GLU Nurbs

- gluNewNurbsRenderer
- gluNurbsProperty
- gluBeginNurbsCurve
- gluNurbsCurve
- gluEndNurbsCurve

gluNewNurbsRenderer

- GLUnurbsObj* gluNewNurbsRenderer (void)

gluNurbsProperty

```
gluNurbsProperty (
    GLUnurbsObj *nobj,
    Glenum property, :
    Glfloat value )

    gluNurbsProperty (nobj,
    GLU_SAMPLING_METHOD,
    GLU_DOMAIN_DISTANCE)

    gluNurbsProperty (nobj, GLU_U_STEP, 100)
```

gluBeginCurve /gluEndCurve

```
gluBeginCurve (
    GLUnurbsObj *nobj)
```

gluNurbsCurve

```
gluNurbsCurve(
    GLUnurbsobj *nobj       :objname
    GLint uknot_count       :12
    Glfloat *uknot          :{0,0,0,0,1,2,3,4,5,5,5,5}
    GLint u_stride          :3
    GLfloat *ctlarray       :ctlarray
    GLint uorder            :4
    GLenum type)            :GL_MAP1_VERTEX_3
```

gluBeginSurface /gluEndSurface

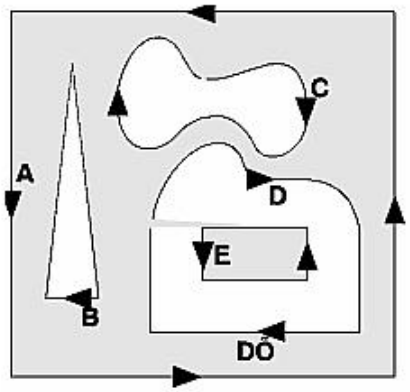
```
gluBeginSurface (
    GLUnurbsObj *nobj)
```

gluNurbsSurface

gluNurbsSurface(

GLUnurbsobj *nobj	:objname
GLint uknot_count	:12
Glfloat *uknot	:{0,0,0,0,1,2,3,4,5,5,5,5}
GLint vknot_count	:12
Glfloat *vknot	:{0,0,0,0,1,2,3,4,5,5,5,5}
GLint u_stride	:12
GLint v_stride	:3
GLfloat *ctlarray	:ctlarray
GLint uorder	:4
GLint vorder	:4
GLenum type)	:GL_MAP2_VERTEX_3

Trimmed Nurbs Surface




```
gluBeginSurface();
gluNurbsSurface(...);
gluBeginTrim();
gluPwlCurve(...); /* A */
gluEndTrim();
gluBeginTrim();
gluPwlCurve(...); /* B */
gluEndTrim();
gluBeginTrim();
gluNurbsCurve(...);/* C */
gluEndTrim();
gluBeginTrim();
gluNurbsCurve(...);/* D */
gluPwlCurve(...); /* DO */
gluEndTrim();
gluBeginTrim();
gluPwlCurve(...); /* E */
gluEndTrim();
gluEndSurface();
```

```
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

The GLU NURBS Interface

Although evaluators are the only OpenGL primitive available to draw curves and surfaces directly, and even though they can be implemented very efficiently in hardware, they're often accessed by applications through higher-level libraries. The GLU provides a NURBS (Non-Uniform Rational B-Spline) interface built on top of the OpenGL evaluator commands.

A Simple NURBS Example

If you understand NURBS, writing OpenGL code to manipulate NURBS curves and surfaces is relatively easy, even with lighting and texture mapping. Follow these steps to draw NURBS curves or untrimmed NURBS surfaces. (See "Trim a NURBS Surface" for information about trimmed surfaces.)

1. If you intend to use lighting with a NURBS surface, call **glEnable()** with GL_AUTO_NORMAL to automatically generate surface normals. (Or you can calculate your own.)

2. Use **gluNewNurbsRenderer()** to create a pointer to a NURBS object, which is referred to when creating your NURBS curve or surface.

3. If desired, call **gluNurbsProperty()** to choose rendering values, such as the maximum size of lines or polygons that are used to render your NURBS object.

4. Call **gluNurbsCallback()** if you want to be notified when an error is encountered. (Error checking may slightly degrade performance but is still highly recommended.)

5. Start your curve or surface by calling **gluBeginCurve()** or **gluBeginSurface()**.

6. Generate and render your curve or surface. Call **gluNurbsCurve()** or **gluNurbsSurface()** at least once with the control points (rational or nonrational), knot sequence, and order of the polynomial basis function for your NURBS object. You might call these functions additional times to specify surface normals and/or texture coordinates.

7. Call **gluEndCurve()** or **gluEndSurface()** to complete the curve or surface.

Example 12-5 renders a NURBS surface in the shape of a symmetrical hill with control points ranging from -3.0 to 3.0. The basis function is a cubic B-spline, but the knot sequence is nonuniform, with a multiplicity of 4 at each endpoint, causing the basis function to behave like a Bézier curve in each direction. The surface is lighted, with a dark gray diffuse reflection and white specular highlights. Figure 12-4 shows the surface as a lit wireframe.

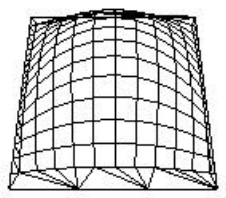


Figure 12-4 : NURBS Surface

Example 12-5 : NURBS Surface: surface.c

