Evaluators and Spline Surfaces

Prerequisites

A modest understanding of parametric functions of two variables together with an understanding of modeling by stepping through the polygons in a surface with techniques such as triangle strips.

Introduction

In the discussions of mathematical fundamentals at the beginning of these notes, we talked about linear, quadratic, and cubic interpolations of points. These are special cases of the interpolation technique called splines, and are straightforward but are not very general and are fairly complex to extend to two dimensions. In general, splines are tools that provide a much broader approach to creating smooth approximations across a line segment (1D interpolation) or a rectangular domain (2D interpolation). This interpolation is usually thought of as a way to derive geometric modeling, but there are other uses we will mention later. Graphics APIs such as OpenGL usually provide tools that allow a graphics programmer to create spline interpolations given only the original set of points, called control points, that are to be interpolated.

In OpenGL, the spline capability is provided by techniques called evaluators, functions that take a set of control points and produce another set of points that interpolate the original control points. This allows you to model curves and surfaces by doing only a modest amount of work to set the control points, and then to get much more detailed curves and surfaces as a result. (Note that interpolation can be used for much more general work, but we will focus mostly on the geometric applications in this section.) You have already seen an example of spline surfaces in the Éadington example for object selection in the general discussion of selection.

There are two kinds of evaluators available to you. If you want to interpolate points to produce one-parameter information (that is, curves or any other data with only one degree of freedom; think 1D textures as well as geometric curves), you can use 1D evaluators. If you want to interpolate points in a 2D array to produce two-parameter information (that is, surfaces or any other data with two degrees of freedom; think 2D textures as well as geometric curves) you can use 2D evaluators. Both are straightforward and allow you to choose how much detail you want in the actual display of the information.

Figure 16.1: a spline curve defined via a 1D evaluator, shown as originally presented (left) and as rotated to show the relationship between control points and the curve shape (right). The cyan control points are the originals; the green control points are added as discussed below.
Figure 16.2: spline surfaces defined via a 2D evaluator. At top, in two views, a single patch defined by four control points; at bottom, a larger surface defined by extending the 16x16 set of control points with interpolated points as defined below.

In Figures 16.1 and 16.2 above we see several images that illustrate the use of evaluators to define geometry in OpenGL. Figure 16.1 shows two views of a 1D evaluator that is used to define a curve in space showing the set of 30 control points as well as additional computed control points for smoothness; Figure 16.2 shows two uses of a 2D evaluator to define surfaces, with the top row showing a surface defined by a 4x4 set of control points and the bottom image showing a surface defined by a 16x16 set of control points with additional smoothness points not shown. These images and the techniques for creating smooth curves will be discussed further below, and some of the code that creates these is given in the Examples section.

The spline surface in the top row of Figure 16.2 has only a 0.7 alpha value so the control points and other parts of the surface can be seen behind the primary surface of the patch. In this example, note the relation between the control points and the actual surface; only the four corner points actually meet the surface, while all the others lie off the surface and act only to influence the shape of the patch. Note also that the entire patch lies within the convex hull of the control points. The specular highlight on the surface should also help you see the shape of the patch from the lighting. In the larger surface at the bottom of Figure 16.2, note how the surface extends smoothly between the different sets of control points.

Definitions

As you see in Figures 16.1 and 16.2, an evaluator working on an array of four control points (1D) or 4x4 control points (2D) actually fits the extreme points of the control point set but does not go through any of the other points. As the evaluator comes up to these extreme control points, the tangent to the curve becomes parallel to the line segment from the extreme point to the adjacent control point, as shown in the figure below, and the speed with which this happens is determined by the distance between the extreme and adjacent control points.
Figure: two spline curves that illustrate the shape of the curve as it goes through an extreme control point

To control the shape of an extended spline curve, you need to arrange the control points so that the direction and distance from a control point to the adjacent control points are the same. This can be accomplished by adding new control points between appropriate pairs of the original control points as indicated in the spline curve figure above. This will move the curve from the first extreme point to the first added point, from the first added point smoothly to the second added point, from the second added point smoothly to the third added point, and so on to moving smoothly through the last added point to the last extreme point.

