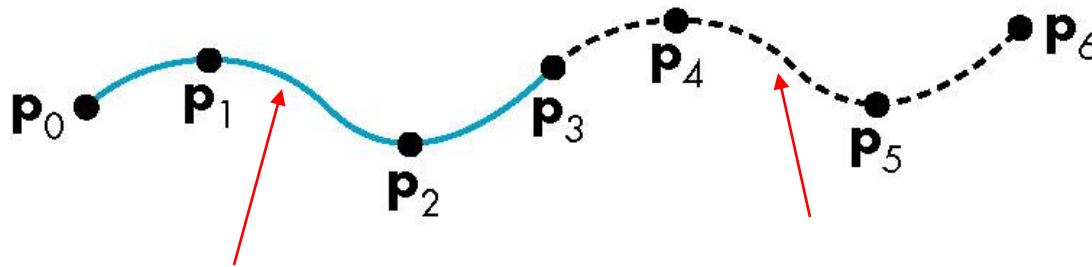
Solving for **c** we find the *interpolation matrix*

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

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

Note that \mathbf{M}_I **does not depend on** input data and can be used for each segment in x, y, and z

Interpolating Multiple Segments



use $\mathbf{p} = [p_0 \ p_1 \ p_2 \ p_3]^T$

use $\mathbf{p} = [p_3 \ p_4 \ p_5 \ p_6]^T$

Get continuity at join points but not continuity of derivatives

Blending Functions

Rewriting the equation for $p(u)$

(where $\mathbf{c} = \mathbf{M}_I \mathbf{p}$ and \mathbf{P} are fitting points)

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

where $\mathbf{b}(u) = [b_0(u) \ b_1(u) \ b_2(u) \ b_3(u)]^T$ is an array of *blending polynomials* such that

$$p(u) = b_0(u)p_0 + b_1(u)p_1 + b_2(u)p_2 + b_3(u)p_3$$

$$b_0(u) = -4.5(u-1/3)(u-2/3)(u-1)$$

$$b_1(u) = 13.5u(u-2/3)(u-1)$$

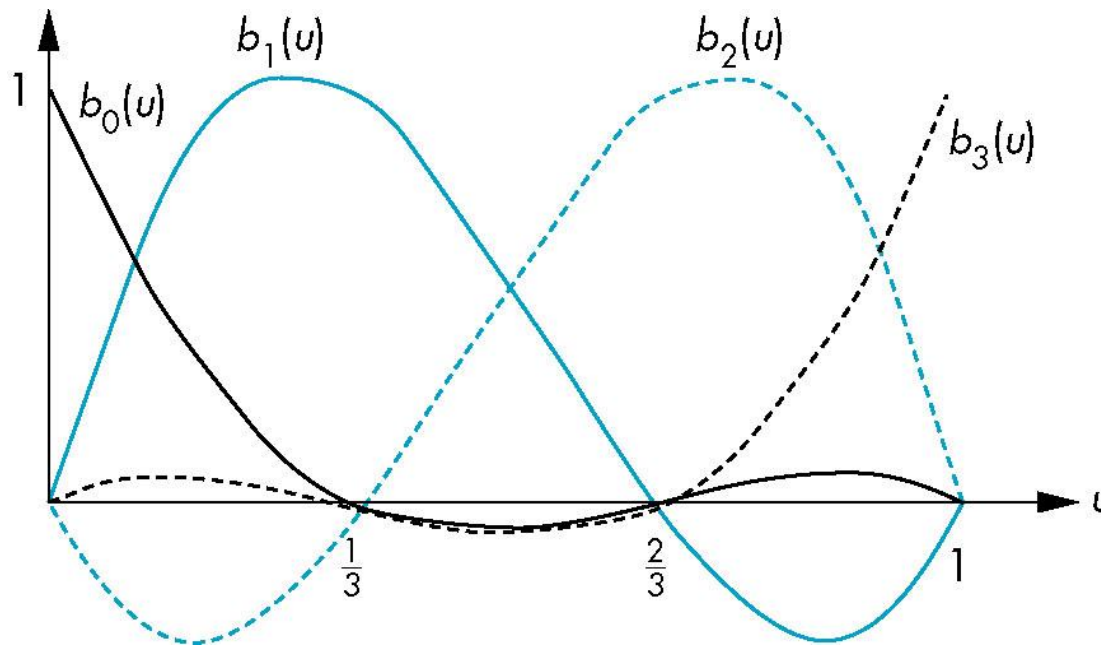
$$b_2(u) = -13.5u(u-1/3)(u-1)$$

$$b_3(u) = 4.5u(u-1/3)(u-2/3)$$

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

Blending Functions

- These functions are not smooth
 - Hence the interpolation polynomial is not smooth

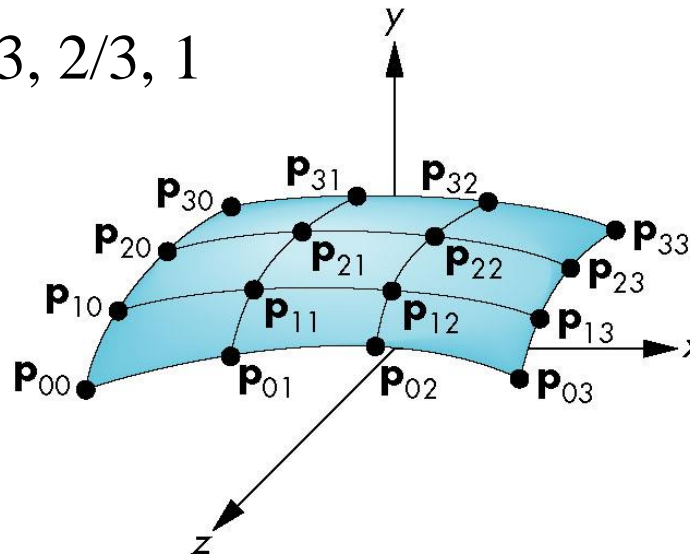


Interpolating Patch

$$p(u, v) = \sum_{i=0}^3 \sum_{j=0}^3 c_{ij} u^i v^j$$

Need **16 conditions** to determine the **16 coefficients** c_{ij}

Choose at $u, v = 0, 1/3, 2/3, 1$



Matrix Form

Define $\mathbf{v} = [1 \ v \ v^2 \ v^3]^T$

$$\mathbf{C} = [c_{ij}] \quad \mathbf{P} = [p_{ij}]$$

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

If we observe that for constant u (v), we obtain interpolating curve in v (u), we can show

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

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

Blending Patches

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

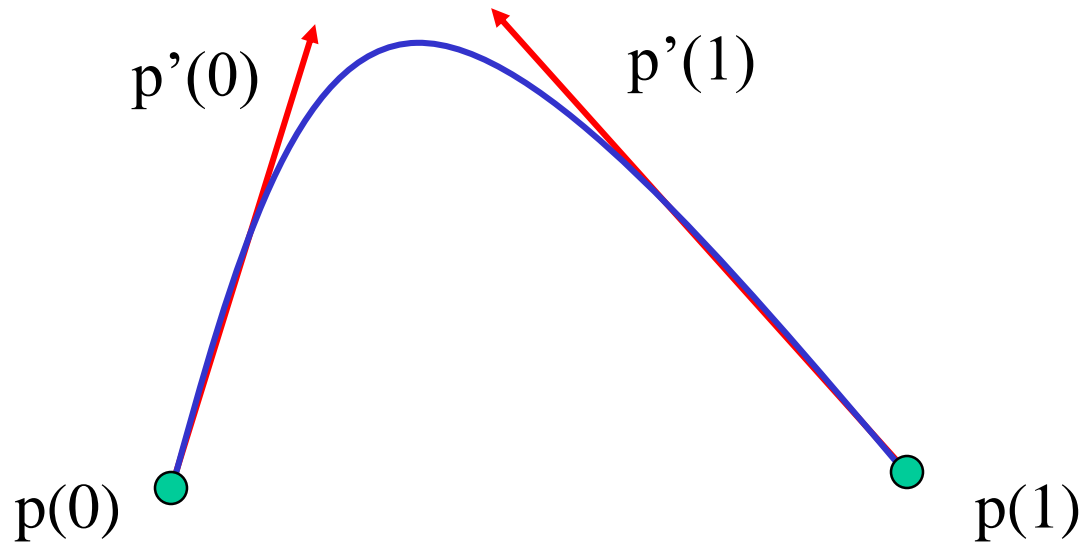
Each $b_i(u)b_j(v)$ is a blending patch

Show that we can build and analyze surfaces from our knowledge of curves

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

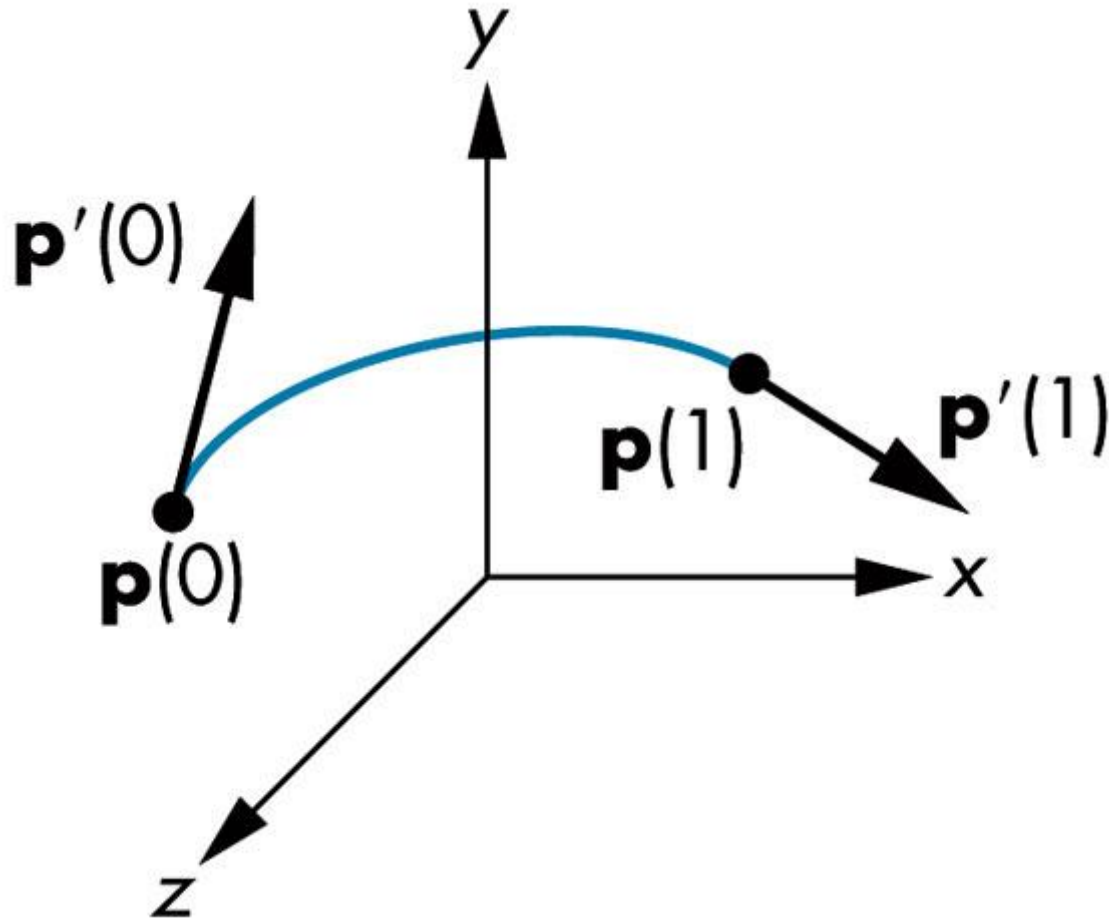
Hermite Form



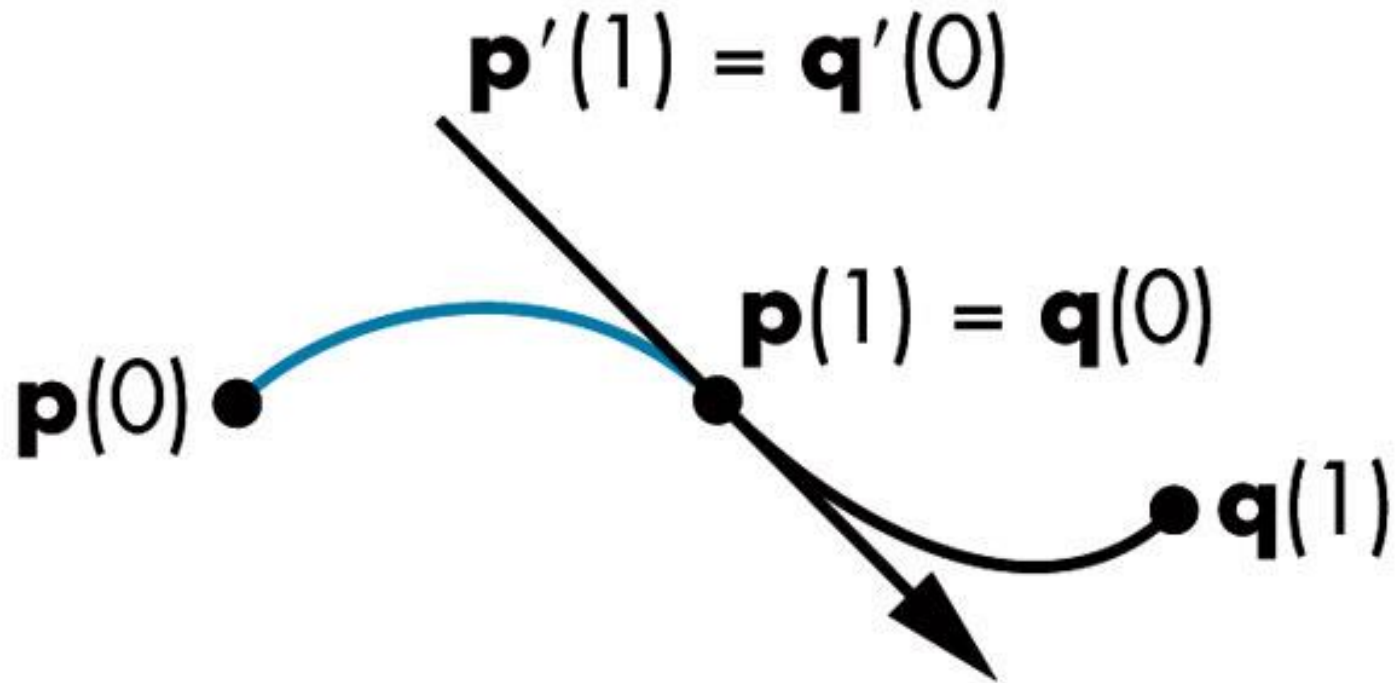
Use **two interpolating conditions** and **two derivative conditions** per segment

