An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 20 Shaders of Lighting

20.1 Lighting Models

We have discussed that the Phong lighting model is a **local illumination** model, in which the shaded color or the illumination at a point of a surface depends purely on the local surface configuration such as the surface normal, the viewing direction, and the light direction. The model only considers light rays originated from point sources and ignores illumination due to second bounces of rays. Also, the model does not consider occlusion of light rays and thus it won't create any shadows in a scene. The main advantage of the model is its simplicity and efficiency in computation.

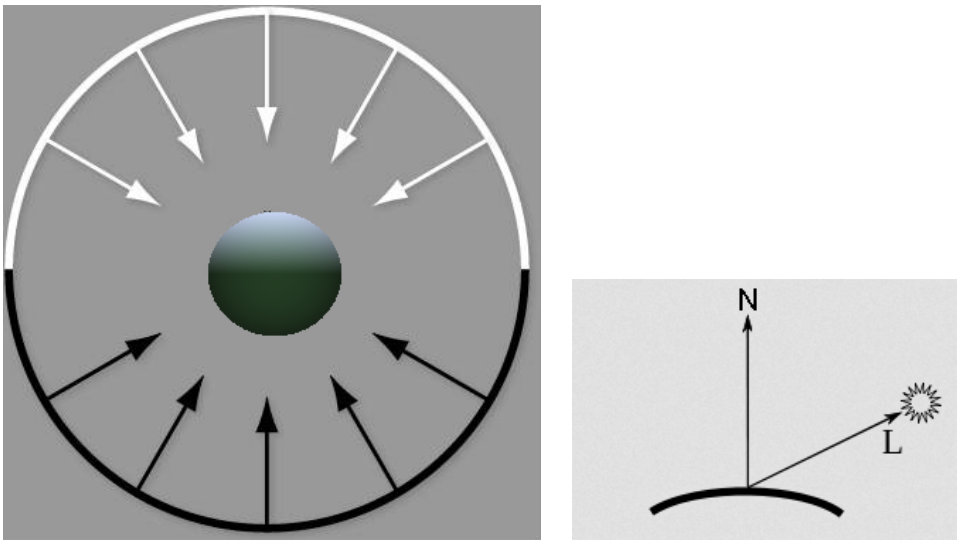
The ray tracing technique discussed in Chapter 16 is a **global illumination** model. It is a sophisticated technique that considers multiple bounces of light rays, casting shadows and producing lighting effects resembling real scenes. However, it is computing intensive and many real-time games may not be able to use the method.

In this chapter, we consider some simple and popular lighting models that approximate global lighting effects. We implement the models using the OpenGL Shading Language (glsl).

20.2 Hemisphere Lighting

Hemisphere lighting can give additional approximation of ambient or diffuse lighting in scenes that have two distinct lighting colors. For example, a scene consisting of a blue sky and green mountains can be illuminated by blue color light from above and and green color light from below. The Phong model only uses fixed colors and is not able to produce such effect.

Figure 20-1 below shows the hemisphere lighting model. The surface of an object receives light from an upper hemisphere and a lower hemisphere as shown in Figure 20-1(a).



(a) (b)

Figure 20-1 Hemishpere Lighting Model

In the model, we assume that each point of a hemisphere around a surface acts as a light source in the direction of **L** (a unit vector) as shown in Figure 20-1(b) (see also Figure 19-14). The unit normal at the illuminated point is **N**. The illumination at an area from each point source is proportional to

$$F(\mathbf{N}, \mathbf{L}) = max(0, \mathbf{N} \cdot \mathbf{L}) \tag{20.1}$$

Therefore, an area with its normal pointing straight up receives all its illumination from the upper hemisphere as $\mathbf{N} \cdot \mathbf{L}$ is negative for all light sources on the lower hemisphere. Similarly, an area with its normal pointing straight down receives all the illumination from the lower hemisphere. For areas with other surface normals, the illumination on the area is a mixture of illuminations from the upper and lower hemispheres.

