PUBLISHED IN THE
JANUARY/FEBRUARY 1996 ISSUE OF *The X Journal*.

OPENGL® AND X, COLUMN 8: CURVES AND SURFACES IN OPENGL

Mark J. Kilgard *
Silicon Graphics Inc.
$Revision: 1.9$

November 5, 1995

Welcome to the OpenGL column that discusses topics in 3D programming using OpenGL. For new readers, the OpenGL graphics system is a programming interface for interactive 3D graphics. OpenGL is window system independent and is available for the X Window System, Windows NT and 95, and OS/2. Long-time readers have read here of OpenGL's growing acceptance among X workstation vendors. This acceptance has now culminated with Hewlett-Packard's September announcement to adopt OpenGL. Now, X developers and users can rely on a single high-performance 3D interface that is portable even to non-X platforms.

This issue's topic is rendering curves and surfaces using OpenGL. Rendering smooth curves and surfaces efficiently is very important to professionals such as mechanical engineers and computer special-effects animators who work with objects that are typically modeled with curves and surfaces. Applications like computer aided design (CAD) often depend on fast rendering of 3D curves and surfaces.

Examples of curves are splines, circles, and spiral paths. What distinguishes a surface from a curve is that a surface has area, while a curve has length. A surface can be a sphere, or a curved patch, or even a flat surface like a rectangle (though for this discussion, we generally assume a curved surface since flat surfaces need no special support to render with OpenGL).

The basic 3D rendering primitive used by OpenGL is the polygon. These polygons are *flat*. Complex objects are typically modeled as tens, hundreds, or even thousands of individual polygons (typically triangles). An elephant, a V8 engine, or a DNA molecule can all be rendered as the amalgamation of enough triangles so that the viewer can recognize the modeled object.

*Mark graduated with B.A. in Computer Science from Rice University and is a Member of the Technical Staff at Silicon Graphics. He can be reached by electronic mail addressed to `mjk@sgi.com`

Why Curves and Surfaces?

While triangles are sufficient to model any 3D solid object, they are generally neither the most convenient, compact, nor accurate graphical representation. Representing something like a sphere or torus (doughnut) with hundreds of triangles works, but a mathematical representation of these objects as surfaces is both smaller and easier to manipulate. Also, using triangles to represent a curved surface is always an approximation.

OpenGL does not support curves and surfaces as fundamental rendering primitives. Instead, OpenGL provides facilities to break curves and surfaces up into lines or polygons for efficient rendering. Curves and surfaces are specified by *control points* that "shape" the curve or surface. Control points may act as "anchors" that the curve or surface must pass through, or they may act like "magnets" that simply attract the curve. From these control points, OpenGL can automatically generate (for any specified precision) a set of lines or polygons that approximate the curve or surface. Figure 1 shows how a set of control points define a 3D spiraling curve.

This approach has several advantages:

- An application can treat curves and surfaces as pure mathematical representations, meaning they are compact and exact (not approximated).
- When converting a surface to polygons for rendering, the application can control the precision with which the surface is rendered. An application can tradeoff rendering accuracy for rendering speed.
- Hardware acceleration for OpenGL need not be burdened by supporting specialized curve and surface primitives. Curve and surface rendering utilizes the speed of accelerated line and polygon rendering hardware.

