CHAPTER 21

Bezier Patches

The last couple of chapters have involved shaders that manipulate the position of the vertex according to some function, but each of these shaders has affected the geometry in limited ways. Some situations require a large number of vertices to be set according to some relatively complex function. The temptation is to process the vertices on the CPU where it's easy to implement complex functions. A better way is to use the CPU to define a coarse representation of the basic shape of the geometry and then let the GPU process the larger number of points. You can do this using Bezier patches. This chapter looks at many aspects of Bezier patches, including the following topics:

- The theory behind Bezier curves and patches.
- Computing the normals of a patch (with an extremely brief introduction to calculus).
- Implementing Bezier patches with a shader.
- Setting up patch data to send to a shader.
- Uses for patches in manipulation and level of detail operations.
- Patch representation for complex objects.

Everything in this chapter hinges on you understanding the complex subject of Bezier curves and patches, so bear with me as we delve into some pretty murky depths.

Lines and Curves and Patches, Oh...

Consider Figure 21.1, which is a screenshot from this chapter's sample application.

The original data for this model was a flat 2D plane of many triangles, yet the resulting surface is far more complex than what any of the previous techniques could have produced. The secret behind this technique is that the surface in Figure 21.1 was produced with a Bezier patch. Patches are 3D extensions of Bezier curves, so I start there, but keep the screenshot in mind as you read the following.

NOTE

Some texts use the letters u and or v to represent the range of the curve. I have chosen to use s and t to avoid confusion with the most common texture coordinates u and v. If you see textbooks that use other letters, know that the concepts are exactly the same. They are just using other symbols.

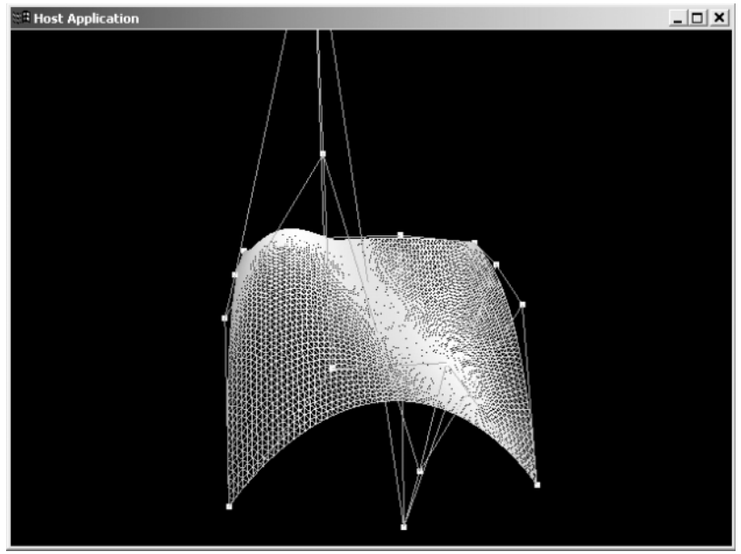

Figure 21.1

The Bezier patch application rendered in wireframe.

Bezier curves are generally defined as curves that can be described by a set of control points. In most cases, these curves are defined with four control points, as shown in Figure 21.2.

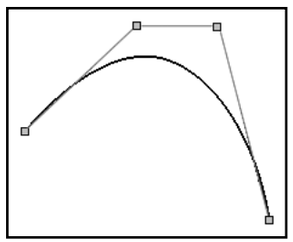

Figure 21.2

A Bezier curve defined with four control points.

These four control points define the curve with P0 and P3 defining the endpoints. If you think of an arbitrary variable s as ranging from 0 to 1 over the length of the curve, you can define the values of the curve as a function of s.

With control points P0-P3 and the range s, the equation for the curve becomes

$$Q(s) = \sum_{i=0}^{3} p_i b_i(s)$$

where Q(s) is the position of a point along the curve and b_i is a basis function that describes the "weight" of each point in the calculation. For our purposes, the basis functions are the following Bernstein polynomials:

$$b_0 = (1 - s)^3$$
$$b_1 = 3s(1 - s)^2$$
$$b_2 = 3s^2(1 - s)$$
$$b_3 = s^3$$

Putting all this together, the long form of the values along the curve as a function of s is the following dreadful-looking function:

$$Q(s) = p_0(1 - s)^3 + p_1 3s(1 - s)^2 + p_2 3s^2(1 - s) + p_3 s^3$$

In graphical terms, the basis functions are shown in Figure 21.3. As you can see, P0 and P3 are the actual endpoints of the curve, but at every other point, the other control points have some contribution to the value of Q. I marked a couple of points of interest on Figure 21.3.

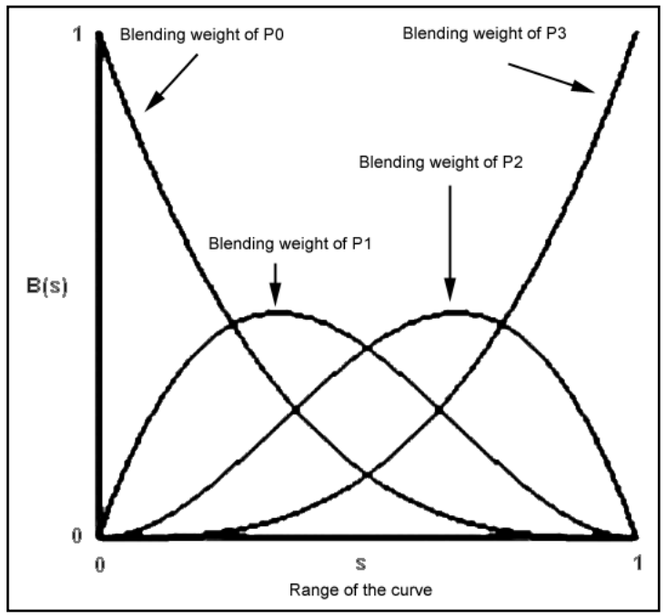

Figure 21.3

The basis functions.