Ensure continuity and first derivative continuity between segments

Definition of the Hermite Cubic



Hermite Form at Join Point



Equations

Interpolating conditions are the same at ends

$$p(0) = p_0 = c_0$$

$$p(1) = p_3 = c_0 + c_1 + c_2 + c_3$$

Differentiating we find $p'(u) = c_1 + 2uc_2 + 3u^2c_3$

Evaluating at end points

$$p'(0) = p'_0 = c_1$$

$$p'(1) = p'_3 = c_1 + 2c_2 + 3c_3$$

Matrix Form

$$\mathbf{q} = \begin{bmatrix} p_0 \\ p_3 \\ p'_0 \\ p'_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \mathbf{c} \Rightarrow \mathbf{c} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix}^{-1} \mathbf{q} = \mathbf{M}_H \mathbf{q}$$

Solving, we find $\mathbf{c} = \mathbf{M}_H \mathbf{q}$ where \mathbf{M}_H is the **Hermite matrix**

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

Blending Polynomials

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

$$\mathbf{b}(u)^T = [1 \ u \ u^2 \ u^3] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix}^T$$

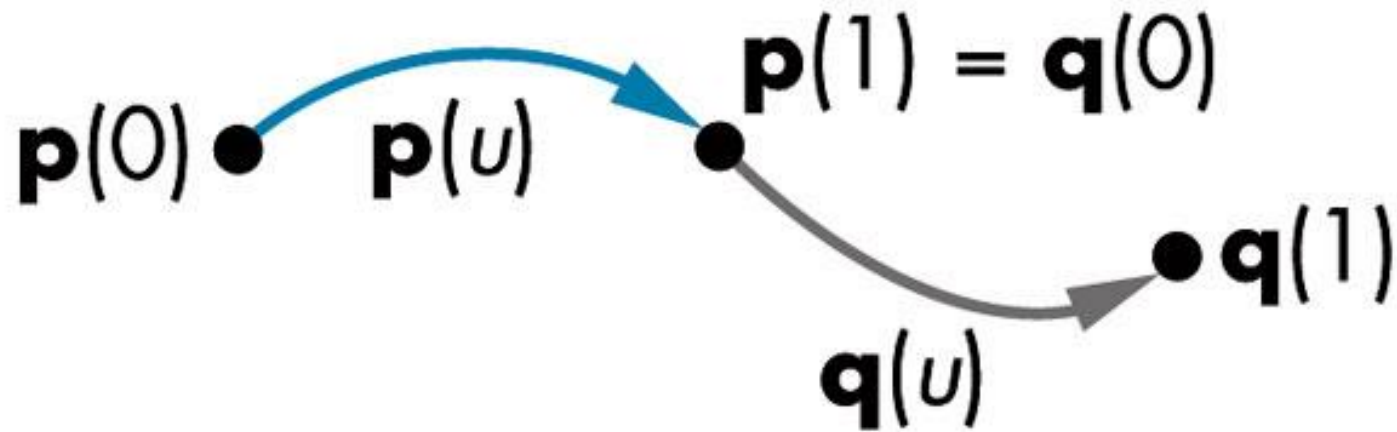
Although these functions are smooth, the Hermite form is not used directly in Computer Graphics and CAD because we usually have **control points** but **not derivatives**

However, the Hermite form is **the basis of the Bezier form**

Parametric and Geometric Continuity

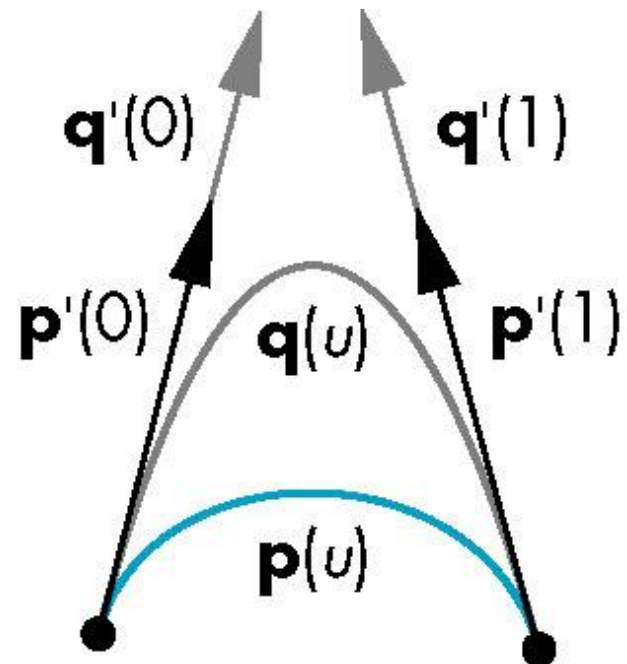
- We can require the derivatives of x , y , and z to each be continuous at join points (*parametric continuity*)
- Alternately, we can only require that the **tangents** of the resulting curve be continuous (*geometry continuity*)
- The latter gives more flexibility as we have need satisfy only two conditions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

Continuity at the 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 technique is used in drawing applications



Higher Dimensional Approximations

- The techniques for both interpolating and Hermite curves can be used with **higher dimensional parametric polynomials**
- For interpolating form, the resulting matrix becomes increasingly **more ill-conditioned** and the resulting curves **less smooth** and more prone to **numerical errors**
- In both cases, there is more work in rendering the resulting polynomial curves and surfaces

Bezier and Spline Curves and Surfaces

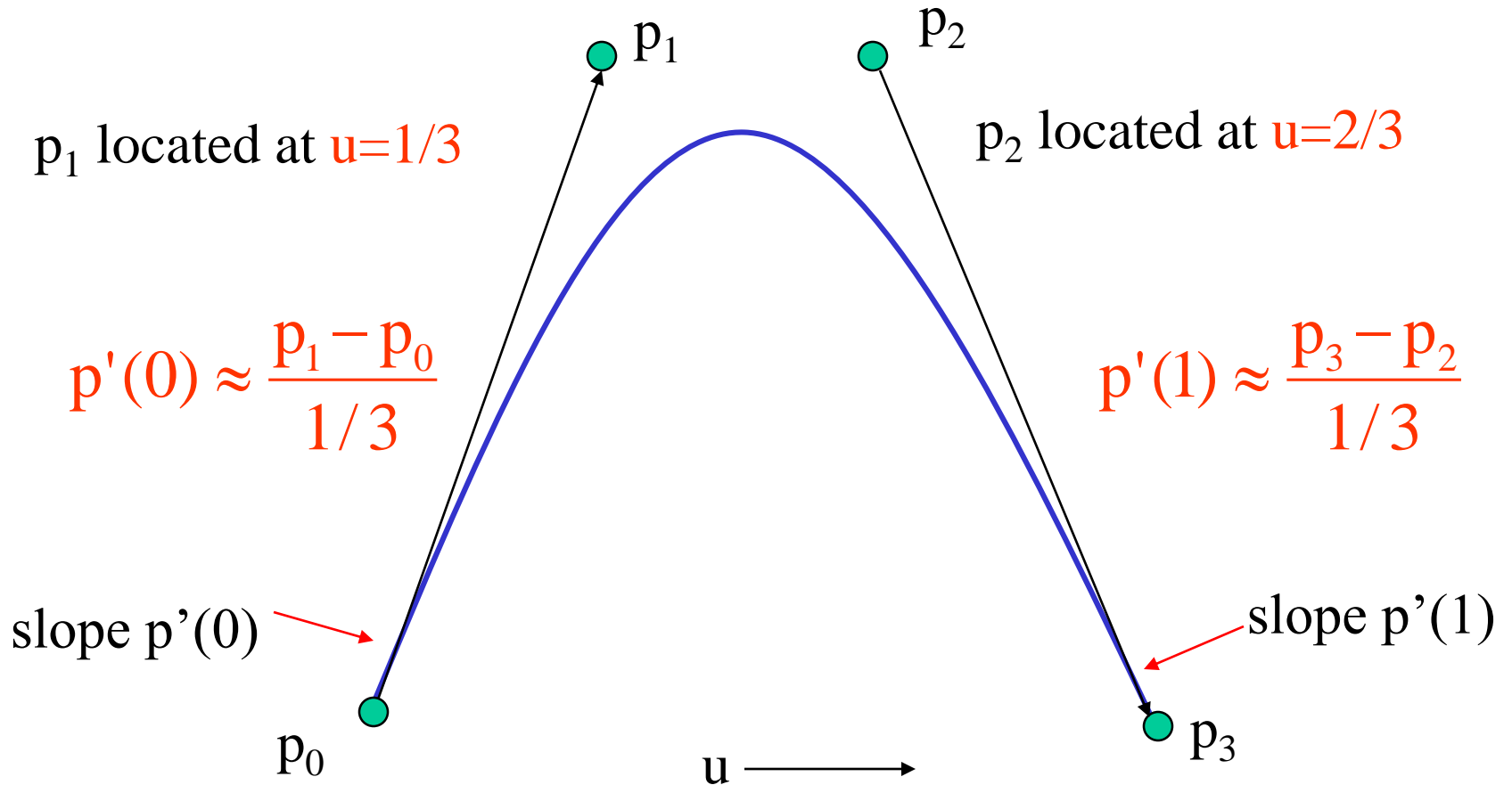
Objectives

- Introduce the **Bezier curves** and surfaces
- Derive the required matrices
- Introduce the **B-spline** and compare it to the standard cubic Bezier

Bezier's Idea

- In graphics and CAD, we **do not usually** 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**

Approximating Derivatives



Equations

Interpolating conditions are the same

$$p(0) = p_0 = c_0$$

$$p(1) = p_3 = c_0 + c_1 + c_2 + c_3$$

Approximating derivative conditions

$$p'(0) = 3(p_1 - p_0) = c_1$$

$$p'(1) = 3(p_3 - p_2) = c_1 + 2c_2 + 3c_3$$

Solve four linear equations for $\mathbf{c} = \mathbf{M}_B \mathbf{p}$

Bezier Curves

$$p(u) = \mathbf{u}^T \mathbf{c} \quad \text{where} \quad u = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix} \quad \text{and} \quad c = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

$$p(0) = p_0 = c_0$$

$$p(1) = p_3 = c_0 + c_1 + c_2 + c_3$$

$$p'(0) = 3(p_1 - p_0) = c_1$$

$$p'(1) = 3(p_3 - p_2) = c_1 + 2c_2 + 3c_3$$


$$p'(u) = [0 \quad 1 \quad 2u \quad 3u^2] c$$

$$\begin{bmatrix} p(0) \\ p(1) \\ p'(0) \\ p'(1) \end{bmatrix} = \begin{bmatrix} p_0 \\ p_3 \\ 3(p_1 - p_0) \\ 3(p_3 - p_2) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_0 + c_1 + c_2 + c_3 \\ c_1 \\ c_1 + 2c_2 + 3c_3 \end{bmatrix}$$


$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_0 + c_1 + c_2 + c_3 \\ c_1 \\ c_1 + 2c_2 + 3c_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} c$$

$$\begin{bmatrix} p(0) \\ p(1) \\ p'(0) \\ p'(1) \end{bmatrix} = \begin{bmatrix} p_0 \\ p_3 \\ 3(p_1 - p_0) \\ 3(p_3 - p_2) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \mathbf{c}$$



$$\mathbf{c} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$



$$\mathbf{c} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

$$\rightarrow c = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} = M_B P$$

Where:

$$M_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

$$\mathbf{b}(u) = \begin{bmatrix} (1-u)^3 \\ 3u(1-u)^2 \\ 3u^2(1-u) \\ u^3 \end{bmatrix}$$

Bezier Curves:

$$P(u) = [1 \ u \ u^2 \ u^3] c = [1 \ u \ u^2 \ u^3] M_B P = \mathbf{b}(u)^T P$$

$$P(u) = (1-u)^3 P_0 + 3u(1-u)^2 P_1 + 3u^2(1-u) P_2 + u^3 P_3$$

Bezier Matrix

$$P(u) = (1 - u)^3 P_0 + 3u(1 - u)^2 P_1 + 3u^2(1 - u) P_2 + u^3 P_3$$

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

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

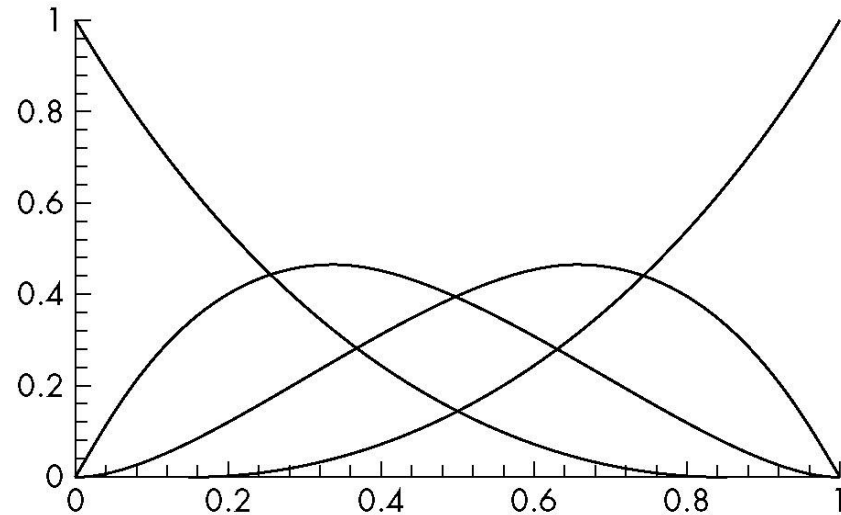
blending functions



$$\mathbf{p} = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

Blending Functions

$$\mathbf{b}(u) = \begin{bmatrix} (1-u)^3 \\ 3u(1-u)^2 \\ 3u^2(1-u) \\ u^3 \end{bmatrix}$$



Note that all zeros are at 0 and 1 which forces the functions to be smooth over $(0,1)$

Bernstein Polynomials

- The blending functions are a special case of the Bernstein polynomials

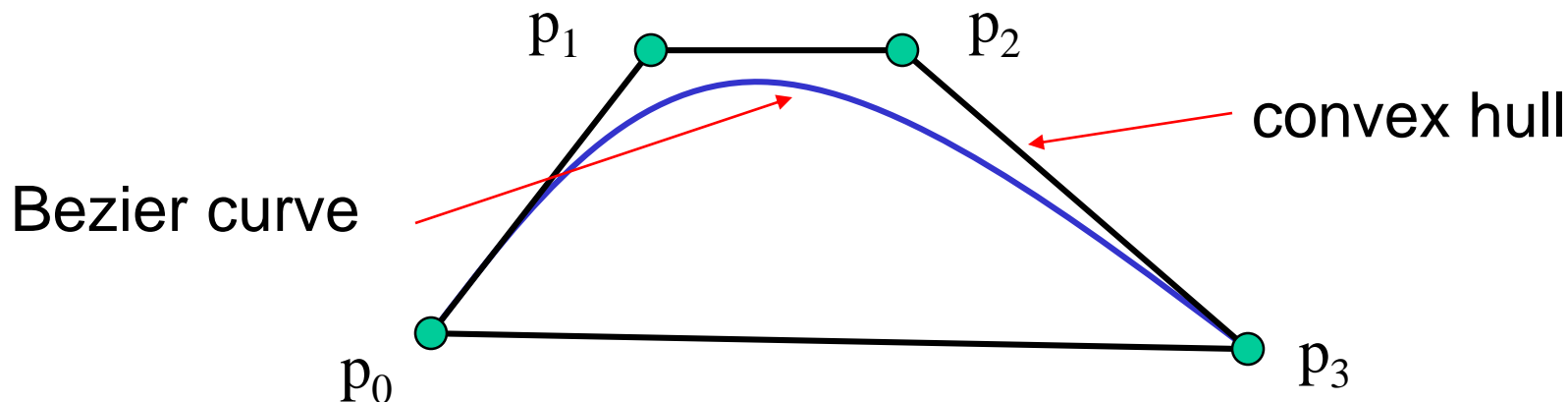
$$b_{kd}(u) = \frac{d!}{k!(d-k)!} u^k (1-u)^{d-k}$$

$$P(u) = \sum_{k=0}^d C_k^d u^k (1-u)^{d-k} P_k$$

- These polynomials give the blending polynomials for any degree Bezier form
 - All zeros at 0 and 1
 - For any degree they all sum to 1
 - They are all between 0 and 1 inside (0,1)