```
#include <GL/gl.h>
#include <GL/glu.h>
```

```
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

GLfloat ctlpoints[4][4][3];
int showPoints = 0;

GLUnurbsObj *theNurb;

void init_surface(void)
{
   int u, v;
   for (u = 0; u < 4; u++) {
      for (v = 0; v < 4; v++) {
         ctlpoints[u][v][0] = 2.0*((GLfloat)u - 1.5);
         ctlpoints[u][v][1] = 2.0*((GLfloat)v - 1.5);

         if ( (u == 1 || u == 2) && (v == 1 || v == 2))
            ctlpoints[u][v][2] = 3.0;
         else
            ctlpoints[u][v][2] = -3.0;
      }
   }
}

void nurbsError(GLenum errorCode)
{
   const GLubyte *estring;

   estring = gluErrorString(errorCode);
   fprintf (stderr, "Nurbs Error: %s\n", estring);
   exit (0);
}

void init(void)
{
   GLfloat mat_diffuse[] = { 0.7, 0.7, 0.7, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 100.0 };

   glClearColor (0.0, 0.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);

   init_surface();

   theNurb = gluNewNurbsRenderer();
   gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
   gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
   gluNurbsCallback(theNurb, GLU_ERROR,
                    (GLvoid (*)()) nurbsError);
}

void display(void)
{
   GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
   int i, j;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix();
   glRotatef(330.0, 1.,0.,0.);
   glScalef (0.5, 0.5, 0.5);

   gluBeginSurface(theNurb);
   gluNurbsSurface(theNurb,
                   8, knots, 8, knots,
                   4 * 3, 3, &ctlpoints[0][0][0],
                   4, 4, GL_MAP2_VERTEX_3);
```

```
   gluEndSurface(theNurb);