Suppose **U** is the unit vector that points in the straight up direction (Figure 20-2b), and f is the fraction of illumination on a surface area due to the upper hemisphere. The fraction f is proportional to the integration of the contribution of each point source over the the hemisphere. In spherical coordinates (see Figure 19-10 of Chapter 19), an infinitesimal area is given by $dA = (rsin\theta d\phi)(rd\theta)$ (Figure 20-2a). For the upper hemisphere, we need to integrate over θ from 0 to π, and ϕ from 0 to π. So

$$f \propto \int_0^{\pi} \int_0^{\pi} F(\mathbf{N}, \mathbf{L})(rsin\theta d\phi)(rd\theta) \tag{20.2}$$

That is,

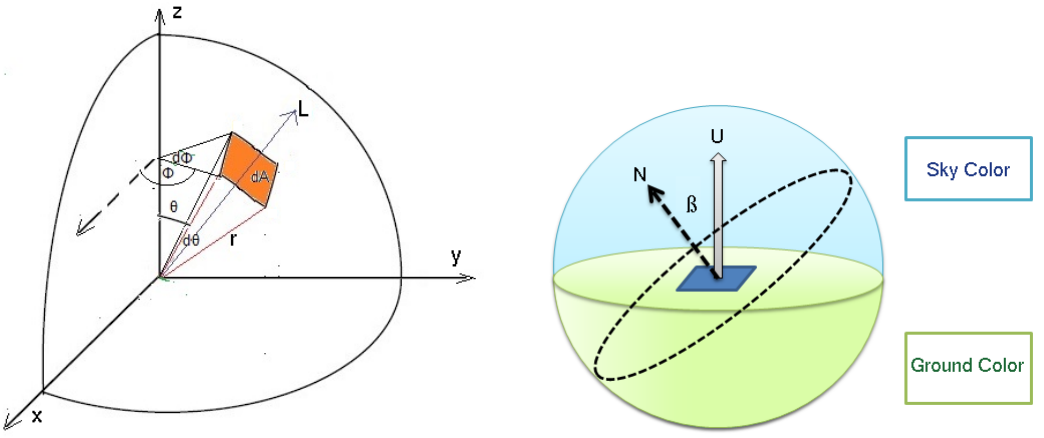
$$f = c \times r^2 \int_0^{\pi} \int_0^{\pi} F(\mathbf{N}, \mathbf{L}) sin\theta d\phi d\theta \tag{20.3}$$

where c is a constant. When **N** is pointing straight up, in the same direction as **U**, f is 1 as there is only contribution from the upper hemisphere, and $F = \mathbf{N} \cdot \mathbf{L} = sin\theta sin\phi$. Therefore,

$$\begin{aligned} 1 &= c \times r^2 \textstyle\int_0^{\pi} \int_0^{\pi} sin^2\theta sin\phi d\phi d\theta \\ &= c \times r^2 \pi \end{aligned} \tag{20.4}$$

Thus $c = 1/(\pi r^2)$. Substituting this back to (20.3), we have

$$f = \frac{1}{\pi} \int_0^{\pi} \int_0^{\pi} F(\mathbf{N}, \mathbf{L}) sin\theta d\phi d\theta \tag{20.5}$$



(a) (b)

Figure 20-2 Integration Over Upper Hemisphere (y-axis is up)

If **N** is not in the same direction as **U**, and the angle between them is β like that shown in Figure 20-2b, then illumination of the upper hemisphere is only from a spherical wedge with ϕ ranging from β to π. The faction f is calculated by integrating over ϕ from β to π:

$$\begin{aligned} f &= \frac{1}{\pi}\int_0^{\pi}\int_{\beta}^{\pi}(\mathbf{N}\cdot\mathbf{L})sin\theta d\phi d\theta \\ &= \frac{1}{\pi}\int_0^{\pi}\int_{\beta}^{\pi} sin^2\theta sin\phi d\phi d\theta \\ &= \frac{1}{2}(1+cos\beta) \\ &= \frac{1}{2}(1+\mathbf{N}\cdot\mathbf{U}) \end{aligned} \tag{20.6}$$

The value of f represents the contribution of the upper hemisphere, so the contribution from the lower hemisphere is $1-f$. If C_{sky} and C_{ground} are the illumination from the the upper and lower hemispheres respectively, the total illumination C at a surface point is

$$\begin{aligned} C &= fC_{sky} + (1-f)C_{ground} \\ &= \frac{1}{2}(1+cos\beta)C_{sky} + \frac{1}{2}(1-cos\beta)C_{ground} \end{aligned} \tag{20.7}$$