Convex Hull Property

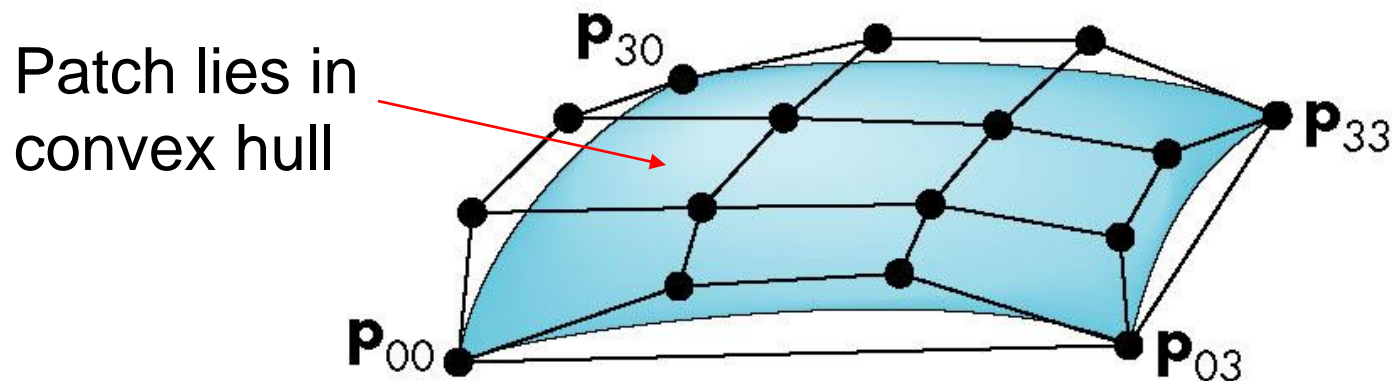
- The properties of the Bernstein polynomials ensure that all Bezier curves lie in the **convex hull** of their control points
- Hence, even though we do not interpolate all the data, we cannot be too far away



Bezier Patches

Using same data array $\mathbf{P}=[p_{ij}]$ as with interpolating form

$$p(u, v) = \sum_{i=0}^3 \sum_{j=0}^3 b_i(u) b_j(v) p_{ij} = \mathbf{u}^T \mathbf{M}_B \mathbf{P} \mathbf{M}_B^T \mathbf{v}$$



Analysis

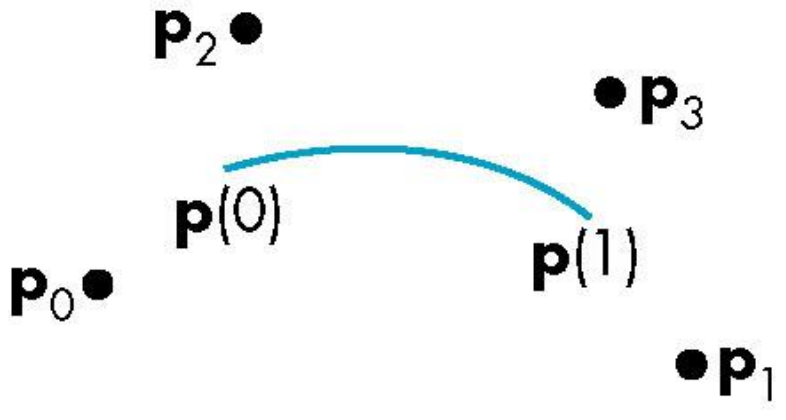
- Although the Bezier form is much better than the interpolating form, we have the derivatives are **not continuous** at join points
- Can we do better?
 - Go to higher order Bezier
 - More work
 - **Derivative continuity** still only approximate
 - Supported by OpenGL
 - Apply different conditions
 - Tricky without letting order increase

B-Splines

- Basis splines: use the data at $\mathbf{p}=[p_{i-2} \ p_{i-1} \ p_i \ p_{i+1}]^T$ to define curve only between p_{i-1} and p_i
- Allows us to apply more continuity conditions to each segment
- For cubics, we can have continuity of function, **first and second derivatives** at join points
- Cost is 3 times as much work for curves
 - Add one new point each time rather than three
- For surfaces, we do 9 times as much work

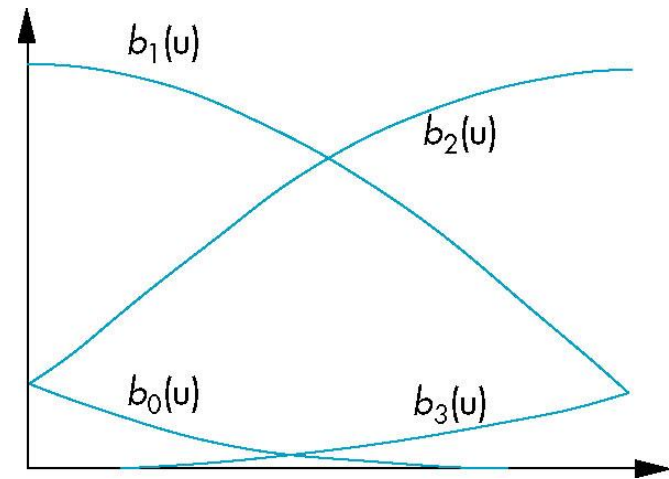
Cubic B-spline

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

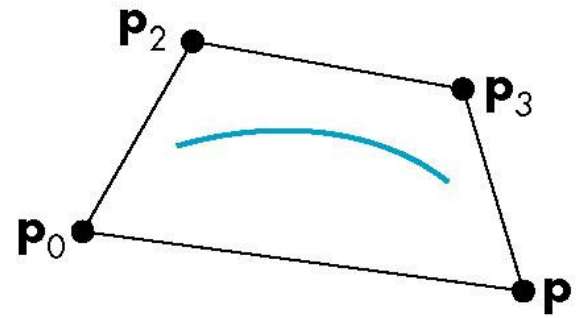
$$\mathbf{M}_s = \begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$


Blending Functions

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



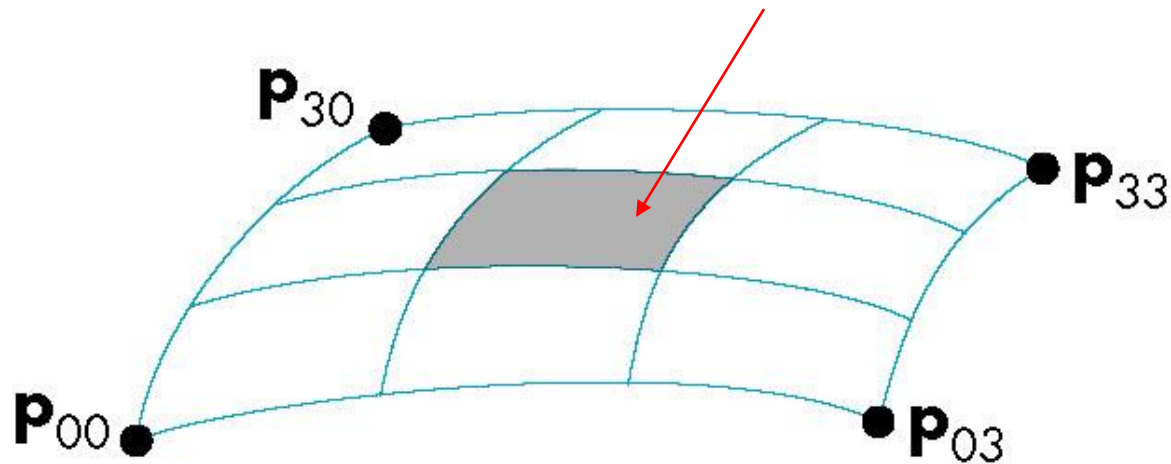
convex hull property



B-Spline Patches

$$p(u, v) = \sum_{i=0}^3 \sum_{j=0}^3 b_i(u) b_j(v) p_{ij} = u^T \mathbf{M}_S \mathbf{P} \mathbf{M}_S^T v$$

defined over **only 1/9 of region**



Splines and Basis

- If we examine the cubic B-spline from the perspective of each control (data) point, **each interior point contributes** (through the blending functions) **to four segments**
- We can **rewrite** $p(u)$ in terms of the data points as

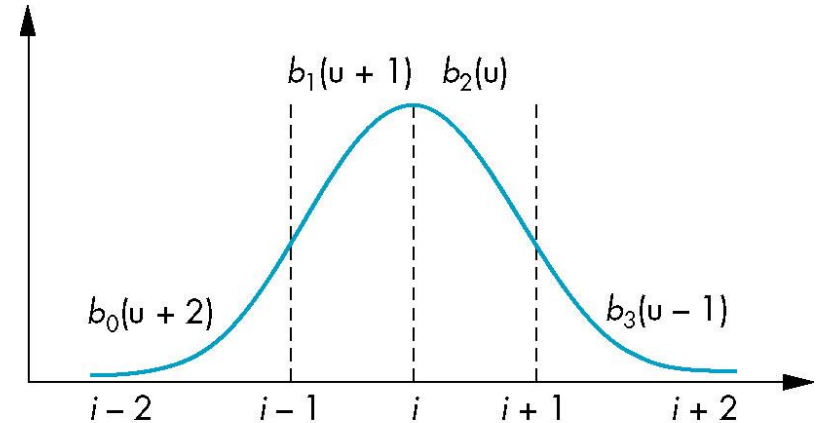
$$p(u) = \sum B_i(u) p_i$$

defining the basis functions $\{B_i(u)\}$

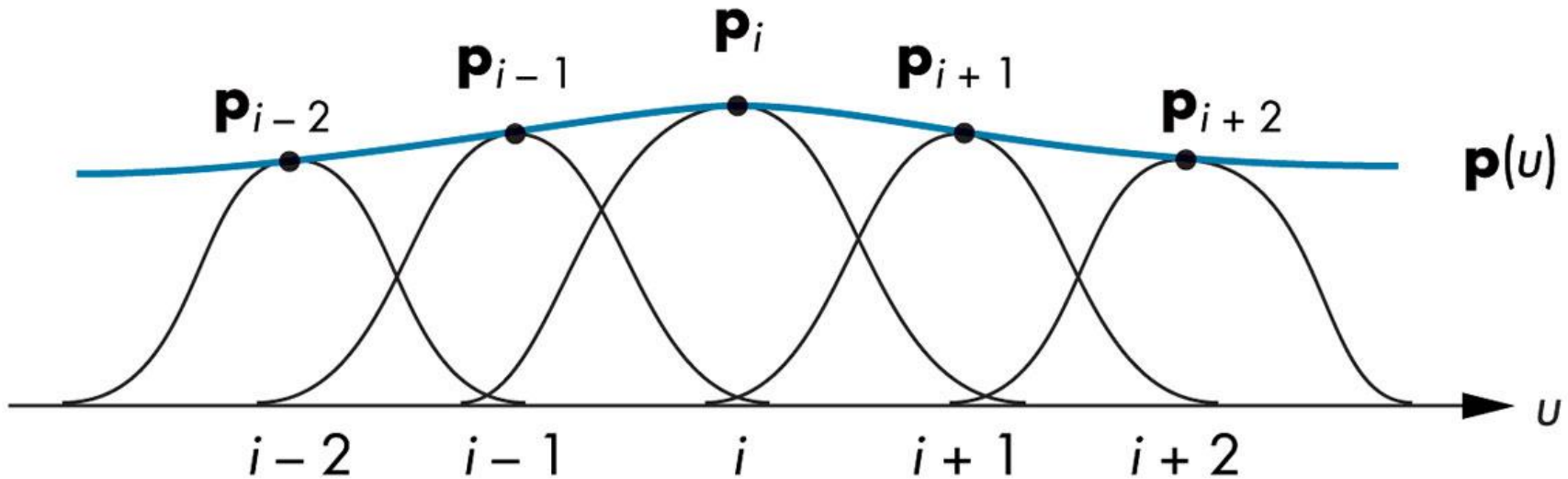
Basis Functions

In terms of the blending polynomials

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



Approximating function over interval



Generalizing Splines

- We can extend to splines of any degree
- Data and conditions not to have to given at equally spaced values (the *knots*)
 - Nonuniform and uniform splines
 - Can have repeated knots
 - Can force spline to interpolate points
- Cox-deBoor recursion gives method of evaluation

NURBS

- Nonuniform Rational B-Spline curves and surfaces add a fourth variable w to x, y, z
 - Can interpret as *weight* to give more importance to some control data
 - Can also interpret as moving to homogeneous coordinate
- Requires a perspective division
 - NURBS act correctly for perspective viewing
- **Quadrics are a special case of NURBS**

Rendering Curves and Surfaces

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

Finite Differences

For **equally spaced** $\{u_k\}$ we define *finite differences*

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

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

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

For a polynomial of **degree** n ,
the **n^{th} finite difference** is **constant**

Finite Differences

If $u_{k+1} - u_k = h$ is constant, then we can show that if $p(u)$ is a polynomial of degree n , then $\Delta^{(n)} p(u_k)$ is constant for all k .

We need the first $n+1$ values of $p(u_k)$ to find $\Delta^{(n)} p(u_0)$.

But once we have $\Delta^{(n)} p(u_0)$, we can copy this value across the table and work upward to compute the successive values of $p(u_k)$, using the rearranged recurrence

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

Building a Finite Difference Table

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

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

Finding the Next Values

Starting at the bottom, we can work up generating new values for the polynomial

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

Diagram illustrating the process of finding the next values for the polynomial \mathbf{p} using finite differences. The table shows the values of \mathbf{p} and its first, second, and third differences ($\Delta^{(1)} \mathbf{p}$, $\Delta^{(2)} \mathbf{p}$, $\Delta^{(3)} \mathbf{p}$) for $t = 0$ to 5 . The values are generated by adding the differences to the previous values, as indicated by the arrows and plus signs. The third difference is constant at 6, which is used to find the next values in the second difference row, and then in the first difference row, and finally in the polynomial row.