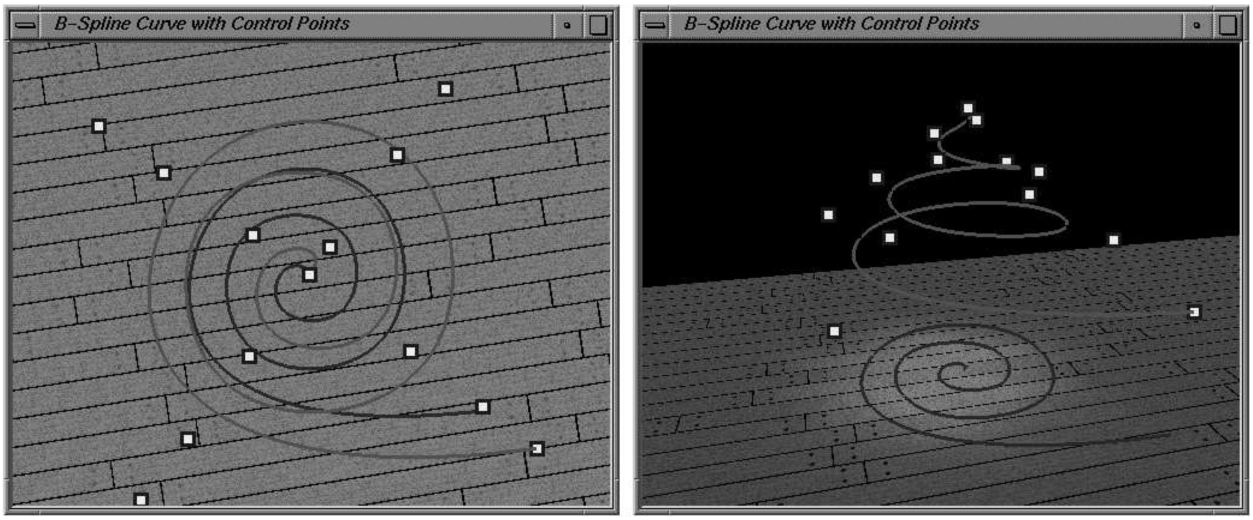

Figure 1: Two views of the control points for a spiral B-spline curve. Note how the control points pull the curve into a descending spiral. The knot multiplicity for the end points is 4, ensuring the end points of the spiral match the beginning and end control points.

- Even though curve and surface primitives are not supported directly, OpenGL has an *evaluator* facility for generating vertices, normals, colors, and texture coordinates from polynomial mappings that aids rendering curves and surfaces *and* that can be accelerated with hardware. The evaluator facility is not limited to supporting curves and surfaces and has other applications (like image warping).

OpenGL supports curves and surfaces in two ways: low-level evaluators and utility routines in the GLU library for generating *Non-Uniform Rational B-Splines* (or NURBS for short).

Do not let yourself be intimidated by the mathematical basis of these techniques; the intuition that underlies the use evaluators and NURBS for generating curves and surfaces is not exceedingly complicated. You can use curves and surfaces in OpenGL without a rigorous mathematical background. A straightforward, intuitive explanation of curves and surfaces can be found in the "Curves and Surfaces" chapter of *The Inventor Mentor* [5]. My purpose in this column is to provide a conceptual overview of evaluators and NURBS to get you started. The article concludes with a simple GLUT program[1] that calls the GLU library to render a few NURBS surfaces. If you are looking for more detail about OpenGL's programming interface to evaluators and NURBS, see the "Evaluators and NURBS" chapter in the *OpenGL Programming Guide* [4].

Evaluators

OpenGL evaluators provide a means to use polynomial or rational polynomial mappings[2] to generate vertices, normals, texture coordinates, and/or colors. Conceptually, evaluators supply a level of indirection between parameterized values (typically describing a curve or surface) and the generation of per-vertex information.

Bézier curves and surfaces are convenient representations of curves and surfaces. Commonly used in computer graphics, Bézier curves and surfaces are easily generated with evaluators. Pierre Bézier developed the theory of Bézier curves and surfaces for use in designing automobiles at Rénault. These curves and surfaces are described with a set of control points that define the end points and shape of the curve or surface.

Evaluator polynomial mappings are setup by calling one of the `glMap` family of routines. Both 1D and 2D evaluators are supported. A 1D evaluator describes a curve; a 2D evaluator describes a surface. `glEnable` and `glDisable` are used to enable and disable the various available evaluators.

Here is an example of generating a Bézier surface with OpenGL evaluators:

```
GLfloat control_pts[4][4][3] = { ... };

glMap2f(GL_MAP2_VERTEX_3,
  /* min u */    0.0,  /* max u */   1.0,
  /* u stride */ 3,    /* u order */ 4,
  /* min v */    0.0,  /* max v */   1.0,
  /* v stride */ 3,    /* v order */ 4,
  &control_pts[0][0][0]);
glEnable(GL_MAP2_VERTEX_3);
glEnable(GL_AUTO_NORMAL);
```

The code above sets up and enables the `GL_MAP2_VERTEX_3` evaluator map so that x, y, z vertices will be generated when the map is evaluated. Assume that the variables u and v are the values over which the evaluator mapping varies. The `glMap2f`

