Curves and Surfaces in OpenGL

Ed Angel
Professor of Computer Science,
Electrical and Computer Engineering, and Media Arts
University of New Mexico
Objectives

• Introduce OpenGL evaluators
• Learn to render polynomial curves and surfaces
• Discuss quadrics in OpenGL
  - GLUT Quadrics
  - GLU Quadrics
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

```c
glMap1f(type, u_min, u_max, stride, order, pointer_to_array)
```

- what we want to evaluate
- max and min of \( u \)
- 1+degree of polynomial
- separation between data points
- pointer to control data

Each type must be enabled by `glEnable(type)`
Example

Consider an evaluator for a cubic Bezier curve over (0,1)

```c
point data[ ]={ ............. } ; * /3d data /*
glMap1f(GL_MAP_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

```c
glEnable(GL_MAP_VERTEX_3);
```
Evaluating

- The function \( \text{glEvalCoord1f}(u) \) causes all enabled evaluators to be evaluated for the specified \( u \)
  - Can replace \( \text{glVertex}, \text{glNormal}, \text{glTexCoord} \)
- 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

  ```c
  glBegin(GL_LINE_STRIP)
  for(i=0; i<100; i++)
    glVertex1f( (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

\[
\text{glMapGrid}(100, 0.0, 1.0);
\]

sets up 100 equally-spaced points on (0,1)

\[
\text{glEvalMesh1(GL_LINE, 0, 99)};
\]

renders lines between adjacent evaluated points from point 0 to point 99
Beziers Surfaces

- Similar procedure to 1D but use 2D evaluators in \( u \) and \( v \)
- Set up with

\[
\text{glMap2f}(\text{type}, \text{u}_\text{min}, \text{umax}, \text{u}_\text{stride}, \text{u}_\text{order}, \text{v}_\text{min}, \text{v}_\text{max}, \text{v}_\text{stride}, \text{v}_\text{order}, \text{pointer}\_\text{to}\_\text{data})
\]

- Evaluate with \( \text{glEvalCoord2f}(u,v) \)
Example

bicubic over \((0,1) \times (0,1)\)

```c
point data[4][4] = {………};
glMap2f(GL_MAP_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.
must draw in both directions

```c
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();
}
```
We can form a quad mesh and render with lines

```c
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 latitude
- GLU: disks, cylinders, spheres
  - Can apply transformations to scale, orient, and position
- GLUT: Platonic solids, torus, Utah teapot, cone
GLUT Objects

Each has a wire and a solid form

- `glutWireCone()`
- `glutWireTorus()`
- `glutWireTeapot()`
GLUT Platonic Solids

- `glutWireTetrahedron()`
- `glutWireOctahedron()`
- `glutWireDodecahedron()`
- `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
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);