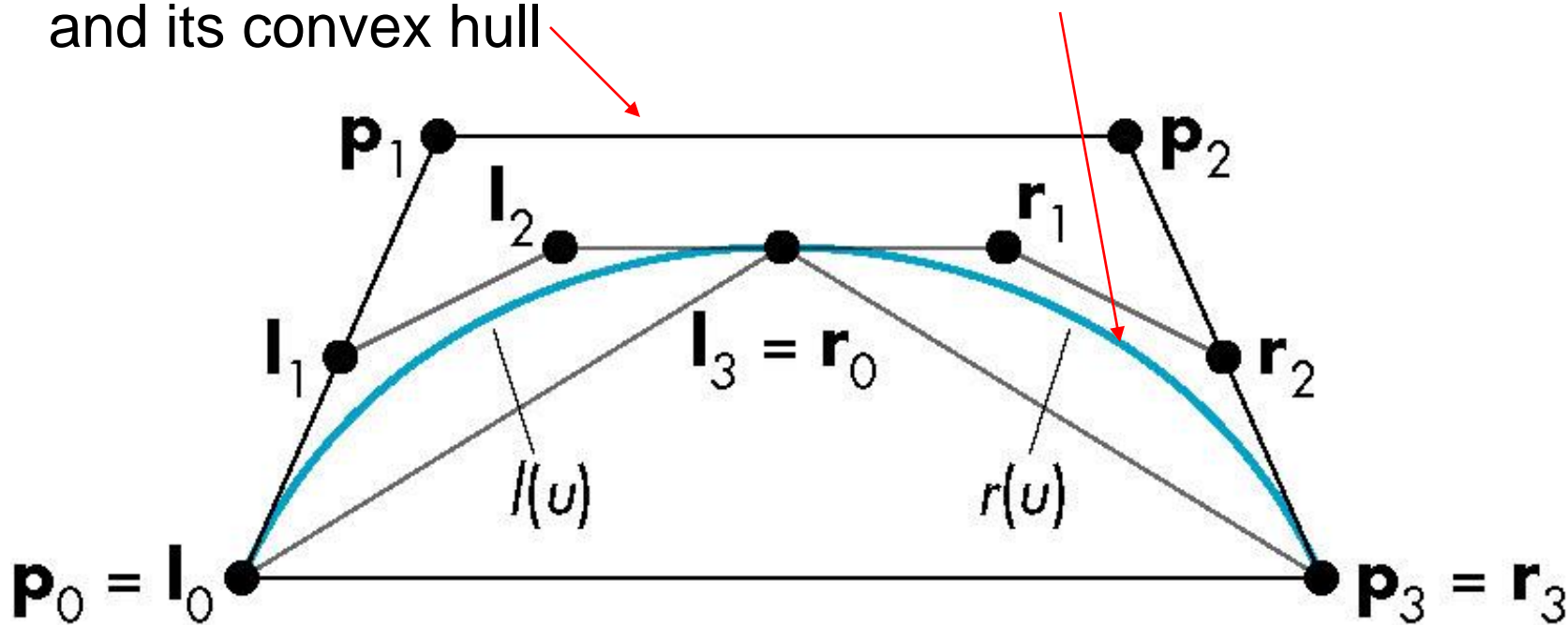
constant

deCasteljau Recursion

- We can use the **convex hull property** of Bezier curves to obtain an efficient recursive method that does not require any function evaluations
 - Uses only the values at the control points
- Based on the idea that “**any polynomial and any part of a polynomial is a Bezier polynomial for properly chosen control data**”

Splitting a Cubic Bezier

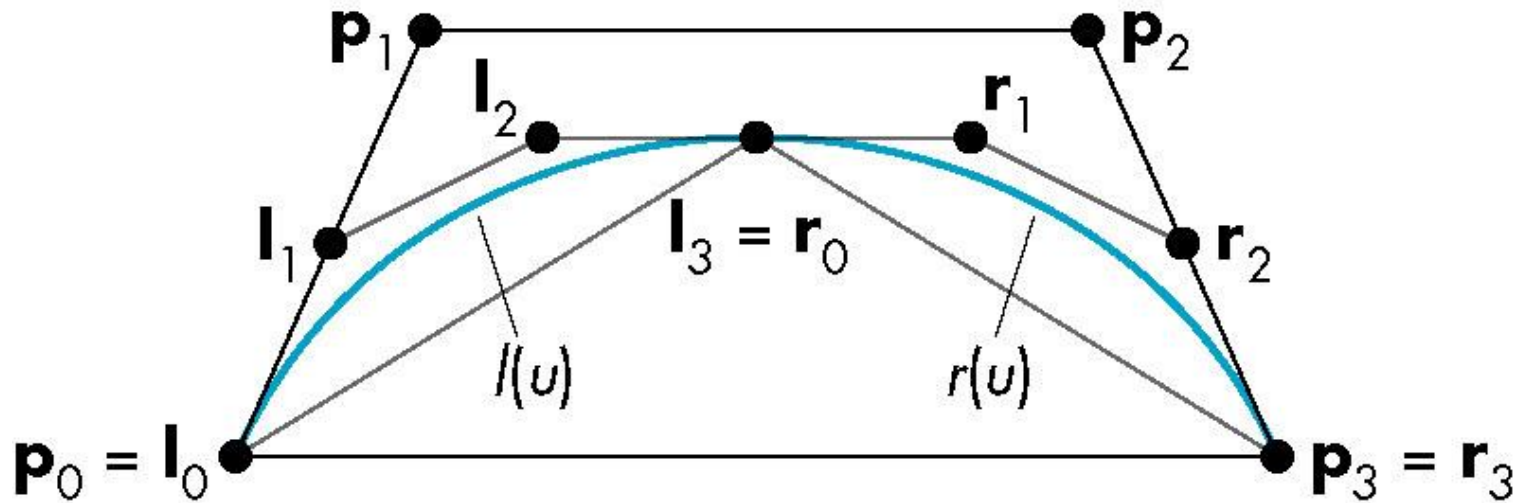
p_0, p_1, p_2, p_3 determine a cubic Bezier polynomial and its convex hull



Consider left half $l(u)$ and right half $r(u)$

$l(u)$ and $r(u)$

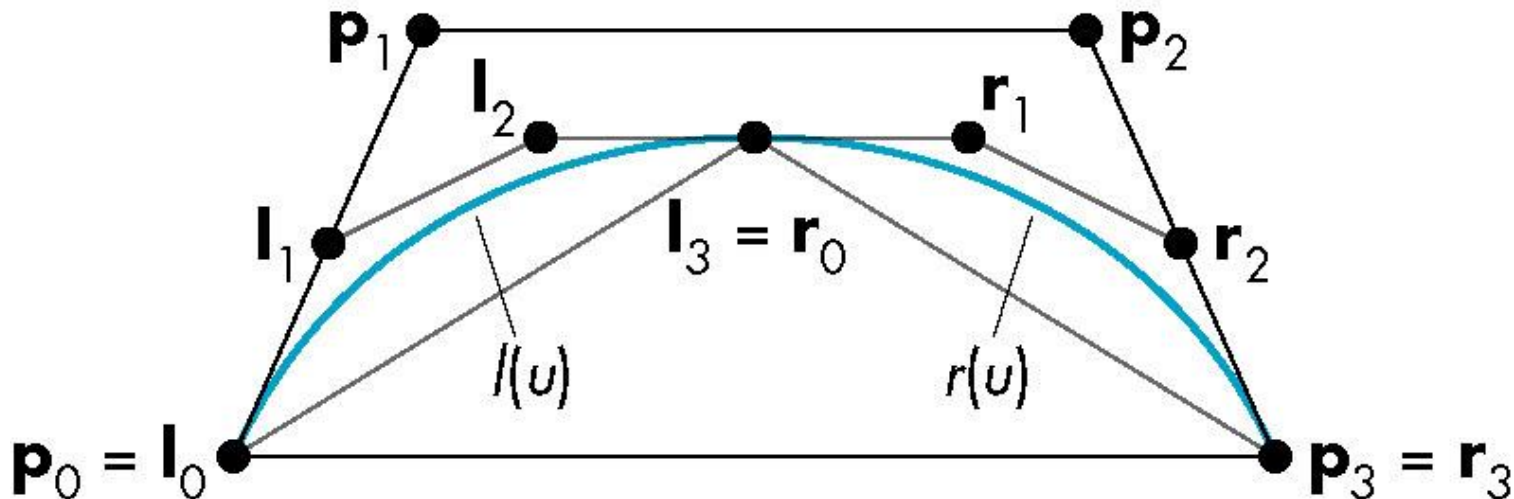
Since $l(u)$ and $r(u)$ are Bezier curves, we should be able to find two sets of control points $\{l_0, l_1, l_2, l_3\}$ and $\{r_0, r_1, r_2, r_3\}$ that determine them



Convex Hulls

$\{l_0, l_1, l_2, l_3\}$ and $\{r_0, r_1, r_2, r_3\}$ each have a convex hull that is closer to $p(u)$ than the convex hull of $\{p_0, p_1, p_2, p_3\}$. This is known as the *variation diminishing property*.

The polyline from l_0 to $l_3 (=r_0)$ to r_3 is an approximation to $p(u)$. Repeating recursively we get better approximations.



Equations

Start with Bezier equations $p(u) = \mathbf{u}^T \mathbf{M}_B \mathbf{p}$

$l(u)$ must interpolate $p(0)$ and $p(1/2)$

$$l(0) = l_0 = p_0$$

$$l(1) = l_3 = p(1/2) = 1/8(p_0 + 3p_1 + 3p_2 + p_3)$$

Matching slopes, taking into account that $l(u)$ and $r(u)$ only go over half the distance as $p(u)$

$$l'(0) = 3(l_1 - l_0) = p'(0) = 3/2(p_1 - p_0)$$

$$l'(1) = 3(l_3 - l_2) = p'(1/2) = 3/8(-p_0 - p_1 + p_2 + p_3)$$

Symmetric equations hold for $r(u)$

Efficient Form

$$l_0 = p_0$$

$$r_3 = p_3$$

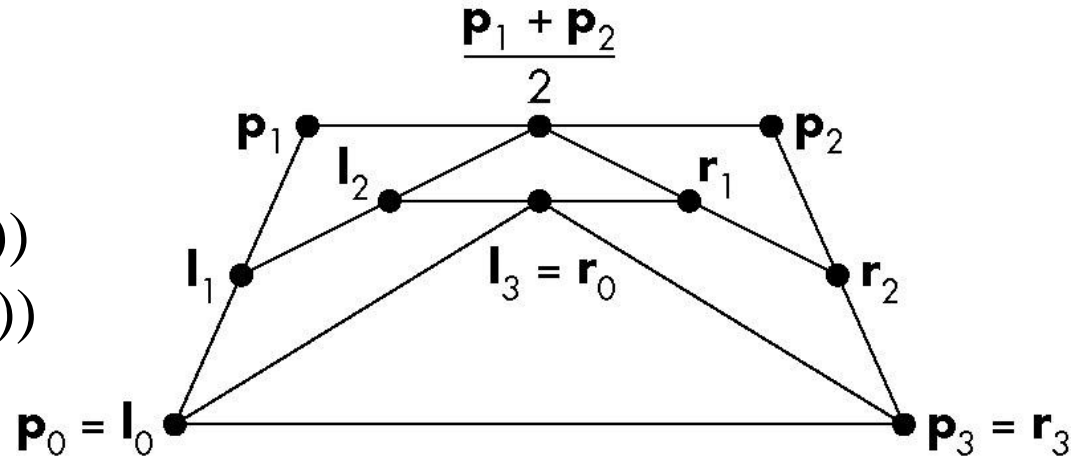
$$l_1 = \frac{1}{2}(p_0 + p_1)$$

$$r_2 = \frac{1}{2}(p_2 + p_3)$$

$$l_2 = \frac{1}{2}(l_1 + \frac{1}{2}(p_1 + p_2))$$

$$r_1 = \frac{1}{2}(r_2 + \frac{1}{2}(p_1 + p_2))$$

$$l_3 = r_0 = \frac{1}{2}(l_2 + r_1)$$



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 $\mathbf{p} = \mathbf{M}_B^{-1} \mathbf{M}_I \mathbf{q}$

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

Matrices

Interpolating to Bezier $\mathbf{M}_B^{-1} \mathbf{M}_I =$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{5}{6} & 3 & -\frac{3}{2} & \frac{1}{3} \\ \frac{1}{3} & -\frac{3}{2} & 3 & -\frac{5}{6} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

 $\Rightarrow \mathbf{p} = \mathbf{M}_B^{-1} \mathbf{M}_I \mathbf{q}$

\mathbf{M}_B :Bezier matrix

B-Spline to Bezier $\mathbf{M}_B^{-1} \mathbf{M}_S = \frac{1}{6}$

$$\begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix}$$

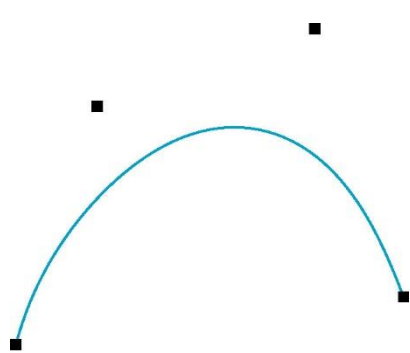
$\mathbf{u}^T \mathbf{M}_B \mathbf{p} = \mathbf{u}^T \mathbf{M}_S \mathbf{q}$

 $\Rightarrow \mathbf{p} = \mathbf{M}_B^{-1} \mathbf{M}_S \mathbf{q}$

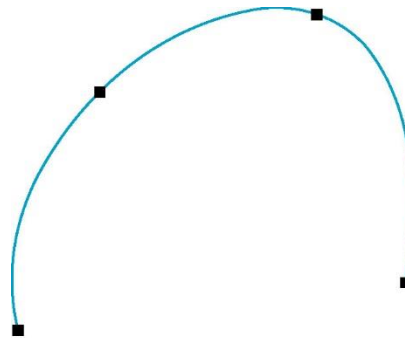
\mathbf{M}_S :B-spline matrix

Example

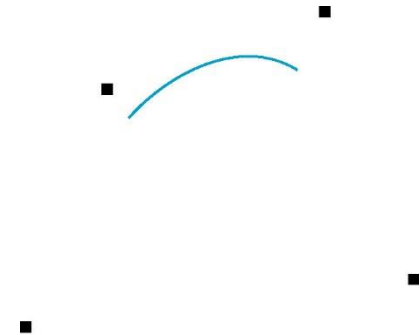
These three curves were all generated from the same original data using Bezier recursion by converting all control point data to Bezier control points



Bezier



Interpolating



B Spline

Subdivision

- Subdivision of Bezier Curves
- Subdivision of Bezier Surfaces
- Mesh Subdivision

