Chapter 14: Interpolation and Spline Modeling

This chapter introduces a new kind of graphical modeling that will let you create complex curves and surfaces by specifying just a few points to your program. This modeling is done by interpolating the points by linear combinations of some functions, and is both straightforward (after a little programming is done, or a graphics API capability is mastered) and flexible. One of the most important applications of this kind of modeling is found in creating curves and surfaces by interaction with the control points, making this the first fundamentally interactive kind of modeling we have seen.

Along with the modeling capabilities of this interpolation, we have relatively straightforward ways to generate normals and texture maps for our geometry based on the parameters of the interpolating functions. This lets us create not only sophisticated curves and surfaces, but also to add full lighting models and textures to them, making these some of the most powerful tools for making complex images that you will see in a first graphics course.

In order to understand this chapter, a student should have 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:

$$(1-t)P_0 + tP_1 \text{ for } t \text{ in } [0.,1.]$$

This form is almost trivial to use and yet it is quite suggestive, because it hints that there may be a very general way to calculate a set of points that interpolate the two given points by creating a function 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. Interpolating functions generally are defined on the interval to give us a standard range in the parameter *t*, the functions generally have only non-negative values on that interval and the sum of the functions at any value *t* is usually 1, so the interpolating points are a convex sum of the original points. We also often see the functions having values so that the interpolating function gives P_0 at one end of the interval [0.,1.] and P_1 at the other end of the interval.

If we examine the functions in the original example, we have $f_0(t) = (1–t)$ and $f_1(t) = t$, and these functions have the properties that we suggested above. 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. We also see that $f_0(t) + f_1(t) = 1$ for any value of *t* in the interval . 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:

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

Here we have three functions f_0, f_1, and f_2 that participate in the interpolation, with $f_0(t) = (1–t)^2$, $f_1(t) = 2t(1–t)$, and $f_2(t) = t^2$. These functions have by now achieved enough importance in our thinking that we will give them a name, and call them the *basis functions* for the interpolation. Further, we will call the points P_0, P_1, and P_2 the *control points* for the interpolation (although the formal literature on spline curves calls them *knots* and calls the endpoints of an interpolation *joints*). This particular set of functions have a similar property to the linear basis functions above, with $f_0(0) = 1$, $f_1(0) = 0$, and $f_2(0) = 0$, as well as $f_0(1) = 0$, $f_1(1) = 0$, and $f_2(1) = 1$, 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* takes any value between 0 and 1, such as .5. The shape of this interpolating curve is shown in Figure 14.1.

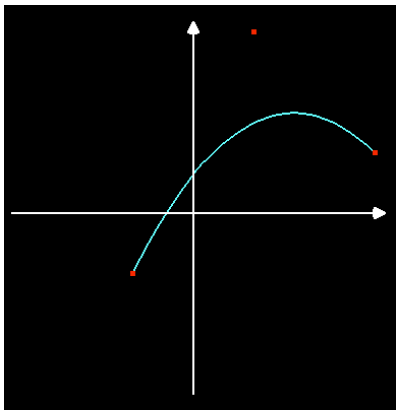

Figure 14.1: a quadratic interpolating curve for three points

The basis functions for these two cases are also of interest because they share a common source. In the case of linear interpolation, we can get the functions $f_0(t)$ and $f_1(t)$ as the terms when we expand $((1 - t) + t)^1$, and in the case of quadratic interpolation, we can get the functions $f_0(t)$, $f_1(t)$, and $f_2(t)$ as the terms when we expand $((1 - t) + t)^2$. In both cases, the functions all add to a value of 1 for any value of t, so we have an interpolation that gives a total weight of 1 for all the coefficients of the geometric points that we are using.

The observation above for the linear and quadratic cases is suggestive of a general approach for interpolating polynomials of degree N. In this approach, we would interpolate the functions that are components of the polynomial $((1 - t) + t)^N$ and take their coefficients from the geometry we want to interpolate. If we follow this pattern for the case of cubic interpolations (N=3), interpolating four points P_0, P_1, P_2, and P_3 would look like:

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

The shape of the curve this determines is illustrated in Figure 14.2. (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

