Johns Hopkins
Engineering for Professionals

605.767 Applied Computer Graphics

Brian Russin

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Module 6H
OpenGL Evaluators
NURBS Support

OpenGL Evaluators

- Evaluators are used to describe splines and surfaces in OpenGL
  - Supports B-splines, NURBS surfaces, Bezier curves and surfaces, and Hermite splines
- Makes splines and surfaces based on Bezier (or Berstein) basis functions
  - To draw in other bases you must convert to a Bezier basis
- Allows specification of the granularity of the subdivision used when rendering the curves or surfaces
  - Highly subdivided -> high quality, longer rendering time
  - Lower subdivision -> lower quality, faster rendering

Defining an Evaluator for a Curve

```
void glMap1{fd}(GLenum target, TYPE u1, TYPE u2, GLint stride,
                     GLint order, const TYPE* controlPoints);
```

Defines a one-dimensional evaluator. Target specifies what the control points represent (see below). u1 and u2 represent the range for the variable u. stride is the number of values in each block of storage (an offset from the beginning of one control point to the beginning of the next). Order is the degree plus one of the basis function. controlPoints is an array of control points.

Types of Control Points for glMap1

```
GL_MAP1_VERTEX_3: x,y,z vertex coordinates
GL_MAP1_VERTEX_4: x,y,z,w vertex coordinates
GL_MAP1_INDEX: Color index
GL_MAP1_COLOR_4: R,G,B,A color
GL_MAP1_NORMAL: Normal coordinate
GL_MAP1_TEXTURE_COORD1: s texture coordinates
GL_MAP1_TEXTURE_COORD2: s.t texture coordinates
GL_MAP1_TEXTURE_COORD3: s,t,r texture coordinates
GL_MAP1_TEXTURE_COORD4: s,t,r,q texture coordinates
```

Using the Evaluator to Produce a Curve

```
void glEvalCoord1{fd}(TYPE u);
void glEvalCoord1{fd}v(TYPE* u);
```

Causes an evaluation of the enabled one-dimensional maps. u is the value (or pointer to the value) of the domain coordinate.

To use evenly spaced values for the evaluator: define a one-dimensional grid using glMapGrid1 and then apply it using glEvalMesh1.

```
void glMapGrid{fd}(GLint n, TYPE u1, TYPE u2);
```

Defines a grid that goes from u1 to u2 in , evenly spaced steps.

```
void glEvalMesh1{fd}(GLenum mode, GLint p1, GLint p2);
```

Applies the currently defined map grid to all enabled evaluators. Mode can be either GL_POINT (draws points) or GL_LINE (draws connected lines). Same as calling glEvalCoord1 for each step (including) p1 and p2.

Defining an Evaluator for a Surface

2D evaluators produce surface meshes. Similar to one-dimensional evaluators except commands take two parameters: u and v. Procedure follows:

1. Define the evaluator(s) with **glMap2***()
2. Enable them by passing the appropriate value to **glEnable**()
3. Invoke them by either calling **glEvalCoord2**() between a glBegin() and glEnd() pair or by specifying an evenly spaced mesh with **glMapGrid2()** and then applying it with **glEvalMesh2**().

```
void glMap2{fd}(GLenum target, TYPE u1, TYPE u2, GLint uStride,
                GLint uOrder, TYPE v1, TYPE v2, GLint vStride,
                GLint vOrder, TYPE* controlPoints);
```

Defines a two-dimensional evaluator. Target specifies what the control points represent (same as for glMap1). Minimum and maximum values for u and v parameters are specified with u1,u2 and v1, v2. uStride and vStride are the number of values between independent settings for these values in the controlPoints array. uOrder and vOrder are the degree plus one of the basis function. controlPoints is an array of control points.

Using the Evaluator to Produce a Surface

```
void glEvalCoord2{fd}(TYPE u, TYPE v);
void glEvalCoord2{fd}v(TYPE* u, TYPE* v);
```

Causes an evaluation of the enabled two-dimensional maps. u is the value (or pointer to the value) of the domain coordinate. The normal to the surface is computed analytically if either of the vertex evaluators is enabled. If `GL_AUTO_NORMAL` is enabled (using `glEnable`), then this normal is associated with the generated vertex. If disabled, the corresponding enabled normal map is used.