Subdivision of Bezier Curves

$$l_0 = p_0$$

$$r_3 = p_3$$

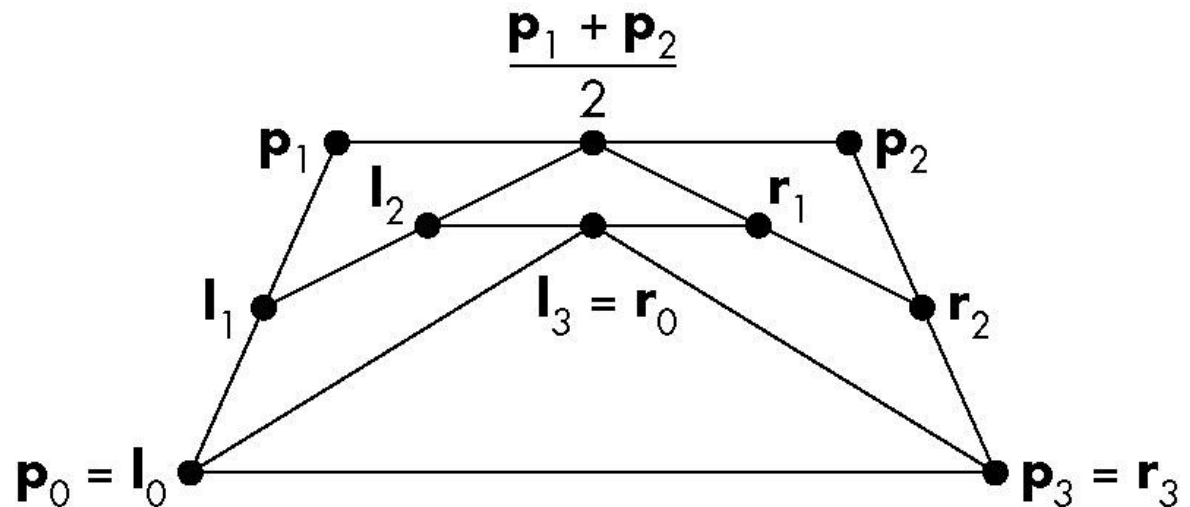
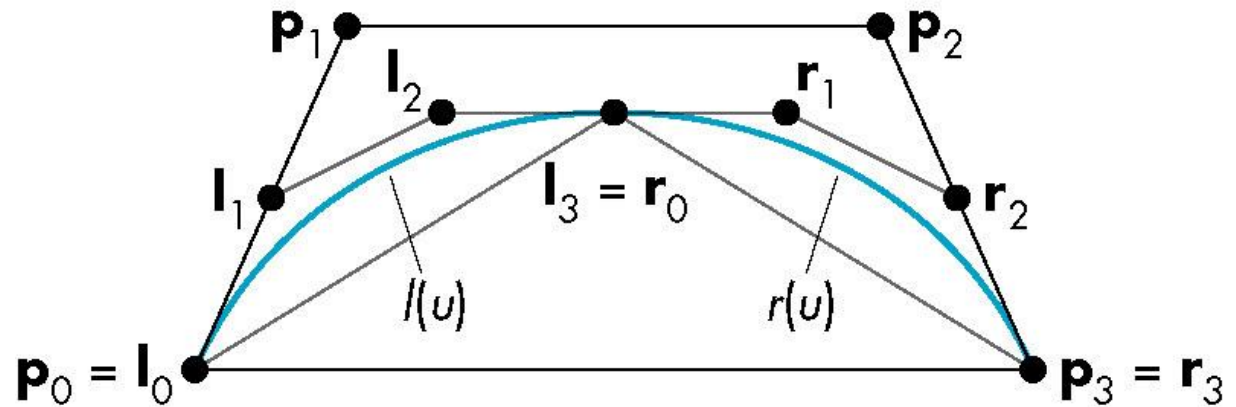
$$l_1 = \frac{1}{2}(p_0 + p_1)$$

$$r_2 = \frac{1}{2}(p_2 + p_3)$$

$$l_2 = \frac{1}{2}(l_1 + \frac{1}{2}(p_1 + p_2))$$

$$r_1 = \frac{1}{2}(r_2 + \frac{1}{2}(p_1 + p_2))$$

$$l_3 = r_0 = \frac{1}{2}(l_2 + r_1)$$

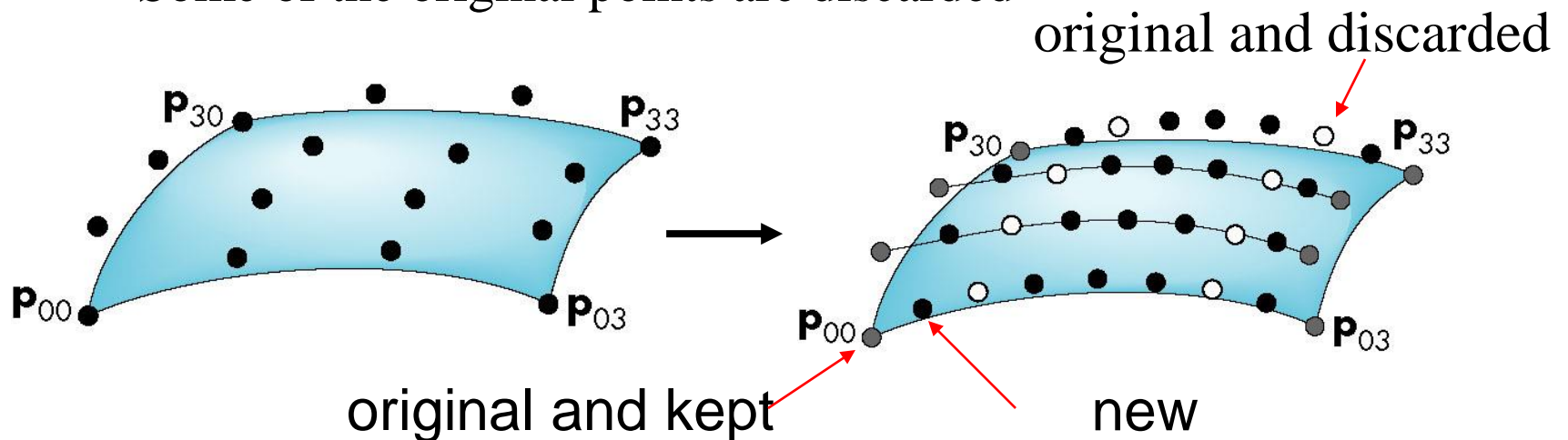


Subdivision of Bezier Surfaces: First Subdivision

- 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)

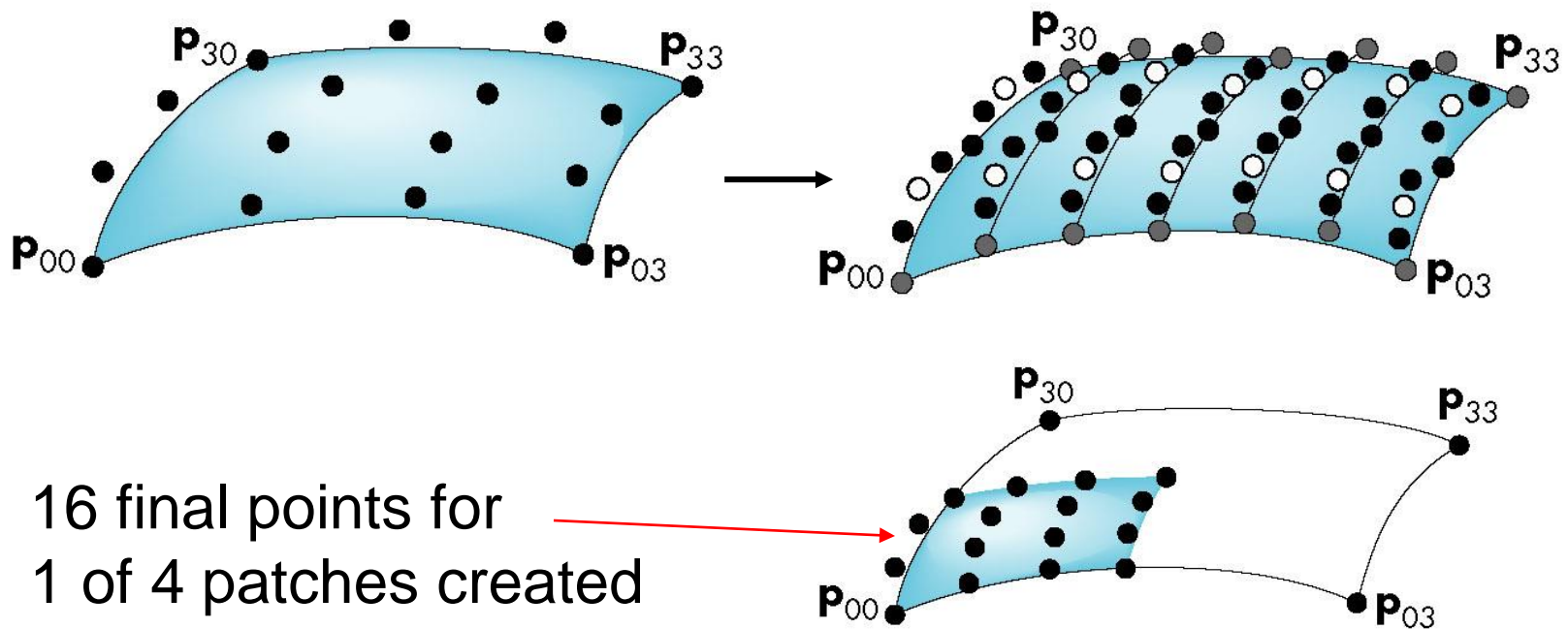
- **First subdivide in u**

- Process creates new points
- Some of the original points are discarded

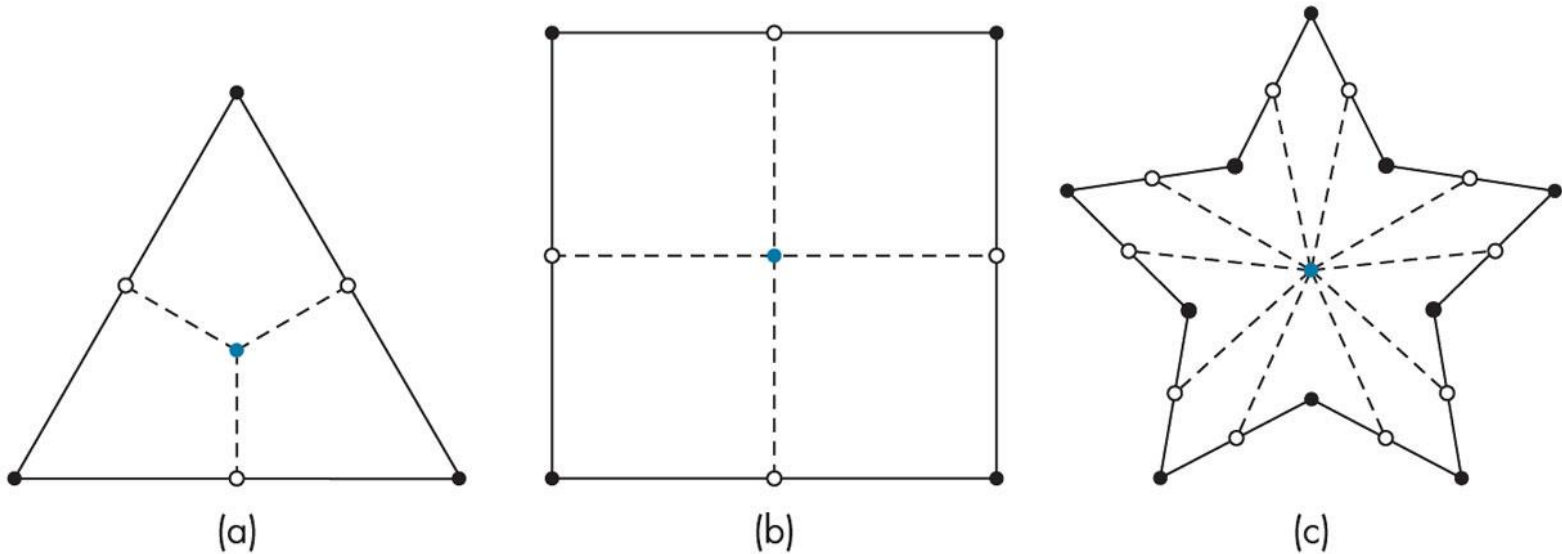


Subdivision of Bezier Surfaces: Second Subdivision

- New points created by subdivision
- Old points discarded after subdivision
- Old points retained after subdivision

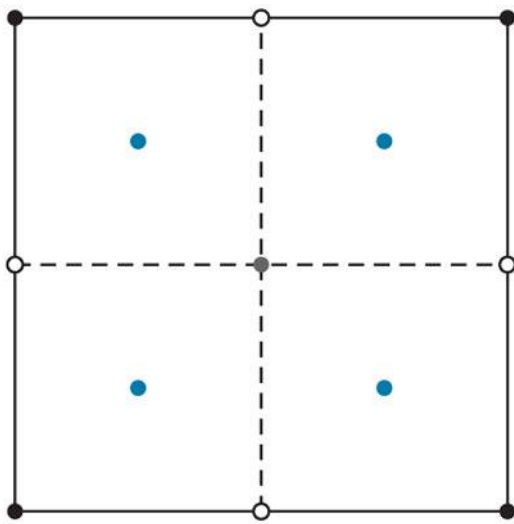


Mesh Subdivision

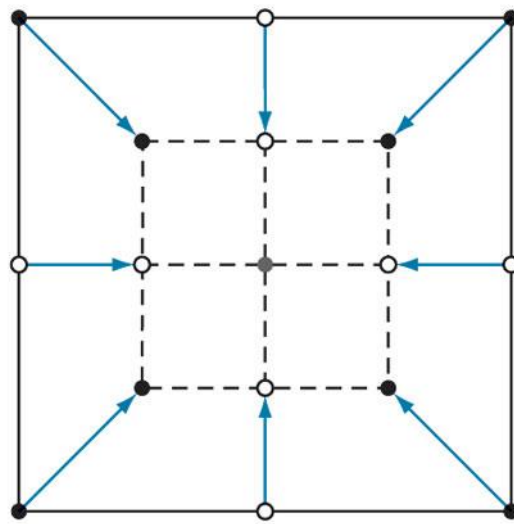


Polygon subdivision. (a) Triangle. (b) Rectangle.
(c) Star-shaped polygon.