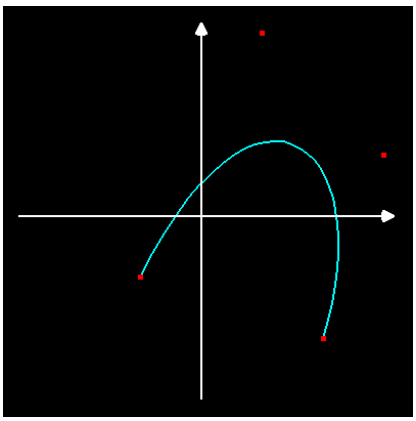

Figure 14.2: interpolating four points with the Bézier spline based on the Bernstein basis functions

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

$f_0(t) = (1-t)^3$

$f_1(t) = 3t(1-t)^2$

$f_2(t) = 3t^2(1-t)$, and

$f_3(t) = t^3$

are called the *cubic Bernstein basis* for the spline curve.

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 (P_1 and P_2). 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 shape of the curve is determined by the two middle control points; these points determine the initial and the ending directions of the curve, and the rest of the shape is determined by the weights given by the basis functions. The smoothness of the basis functions gives us the smoothness of the curve.

The approach above to defining the basis functions for our interpolations is not the only way to derive appropriate sets of functions. 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

$f_0(t)P_0 + f_1(t)P_1 + f_2(t)P_2 + f_3(t)P_3$ for t 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 it only interpolates the second and third control points as shown in Figure 14.3. This is a different kind of interpolating behavior, and 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, and actually goes through those two points, instead of P_0 and P_3. 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.

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.

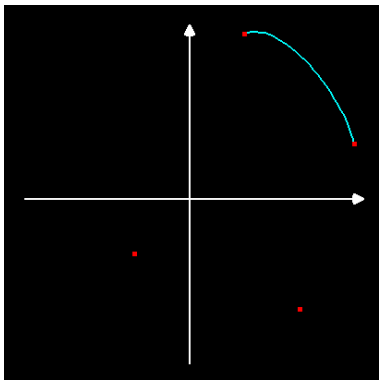

Figure 14.3: interpolating four points with the Catmull-Rom cubic spline

Extending interpolations to more control points

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 $N \geq 1$ 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	P_1	P_2		P_3	P_4		P_5	P_6	P_7
new:	P_0	P_1	P_2	Q_0	P_3	P_4	Q_1	P_5	P_6	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.

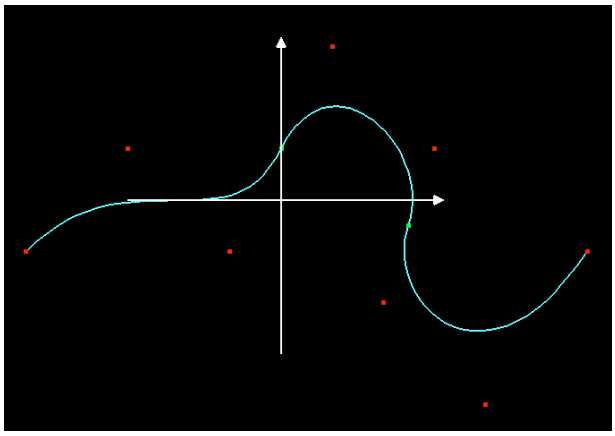

Figure 14.4: extending the Bézier curve by adding intermediate control points, shown in green

For the Catmull-Rom cubic spline, the fact that the interpolating curve only connects the control points P_1 and P_2 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 P_0 nor the ending control point P_3 is not included in the curve that interpolates P_0–P_3. 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 P_0, P_0, P_1, and P_2, and the first piece of the curve will then interpolate P_0 and P_1 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 P_1, P_2, P_3, and P_3, so the curve would interpolate the middle points, P_2 and P_3. 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 14.5.

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 P_0, P_1, P_2, and P_3 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 P_1, P_2, P_3, and P_4, and the one after that P_2, P_3, P_4, and P_5, 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 P_0–P_0–P_1–P_2; P_0–P_1–P_2–P_3; P_1–P_2–P_3–P_4; P_2–P_3–P_4–P_5; P_3–P_4–P_5–P_6; P_4–P_5–P_6–P_7; P_5–P_6–P_7–P_8; and P_6–P_7–P_8–P_8. 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 14.5, where we have taken the same set of control points that we used for the extended Bézier spline example.

