UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

COSC 4370 Fall 2023

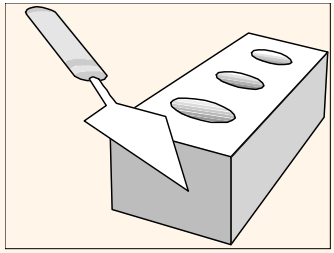
Interactive Computer Graphics

M & W 5:30 to 7:00 PM

Prof. Victoria Hilford

PLEASE TURN your webcam ON

NO CHATTING during LECTURE



COSC 4370

5:30 to 7

PLEASE

LOG IN

CANVAS

Please close all other windows.

11.13.2023 (M 5:30 to 7) (24)	Homework 8	**Lecture 12** **(Curves and Surfaces)**
11.15.2023 (W 5:30 to 7) (25)		**Lecture 13** **(Advanced Rendering)**
11.20.2023 (M 5:30 to 7) (26)		**PROJECT 4**
11.27.2023 (M 5:30 to 7) (27)		**EXAM 4 REVIEW**
11.29.2023 (M 5:30 to 7) (28)		**EXAM 4**
12.11.2023 (M 5:30 to 7)		**FINAL EXAM**

COSC 4370 – Computer Graphics

Lecture 12

Curves and Surfaces

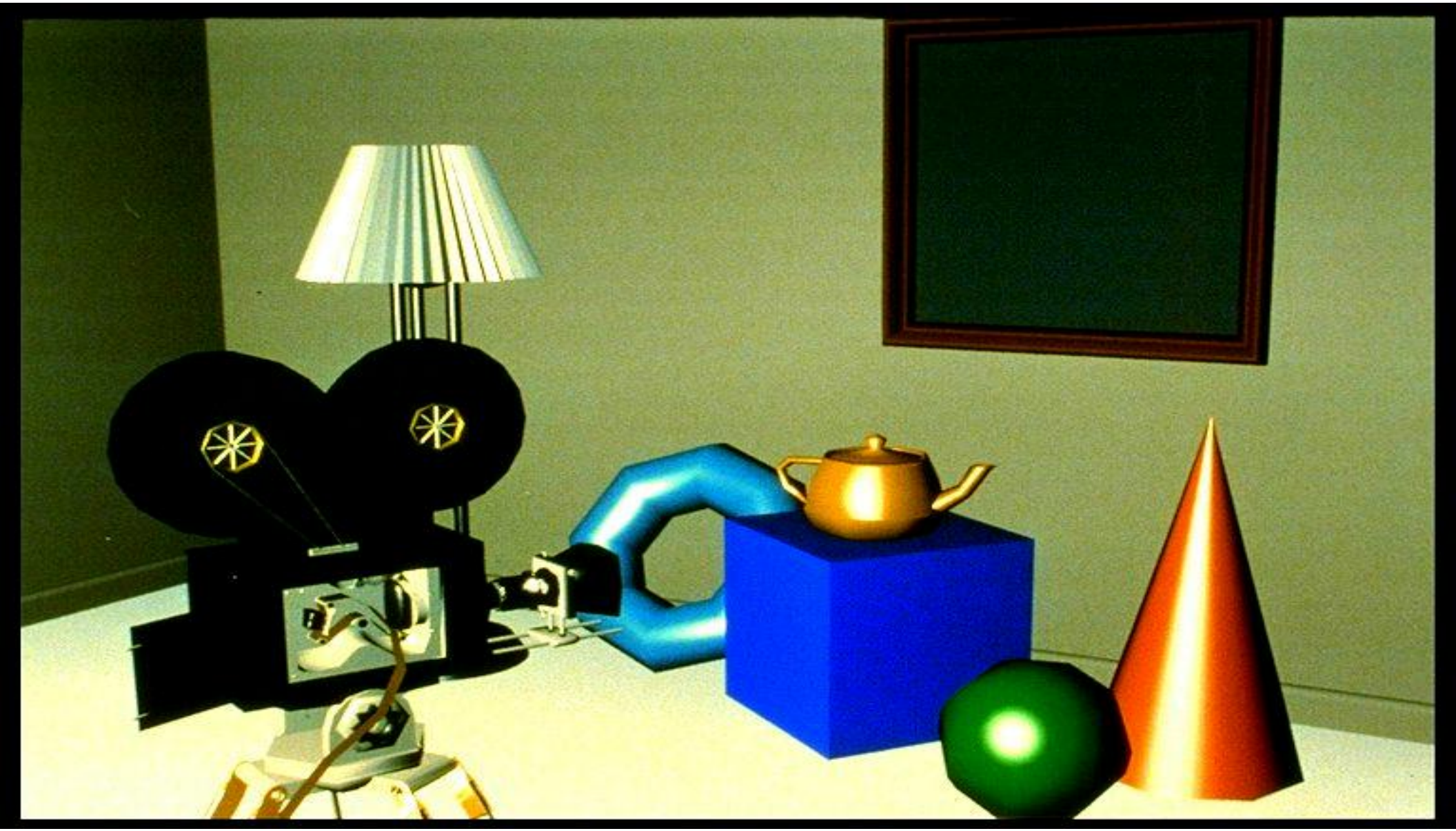
Chapter 12

Curves and Surfaces

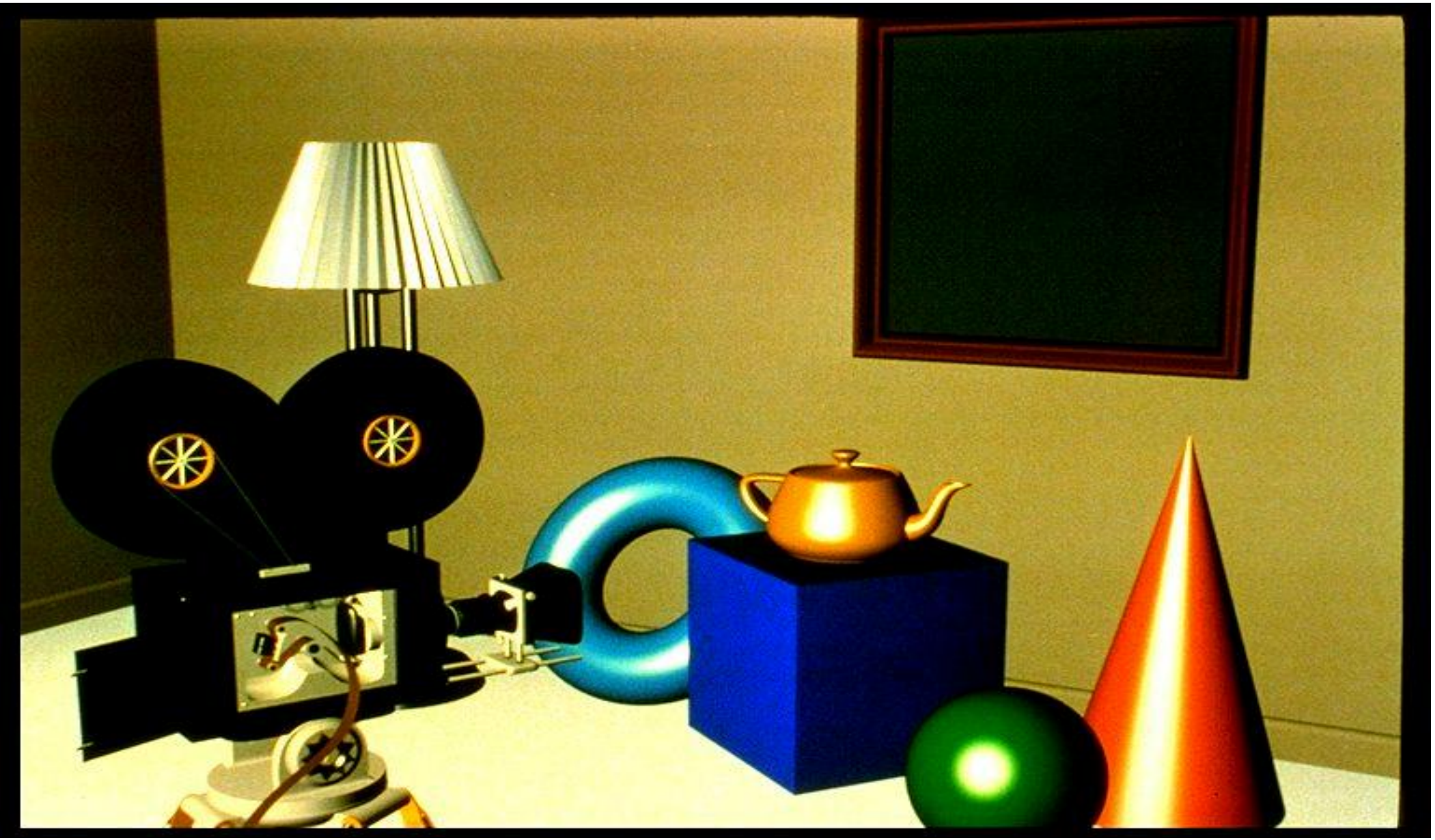
The world around us is full of objects of remarkable shapes. Nevertheless, in computer graphics, we continue to populate our virtual worlds with flat objects. We have a good reason for such persistence. Graphics systems can render flat three-dimensional polygons at high rates, including doing hidden-surface removal, shading, and texture mapping. We could take the approach that we took with our sphere model and define curved objects that are, in (virtual) reality, collections of flat polygons. **Alternatively, and as we will do here, we can provide the application programmer with the means to work with curved objects in her program**, leaving the eventual rendering of these objects to the implementation.

We introduce **three ways to model curves and surfaces, paying most attention to the parametric polynomial forms.** We also discuss how **curves and surfaces** can be **rendered on current graphics systems, a process that usually involves subdividing the curved objects into collections of flat primitives**. From the application programmer's perspective, this process is transparent because it is part of the implementation. It is important to understand the work involved, however, so that we can appreciate the practical limitations we face in using **curves and surfaces**.

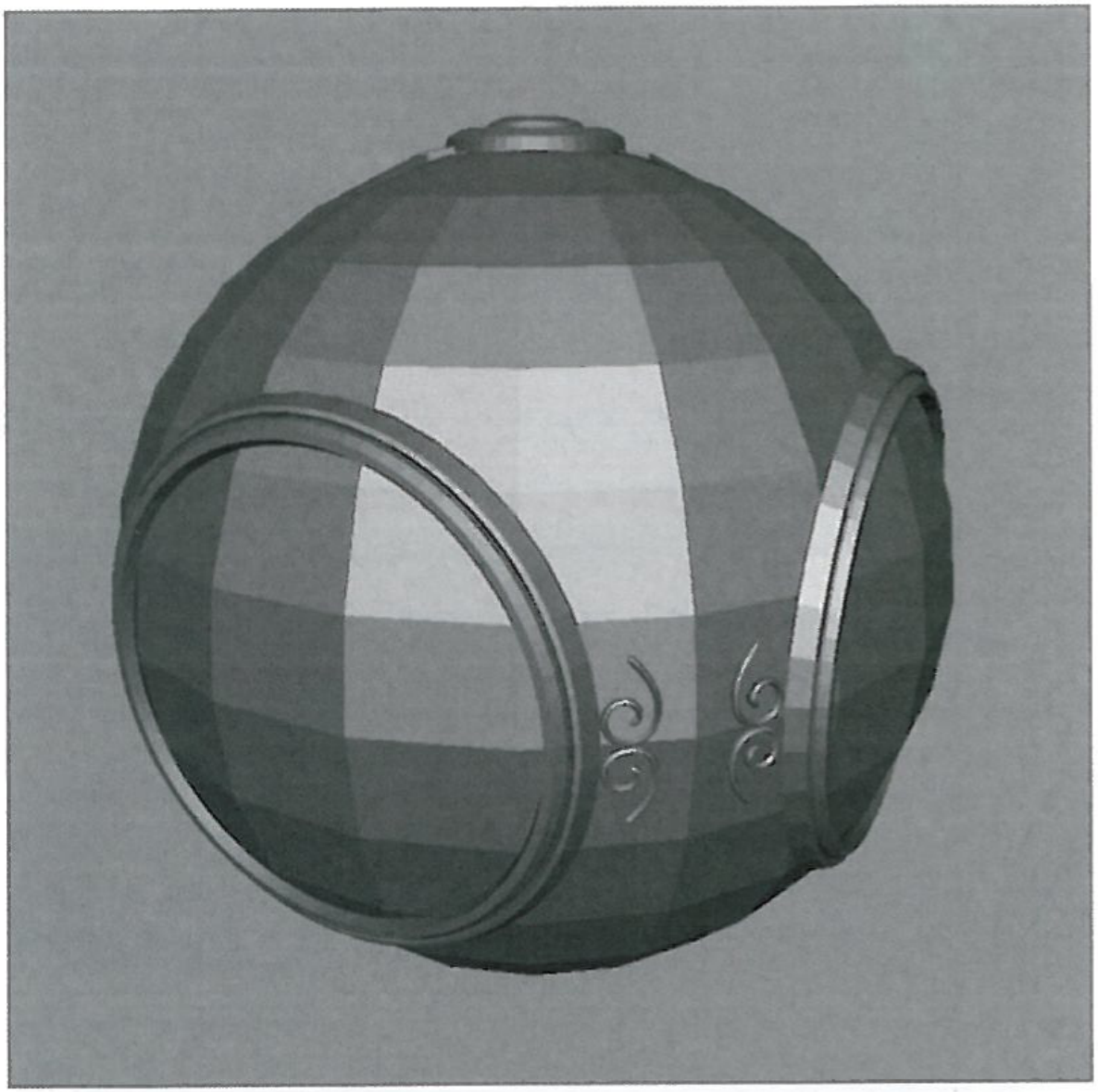
Polygonal Surfaces



Curved Surfaces



Polygonal Surfaces



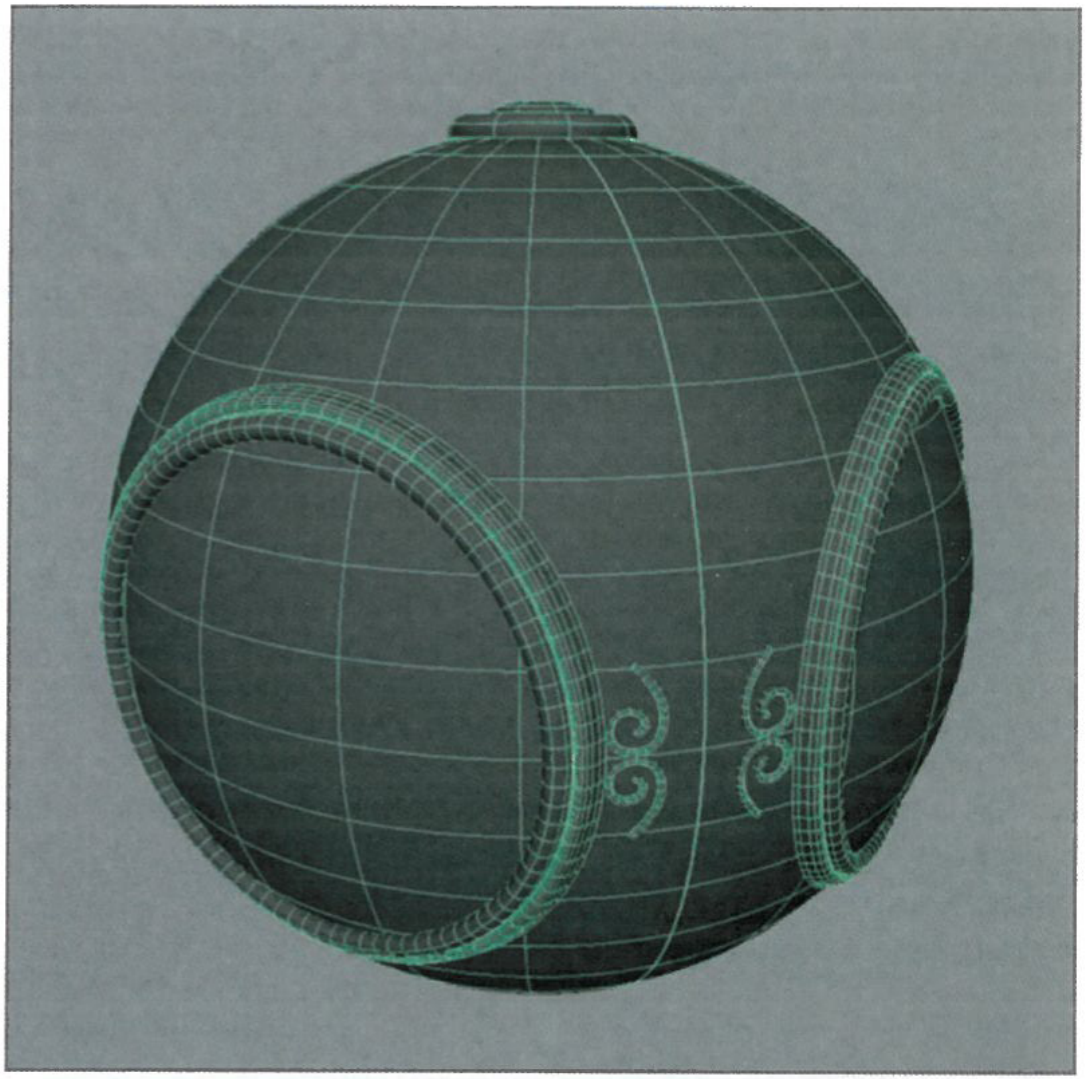
Color Plate 3 Flat-shaded polygonal rendering of sun object.

(Courtesy of Full Dome Project, University of New Mexico.)