To use evenly spaced values for the evaluator: define a two-dimensional grid using glMapGrid2 and then apply it using glEvalMesh2.

```
void glMapGrid2{fd}(GLint nu, TYPE u1, TYPE u2, GLint nv, TYPE v1, TYPE v2);
void glEvalMesh1{fd}(GLenum mode, GLint i1, GLint i2, Glint j1, GLint j2);
```

Defines a grid that goes from u1 to u2 in nu , evenly spaced steps and a grid from v1 to v2 in nv steps. Applies this grid to all enabled evaluators in `glEvalMesh2`. Mode can be either `GL_POINT` (draws points), `GL_LINE` (draws connected lines), or GL_FILL (generates filled polygons using the `QUAD_MESH` primitive.

Bezier Mesh with OpenGL

- Red book example

Control Points:

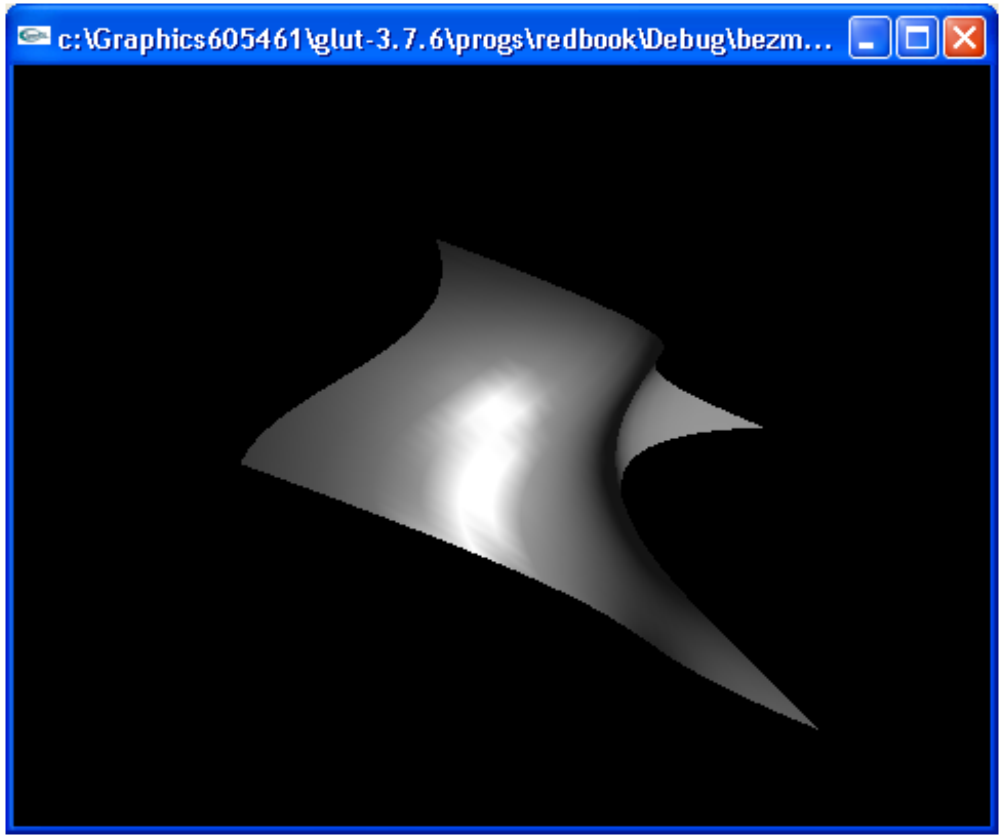
```
GLfloat ctrlpoints[4][4][3] =
      {{-1.5, -1.5,  4.0}, {-0.5, -1.5,  2.0},
       { 0.5, -1.5, -1.0}, { 1.5, -1.5,  2.0}},
      {{-1.5, -0.5,  1.0}, {-0.5, -0.5,  3.0},
       { 0.5, -0.5,  0.0},{  1.5, -0.5, -1.0}},
      {{-1.5,  0.5,  4.0}, {-0.5,  0.5,  0.0},
       { 0.5,  0.5,  3.0}, { 1.5,  0.5,  4.0}},
      {{-1.5,  1.5, -2.0}, {-0.5,  1.5, -2.0},
       { 0.5,  1.5,  0.0}, { 1.5,  1.5, -1.0}}
```