This construction and relationship is indicated by the green (added) control points in the first figure in this section. Review that figure and note again how there is one added point after each two original points, excepting the first and last points; that the added points bisect the line segment between the two points they interpolate; and that the curve actually only meets the added points, not the original points, again excepting the two end points. If we were to define an interactive program to allow a user to manipulate control points, we would only give the user access to the original control points; the added points are not part of the definition but only of the implementation of a smooth surface.

Similarly, one can define added control points in the control mesh for a 2D evaluator, creating a richer set of patches with the transition from one patch to another following the same principle of equal length and same direction in the line segments coming to the edge of one patch and going from the edge of the other. This allows you to achieve a surface that moves smoothly from one patch to the next. Key points of this code are included in the example section below, but it does take some effort to manage all the cases that depend on the location of a particular patch in the surface. The example code in the file fullSurface.c included with this material will show you these details.

So how does this all work? A cubic spline curve is determined by a cubic polynomial in a parametric variable $u$ as indicated by the first equation in (1) below, with the parameter taking values between 0 and 1. The four coefficients $a_i$ can be determined by knowing four constraints on the curve. These are provided by the four control points needed to determine a single segment of a cubic spline curve, and an OpenGL 1D evaluator provides those coefficients and, as needed, evaluates the resulting polynomial to generate a point on the curve or the curve itself. A bicubic spline surface is determined by a bicubic polynomial in parametric variables $u$ and $v$ as indicated by the second equation in (1) below, with both parameters taking values between 0 and 1. This requires computing the 16 coefficients $a_{ij}$ which can be done by using the 16 control points for a single bicubic spline patch. Again, an OpenGL 2D evaluator provides those 16 coefficients and does the appropriate evaluations as needed.

\[
\begin{align*}
(1) & \quad \sum_{i=0}^{3} a_i u^i \\
& \quad \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} u^i v^j
\end{align*}
\]
Some examples

Spline curves: the setup to generate curves is given in some detail below. This involves defining a set of control points for the evaluator to use, enabling the evaluator for your target data type, defining overall control points for the curve, stepping through the overall control points to build four-tuples of segment control points, and then invoking the evaluator to draw the actual curve. This code produced the figures shown in the figure above on spline curves. A few details have been omitted in the code below, but they are all in the sample code `splineCurve.c` that is included with this module. Note that this code returns the points on the curve using the `glEvalCoord1f(...)` function instead of the `glVertex*(...)` function within a `glBegin(...)` ... `glEnd()` pair; this is different from the more automatic approach of the 2D patch example that follows it.

Probably the key point in this sample code is the way the four-tuples of segment control points have been managed. The original points would not have given smooth curves, so as discussed above, new points were defined that interpolated some of the original points to make the transition from one segment to the other continuous and smooth.

```
// enable GL_MAP2_VERTEX_3

glEnable( GL_MAP1_VERTEX_3)

void makeCurve( void )
{

  ...  
  for (i=0; i<CURVE_SIZE; i++) {
    ctrlpts[i][0]= RAD*cos(INITANGLE + i*STEPANGLE);
    ctrlpts[i][1]= RAD*sin(INITANGLE + i*STEPANGLE);
    ctrlpts[i][2]= -4.0 + i * 0.25;
  }

}

// NPTS refers to the number of data points to be drawn

#define NPTS 30

void curve(void) {

#define LAST_STEP (CURVE_SIZE/2)-1

  int step, i, j;

  makeCurve(); // calculate the control points for the entire curve
  // copy/compute points from ctrlpts to segpts to define each segment
  // of the curve. First/last cases are different from middle cases...
  for ( step = 0; step < LAST_STEP; step++ ) {
    if (step==0) { // first case
      for (j=0; j<3; j++) {
        segpts[0][j]=ctrlpts[0][j];
        segpts[1][j]=ctrlpts[1][j];
        segpts[2][j]=ctrlpts[2][j];
        segpts[3][j]=(ctrlpts[2][j]+ctrlpts[3][j])/2.0;
      }
    } else if (step==LAST_STEP-1) { // last case
      for (j=0; j<3; j++) {
        segpts[0][j]=(ctrlpts[CURVE_SIZE-4][j] +
                      ctrlpts[CURVE_SIZE-3][j])/2.0;
        segpts[1][j]=ctrlpts[CURVE_SIZE-3][j];
        segpts[2][j]=ctrlpts[CURVE_SIZE-2][j];
        segpts[3][j]=ctrlpts[CURVE_SIZE-1][j];
      }
    }
  }
}
```
else for (j=0; j<3; j++) { // general case
    segpts[0][j]=(ctrlpts[2*step][j]+ctrlpts[2*step+1][j])/2.0;
    segpts[1][j]=ctrlpts[2*step+1][j];
    segpts[2][j]=ctrlpts[2*step+2][j];
    segpts[3][j]=(ctrlpts[2*step+2][j]+ctrlpts[2*step+3][j])/2.0;
}