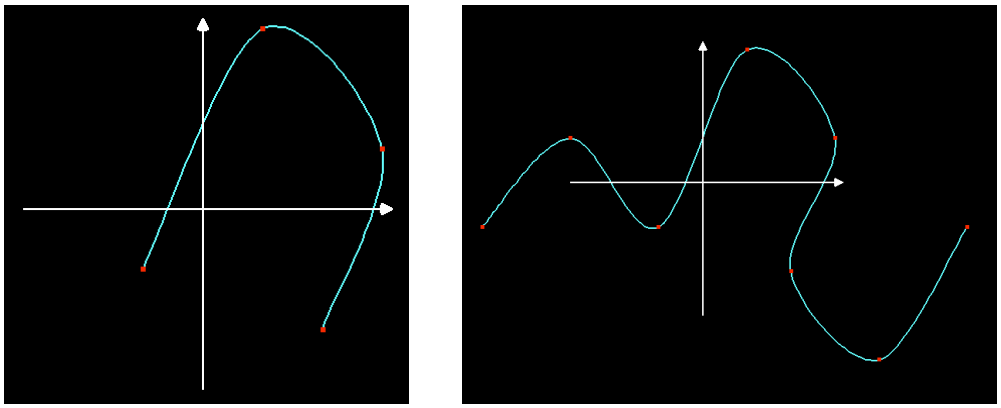

Figure 14.5: extending the Catmull-Rom curve by including the endpoints of the set (left) and by stepping along the extended set of control points (right)

The techniques defined above can easily be extended to generate interpolated surfaces. If you consider functions $f_0(t)$, $f_1(t)$, $f_2(t)$, and $f_3(t)$ as the basis for your cubic spline, you can apply them for two parameters, u and v, along with an extended set of 16 control points P_{ij} for i and j between 0 and 3, to give you a function of two variables:

$$f(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} f_i(u)\, f_j(v) P_{ij}$$

where the sum is taken over the 16 possible values of i and j. You can then step along the two variables u and v in any way you like and draw the surface in exactly the same way you would graph a function of two variables, as we discussed in a much earlier chapter. Some code to do this is sketched below, where the interpolating functions F0, F1, F2, and F3 have been defined elsewhere.

```
float u, u1, v, v1, ustep, vstep, x, y, z;
float cp[4][4][3];   // 4x4 array of 3D control points

for (u=0.; u<1.; u+=ustep)
   for (v=0.; v<1.; v+=vstep) {
      u1 = u+ustep; v1 = v+vstep;
      beginQuad();
         vertex3(eval(u,v,0),eval(u,v,1),eval(u,v,2));
         vertex3(eval(u,v1,0),eval(u,v1,1),eval(u,v1,2));
         vertex3(eval(u1,v1,0),eval(u1,v1,1),eval(u1,v1,2));
         vertex3(eval(u1,v,0),eval(u1,v,1),eval(u1,v,2));
      endQuad();
   }

float eval(float u, float v, int i)
{
   float result = 0.;

   result += F0(u)*F0(v)*cp[0][0][i]+F1(u)*F0(v)*cp[1][0][i];
   result += F2(u)*F0(v)*cp[2][0][i]+F3(u)*F0(v)*cp[2][0][i];
   result += F0(u)*F1(v)*cp[0][1][i]+F1(u)*F1(v)*cp[1][1][i];
   result += F2(u)*F1(v)*cp[2][1][i]+F3(u)*F1(v)*cp[3][1][i];
   result += F0(u)*F2(v)*cp[0][2][i]+F1(u)*F2(v)*cp[1][2][i];
```

```
      result += F2(u)*F2(v)*cp[2][2][i]+F3(u)*F2(v)*cp[3][2][i];
      result += F0(u)*F3(v)*cp[0][3][i]+F1(u)*F3(v)*cp[1][3][i];
      result += F2(u)*F3(v)*cp[2][3][i]+F3(u)*F3(v)*cp[3][3][i];
      return result;
   }
```

Here the function `eval(...)` computes the i^{th} coordinate of the interpolated point for the parameters u and v, while the first piece of code steps across the coordinate space and generates the quads that are to be drawn. The visual properties of the quads are not determined, but would be worked out through operations we have seen earlier in the book.

Generating normals for a patch

As we see, any patch can be written as a bilinear combination of the basis functions, giving us an analytic expression for any point on the patch. In particular, for each vertex in the grid that we will use to draw the patch, we can calculate the two directional derivatives for the surface's analytic function. With these derivatives, we can calculate two vectors tangent to the surface, and we can then compute the cross product of these vectors, and normalize the result, giving us a unit normal to the surface at the vertex. This will let us add the normals to whatever vertex list we use for our programming, giving us the ability to apply a full lighting model to the surface.