The line at s = a shows where the first two control points have equal influence on the value of Q, and the line at s = b shows where the inner points have equal control

and the outer points have equal (but far less) control. At this point, the easiest way to internalize the concepts might be to break out a sheet of paper and a pencil, plot four points, and plot a couple of points along the curve to see for yourself how they work. Take your time; I'll wait.

One of the nice things about these curves is that the math doesn't care how many points you actually plot along the curve. If you plot 10 points, you'll create a rough but accurate curve. If you plot one million points, you will have a very accurate curve, but you will be old and lonely. I revisit this point later when I talk about some level of detail considerations.

La Vie de Msr. Bezier

Bezier developed the idea of Bezier curves while working for Renault. At the time, curves given in engineering drawings were fairly arbitrary and not very consistent. Bezier developed these curves as a way to give a more rigorous definition to the curves in design drawings. This was especially useful when manufacturing CAD/CAM machines emerged. Incidentally, similar ideas were developed by James Ferguson and Paul de Casteljau, but both innovations were kept secret by their employers. Bezier was credited with the curves that now bear his name. Pierre Bezier passed away in 1999.

Extending these ideas to another dimension is a relatively simple matter of extending the four control points to a four-by-four control grid. If we use t to specify the range in the other dimension, the equation for values anywhere in the patch becomes

$$Q(s,t) = \sum_{i=0}^{3} \sum_{j=0}^{3} p_{i,j} b_i(s) b_j(t)$$

In this equation, the basis functions are the same, and all the underlying concepts are the same; I've just extended the ideas into another dimension. Again, if you'd like, plot out a couple points if it helps you internalize the concepts. I refrain from giving the equation in its long form.

You now have an equation that gives you the value of any point in the space of the patch based on the 16 control points within the patch. Soon I show you the shader that makes this possible, but first I have to address the method of deriving the surface normals that is necessary for lighting calculations.

Deriving Surface Normals with "Calculus"

I'm placing calculus in quotes because a full explanation of calculus is well outside the scope of this humble tome. If you already know calculus, you should understand the following ideas, but you'll also see that my explanation is far less than complete. If you have not already studied calculus, please take this quick and dirty explanation with a grain of salt.

There are two branches of calculus, but the one applicable here is differential calculus. If you think of every function as describing some curve on a graph, then you can use differential calculus to find the derivative of that function. The derivative is a new function that describes the slope of the curve at any given point. For the simplest case, consider the function shown in Figure 21.4.

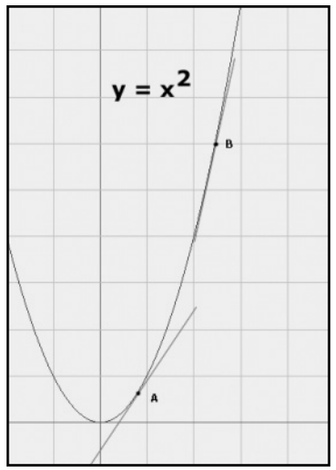

Figure 21.4

Changing slopes on a curve.

As you can see, the slope of the curve at point A is very different from the slope of the curve at point B. To approximate the slope at any given point, you could find the value of the function at point A and the value of the function at a very short distance away. The approximation then takes the form of the following equation:

$$f'(a) = \lim_{h \to 0} \frac{f(a + h) - f(a)}{h}$$

I don't delve into the exact mathematical proof of differential calculus, but the short version of the story is that you find the derivative by shrinking that interval to

an infinitesimal value. Figure 21.5 revisits the graph from Figure 21.4. The second graph shows the derivative of the first function. As you can see, as X increases, the slope also increases. You can see this increase on the graph of the derivative.

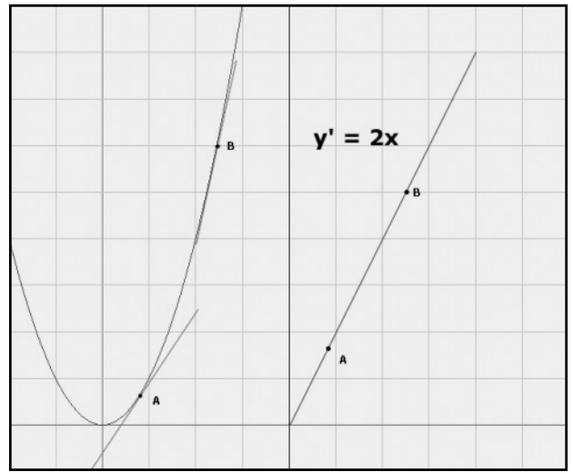

Figure 21.5

Graphing the derivative.

There are many rules for finding different kinds of derivatives, but in this chapter you only need to find the derivatives of the polynomials that define the Bezier patch. Derivatives of simple polynomials take the following form:

$$\text{if} \quad f(a) = ca^x, \quad \text{then}$$

$$\frac{d}{dx} f(a) = f'(a) = xca^{x-1}$$

One thing to remember is that the derivative of a constant is zero. If you plot a constant function, there is no slope. Again, you might want to take a break and sketch out a few of these graphs on graph paper and get a feel for what's going on. Once you try it yourself, it will probably be much clearer. Using these rules, you can compute the following derivatives of the basis functions for a 2D Bezier curve. The derivatives for the other dimension of a 3D patch are analogous:

$$b'_0 = -3(1 - s)^2$$
$$b'_1 = 3 - 12s + 9s^2$$
$$b'_2 = 6s - 9s^2$$
$$b'_3 = 3s^2$$

Derivatives of Other Functions

Please remember that this is only the simplest explanation of how to find the derivative for a simple polynomial and that some functions might not adhere to this simple rule. I don't go into all the rules, but it might be worthwhile to mention the derivatives for sine and cosine. The derivatives for sin(x) and cos(x) are cos(x) and –sin(x), respectively. This becomes pretty apparent if you plot both curves and look at their slopes at various points. If you are interested, you can go back and use this information to compute the proper lighting values for the example from Chapter 18.