[1] The OpenGL Utility Toolkit (GLUT) is used for all the examples presented within this column. Source code for GLUT is available over the Internet at `ftp://sgigate.sgi.com/pub/opengl/xjournal/GLUT/`

[2] Evaluators are based on *Bernstein* polynomials.

call specifies that u and v vary over the range 0 and 1. Sixteen (4×4) control points are supplied. Each point is an x, y, and z float packed tightly in the `control_pts` array, so the stride for extracting points from the array is 3. The splines for both u and v on the surface are order 4. Enabling `GL_AUTO_NORMAL` will compute surface normal vectors analytically when the `GL_MAP2_VERTEX_3` is used to generate vertices (this makes lighting of surfaces easy!). Next, call:

```
glMapGrid2f(
  8,        /* u partitions */
  0.0, 1.0, /* range over u */
  8,        /* v partitions */
  0.0, 1.0  /* range over v */);
```

This establishes a 9×9 grid (8×8 steps) over the full u and v range of the mapping. The evaluator is now setup and ready to use. To render the surface, call:

```
glEvalMesh2(GL_FILL, 0, 8, 0, 8);
```

This command generates quad strips that will render the Bézier surface defined by the `control_pts` and automatically generates surface normals across the entire grid.

The same surface could be evaluated and rendered explicitly by an OpenGL program, but using evaluators has two important advantages:

- The single `glEvalMesh2` call above replaces 144 `glVertex3f` and 144 `glNormal3f` calls. This may greatly reduce the bus bandwidth to the graphics hardware (or network bandwidth if using OpenGL on a remote machine) required to render the surface.
- If the OpenGL implementation supports evaluators in the graphics hardware subsystem, evaluators off-load significant amounts of computation from the host CPU when rendering curves and surfaces. In the example above, a 4th order polynomial evaluation is required for each of 81 grid points.

In addition to supplying parameterized values as a mesh with `glEvalMesh2` or `glEvalMesh1`, the `glEvalCoord` family of routines generate evaluated per-vertex information for explicit parameterized values.

The GLU NURBS Routines

You can use evaluators to describe any polynomial or rational polynomial splines or surfaces of any degree, including Bézier curves and surfaces, Hermite splines, uniform B-splines, and NURBS; however, evaluators operate at a low level. Higher-level interfaces for rendering curves and surfaces can use evaluators for efficient implementation, but provide a more sophisticated interface for generating curves and surfaces. The OpenGL Utility library (called GLU for short) provides a powerful set of routines for rendering NURBS that use evaluators for actual rendering.

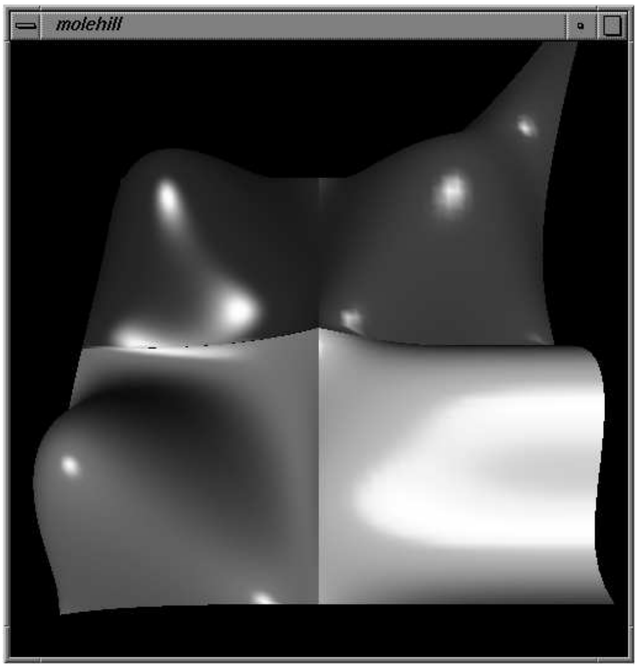

Figure 2: Snapshot of `molehill.c`'s display. The four colored NURBS surfaces have control points in different configurations.