Generating texture coordinates for a patch

As we generate a patch, we will be creating a grid of points in the two-dimensional parameter space that is used for the surface as noted above. This is already a grid in a two-dimensional space and can readily be mapped linearly to any grid in any other two-dimensional space, such as a 2D texture map. With this mapping, you can calculate appropriate texture coordinates to add to your specification of vertices (and, as we just saw, of normals) for your patch.

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 are two kinds of evaluators available to you in OpenGL: one-dimensional and two-dimensional evaluators. 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.

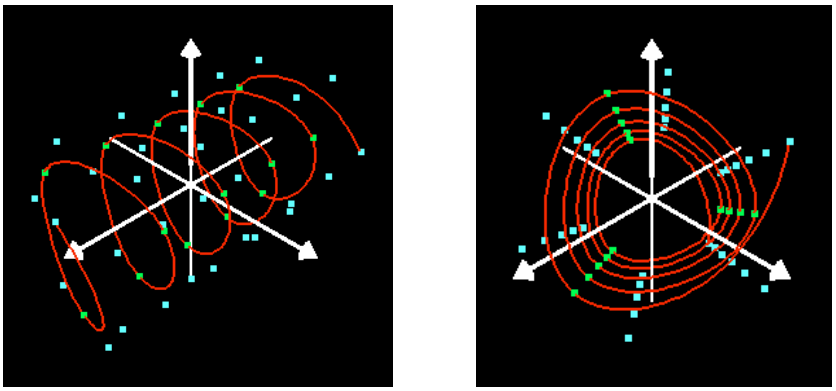

Figure 14.6: a spline curve defined via a 1D evaluator, from a point of view with $x = y = z$ (left) and rotated to show the relationship between control points and the curve shape (right) The cyan control points are the originals; the green control points are added as discussed above.

In Figures 14.6 through 14.8 we see several images that illustrate the use of evaluators to define geometry in OpenGL. Figure 14.6 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 14.7 shows a 2D evaluator used to define a single surface patch based on a 4x4 set of control points; and Figure 14.8 shows a surface defined by a 16x16 set of control points with additional intermediate control 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.

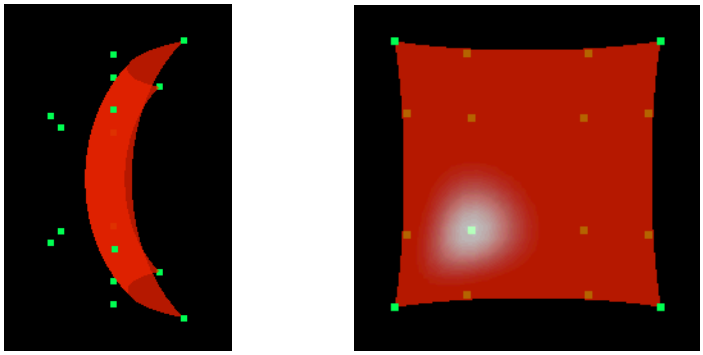

Figure 14.7: a spline patch defined by four control points using a 2D evaluator

The spline surface in Figure 14.7 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 patch should also help you see the shape of the patch from the lighting. In the larger surface of Figure 14.8, note how the surface extends smoothly between the different sets of control points.

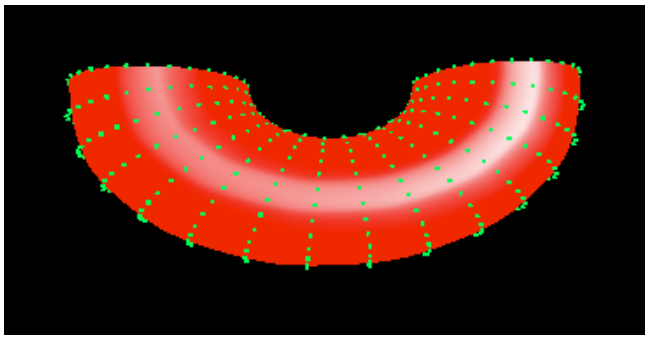

Figure 14.8: a spline surface defined by a 2D evaluator using a 16x16 set of control points. The original set of control points is extended with interpolated points as defined below

While these examples have used the full curve or surface generation capabilities of evaluators; this generates the entire mesh based on the control points. However, there are times when you might want to generate only a single point on a curve or a surface that corresponds to particular values of the parameter or parameters. For example, if you have developed a surface and you want to show a "fly-over" of the surface, you may want to position the eye along a curve that you define with control points. Then to generate images with eye points along this curve, you will want to define the eye position with particular values of the parameter that defines the curve. For each of these eye positions, you will want to evaluate the curve at the corresponding parameter value, getting the coordinates of that point to use with the `gluLookAt(...)` function.