Curved Surfaces

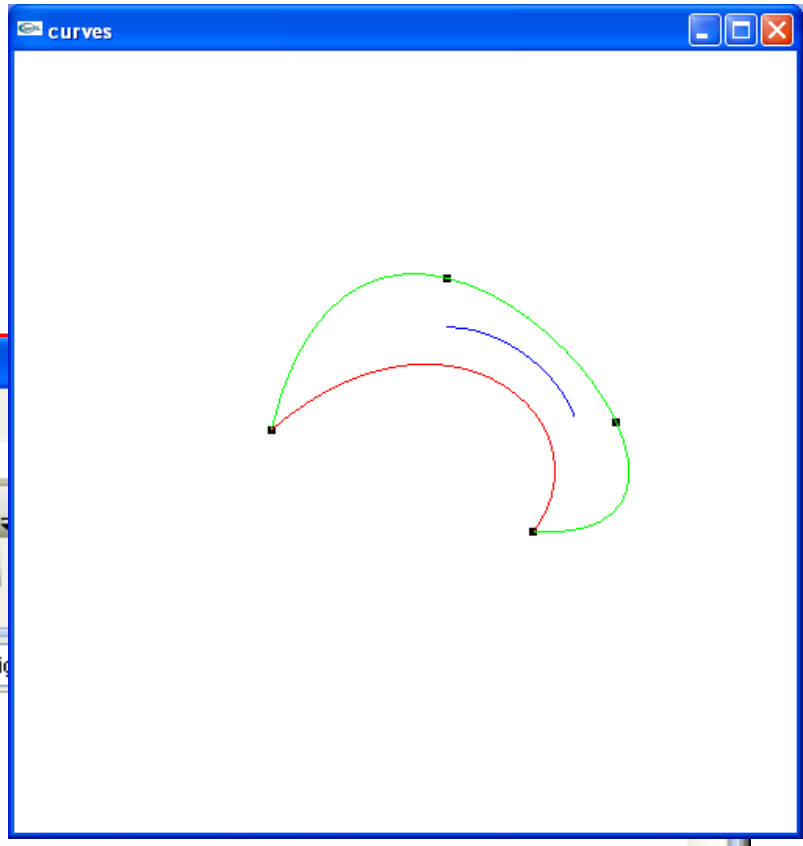
Color Plate 5 Wire-frame of NURBS representation of sun object showing the high number of polygons used in rendering the NURBS surfaces.

(Courtesy of Full Dome Project, University of New Mexico.)



NURBS - NonUniform Rational B-Spline Curve

curves.c



- Microsoft Visual Studio

Build Debug Tools Visual Assert Test Window Help

Debug Win32 texenv

Hex

Thread: Stack Frame:

curves.c

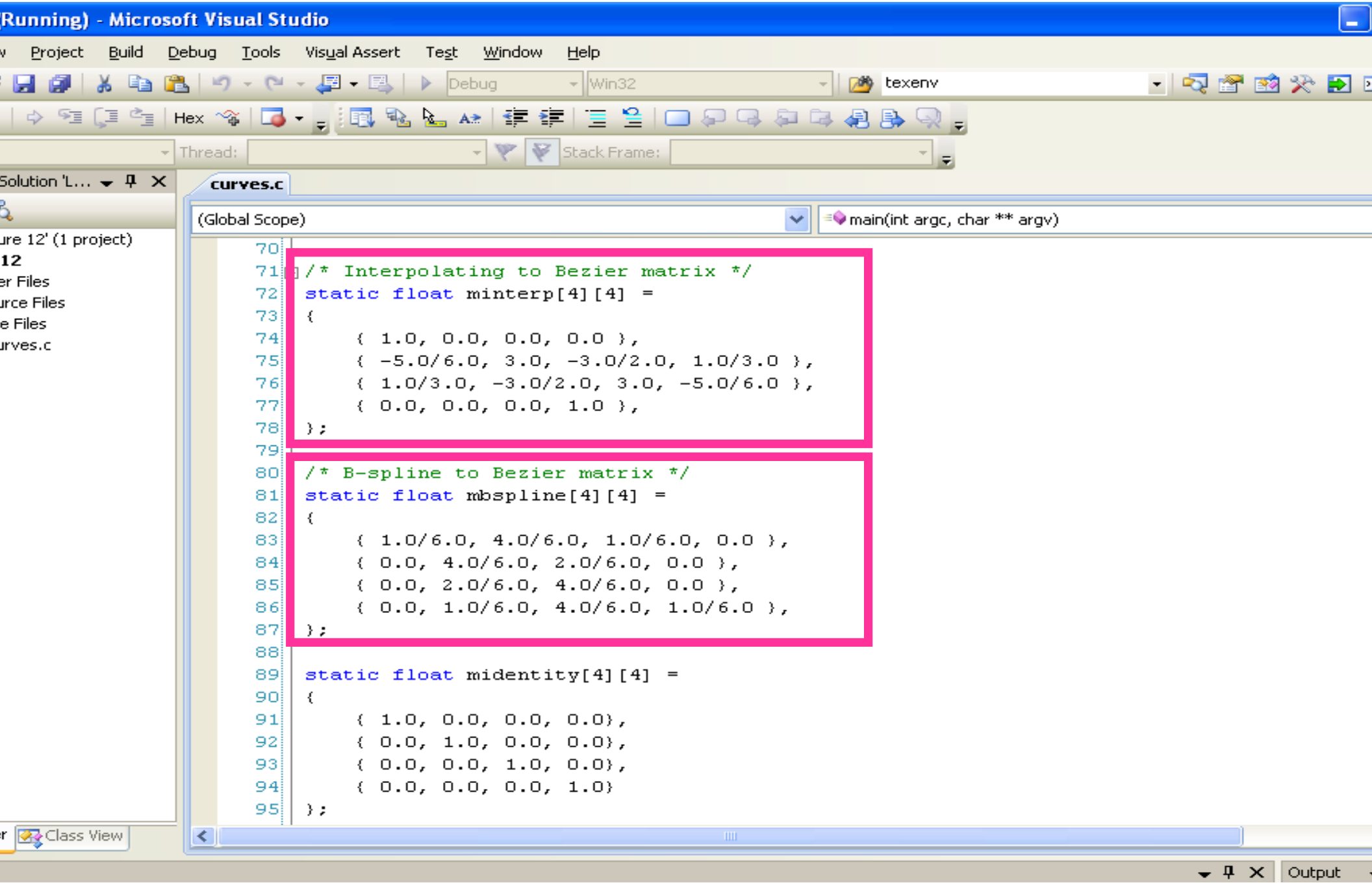
(Global Scope) keyboard(unsi

```
217     switch (key)
218     {
219         case 'q': case 'Q':
220             exit(0);
221             break;
222         case 'c': case 'C':
223             ncpts = 0;
224             glutPostRedisplay();
225             break;
226         case 'e': case 'E':
227             glutPostRedisplay();
228             break;
229         case 'b': case 'B':
230             drawCurves(BEZIER);
231             lasttype = BEZIER;
232             break;
233         case 'i': case 'I':
234             drawCurves(INTERPOLATED);
235             lasttype = INTERPOLATED;
236             break;
237         case 's': case 'S':
238             drawCurves(BSPLINE);
239             lasttype = BSPLINE;
240             break;
241     }
242 }
```

RUN IT!

curves.c

curves.c


(Running) - Microsoft Visual Studio
Project Build Debug Tools Visual Assert Test Window Help
Debug
Win32
texenv
Hex
Thread:
Stack Frame:
Solution 'L...
ure 12' (1 project)
12
er Files
urce Files
e Files
urves.c
Class View
curves.c
(Global Scope)
main(int argc, char ** argv)
70
71 /* Interpolating to Bezier matrix */
72 static float minterp[4][4] =
73 {
74 { 1.0, 0.0, 0.0, 0.0 },
75 { -5.0/6.0, 3.0, -3.0/2.0, 1.0/3.0 },
76 { 1.0/3.0, -3.0/2.0, 3.0, -5.0/6.0 },
77 { 0.0, 0.0, 0.0, 1.0 },
78 };
79
80 /* B-spline to Bezier matrix */
81 static float mbspline[4][4] =
82 {
83 { 1.0/6.0, 4.0/6.0, 1.0/6.0, 0.0 },
84 { 0.0, 4.0/6.0, 2.0/6.0, 0.0 },
85 { 0.0, 2.0/6.0, 4.0/6.0, 0.0 },
86 { 0.0, 1.0/6.0, 4.0/6.0, 1.0/6.0 },
87 };
88
89 static float midentity[4][4] =
90 {
91 { 1.0, 0.0, 0.0, 0.0},
92 { 0.0, 1.0, 0.0, 0.0},
93 { 0.0, 0.0, 1.0, 0.0},
94 { 0.0, 0.0, 0.0, 1.0}
95 };
Output

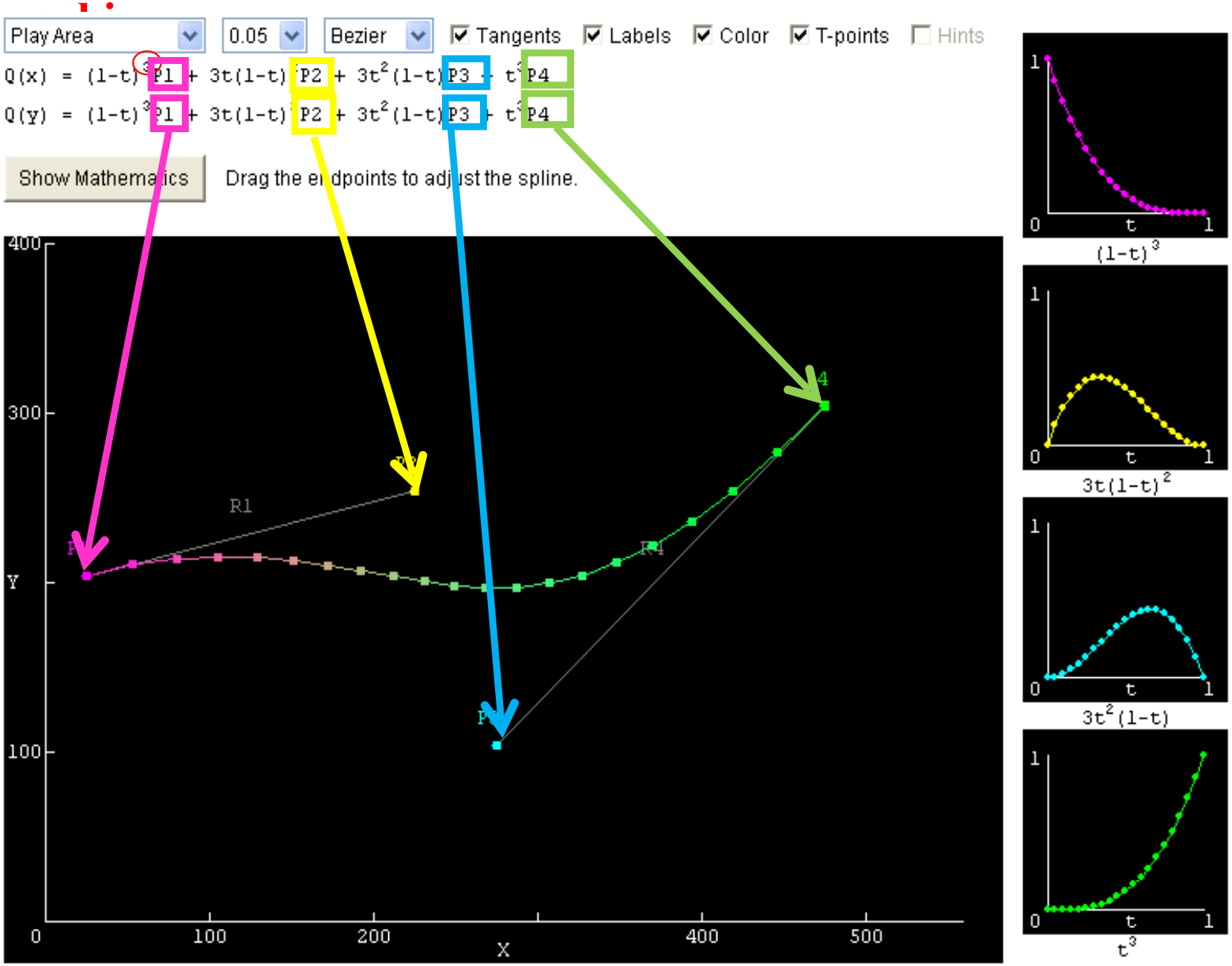
curves.c

curves.c*

(Global Scope)

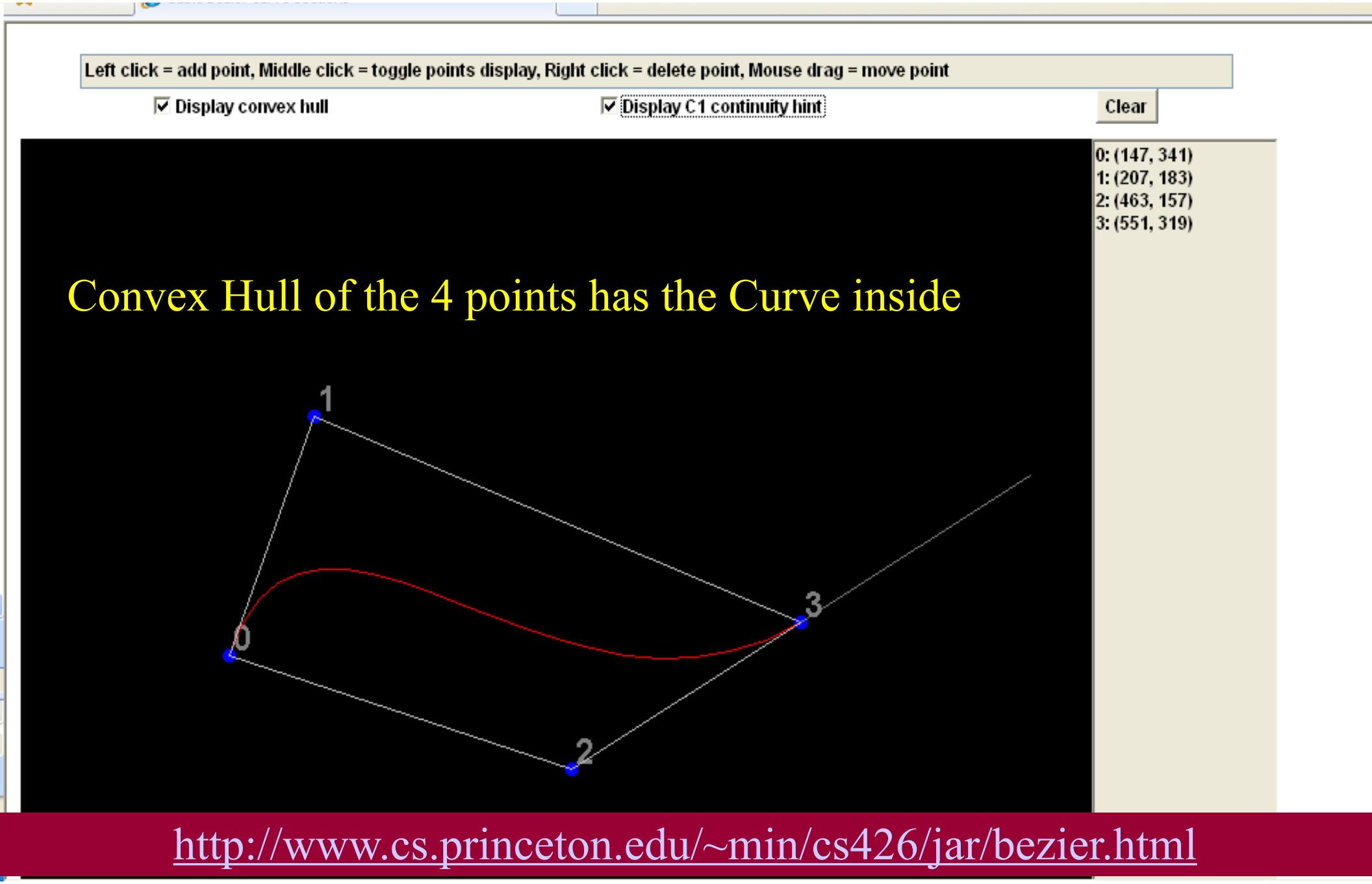
```
 5  **
 6  **  The following keyboard commands are used to control the program:
 7  **
 8  **    q - Quit the program
 9  **    c - Clear the screen
10  **    e - Erase the curves
11  **    b - Draw Bezier curves
12  **    i - Draw interpolating curves
13  **    s - Draw B-spline curves
14  */
15
16  #include <stdlib.h>
17
18  #ifdef __APPLE__
19  #include <GLUT/glut.h>
20  #else
21  #include <GL/glut.h>
22  #endif
23
24  typedef enum
25  {
26      BEZIER,
27      INTERPOLATED,
28      BSPLINE
29  } curveType;
30
```

Bezier Splines Applet

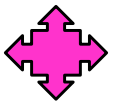


http://www.cs.brown.edu/exploratories/freeSoftware/repository/edu/brown/cs/exploratories/applets/bezierSplines/bezier_splines_java_brov

Bezier Applet



http://www.cs.princeton.edu/~min/cs426/jar/bezier.html



Objectives

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

Objectives

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

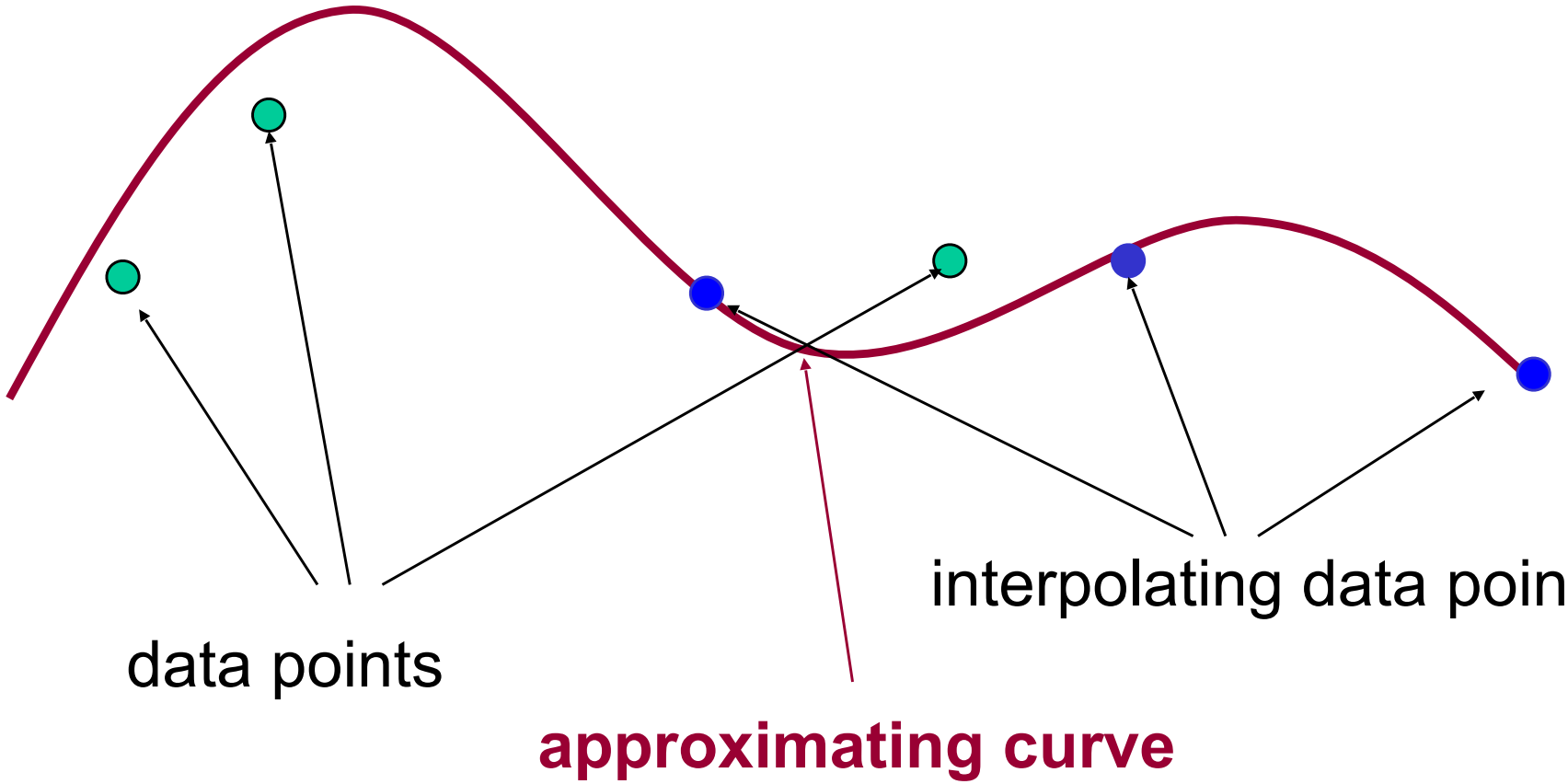
Escaping Flatland

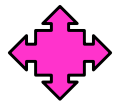
- Until now we have worked with flat entities such as lines and flat polygons
  - **Fit well with graphics hardware**
  - **Mathematically simple**

- 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 fl primitives

Modeling with Curves





What Makes a Good Representation?

