CS 4204 Computer Graphics

Curves and Surfaces (Continue)

Yong Cao

Virginia Tech

Reference: Ed Angle, Interactive Computer Graphics, University of New Mexico, class notes

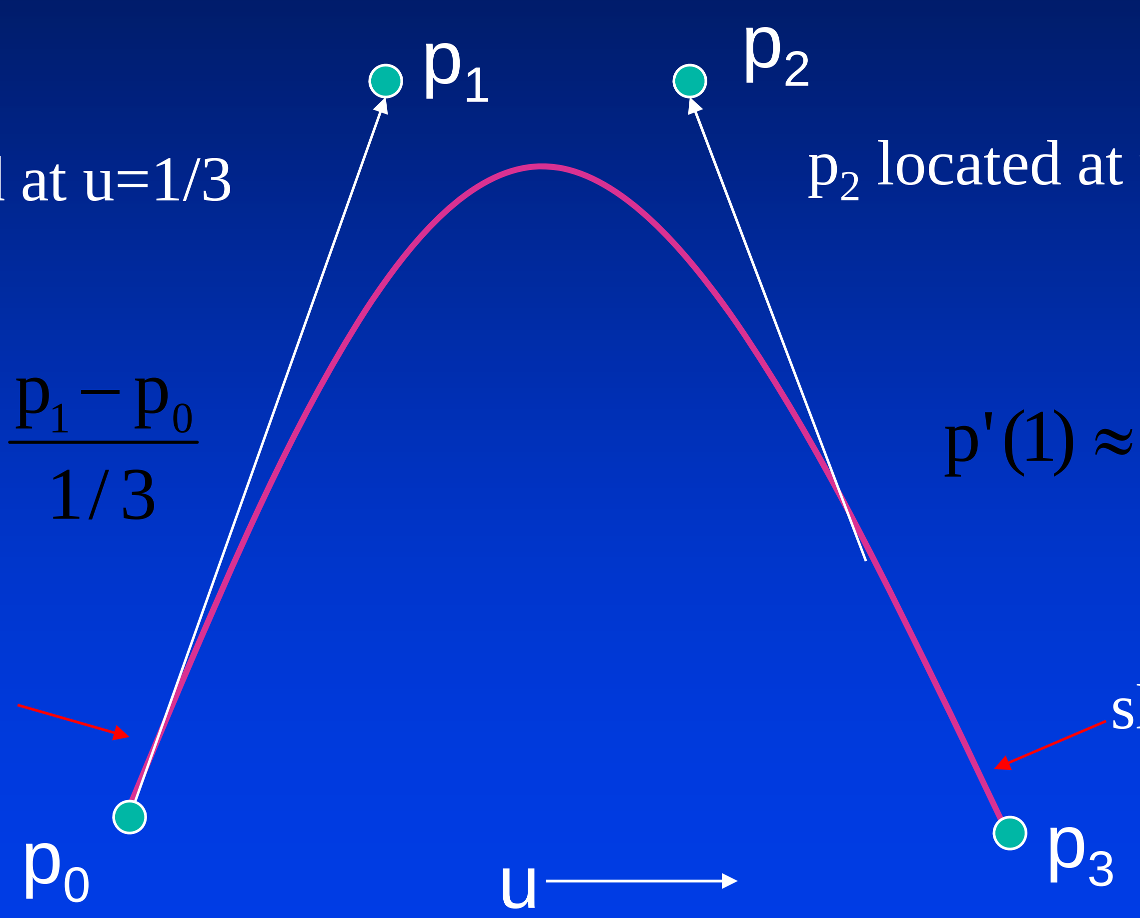
Objectives

- *Bezier curves and surfaces*
- *B-spline and compare it to the standard cubic Bezier*
- *Introduce OpenGL evaluators*
- *Learn to render polynomial curves and surfaces*

