Tessellating a curve

In this recipe, we'll take a look at the basics of tessellation shaders by drawing a **cubic Bezier curve**. A Bezier curve is a parametric curve defined by four control points. The control points define the overall shape of the curve. The first and last of the four points define the start and end of the curve, and the middle points guide the shape of the curve, but do not necessarily lie directly on the curve itself. The curve is defined by interpolating the four control points using a set of **blending functions**. The blending functions define how much each control point contributes to the curve for a given position along the curve. For Bezier curves, the blending functions are known as the **Bernstein polynomials**.

$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i$$

In the preceding equation, the first term is the binomial coefficient function (shown in the following equation), **n** is the degree of the polynomial, **i** is the polynomial number, and **t** is the parametric parameter.

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

The general parametric form for the Bezier curve is then given as a sum of the products of the Bernstein polynomials with the control points ($\mathbf{P}_i$).

$$P(t) = \sum_{i=0}^{n} B_i^n(t)P_i$$

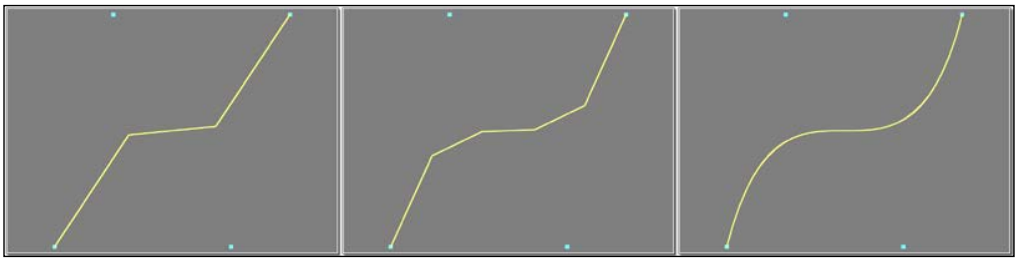
In this example, we will draw a cubic Bezier curve, which involves four control points (n = 3).

$$P(t) = B_0^3(t)P_0 + B_1^3(t)P_1 + B_2^3(t)P_2 + B_3^3(t)P_3$$

And the cubic Bernstein polynomials are:

$$B_0^3(t) = (1-t)^3$$
$$B_1^3(t) = 3(1-t)^2t$$
$$B_2^3(t) = 3(1-t)t^2$$
$$B_3^3(t) = t^3$$

As stated in the introduction of this chapter, the tessellation functionality within OpenGL involves two shader stages. They are the tessellation control shader (TCS) and the tessellation evaluation shader (TES). In this example, we'll define the number of line segments for our Bezier curve within the TCS (by defining the outer tessellation levels), and evaluate the Bezier curve at each particular vertex location within the TES. The following screenshot shows the output of this example for three different tessellation levels. The left figure uses three line segments (level 3), the middle uses level 5, and the right-hand figure is created with tessellation level 30. The small squares are the control points.



The control points for the Bezier curve are sent down the pipeline as a patch primitive consisting of four vertices. A patch primitive is a programmer-defined primitive type. Basically, it is a set of vertices that can be used for anything that the programmer chooses. The TCS is executed once for each vertex within the patch, and the TES is executed, a variable number of times, depending on the number of vertices produced by the TPG. The final output of the tessellation stages is a set of primitives. In our case, it will be a line strip.

Part of the job for the TCS is to define the tessellation level. In very rough terms, the tessellation level is related to the number of vertices that will be generated. In our case, the TCS will be generating a line strip, so the tessellation level is the number of line segments in the line strip. Each vertex that is generated for this line strip will be associated with a tessellation coordinate that will vary between zero and one. We'll refer to this as the u coordinate, and it will correspond to the parametric parameter t in the preceding Bezier curve equation.

What we've looked at so far is not, in fact, the whole story. Actually, the TCS will trigger a generation of a set of line strips called isolines. Each vertex in this set of isolines will have a u and a v coordinate. The u coordinate will vary from zero to one along a given isoline, and v will be constant for each isoline. The number of distinct values of u and v is associated with two separate tessellation levels, the so-called "outer" levels. For this example, however, we'll only generate a single line strip, so the second tessellation level (for v) will always be one.