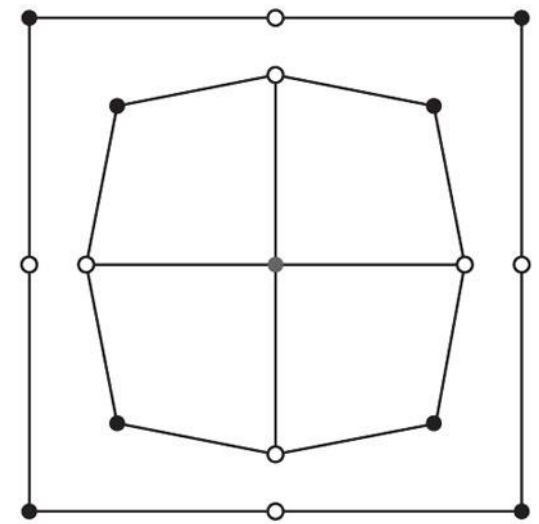
Mesh Subdivision



(a)



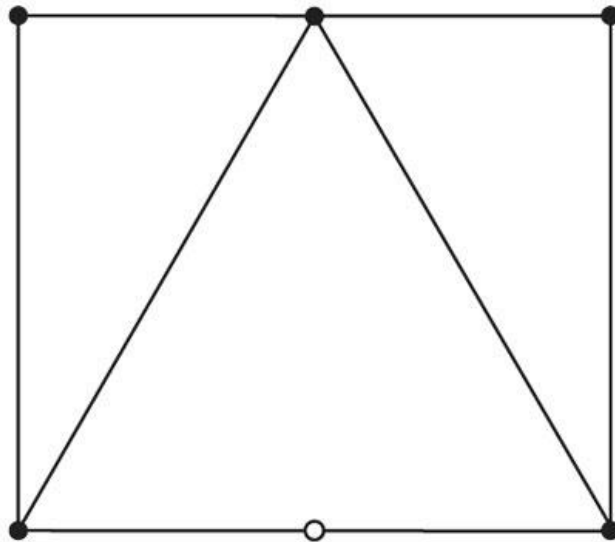
(b)



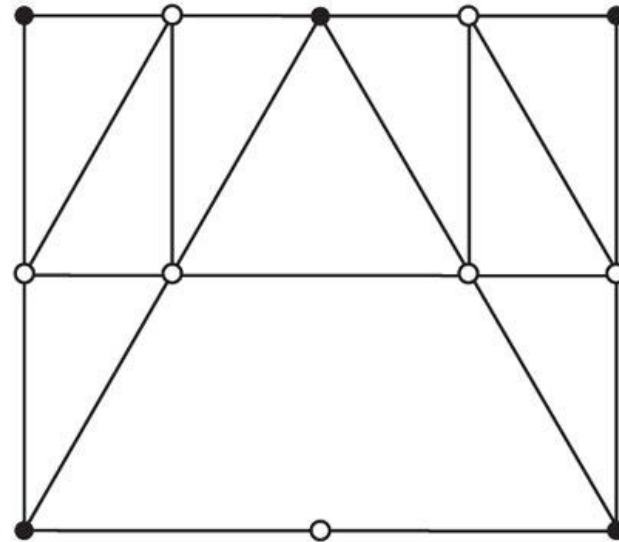
(c)

Catmull-Clark subdivision

Mesh Subdivision



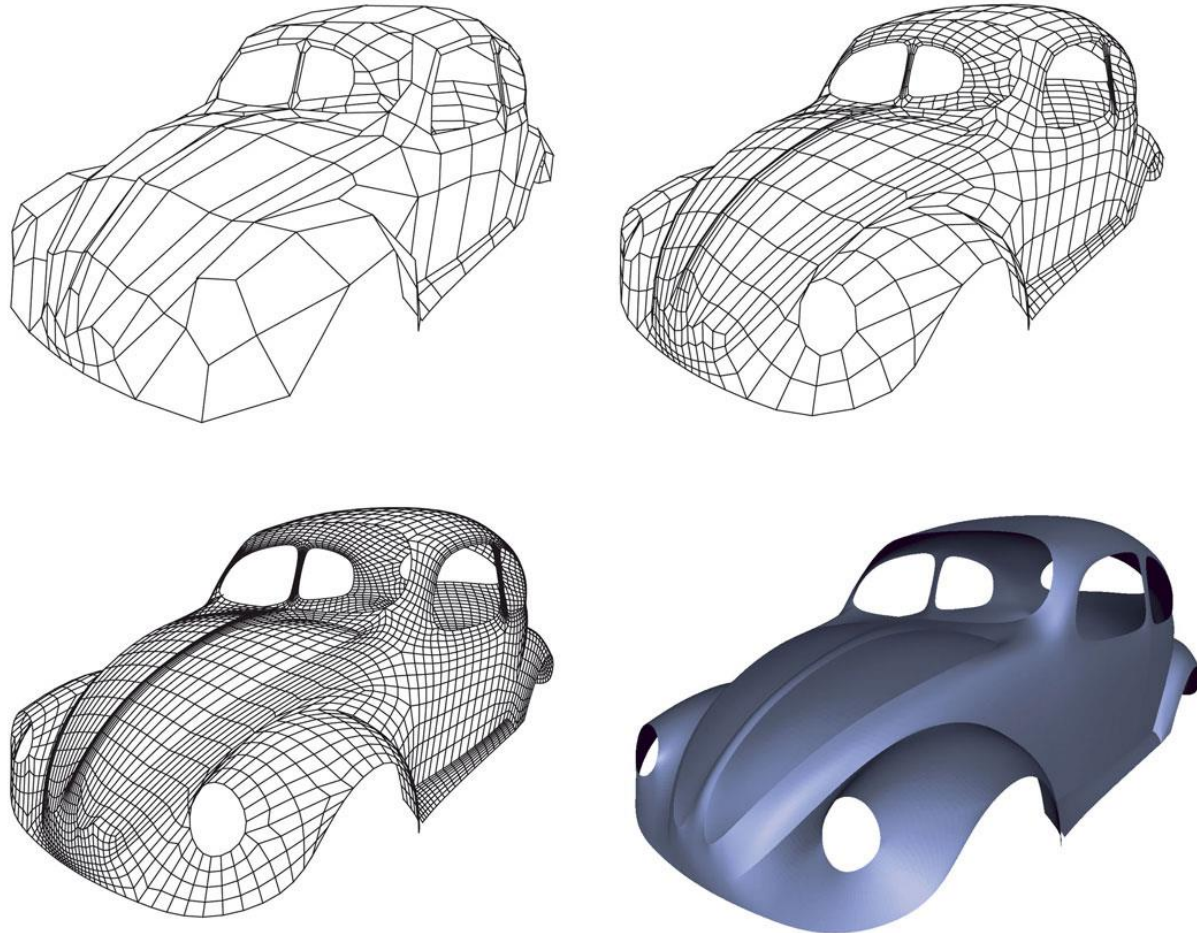
(a)



(b)

Loop subdivision. (a) Triangular mesh.
(b) Triangles after one subdivision.

Successive subdivisions of polygonal mesh and rendered surface. (Images courtesy Caltech Multi-Res Modeling Group)



Normals

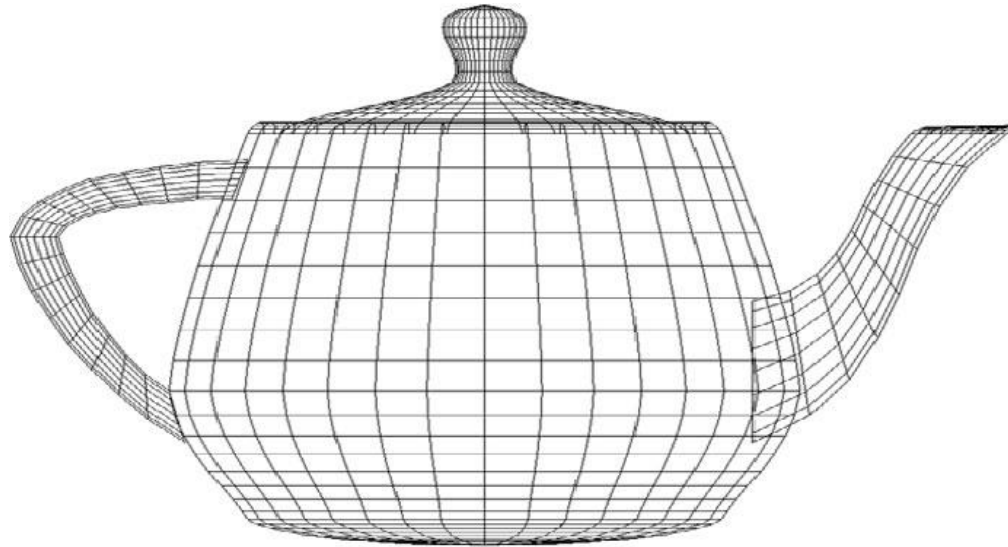
- For rendering we need the normals if we want to shade
 - Can compute from parametric equations

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

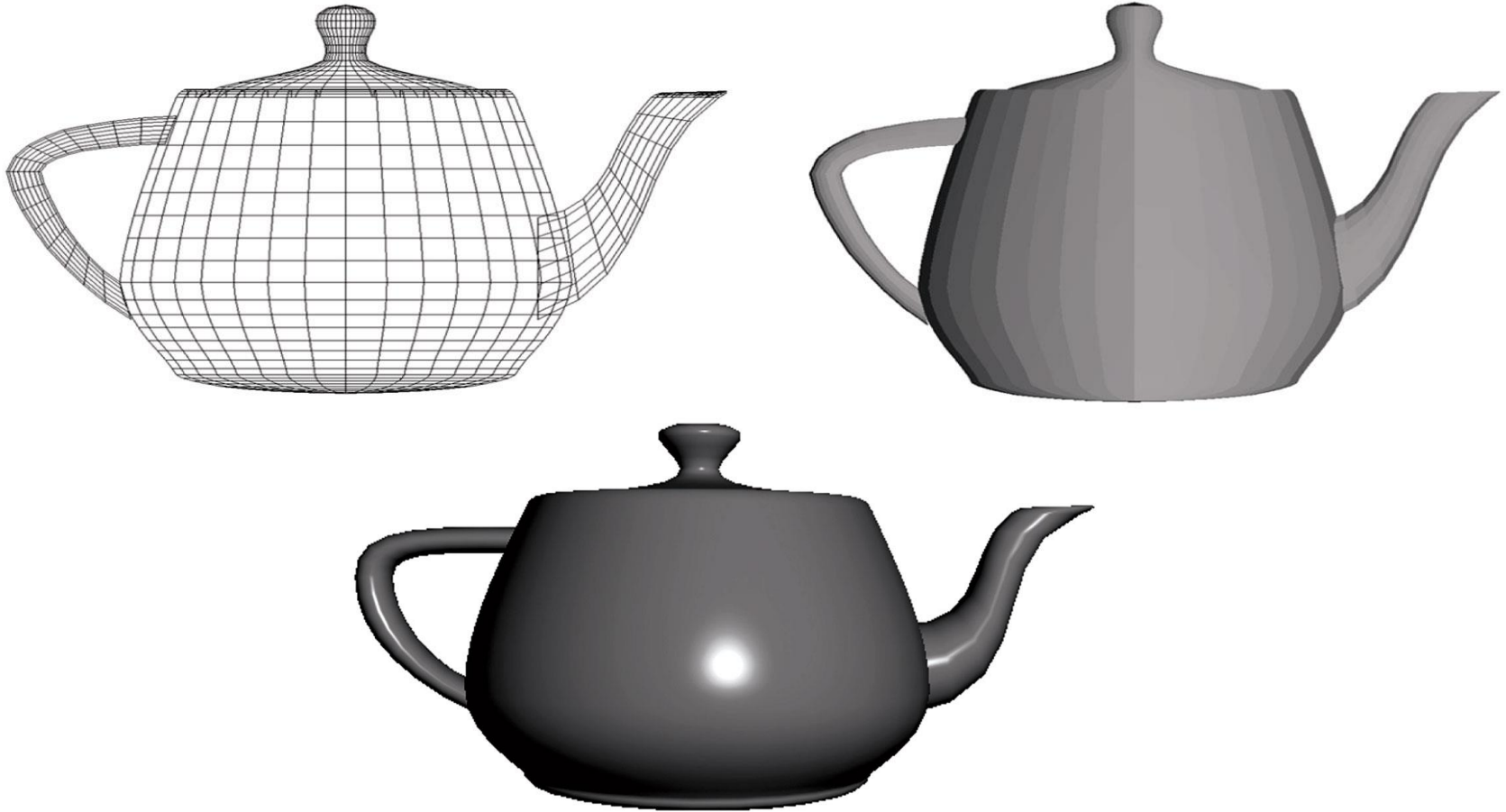
- Can use vertices of corner points to determine
- OpenGL can compute automatically

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



Rendered Teapots



Quadrics

- Any quadric can be written as the quadratic form

$$\mathbf{p}^T \mathbf{A} \mathbf{p} + \mathbf{b}^T \mathbf{p} + c = 0 \text{ where } \mathbf{p} = [x, y, z]^T$$

with \mathbf{A} , \mathbf{b} and c giving the coefficients

- Render by ray casting
 - Intersect with parametric ray $\mathbf{p}(\alpha) = \mathbf{p}_0 + \alpha \mathbf{d}$ that passes through a pixel
 - Yields a scalar quadratic equation
 - No solution: ray misses quadric
 - One solution: ray tangent to quadric
 - Two solutions: entry and exit points

Curves and Surfaces in OpenGL

Objectives

- Introduce OpenGL evaluators
- Learn to render polynomial curves and surfaces
- Discuss quadrics in OpenGL
 - GLUT Quadrics
 - GLU Quadrics

What Does OpenGL Support?

- Evaluators: a general mechanism for working with the **Bernstein polynomials**
 - Can use any degree polynomials
 - Can use in 1-4 dimensions
 - Automatic generation of **normals** and **texture coordinates**
 - NURBS supported in GLU
- Quadrics
 - GLU and GLUT contain polynomial approximations of quadrics

One-Dimensional Evaluators

- Evaluate a Bernstein polynomial of any degree at a set of specified values
- Can evaluate a variety of variables
 - Points along a 2, 3 or 4 dimensional curve
 - Colors
 - Normals
 - Texture Coordinates
- We can set up **multiple evaluators** that are all evaluated for the same value

Setting Up an Evaluator

what we want to evaluate

max and min of u

`glMap1f(type, u_min, u_max, stride,
order, pointer_to_array)`

1+degree of polynomial

separation between
data points

pointer to control data

Each type must be enabled by `glEnable(type)`

Example

Consider an evaluator for a **cubic Bezier** curve over (0,1)

```
point data[ ]={.....}; /* 3d data */  
glMap1f(GL_MAP_VERTEX_3,0.0,1.0,3,4,data);
```

data are 3D vertices

data are arranged as x,y,z,x,y,z.....

three floats between data points in array

cubic

```
glEnable(GL_MAP_VERTEX_3);
```

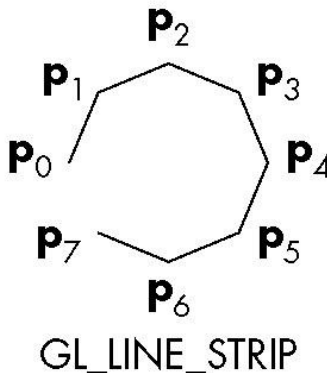
Evaluating

- The function `glEvalCoord1f(u)` causes all enabled evaluators to be evaluated for the specified `u`
 - Can replace `glVertex`, `glNormal`, `glTexCoord`
- The values of `u` need not be equally spaced

Example

- Consider the previous evaluator that was set up for a **cubic Bezier** over **(0,1)**
- Suppose that we want to approximate the curve with a **100 point polyline**

```
glBegin(GL_LINE_STRIP)
    for(i=0; i<100; i++)
        glEvalCoord1f( (float) i/100.0 );
glEnd();
```



Equally Spaced Points

Rather than use a loop, we can **set up** an **equally spaced** mesh (grid) and then **evaluate** it with **one function call**

```
glMapGrid(100, 0.0, 1.0);
```

sets up 100 equally-spaced points on (0,1)

```
glEvalMesh1(GL_LINE, 0, 99);
```

renders lines between adjacent evaluated points from point 0 to point 99

Bezier Surfaces

- Similar procedure to 1D but use 2D evaluators in \mathbf{u} and \mathbf{v}
- Set up with


```
glMap2f(type, u_min, umax, u_stride,  
         u_order, v_min, v_max, v_stride,  
         v_order, pointer_to_data)
```