You are now very close to being able to compute the normals for a Bezier patch. The derivatives for the basis functions allow you to compute the slopes of the surface in both directions on the patch (s and t in this case). These are the surface tangents, not the normals. That's where the cross product comes in. In Chapter 2, I said that the cross product of two vectors yields the vector that was perpendicular to both. In this case, you want the vector that is perpendicular to the two surface tangents. You can now find the normal vector for any point on the surface of a Bezier patch with the following equation:

$$N(s,t) = \sum_{i=0}^{3}\sum_{j=0}^{3} p_{i,j} b'_i(s) b_j(t) \times \sum_{i=0}^{3}\sum_{j=0}^{3} p_{i,j} b_i(s) b'_j(t)$$

This is relatively difficult to illustrate on a 2D page, but consider the very simple case of a flat control grid producing a flat patch. The surface tangents would then be straight vectors lying on the surface of the plane, and the cross product of those two lines would be a vector pointing straight up from the plane, as shown in Figure 21.6.

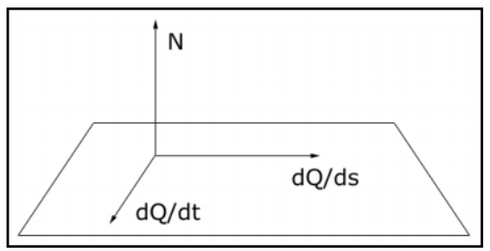

Figure 21.6

A very simple surface normal.

One thing to remember is that order matters when finding the cross product of two vectors. If you compute the cross product in the wrong order, the vector will point the opposite direction. This mistake is fairly easy to identify because your lights show up "backwards."

This all sounds like a lot of work, but it's not really that bad. In the next few pages, I show how you can set up the nasty math once and then control everything else with the control points.

Computing the Patch Values in a Shader

If you compare and contrast my version of this technique with a similar effect found on the nVidia site, you'll see a major difference in the way we handle the basis functions. The nVidia effect encodes the s and t positions in the vertices and then computes the basis functions at the beginning of the shader before applying the control points. My version computes the basis values once when the vertices are loaded and then performs the relatively simple task of multiplying and adding up the influences of all the control points. The tradeoff is that my shader has less computational overhead, but at a higher overall data cost. To be honest, I haven't tested both versions to see which is faster under equal conditions (the effect example handles lighting differently), but a determination of which way is better would have to account for the exact requirements of your project and where bottlenecks are found to occur. In any case, be aware that the tradeoff exists.

The following shader code expects `v7` to contain the precomputed basis functions in the s direction and `v8` to contain the functions for the t direction. The `v9` and `v10` input registers also contain the precomputed derivatives used to compute the vertex normal. Also, the positions of the 16 control points are stored in constants `c10` through `c25`. I show you how that data is stored when I get to the application code. The full code appears in Bezier.vsh in the shaders directory:

```
vs.1.1
```

These first lines move the data from the input registers to temporary registers. This makes it easier to use the data because the shader allows better concurrent access to temporary registers than it does to the input registers:

```
mov r7, v7
mov r8, v8

mov r9, v9
mov r10, v10
```

This first chunk of code computes the influence of the `S0T0` control point. This is the equivalent to P0 on the Bezier curve, but I name the control points by their s

and t values because they are in two dimensions. The block of code multiplies the two precomputed basis functions together and then multiplies that by the position of S0T0 given in the constant c10. It then multiplies each of the precomputed derivatives with the position of the control point. In the case of the tangent vectors, you do not want to multiply the two values together because you must use the final tangent vectors to compute the surface normal. Throughout this shader, r0 is a temporary working variable, r1 contains the vertex position, r2 is the tangent vector in the s direction, and r3 is the tangent vector in the t direction:

```
;S0T0 control point
mul r0.x, r7.x, r8.x
mul r1, c10, r0.x

mul r2, c10, r9.x
mul r3, c10, r10.x
```

This next block computes the influence of the S0T1 control point. It multiples the appropriate basis values and then uses the mad instruction to multiply the control point value and add it to the position stored in r1. The same procedure is repeated for the tangent vectors, but again they are not multiplied together:

```
;S0T1 control point
mul r0.x, r7.x, r8.y
mad r1, c11, r0.x, r1

mad r2, r9.x,  c11, r2
mad r3, r10.y, c11, r3
```

I removed several lines of the shader from this listing because they basically repeat the same operation for each of the control points. You can find the full listing on the CD. At first glance, the lines of code are exactly the same, but remember that each of the four values of r7, r8, r9, and r10 contains separate precomputed basis

TIP

You never want to use more shader instructions than you have to. In some cases, shader limitations will force you to. In this case, the shader cannot access multiple input registers in the same line, so I move the input values into temporary registers. This is one of the few places where you really need to move data without performing some other value added operation.

values. It's extremely important to match the proper control point with the proper basis functions. For instance, S0T0 matched with r7.x and r8.x and now the final point S3T3 matches with r7.w and r8.w:

```
;S3T3 control point
mul r0.x, r7.w, r8.w
mad r1, c25, r0.x, r1

mad r2, r9.w,  c25, r2
mad r3, r10.w, c25, r3
```

The r1 register now contains the interpolated position of this vertex on the Bezier patch. In the application code, I loaded a mesh that was centered on the origin from –0.5 to 0.5, and then I added 0.5 to compute s and t values in the range of 0 to 1. Here, I subtract 0.5 again to undo that correction. This might not be needed for other implementations, but I wanted to add a complication into the mesh-loading process just to show how you might handle something. You could have just as easily derived the s and t values for each vertex in a different way and saved this instruction. I talk more about this when I get into the application code:

```
add r1, r1, c6
```