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



$$p'(0) \approx \frac{p_1 - p_0}{1/3}$$

$$p'(1) \approx \frac{p_3 - p_2}{1/3}$$

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

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

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

Bezier Matrix

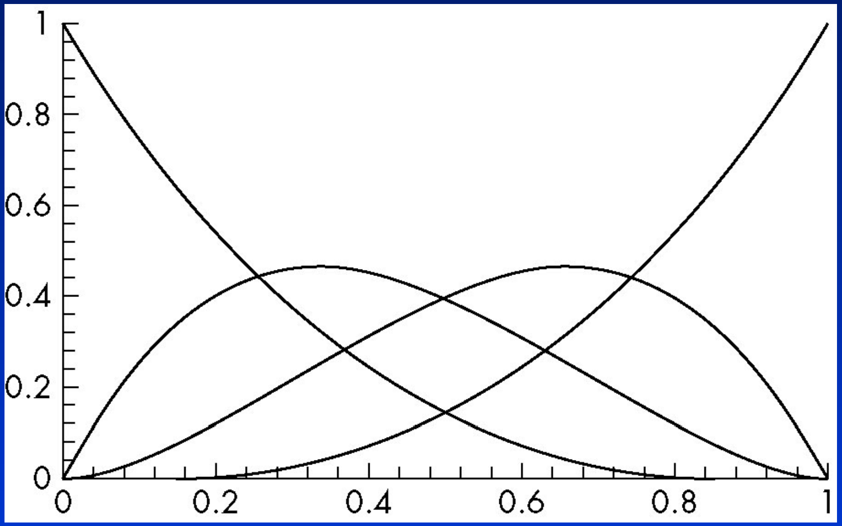
$$\mathbf{M}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

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

blending functions

Blending Functions

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



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***

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

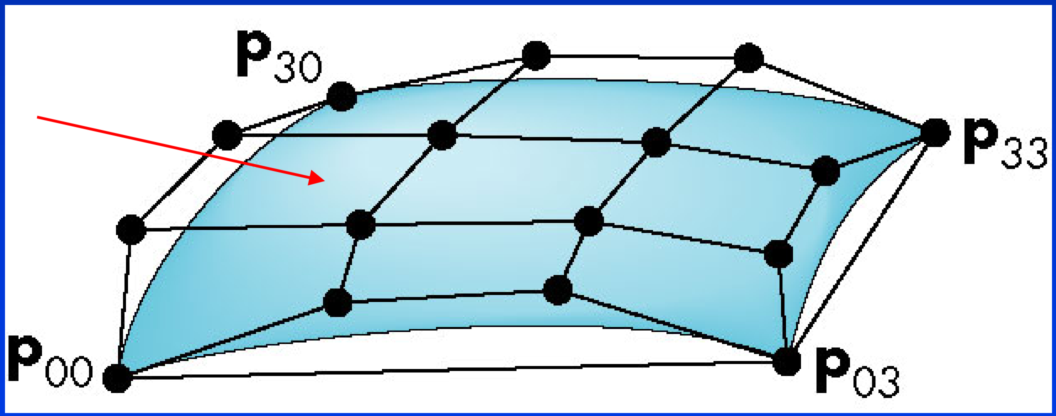
- ***These polynomials give the blending polynomials for any degree Bezier form***
  - All zeros at 0 and 1
  - For any degree they all sum to 1
  - They are all between 0 and 1 inside (0,1)

Bezier Patches

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

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

Patch lies in convex hull



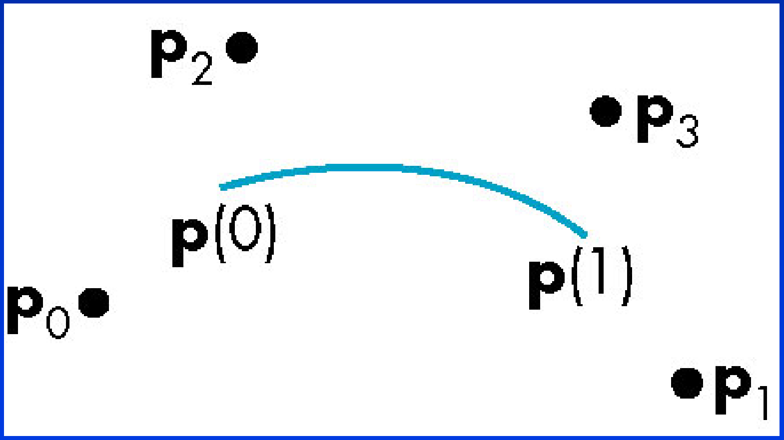
B-Splines

- ***B**asis splines: use the data at* $p=[p_{i-2}\ p_{i-1}\ p_i\ p_{i-1}]^T$ *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

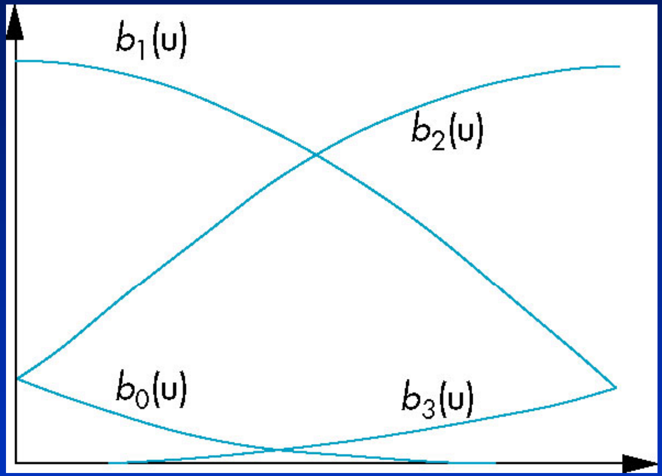
$$p(u) = \mathbf{u}^T\mathbf{M}_S\mathbf{p} = \mathbf{b(u)}^T\mathbf{p}$$