Implementation of the shader of this model is straightforward. Suppose our illuminated object is the *Sphere* object we have discussed in Chapter 19; the normal at each point is easy to calculate. The following is the code for the vertex shader and the fragment shader:

Program Listing 20-1 Shaders for Hemisphere Lighting

(a) Vertex Shader: sphereh.vert

```
// sphereh.vert: Source code of vertex shader
uniform mat4 mvpMatrix;
attribute vec4 vPosition;

varying vec3 N; //normal direction

uniform int objType;

void main()
{
   if ( objType == 0 )
     N = vPosition.xyz;   //normal of  a point on sphere
   else
     N = gl_NormalMatrix * gl_Normal;

   gl_Position = mvpMatrix * vPosition;
}
```

(b) Fragment Shader: sphereh.frag

```
// spherel.frag Source code of fragment shader

varying vec3 N;

uniform vec4 skyColor;
uniform vec4 groundColor;
uniform vec3 skyDirection;
```

```
  void main()
  {
    vec3 norm = normalize(N);
    float cosbeta = dot ( norm, skyDirection );
    float f = 0.5 + 0.5 * cosbeta;
    vec4 color = mix (groundColor, skyColor, f);
    gl_FragColor = color;
  }
```

The glsl built-in function **mix** has been used to combine the color contributions from the lower and upper hemispheres using equation (20.7). In the fragment shader, the **uniform** variables *skyColor, groundColor*, and *skyDirection* are passed from the application. The code is simple and efficient, appropriate to be used in applications for model preview, or in conjunction with traditional graphics lighting models. Hemisphere lighting can also be superimposed with directional or spot lights to provide more detailed illumination to certain parts of the scene. Figure 20-3 shows a sample output of the code, where *skyColor, groundColor*, and *skyDirection* are set to $\{0.5, 0.5, 1, 1\}$, $\{0.2, 1, 0.2, 1\}$, and $\{0, 1, 0\}$ respectively and the y-axis is the up direction.

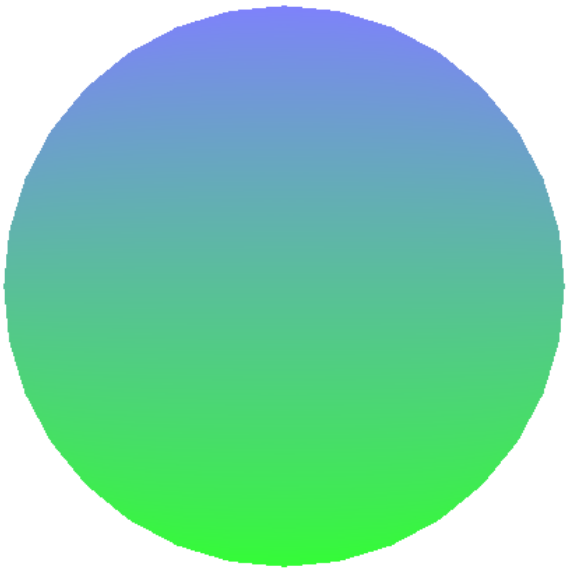


Figure 20-3 Lit Sphere Using Hemisphere Lighting

20.3 Image-Based Lighting (IBL)

Image-based lighting (IBL) is the process of lighting graphical scenes and objects using images of light from the real world. It evolves from the reflection-mapping technique in which panoramic images are used as texture maps to simulate shiny objects reflection of real and synthetic environments.