   if (showPoints) {
      glPointSize(5.0);
      glDisable(GL_LIGHTING);
      glColor3f(1.0, 1.0, 0.0);
      glBegin(GL_POINTS);
      for (i = 0; i < 4; i++) {
         for (j = 0; j < 4; j++) {
            glVertex3f(ctlpoints[i][j][0],
                       ctlpoints[i][j][1], ctlpoints[i][j][2]);
         }
      }
      glEnd();
      glEnable(GL_LIGHTING);
   }
   glPopMatrix();
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective (45.0, (GLdouble)w/(GLdouble)h, 3.0, 8.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef (0.0, 0.0, -5.0);
}
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case `c':
      case `C':
         showPoints = !showPoints;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutKeyboardFunc (keyboard);
   glutMainLoop();
   return 0;
}
```

Manage a NURBS Object

As shown in Example 12-5, **gluNewNurbsRenderer()** returns a new NURBS object, whose type is a pointer to a GLUnurbsObj structure. You must make this object before using any other NURBS routine. When you're done with a NURBS object, you may use **gluDeleteNurbsRenderer()** to free up the memory that was used.

*GLUnurbsObj** ***gluNewNurbsRenderer*** *(void);*

Creates a new NURBS object, nobj. Returns a pointer to the new object, or zero, if OpenGL cannot allocate memory for a new NURBS object.

void ***gluDeleteNurbsRenderer*** *(GLUnurbsObj *nobj);*

Destroys the NURBS object nobj.

Control NURBS Rendering Properties

A set of properties associated with a NURBS object affects the way the object is rendered. These properties include how the surface is rasterized (for example, filled or wireframe) and the precision of tessellation.

*void **gluNurbsProperty**(GLUnurbsObj *nobj, GLenum property,*
GLfloat value);

Controls attributes of a NURBS object, nobj. The property argument specifies the property and can be GLU_DISPLAY_MODE, GLU_CULLING, GLU_SAMPLING_METHOD, GLU_SAMPLING_TOLERANCE, GLU_PARAMETRIC_TOLERANCE, GLU_U_STEP, GLU_V_STEP, or GLU_AUTO_LOAD_MATRIX. The value argument indicates what the property should be.

The default value for GLU_DISPLAY_MODE is GLU_FILL, which causes the surface to be rendered as polygons. If GLU_OUTLINE_POLYGON is used for the display-mode property, only the outlines of polygons created by tessellation are rendered. GLU_OUTLINE_PATCH renders the outlines of patches and trimming curves. (See "Create a NURBS Curve or Surface".)

GLU_CULLING can speed up performance by not performing tessellation if the NURBS object falls completely outside the viewing volume; set this property to GL_TRUE to enable culling (the default is GL_FALSE).

Since a NURBS object is rendered as primitives, it's sampled at different values of its parameter(s) (u and v) and broken down into small line segments or polygons for rendering. If property is GLU_SAMPLING_METHOD, then value is set to one of GLU_PATH_LENGTH (which is the default), GLU_PARAMETRIC_ERROR, or GLU_DOMAIN_DISTANCE, which specifies how a NURBS curve or surface should be tessellated. When value is set to GLU_PATH_LENGTH, the surface is rendered so that the maximum length, in pixels, of the edges of tessellated polygons is no greater than what is specified by GLU_SAMPLING_TOLERANCE. When set to GLU_PARAMETRIC_ERROR, then the value specified by GLU_PARAMETRIC_TOLERANCE is the maximum distance, in pixels, between tessellated polygons and the surfaces they approximate. When set to GLU_DOMAIN_DISTANCE, the application specifies, in parametric coordinates, how many sample points per unit length are taken in the u and v dimensions, using the values for GLU_U_STEP and GLU_V_STEP.

If property is GLU_SAMPLING_TOLERANCE and the sampling method is GLU_PATH_LENGTH, value controls the maximum length, in pixels, to use for tessellated polygons. The default value of 50.0 makes the largest sampled line segment or polygon edge 50.0 pixels long. If property is GLU_PARAMETRIC_TOLERANCE and the sampling method is GLU_PARAMETRIC_ERROR, value controls the maximum distance, in pixels, between the tessellated polygons and the surfaces they approximate. The default value for GLU_PARAMETRIC_TOLERANCE is 0.5, which makes the tessellated polygons within one-half pixel of the approximated surface. If the sampling method is GLU_DOMAIN_DISTANCE and property is either GLU_U_STEP or GLU_V_STEP, then value is the number of sample points per unit length taken along the u or v dimension, respectively, in parametric coordinates. The default for both GLU_U_STEP and GLU_V_STEP is 100.

*The GLU_AUTO_LOAD_MATRIX property determines whether the projection matrix, modelview matrix, and viewport are downloaded from the OpenGL server (GL_TRUE, the default), or whether the application must supply these matrices with **gluLoadSamplingMatrices()** (GL_FALSE).*

*void **gluLoadSamplingMatrices** (GLUnurbsObj *nobj, const GLfloat modelMatrix[16], const GLfloat projMatrix[16], const GLint viewport[4]);*

*If the GLU_AUTO_LOAD_MATRIX is turned off, the modelview and projection matrices and the viewport specified in **gluLoadSamplingMatrices()** are used to compute sampling and culling matrices for each NURBS curve or surface.*

If you need to query the current value for a NURBS property, you may use **gluGetNurbsProperty()**.

*void **gluGetNurbsProperty** (GLUnurbsObj *nobj, GLenum property,*
*GLfloat *value);*

Given the property to be queried for the NURBS object nobj, return its current value.

Handle NURBS Errors

Since there are 37 different errors specific to NURBS functions, it's a good idea to register an error callback to let you know if you've stumbled into one of them. In Example 12-5, the callback function was registered with

