Interpolation and Spline Modeling

Prerequisites

A modest understanding of parametric functions of two variables together with an understanding of simple modeling with techniques such as triangle strips.

Introduction

In the discussions of mathematical fundamentals at the beginning of these notes, we talked about line segments as linear interpolations of points. Here we introduce other kinds of interpolations of points involving techniques called spline curves and surfaces. The specific spline techniques we will discuss are straightforward but we will limit them to one-dimensional spline curves. Extending these to two dimensions to model surfaces is a bit more complex and we will only cover this in terms of the evaluators that are built into the OpenGL graphics API. In general, spline techniques provide a very broad approach to creating smooth curves that approximate a number of points in a one-dimensional domain (1D interpolation) or smooth surfaces that approximate a number of points in a two-dimensional domain (2D interpolation). This interpolation is usually thought of as a way to develop geometric models, but there are a number of other uses of splines that 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 general, we think of an entire spline curve or spline surface as a single piece of geometry in the scene graph. These curves and surfaces are defined in a single modeling space and usually have a single set of appearance parameters, so in spite of their complexity they are naturally represented by a single shape node that is a leaf in the scene graph.

Interpolations

When we talked about the parametric form for a line segment in the early chapter on mathematical foundations for graphics, we created a correspondence between the unit line segment and an arbitrary line segment and were really interpolating between the two points by creating a line segment between them. If the points are named \( P_0 \) and \( P_1 \), this interpolating line segment can be expressed in terms of the parametric form of the segment:

\[
(1-t)*P_0 + t*P_1, \text{ for } t \in [0,1] 
\]

This form is almost trivial to use, and yet it is quite suggestive, because it hints that the set of points that interpolate the two given points can be computed by an expression such as

\[
f_0(t)*P_0 + f_1(t)*P_1 
\]

for two fixed functions \( f_0 \) and \( f_1 \). This suggests a relationship between points and functions that interpolate them that would allow us to consider the nature of the functions and the kind of interpolations they provide. In this example, we have \( f_0(t) = (1-t) \) and \( f_1(t) = t \), and there are interesting properties of these functions. We see that \( f_0(0) = 1 \) and \( f_1(0) = 0 \), so at \( t = 0 \), the interpolant value is \( P_0 \), while \( f_0(1) = 0 \) and \( f_1(1) = 1 \), so at \( t = 1 \), the interpolant value is \( P_1 \). This tells us that the interpolation starts at \( P_0 \) and ends at \( P_1 \), which we had already found to be a useful property for the interpolating line segment. Note that because each of the interpolating functions is linear in the parameter \( t \), the set of interpolating points forms a line.

As we move beyond line segments that interpolate two points, we want to use the term interpolation to mean determining a set of points that approximate the space between a set of given points in the order the points are given. This set of points can include three points, four points, or even more. We assume throughout this discussion that the points are in 3-space, so we will be
creating interpolating curves (and later on, interpolating surfaces) in three dimensions. If you want
to do two-dimensional interpolations, simply ignore one of the three coordinates.

Finding a way to interpolate three points \( P_0, P_1, \) and \( P_2 \) is more interesting than interpolating only
two points, because one can imagine many ways to do this. However, extending the concept of
the parametric line we could consider a quadratic interpolation in \( t \) as:

\[
(1-t)^2 * P_0 + 2t * (1-t) * P_1 + t^2 * P_2, \text{ for } t \in [0.,1.]
\]

Here we have three functions \( f_0, f_1, \) and \( f_2 \) that participate in the interpolation, with

\[
\begin{align*}
 f_0(t) &= (1-t)^2, \\
 f_1(t) &= 2t * (1-t), \\
 f_2(t) &= t^2. 
\end{align*}
\]