IBL utilizes an image to simulate lighting in a scene. The image represents omni-directional real world lights and is often captured by a special camera and projected onto a dome or a sphere. The image may contain detailed real-world lighting information saved as an enironment map. The technique has been popularized by research professor Paul Debevec of USC (http://www.pauldebevec.com/), who gives a tutorial on the subject at the site

http://ict.usc.edu/pubs/Image-Based%20Lighting.pdf

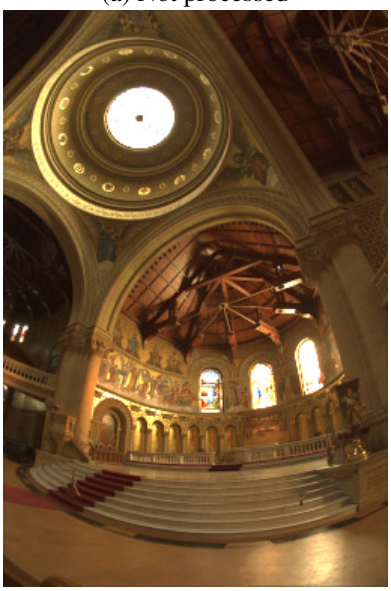
20.3.1 High-Dynamic Range Image

The method often uses a **light probe** to capture the lighting of a real-world scene and saves it as a **high-dynamic range** (HDR) image, usually referred to as a light probe image, which is used to create an environment map, such as a cubemap, where a light probe sphere is projected onto six unwrapped cube faces. One may obtain light probe images from Debevec's web site mentioned above.

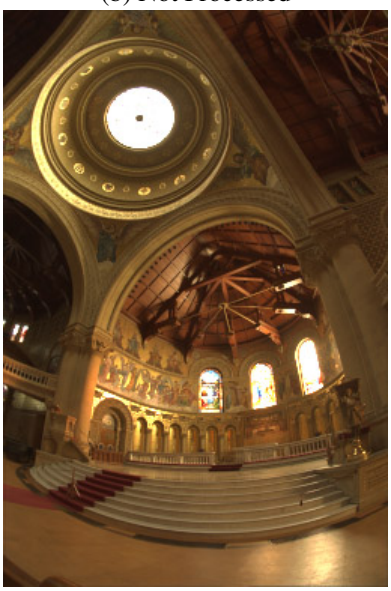
A scene's dynamic range is the contrast ratio of its brightest region to its darkest region. An 8-bit image can have a range of 256 to 1. A high-dynamic range (HDR) image is an image with a greater dynamic range than one that can be shown on a standard display device or captured by a standard camera that uses only one exposure. Its value at a pixel value is proportional to the amount of light in the world region mapped to that pixel, unlike a normal image in which pixel values are nonlinearly encoded. The higher range can be achieved by capturing several different narrower range exposures of the same subject and then combining them to form a single HDR image. Non-HDR cameras take photographs with a limited exposure range, which may result in a loss of highlight or shadow details. To represent a larger range, HDR pixel values are represented by floating-point numbers. On the other hand, traditional low-dynamic range image pixels are usually represented by eight bits per channel, with each pixel value ranging as integers from 0 to 255.

Figure 2-4 compares some images, which are downloaded from the site *http://www.hdrshop.com/*, that are originated from an HDR image and a normal 8-bit image. As shown in (a) and (b), the two images look the same when they are displayed on a computer screen as each screen pixel channel is of 8 bits. However, as shown in (c) and (d), when the image brigtnesses are reduced by 64, the two images appear significantly different. A tool called *HDRshop* is available at the site for manipulating HDR images.

Normal 8-bit Image
(a) Not processed



HDR Image
(b) Not Processed



(c) Darkened by 64 (b) Darkened by 64

Figure 20-4 Comparing HDR Image with Normal 8-bit Image
(Images downloaded from *http://www.hdrshop.com/*)

More detailly, image-based lighting using HDR images involves a few steps:

1. Capture real-world illumination as an omnidirectional HDR image.
2. Create from the HDR image a representation of the environment such as an environment map.
3. Put the 3D graphical object inside the environment.
4. Simulate lighting from the environment representation of step 2.

However, if we do not have HDR images at hand, we still can apply the technique of image-based lighting to illuminate objects using images as light sources as we do here.

20.3.2 Cubemaps

Cube mapping can be used for environment mapping, where the reflection color at a point of a surface area is looked up from the the *environment* along the area's reflection vector, and the *environment color* is stored in a texture map (see also Figure 20-6 below). In cube mapping, we typically map six images as textures to the six faces of a large cube surrounding a scene. The color that we see along a direction in the environment is represented by a texture pixel on the cube. The GLSL data type **samplerCube** is specifically provided for cube mapping; when shading a point of a scene, we can treat the surface area at the point as a perfect reflecting surface and get the environment data by calculating the reflection direction of the viewing direction. Another popular application of cube mapping is for creating sky boxes, which provide the illusion of 3-dimensional backgrounds looked like a sky; in the process a big box is created to encase the camera as it moves around.