- There are many ways to represent curves and surfaces

- Want a representation that is

  Stable

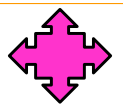
  Smooth

  Easy to evaluate

  Must we interpolate or can we just come close to data?

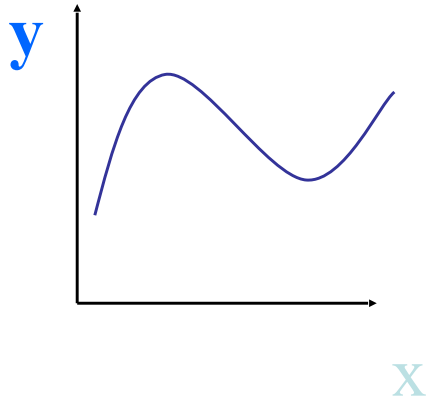
  Do we need derivatives?

Explicit Representation
2D & 3D Curves & Surfaces



- Most familiar form of **curve in 2D**

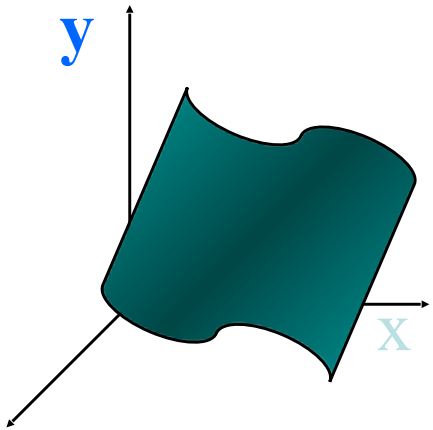
$$y=f(x)$$



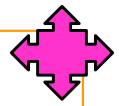
- **Cannot represent all curves**
  - **Vertical lines**
  - **Circles**

- **Curve** Extension to **3D**

$$y=f(x),\ z=g(x)$$



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



Implicit Representation
2D & 3D Curves & Surfaces

Two dimensional **2D curve**(s)

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

Much more **robust**

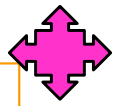
– All **lines** $ax+by+c=0$

– **Circles** $x^2+y^2-r^2=0$

Three dimensions **3D** $g(x,y,z)=0$ defines a **surface**

– Intersect two **surfaces** to get a **curve**

In general, we **cannot** solve for **Points** that satisfy



Parametric Representation
2D & 3D Curves & Surfaces

Curves: Separate equation **3D** for each spatial **parameter**

$$x=x(u)$$
$$y=y(u)$$
$$z=z(u)$$

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

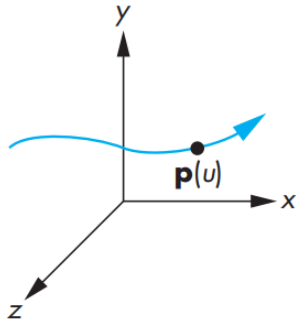


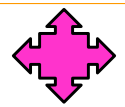
FIGURE 12.1 Parametric curve.

Surfaces require **2 parameters**

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

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

Algebraic Surface



- **Quadric surface** $2 \geq i+j+k$
- At most 10 terms

- **Can solve intersection with a ray** by reducing problem to solving **quadratic equation**

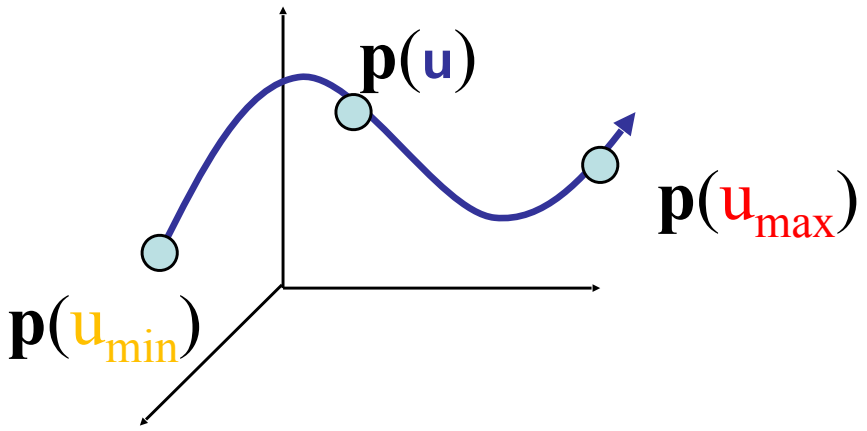
Parametric Curves

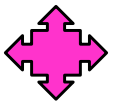
Separate equation **3D** for each spatial variable

$$x = x(u)$$
$$y = y(u)$$
$$z = z(u)$$

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

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





Selecting Functions

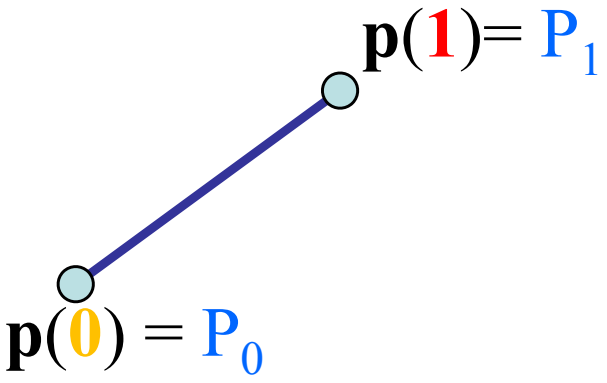
- Usually we can select **"good" functions**
  - Not unique for a given **spatial curve**
  - **Approximate or interpolate known data**
  - Want **functions** which are easy to evaluate
  - Want **functions** which are easy to differentiate
    - **Computation of normals**
    - **Connecting pieces (segments)**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

Parametric Lines

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

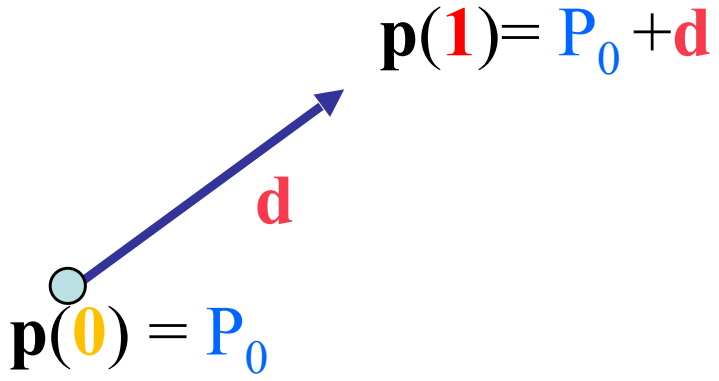
Line connecting two Points P_0 and P_1

$$\mathbf{p}(\mathbf{u})=(1-\mathbf{u})P_0+\mathbf{u}P_1$$



Ray from P_0 in the direction **d**

$$\mathbf{p}(\mathbf{u})=P_0+\mathbf{u}\mathbf{d}$$

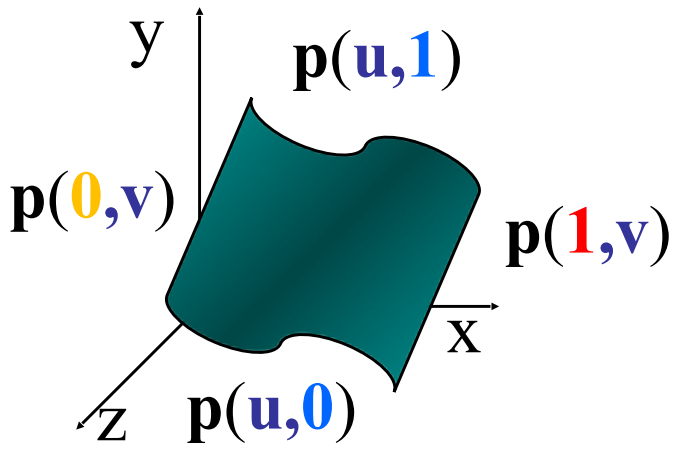


Parametric Surfaces

- **Surfaces** require **2 parameters**

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

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



Surface patch

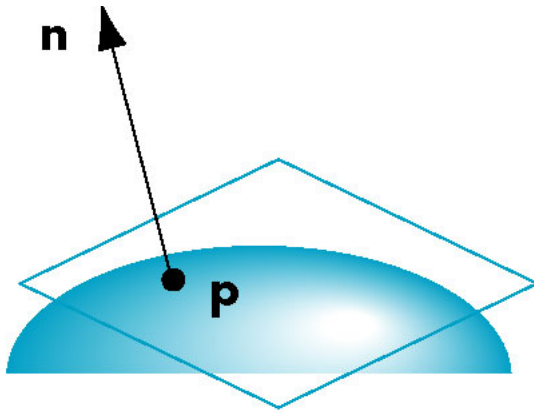
- Want same properties as **curves**:
  - **Smoothness**
  - **Differentiability**
  - **Ease of evaluation**

Normals n

We can **differentiate** with respect to **u** and **v** to obtain the normal **n** at any Point P

n **x**

x cross product



Tangent plane and normal at a point on a **parametric surface**.

Parametric **Planes**

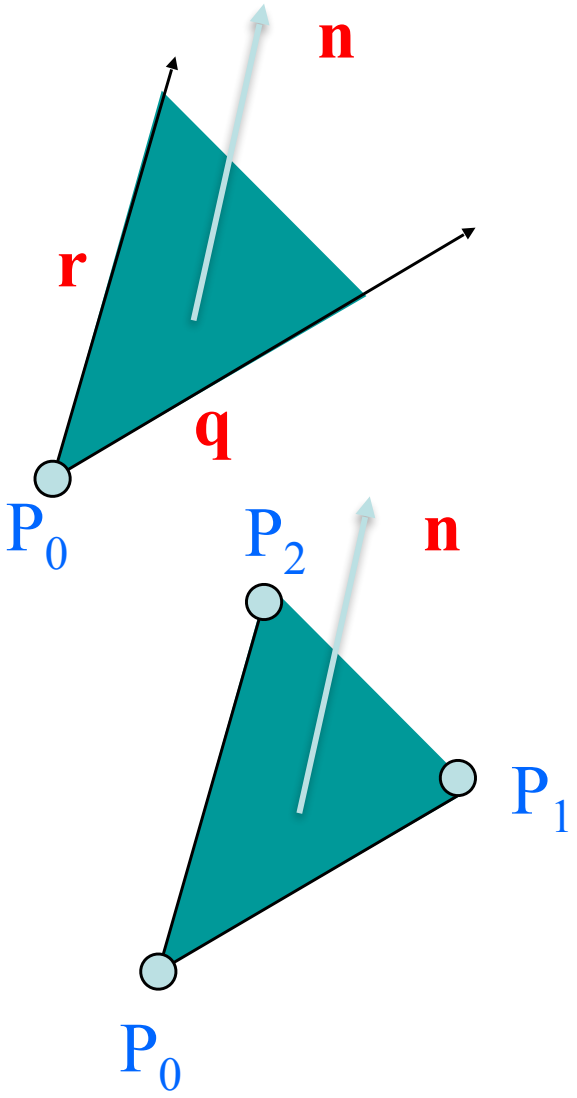
Point-**vectors** form

$$\mathbf{p}(u,v)=P_0+u\mathbf{q}+v\mathbf{r}$$

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

3-Point form

$$\mathbf{q} = P_1 - P_0$$
$$\mathbf{r} = P_2 - P_0$$

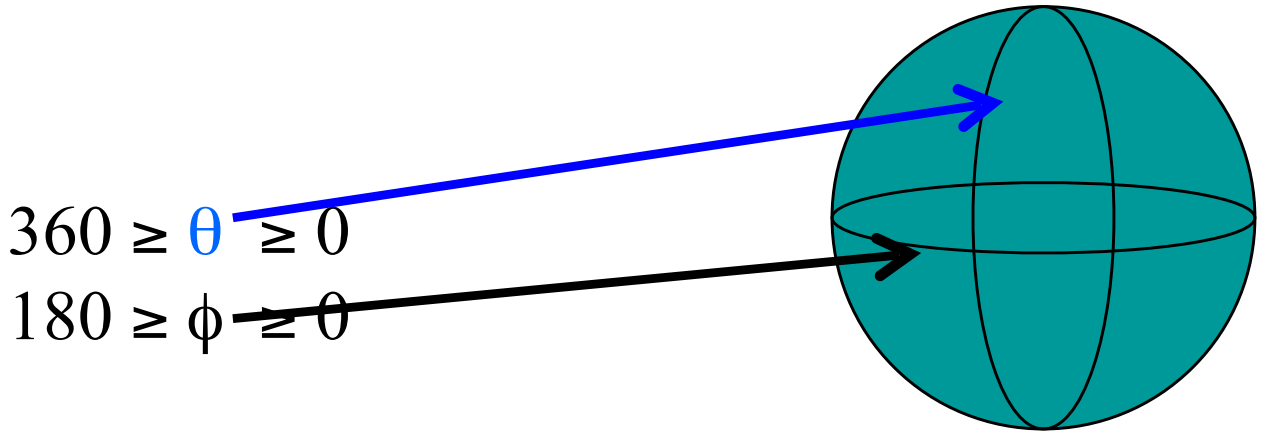


Parametric Sphere

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

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

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



θ constant: circles of constant longitude

ϕ constant: circles of constant latitude

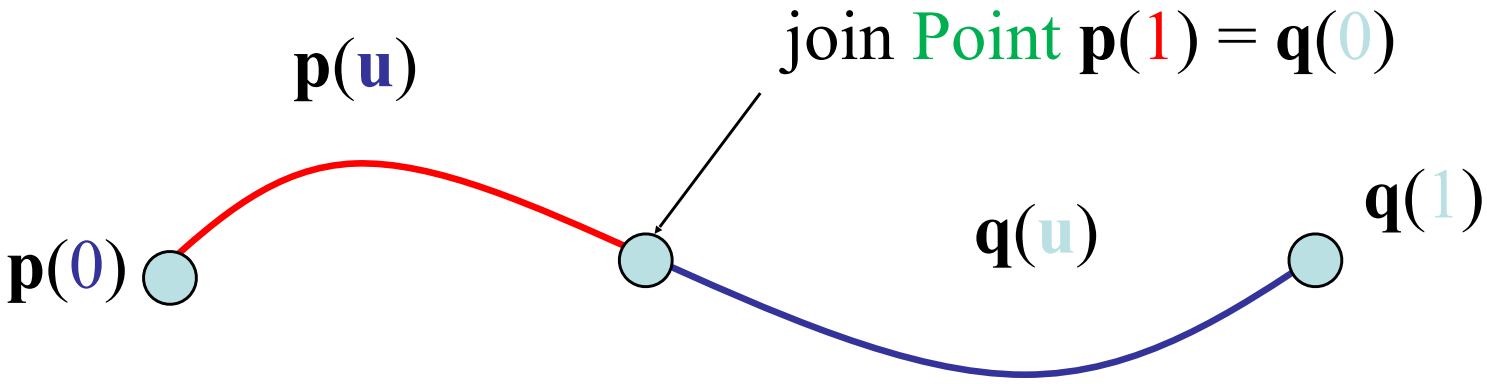
differentiate to show **n** = **p**

Curve Segments

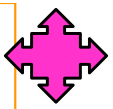
- After normalizing **u**, each **curve** is written

  $\mathbf{p}(u)=[x(u), y(u), z(u)]^T, \quad 1 \geq u \geq 0$

- In **classical numerical methods**, we design a **single global curve**
- In **computer graphics** and **CAD**, it is better to design small **connected curve *segments***



Parametric Polynomial **Curves**



N	M	L

- If **N=M=L**, we need to determine **3(N+1) coefficients (3 x,y,z)**
- 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

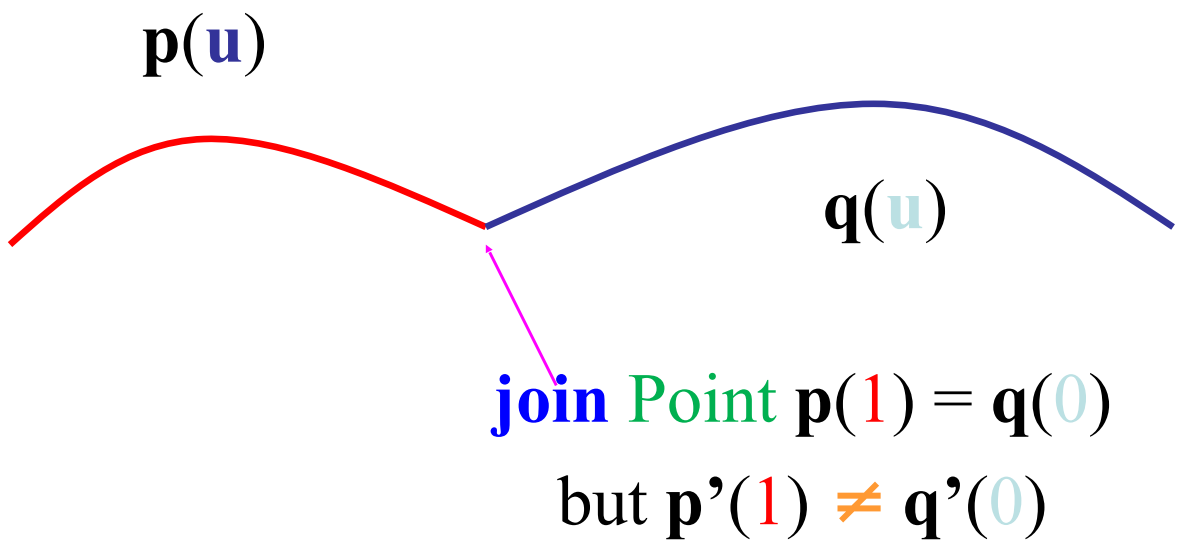
  where **p** can be any of x, y, z

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=L=3, gives balance between **ease of evaluation** and **flexibility in design**