Non-uniform rational B-splines provide a general and convenient description for curves and surfaces.[3] A NURBS curve or surface can be converted to an evaluator based description. The GLU NURBS routines provide a standard interface to do just that. This means you can use the GLU NURBS routines to describe curves and surfaces conveniently, and still obtain the potential for hardware acceleration of evaluators.

To specify a NURBS curve, you need a set of *control points*, a *knot sequence*, and the *order* of the curve.

Control points are points in object space that affect the shape of the curve in some way. The knot sequence lists non-decreasing values in the parameter domain that divide the parameter domain into segments. Conceptually, each knot serves as a boundary for the influence of corresponding control points. Having a knot value repeated multiple times in the sequence can eliminate the influence of neighboring control points, and thereby decreases the continuity of the curve. The number of times a knot is repeated for a given control point is called the *knot multiplicity* of the control point.

The order is implicitly determined by the number of knots *minus* the number of control points. The order affects how "smooth" the curve will be. Mathematically, the order is one more than the largest exponent in the parametric polynomial equations describing the curve.

Particular knot sequences specify various important classes of curves:

[3] The *Bézier* curve that is used as a basis for evaluators is a special of case of a NURBS curve.

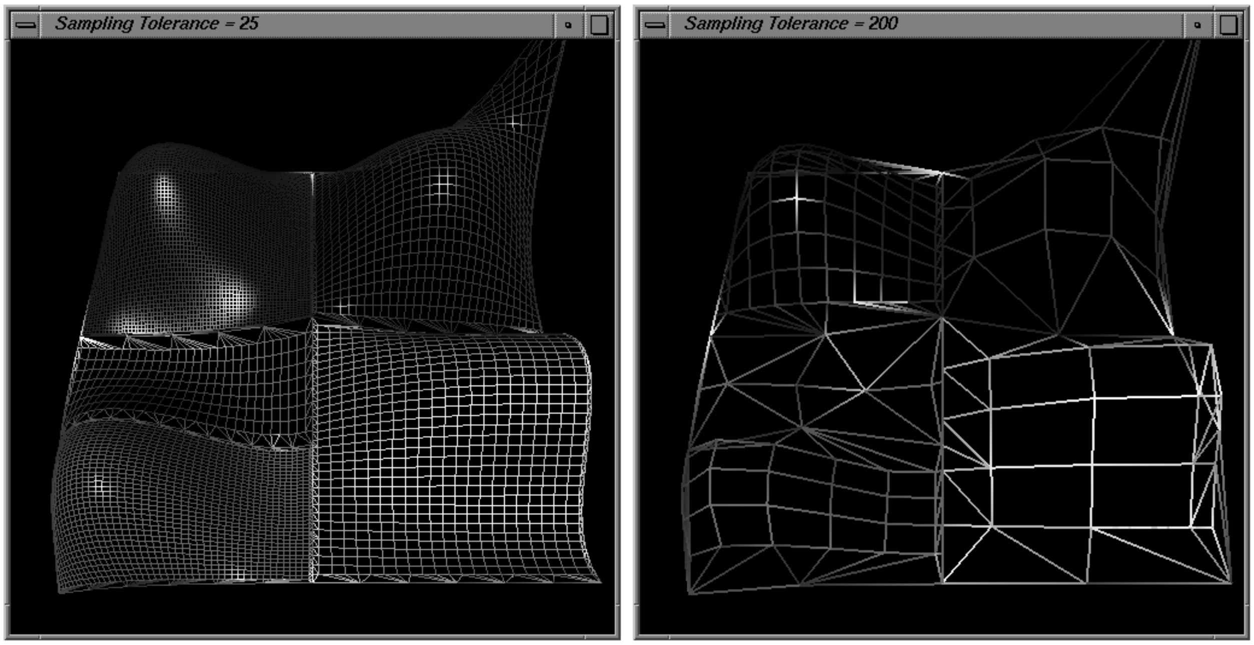

Figure 3: The sampling tolerance when tessellating a NURBS surface or curve can be adjusted to trade off accuracy of the approximation with rendering speed. Notice how the left-hand version of `molehill` is a considerably more accurate rendering of the surfaces, but it contains substantially more polygons. While less accurate, the right-hand image can be redrawn much faster than the left-hand image.

Uniform cubic B-splines. Knots are uniformly spaced and each knot is unique. For example: 0, 1, 2, 3, 4, 5, 6, 7.