Automatic normal and texture generation with evaluators

The images in Figure 14.7 and 14.8 include the standard lighting model and show specular highlighting, but as you will see from the code example for spline surfaces later in this chapter, neither explicit vertex definitions nor explicit normal definitions are used. Instead, 2D evaluators are used to generate the vertices and vertex normals are generated automatically. Basically, the key elements are as follows:

- to specify an array of 16 control points,
- to enable `GL_MAP2_VERTEX_3` to specify that you will generate points in 3D space with a 2D map,
- to enable `GL_AUTO_NORMAL` to specify that normals are to be generated analytically from the evaluator,
- to use `glMap2f(…)` to specify the details of the 2D mapping and to identify that the control point array above is to be used,
- to use `glMapGrid2f(…)` to specify how the 2D domain space is to be used in creating the vertex grid to be displayed, and
- to use `glEvalMesh2(…)` to carry out the evaluation and display the surface or, if you only want to calculate a point on the surface that corresponds to a single parameter pair (*u*, *v*), use `glEvalCoord2f(u,v)` to get the coordinates of that point.

There are, in fact, a number of similar functions that you can use for similar operations in different dimensions, and you should look these up in an OpenGL manual for more details.

Besides generating automatic normals, there are other capabilities for generating information for

your evaluator surface. You can generate automatic texture coordinates for your evaluator surfaces. If you enable `GL_MAP2_TEXTURE_COORD_2`, for example, and use `glMap2f(…)` with the first parameter `GL_MAP2_TEXTURE_COORD_2` and with similar parameters to those you used for vertex generation, then the `glEvalMesh2(…)` function will generate *s* and *t* coordinates for the texture on your 2D patch. As above, there are many variations to generate 1D through 4D textures from 1D or 2D meshes; see the OpenGL manuals for details. The image from Figure 14.7 is extended to Figure 14.9 by changing the color to white and applying the texture with modulation with automatic texture coordinates. Sample code for Figures 14.7 and 14.9 that includes automatic normal and texture coordinate generation is included later in this chapter.

Figure 14.9: a texture map on the patch of Figure 14.7 created with automatic texture coordinates

You can also generate the normals from your surface from the `glMap2f(...)` function if you use the `GL_MAP2_NORMAL_4` parameter, or the colors from your surface if you use the `GL_MAP2_COLOR_4` parameter.

Additional techniques

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 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 `glEvalCoord1f(u)` or `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.

To be more concrete, suppose you have a surface defined by two parameters, each having values lying in [0, 1]. If you wanted to impose a set of normals on the surface to achieve a lighting effect, you could define a set of control points that approximated the normals you want. (Recall

that a normal is just a 3D vector, just as is a point.) You can then define an evaluator for your new surface defined by the control points and a mapping from the parameters of the original surface to the parameters on the new surface. To define a normal at a point with parametric coordinates (*u*, *v*), then, you would determine the corresponding parameters (*u*',*v*') on the new surface, get the value (*x*, *y*, *z*) of the original surface as f(*u*, *v*) for whatever parametric function you are using and get the value (*r*, *s*, *t*) of the new surface with the `glEval2f(u',v')` function, and then use these values for the vertex in the function calls

```
glNormal3f(r,s,t); glVertex3f(x,y,z);
```

that would be used in defining the geometry for your image.

Another common 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. Other applications of splines in animation include positioning parts of a model to get smooth motion.

Definitions

As you see in Figures 14.6 and 14.7, 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 14.10, and the speed with which this happens is determined by the distance between the extreme and adjacent control points.

Figure 14.10: two spline curves showing the curves with different control point layouts

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 Figure 14.6. 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 below 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 equation (1) below, with the single parameter *u* taking values between 0 and 1.

$$(1) \qquad \sum_{i=0}^{3} a_i u^i \qquad \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij} u^i v^j$$

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), 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.