```
gluNurbsCallback(theNurb, GLU_ERROR, (GLvoid (*)()) nurbsError);
```

*void **gluNurbsCallback** (GLUnurbsObj *nobj, GLenum which,*
*void (*fn)(GLenum errorCode));*

*which is the type of callback; it must be GLU_ERROR. When a NURBS function detects an error condition, fn is invoked with the error code as its only argument. errorCode is one of 37 error conditions, named GLU_NURBS_ERROR1 through GLU_NURBS_ERROR37. Use **gluErrorString()** to describe the meaning of those*

error codes.

In Example 12-5, the **nurbsError()** routine was registered as the error callback function:

```
void nurbsError(GLenum errorCode)
{
   const GLubyte *estring;

   estring = gluErrorString(errorCode);
   fprintf (stderr, "Nurbs Error: %s\n", estring);
   exit (0);
}
```

Create a NURBS Curve or Surface

To render a NURBS surface, **gluNurbsSurface()** is bracketed by **gluBeginSurface()** and **gluEndSurface()**. The bracketing routines save and restore the evaluator state.

*void **gluBeginSurface** (GLUnurbsObj *nobj);*
*void **gluEndSurface** (GLUnurbsObj *nobj);*

*After **gluBeginSurface()**, one or more calls to **gluNurbsSurface()** defines the attributes of the surface. Exactly one of these calls must have a surface type of GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4 to generate vertices. Use **gluEndSurface()** to end the definition of a surface. Trimming of NURBS surfaces is also supported between **gluBeginSurface()** and **gluEndSurface()**. (See "Trim a NURBS Surface".)*

*void **gluNurbsSurface** (GLUnurbsObj *nobj, GLint uknot_count,*
*GLfloat *uknot, GLint vknot_count, GLfloat *vknot,*
*GLint u_stride, GLint v_stride, GLfloat *ctlarray,*
GLint uorder, GLint vorder, GLenum type);

Describes the vertices (or surface normals or texture coordinates) of a NURBS surface, nobj. Several of the values must be specified for both u and v parametric directions, such as the knot sequences (uknot and vknot), knot counts (uknot_count and vknot_count), and order of the polynomial (uorder and vorder) for the NURBS surface. Note that the number of control points isn't specified. Instead, it's derived by determining the number of control points along each parameter as the number of knots minus the order. Then, the number of control points for the surface is equal to the number of control points in each parametric direction, multiplied by one another. The ctlarray argument points to an array of control points.

The last parameter, type, is one of the two-dimensional evaluator types. Commonly, you might use GL_MAP2_VERTEX_3 for nonrational or GL_MAP2_VERTEX_4 for rational control points, respectively. You might also use other types, such as GL_MAP2_TEXTURE_COORD_* or GL_MAP2_NORMAL to calculate and assign texture coordinates or surface normals. For example, to create a lighted (with surface normals) and textured NURBS surface, you may need to call this sequence:

```
gluBeginSurface(nobj);
   gluNurbsSurface(nobj, ..., GL_MAP2_TEXTURE_COORD_2);
   gluNurbsSurface(nobj, ..., GL_MAP2_NORMAL);
   gluNurbsSurface(nobj, ..., GL_MAP2_VERTEX_3);
gluEndSurface(nobj);
```

*The u_stride and v_stride arguments represent the number of floating-point values between control points in each parametric direction. The evaluator type, as well as its order, affects the u_stride and v_stride values. In Example 12-5, u_stride is 12 (4 * 3) because there are three coordinates for each vertex (set by GL_MAP2_VERTEX_3) and four control points in the parametric v direction; v_stride is 3 because each vertex had three coordinates, and v control points are adjacent to one another.*

Drawing a NURBS curve is similar to drawing a surface, except that all calculations are done with one parameter, *u*, rather than two. Also, for curves, **gluBeginCurve()** and **gluEndCurve()** are the bracketing routines.

*void **gluBeginCurve** (GLUnurbsObj *nobj);*
*void **gluEndCurve** (GLUnurbsObj *nobj);*

*After **gluBeginCurve()**, one or more calls to **gluNurbsCurve()** define the attributes of the surface. Exactly one of these calls must have a surface type of GL_MAP1_VERTEX_3 or GL_MAP1_VERTEX_4 to generate vertices. Use **gluEndCurve()** to end the definition of a surface.*

*void **gluNurbsCurve** (GLUnurbsObj *nobj, GLint uknot_count,*
*GLfloat *uknot, GLint u_stride, GLfloat *ctlarray,*
GLint uorder, GLenum type);

*Defines a NURBS curve for the object nobj. The arguments have the same meaning as those for **gluNurbsSurface()**.*

*Note that this routine requires only one knot sequence and one declaration of the order of the NURBS object. If this curve is defined within a **gluBeginCurve()/gluEndCurve()** pair, then the type can be any of the valid one-dimensional evaluator types (such as GL_MAP1_VERTEX_3 or GL_MAP1_VERTEX_4).*

Trim a NURBS Surface

To create a trimmed NURBS surface with OpenGL, start as if you were creating an untrimmed surface. After calling **gluBeginSurface()** and **gluNurbsSurface()** but before calling **gluEndSurface()**, start a trim by calling **gluBeginTrim()**.

*void **gluBeginTrim** (GLUnurbsObj *nobj);*
*void **gluEndTrim** (GLUnurbsObj *nobj);*

Marks the beginning and end of the definition of a trimming loop. A trimming loop is a set of oriented, trimming curve segments (forming a closed curve) that defines the boundaries of a NURBS surface.

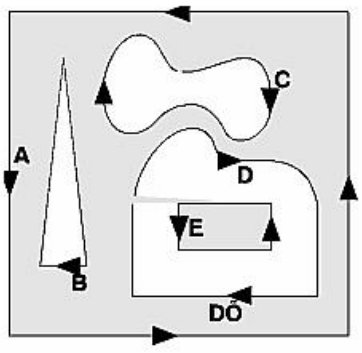
You can create two kinds of trimming curves, a piecewise linear curve with **gluPwlCurve()** or a NURBS curve with **gluNurbsCurve()**. A piecewise linear curve doesn't look like what's conventionally called a curve, because it's a series of straight lines. A NURBS curve for trimming must lie within the unit square of parametric (*u*, *v*) space. The type for a NURBS trimming curve is usually GLU_MAP1_TRIM2. Less often, the type is GLU_MAP1_TRIM3, where the curve is described in a two-dimensional homogeneous space (*u'*, *v'*, *w'*) by (*u*, *v*) = (*u'*/*w'*, *v'*/*w'*).

*void **gluPwlCurve** (GLUnurbsObj *nobj, GLint count, GLfloat *array,*
GLint stride, GLenum type);

Describes a piecewise linear trimming curve for the NURBS object nobj. There are count points on the curve, and they're given by array. The type can be either GLU_MAP1_TRIM_2 (the most common) or GLU_MAP1_TRIM_3 ((u, v, w) homogeneous parameter space). The type affects whether stride, the number of floating-point values to the next vertex, is 2 or 3.

You need to consider the orientation of trimming curves - that is, whether they're counterclockwise or clockwise - to make sure you include the desired part of the surface. If you imagine walking along a curve, everything to the left is included and everything to the right is trimmed away. For example, if your trim consists of a single counterclockwise loop, everything inside the loop is included. If the trim consists of two nonintersecting counterclockwise loops with nonintersecting interiors, everything inside either of them is included. If it consists of a counterclockwise loop with two clockwise loops inside it, the trimming region has two holes in it. The outermost trimming curve must be counterclockwise. Often, you run a trimming curve around the entire unit square to include everything within it, which is what you get by default by not specifying any trimming curves.

Trimming curves must be closed and nonintersecting. You can combine trimming curves, so long as the endpoints of the trimming curves meet to form a closed curve. You can nest curves, creating islands that float in space. Be sure to get the curve orientations right. For example, an error results if you specify a trimming region with two counterclockwise curves, one enclosed within another: The region between the curves is to the left of one and to the right of the other, so it must be both included and excluded, which is impossible. Figure 12-5 illustrates a few valid possibilities.