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

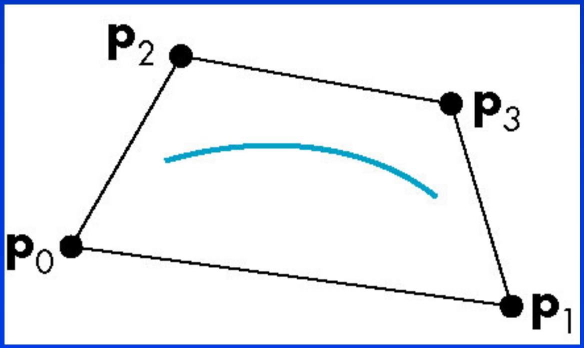


Blending Functions

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



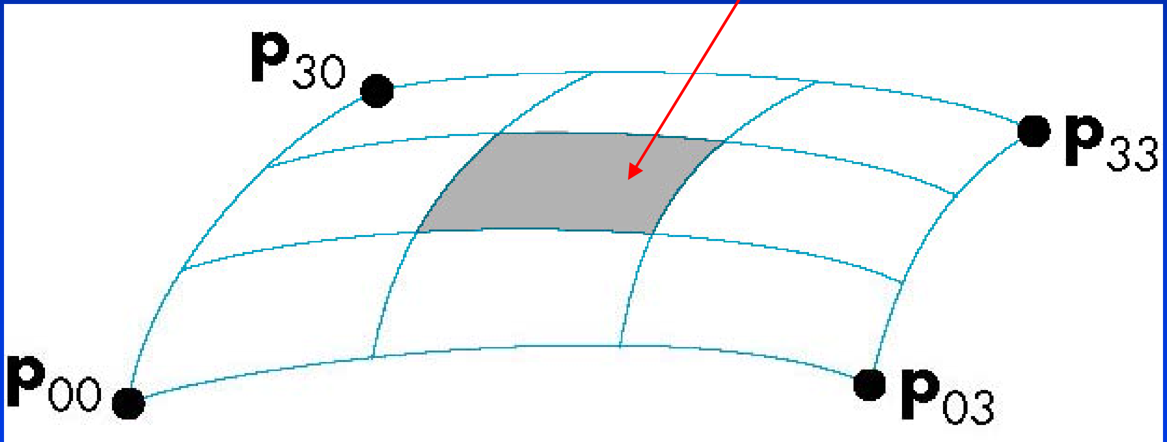
convex hull property



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*

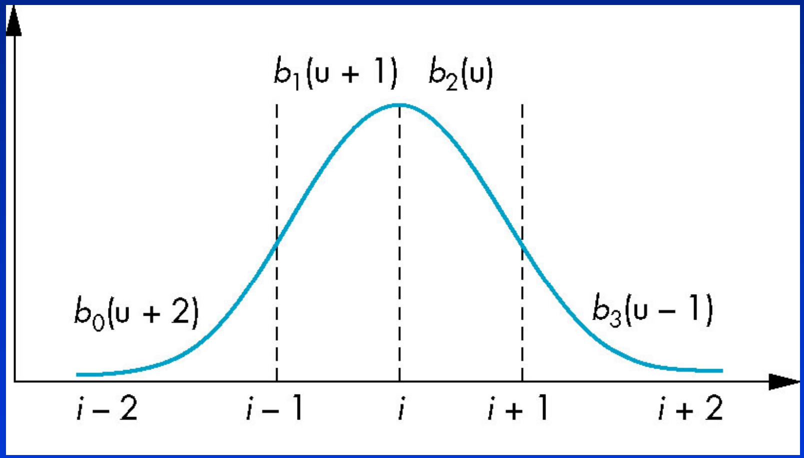
$$p(u) = \sum B_i(u)\, p_i$$

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

Basis Functions

In terms of the blending polynomials

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



NURBS

- ***N*onuniform *R*ational *B*-*S*pline 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***