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 the essential parts of setting up the control points are fully included. 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.

```
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;
   }
}

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 and 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();
      ...
   }
}
```

As we noted, we used the OpenGL evaluator to give us points on the curve, and got those points manually with the `glEvalCoord1f(...)` function. We could have also taken a more automatic approach, using the function `glMapGrid1f(...)` to generate a series of evenly-spaced points along the curve and the function `glEvalMesh1(...)` to create the curve. This is done in the example below.

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 Figure 14.7 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. This code also includes the `glEnable(…)` and `glMapGrid2f(GL_MAP2_TEXTURE_COORD_2,...)` statements that are used to generate automatic texture coordinates for Figure 14.9, but does not include the details of the texture mapping code. Note that the third parameter of the `glMapGrid2f(...)` function that specifies the texture coordinate generation is 4.0; this corresponds to mapping the texture coordinates over four grid points.

```
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_TEXTURE_COORD_2);
   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]);
   glMap2f(GL_MAP2_TEXTURE_COORD_2,0.0,4.0,3,4,0.0,4.0,12,4,,&patch[0][0][0])
   glMapGrid2f(NUM, 0.0, 1.0, NUM, 0.0, 1.0);
   glEvalMesh2(GL_FILL, 0, NUM, 0, NUM);
}
```

As in the example above, we could have drawn the patch manually by following the `glMap2f(...)` function by a double loop in which we use `glEvalCoord2f(...)` to generate the actual vertices. You might want to think about which of these is preferable for a particular image based on the tradeoff between automatic operation and manual control.

The considerations for extending a single patch to create 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 by the function `doPatch(...)` above in just the same way as it is handled for the patch example, so it is omitted here.

```
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...

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.

Summary

In this chapter we have seen how to interpolate geometric points based on various kinds of basis functions, giving us powerful tools for creating curves and surfaces with relatively few control points. This result is a new kind of modeling that extends the simple primitives we saw early in this book and lets you create much more sophisticated images. This interpolation also allows you to add shading and texture mapping to your surfaces, adding to their value to the user.

Questions

1. Each of the sets of basis functions in this chapter has the property that at any value in [0.,1.]. each of the functions is non-negative and the sum of their values is 1. In addition, all the basis functions but one have a zero value at one endpoint, and that one has a value of 1. Could you use any set of functions with these properties as the basis functions for an interpolation? Would the results be similar to, or different from, the results of using the standard basis functions? Why?

2. Take some pairs of functions with the properties of the previous exercise and interpolate two points with them. Can the interpolating points ever have any shape but a straight line? Even if the interpolating points form a straight line, can you find an example where the interpolating function is not linear?

3. Find a set of functions that interpolate a set of three or four points and have the properties of the standard basis functions in this chapter, but that are different from the sets of basis functions we have seen so far. Write the interpolating function in closed form and use any technique to draw the resulting curve. How much is it like (or different from) the standard interpolating curves we have seen in this chapter? Can you create an interpolating curve that is very different from these?

Exercises

4. Choose an object with a shape you find pleasing and try to design a set of points that might determine that shape, or a shape close to it, when used as control points for a spline curve or surface. Use the evaluator approach in the code examples in this chapter to draw the curve or surface from your points to check this out.

5. Given a set of six or more control points for a cubic curve, remembering that you will need to have an even number of points. Hand-build the set of additional control points you will need to extend a smooth piecewise-cubic curve across all these control points. Do a similar exercise for control points for a surface, using a 2D array of control points that is larger than 4x4.

Experiments

6. The code examples in this chapter set up the control points for a spline curve and then call the OpenGL evaluator functions to create the actual curves or patches. However, early in the chapter we discussed how to calculate the curve or surface points for any parameter or parameter pair. Program these calculations and generate your own curve or surface points for an interpolation.

7. Take the same control points of the previous exercise and set up control point arrays for the OpenGL evaluator functions that we saw in this chapter. Again, generate the curve or surface determined by these control points, using the evaluators. Compare the programming effort needed to generate these curves or surfaces both from first principles, as in the previous experiment, and with evaluators.

Projects

8. (The small house) In the original modeling for the small house, you probably used a very simple shape for the house's roof. Update this and design a roof that is built as a spline surface in order to get a more sophisticated look to the house. Then comment out your access to the original roof function and add the new roof design to the house scene.

9. (A scene graph parser) How would you represent the geometry of an evaluator in the geometry node of a scene graph? How would you generate the code for the evaluator when you parsed the scene graph? What would you do in these two cases if you used an interpolator instead of an evaluator?