Within the TES, the main task is to determine the position of the vertex associated with this execution of the shader. We have access to the u and v coordinates associated with the vertex, and we also have (read-only) access to all of the vertices of the patch. We can then determine the appropriate position for the vertex by using the parametric equation described above, with u as the parametric coordinate (t in the preceding equation).

Getting ready

The following are the important uniform variables for this example:

- `NumSegments`: This is the number of line segments to be produced.
- `NumStrips`: This is the number of isolines to be produced. For this example, this should be set to one.
- `LineColor`: This is the color for the resulting line strip.

Set the uniform variables within the main OpenGL application. There are a total of four shaders to be compiled and linked. They are the vertex, fragment, tessellation control, and tessellation evaluation shaders.

How to do it...

To create a shader program that will generate a Bezier curve from a patch of four control points, use the following steps:

1. Use the following code for the simple vertex shader:

```
layout (location = 0 ) in vec2 VertexPosition;

void main()
{
    gl_Position = vec4(VertexPosition, 0.0, 1.0);
}
```

2. Use the following code as the tessellation control shader:

```
layout( vertices=4 ) out;

uniform int NumSegments;
uniform int NumStrips;

void main()
{
    // Pass along the vertex position unmodified
    gl_out[gl_InvocationID].gl_Position =
              gl_in[gl_InvocationID].gl_Position;
    // Define the tessellation levels
    gl_TessLevelOuter[0] = float(NumStrips);
```

```
    gl_TessLevelOuter[1] = float(NumSegments);
}
```

3. Use the following code as the tessellation evaluation shader:

```
layout( isolines ) in;
uniform mat4 MVP;  // projection * view * model

void main()
{
    // The tessellation u coordinate
    float u = gl_TessCoord.x;

    // The patch vertices (control points)
    vec3 p0 = gl_in[0].gl_Position.xyz;
    vec3 p1 = gl_in[1].gl_Position.xyz;
    vec3 p2 = gl_in[2].gl_Position.xyz;
    vec3 p3 = gl_in[3].gl_Position.xyz;

    float u1 = (1.0 - u);
    float u2 = u * u;

    // Bernstein polynomials evaluated at u
    float b3 = u2 * u;
    float b2 = 3.0 * u2 * u1;
    float b1 = 3.0 * u * u1 * u1;
    float b0 = u1 * u1 * u1;

    // Cubic Bezier interpolation
    vec3 p = p0 * b0 + p1 * b1 + p2 * b2 + p3 * b3;

    gl_Position = MVP * vec4(p, 1.0);

}
```

4. Use the following code for the fragment shader:

```
uniform vec4 LineColor;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
    FragColor = LineColor;
}
```

5. It is important to define the number of vertices per patch within the OpenGL application. You can do so using the `glPatchParameter` function:

```
glPatchParameteri( GL_PATCH_VERTICES, 4);
```

6. Render the four control points as a patch primitive within the OpenGL application's render function:

```
glDrawArrays(GL_PATCHES, 0, 4);
```

How it works...

The vertex shader is just a "pass-through" shader. It sends the vertex position along to the next stage without any modification.

The tessellation control shader begins by defining the number of vertices in the output patch:

```
layout (vertices = 4) out;
```

Note that this is not the same as the number of vertices that will be produced by the tessellation process. In this case, the patch is our four control points, so we use a value of four.

The main method within the TCS passes the input position (of the patch vertex) to the output position without modification. The arrays `gl_out` and `gl_in` contain the input and output information associated with each vertex in the patch. Note that we assign and read from location `gl_InvocationID` in these arrays. The `gl_InvocationID` variable defines the output patch vertex for which this invocation of the TCS is responsible. The TCS can access all of the array `gl_in`, but should only write to the location in `gl_out` corresponding to `gl_InvocationID`.

Next, the TCS sets the tessellation levels by assigning to the `gl_TessLevelOuter` array. Note that the values for `gl_TessLevelOuter` are floating point numbers rather than integers. They will be rounded up to the nearest integer and clamped automatically by the OpenGL system.

The first element in the array defines the number of isolines that will be generated. Each isoline will have a constant value for `v`. In this example, the value of `gl_TessLevelOuter[0]` should be one. The second defines the number of line segments that will be produced in the line strip. Each vertex in the strip will have a value for the parametric `u` coordinate that will vary from zero to one.