Four coefficients c_k 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**

Cubic Parametric Polynomials

Expand p(u)

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

Cubic Polynomial Surfaces

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

where

$\mathbf{u},\mathbf{v}$

$\mathbf{p}$ is any of x, y or z

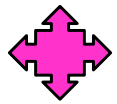
Need 48 coefficients (**3 independent** sets of (3+1)(3+1)) to determine a **surface patch**

Objectives

- Introduce types of **curves** and **surfaces**
  - **Explicit**
  - **Implicit**
  - **Parametric**
  - Strengths and weaknesses

- Discuss **Modeling** and **Approximations**
  - Conditions
  - Stability

Designing Parametric Cubic Curves

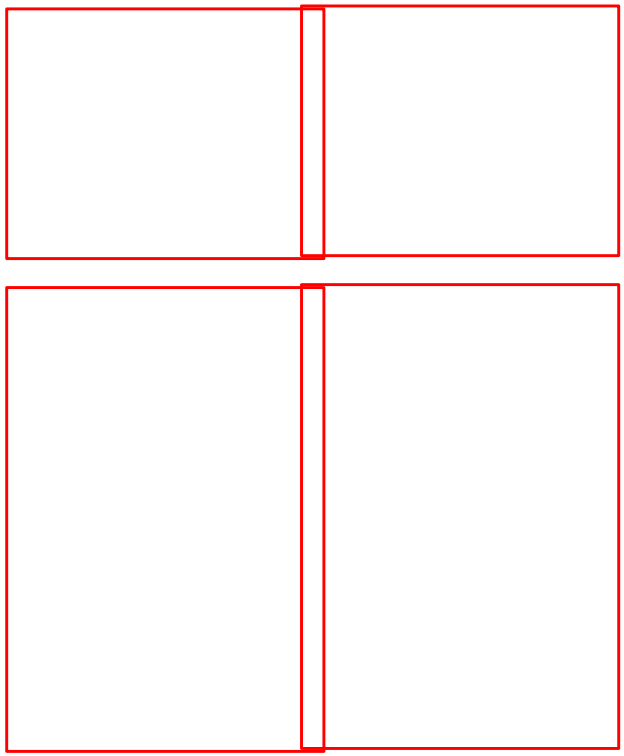


Objectives

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

Matrix Column-Vector Form

A parametric polynomial curve of degree 3



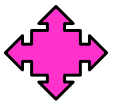
Each c_k has independent x,y, z components:

define

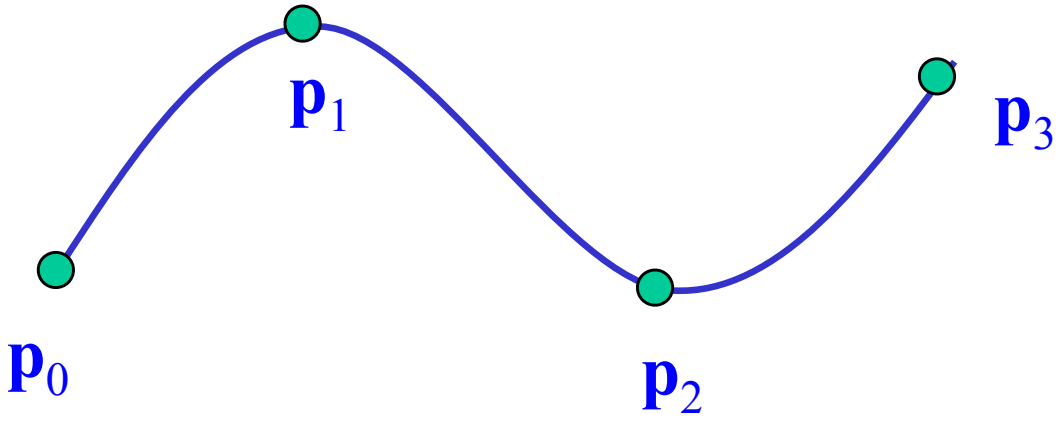
$$c_k = \begin{bmatrix} c_{xk} \\ c_{yk} \\ c_{zk} \end{bmatrix}.$$

then

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



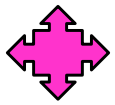
Interpolating Curve



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

$$p(u) = c_0 + c_1u + c_2u^2 + c_3u^3$$



Interpolation Equations

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

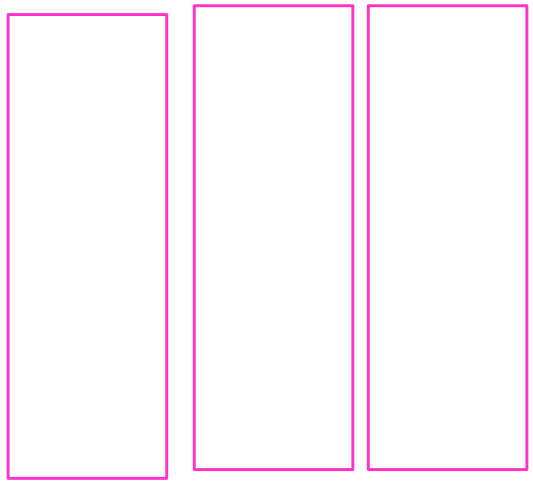
apply the **interpolating conditions** at $u = 0, 1/3, 2/3, 1$

$$p_0 = p(0) = c_0$$
$$p_1 = p(1/3) = c_0 + (1/3)c_1 + (1/3)^2 c_2 + (1/3)^3 c_3$$
$$p_2 = p(2/3) = c_0 + (2/3)c_1 + (2/3)^2 c_2 + (2/3)^3 c_3$$
$$p_3 = p(1) = c_0 + c_1 + c_2 + c_3$$

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

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

$\mathbf{p=Ac}$



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

p=Ac

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

to obtain the **interpolating geometry matrix**

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

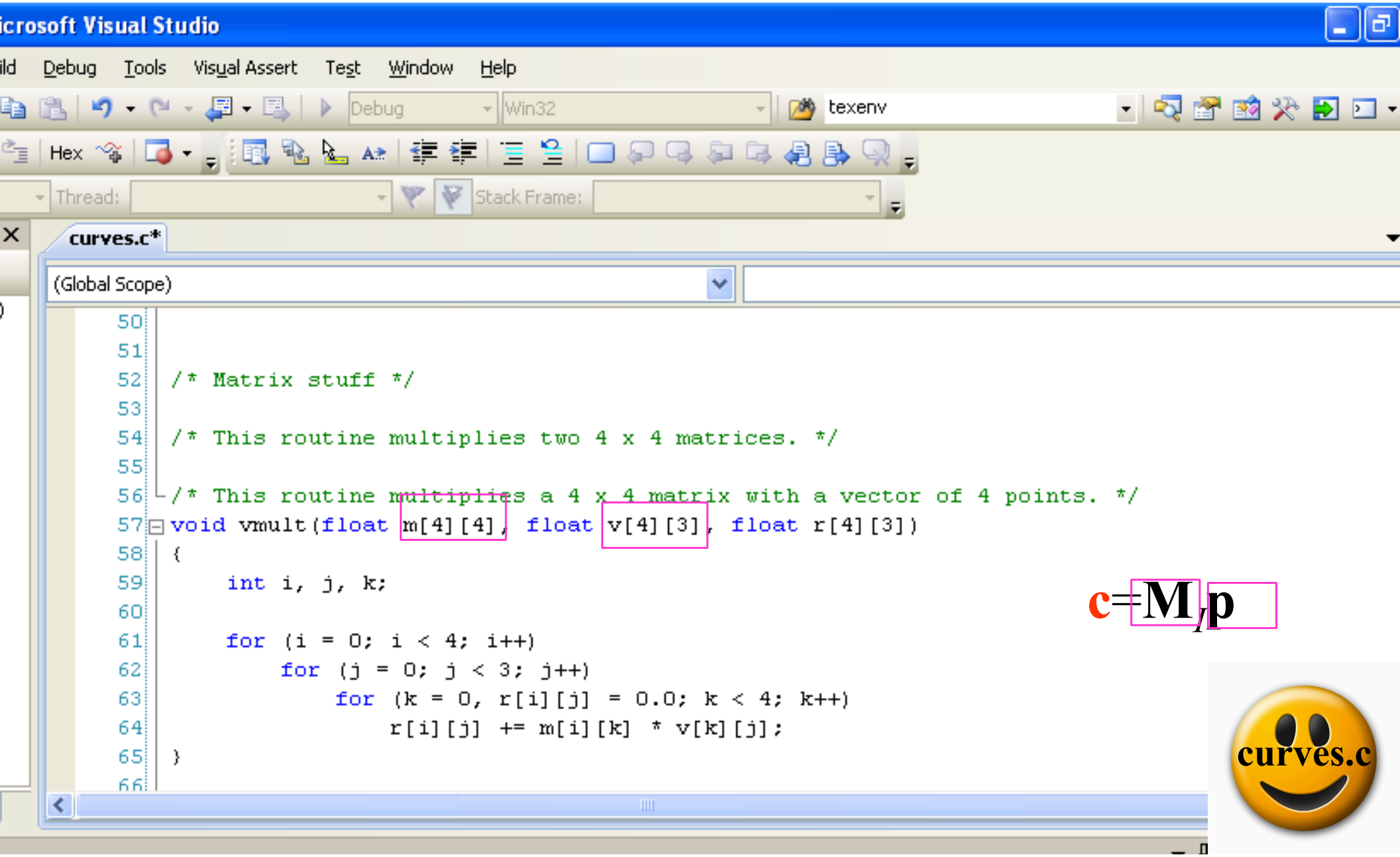
and the desired coefficients

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

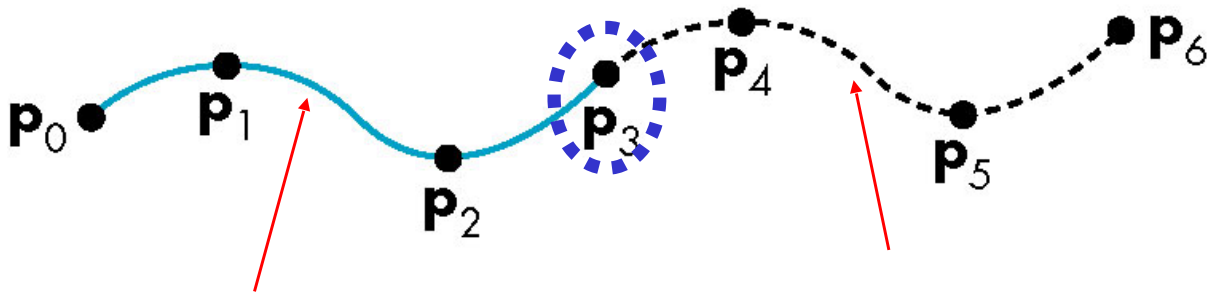
$$\mathbf{p} = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

Interpolation Matrix

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



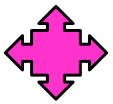
Interpolating Multiple Segments



use $\mathbf{p} = [p_0\ p_1\ p_2\ \mathbf{p_3}]^T$ use $\mathbf{p} = [\mathbf{p_3}\ p_4\ p_5\ p_6]^T$

Get **continuity** at join Points but
not continuity of **derivatives**

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



Blending Functions

Rewriting the equation for p(u)

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

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

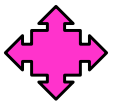
where $b(u)^T = [b_0(u)\ b_1(u)\ b_2(u)\ b_3(u)]^T$ is an array of *blending polynomials* such that

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

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

$$b_0(u) = -4.5(u-1/3)(u-2/3)(u-1)$$
$$b_1(u) = 13.5u\,(u-2/3)(u-1)$$
$$b_2(u) = -13.5u\,(u-1/3)(u-1)$$
$$b_3(u) = 4.5u\,(u-1/3)(u-2/3)$$

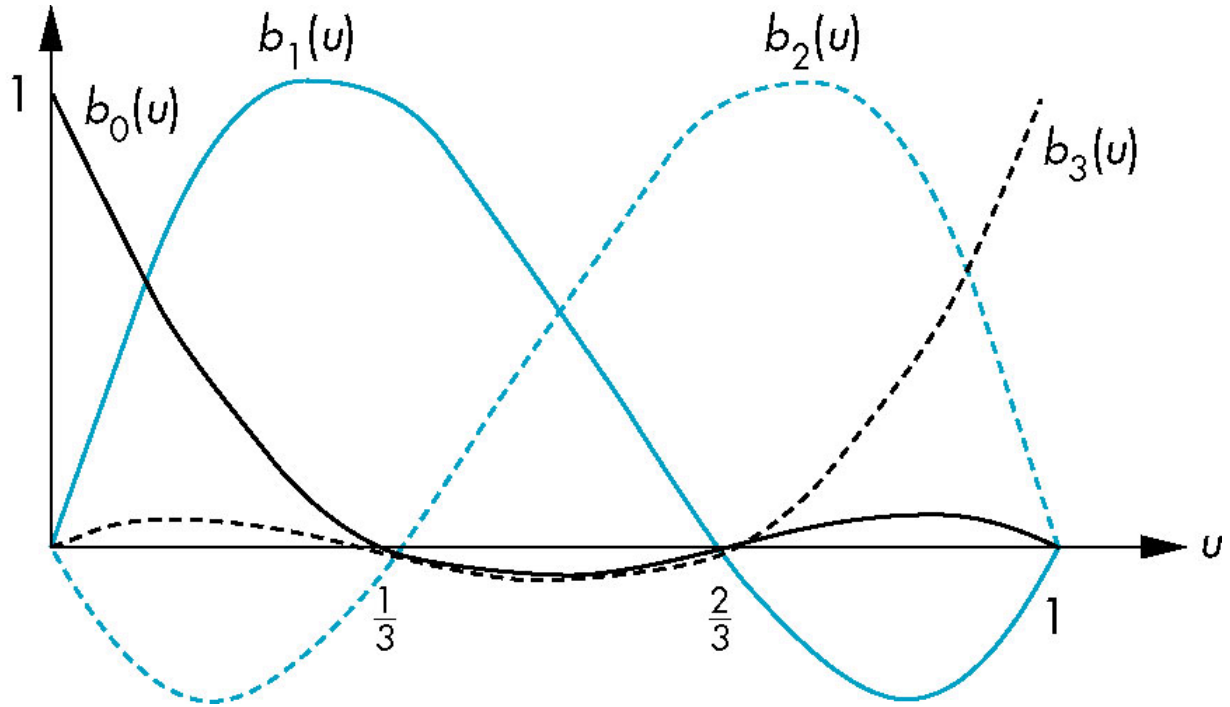
$$\mathbf{b}(u) = \begin{bmatrix} b_0(u) \\ b_1(u) \\ b_2(u) \\ b_3(u) \end{bmatrix}.$$



Blending Functions

- These functions are **not smooth**

  Hence the **interpolation polynomial is not smooth**



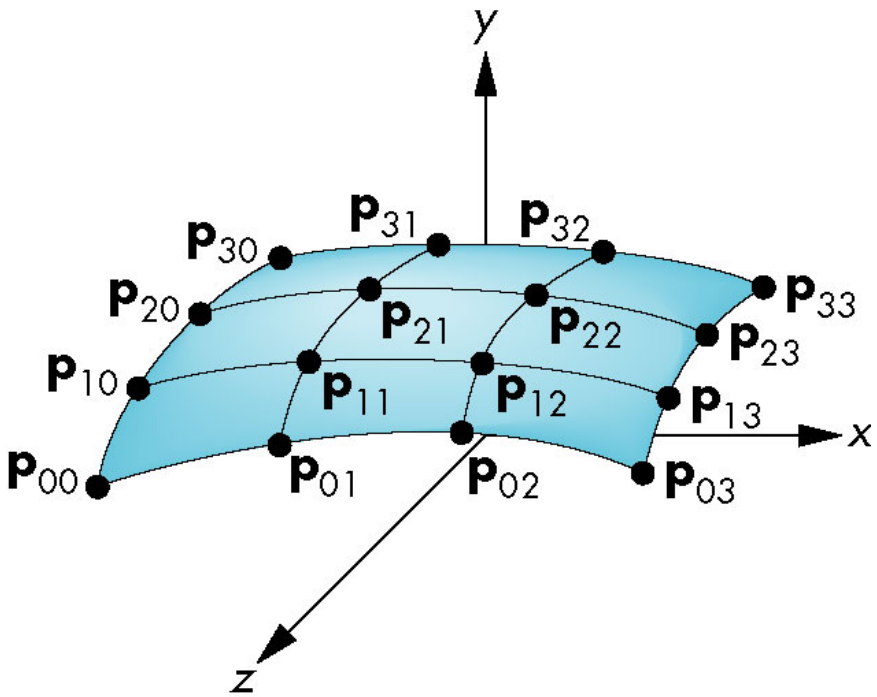
$$\mathbf{b}(u) = \begin{bmatrix} b_0(u) \\ b_1(u) \\ b_2(u) \\ b_3(u) \end{bmatrix}.$$

Polynomials blend together the **individual contributions** of each **control point** and enable us to see the **effect of a given control point on the entire curve**.