Initialization Code:

```
glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
        0, 1, 12, 4, &ctrlpoints[0][0][0]);
 glEnable(GL_MAP2_VERTEX_3);
 glEnable(GL_AUTO_NORMAL);
 glEnable(GL_NORMALIZE);
 glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
```

Display Code:

```
glEvalMesh2(GL_FILL, 0, 20, 0, 20);
```



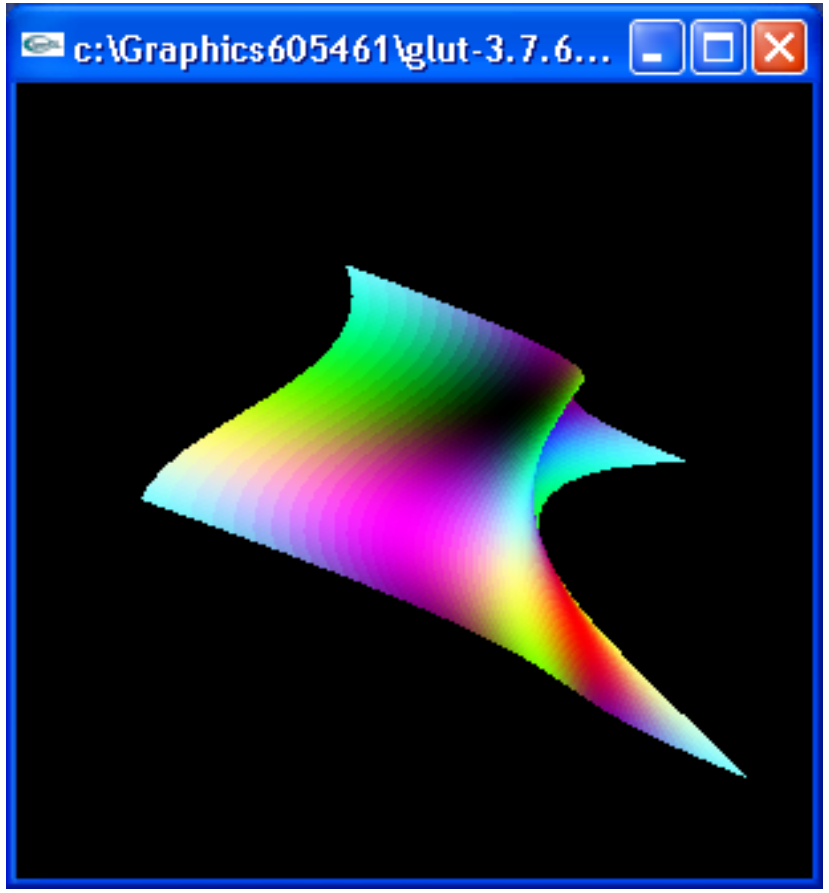
Using Evaluators for Textures

- Can use evaluators to generate texture coordinates for the surface
  - Converts u,v into texture s,t
    - Same as the u,v

```
// Define flat patch for texture points
GLfloat texpts[2][2][2] =
          {{{0.0, 0.0}, {0.0, 1.0}},
          {{ 1.0, 0.0}, {1.0, 1.0}}};

// Initialization
glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
        0, 1, 12, 4, &ctrlpoints[0][0][0]);
glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2,
        0, 1, 4, 2, &texpts[0][0][0]);
glEnable(GL_MAP2_TEXTURE_COORD_2);
glEnable(GL_MAP2_VERTEX_3);
glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);

// Texturing parameters…use texture wrapping
glEnable(GL_TEXTURE_2D);
```



OpenGL GLU NURBS Interface

- OpenGL Utility Library offers a Non-Uniform Rational B-Spline (NURBS) interface
  - Built on top of the OpenGL evaluator methods
- Basic steps to drawing NURBS curves and surfaces:

1. Call glEnable(GL_AUTO_NORMAL) to automatically generate surface normals
2. Create a NURBS object using gluNewNurbsRenderer()
3. Set NURBS properties with gluNurbsProperty()
4. Set error notification callbacks using gluNurbsCallback()
5. Start the curve or surface with gluBeginCurve() or gluBeginSurface()
6. Generate and render the curve with gluNurbsCurve() or gluNurbsSurface(). These methods accept control points, knot sequence, and order of the polynomial basis function. Call these methods additional times to specify surface normals or texture coordinates.
7. Complete the curve or surface with gluEndCurve() or gluEndSurface()

Managing a NURBS Object and Controlling Rendering Properties

`GLUNurbsObj* gluNewNurbsRenderer(void);`
Creates a new NURBS object and returns a pointer to it.