In OpenGL, we can create a *cubemap*, which is a texture, the way we create any other textures. We generate a texture and bind it to the target GL_TEXTURE_CUBE_MAP:

```
int texName;
glGenTextures(1, &texName);
glBindTexture(GL_TEXTURE_CUBE_MAP, texName);
```

As a cube has 6 faces, we need to provide 6 images, one for each face, to be used as textures. To simplify the task, OpenGL provides 6 special texture targets for mapping an image to a face of the cube:

Texture target	Orientation
GL_TEXTURE_CUBE_MAP_POSITIVE_X	Right
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	Left
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	Top
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	Bottom
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	Front
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	Back

The texture targets are of **enum** data type and their values are linearly incremented by 1. So we could loop through all texture targets by starting from GL_TEXTURE_CUBE_MAP_POSITIVE_X and incrementing the **enum** texture target by 1 in each iteration, like the following:

```
char *cubeMapImages[6] =
      {"right.png", "left.png", "top.png", "bottom.png",
                                      "front.png", "back.png"};
for ( int i = 0; i < 6; i++ ) {
   GLubyte *texImage = makeTexImage ( (char *) cubeMapImages[i] );
   glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGBA,
     texImageWidth,texImageHeight,0,GL_RGBA,GL_UNSIGNED_BYTE,texImage);
   delete texImage;
}
```

Or better if we specify the targets explicitly:

```
GLuint targets[] = {
      GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
      GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
      GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z
};
.....
for ( int i = 0; i < 6; i++ ) {
  GLubyte *texImage = makeTexImage ( (char *) cubeMapImages[i] );
  glTexImage2D(targets[i], 0, GL_RGBA, texImageWidth,
              texImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, texImage);
  delete texImage;
}
```

As a cubemap is a texture, we could also specify its filtering and wrapping methods like what we do to other textures:

```
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_S,GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_T,GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_R,GL_CLAMP_TO_EDGE);
```

In the code, (S, T, R) represents a texel's 3-dimensional coordinates. (Because the images are mapped to a cube, the texture is 3-dimensional.) So GL_TEXTURE_WRAP_R simply sets the wrapping method for a texel's R coordinate. The wrapping method is set to GL_CLAMP_TO_EDGE so that a texture coordinate is clamped within the range of edges; that is, it is clamped to $[\frac{1}{2N}, 1 - \frac{1}{2N}]$ where N is the size of the texture in the direction of clamping.

GLSL provides the sampler type **samplerCube** to handle cubemaps. A fragment shader that calculates the color at a point due to a cubemap texel would look like the following:

```
uniform samplerCube cubeEnvMap;
varying vec3 envReflectDir;   //texel direction

void main()
{
  vec3 envColor = vec3 (textureCube(cubeEnvMap, envReflectDir));
}
```

Let us consider a very simple example that utilizes cube mapping to shade a sphere. In the application, we use the same *Sphere* class that we have developed in the previous chapter. Instead of using the Phong model, for simplicity, we use a simplified lighting model, in which the illumination at a point on a surface due to a point light source is given by

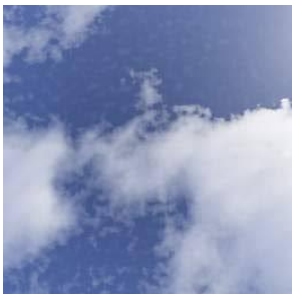
$$I = I^{in} \times F(\mathbf{N}, \mathbf{L}) \tag{20.8}$$

where $F(\mathbf{N}, \mathbf{L})$ is given by (20.1) and the vectors $\mathbf{N}$ and $\mathbf{L}$ are shown in Figure 20-1 (b). The total illumination is a mixture of this illumination and that due to the environment. The sphere is inside the cube so that the environment the sphere sees is the texture images of the cube.