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)
```

separation between data points

1+degree of polynomial

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_MAP1_VERTEX_3,0.0,1.0,3,4,data);
```

data are 3D vertices

cubic

data are arranged as x,y,z,x,y,z……

three floats between data points in array

```
glEnable(GL_MAP1_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 u and 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

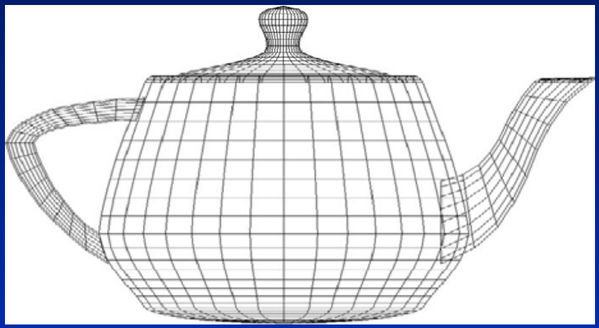
bicubic over (0,1) x (0,1)

```
point data[4][4]={………};
glMap2f(GL_MAP1_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++) {
      glEvalCoord2f ((float) i/100.0,
          (float) j/100.0);
      glEvalCoord2f ((float)(i+1)/100.0,
          (float)j/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)
```

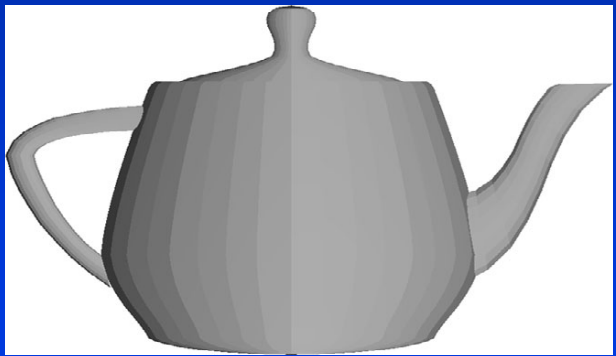
- *Can evaluate as before with lines or if want filled polygons*