The biCubic Interpolating Patch

Need 16 conditions to determine the 16 coefficients $\mathbf{c}_{ij}$

Choose at $\mathbf{u},\mathbf{v} = 0, 1/3, 2/3, 1$



Matrix Form

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

$$\mathbf{v} = \begin{bmatrix} 1 \\ v \\ v^2 \\ v^3 \end{bmatrix}$$

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

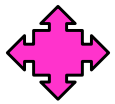
$p(u,v) = u^T C v$

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

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

$C = M_I P M_I^T$

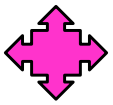
$p(u,v) = u^T M_I P M_I^T v$



Blending Patches

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

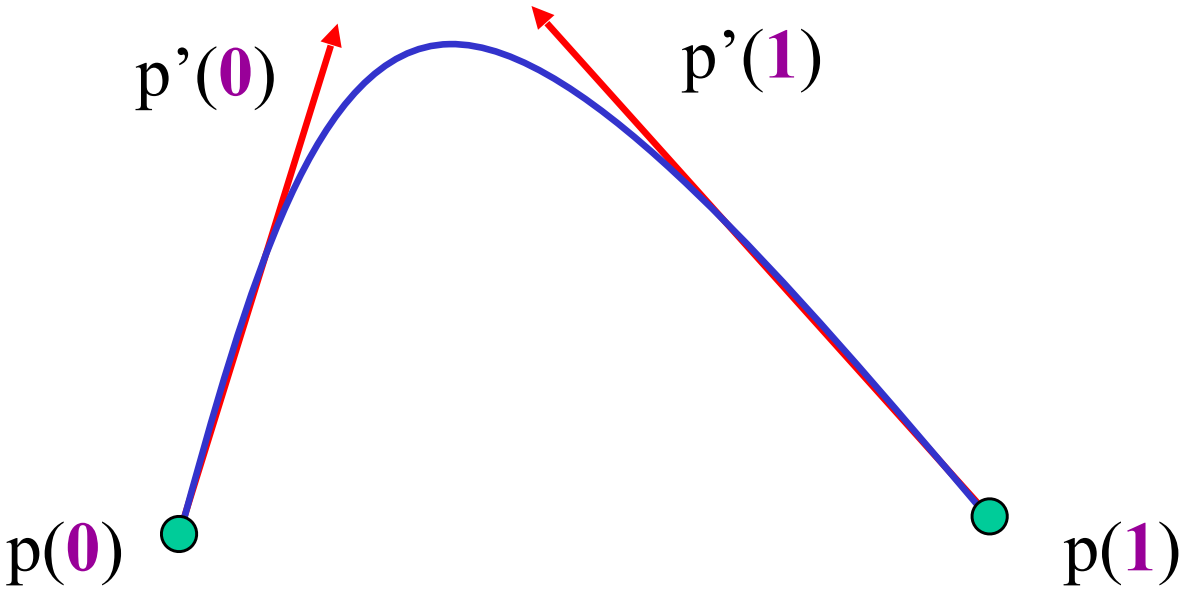
Shows that we can build and analyze **surfaces** from our knowledge of **curves**

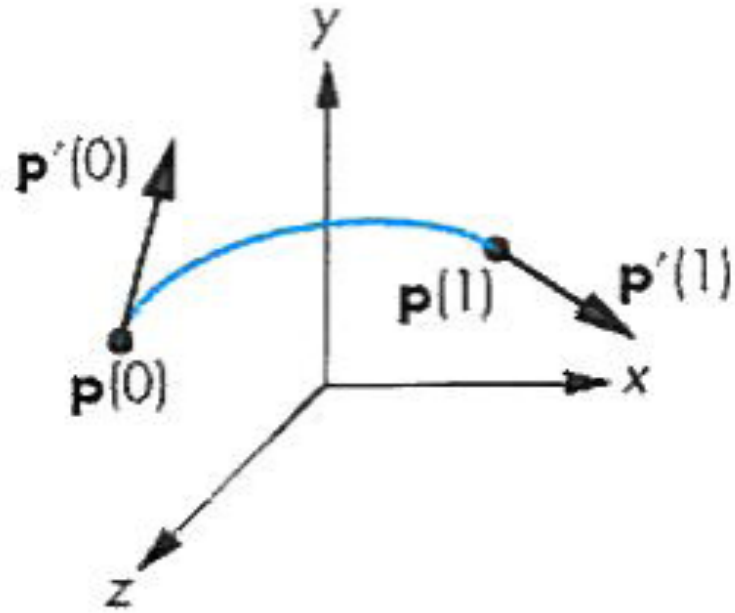


Other Types of Curves and Surfaces

- 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 Curves





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

Ensures **continuity** and first **derivative continuity** between **segments**

Equations

$$p(u) = c_0 + c_1u + c_2u^2 + c_3u^3$$

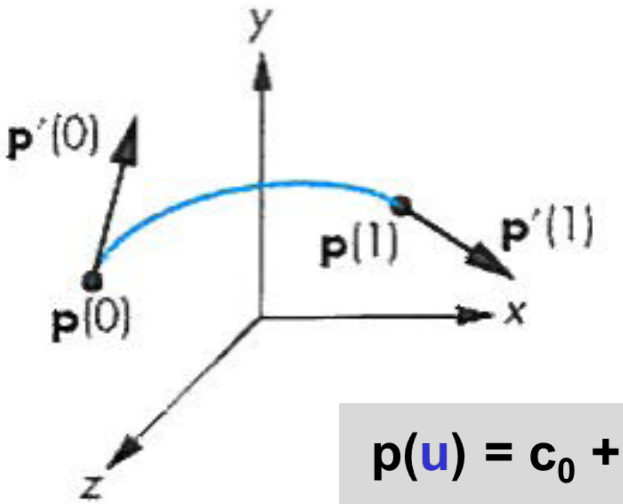
Interpolating conditions are the same at ends

$$p(0) = p_0 = c_0$$
$$p(1) = p_3 = c_0+c_1+c_2+c_3$$

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

Evaluating at end points

$$p'(0) = p'(0) = c_1$$
$$p'(1) = p'(1) = c_1+2c_2+3c_3$$



$$p(u) = c_0 + c_1u + c_2u^2 + c_3u^3$$

$$p_0=p(0) = c_0$$
$$p_1=p(1/3) = c_0+(1/3)c_1+(1/3)^2c_2+(1/3)^3c_3$$
$$p_2=p(2/3) = c_0+(2/3)c_1+(2/3)^2c_2+(2/3)^3c_3$$
$$p_3=p(1) = c_0+c_1+c_2+c_3$$

Matrix Form

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

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

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

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

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

$b_0(u) = -4.5(u-1/3)(u-2/3)(u-1)$
$b_1(u) = 13.5u\,(u-2/3)(u-1)$
$b_2(u) = -13.5u\,(u-1/3)(u-1)$
$b_3(u) = 4.5u\,(u-1/3)(u-2/3)$

Blending Polynomials

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

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

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**

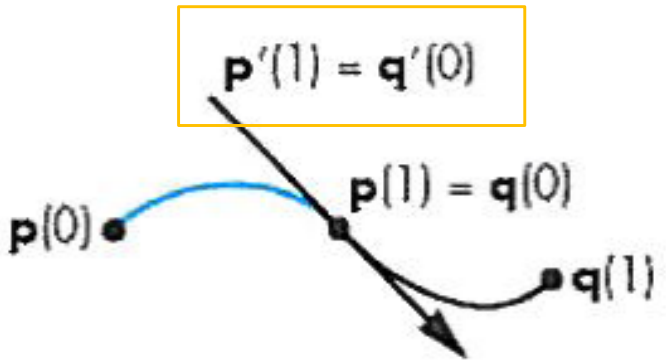
However, the **Hermite form** is the basis of the **Bezier form**

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

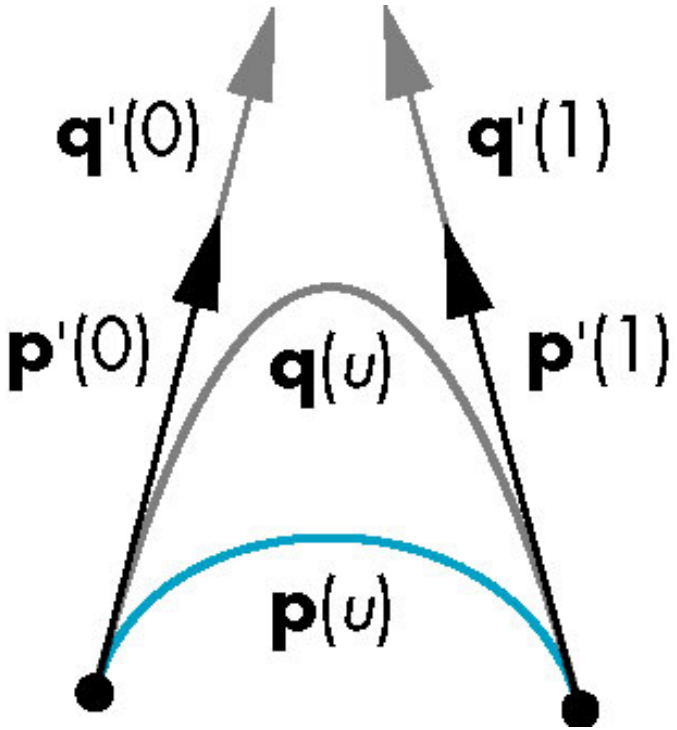
- Here the **p** and **q** have the same **tangents at the ends of the segment** but **different derivatives**

- Generate different

  Hermite curves

- This techniques is used

in **drawing applications**



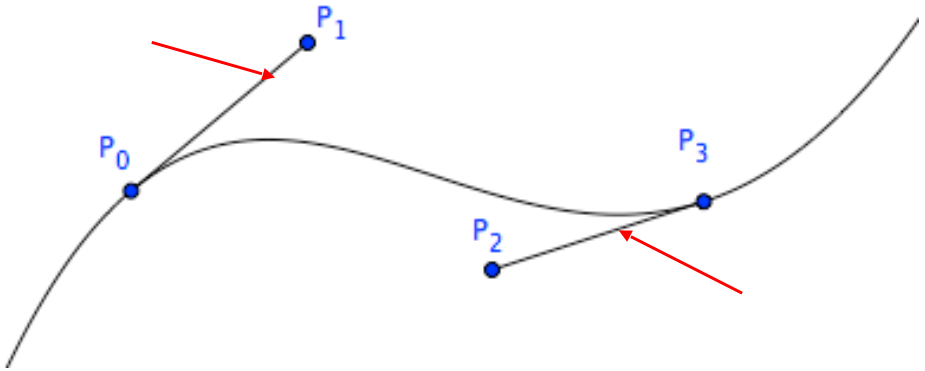
Bezier and Spline Curves and Surfaces

Objectives

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

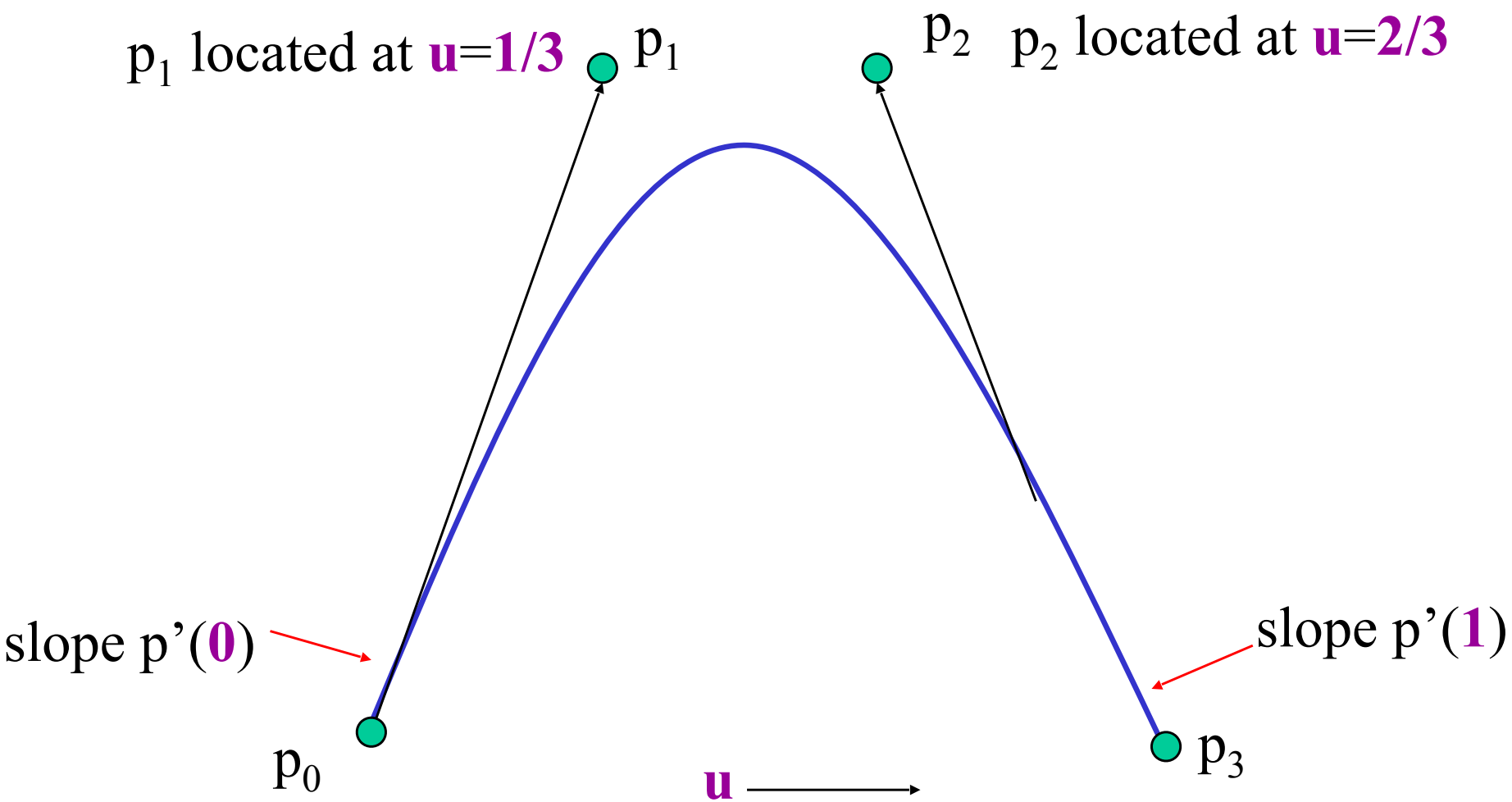
Bezier's Idea

- 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**



http://www.cs.brown.edu/exploratories/freeSoftware/repository/edu/brown/cs/exploratories/applets/bezierSplines/bezier_splines_java_brow

Approximating Derivatives



Equations

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

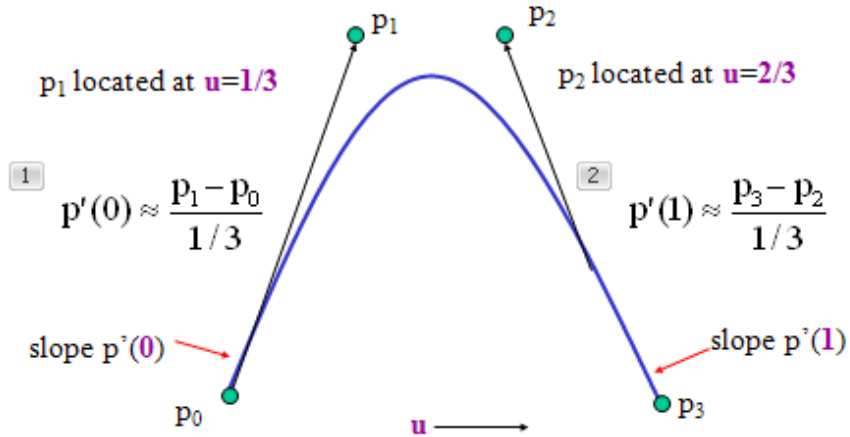
Interpolating **conditions** are the same

$$p(0) = p_0 = c_0$$
$$p(1) = p_3 = c_0 + c_1 + c_2 + c_3$$

Approximating **derivative conditions**

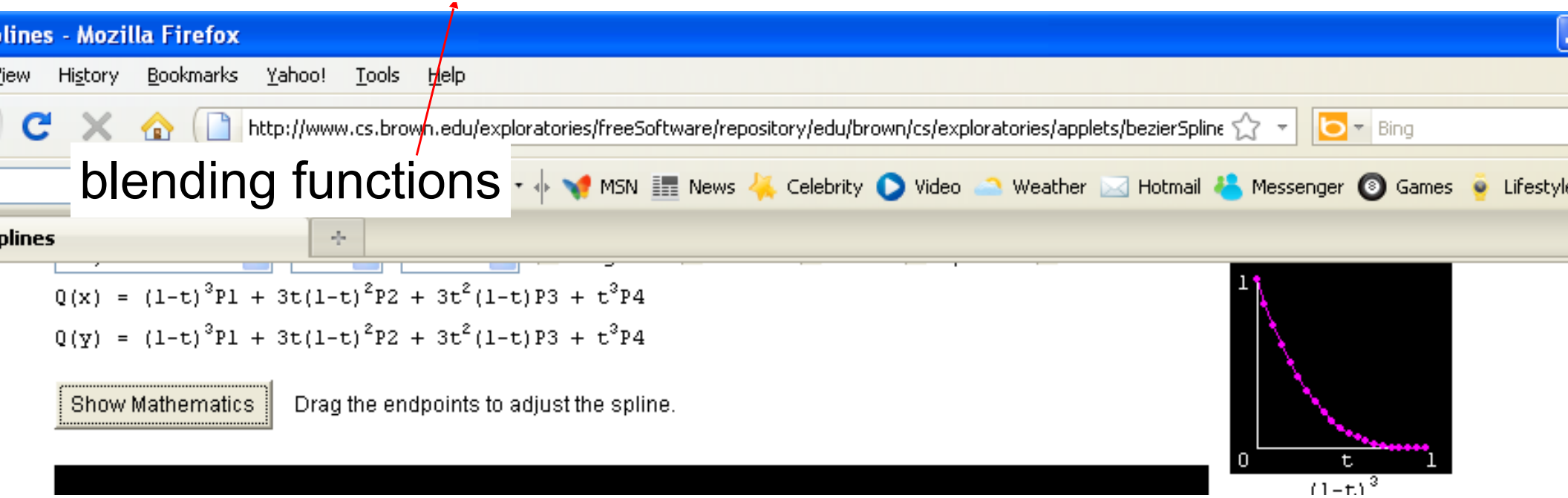
$$p'(0) = 3(p_1 - p_0) = c_0$$
$$p'(1) = 3(p_3 - p_2) = c_1 + 2c_2 + 3c_3$$