The following lines compute the cross product in a manner described in some of the nVidia documentation. Shaders provide native support for the dot product, but cross product requires two instructions and some clever swizzling. If you deconstruct these two lines, you can see that it does match the cross product shown in Chapter 2:

```
mul r6, r3.yzxw, r2.zxyw
mad r6, -r2.yzxw, r3.zxyw, r6
```

The r6 register now contains the normal vector. You must normalize it in the usual way before using it in the lighting instructions:

```
dp3 r5.w, r6, r6
rsq r5.w, r5.w
mul r6, r6, r5.w
```

For the sake of simplicity, I am computing only simple diffuse lighting by finding the dot product of the normal and the light vector. There is no reason you couldn't add more, but a full description of lighting in vertex shaders does not appear until Chapter 24:

```
dp3 oD0, r6, -c5
```

The final four lines transform the new vertex position to clip space. You can use the control grid to transform the positions of each point, but the world matrix is still good for positioning, rotating, or scaling the actual mesh:

```
dp4 oPos.x, r1, c0
dp4 oPos.y, r1, c1
dp4 oPos.z, r1, c2
dp4 oPos.w, r1, c3
```

That's the shader (or at least part of it), but the picture isn't really complete until you take a look at the application code. Next, I show you how the application prepares the vertex data, computes the control points, and feeds the shader.

The Bezier Application

This section contains the abridged code for the Bezier application. The application loads the patch vertices from a mesh in a file and creates another set of vertices to use when displaying the control grid. In the render loop, the application sets the 16 control points and feeds them to the shader through a set of 16 constants. It then renders the mesh, and the vertex shader computes the actual vertex positions. See the source code for the complete code listing.

The following structure defines the vertex format for the patch vertices. The shader doesn't actually use the three position values, but I included them here because sometimes you might want to display the original data either to debug or to simply render the existing model. In the following sample, the mesh is a plane of many vertices, but I can also imagine scenarios when you might want to use the control grid to warp a 3D model instead of a simple plane. For instance, you might want to compute the influence of the control points, but instead of setting the position, you could add the interpolated position to the real coordinates. This would allow you to warp a real 3D object using the control grid and the original position, but generating the normals could be a bit tricky. Finally, including the position is convenient when cloning the mesh because the cloning functions have a place to put the position data. For this sample, I could have restructured the vertex format and saved the three unused floats. Also, as I mentioned earlier, I could have encoded the s and t values into the vertex structure and generated the basis values in the shader. In the best case, the vertex structure could have been as small as two floats. This would be nearly one-eighth of amount of data (if the following format were more efficient), but it would require in the neighborhood of 50 percent more shader instructions. Again, there are advantages and disadvantages to either approach. Chances are that the other approach is faster on very fast hardware, but it depends on your exact needs. In any case, be aware that at least two different approaches both use the same underlying concepts:


```
struct BEZIER_VERTEX
{
       float x, y, z;
       float Bs0, Bs1, Bs2, Bs3;
       float Bt0, Bt1, Bt2, Bt3;
       float dBs0, dBs1, dBs2, dBs3;
       float dBt0, dBt1, dBt2, dBt3;
};
```

The following declaration defines the vertex structure for the vertex shader. The unused position information is stored in `v0`, and the precomputed basis values for both position and tangent vectors are stored as four values in the first four texture coordinate registers. This doesn't mean that you must use these values as texture coordinates; it just defines which registers contain the values:

```
DWORD BezierDeclaration[] =
{
       D3DVSD_STREAM(0),
       D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3),
       D3DVSD_REG(D3DVSDE_TEXCOORD0,D3DVSDT_FLOAT4),
       D3DVSD_REG(D3DVSDE_TEXCOORD1,D3DVSDT_FLOAT4),
       D3DVSD_REG(D3DVSDE_TEXCOORD2,D3DVSDT_FLOAT4),
       D3DVSD_REG(D3DVSDE_TEXCOORD3,D3DVSDT_FLOAT4),
       D3DVSD_END()
};
```

The following structure renders the control grid. The vertex structure contains the position, and the shader sets the color according to a constant. The extremely simple shader is in the shader directory as BezierControl.vsh:

```
struct CONTROL_VERTEX
{
       float x, y, z;
};
```

This is the FVF and declaration that creates a buffer of the simple control vertices. Rendering the control grid is very simple and could have been done without a shader, but I used a shader to increase your exposure to shader code. Also, you can use the shader to easily set the vertex color with a constant. If I had used the fixed

function pipeline, I would have had to add code that sets a material, adjusts lighting, and so on. The shader method is arguably easier:

```
#define D3DFVF_CONTROLVERTEX (D3DFVF_XYZ)

DWORD ControlDeclaration[] =
{
        D3DVSD_STREAM(0),
        D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3),
        D3DVSD_END()
};
```

The `ExtractBuffers` function assumes that a mesh has already been loaded and cloned to the Bezier vertex format. In this case, the mesh is a plane of vertices with position values ranging from –0.5 to 0.5 in both the X and the Z directions. If you shift these values by 0.5, it creates a mesh that has a convenient range of 0 to 1, and the shader corrects this shifting by subtracting 0.5 later.

Creating s and t Values

I created a mesh that had convenient values so I could concentrate on more important matters, but using meshes with less convenient values is still easy. For any mesh, you can find the bounding box and use the extents of that box to find s and t values in the range of 0 to 1. For instance, if the mesh has x values that range from 0 to 100, you can create the correct s value by dividing each x value by 100.