```
gluBeginSurface();
gluNurbsSurface(...);
gluBeginTrim();
gluPwlCurve(...); /* A */
gluEndTrim();
gluBeginTrim();
gluPwlCurve(...); /* B */
gluEndTrim();
gluBeginTrim();
gluNurbsCurve(...);/* C */
gluEndTrim();
gluBeginTrim();
gluNurbsCurve(...);/* D */
gluPwlCurve(...); /* DO */
gluEndTrim();
gluBeginTrim();
gluPwlCurve(...); /* E */
gluEndTrim();
gluEndSurface();
```

Figure 12-5 : Parametric Trimming Curves

Figure 12-6 shows the same small hill as in Figure 12-4, this time with a trimming curve that's a combination of a piecewise linear curve and a NURBS curve. The program that creates this figure is similar to that shown in Example 12-5; the differences are in the routines shown in Example 12-6.

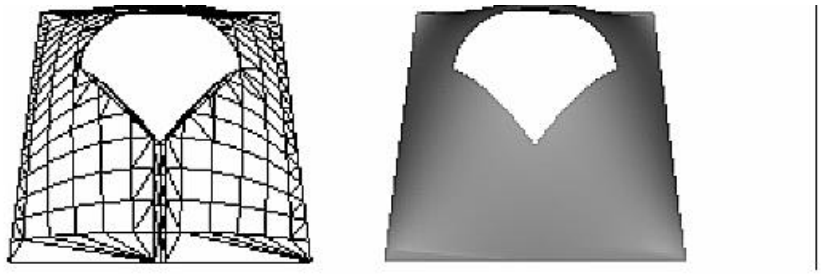


Figure 12-6 : Trimmed NURBS Surface

Example 12-6 : Trimming a NURBS Surface: trim.c

```
void display(void)
{
   GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
   GLfloat edgePt[5][2] = /* counter clockwise */
      {{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
       {0.0, 0.0}};
   GLfloat curvePt[4][2] = /* clockwise */
      {{0.25, 0.5}, {0.25, 0.75}, {0.75, 0.75}, {0.75, 0.5}};
   GLfloat curveKnots[8] =
      {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
   GLfloat pwlPt[4][2] = /* clockwise */
      {{0.75, 0.5}, {0.5, 0.25}, {0.25, 0.5}};

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
   glRotatef(330.0, 1.,0.,0.);
   glScalef (0.5, 0.5, 0.5);

   gluBeginSurface(theNurb);
   gluNurbsSurface(theNurb, 8, knots, 8, knots,
                   4 * 3, 3, &ctlpoints[0][0][0],
                   4, 4, GL_MAP2_VERTEX_3);
   gluBeginTrim (theNurb);
      gluPwlCurve (theNurb, 5, &edgePt[0][0], 2,
                   GLU_MAP1_TRIM_2);
   gluEndTrim (theNurb);
   gluBeginTrim (theNurb);
      gluNurbsCurve (theNurb, 8, curveKnots, 2,
                     &curvePt[0][0], 4, GLU_MAP1_TRIM_2);
      gluPwlCurve (theNurb, 3, &pwlPt[0][0], 2,
                   GLU_MAP1_TRIM_2);
   gluEndTrim (theNurb);
   gluEndSurface(theNurb);

   glPopMatrix();
   glFlush();
}
```

In Example 12-6, **gluBeginTrim()** and **gluEndTrim()** bracket each trimming curve. The first trim, with vertices defined by the array *edgePt[][]*, goes counterclockwise around the entire unit square of parametric space. This ensures that everything is drawn, provided it isn't removed by a clockwise trimming curve inside of it. The second trim is a combination of a NURBS trimming curve and a piecewise linear trimming curve. The NURBS curve ends at the points (0.9, 0.5) and (0.1, 0.5), where it is met by the piecewise linear curve, forming a closed clockwise curve.