Cubic Bézier curves. There are 4 control points. Knot multiplicity is 4 at the beginning and end. A knot multiplicity of 4 for the beginning and ending control points ensures that the curve must pass through the end points. For example: 0, 0, 0, 0, 1, 1, 1, 1.

Uniform cubic B-splines that passes through the endpoints. Multiplicity of 4 at the beginning and ending control points; uniformly spaced single knots between. For example: 0, 0, 0, 0, 1, 2, 3, 4, 5, 5, 5, 5.

A NURBS surface varies with two parameters (u and v), so there are two knot sequences (one for each parameter). The control points are arranged logically as a grid of points over u and v.

The phrase "non-uniform rational B-spline" denotes a set of mathematical properties of the NURBS curve or surface description:

Non-Uniform Knot spacing need not be uniform.

Rational The parametric equations describing the spline can have a polynomial denominator (that is, they can be ratios). One nice advantage of rational splines, unlike nonrationals, is that conic sections can be represented exactly.

B-Spline The influence of the control points is based on the B-spline basis function.

The `molehill.c` sample program at the end of the article demonstrates using the GLU NURBS routines to generate four different NURBS surfaces. The control points are modified differently for each NURBS surface to demonstrate how differing control points manipulate the generated surface. Lighting is enabled because good lighting helps distinguish the curvature across surfaces. The result is shown in Figure 2.

The GLU NURBS routines make it straightforward to generate NURBS curves and surfaces. First, allocate a NURBS renderer:

```
nurbsHandle = gluNewNurbsRenderer();
```

All other NURBS routines manipulate the NURBS renderer to perform NURBS operations. Either curves or surfaces can be generated. Since curves are a simplification of the procedure for rendering surfaces, the discussion that follows deals only with surfaces.

`gluBeginSurface` and `gluEndSurface` delimit a surface definition. Within a surface definition, calling `gluNurbsSurface` generates a NURBS surface. The routine is passed a set of control points, two knot sequences (for the u and v direction), and the order for the NURBS surface to be generated. Additionally, an evaluator type parameter (like `GL_MAP2_VERTEX3` or `GL_MAP2_TEXTURE_CORD2`) determines what type of per-vertex information will be generated. You always need to generate vertices, but additional calls to

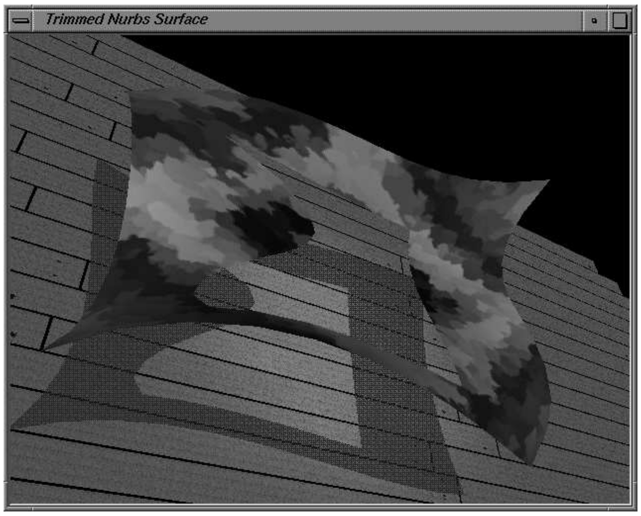

Figure 4: Example of a trimmed NURBS surface. Notice NURBS surfaces may be texture mapped.

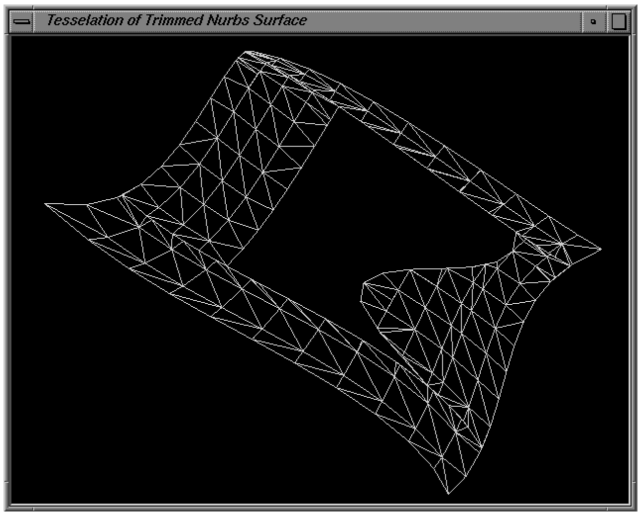

Figure 5: Wire-frame rendering of the trimmed NURBS surface to show the Tessellation into polygons.