- Evaluate with `glEvalCoord2f(u,v)`

Example

bicubic over $(0,1) \times (0,1)$

```
point data[4][4]={.....};  
glMap2f(GL_MAP_VERTEX_3, 0.0, 1.0, 3, 4,  
        0.0, 1.0, 12, 4, data);
```

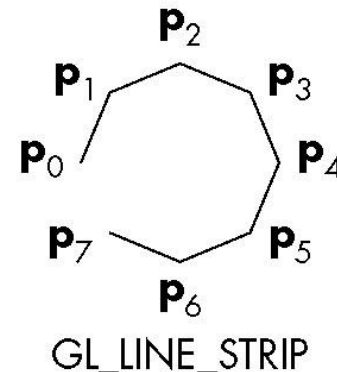
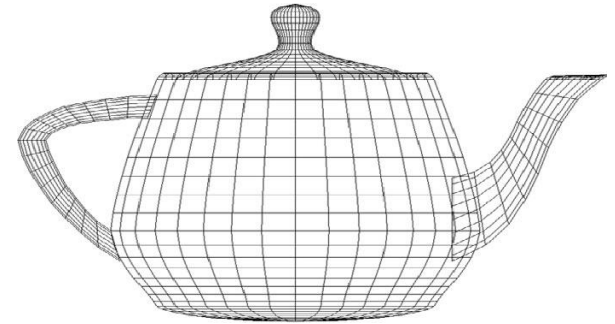


Note that in v direction data points
are separated by 12 floats since array
data is stored by rows

Rendering with Lines

must draw in both directions

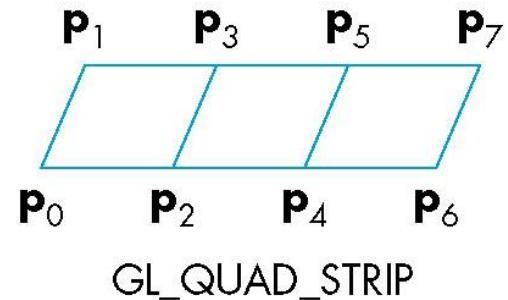
```
for(j=0;j<100;j++) {  
    glBegin(GL_LINE_STRIP);  
        for(i=0;i<100;i++)  
            glEvalCoord2f((float) i/100.0, (float) j/100.0);  
    glEnd();  
    glBegin(GL_LINE_STRIP);  
        for(i=0;i<100;i++)  
            glEvalCoord2f((float) j/100.0, (float) i/100.0);  
    glEnd();  
}
```



Rendering with Quadrilaterals

We can form a quad mesh and render with lines

```
for(j=0; j<99; j++) {  
    glBegin(GL_QUAD_STRIP);  
    for(i=0; i<100; i++) {  
        glVertex2f ((float) i/100.0,  
                    (float) j/100.0);  
        glVertex2f ((float) i/100.0,  
                    (float) (j+1)/100.0);  
    }  
    glEnd();  
}
```



Uniform Meshes

- We can form a 2D mesh (grid) in a similar manner to 1D for **uniform spacing**

```
glMapGrid2(u_num, u_min, u_max,  
           v_num, v_min, v_max)
```

- Can evaluate as before with **lines** or if want **filled polygons**

```
glEvalMesh2( GL_FILL, u_start,  
            u_num, v_start, v_num)
```

Rendering with Lighting

- If we use **filled** polygons, we have to **shade** or we will see solid color uniform rendering
- Can specify **lights** and **materials** but we need **normals**

- Let OpenGL find them

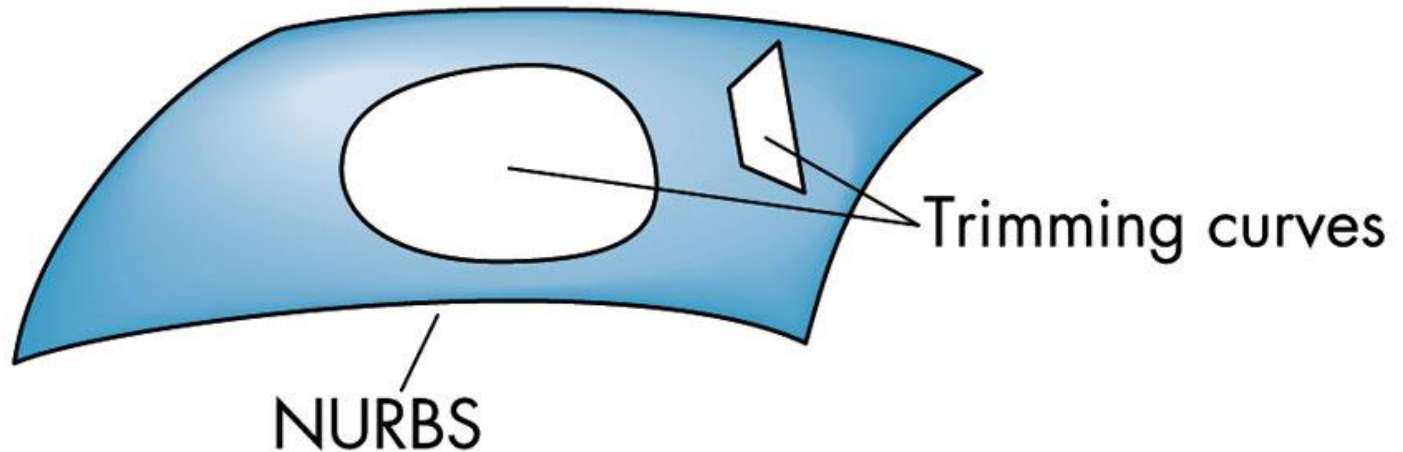
`glEnable(GL_AUTO_NORMAL)`



NURBS

- OpenGL supports **NURBS** surfaces through the **GLU library**
- Why GLU?
 - Can use evaluators in 4D with standard OpenGL library
 - However, there are many complexities with NURBS that need a lot of code
 - There are **five** NURBS surface **functions** plus functions for **trimming curves** that can remove pieces of a NURBS surface

Trimmed Surface

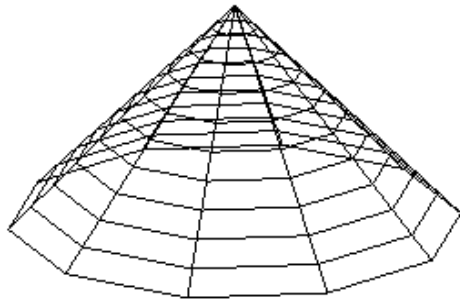


Quadrics

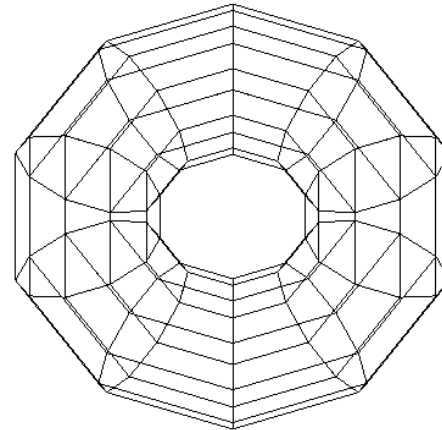
- Quadrics are in both the GLU and GLUT libraries
 - Both use polygonal approximations where the application specifies the resolution
 - Sphere: lines of longitude and latitude
- GLU: disks, cylinders, spheres
 - Can apply transformations to scale, orient, and position
- GLUT: Platonic solids, torus, Utah teapot, cone

GLUT Objects

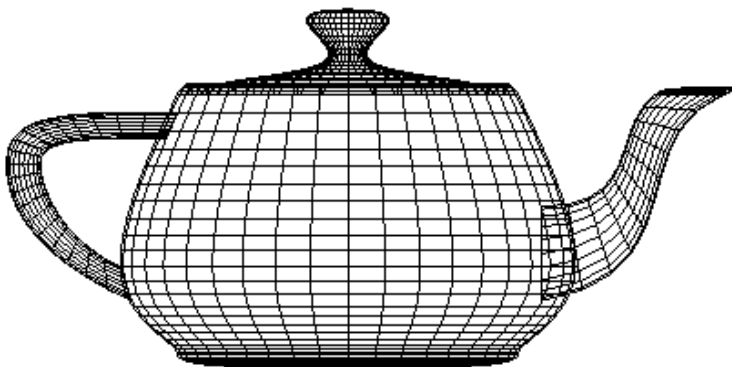
Each has a wire and a solid form



`glutWireCone()`

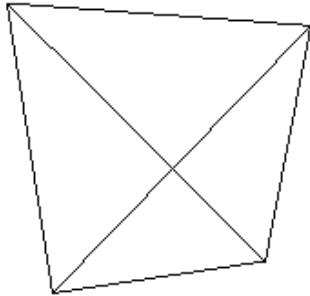


`glutWireTorus()`

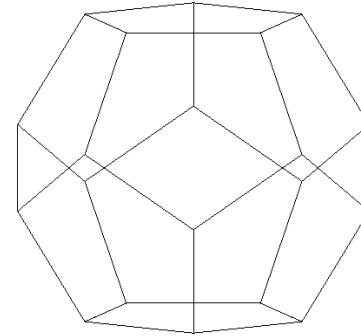


`glutWireTeapot()`

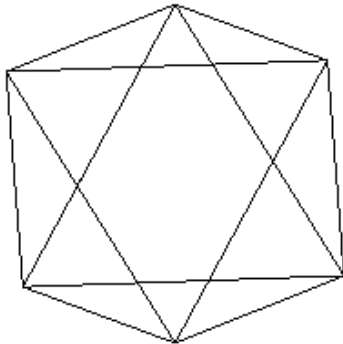
GLUT Platonic Solids



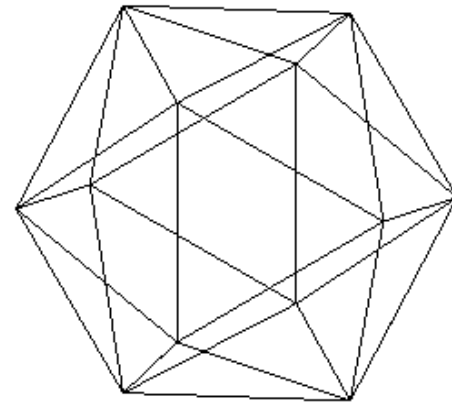
`glutWireTetrahedron()`



`glutWireDodecahedron()`



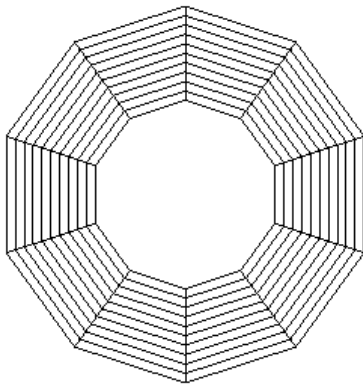
`glutWireOctahedron()`



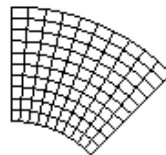
`glutWireIcosahedron()`

Quadric Objects in GLU

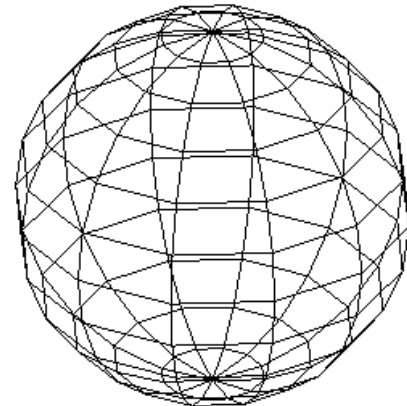
- GLU can automatically generate **normals** and **texture coordinates**
- Quadrics are objects that include properties such as how we would like the object to be rendered



disk



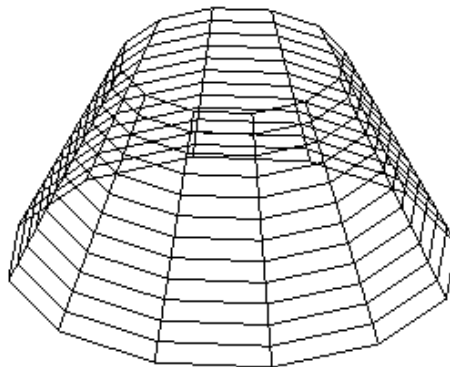
partial disk



sphere

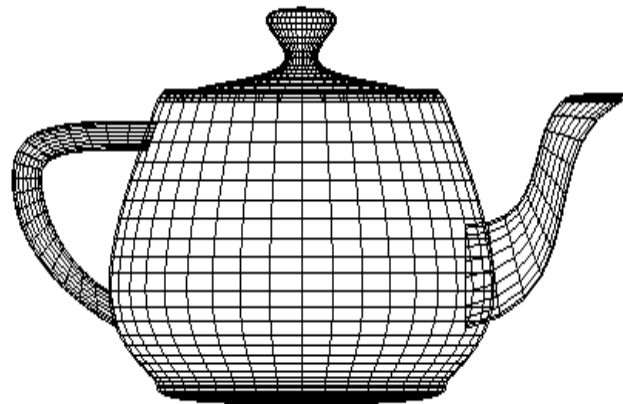
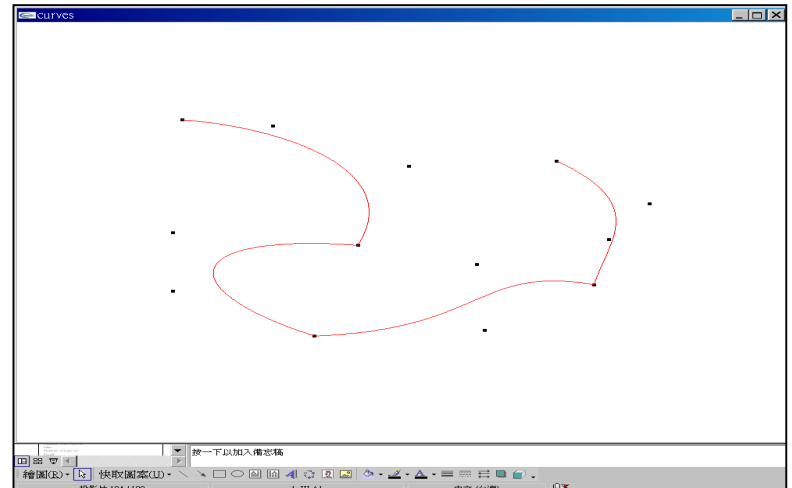
Defining a Cylinder

```
GLUQuadricOBJ *p;  
P = gluNewQuadric(); /*set up object */  
gluQuadricDrawStyle(GLU_LINE); /*render  
style*/  
gluCylinder(p, BASE_RADIUS, TOP_RADIUS,  
            BASE_HEIGHT, sections, slices);
```