Figure 20-5 shows the 6 images that we will use for the cubemap:


Right (+X)


Left (-X)


Top (+Y)


Bottom (-Y)


Front (+Z)

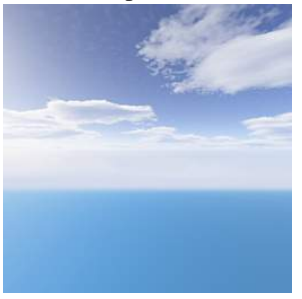

Back (-Z)

Figure 20-5 Images for Cubemap

Our vertex shader and fragment shader are simple. The vertex shader calculates the perfect reflection direction of the viewing vector using the formula

$$\mathbf{r_v} = 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n} - \mathbf{v} \tag{20.9}$$

which is also shown in Equation (16.9) of Chapter 16. Figure 20-6 again shows this relation.

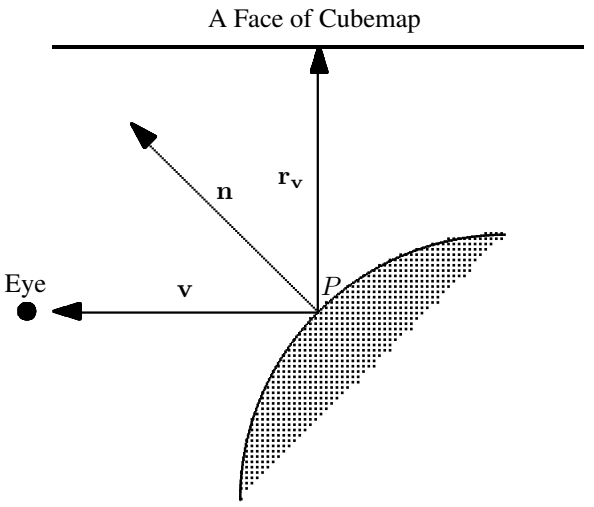


Figure 20-6 Cubemap Viewing and Reflection Vectors

```
   // sphereibl.vert: Source code of vertex shader
   uniform mat4 mvpMatrix;
   attribute vec4 vPosition;
   uniform vec4 eyePosition;
   uniform vec4 lightPosition;
   uniform vec4 lightDiffuse;
   uniform vec4 materialColor;
   varying vec4 lightReflection;
   varying vec3 envReflectDir;   //environment reflection direction

   void main() {
      vec3 N = vPosition.xyz;   //normal of  a point on sphere
                                // (= vPosition.xyz - origin.xyz)
      vec3 L = lightPosition.xyz - vPosition.xyz;
      vec3 V = eyePosition.xyz - vPosition.xyz;

      N = normalize(N);    //unit normal direction
      L = normalize(L);    //unit light source direction
      V = normalize(V);    //unit view vector

      // using a simplified lighting model
      float F = max(0.0, dot(N, L));
      lightReflection =  F * materialColor *  lightDiffuse;
      //environment reflection direction
      envReflectDir = 2.0 *  dot ( N, V) * N - V;
      gl_Position = mvpMatrix * vPosition;
  }
-----------------------------------------------------------------------
   // sphereibl.frag: Source code of fragment shader
   uniform samplerCube cubeEnvMap;
   varying vec4 lightReflection;
   varying vec3 envReflectDir;   //environment reflection direction

   void main()   {
     vec4 envColor = vec4(vec3(textureCube(cubeEnvMap,envReflectDir)),1);
```

```
    float envPercent = 0.7;
    vec4 color = mix ( lightReflection, envColor, envPercent );
    gl_FragColor = color;
}
```

The vertex shader passes the reflection vector to the fragment shader, which utilizes it in the glsl function **textureCube** to fetch the environment color (texel color) along that reflected direction, saving its value in the **vec4** variable *envColor*. We use the formula (20.9) to calculate the reflection vector. (Alternatively, one may use the glsl function **reflect** to calculate it with **-v** and **n** as input parameters.) The function **textureCube** performs a cube mapping operation, in which the (s, t, r) texture coordinates are treated as a direction vector (r_x, r_y, r_z) emanating from the center of the cube, and the texture color at that position is returned. Therefore, when we pass $\mathbf{r_v}$ to **textureCube**, the cube center is at the point P on the sphere (Figure 20-6), which is only an approximation to the real scene.