These functions have by now achieved enough importance in our thinking that we will give them a name, and call them the \textit{basis functions} for the interpolation. Further, we will call the points \( P_0, P_1, \) and \( P_2 \) the \textit{control points}
for the interpolation (although the formal literature on spline curves calls them \textit{knots} and calls the
endpoints of an interpolation \textit{joints}). This particular set of functions have a similar property to the
linear basis functions above, with

\[
\begin{align*}
 f_0(0) &= 1, \\
 f_1(0) &= 0, \\
 f_2(0) &= 0, \\
 f_0(1) &= 0, \\
 f_1(1) &= 0, \\
 f_2(1) &= 1, 
\end{align*}
\]

giving us a smooth quadratic interpolating function in \( t \) that has value

\( P_0 \) if \( t=0 \) and value \( P_1 \) if \( t=1 \), and that is a linear combination of the three points if \( t=.5 \). The
shape of this interpolating curve is shown in Figure 15.1.

![Figure 15.1: a quadratic interpolating curve for three points](image)

The particular set of interpolating polynomials \( f_0, f_1, \) and \( f_2 \) in the interpolation of three points is
suggestive of a general approach in which we would use components which are products of these
polynomials and take their coefficients from the geometry we want to interpolate. If we follow this
pattern, interpolating four points \( P_0, P_1, P_2, \) and \( P_3 \) would look like:

\[
(1-t)^3 * P_0 + 3t * (1-t)^2 * P_1 + 3t^2 * (1-t) * P_2 + t^3 * P_3, \text{ for } t \in [0.,1.]
\]

and the shape of the curve this determines is illustrated in Figure 15.2. (We have chosen the first
three of these points to be the same as the three points in the quadratic spline above to make it
easier to compare the shapes of the curves). In fact, this curve is an expression of the standard
Bézier spline function to interpolate four control points, and the four polynomials

\[
\begin{align*}
 f_0(t) &= (1-t)^3, \\
 f_1(t) &= 3t (1-t)^2, \\
 f_2(t) &= 3t^2 (1-t), \\
 f_3(t) &= t^3, 
\end{align*}
\]
\[ f_2(t) = 3t^2 (1-t), \text{ and} \]
\[ f_3(t) = t^3 \]
are called the cubic Bernstein basis for the spline curve.

Figure 15.2: interpolating four points with the Bézier spline based on the Bernstein basis functions

When you consider this interpolation, you will note that the interpolating curve goes through the first and last control points (\(P_0\) and \(P_3\)) but does not go through the other two control points. This is because the set of basis functions for this curve behaves the same at the points where \(t=0\) and \(t=1\) as we saw in the quadratic spline: \(f_0(0)=1, f_1(0)=0, f_2(0)=0, \) and \(f_3(0)=0,\) as well as \(f_0(1)=0, f_1(1)=0, f_2(1)=0, \) and \(f_3(1)=1.\) You will also note that as the curve goes through the first and last control points, it is moving in the direction from the first to the second control point, and from the third to the fourth control points. Thus the two control points that are not met control the shape of the curve by determining the initial and the ending directions of the curve, and the rest of the shape is determined in order to get the necessary smoothness.

In general, curves that interpolate a given set of points need not go through those points, but the points influence and determine the nature of the curve in other ways. If you need to have the curve actually go through the control points, however, there are spline formulations for which this does happen. The Catmull-Rom cubic spline has the form
\[ f_0(t) \times P_0 + f_1(t) \times P_1 + f_2(t) \times P_2 + f_3(t) \times P_3, \text{ for } t \text{ in } [0, 1]. \]
for basis functions
\[ f_0(t) = (-t^3 + 2t^2 - t)/2 \]
\[ f_1(t) = (3t^3 - 5t^2 + 2)/2 \]
\[ f_2(t) = (-3t^3 + 4t^2 + t)/2 \]
\[ f_3(t) = (t^3 - t^2)/2 \]
This interpolating curve has a very different behavior from that of the Bézier curve above, because as shown in Figure 15.3. This is a different kind of interpolating behavior that is the result of a set of basis functions that have \(f_0(0)=0, f_1(0)=1, f_2(0)=0, \) and \(f_3(0)=0,\) as well as \(f_0(1)=0, f_1(1)=0, f_2(1)=1, \) and \(f_3(1)=0.\) This means that the curve interpolates the points \(P_1\) and \(P_2\) instead of \(P_0\) and \(P_3\) and actually goes through those two points. Thus the
Catmull-Rom spline curve is useful when you want your interpolated curve to include all the control points, not just some of them.