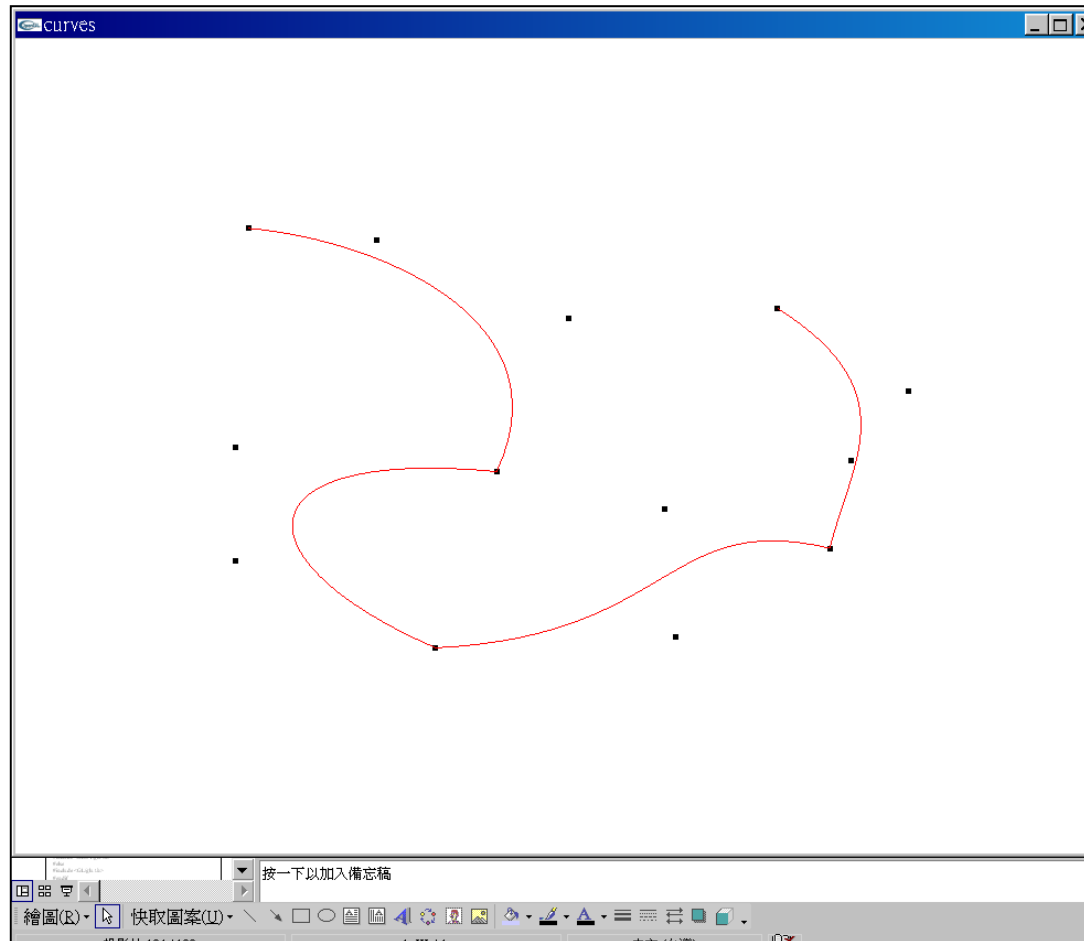
Sample Programs

- Program for Drawing Bezier Curves
 - A.19 curves.c
- Shaded teapot using Bezier surfaces
 - bteapot.c



Program for Drawing Bezier Curves

A.19 curves.c



```
/*  
** curves.c  
**  
** Simple curve drawing program.  
**  
** The following keyboard commands are used to  
control the  
** program:  
**  
** q - Quit the program  
** c - Clear the screen  
** e - Erase the curves  
** b - Draw Bezier curves  
** i - Draw interpolating curves  
** s - Draw B-spline curves  
**  
*/
```

```
#include <stdlib.h>
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif
```

```
typedef enum
{
    BEZIER,
    INTERPOLATED,
    BSPLINE
} curveType;
```

```
void keyboard(unsigned char key, int x, int y);
void computeMatrix(curveType type, float m[4][4]);
void vmult(float m[4][4], float v[4][3], float r[4][3])
```

```
/* Colors to draw them in */
```

```
GLfloat colors[][3] =
```

```
{
```

```
    { 1.0, 0.0, 0.0 },
```

```
    { 0.0, 1.0, 0.0 },
```

```
    { 0.0, 0.0, 1.0 }
```

```
};
```

```
#define MAX_CPTS 25    /* Fixed maximum number of  
                        control points */
```

```
GLfloat cpts[MAX_CPTS][3];
```

```
int ncpts = 0;
```

```
static int width = 500, height = 500;
```

```
/* Window width and height */
```

```
/* Matrix stuff */

/* This routine multiplies two 4 x 4 matrices. */

/* This routine multiplies a 4 x 4 matrix with a vector of 4 points.
*/
void vmult(float m[4][4], float v[4][3], float r[4][3])
{
    int i, j, k;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 3; j++)
            for (k = 0, r[i][j] = 0.0; k < 4; k++)
                r[i][j] += m[i][k] * v[k][j];
}
```



```
/* Interpolating to Bezier matrix */
```

```
static float minterp[4][4] =
{
    { 1.0, 0.0, 0.0, 0.0 },
    { -5.0/6.0, 3.0, -3.0/2.0, 1.0/3.0 },
    { 1.0/3.0, -3.0/2.0, 3.0, -5.0/6.0 },
    { 0.0, 0.0, 0.0, 1.0 },
};
```

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

```
/* B-spline to Bezier matrix */
```

```
static float mbspline[4][4] =
{
    { 1.0/6.0, 4.0/6.0, 1.0/6.0, 0.0 },
    { 0.0, 4.0/6.0, 2.0/6.0, 0.0 },
    { 0.0, 2.0/6.0, 4.0/6.0, 0.0 },
    { 0.0, 1.0/6.0, 4.0/6.0, 1.0/6.0 },
};
```

$$\mathbf{M}_B^{-1} \mathbf{M}_S = \frac{1}{6} \begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix}$$

```
static float midentity[4][4] =  
{  
    { 1.0, 0.0, 0.0, 0.0},  
    { 0.0, 1.0, 0.0, 0.0},  
    { 0.0, 0.0, 1.0, 0.0},  
    { 0.0, 0.0, 0.0, 1.0}  
};
```

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

/* Calculate the **matrix** used to transform the control points */

void computeMatrix(curveType type, float m[4][4])

{

int i, j;

switch (type)

{

case BEZIER:

/* Identity matrix */

for (i = 0; i < 4; i++)

for (j = 0; j < 4; j++)

m[i][j] = m**identity**[i][j];

break;

case INTERPOLATED:

for (i = 0; i < 4; i++)

for (j = 0; j < 4; j++)

m[i][j] = m**interp**[i][j];

break;

A.19 curves.c (8/18)

```
case BSPLINE:
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            m[i][j] = mbspline[i][j];
    break;
}
```

/* Draw the indicated curves using the current control points. */

static void drawCurves(curveType type)

A.19 curves.c (10/18)

{

int i;

int step;

GLfloat newcpts[4][3];

float m[4][4];

/* Set the control point computation matrix and the step size. */

computeMatrix(type, m);

if (type == BSPLINE) step = 1;

else step = 3;

glColor3fv(colors[type]);

```
/* Draw the curves */
```

```
i = 0;
```

```
while (i + 3 < ncpts)
```

```
{
```

```
    /* Calculate the appropriate control points */
```

```
    vmult(m, &cpts[i], newcpts);
```

```
    /* Draw the curve using OpenGL evaluators */
```

```
    glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &newcpts[0][0]);
```

```
    glMapGrid1f(30, 0.0, 1.0);
```

```
    glEvalMesh1(GL_LINE, 0, 30);
```

```
    /* Advance to the next segment */
```

```
    i += step;
```

```
}
```

```
glFlush();
```

```
}
```

```
/* This routine displays the control points */
static void display(void)
{
    int i;
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(0.0, 0.0, 0.0);
    glPointSize(5.0);
    glBegin(GL_POINTS);
        for (i = 0; i < ncpts; i++)
            glVertex3fv(cpts[i]);
    glEnd();

    glFlush();
}
```

/* This routine **inputs new control points** */ **A.19 curves.c (13/18)**

static void mouse(int button, int state, int x, int y)

```
{ float wx, wy;
```

```
    /* We are only interested in left clicks */
```

```
    if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN)
```

```
        return;
```

```
    /* Translate back to our coordinate system */
```

```
    wx = (2.0 * x) / (float)(width - 1) - 1.0;
```

```
    wy = (2.0 * (height - 1 - y)) / (float)(height - 1) - 1.0;
```

```
    /* See if we have room for any more control points */
```

```
    if (ncpts == MAX_CPTS) return;
```

```
    /* Save the point */
```

```
    cpts[ncpts][0] = wx;
```

```
    cpts[ncpts][1] = wy;
```

```
    cpts[ncpts][2] = 0.0;
```

```
    ncpts++;
```



```
/* Draw the point */  
glColor3f(0.0, 0.0, 0.0);  
glPointSize(5.0);  
glBegin(GL_POINTS);  
    glVertex3f(wx, wy, 0.0);  
glEnd();  
  
glFlush();  
}
```

```
/* This routine handles keystroke commands */  
void keyboard(unsigned char key, int x, int y)  
{  
    static curveType lasttype = BEZIER;  
  
    switch (key)  
    {  
        case 'q': case 'Q':  
            exit(0);  
            break;  
        case 'c': case 'C':  
            ncpts = 0;  
            glutPostRedisplay();  
            break;  
        case 'e': case 'E':  
            glutPostRedisplay();  
            break;
```

```
case 'b': case 'B':  
    drawCurves(BEZIER);  
    lasttype = BEZIER;  
    break;  
case 'i': case 'I':  
    drawCurves(INTERPOLATED);  
    lasttype = INTERPOLATED;  
    break;  
case 's': case 'S':  
    drawCurves(BSPLINE);  
    lasttype = BSPLINE;  
    break;  
}  
}
```

```
/* This routine handles window resizes */  
void reshape(int w, int h)  
{  
    width = w;  
    height = h;  
  
    /* Set the transformations */  
    glMatrixMode(GL_PROJECTION);  
    glLoadIdentity();  
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);  
    glMatrixMode(GL_MODELVIEW);  
    glViewport(0, 0, w, h);  
}
```

```
main(int argc, char **argv)
{ /* Intialize the program */
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB);
    glutInitWindowSize(width, height);
    glutCreateWindow("curves");

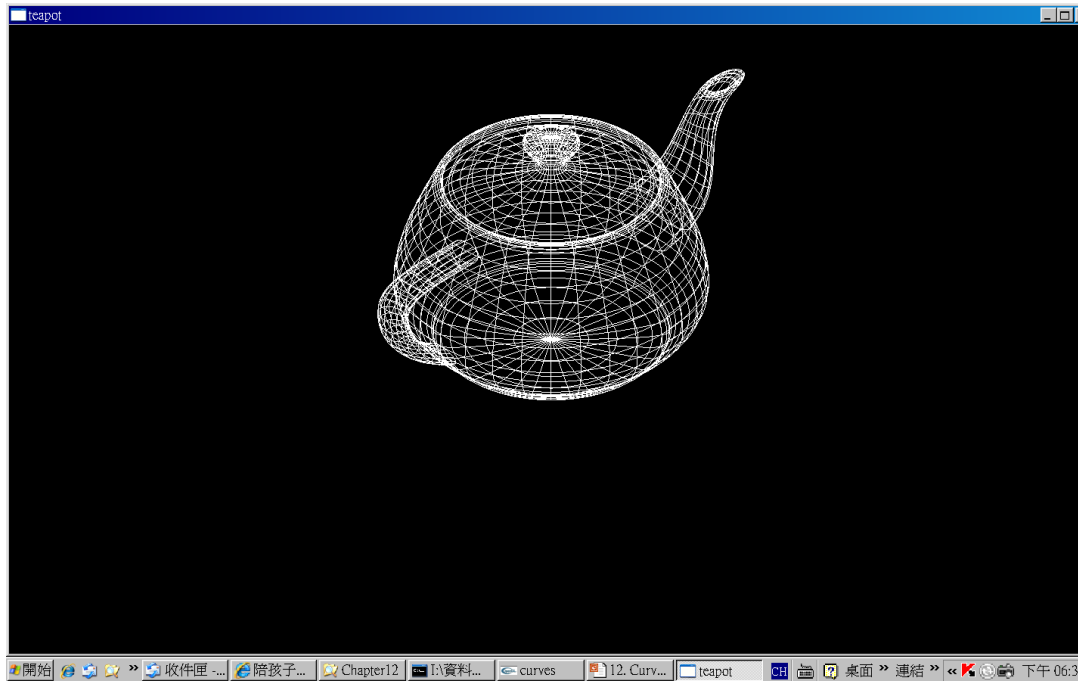
    /* Register the callbacks */
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    glutKeyboardFunc(keyboard);
    glutReshapeFunc(reshape);
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glEnable(GL_MAP1_VERTEX_3);

    glutMainLoop();
}
```

Shaded teapot using Bezier surfaces

bteapot.c (1/7)

- bteapot.c



/* Shaded teapot using Bezier surfaces */

bteapot.c (2/7)

```
#include <stdlib.h>
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif
```

```
typedef GLfloat point[3];
point data[32][4][4];
```

```
/* 306 vertices */
```

```
#include "vertices.h"
```

```
/* 32 patches each defined by 16 vertices, arranged in a 4 x 4 array */
```

```
/* NOTE: numbering scheme for teapot has vertices labeled from 1 to 306 */
```

```
#include "patches.h"
```

bteapot.c (3/7)

```
void display(void)
```

```
{
```

```
    int i, j, k;
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    glColor3f(1.0, 1.0, 1.0);
```

```
    glLoadIdentity();
```

```
    glTranslatef(0.0, 0.0, -10.0);
```

```
    glRotatef(-35.26, 1.0, 0.0, 0.0);
```

```
    glRotatef(-45.0, 0.0, 1.0, 0.0);
```



```

    /* gluLookAt(-1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); */
/* data aligned along z axis, rotate to align with y axis */
glRotatef(-90.0, 1.0,0.0, 0.0);
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(); }
  }
glFlush();
}

```

bteapot.c (4/7)

```
void  
myReshape(int w, int h)  
{  
    glViewport(0, 0, w, h);  
    glMatrixMode(GL_PROJECTION);  
    glLoadIdentity();  
    if (w <= h)  
        glOrtho(-4.0, 4.0, -4.0 * (GLfloat) h / (GLfloat) w,  
                4.0 * (GLfloat) h / (GLfloat) w, -20.0, 20.0);  
    else  
        glOrtho(-4.0 * (GLfloat) w / (GLfloat) h,  
                4.0 * (GLfloat) w / (GLfloat) h, -20.0, 20.0);  
    glMatrixMode(GL_MODELVIEW);  
    display();  
}
```

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

```
main(int argc, char *argv[])
```

bteapot.c (7/7)

```
{  
int i,j, k, m, n;  
for(i=0;i<32;i++) for(j=0;j<4;j++) for(k=0;k<4;k++) for(n=0;n<3;n++)  
{    /* put teapot data into single array for subdivision */  
    m=indices[i][j][k];  
    for(n=0;n<3;n++) data[i][j][k][n]=vertices[m-1][n];  
}  
glutInit(&argc, argv);  
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);  
glutInitWindowSize(500, 500);  
glutCreateWindow("teapot");  
myinit();  
glutReshapeFunc(myReshape);  
glutDisplayFunc(display);  
glutMainLoop();  
}
```