The **varying vec4** variable *lightReflection* calculates the illumination by the point light source in the vertex shader according to the formula (20.8) and passes its value to the fragment shader. The overall illumination is a combination of this illumination and the environment illumination *envColor*, which are combined by the glsl function **mix** with the weight of *envColor* specified by the variable *envPercent*, which is equal to 0.7 in the example.

Note that as discussed in Chapter 3, a point is not the same as a vector and the difference between two points (positions) is a vector. So the statement

vec3 N = vPosition.xyz;

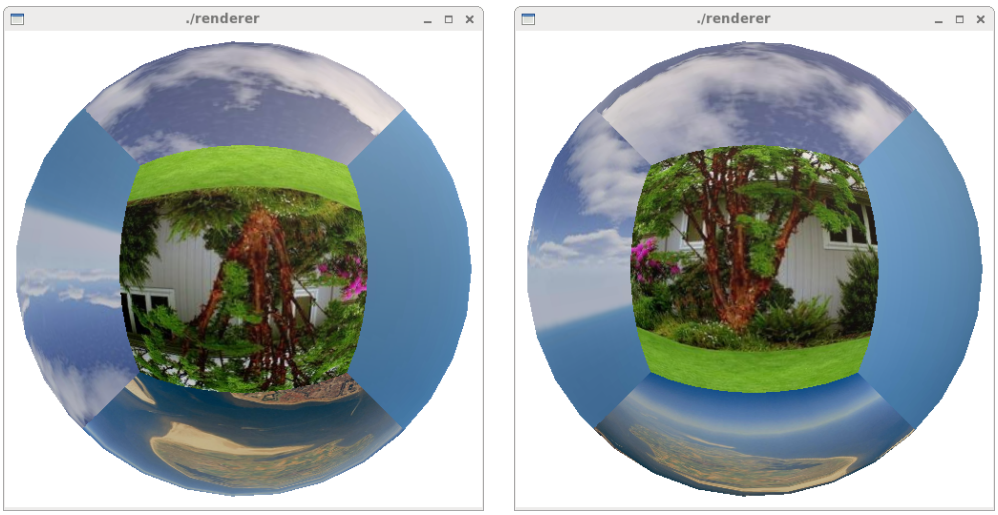
actually means

vec3 N = vPosition.xyz - origin.xyz;

and the origin is at $(0, 0, 0)$. (Also note that a translation transformation changes a point but has no effect on a vector.)

A sample output of this application is shown in Figure 20-7(a) with the following parameters:

```
materialColor[] = {1, 1, 1, 1};
lightDiffuse[] = {1, 0.8, 0.6, 1};
lightPosition[] = {5, 10, 5, 1};
eyePosition[] = {0, 0, 6.3, 1};
```



(a) (b)

Figure 20-7 Outputs of Cubemap Shader

One can see that the mapped images are "upside-down". This is due to the way that **textureCube** maps the textures, which is explained in the official GLSL specifications. A simple way to 'fix' this problem is to rotate all the texture images by 180^o; Figure 20-6(b) shows the output of the shader when the images of Figure 20-5 have been rotated by 180^o.

20.3.3 Cubemaps for Image-based Rendering

Each texel in a cubemap can be regarded as a light source. We can describe static lighting environments by a cubemap, which can be considered as an environment map. In an image-based lighting model, we compute the lighting by an arbitrary number of light sources with a single texture lookup in a cubemap.

Referring to Figure 20-1(b), the diffuse illumination at an area due to a light source is given by

$$I_d = I_d^{in} c_d \times max(0, \mathbf{N} \cdot \mathbf{L}) \tag{20.10}$$

where I_d^{in} is the incoming diffuse illumination, c_d is the diffuse reflection coefficient, and I_d is the resulted diffuse illumination. (This is also the point source light model we have used in the previous section with c_d set to 1.) Suppose the environment is mapped to the six faces of a cube as textures (i.e. cubemap). Then the light sources are the texels of the cube surfaces and the light direction **L** is the direction from the cube center to the texel in the cube map.