`void gluDeleteNurbsRenderer(void);`
Destroys the specified NURBS object.

`void gluNurbsProperty(GLUNurbsObj* obj, GLenum property, GLfloat value);`
Controls attributes of the specified NURBS object. Property controls include:
`GLU_DISPLAY_MODE`: `GLU_FILL` (default), `GLU_OUTLINE_POLYGON`, `GLU_OUTLINE_PATCH`
`GLU_CULLING`: Can speed up performance by not tessellating NURBS object falls completely outside the view volume

The following control how the curves are sampled in the u,v parameters
`GLU_SAMPLING_METHOD`, `GLU_SAMPLING_TOLERANCE`, `GLU_U_STEP`, `GLU_V_STEP`

`GLU_AUTO_LOAD_MATRIX`: If true the current projection, modelview, and viewport are used to compute sampling and culling. Otherwise the user supplies these matrices with **gluLoadSamplingMatrices**.

Creating a NURBS Surface

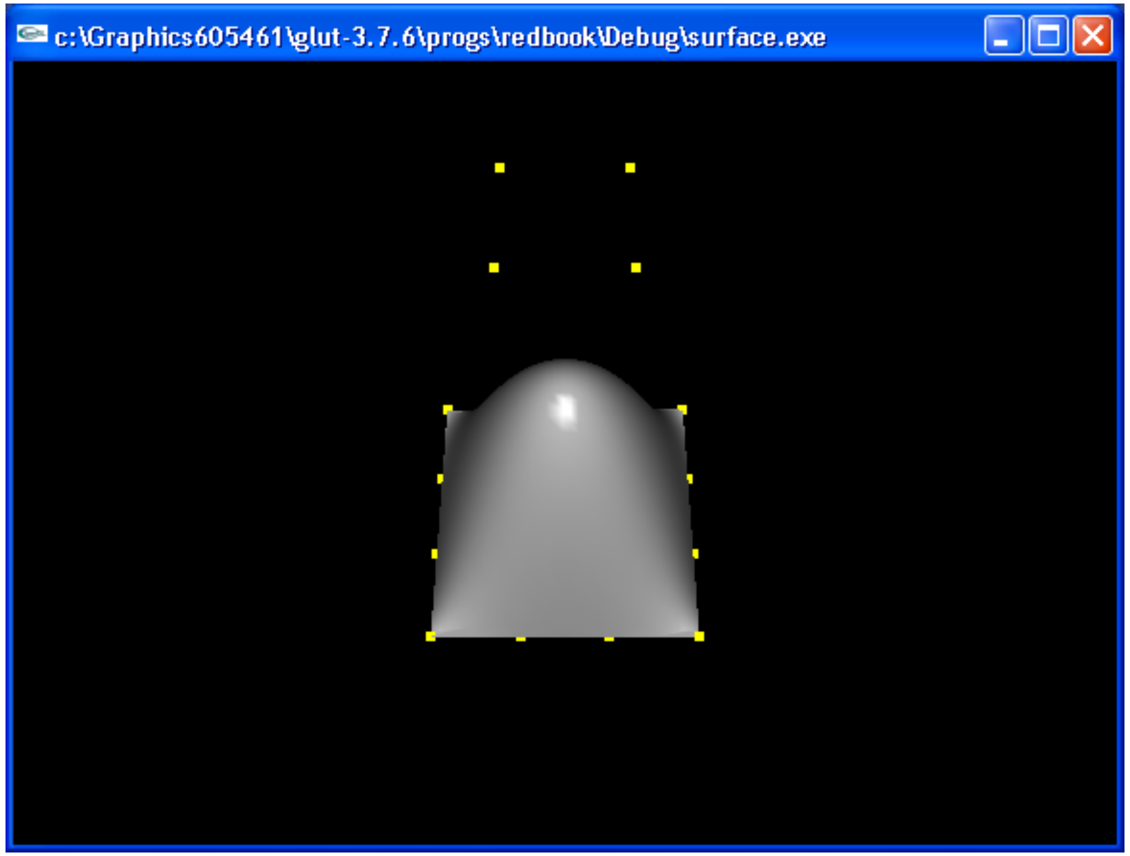
```
void gluBeginSurface(GLUNurbsObj* obj);
void gluEndSurface (GLUNurbsObj* obj);
```