![Figure 15.3: interpolating four points with the Catmull-Rom cubic spline](image)

We will not carry the idea of spline curves beyond cubic interpolations, but we want to provide this much detailed background because it can sometimes be handy to manage cubic spline curves ourselves, even though OpenGL provides evaluators that can make spline computations easier and more efficient. Note that if the points we are interpolating lie in 3D space, each of these techniques provides a 3D curve, that is, a function from a line segment to 3D space.

While we have only shown the effect of these interpolations in the smallest possible set of points, it is straightforward to extend the interpolations to larger sets of points. The way we do this will depend on the kind of interpolation that is provided by the particular curve we are working with, however.

In the Bézier curve, we see that the curve meets the first and last control points but not the two intermediate control points. If we simply use the first four control points, then the next three (the last point of the original set plus the next three control points), and so on, then we will have a curve that is continuous, goes through every third control point (first, fourth, seventh, and so on), but that changes direction abruptly at each of the control points it meets. In order to extend these curves so that they progress smoothly along their entire length, we will need to add new control points that maintain the property that the direction into the last control point of a set is the same as the direction out of the first control point of the next set. In order to do this, we need to define new control points between each pair of points whose index is 2N and 2N+1 for N≥1 up to, but not including, the last pair of control points. We can define these new control points as the midpoint between these points, or \((P_{2N}+P_{2N+1})/2\). When we do, we get the following relation between the new and the original control point set:

original:  \(P_0 \quad P_1 \quad P_2 \quad P_3 \quad P_4 \quad P_5 \quad P_6 \quad P_7\)

new:  \(P_0 \quad P_1 \quad P_2 \quad Q_0 \quad P_3 \quad P_4 \quad Q_1 \quad P_5 \quad P_6 \quad P_7\)

where each point \(Q\) represents a new point calculated as an average of the two on each side of it, as above. Then the computations would use the following sequences of points: \(P_0-P_1-P_2-Q_0\); \(Q_0-P_3-P_4-Q_1\); and \(Q_1-P_5-P_6-P_7\). Note that we must have an even number of control points for a Bézier curve, that we only need to extend the original control points if we have at least six control points, and that we always have three of the original points participating in each of the first and last segments of the curve.
Figure 15.4: extending the Bézier curve by adding intermediate control points, shown in green

For the Catmull-Rom cubic spline, the fact that the interpolating curve only connects the control points P1 and P2 gives us a different kind of approach to extending the curve. However, it also gives us a challenge in starting the curve, because neither the starting control point P0 nor the ending control point P3 is not included in the curve that interpolates P0-P3. Hence we will need to think of the overall interpolation problem in three parts: the first segment, the intermediate segments, and the last segment.

For the first segment, the answer is simple: repeat the starting point twice. This gives us a first set of control points consisting of P0, P0, P1, and P2, and the first piece of the curve will then interpolate P0 and P1 as the middle points of these four. In the same way, to end the curve we would repeat the ending point, giving us the four control points P1, P2, P3, and P3, so the curve would interpolate the middle points, P2 and P3. If we only consider the first four control points and add this technique, we see the three-segment interpolation of the points shown in the left-hand image of Figure 15.5.