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

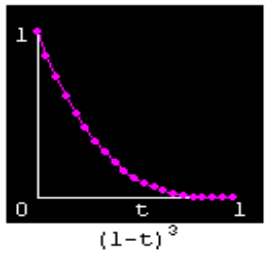


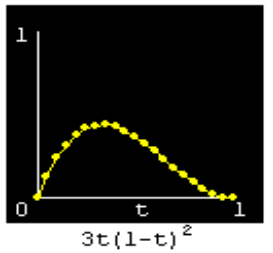
Bezier Matrix

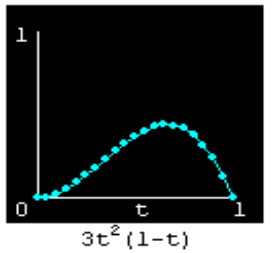
$$p(u) = u^T M_B p = b(u)^T p$$

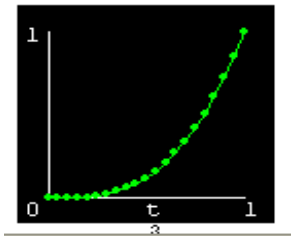


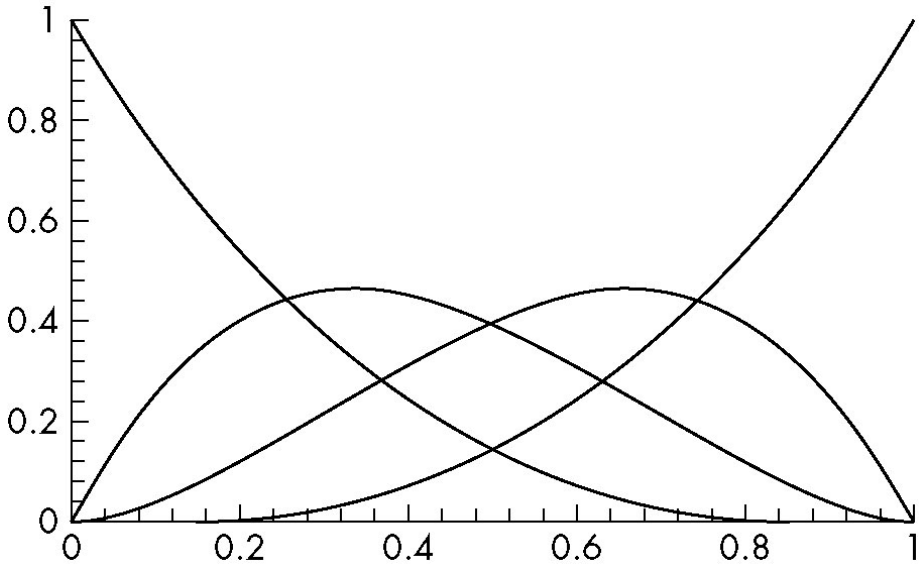
Blending Functions











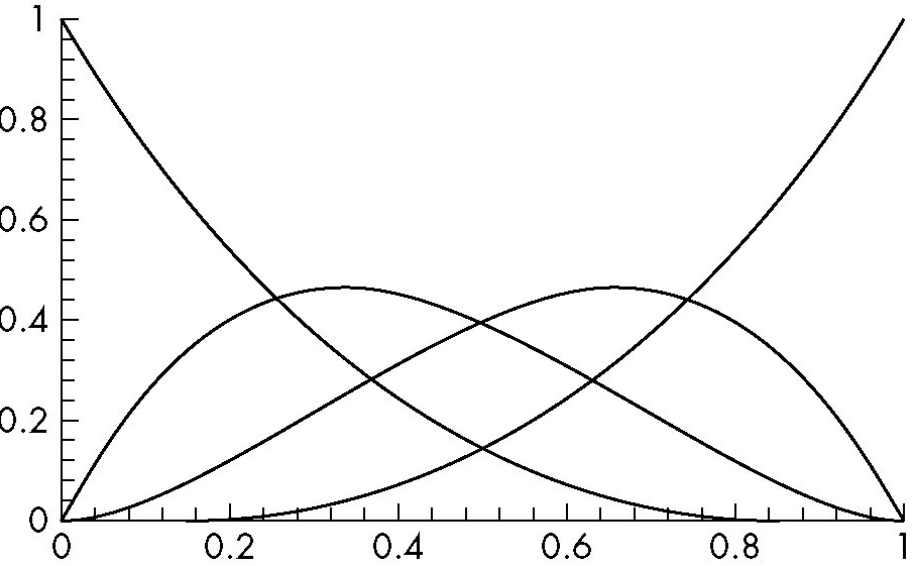
Note that all zeros are at **0** and **1** which forces the **functions to be smooth** over (**0**,**1**)

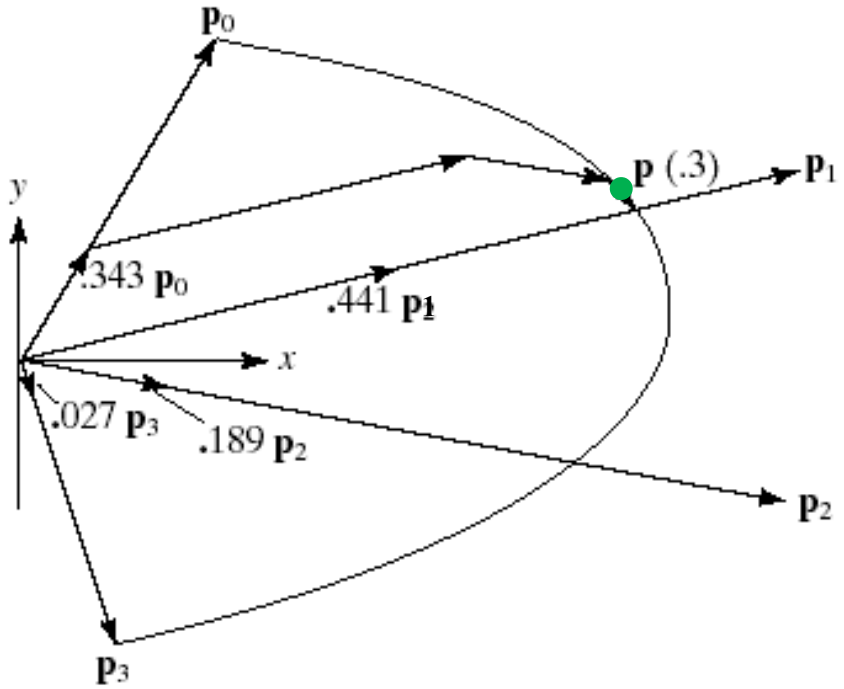
$$P(\mathbf{u}) = P_0(1-\mathbf{u})^3 + P_1 3(1-\mathbf{u})^2\mathbf{u} + P_2 2(1-\mathbf{u})\,\mathbf{u}^2 + P_3\mathbf{u}^3$$

$$Q(x) = (1-t)^3 P1 + 3t(1-t)^2 P2 + 3t^2(1-t)P3 + t^3 P4$$

$$Q(y) = (1-t)^3 P1 + 3t(1-t)^2 P2 + 3t^2(1-t)P3 + t^3 P4$$

Blending Functions





$$P(\mathbf{u}) = P_0(1-\mathbf{u})^3 + P_1 3(1-\mathbf{u})^2\mathbf{u} + P_2 3(1-\mathbf{u})\,\mathbf{u}^2 + P_3\mathbf{u}^3$$

$$P(\mathbf{0.3}) = P_0(1-\mathbf{0.3})^3 + P_1 3(1-\mathbf{0.3})^2\mathbf{0.3} + P_2 3(1-\mathbf{0.3})\,\mathbf{0.3}^2 + P_3\mathbf{0.3}^3$$

$$P(\mathbf{0.3}) = 0.343P_0 + 0.441P_1 + 0.189P_2 + 0.027P_3$$

Bernstein Polynomials

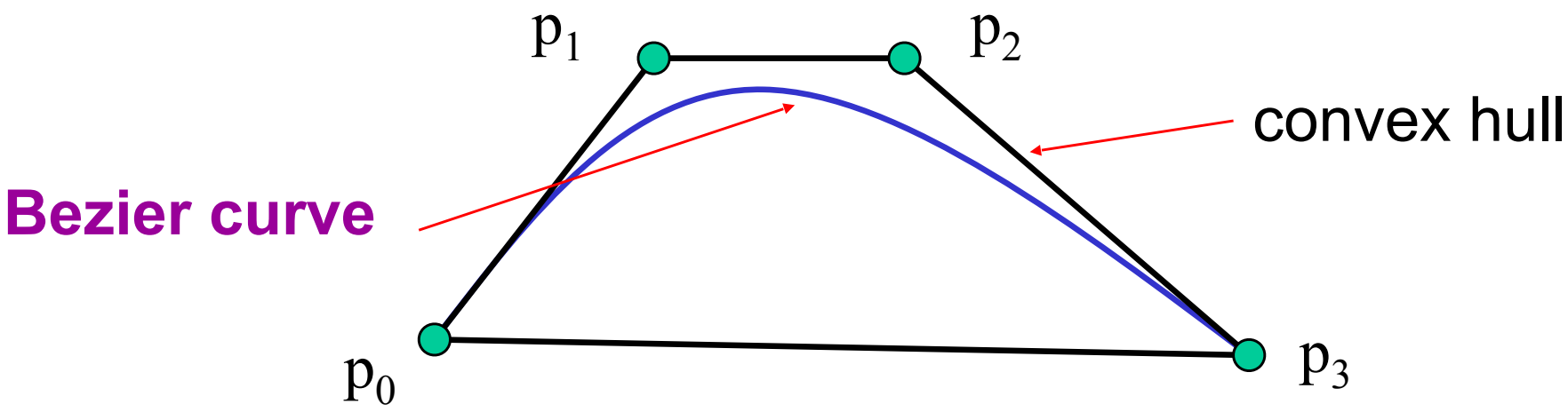
The **blending functions are a special case of the Bernstein polynomials**

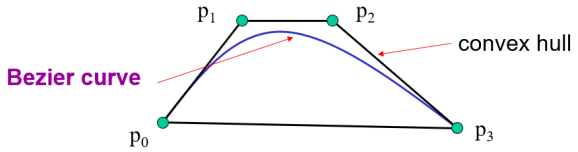
These **polynomials give the blending polynomials for any degree Bezier form**

- 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**



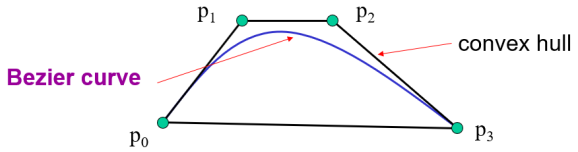


Properties of Bezier Curves

- Bezier curves have properties **well-suited to CAD**.

Endpoint Interpolation: The Bezier curve $P(t)$ based on control points $\boldsymbol{P_0}, P_1, \ldots, \boldsymbol{P_L}$ always interpolates $\boldsymbol{P_0}$ and $\boldsymbol{P_L}$.

Affine Invariance: to apply an **affine transformation T to all** Points ***P*(*t*) on the Bezier curve**, **we transform the control points (once),** and **use the new control points to re-create the transformed Bezier curve** Q(t) at any *t.*

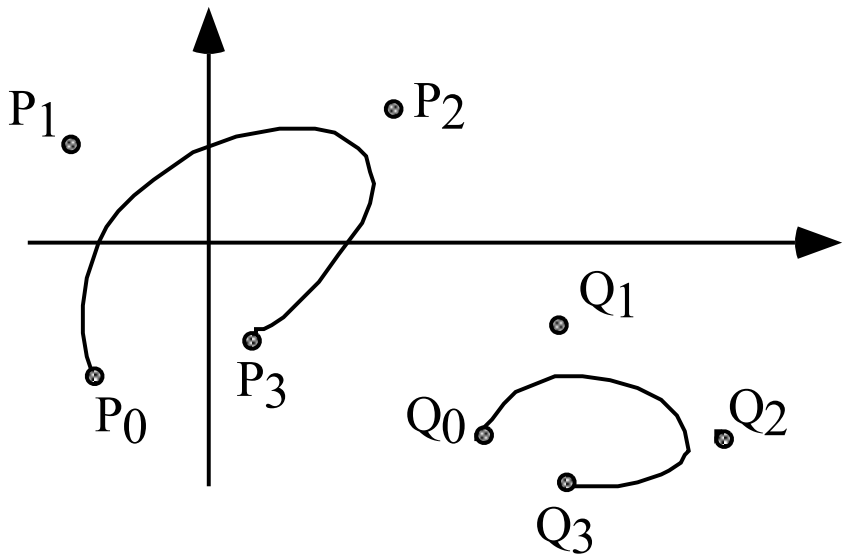


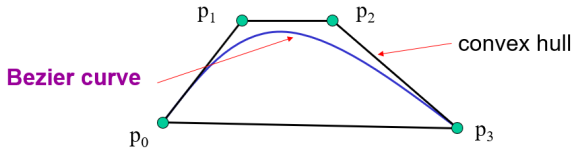
Properties of Bezier Curves

Example: A **Bezier curve** based on four control points P_0,..., P_3.

The points are rotated, scaled, and translated to the new control points Q_k.

The **Bezier curve** for Q_k is drawn. It is identical to the result of transforming the original **Bezier curve**.



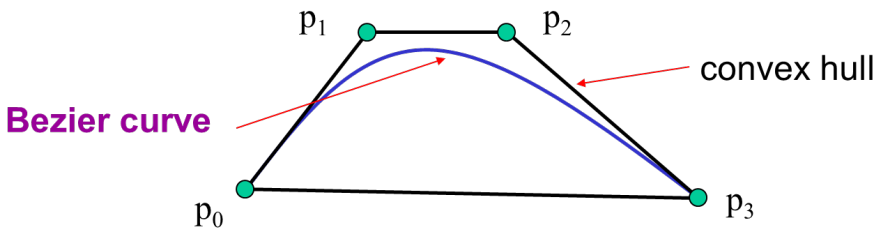


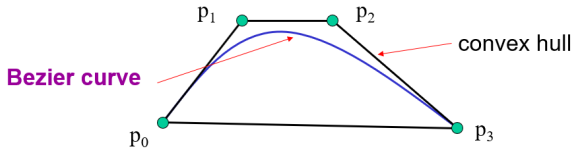
Properties of Bezier Curves

Convex Hull Property: a **Bezier curve**, $P(t)$, never wanders outside **its convex hull**.

The **convex hull** of a set of points P_0, P_1, . . ., PL is the set of all *convex combinations* of the Points; that is, the set of all Points given by

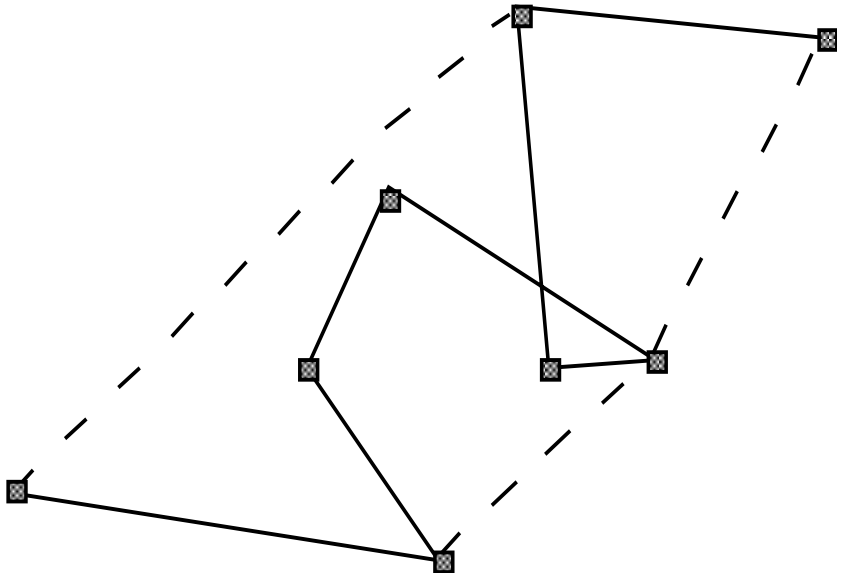
where each α_k is non-negative, and **they sum to 1**.

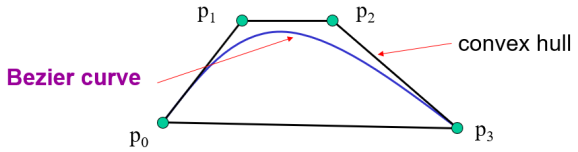




Properties of Bezier Curves

- Example: Even though the **eight control points** form a jagged control polygon, the designer knows the Bezier curve will flow smoothly between the two endpoints, **never extending outside the convex hull**.





Properties of Bezier Curves

- Derivatives of **Bezier Curves**: the first derivative is

where $\Delta P_k = P_{k+1} - P_k$

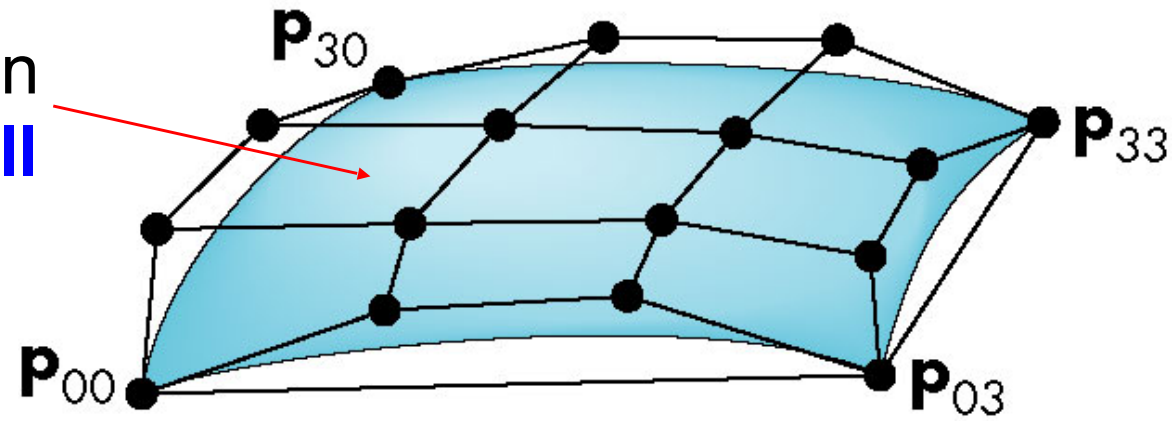
- The **velocity** is another **Bezier curve**, built on a new set of **control vectors** ΔP_k .

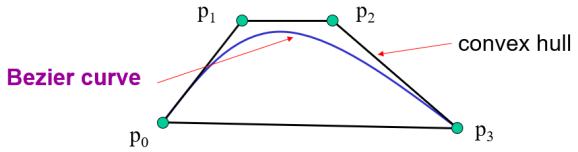
- Taking the **derivative** lowers the order of the **curve** by 1: the **derivative** of a **cubic Bezier curve** is a **quadratic Bezier curve**.