Delineates the begin and end of a NURBS surface. One or more calls to gluNurbsSurface occur in between these method.

```
void gluNurbsSurface(GLUNurbsObj* obj, GLint uKnotCount, GLfloat* uknot,
                     GLint vKnotCount, GLfloat* vknot, GLint uStride, GLint vStride,
                     GLfloat* controlPoints, GLint uRrder, GLInt vOrder, GLenum type);
```

Describes the surface. u and v knot vectors are supplied. Order of the polynomial for both u and v are supplied. Control points are supplied. Stride values specify the number of floating point values between control points in each parametric direction.
Type is one of the two-dimensional evaluator types:
GL_MAP2_VERTEX_3 for non-rational control points
GL_MAP2_VERTEX_4 for rational control points
GL_MAP2_TEXTURE_COORD_2 to calculate and assign texture coordinates
GL_MAP2_NORMAL to calculate and assign surface normals

OpenGL GLU NURBS Interface


c:\Graphics605461\glut-3.7.6\progs\redbook\Debug\surface.exe

NURBS surface with control points shown

Basis function is a cubic B-spline, non-uniform knot sequence with multiplicity of 4 at each endpoint (causing the basis function to behave like a Bezier curve in each direction).

NURBS with OpenGL

Knot Values:

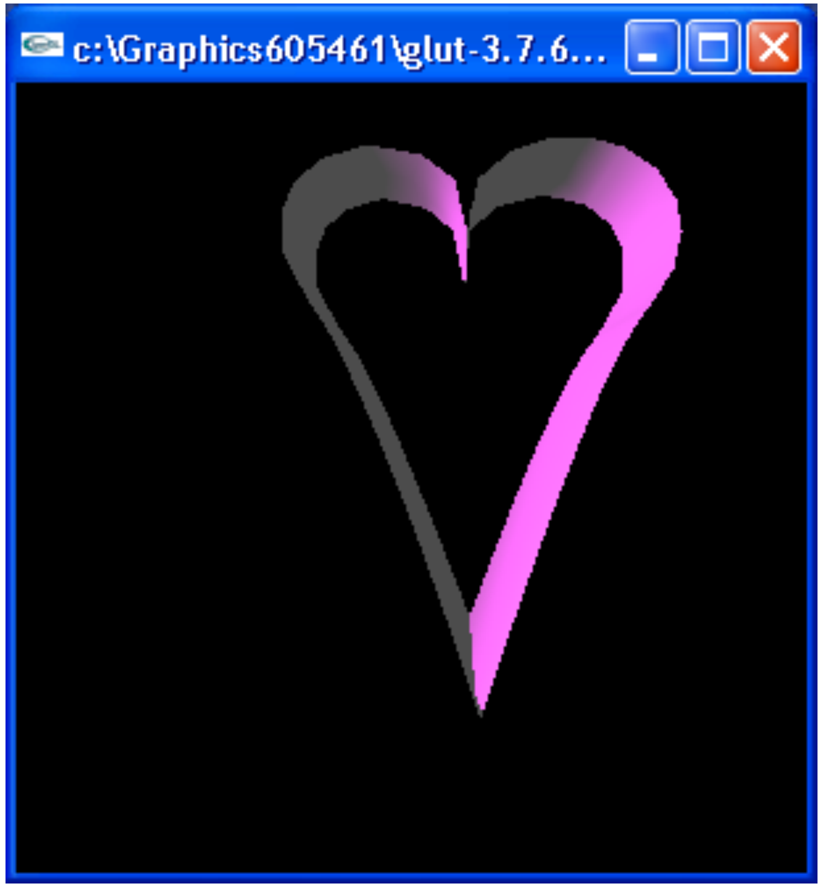
```
GLfloat sknots[S_NUMKNOTS] =
    {-1.0, -1.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0,
      4.0,  5.0,  6.0, 7.0, 8.0, 9.0, 9.0, 9.0};
GLfloat tknots[T_NUMKNOTS] = {1.0, 1.0, 1.0,
                              2.0, 2.0, 2.0};
```