In the TES, we start by defining the input primitive type using a layout declaration:

```
layout (isolines) in;
```

This indicates the type of subdivision that is performed by the tessellation primitive generator. Other possibilities here include `quads` and `triangles`.

Within the `main` function of the TES, the variable `gl_TessCoord` contains the tessellation `u` and `v` coordinates for this invocation. As we are only tessellating in one dimension, we only need the `u` coordinate, which corresponds to the x coordinate of `gl_TessCoord`.

The next step accesses the positions of the four control points (all the points in our patch primitive). These are available in the `gl_in` array.

The cubic Bernstein polynomials are then evaluated at `u` and stored in `b0`, `b1`, `b2`, and `b3`. Next, we compute the interpolated position using the Bezier curve equation described some time back. The final position is converted to clip coordinates and assigned to the output variable `gl_Position`.

The fragment shader simply applies `LineColor` to the fragment.

There's more...

There's a lot more to be said about tessellation shaders, but this example is intended to be a simple introduction so we'll leave that for the following recipes. Next, we'll look at tessellation across surfaces in two dimensions.

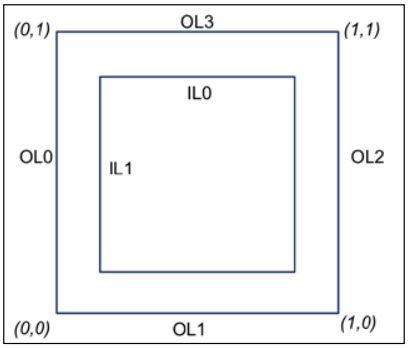
Tessellating a 2D quad

One of the best ways to understand OpenGL's hardware tessellation is to visualize the tessellation of a 2D quad. When linear interpolation is used, the triangles that are produced are directly related to the tessellation coordinates (u,v) that are produced by the tessellation primitive generator. It can be extremely helpful to draw a few quads with different inner and outer tessellation levels, and study the triangles produced. We will do exactly that in this recipe.

When using quad tessellation, the tessellation primitive generator subdivides (u,v) parameter space into a number of subdivisions based on six parameters. These are the inner tessellation levels for u and v (inner level 0 and inner level 1), and the outer tessellation levels for u and v along both edges (outer levels 0 to 3). These determine the number of subdivisions along the edges of parameter space and internally. Let's look at each of these individually:

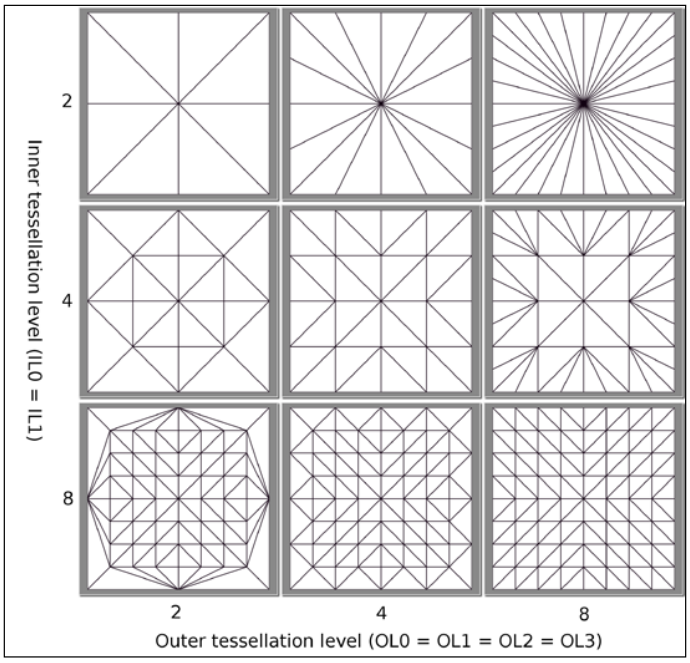
- **Outer level 0 (OL0)**: This is the number of subdivisions along the v direction where u = 0
- **Outer level 1 (OL1)**: This is the number of subdivisions along the u direction where v = 0
- **Outer level 2 (OL2)**: This is the number of subdivisions along the v direction where u = 1
- **Outer level 3 (OL3)**: This is the number of subdivisions along the u direction where v = 1
- **Inner level 0 (IL0)**: This is the number of subdivisions along the u direction for all internal values of v
- **Inner level 1 (IL1)**: This is the number of subdivisions along the v direction for all internal values of u

The following diagram represents the relationship between the tessellation levels and the areas of parameter space that are affected by each. The outer levels defines the number of subdivisions along the edges, and the inner levels define the number of subdivisions internally.