```
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);
```



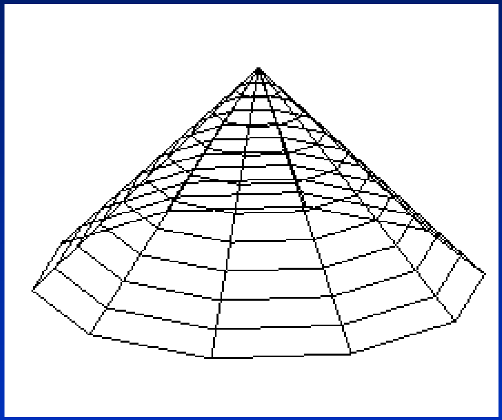
NURBS

- ***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

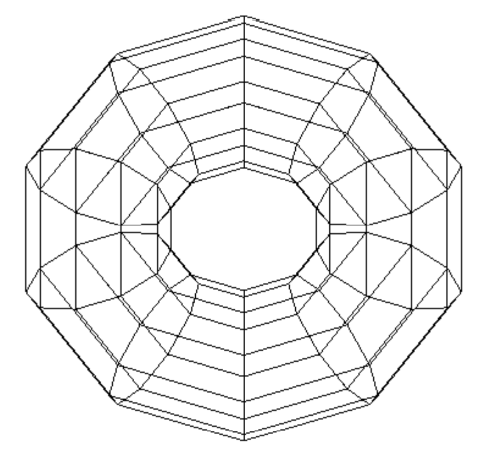
Quadrics

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

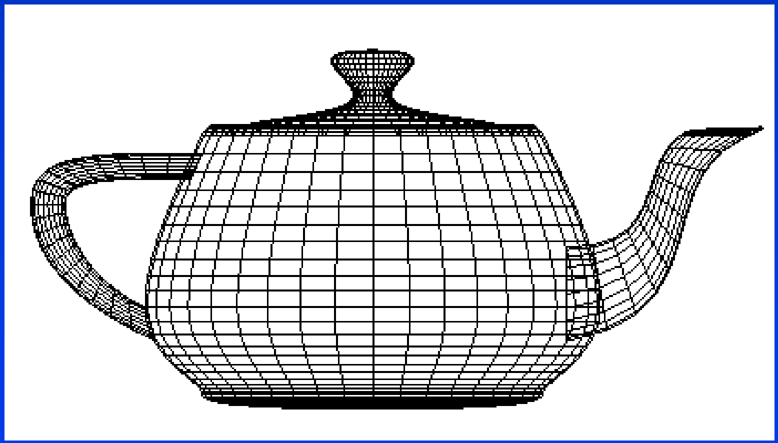
GLUT Objects



`glutWireCone()`

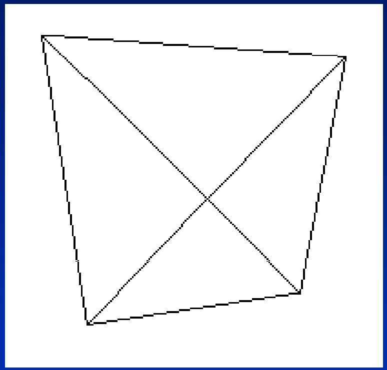


`glutWireTorus()`

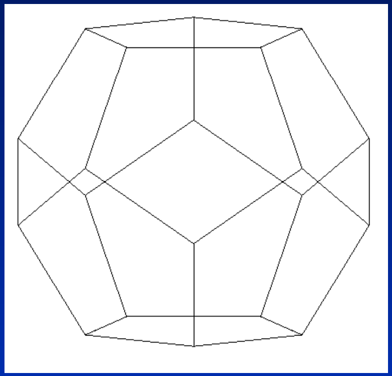


`glutWireTeapot()`

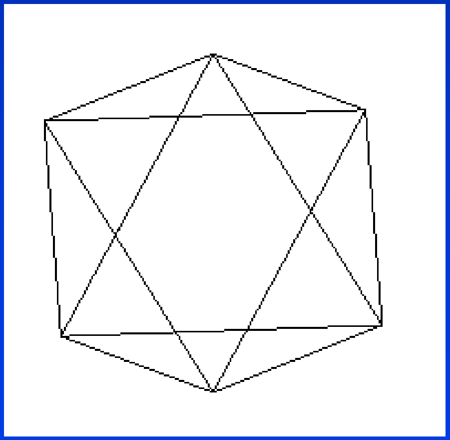
GLUT Platonic Solids



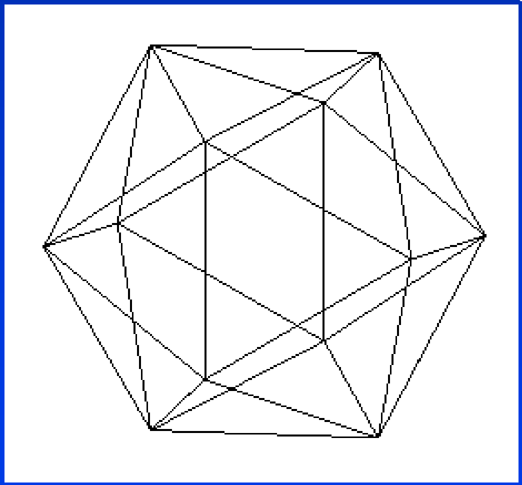
glutWireTetrahedron()



glutWireDodecahedron()



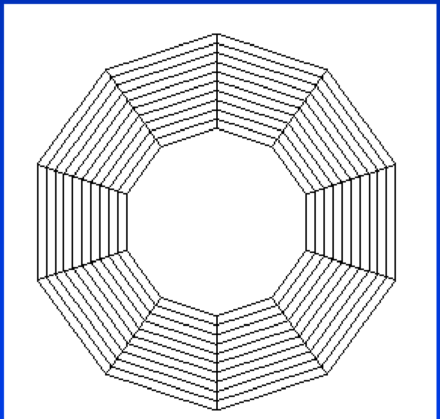
glutWireOctahedron()



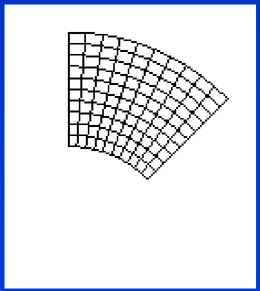
glutWireIcosahedron()

Quadric Objects in GLU

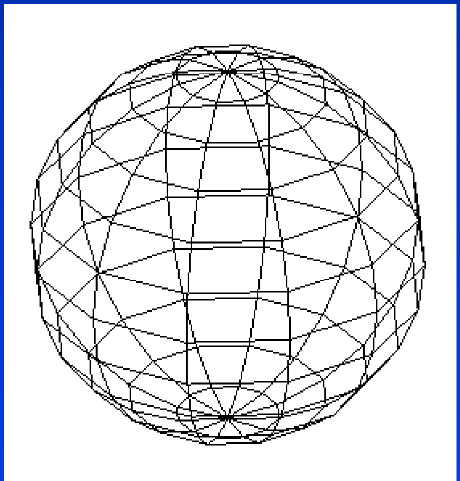
- ***GLU can automatically generate normals and texture coordinates***
- ***Quadrics are objects that include properties such as how we would like the object to be rendered***



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);
```