If we have a larger set of control points, and if we wish to extend the curve to cover the total set of points, we can consider a “sliding set” of control points that starts with P0, P1, P2, and P3 and, as we move along, includes the last three control points from the previous segment as the first three of the next set and adds the next control point as the last point of the set of four points. That is, the second set of points would be P1, P2, P3, and P4, and the one after that P2, P3, P4, and P5, and so on. This kind of sliding set is simple to implement (just take an array of four points, move each one down by one index so P[1] becomes P[0], P[2] becomes P[1], P[3] becomes P[2], and the new point becomes P[3]. The sequence of points used for the individual segments of the curve are then P0-P0-P1-P2; P0-P1-P2-P3; P1-P2-P3-P4; P2-P3-P4-P5; P3-P4-P5-P6; P4-P5-P6-P7; P5-P6-P7-P8; and P6-P7-P8-P8. The curve that results when we extend the computation across a larger set of control points is shown as the right-hand image of Figure 15.5, where we have taken the same set of control points that we used for the extended Bézier spline example.
Interpolations in OpenGL

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 the work to set the control points and set up the evaluator, and then to get much more detailed curves and surfaces as a result. There is an excellent example of spline surfaces in the Eadington example for selecting and manipulating control points in the chapter of these notes on object 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.

In Figures 15.6 and 15.7 below we see several images that illustrate the use of evaluators to define geometry in OpenGL. Figure 15.6 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 15.7 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.
Figure 15.6: a spline curve defined via a 1D evaluator, shown from a point of view with $x = y = z$ (left) and 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 above.

Figure 15.7: 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.

The spline surface in the top row of Figure 15.7 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 15.7, note how the surface extends smoothly between the different sets of control points.
Definitions

As you see in Figures 15.6 and 15.7, an OpenGL 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 Figure 15.8 below, and the speed with which this happens is determined by the distance between the extreme and adjacent control points.

![Figure 15.8: two spline curves that illustrate the shape of the curve as it goes through an extreme control point](image)

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 left-hand equation in (1) below, with the single parameter $u$ 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. We saw ways that these four values could be
represented in terms of the values of four basis polynomials, and an OpenGL 1D evaluator
computes those four coefficients based on the Bézier curve definition 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 right-
hand 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 that define a
single bicubic spline patch. Again, an OpenGL 2D evaluator takes the control points, determines
those 16 coefficients based on the basis functions from the Bézier process, and evaluates the
function as you specify to create your surface model.

\[
\sum_{i=0}^{3} a_i u^i \quad \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} u^i v^j
\]

(1)

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.

```c
#include <stdlib.h>
#include <GL/gl.h>

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;
    }
}

void curve(void) {
#define LAST_STEP (CURVE_SIZE/2)-1
#define NPTS 30

    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.

    // 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]
                   +ctrlpts[2*xstep+1][2*ystep+1][k])/4.0;
    patch[3][0][k]=(ctrlpts[2*xstep+2][2*ystep][k]
                   +ctrlpts[2*xstep+3][2*ystep][k]
                   +ctrlpts[2*xstep+2][2*ystep+1][k]
                   +ctrlpts[2*xstep+3][2*ystep+1][k])/4.0;
    patch[0][3][k]=(ctrlpts[2*xstep][2*ystep+2][k]
                   +ctrlpts[2*xstep+1][2*ystep+2][k]
                   +ctrlpts[2*xstep][2*ystep+3][k]
                   +ctrlpts[2*xstep+1][2*ystep+3][k])/4.0;
    patch[3][3][k]=(ctrlpts[2*xstep+2][2*ystep+2][k]
                   +ctrlpts[2*xstep+3][2*ystep+2][k]
                   +ctrlpts[2*xstep+2][2*ystep+3][k]
                   +ctrlpts[2*xstep+3][2*ystep+3][k])/4.0;
}

A word to the wise...

Splines 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 just about 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 \texttt{glEvalCoord1f} \((u)\) or \texttt{glEvalCoord2f} \((u,v)\), and then use these points in the same way you would use any points you had defined in another way. These points, then, may represent colors, or normals, or texture coordinates, depending on what you need to create 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 at 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.