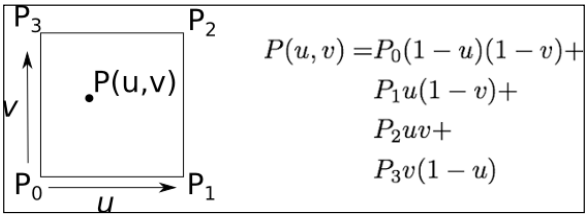
The six tessellation levels described some time back can be configured via the arrays `gl_TessLevelOuter` and `gl_TessLevelInner`. For example, `gl_TessLevelInner[0]` corresponds to **IL0**, `gl_TessLevelOuter[2]` corresponds to **OL2**, and so on.

If we draw a patch primitive that consists of a single quad (four vertices), and use linear interpolation, the triangles that result can help us to understand how OpenGL does quad tessellation. The following diagram shows the results for various tessellation levels:



When we use linear interpolation, the triangles that are produced represent a visual representation of parameter (u, v) space. The x axis corresponds to the u coordinate and the y axis corresponds to the v coordinate. The vertices of the triangles are the (u,v) coordinates generated by the tessellation primitive generator. The number of subdivisions can be clearly seen in the mesh of triangles. For example, when the outer levels are set to 2 and the inner levels are set to 8, you can see that the outer edges have two subdivisions, but within the quad, u and v are subdivided into 8 intervals.

Before jumping into the code, let's discuss linear interpolation. If the four corners of the quad are as shown in the following figure, then any point within the quad can be determined by linearly interpolating the four corners with respect to parameters u and v.



We'll let the tessellation primitive generator create a set of vertices with appropriate parametric coordinates, and we'll determine the corresponding positions by interpolating the corners of the quad using the preceding equation.

Getting ready

The outer and inner tessellation levels will be determined by the uniform variables `Inner` and `Outer`. In order to display the triangles, we will use the geometry shader described earlier in this chapter.

Set up your OpenGL application to render a patch primitive consisting of four vertices in counter clockwise order as shown in the preceding figure.

How to do it...

To create a shader program that will generate a set of triangles using quad tessellation from a patch of four vertices, use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0 ) in vec2 VertexPosition;

void main()
{
    gl_Position = vec4(VertexPosition, 0.0, 1.0);
}
```

2. Use the following code as the tessellation control shader:

```
layout( vertices=4 ) out;

uniform int Outer;
uniform int Inner;
void main()
{
    // Pass along the vertex position unmodified
    gl_out[gl_InvocationID].gl_Position =
               gl_in[gl_InvocationID].gl_Position;

    gl_TessLevelOuter[0] = float(Outer);
    gl_TessLevelOuter[1] = float(Outer);
    gl_TessLevelOuter[2] = float(Outer);
    gl_TessLevelOuter[3] = float(Outer);

    gl_TessLevelInner[0] = float(Inner);
    gl_TessLevelInner[1] = float(Inner);
}
```

3. Use the following code as the tessellation evaluation shader:

```
layout( quads, equal_spacing, ccw ) in;

uniform mat4 MVP;

void main()
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;

    vec4 p0 = gl_in[0].gl_Position;
    vec4 p1 = gl_in[1].gl_Position;
    vec4 p2 = gl_in[2].gl_Position;
    vec4 p3 = gl_in[3].gl_Position;

    // Linear interpolation
    gl_Position =
        p0 * (1-u) * (1-v) +
        p1 * u * (1-v) +
        p3 * v * (1-u) +
        p2 * u * v;

    // Transform to clip coordinates
    gl_Position = MVP * gl_Position;
}
```

4. Use the geometry shader from the recipe, *Drawing a wireframe on top of a shaded mesh*.
5. Use the following code as the fragment shader:

```
uniform float LineWidth;
uniform vec4 LineColor;
uniform vec4 QuadColor;

noperspective in vec3 EdgeDistance;  // From geom. shader

layout ( location = 0 ) out vec4 FragColor;

float edgeMix()
{
   // ** insert code here to determine how much of the edge
   // color to include (see recipe "Drawing a wireframe on
   // top of a shaded mesh").  **
}

void main()
{
    float mixVal = edgeMix();

    FragColor = mix( QuadColor, LineColor, mixVal );

}
```

6. Within the render function of your main OpenGL program, define the number of vertices within a patch:

```
glPatchParameteri(GL_PATCH_VERTICES, 4);
```

7. Render the patch as four 2D vertices in counter clockwise order.

How it works...

The vertex shader passes the position along to the TCS unchanged.

The TCS defines the number of vertices in the patch using the layout directive:

```
layout (vertices=4) out;
```

In the `main` function, it passes along the position of the vertex without modification, and sets the inner and outer tessellation levels. All four of the outer tessellation levels are set to the value of `Outer`, and both of the inner tessellation levels are set to `Inner`.

In the tessellation evaluation shader, we define the tessellation mode and other tessellation parameters with the input layout directive:

```
layout ( quads, equal_spacing, ccw ) in;
```

The parameter `quads` indicates that the tessellation primitive generator should tessellate the parameter space using quad tessellation as described some time back. The parameter `equal_spacing` says that the tessellation should be performed such that all subdivisions have equal length. The last parameter, `ccw`, indicates that the primitives should be generated with counter clockwise winding.

The `main` function in the TES starts by retrieving the parametric coordinates for this vertex by accessing the variable `gl_TessCoord`. Then we move on to read the positions of the four vertices in the patch from the `gl_in` array. We store them in temporary variables to be used in the interpolation calculation.

The built-in output variable `gl_Position` then gets the value of the interpolated point using the preceding equation. Finally, we convert the position into clip coordinates by multiplying by the model-view projection matrix.

Within the fragment shader, we give all fragments a color that is possibly mixed with a line color in order to highlight the edges.

See also

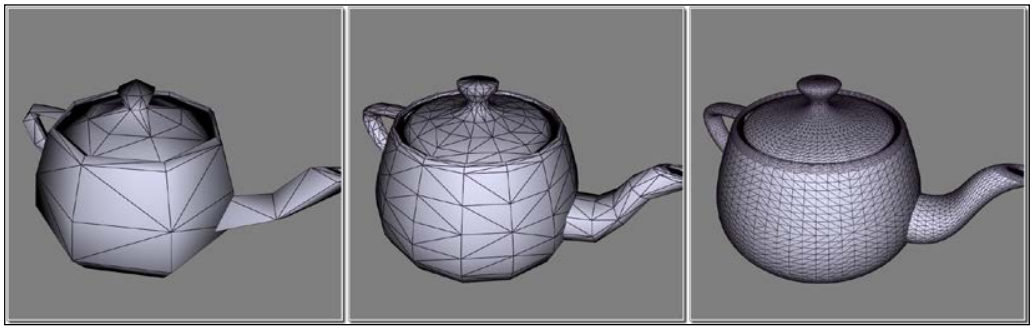
- The *Drawing a wireframe on top of a shaded mesh* recipe

Tessellating a 3D surface

As an example of tessellating a 3D surface, let's render (yet again) the "teapotahedron". It turns out that the teapot's data set is actually defined as a set of 4 x 4 patches of control points, suitable for cubic Bezier interpolation. Therefore, drawing the teapot really boils down to drawing a set of cubic Bezier surfaces.

Of course, this sounds like a perfect job for tessellation shaders! We'll render each patch of 16 vertices as a patch primitive, use quad tessellation to subdivide the parameter space, and implement the Bezier interpolation within the tessellation evaluation shader.

The following figure shows an example of the desired output. The left teapot is rendered with inner and outer tessellation level 2, the middle uses level 4 and the right-hand teapot uses tessellation level 16. The tessellation evaluation shader computes the Bezier surface interpolation.



First, let's take a look at how cubic Bezier surface interpolation works. If our surface is defined by a set of 16 control points (laid out in a 4 x 4 grid) P_{ij}, with i and j ranging from 0 to 3, then the parametric Bezier surface is given by the following equation:

$$P(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i^3(u)B_j^3(v)P_{ij}$$

The instances of **B** in the preceding equation are the cubic Bernstein polynomials (refer to the previous recipe, *Tessellating a 2D quad*).

We also need to compute the normal vector at each interpolated location. To do so, we have to compute the cross product of the partial derivatives of the preceding equation:

$$\mathbf{n}(u,v) = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}$$

The partial derivatives of the Bezier surface boil down to the partial derivatives of the Bernstein polynomials:

$$\frac{\partial P}{\partial u} = \sum_{i=0}^{3}\sum_{j=0}^{3} \frac{\partial B_i^3(u)}{\partial u} B_j^3(v)P_{ij}$$

$$\frac{\partial P}{\partial v} = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i^3(u) \frac{\partial B_j^3(v)}{\partial v} P_{ij}$$

We'll compute the partials within the TES and compute the cross product to determine the normal to the surface at each tessellated vertex.

Getting ready

Set up your shaders with a vertex shader that simply passes the vertex position along without any modification (you can use the same vertex shader as was used in the *Tessellating a 2D quad* recipe). Create a fragment shader that implements whatever shading model you choose. The fragment shader should receive the input variables `TENormal` and `TEPosition`, which will be the normal and position in camera coordinates.

The uniform variable `TessLevel` should be given the value of the tessellation level desired. All of the inner and outer levels will be set to this value.

How to do it...

To create a shader program that creates Bezier patches from input patches of 16 control points, use the following steps:

1. Use the vertex shader from the *Tessellating a 2D quad* recipe.
2. Use the following code for the tessellation control shader:

```
layout( vertices=16 ) out;

uniform int TessLevel;

void main()
{
    // Pass along the vertex position unmodified
    gl_out[gl_InvocationID].gl_Position =
                gl_in[gl_InvocationID].gl_Position;

    gl_TessLevelOuter[0] = float(TessLevel);
    gl_TessLevelOuter[1] = float(TessLevel);
    gl_TessLevelOuter[2] = float(TessLevel);
    gl_TessLevelOuter[3] = float(TessLevel);

    gl_TessLevelInner[0] = float(TessLevel);
    gl_TessLevelInner[1] = float(TessLevel);
}
```

3. Use the following code for the tessellation evaluation shader:

```
layout( quads ) in;
out vec3 TENormal;   // Vertex normal in camera coords.
out vec4 TEPosition; // Vertex position in camera coords
```

```
uniform mat4 MVP;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;

void basisFunctions(out float[4] b, out float[4] db, float t)
{
    float t1 = (1.0 - t);
    float t12 = t1 * t1;

    // Bernstein polynomials
    b[0] = t12 * t1;
    b[1] = 3.0 * t12 * t;
    b[2] = 3.0 * t1 * t * t;
    b[3] = t * t * t;

    // Derivatives
    db[0] = -3.0 * t1 * t1;
    db[1] = -6.0 * t * t1 + 3.0 * t12;
    db[2] = -3.0 * t * t + 6.0 * t * t1;
    db[3] = 3.0 * t * t;
}

void main()
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;

    // The sixteen control points
    vec4 p00 = gl_in[0].gl_Position;
    vec4 p01 = gl_in[1].gl_Position;
    vec4 p02 = gl_in[2].gl_Position;
    vec4 p03 = gl_in[3].gl_Position;
    vec4 p10 = gl_in[4].gl_Position;
    vec4 p11 = gl_in[5].gl_Position;
    vec4 p12 = gl_in[6].gl_Position;
    vec4 p13 = gl_in[7].gl_Position;
    vec4 p20 = gl_in[8].gl_Position;
    vec4 p21 = gl_in[9].gl_Position;
    vec4 p22 = gl_in[10].gl_Position;
    vec4 p23 = gl_in[11].gl_Position;
    vec4 p30 = gl_in[12].gl_Position;
    vec4 p31 = gl_in[13].gl_Position;
    vec4 p32 = gl_in[14].gl_Position;
    vec4 p33 = gl_in[15].gl_Position;
    // Compute basis functions
    float bu[4], bv[4];   // Basis functions for u and v
    float dbu[4], dbv[4]; // Derivitives for u and v
```

```
    basisFunctions(bu, dbu, u);
    basisFunctions(bv, dbv, v);