Bezier Patches

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

Patch lies in **convex hull**





Analysis

- Although the **Bezier form** is much better than the **interpolating form**, **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**

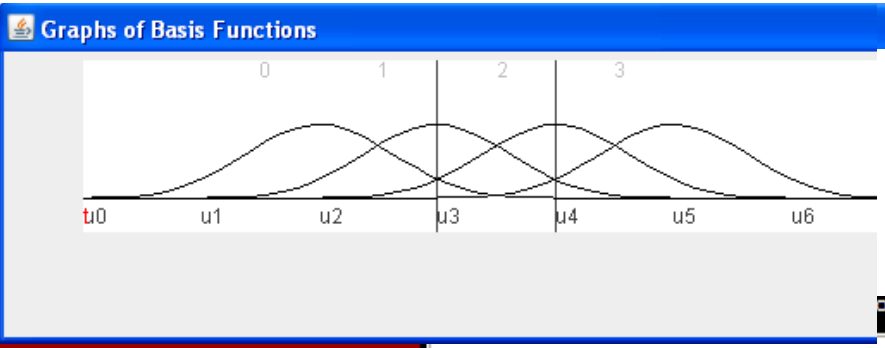
  Apply different conditions

  - Tricky without letting order increase

B-Splines

- Basis splines: use the data at $\mathbf{p}=[p_{i-2}\ \mathbf{p_{i-1}}\ \mathbf{p_i}\ p_{i-1}]^T$ to define **curve** only between $\mathbf{p_{i-1}}$ and $\mathbf{p_i}$
- 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



B-Splines

Applets | Materials | Course Notes | Publications

Cubic B-Spline Applet

DIRECTIONS: In this applet, you create a cubic B-Spline by specifying control points p_i with $i=0,1,2,\ldots, n$ and a knot-sequence u_j with $j=0,1,2,\ldots,m$. The number of knots is related to the number of control points by $m=n+4$. The curve $c(t)$ is defined on the interval $u_3 < t < u_{m-3}$, and given by

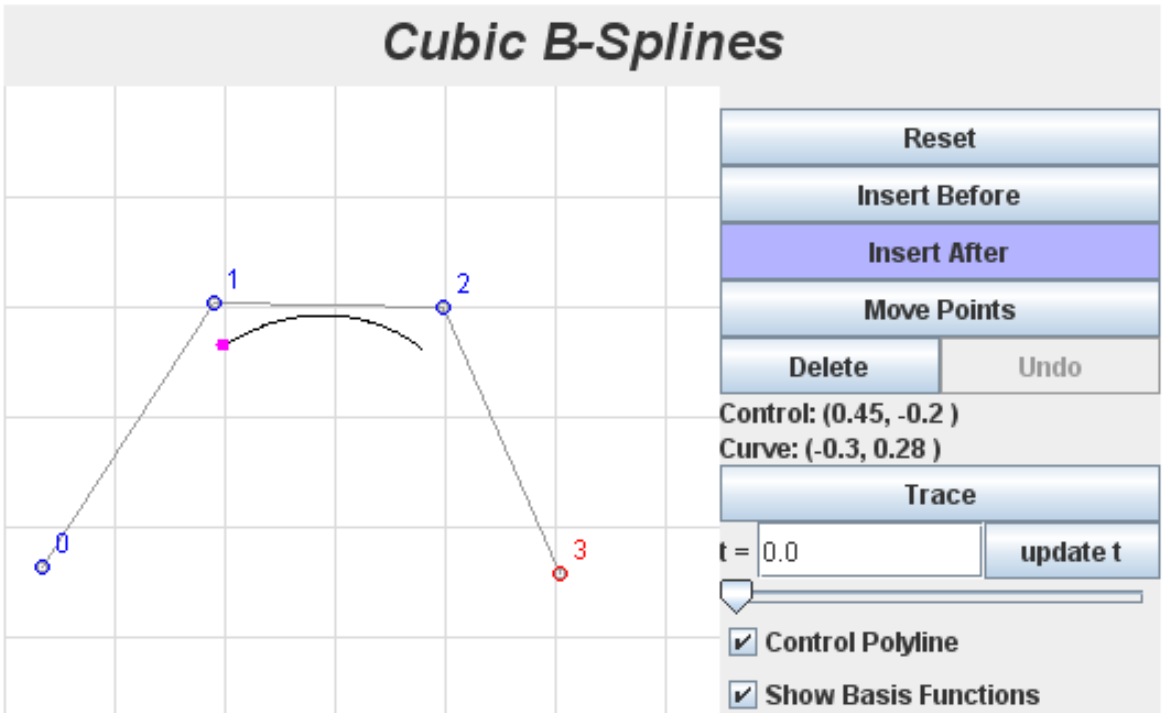
$$c(t) = \Sigma N_i^3 p_i$$

With the basis functions defined iteratively by the relation

$$N_i^{k+1}(t) = ((t-u_i)/(u_{i+k} - u_i)) N_i^k(t) + ((u_{i+k+1}-t)/(u_{i+k+1}-u_{i+1})) N_{i+1}^k(t)$$

with $N_i^0 = 1$ when $u_i < t < u_{i+1}$ and 0 otherwise.

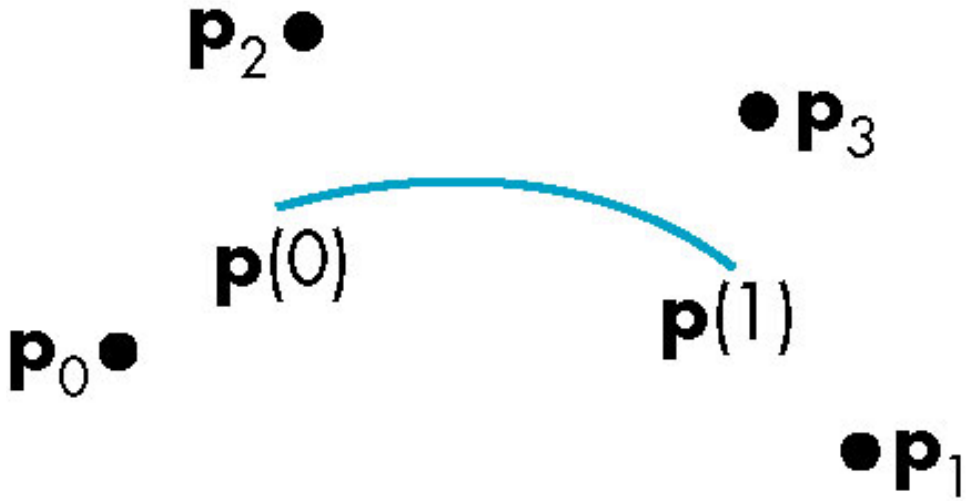
B-Splines are a generalization of Bezier curves and have many of the same properties but the knot sequence allows one to shape the curve and interpolation properties of the curve at specific points. The applet original chooses after every point to reset t the knots to be equally spaced apart on the interval $0 < t < 1$.



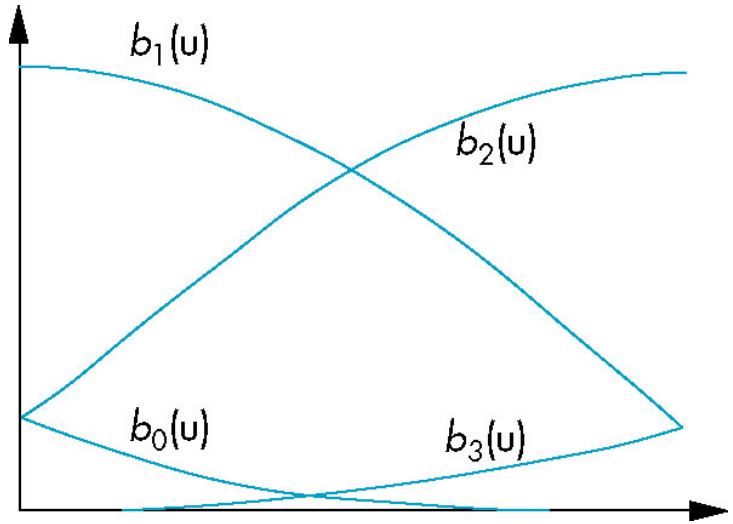
Cubic B-spline

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

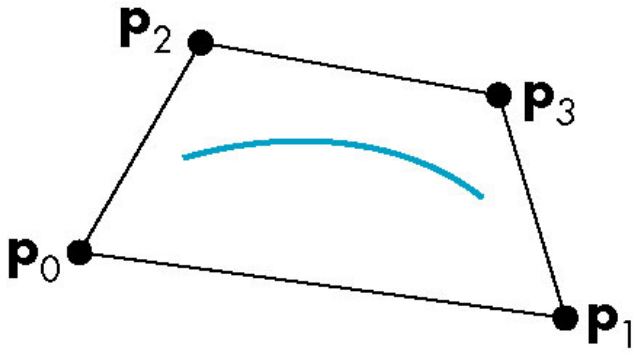
B-spline Matrix



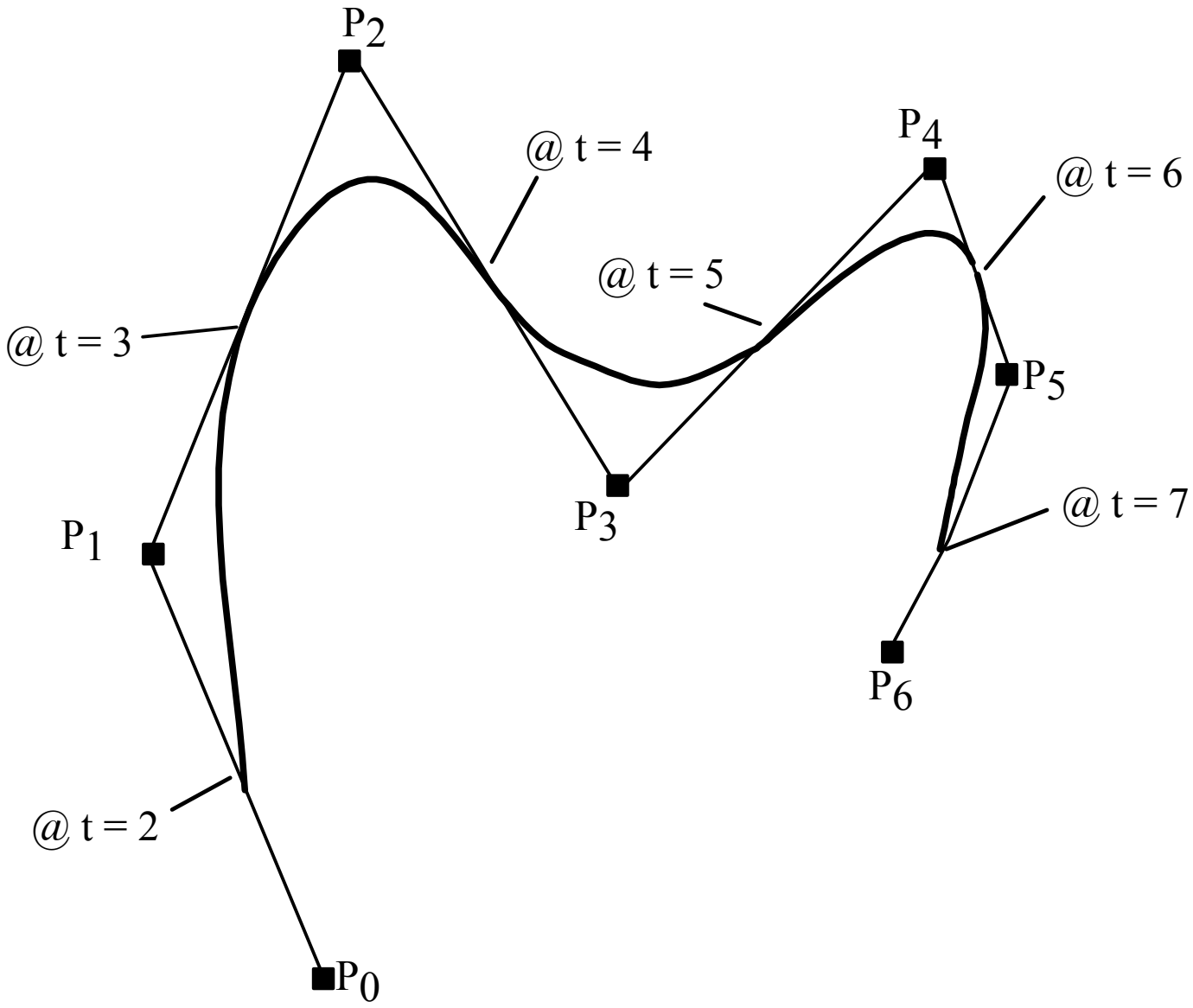
Blending Functions



convex hull property

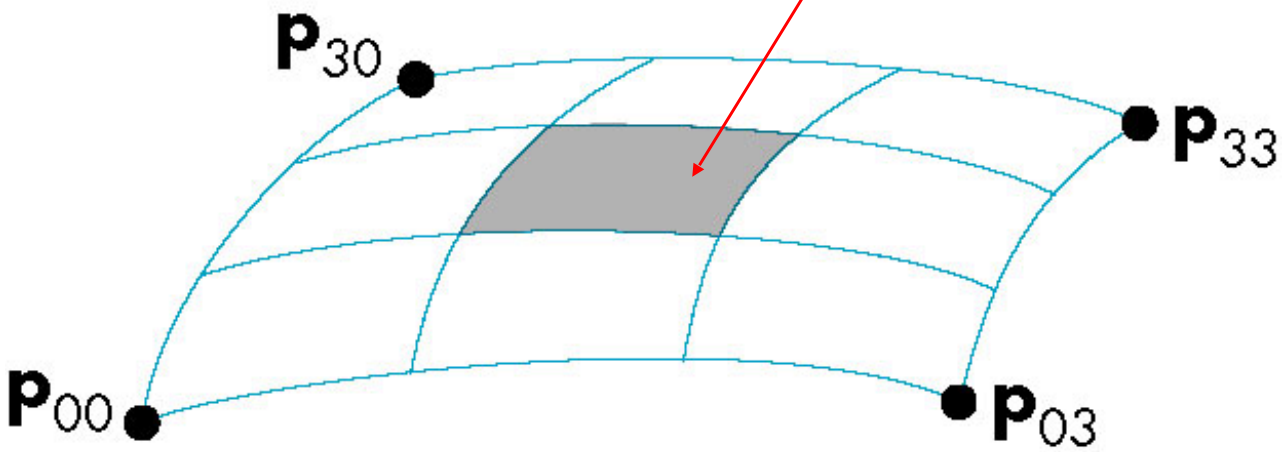


Making Blending Functions from Splines: Example



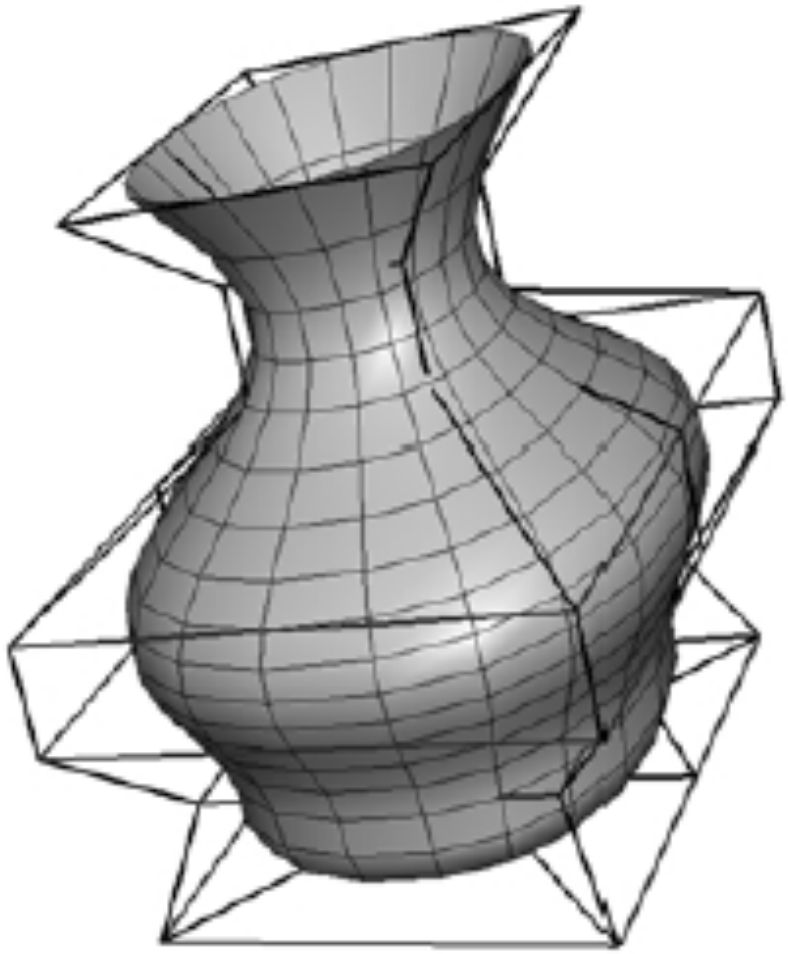
B-Spline Patches

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

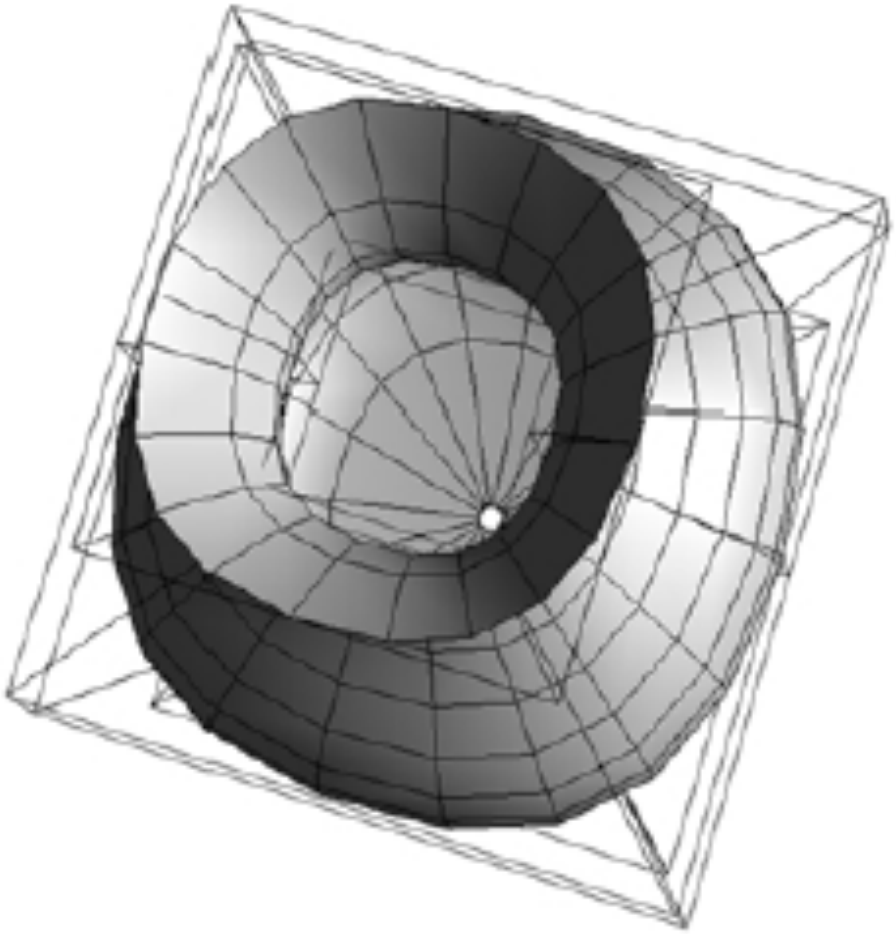


B- Spline Patches

a)