Initialization:

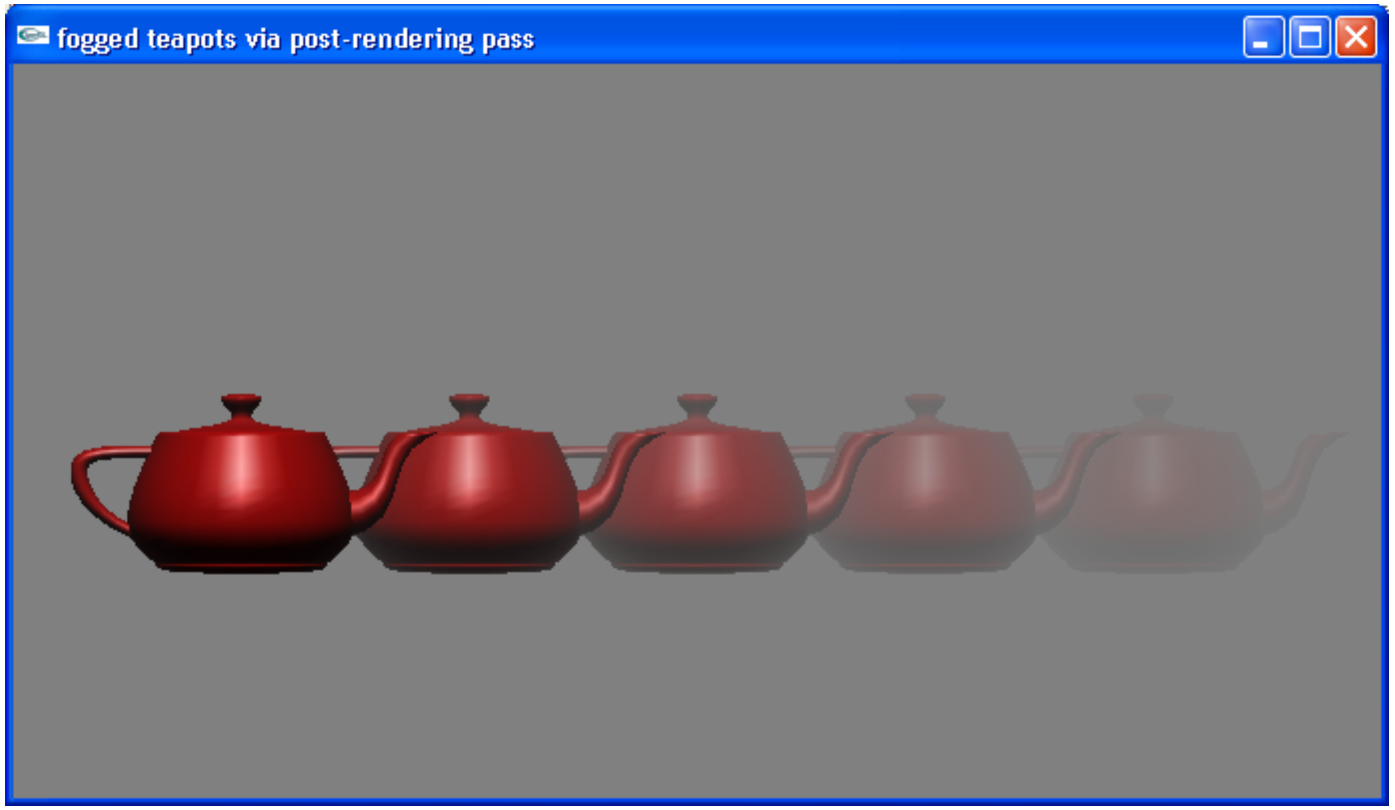
```
nurb = gluNewNurbsRenderer();
gluNurbsProperty(nurb,
                 GLU_SAMPLING_TOLERANCE,
                 25.0);
gluNurbsProperty(nurb,
                 GLU_DISPLAY_MODE,
                 GLU_FILL);
```

Display:

```
gluBeginSurface(nurb);
gluNurbsSurface(nurb,
              S_NUMKNOTS, sknots,
              T_NUMKNOTS, tknots,
              4 * T_NUMPOINTS,
              4, &ctlpoints[0][0][0],
              S_ORDER, T_ORDER,
              GL_MAP2_VERTEX_4);
gluEndSurface(nurb);
```