    // Bezier interpolation
    TEPosition =
     p00*bu[0]*bv[0] + p01*bu[0]*bv[1] + p02*bu[0]*bv[2] +
     p03*bu[0]*bv[3] +
     p10*bu[1]*bv[0] + p11*bu[1]*bv[1] + p12*bu[1]*bv[2] +
     p13*bu[1]*bv[3] +
     p20*bu[2]*bv[0] + p21*bu[2]*bv[1] + p22*bu[2]*bv[2] +
     p23*bu[2]*bv[3] +
     p30*bu[3]*bv[0] + p31*bu[3]*bv[1] + p32*bu[3]*bv[2] +
     p33*bu[3]*bv[3];

    // The partial derivatives
    vec4 du =
     p00*dbu[0]*bv[0]+p01*dbu[0]*bv[1]+p02*dbu[0]*bv[2]+
     p03*dbu[0]*bv[3]+
     p10*dbu[1]*bv[0]+p11*dbu[1]*bv[1]+p12*dbu[1]*bv[2]+
     p13*dbu[1]*bv[3]+
     p20*dbu[2]*bv[0]+p21*dbu[2]*bv[1]+p22*dbu[2]*bv[2]+
     p23*dbu[2]*bv[3]+
     p30*dbu[3]*bv[0]+p31*dbu[3]*bv[1]+p32*dbu[3]*bv[2]+
     p33*dbu[3]*bv[3];

    vec4 dv =
     p00*bu[0]*dbv[0]+p01*bu[0]*dbv[1]+p02*bu[0]*dbv[2]+
     p03*bu[0]*dbv[3]+
     p10*bu[1]*dbv[0]+p11*bu[1]*dbv[1]+p12*bu[1]*dbv[2]+
     p13*bu[1]*dbv[3]+
     p20*bu[2]*dbv[0]+p21*bu[2]*dbv[1]+p22*bu[2]*dbv[2]+
     p23*bu[2]*dbv[3]+
     p30*bu[3]*dbv[0]+p31*bu[3]*dbv[1]+p32*bu[3]*dbv[2]+
     p33*bu[3]*dbv[3];

    // The normal is the cross product of the partials
    vec3 n = normalize( cross(du.xyz, dv.xyz) );

    // Transform to clip coordinates
    gl_Position = MVP * TEPosition;

    // Convert to camera coordinates
    TEPosition = ModelViewMatrix * TEPosition;
    TENormal = normalize(NormalMatrix * n);
}
```

4. Implement your favorite shading model within the fragment shader utilizing the output variables from the TES.
5. Render the Bezier control points as a 16-vertex patch primitive. Don't forget to set the number of vertices per patch within the OpenGL application:

```
glPatchParameteri(GL_PATCH_VERTICES, 16);
```

How it works...

The tessellation control shader starts by defining the number of vertices in the patch using the layout directive:

```
layout( vertices=16 ) out;
```

It then simply sets the tessellation levels to the value of `TessLevel`. It passes the vertex position along, without any modification.

The tessellation evaluation shader starts by using a layout directive to indicate the type of tessellation to be used. As we are tessellating a 4 x 4 Bezier surface patch, quad tessellation makes the most sense.

The `basisFunctions` function evaluates the Bernstein polynomials and their derivatives for a given value of the parameter `t`. The results are returned in the output parameters `b` and `db`.

Within the `main` function, we start by assigning the tessellation coordinates to variables `u` and `v`, and reassigning all 16 of the patch vertices to variables with shorter names (to shorten the code that appears later).

We then call `basisFunctions` to compute the Bernstein polynomials and their derivatives at `u` and at `v`, storing the results in `bu`, `dbu`, `bv`, and `dbv`.

The next step is the evaluation of the sums from the preceding equations for the position (`TEPosition`), the partial derivative with respect to `u` (`du`), and the partial derivative with respect to `v` (`dv`).

We compute the normal vector as the cross product of `du` and `dv`.

Finally, we convert the position (`TEPosition`) to clip coordinates and assign the result to `gl_Position`. We also convert it to camera coordinates before it is passed along to the fragment shader.

The normal vector is converted to camera coordinates by multiplying with the `NormalMatrix`, and the result is normalized and passed along to the fragment shader via `TENormal`.

See also

- The *Tessellating a 2D quad* recipe