`gluNurbsSurface` can be used to generate texture coordinates, normals, or colors.

You can change properties that control the way NURBS are rendered with `gluNurbsProperty`. For example, to change the sampling tolerance used when tessellating a NURBS surface, call:

```
gluNurbsProperty(nurbsHandle,
  GLU_SAMPLING_TOLERANCE, 25.0);
```

You can see the effect of large changes in the sampling tolerance when tessellating a NURBS surface in Figure 3.

GLU's NURBS support includes a sophisticated capability for NURBS *trimming*. Only within a NURBS surface definition (not in a curve definition), you can specify a trim curve. Conceptually, trimming permits you "cut away" areas from a NURBS surface. You can cut "holes" in a surface (see Figure 4) or create an irregular (non-rectangular) boundary for the surface. The region to trim is specified as a set of closed trimming loops specified by more GLU calls within a surface definition. Use the calls `gluBeginTrim`, `gluPwlCurve` (specifies a piecewise linear curve), `gluNurbsCurve`, and `gluEndTrim`. You can see in Figure 5 how trimming affects the resulting tessellation of the surface.

Be aware that converting NURBS descriptions to OpenGL evaluators is reasonably expensive. If your program renders the same NURBS surface repeatedly, it may be to your advantage to capture the resulting OpenGL commands generated by the GLU NURBS routines within a display list. Then, you can execute the display list to render the NURBS surface to avoid recomputing the surface on each rendering. The `molehill` example demonstrates this practice.

More Information

Most computer graphics textbooks devote a chapter to the mathematical aspects of representing curves and surfaces. There are also entire books on the subject [1, 3]. A dissertation by Derrick Burns [2] describes efficient surface trimming. To get a "hands on" feel for how NURBS surfaces operate, try changing the `molehill.c` sample program (and other OpenGL NURBS examples) to use different control points and knot multiplicities. Also, the Open Inventor toolkit, which is layered on top of OpenGL, contains an object-oriented interface to using NURBS.

Next issue's column will survey software packages (public-domain and otherwise) that let you better explore OpenGL's functionality and quickly build OpenGL applications.

References

[1] Richard Bartels, et.al., *An Introduction to Splines for use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann Publishers, 1987.

[2] Derrick Burns, *Dynamic Trimmed Surface Rendering*, Ph.D. dissertation, Stanford University, 1993.

[3] Gerald Farin, *Curves and Surfaces for Computer-Aided Geometric Design*, Academic Press, 1990.

[4] Jackie Neider, Tom Davis, Mason Woo, *OpenGL Programming Guide: The official guide to learning OpenGL, Release 1*, Addison Wesley, 1993.

[5] Josie Wernecke, *The Inventor Mentor*, Addison Wesley, 1993.

A molehill.c

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

GLfloat mat_red_diffuse[] = { 0.7, 0.0, 0.1, 1.0 };
GLfloat mat_green_diffuse[] = { 0.0, 0.7, 0.1, 1.0 };
GLfloat mat_blue_diffuse[] = { 0.0, 0.1, 0.7, 1.0 };
GLfloat mat_yellow_diffuse[] = { 0.7, 0.8, 0.1, 1.0 };
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat mat_shininess[] = { 100.0 };
GLfloat knots[8] = { 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0 };
GLfloat pts1[4][4][3], pts2[4][4][3];
GLfloat pts3[4][4][3], pts4[4][4][3];
GLUnurbsObj *nurb;
int u, v;

static void
display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  /* Execute display list containing all the NURBS surfaces. */
  glCallList(1);
  glFlush();
}

void
main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutCreateWindow("molehill");
  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);
  nurb = gluNewNurbsRenderer();
  gluNurbsProperty(nurb, GLU_SAMPLING_TOLERANCE, 25.0);
  gluNurbsProperty(nurb, GLU_DISPLAY_MODE, GLU_FILL);