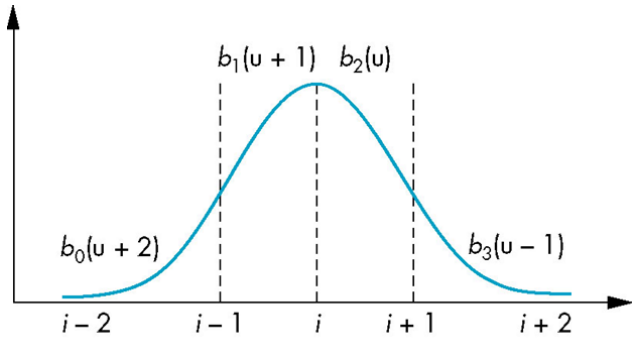
b)



Splines and Basis

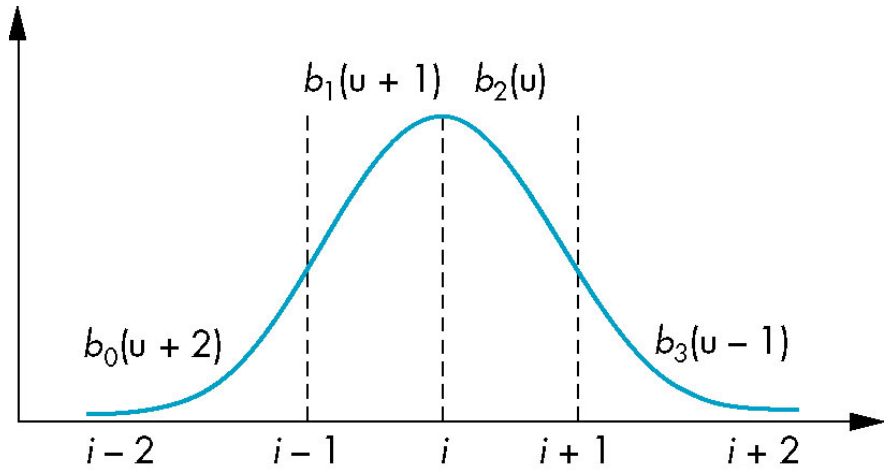
- If we examine the **cubic B-spline** from the perspective of each **control (data) point**, each interior Point contributes (through the blending functions) to four segments
- We can rewrite p(**u**) in terms of the **data points** as

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



Basis Functions

In terms of the blending polynomials



NURBS

Nonuniform Rational B-Spline

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

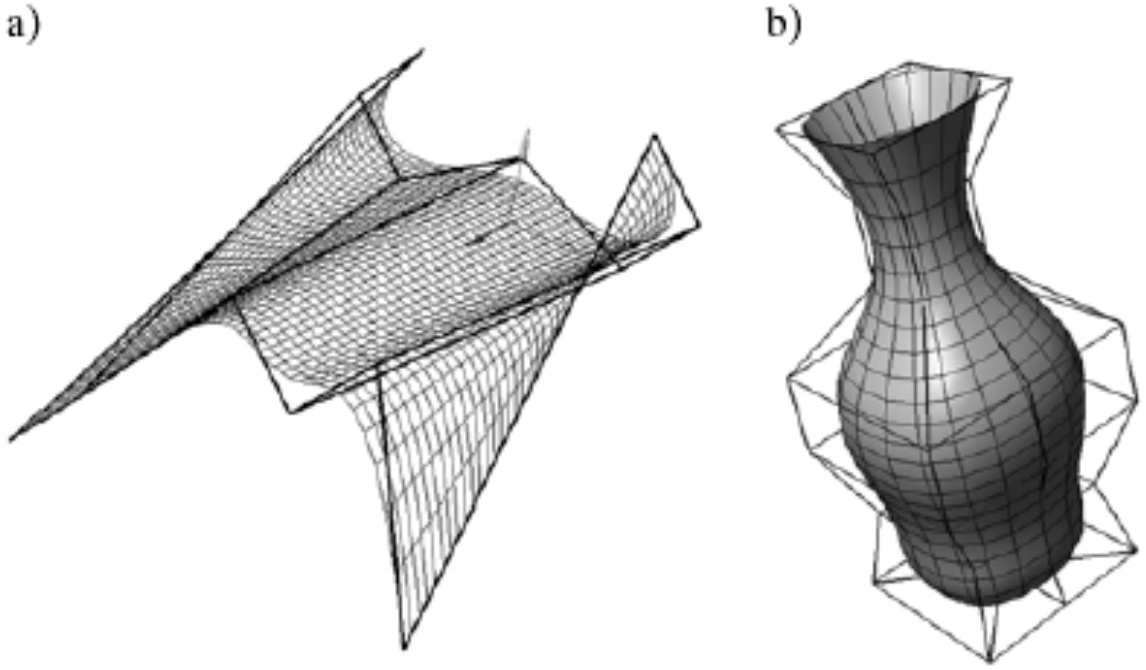
- **NURBS act correctly for perspective viewing**

Quadrics are a special case of **NURBS**

Example

A) Extruded Surfaces - The prism is a NURBS curve in *u*; the straight sides are a first order NURBS in *v*.

Ruled Surfaces - The two edge curves are NURBS curves in *u*; the rulings are first order NURBS in *v*.



ng) - Microsoft Visual Studio

ect Build Debug Tools Visual Assert Test Window Help

Debug Win32 texenv

Hex

Thread: Stack Frame:

curves.c

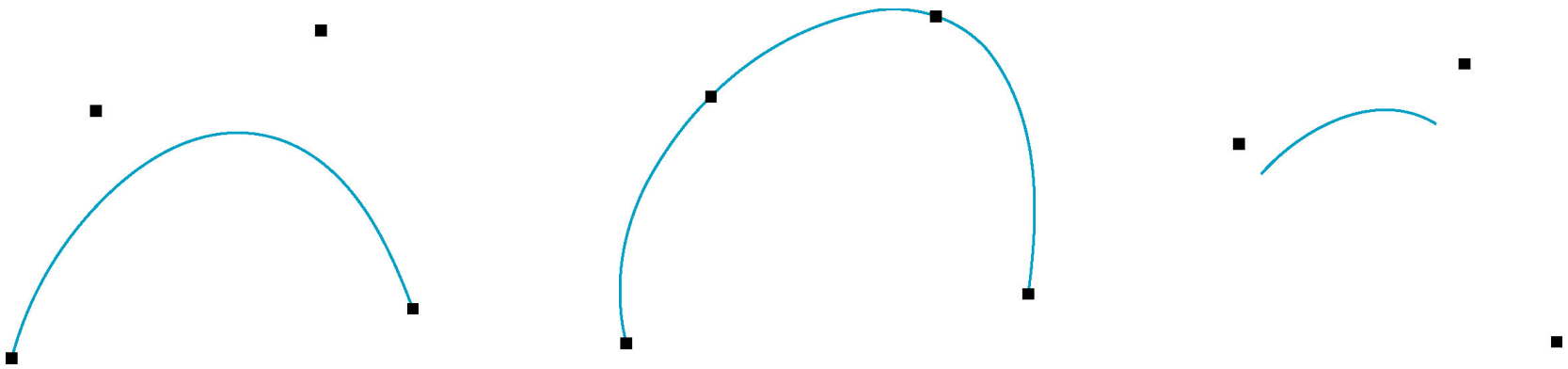
(Global Scope)

```
 95
 96  /* Calculate the matrix used to transform the control points */
 97  void computeMatrix(curveType type, float m[4][4])
 98  {
 99      int i, j;
100
101      switch (type)
102      {
103          case BEZIER:
104              /* Identity matrix */
105              for (i = 0; i < 4; i++)
106                  for (j = 0; j < 4; j++)
107                      m[i][j] = midentity[i][j];
108              break;
109          case INTERPOLATED:
110              for (i = 0; i < 4; i++)
111                  for (j = 0; j < 4; j++)
112                      m[i][j] = minterp[i][j];
113              break;
114          case BSPLINE:
115              for (i = 0; i < 4; i++)
116                  for (j = 0; j < 4; j++)
117                      m[i][j] = mbspline[i][j];
118              break;
119       }
120  }
```

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

Example

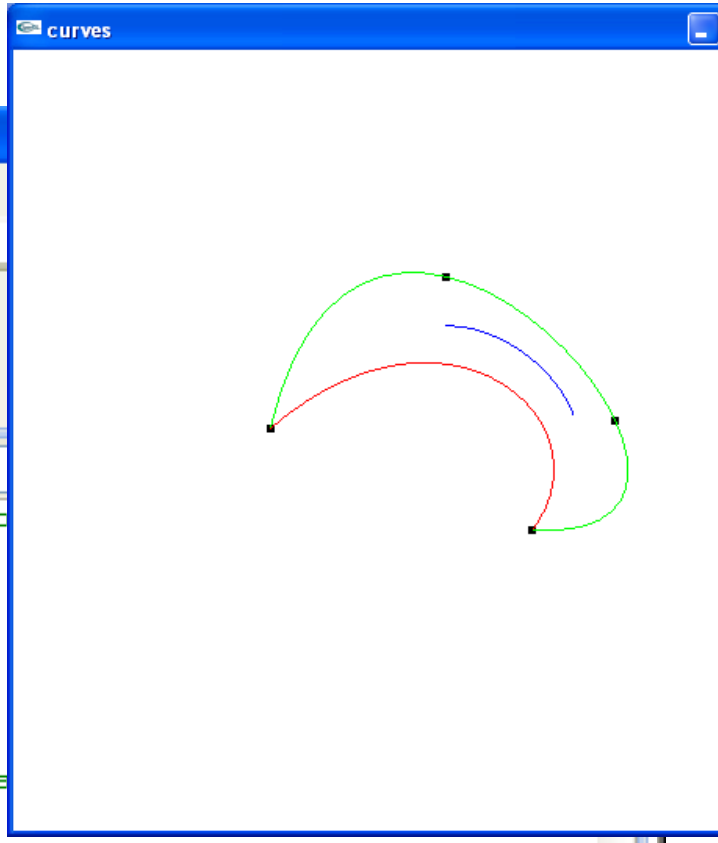
These **three curves** were all generated from the same original data using **Bezier recursion** by converting all control point data to **Bezier control points**



Bezier

Interpolating

B Spline



Running) - Microsoft Visual Studio

Project Build Debug Tools Visual Assert Test Window Help

Debug Win32 texenv

Hex

Thread: Stack Frame:

Solutio...

ure 12' (1 project)
12
er Files
urce Files
e Files
urves.c

Class View

curves.c*

(Global Scope)

```
122  /* Draw the indicated curves using the current control po
123  static void drawCurves(curveType type)
124  {
125      int i;
126      int step;
127      GLfloat newcpts[4][3];
128      float m[4][4];
129      /* Set the control point computation matrix and the s
130      computeMatrix(type, m);
131      if(type == BSPLINE) step = 1;
132      else step = 3;
133      glColor3fv(colors[type]);
134      /* Draw the curves */
135      i = 0;
136      while (i + 3 < ncpts)
137      {
138          /* Calculate the appropriate control points */
139          vmult(m, &cpts[i], newcpts);
140          /* Draw the curve using OpenGL evaluators */
141          glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &newcpts[0][0]);
142          glMapGrid1f(30, 0.0, 1.0);
143          glEvalMesh1(GL_LINE, 0, 30);
144          /* Advance to the next segment */
145          i += step;
146      }
147      glFlush();
```

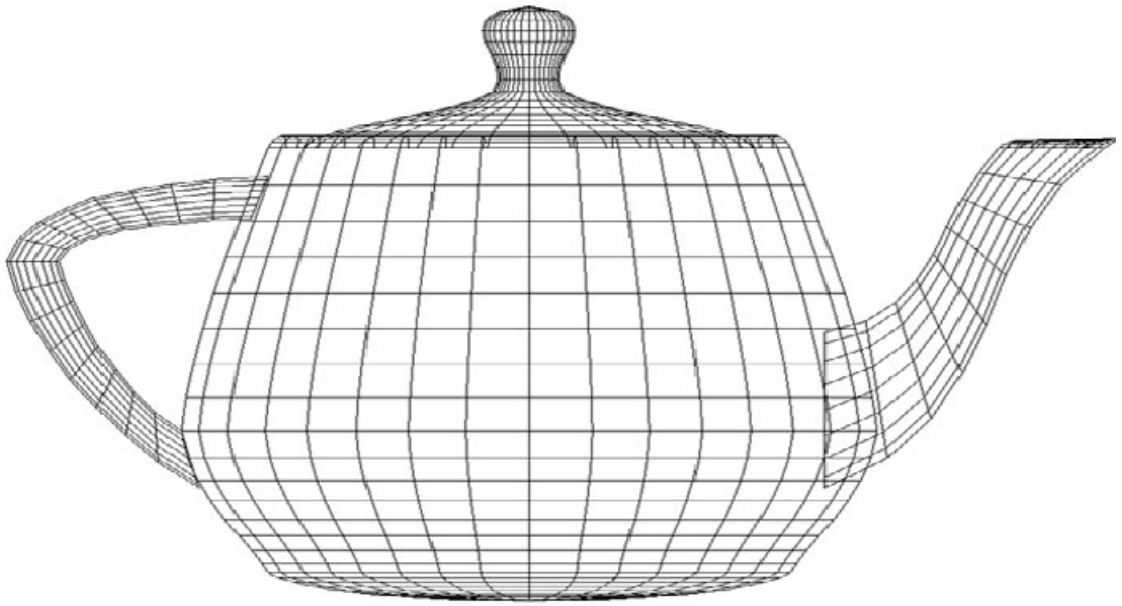
RUN IT!

curves.c

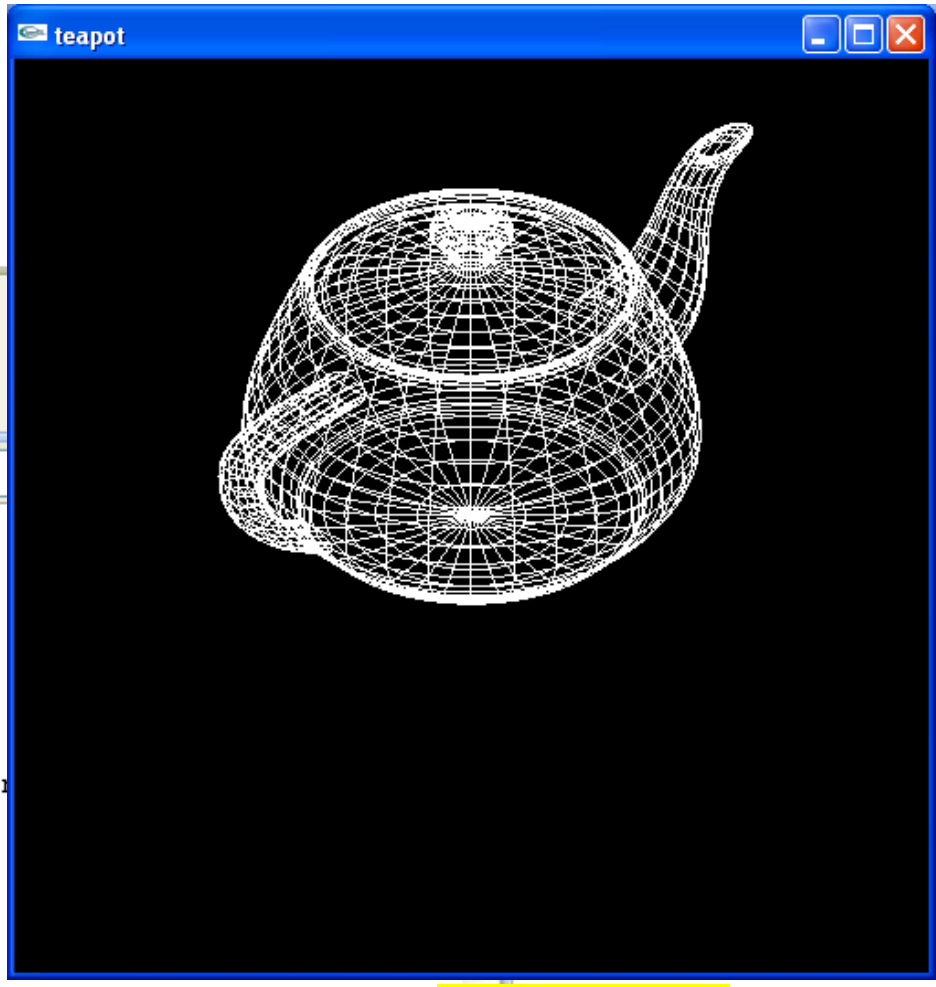
Utah Teapot

- Most famous data set in **computer graphics**

- Widely available as a list of 306 3D vertices and the indices that define 32 Bezier patches



Utah Teapot




```
}

main(int argc, char *argv[])
{
    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;
    {
       /* 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();

}
```

RUN IT!

bteapot.c

NEXT.

11.15.2023 (W 5:30 to 7) (25)		**Lecture 13** **(Advanced Rendering)**
11.20.2023 (M 5:30 to 7) (26)		**PROJECT 4**
11.27.2023 (M 5:30 to 7) (27)		**EXAM 4 REVIEW**
11.29.2023 (M 5:30 to 7) (28)		**EXAM 4**
12.11.2023 (M 5:30 to 7)		**FINAL EXAM**

At 6:45 PM.

End Class 24

VH, Download Attendance Report
Rename it:
11.13.2023 Attendance Report FINAL

VH, upload Lecture 12 to CANVAS.