I could have just as easily created the vertices in code by creating a vertex buffer and index buffer. It was a little bit less code to create the patch vertices with a mesh, and you will see a how to create the vertex and index buffers in the next function:

```
HRESULT CTechniqueApplication::ExtractBuffers()
{
        m_pMesh->GetVertexBuffer(&m_pMeshVertexBuffer);
        m_pMesh->GetIndexBuffer(&m_pMeshIndexBuffer);

        BEZIER_VERTEX *pMeshVertices;

        m_pMeshVertexBuffer->Lock(0, m_pMesh->GetNumVertices() *
                                  sizeof(BEZIER_VERTEX),
                                 (BYTE **)&pMeshVertices, 0);

        for (long Vertex = 0;
```



```
             Vertex < m_pMesh->GetNumVertices();
             Vertex++)
        {
```

The patch shader works on two dimensions, so I use the x and z coordinates and ignore y. There is nothing about Bezier patches that explicitly forces you to work with a horizontal plane. The vertices and control points could also be vertically oriented. Theoretically, any orientation is fine as long as you are consistent all the way through. Again, the half-unit shift is an idiosyncrasy of the particular mesh I'm using; it is not a general requirement:

```
            float S = pMeshVertices[Vertex].x + 0.5;
            float T = pMeshVertices[Vertex].z + 0.5;
```

The following four values are the precomputed basis functions for this particular s value. The underlying mesh data does not change, so there is no reason to compute these values each time:

```
            pMeshVertices[Vertex].Bs0 = (1.0f - S) * (1.0f - S) *
                                        (1.0f - S);
            pMeshVertices[Vertex].Bs1 = 3.0f * S * (1.0f - S) *
                                        (1.0f - S);
            pMeshVertices[Vertex].Bs2 = 3.0f * S * S * (1.0f - S);
            pMeshVertices[Vertex].Bs3 = S * S * S;
```

Here the process is repeated for the t value. The basis functions are exactly the same. Only this time, they use the t value instead of the s value:

```
            pMeshVertices[Vertex].Bt0 = (1.0f - T) * (1.0f - T) *
                                        (1.0f - T);
            pMeshVertices[Vertex].Bt1 = 3.0f * T * (1.0f - T) *
                                        (1.0f - T);
            pMeshVertices[Vertex].Bt2 = 3.0f * T * T * (1.0f - T);
            pMeshVertices[Vertex].Bt3 = T * T * T;
```

These next eight lines compute the values of the derivatives of the basis functions. You derive these functions using the rule for simple polynomials. If you expand the basis functions and then take the derivative, you obtain the functions that follow:

```
            pMeshVertices[Vertex].dBs0 = (6.0f * S) -
            (3.0f * S *S) - 3.0f;
            pMeshVertices[Vertex].dBs1 = 3.0f - (12.0f * S) +
                                         (9.0f * S * S);
            pMeshVertices[Vertex].dBs2 = (6.0f * S) - (9.0f * S * S);
```

```
            pMeshVertices[Vertex].dBs3 = 3.0f * S * S;

            pMeshVertices[Vertex].dBt0 = (6.0f * T) - (3.0f * T *
             T) - 3.0f;
            pMeshVertices[Vertex].dBt1 = 3.0f - (12.0f * T) +
                                         (9.0f * T * T);
            pMeshVertices[Vertex].dBt2 = (6.0f * T) - (9.0f * T * T);
            pMeshVertices[Vertex].dBt3 = 3.0f * T * T;
        }
```

After you have computed the basis values, unlock the buffer. The vertex shader makes any further changes:

```
        m_pMeshVertexBuffer->Unlock();

        return S_OK;
}
```

`CreateGridVisuals` creates the simple vertices used to display the control grid. This is really only a debugging and learning tool, so I don't spend too much time optimizing their usage:

```
HRESULT CTechniqueApplication::CreateGridVisuals()
{
```

You need 16 vertices to show the 16 control points. You use these vertices to render both the points and the grid of lines that connect them. They are created in managed memory so that they do not need to be explicitly recreated if the device is reset:

```
        if (FAILED(m_pD3DDevice->CreateVertexBuffer(16 *
                                                    sizeof(CONTROL_VERTEX),
                                    0, D3DFVF_CONTROLVERTEX,
                                    D3DPOOL_MANAGED,
                                    &m_pControlVertexBuffer)))
                return E_FAIL;
```

The index buffer allows you to reuse the 16 vertices to draw the lines that interconnect the control points. This is not just an optimization; I think it's easier than creating more points:

```
        if (FAILED(m_pD3DDevice->CreateIndexBuffer(48 * sizeof(short),
                                        0, D3DFMT_INDEX16,
                                        D3DPOOL_MANAGED,
```



```
                                        &m_pControlIndexBuffer)))
                return E_FAIL;
```

The index buffer contains the data for a line list of interconnections between the control points. You can hardcode these values because they won't ever change. It is also possible to generate these values in a loop, but writing this way is easier to show what is actually going on:

```
        short *pIndex;
        m_pControlIndexBuffer->Lock(0, 48 * sizeof(short),
                                    (BYTE**)&pIndex, 0);
        short Indices[] = {0, 1, 1, 2, 2, 3, 4, 5, 5, 6, 6, 7, 8, 9, 9,
                10, 10, 11, 12, 13, 13, 14, 14, 15, 0, 4, 4, 8, 8, 12,
                1, 5, 5, 9, 9, 13, 2, 6, 6, 10, 10, 14, 3, 7, 7,
                11, 11, 15};
        memcpy(pIndex, &Indices, 48 * sizeof(short));
        m_pControlIndexBuffer->Unlock();
}
```

The render function is where the real magic happens. Figures 21.7 and 21.8 show the application in action both in solid and wireframe renderings:

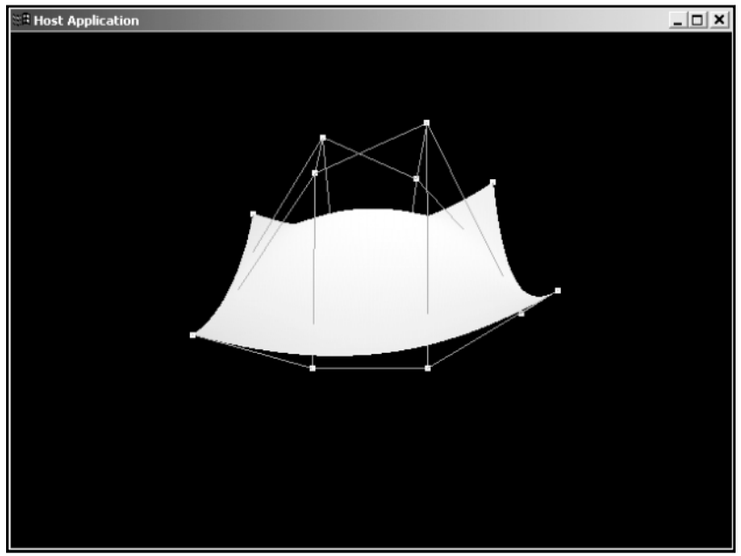

Figure 21.7

Bezier patch solid rendering.

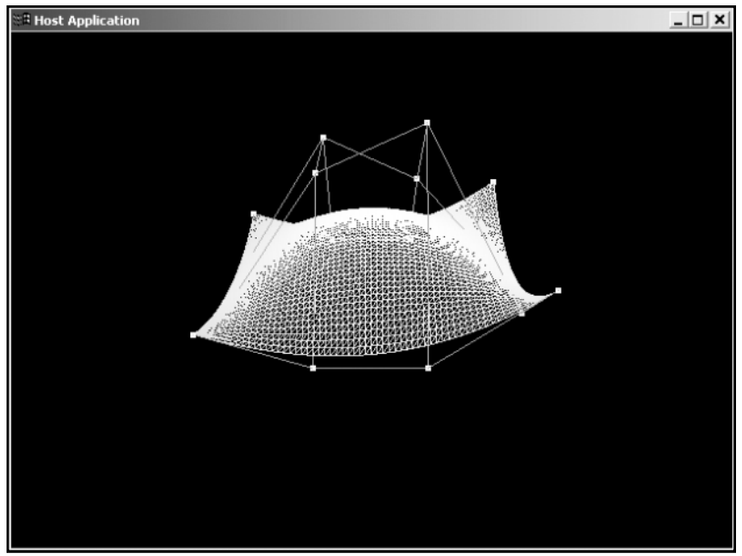

Figure 21.8

Bezier patch wireframe.

```
void CTechniqueApplication::Render()
{
```

The underlying data is based on a very small one-unit array of vertices and corresponding control points. Computing everything with those values is convenient, and the scaling matrix lets you scale the final mesh to any size you need:

```
D3DXMatrixScaling(&m_WorldMatrix, 5.0f, 5.0f, 5.0f);
```

I also added a rotation to the mesh so that you can see the Bezier functionality working with the standard matrix transformations. The rotation matrix is concatenated with the scaling values in the world matrix:

```
D3DXMATRIX Rotation;
D3DXMatrixRotationY(&Rotation, (float)GetTickCount() / 1000.0f);
m_WorldMatrix *= Rotation;
```

Once the new world matrix is computed, you still need to concatenate, transpose, and send the matrix to the vertex shader:

```
D3DXMATRIX ShaderMatrix = m_WorldMatrix *
                          m_ViewMatrix *
                          m_ProjectionMatrix;

D3DXMatrixTranspose(&ShaderMatrix, &ShaderMatrix);
m_pD3DDevice->SetVertexShaderConstant(0, &ShaderMatrix, 4);
```


The light is shining straight down, and I didn't bother to specify a specific light color or any other attributes. Also, when you get to Chapter 24 you'll see some of the considerations you must take to make sure the lighting is consistent with the matrix transformations. If you want to experiment with this code, you might need to augment it to make the lighting work correctly. See Chapter 24 for more details:

```
D3DXVECTOR4 LightDir(0.0f, -1.0f, 0.0f, 0.0f);
m_pD3DDevice->SetVertexShaderConstant(5, &LightDir, 1);
```

Here I also set the correction values to account for this particular mesh. In other cases, you may be able to save the constant and the instruction count:

```
D3DXVECTOR4 Correction(-0.5f, 0.0f, -0.5f, 0.0f);
m_pD3DDevice->SetVertexShaderConstant(6, &Correction, 1);
```

These warp values are just arbitrary values I picked to animate the control grid. I highly recommend you experiment with these values or any of the control grid parameters. Just remember that if you pull the grid in too many different directions, the patch may be mathematically correct but very ugly. Experiment all you want, but change the code is small increments until you are comfortable with what is going on:

```
float Warp1 = 2.0f * sin((float)GetTickCount() / 1000.0f);
float Warp2 = 2.0f * cos((float)GetTickCount() / 1000.0f);
```

