We can do a similar interpolation on all the edges. The normal at any interior point can be obtained from points on the edges by

$$\mathbf{n}(\alpha, \beta) = (1 - \beta)\mathbf{n}_C + \beta\mathbf{n}_D.$$

Once we have the normal at each point, we can make an independent shading calculation. Usually, this process can be combined with rasterization of the polygon. Until recently, Phong shading could only be carried out off-line because it requires the interpolation of normals across each polygon. In terms of the pipeline, Phong shading requires that the lighting model be applied to each fragment, hence, the name **per-fragment shading**. We will implement Phong shading through a fragment shader.

6.6 APPROXIMATION OF A SPHERE BY RECURSIVE SUBDIVISION

We have used the sphere as an example curved surface to illustrate shading calculations. However, the sphere is not an object supported withinWebGL, so we will generate approximations to a sphere using triangles through a process known as **recursive subdivision**, a technique we introduced in Chapter 2 for constructing the Sierpinski gasket. Recursive subdivision is a powerful technique for generating approximations to curves and surfaces to any desired level of accuracy. The sphere approximation provides a basis for us to write simple programs that illustrate the interactions between shading parameters and polygonal approximations to curved surfaces.

Our starting point is a tetrahedron, although we could start with any regular polyhedron whose facets could be divided initially into triangles.[2] The regular tetrahedron is composed of four equilateral triangles, determined by four vertices. We start with the four vertices $(0, 0, 1)$, $(0, 2\sqrt{2}/3, -1/3)$, $(-\sqrt{6}/3, -\sqrt{2}/3, -1/3)$, and $(\sqrt{6}/3, -\sqrt{2}/3, -1/3)$. All four lie on the unit sphere, centered at the origin. (Exercise 6.6 suggests one method for finding these points.)

We get a first approximation by drawing a wireframe for the tetrahedron. We specify the four vertices by

```
var va = vec4(0.0, 0.0, -1.0, 1);
var vb = vec4(0.0, 0.942809, 0.333333, 1);
var vc = vec4(-0.816497, -0.471405, 0.333333, 1);
var vd = vec4(0.816497, -0.471405, 0.333333, 1);
```

These are the same points we used in Chapter 2 to draw the three-dimensional Sierpinski gasket. The order of vertices obeys the right-hand rule, so we can convert the code to draw shaded polygons with little difficulty. If we add the usual code for initialization, setting up a vertex buffer object and drawing the array, our program will generate an image such as that in Figure 6.32: a simple regular polyhedron, but a poor approximation to a sphere.

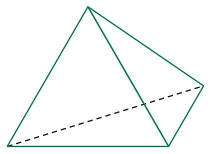
FIGURE 6.32 Tetrahedron.

2. The regular icosahedron is composed of 20 equilateral triangles. It makes a nice starting point for generating spheres.

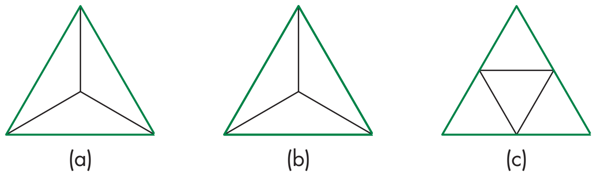

FIGURE 6.33 Subdivision of a triangle by (a) bisecting angles, (b) computing the centroid, and (c) bisecting sides.

We can get a closer approximation to the sphere by subdividing each facet of the tetrahedron into smaller triangles. Subdividing into triangles will ensure that all the new facets will be flat. There are at least three ways to do the subdivision, as shown in Figure 6.33. We can bisect each of the angles of the triangle and draw the three bisectors, which meet at a common point, thus generating three new triangles. We can also compute the center of mass (centroid) of the vertices by simply averaging them and then draw lines from this point to the three vertices, again generating three triangles. However, these techniques do not preserve the equilateral triangles that make up the regular tetrahedron. Instead—recalling a construction for the Sierpinski gasket of Chapter 2—we can connect the bisectors of the sides of the triangle, forming four equilateral triangles, as shown in Figure 6.33(c). We use this technique for our example.

There are two main differences between the gasket program and the sphere program. First, when we subdivide a face of the tetrahedron, we do not throw away the middle triangle formed from the three bisectors of the sides. Second, although the vertices of a triangle lie on the circle, the bisector of a line segment joining any two of these vertices does not. We can push the bisectors to lie on the unit circle by normalizing their representations to have a unit length.

We initiate the recursion by

```
tetrahedron(va, vb, vc, vd, numTimesToSubdivide);
```

which divides the four sides,

```
function tetrahedron(a, b, c, d, n)
{
  divideTriangle(a, b, c, n);
  divideTriangle(d, c, b, n);
  divideTriangle(a, d, b, n);
  divideTriangle(a, c, d, n);
}
```

with the midpoint subdivision:

```
function divideTriangle(a, b, c, count)
{
  if (count > 0) {
    var ab = normalize(mix(a, b, 0.5), true);
    var ac = normalize(mix(a, c, 0.5), true);
    var bc = normalize(mix(b, c, 0.5), true);
    divideTriangle(a, ab, ac, count - 1);
    divideTriangle(ab, b, bc, count - 1);
    divideTriangle(bc, c, ac, count - 1);
    divideTriangle(ab, bc, ac, count - 1);
  }
  else {
    triangle(a, b, c);
  }
}
```

Note that we use the `mix` function to find the midpoint of two vertices. The `true` parameter in `normalize` indicates that we are normalizing the homogeneous representation of a point and that the fourth component of 1 should not be used in the normalization. The triangle function adds the vertex positions to a vertex array:

```
function triangle(a, b, c){
  pointsArray.push(a);
  pointsArray.push(a);
  pointsArray.push(a);
  index += 3;
}
```

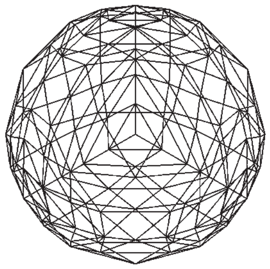
FIGURE 6.34 Sphere approximations using subdivision.

Figure 6.34 shows an approximation to the sphere drawn with this code. We now add lighting and shading to our sphere approximation.

6.7 SPECIFYING LIGHTING PARAMETERS

For many years, the Blinn-Phong lighting model was the standard in computer graphics. It was implemented in hardware and was specified as part of the OpenGL fixed functionality pipeline. With programmable shaders, we are free to implement other lighting models. We can also choose where to apply a lighting model: in the application, in the vertex shader, or in the fragment shader. Consequently, we must define a group of lighting and material parameters and then either use them in the application code or send them to the shaders.

6.7.1 Light Sources

In Section 6.2, we introduced four types of light sources: ambient, point, spotlight, and distant. However, because spotlights and distant light sources can be derived from a point source, we will focus on point sources and ambient light. An ideal point source emits light uniformly in all directions. To obtain a spotlight from a point source, we need only limit the directions of the point source and make the light emissions follow a desired profile. To get a distant source from a point source, we need to allow the location of the source to go to infinity so the position of the source becomes the direction of the source. Note that this argument is similar to the argument that parallel viewing is the limit of perspective viewing as the center of projection moves to infinity. As we argued in deriving the equations for parallel projections, we will find it easier to derive the equations for lighting with distant sources directly rather than by taking limits.