// define the evaluator
glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &segpts[0][0]);
glBegin(GL_LINE_STRIP);
for (i=0; i<=NPTS; i++)
    glEvalCoord1f( (GLfloat)i/(GLfloat)NPTS );
glEnd();
...
}
}

Spline surfaces: we have two examples, the first showing drawing a simple patch (surface based on a 4x4 grid of control points) and the second showing drawing of a larger surface with more control points. Below is some simple code to generate a surface given a 4x4 array of points for a single patch, as shown in the top row of the second figure above. This code initializes a 4x4 array of points, enables auto normals (available through the glEvalMesh(...) function) and identifies the target of the evaluator, and carries out the evaluator operations. The data for the patch control points is deliberately over-simplified so you can see this easily, but in general the patch points act in a parametric way that is quite distinct from the indices, as is shown in the general surface code.

    point3 patch[4][4] = {{{-2.,-2.,0.},{-2.,-1.,1.},{-2.,1.,1.},{-2.,2.,0.}},
                        {{-1.,-2.,1.},{-1.,-1.,2.},{-1.,1.,2.},{-1.,2.,1.}},
                        {{1.,-2.,1.},{1.,-1.,2.},{1.,1.,2.},{1.,2.,1.}},
                        {{2.,-2.,0.},{2.,-1.,1.},{2.,1.,1.},{2.,2.,0.}}};

void myinit(void)
{
    ...
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_MAP2_VERTEX_3);
}

void doPatch(void)
{
    // draws a patch defined by a 4 x 4 array of points
    #define NUM 20 //
    glMaterialfv(...); // whatever material definitions are needed
    glMap2f(GL_MAP2_VERTEX_3,0.0,1.0,3,4,0.0,1.0,12,4,&patch[0][0][0]);
    glMapGrid2f(NUM, 0.0, 1.0, NUM, 0.0, 1.0);
    glEvalMesh2(GL_FILL, 0, NUM, 0, NUM);
}