Each of these four blocks of code sets the control-point positions for one row of points. For the sample, I change only the height of the points and keep the other values evenly spaced along the grid:

```
D3DXVECTOR4 ControlS0T0(0.0f, 0.25f * Warp1, 0.0f, 1.0f);
D3DXVECTOR4 ControlS0T1(0.0f, 0.0f, 0.33f, 1.0f);
D3DXVECTOR4 ControlS0T2(0.0f, 0.0f, 0.66f, 1.0f);
D3DXVECTOR4 ControlS0T3(0.0f, 0.25f * Warp1, 1.0f, 1.0f);

D3DXVECTOR4 ControlS1T0(0.33f, 0.0f, 0.0f, 1.0f);
D3DXVECTOR4 ControlS1T1(0.33f, 0.33f * Warp1 + 0.33 * Warp2,
                        0.33f, 1.0f);
```

```
D3DXVECTOR4 ControlS1T2(0.33f, -0.33f * Warp1 + 0.66f * Warp2,
                        0.66f, 1.0f);
D3DXVECTOR4 ControlS1T3(0.33f, 0.0f, 1.0f, 1.0f);

D3DXVECTOR4 ControlS2T0(0.66f, 0.0f, 0.0f, 1.0f);
D3DXVECTOR4 ControlS2T1(0.66f, 0.66f * Warp1 + 0.33 * Warp2,
                        0.33f, 1.0f);
D3DXVECTOR4 ControlS2T2(0.66f, -0.66f * Warp1 + 0.66f * Warp2,
                        0.66f, 1.0f);
D3DXVECTOR4 ControlS2T3(0.66f, 0.0f, 1.0f, 1.0f);

D3DXVECTOR4 ControlS3T0(1.0f, 0.25f * Warp2, 0.0f, 1.0f);
D3DXVECTOR4 ControlS3T1(1.0f, 0.0f, 0.33f, 1.0f);
D3DXVECTOR4 ControlS3T2(1.0f, 0.0f, 0.66f, 1.0f);
D3DXVECTOR4 ControlS3T3(1.0f, 0.25f * Warp2, 1.0f, 1.0f);
```

When the control points are set, each control point is set in the shader. It may have been more optimal to create an array of 16 vectors and then send the complete block of vectors in a single call to `SetVertexShaderConstant`. However, this was a better way to demonstrate which constant matched with which control point. If you are interested, you can optimize the way that constants are set:

```
m_pD3DDevice->SetVertexShaderConstant(10, &ControlS0T0, 1);
m_pD3DDevice->SetVertexShaderConstant(11, &ControlS0T1, 1);
m_pD3DDevice->SetVertexShaderConstant(12, &ControlS0T2, 1);
m_pD3DDevice->SetVertexShaderConstant(13, &ControlS0T3, 1);
m_pD3DDevice->SetVertexShaderConstant(14, &ControlS1T0, 1);
m_pD3DDevice->SetVertexShaderConstant(15, &ControlS1T1, 1);
m_pD3DDevice->SetVertexShaderConstant(16, &ControlS1T2, 1);
m_pD3DDevice->SetVertexShaderConstant(17, &ControlS1T3, 1);
m_pD3DDevice->SetVertexShaderConstant(18, &ControlS2T0, 1);
m_pD3DDevice->SetVertexShaderConstant(19, &ControlS2T1, 1);
m_pD3DDevice->SetVertexShaderConstant(20, &ControlS2T2, 1);
m_pD3DDevice->SetVertexShaderConstant(21, &ControlS2T3, 1);
m_pD3DDevice->SetVertexShaderConstant(22, &ControlS3T0, 1);
m_pD3DDevice->SetVertexShaderConstant(23, &ControlS3T1, 1);
m_pD3DDevice->SetVertexShaderConstant(24, &ControlS3T2, 1);
m_pD3DDevice->SetVertexShaderConstant(25, &ControlS3T3, 1);
```


Everything is now ready for some actual rendering. First make sure that the shader is set, and then set the vertex and index buffers:

```
m_pD3DDevice->SetVertexShader(m_BezierShader);

m_pD3DDevice->SetStreamSource(0, m_pMeshVertexBuffer,
                              sizeof(BEZIER_VERTEX));
m_pD3DDevice->SetIndices(m_pMeshIndexBuffer, 0);
```

This line is commented out in the source code, but you can uncomment it if you want to see the mesh rendered in wireframe:

```
//m_pD3DDevice->SetRenderState(D3DRS_FILLMODE,
                               D3DFILL_WIREFRAME);
```

Draw the mesh with the vertex shader. If you enable the preceding line, the output displays the wireframe view of the mesh; otherwise it displays a solid rendering of the mesh:

```
m_pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0,
                                   m_pMesh->GetNumVertices(), 0,
                                   m_pMesh->GetNumFaces());
```

This line ensures that the rendering mode is solid for all subsequent calls. If you want to optimize, you can make sure that this call is enabled only if the wireframe call is enabled:

```
    m_pD3DDevice->SetRenderState(D3DRS_FILLMODE, D3DFILL_SOLID);
```

The last thing the render function does is call the function that renders the control grid. You can remove this line if you like:

```
    RenderControlGrid();
}
```

The `RenderControlGrid` function is not at all optimized, but it is still useful to see what is going on with the control points:

```
void CTechniqueApplication::RenderControlGrid()
{
```

This code locks the vertex buffer so that it can be filled with control point data. In the next chapter, you'll see a method for accessing specific constants with the address register. I could have done that here and avoided the lock and the need to

retrieve the constants, but it seemed overly complicated and there is no real need to optimize this function:

```
CONTROL_VERTEX *pVertices;

m_pControlVertexBuffer->Lock(0, 16 * sizeof(CONTROL_VERTEX),
                             (BYTE **)&pVertices, 0);
```

The loop walks through each vertex and retrieves the value of the constant. This is a bit heavy handed because you just passed the constants into the shader in the previous function. The advantage of this method is that you can be sure the values are consistent. Once the constants are retrieved, they are used to set the vertex position. Note the correction here keeps everything together with the mesh object. I don't go into too much detail, but keep this in mind as you read the next chapter. You could encode each vertex with the address of the constant that it matches to and then rewrite the control grid shader to retrieve the real position from that constant. If you did that, this function would become much simpler:

```
for (long Vertex = 0; Vertex < 16; Vertex++)
{
        D3DXVECTOR4 Temp;
        m_pD3DDevice->GetVertexShaderConstant(10 + Vertex,
                                              &Temp, 1);
        pVertices[Vertex].x = Temp.x - 0.5f;
        pVertices[Vertex].y = Temp.y;
        pVertices[Vertex].z = Temp.z - 0.5f;
}

m_pControlVertexBuffer->Unlock();
```