  /* Build control points for NURBS mole hills. */
  for (u=0; u<4; u++) {
    for (v=0; v<4; v++) {
      /* Red. */
      pts1[u][v][0] = 2.0*((GLfloat)u);
      pts1[u][v][1] = 2.0*((GLfloat)v);
      if ((u==1 || u == 2) && (v == 1 || v == 2))
        /* Stretch up middle. */
        pts1[u][v][2] = 6.0;
      else
        pts1[u][v][2] = 0.0;

      /* Green. */
      pts2[u][v][0] = 2.0*((GLfloat)u - 3.0);
      pts2[u][v][1] = 2.0*((GLfloat)v - 3.0);
      if ((u==1 || u == 2) && (v == 1 || v == 2))
        if (u == 1 && v == 1)
          /* Pull hard on single middle square. */
          pts2[u][v][2] = 15.0;
```

```
      else
        /* Push down on other middle squares. */
        pts2[u][v][2] = -2.0;
    else
      pts2[u][v][2] = 0.0;

    /* Blue. */
    pts3[u][v][0] = 2.0*((GLfloat)u - 3.0);
    pts3[u][v][1] = 2.0*((GLfloat)v);
    if ((u==1 || u == 2) && (v == 1 || v == 2))
      if (u == 1 && v == 2)
        /* Pull up on single middple square. */
        pts3[u][v][2] = 11.0;
      else
        /* Pull up slightly on other middle squares. */
        pts3[u][v][2] = 2.0;
    else
      pts3[u][v][2] = 0.0;

    /* Yellow. */
    pts4[u][v][0] = 2.0*((GLfloat)u);
    pts4[u][v][1] = 2.0*((GLfloat)v - 3.0);
    if ((u==1 || u == 2 || u == 3) && (v == 1 || v == 2))
      if (v == 1)
        /* Push down front middle and right squares. */
        pts4[u][v][2] = -2.0;
      else
        /* Pull up back middle and right squares. */
        pts4[u][v][2] = 5.0;
    else
      pts4[u][v][2] = 0.0;
  }
}
/* Stretch up red's far right corner. */
pts1[3][3][2] = 6;
/* Pull down green's near left corner a little. */
pts2[0][0][2] = -2;
/* Turn up meeting of four corners. */
pts1[0][0][2] = 1;
pts2[3][3][2] = 1;
pts3[3][0][2] = 1;
pts4[0][3][2] = 1;

glMatrixMode(GL_PROJECTION);
gluPerspective(55.0, 1.0, 2.0, 24.0);
glMatrixMode(GL_MODELVIEW);
glTranslatef(0.0, 0.0, -15.0);
glRotatef(330.0, 1.0, 0.0, 0.0);

/* Collect the generated OpenGL commands for NURBS surfaces
   into a display list. */
glNewList(1, GL_COMPILE);
  /* Render red hill. */
  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_red_diffuse);
  gluBeginSurface(nurb);
    gluNurbsSurface(nurb, 8, knots, 8, knots,
      4 * 3, 3, &pts1[0][0][0],
      4, 4, GL_MAP2_VERTEX_3);
  gluEndSurface(nurb);
```

```
    /* Render green hill. */
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_green_diffuse);
    gluBeginSurface(nurb);
      gluNurbsSurface(nurb, 8, knots, 8, knots,
        4 * 3, 3, &pts2[0][0][0],
        4, 4, GL_MAP2_VERTEX_3);
    gluEndSurface(nurb);

    /* Render blue hill. */
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_blue_diffuse);
    gluBeginSurface(nurb);
      gluNurbsSurface(nurb, 8, knots, 8, knots,
        4 * 3, 3, &pts3[0][0][0],
        4, 4, GL_MAP2_VERTEX_3);
    gluEndSurface(nurb);

    /* Render yellow hill. */
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_yellow_diffuse);
    gluBeginSurface(nurb);
      gluNurbsSurface(nurb, 8, knots, 8, knots,
        4 * 3, 3, &pts4[0][0][0],
        4, 4, GL_MAP2_VERTEX_3);
    gluEndSurface(nurb);
  glEndList();

  glutDisplayFunc(display);
  glutMainLoop();
}
```