The considerations for creating a complete surface with a 2D evaluator is similar to that for creating a curve with a 1D evaluator. You need to create a set of control points, to define and enable an appropriate 2D evaluator, to generate patches from the control points, and to draw the individual patches. These are covered in the sample code below.
The sample code below has two parts. The first is a function that generates a 2D set of control points procedurally; this differs from the manual definition of the points in the patch example above or in the pool example of the selection section. This kind of procedural control point generation is a useful tool for procedural surface generation. The second is a fragment from the section of code that generates a patch from the control points, illustrating how the new intermediate points between control points are built. Note that these intermediate points all have indices 0 or 3 for their locations in the patch array because they are the boundary points in the patch; the interior points are always the original control points. Drawing the actual patch is handled in just the same way as it is handled for the patch example, so it is omitted here.

```c
// control point array for pool surface
point3 ctrlpts[GRIDSIZE][GRIDSIZE];

void genPoints(void)
{
  #define PI 3.14159
  #define R1 6.0
  #define R2 3.0
  int i, j;
  GLfloat alpha, beta, step;

  alpha = -PI;
  step = PI/(GLfloat)(GRIDSIZE-1);
  for (i=0; i<GRIDSIZE; i++) {
    beta = -PI;
    for (j=0; j<GRIDSIZE; j++) {
      ctrlpts[i][j][0] = (R1 + R2*cos(beta))*cos(alpha);
      ctrlpts[i][j][1] = (R1 + R2*cos(beta))*sin(alpha);
      ctrlpts[i][j][2] = R2*sin(beta);
      beta -= step;
    }
    alpha += step;
  }
}

void surface(point3 ctrlpts[GRIDSIZE][GRIDSIZE])
{
  ...

  ...( // general case (internal patch)
    for(i=1; i<3; i++)
      for(j=1; j<3; j++)
        for(k=0; k<3; k++) {
          patch[i][j][k]=ctrlpts[2*xstep+i][2*ystep+j][k];
        }
    for(i=1; i<3; i++)
      for(k=0; k<3; k++) {
        patch[i][0][k]=(ctrlpts[2*xstep+i][2*ystep][k]
            +ctrlpts[2*xstep+i][2*ystep+1][k])/2.0;
        patch[i][3][k]=(ctrlpts[2*xstep+i][2*ystep+2][k]
            +ctrlpts[2*xstep+i][2*ystep+3][k])/2.0;
        patch[0][i][k]=(ctrlpts[2*xstep][2*ystep+i][k]
            +ctrlpts[2*xstep+1][2*ystep+i][k])/2.0;
        patch[3][i][k]=(ctrlpts[2*xstep+2][2*ystep+i][k]
            +ctrlpts[2*xstep+3][2*ystep+i][k])/2.0;
      }
    for(k=0; k<3; k++) {
      patch[0][0][k]=(ctrlpts[2*xstep][2*ystep][k]
          +ctrlpts[2*xstep+1][2*ystep][k]
          +ctrlpts[2*xstep][2*ystep+1][k]
```
A word to the wise...

Spline techniques may also be used for much more than simply modeling. Using them, you can generate smoothly changing sets of colors, or of normals, or of texture coordinates — or probably any other kind of data that one could interpolate. There aren’t built-in functions that allow you to apply these points automatically as there are for creating curves and surfaces, however. For these you will need to manage the parametric functions yourself. To do this, you need to define each point in the (u,v) parameter space for which you need a value and get the actual interpolated points from the evaluator using the functions glEvalCoord1f(u) or glEvalCoord2f(u,v), and then use the points in the same way you would use points you had defined in another way. This point, then, may be a color, a normal, or a texture coordinate as you need in your image.

Another example of spline use is in animation, where you can get a smooth curve for your eyepoint to follow by using splines. As your eyepoint moves, however, you also need to deal with the other issues in defining a view. The up vector is fairly straightforward; for simple animations, it is probably enough to keep the up vector constant. The center of view is more of a challenge, however, because it has to move to keep the motion realistic. The suggested approach is to keep three points from the spline curve: the previous point, the current point, and the next point, and to use the previous and next points to set the direction of view; the viewpoint is then a point a fixed distance from the current point in the direction set by the previous and next points. This should provide a reasonably good motion and viewing setup.

This discussion has only covered cubic and bicubic splines, because these are readily provided by OpenGL evaluators. OpenGL also has the capability of providing NURBS (non-uniform rational B-splines) but these are beyond the scope of this discussion. Other applications may find it more appropriate to use other kinds of splines, and there are many kinds of spline curves and surfaces available; the interested reader is encouraged to look into this subject further.

Code examples

- **splineCurve.c** — code that draws the spline-based spiral display in the first figure
- **splineSurf4.c** — code that draws a single patch with a 2D evaluator as in the second figure.
- **fullSurface.c** — code that draws a full surface from a 16x16 set of control points and includes all the details of creating added points for patches so they will blend smoothly with each other.