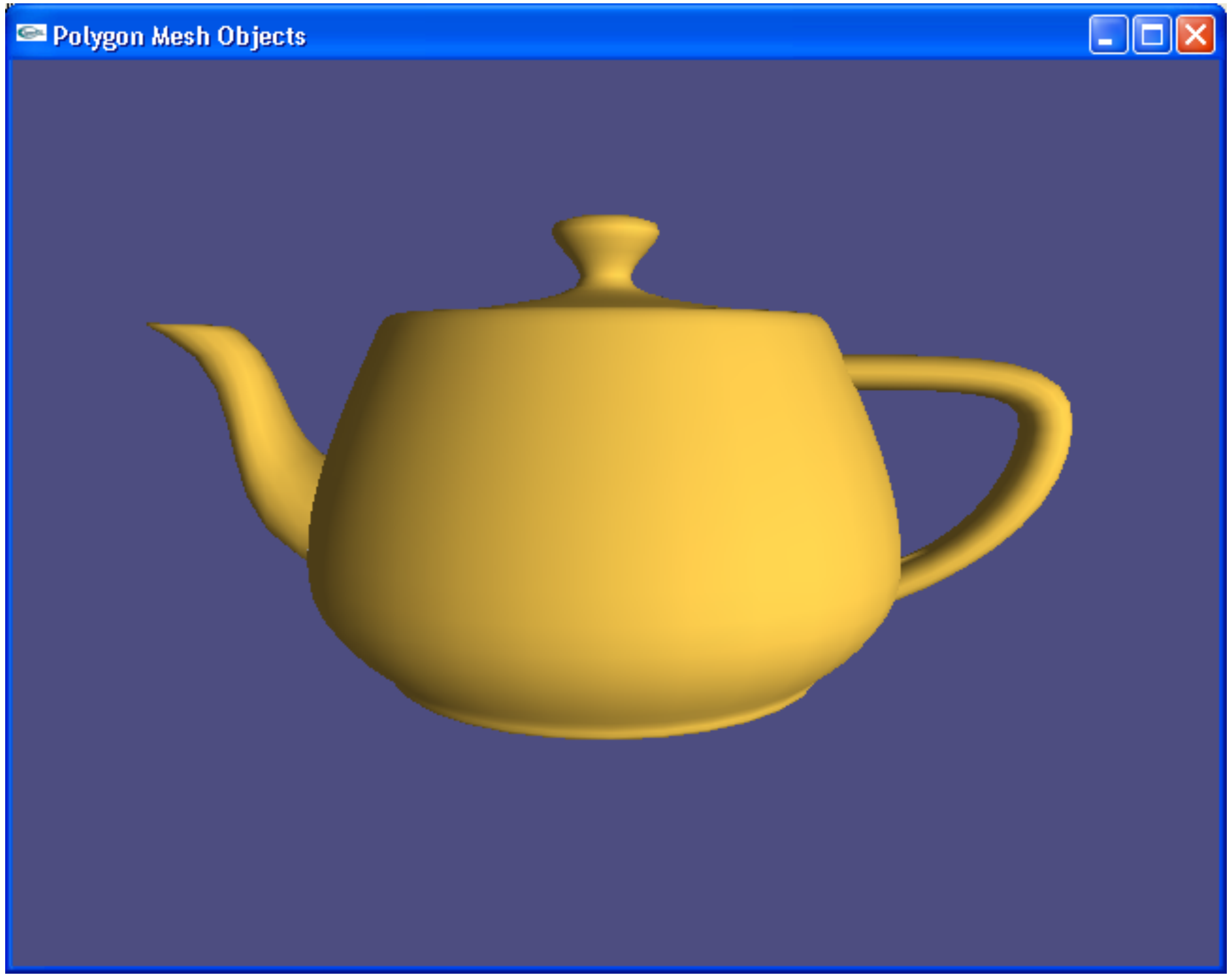
Teapot using GLUT



```
glutSolidTeapot(1.0);
```

Argument: Relative size of the teapot
The teapot is generated with OpenGL evaluators
This teapot is rendered with its front facing polygon vertices winding clockwise

Teapot using Subdivision (3 levels)


Polygon Mesh Objects