The control grid shader uses `c4` to set the vertex color. The first pass draws all the lines between the control points using a red color.

```
D3DXVECTOR4 LineColor(1.0f, 0.0f, 0.0f, 0.0f);
m_pD3DDevice->SetVertexShaderConstant(4, &LineColor, 1);
```

These next lines set the shader and the proper data sources used for two passes:

```
m_pD3DDevice->SetVertexShader(m_ControlShader);
m_pD3DDevice->SetIndices(m_pControlIndexBuffer, 0);
m_pD3DDevice->SetStreamSource(0, m_pControlVertexBuffer,
                              sizeof(CONTROL_VERTEX));
```


The first rendering call draws the lines:

```
m_pD3DDevice->DrawIndexedPrimitive(D3DPT_LINELIST, 0,
                                   16, 0, 24);
```

These lines now reset the vertex color constant so that the control points can be rendered as yellow points:

```
D3DXVECTOR4 PointColor(1.0f, 1.0f, 0.0f, 0.0f);
m_pD3DDevice->SetVertexShaderConstant(4, &PointColor, 1);
```

Before I render the points, I set the point size. It's possible that some devices might not support this. If that's the case, you may not actually see the points:

```
        float PointSize = 5.0f;
        m_pD3DDevice->SetRenderState(D3DRS_POINTSIZE,
                                     *((DWORD*)&PointSize));
        m_pD3DDevice->DrawPrimitive(D3DPT_POINTLIST, 0, 16);
}
```

Uses and Advantages of Bezier Patches

There are many uses for Bezier curves and Bezier patches, but most have to do with the fact that the parametric representation allows you to apply the curve functions to an arbitrary number of vertices. For instance, you could use a lower-resolution mesh in the sample application and get a shape that was correct but much coarser. This lends itself well to the dynamic level of detail meshes.

Imagine a section of terrain defined with a set of Bezier patches that created rolling hills and deep valleys. If you are actually standing in a valley, you might want to render the terrain with a very high number of vertices so that all the edges appear smooth. If you hop into a plane and fly above the valley, you can render the same Bezier patches using fewer vertices. The patch calculations ensure that the general shape of the patches are correct, yet you can save calculations because you don't need as much detail when you are farther away.

This idea is not limited to something like terrain. Most algorithms for higher-order primitives use similar concepts to render 3D objects. For example, you can represent an object as a collection of parametric patches rather than a set of vertices. Using the patches, you can dynamically generate different meshes at any

level of detail. This is the basis for hardware implementations as well. Some hardware implementations use lower-resolution meshes to interpolate smoother curved values without the data-transfer overhead of additional vertices. The underlying algorithms might not be exactly the same as the algorithms shown here, but the basic concepts are the same.

Also, as I mentioned earlier, you can use the patches to define how to warp a real 3D object. To do this, use the patch values to increment or scale the vertex positions rather than set them directly. You can use this method to create very organic and smooth warping effects. You can also use it to warp or move materials such as cloth. Imagine a waving flag. You can use the CPU to generate the rough control points of the flag and let the Bezier functionality control the smooth interpolation of the points on the flag. You can also apply this idea to moving capes and so on.

Finally, you can apply the concepts behind this chapter to other areas outside of rendering. For instance, you can describe a path of motion with a Bezier curve and use the basis functions to generate a smooth interpolated position at any time value. This type of approach can be useful for any situation where you might need to derive smooth values from relatively coarse data. Keep in mind that this sample generates a very smooth surface based on only four points.

Connecting Curves and Patches

Many of these ideas involve shapes or paths with more control points. You can specify a Bezier curve with more than 4 control points or a Bezier patch with more than 16, but this often becomes computationally expensive. Instead, you can join curves by having two curves share a common endpoint. This is an acceptable solution, but it can create abrupt transitions if the two curves are very different. You can also join two patches by having them share a common row of four control points, but the same caveat about abrupt transitions also applies.

If you do choose to add more control points, the generalized definition for the set of basis functions is as follows:

$$b_{i,n}(s) = s^i(1 - s)^{(n-i)}n!/i!(n - i)!$$

In Conclusion...

If you have not studied a lot of geometry or calculus, this chapter may have been like drinking from the fire hose. My hope is that I watered down the math enough

to make the overall concepts understandable. If you have not studied calculus, you might have to simply trust me about the derivatives, but I do recommend trying a couple simple calculations for yourself. It should be pretty apparent that the methodology works even if you don't understand the underlying mechanics. If you are really interested in learning more, many math resources on the Web attack these subjects from a variety of different angles. Do a couple searches and see which explanations work for you.

In the meantime, I finish this chapter off with a recap of some of the major points:

- Bezier curves and patches let you define curves and surfaces with a very small amount of data.
- The equation for Bezier curves is a function of a set of basis functions and a set of control points.
- You can use the derivatives of the basis functions to find the tangent vectors for a surface. Once those vectors are found, the cross product of two tangent vectors yields the surface normal at a given point.
- Once the control points are found, you can then use a vertex shader to apply the influences of the control points to an arbitrary number of vertices.
- The vertex shader can also compute the surface normal and apply lighting calculations.
- My method involves encoding more data into the vertices and using fewer instructions. A sample effect on the nVidia site does the opposite. There are advantages and disadvantages to each approach.
- Bezier patches are most useful in situations where you want to control a large amount of vertices with a very small amount of changing data.
- Bezier patches are also good when you don't necessarily want the number of vertices to remain constant. This is useful in cases where you want a dynamic level of detail control and you want to render the same general shape with fewer vertices.
- If you want more control over the shape, you can add more control points, but it might be better to connect multiple curves. The result is fewer calculations, but abrupt changes at the interface might be an issue.