For a static cube map, we can speed up the process by precomputing the diffuse lighting for all possible surface normals **N** and storing them in a lookup table. The lighting from a source at a point on a surface with a normal **N** can be found by looking up the diffuse illumination for the specific normal from the lookup table, and the total illumination at the point is the sum of the diffuse illumination of all texels of the cube map. This summed diffuse illumination for a specific normal can be stored in a second cube map, which is usually referred to as the *diffuse irradiance environment map* or *diffuse environment map*. This second cube map is simply a lookup table, where each normal **N** is mapped to a color. The diffuse environment can be created using the graphics tool HDRshop. (Irradiance is the flux of light incident on a unit area, measured in $watts/meter^2$. Our color intensity at a pixel represents the corresponding irradiance.)

The illumination due to specular lights can be similarly treated. The specular illumination due to one point source is:

$$I_s = I_d^{in} c_s \times max(0, \mathbf{r_v} \cdot \mathbf{L})^f \tag{20.11}$$

where c_s is the reflection coefficient, r_v is the reflection direction, and f is the surface shininess. Using this equation, we can compute a cubemap (used as lookup table) that contains the sum of specular illumination from many source for any r_v.

If the cubemaps are given and we use our *Sphere* object as the scene to be illuminated, the shader codes are quite simple. The vertex shader just needs to compute the normal and the reflected view direction at each vertex and sends their values to the fragment shader via varying variables. The values are interpreted across each polygon and the fragment shader utilizes the interpreted values to lookup the illuminations from two cubemaps and combine them using the **mix** function to obtain the resulted illumination:

```
// sphereibl.vert: Source code of vertex shader
uniform mat4 mvpMatrix;
attribute vec4 vPosition;
uniform vec4 eyePosition;
varying vec3 reflectDir;   //environment reflection direction
varying vec3 N;            //normal at a point

void main()
```

```
  {
     N = vPosition.xyz;   //normal of  a point on sphere
                           // (= vPosition.xyz - origin.xyz)
     vec3 V = eyePosition.xyz - vPosition.xyz;

     N = normalize(N);    //unit normal direction
     V = normalize(V);    //unit view vector

     //environment reflection direction
     reflectDir = 2.0 *  dot ( N, V) * N - V;

     gl_Position = mvpMatrix * vPosition;
  }
-------------------------------------------------------------------------
  // sphereibl.frag: Source code of fragment shader
  uniform samplerCube diffuseMap;
  uniform samplerCube specularMap;
  varying vec3 reflectDir;   //environment reflection direction
  varying vec3 N;

  void main()
  {
    float c_d = 0.8, c_s = 0.7;

    vec4 diffuseColor=c_d*vec4(vec3(textureCube(diffuseMap, N)), 1);
    vec4 specularColor = c_s * vec4(vec3(textureCube(specularMap,
                                                  reflectDir)), 1 );

    vec4 color = mix ( diffuseColor, specularColor, 0.5 );

    gl_FragColor = color;
  }
```

20.4 Spherical Harmonic Lighting

20.4.1 Irradiance Environement Map

Spherical Harmonic lighting (SH lighting) computes the diffuse illumination on 3D scenes from broad light sources by expressing light intensities using spherical harmonics. It is based on the observation that the reflected intensity from a diffuse surface varies slowly as a function of surface orientation. Consequently, one can decompose a light source image into spherical harmonic terms so that it can be represented by an analytic expression to reproduce the lighting effects of image-based or environment mapping models without using the actual image source at the runtime.

Ravi Ramamoorthi and Pat Hanrahan of Stanford University showed in 2001 that the irradiance at an area is not sensitive to the high frequency components of the light source and one only needs to compute 9 coefficients of an analytic formula, which correspond to the lowest-frequency modes of the light, to achieve average errors of only 1%. They showed that the irradiance can be procedurally represented explicitly as a quadratic polynomial of the surface normal. Ravi and Pat refer to a diffuse reflection map indexed by the surface normal, as an *irradiance environment map* because the map at each pixel location essentially stores the irradiance for a particular orientation of the surface.

Suppose $L(\theta, \phi) = |\mathbf{L}(\theta, \phi)|$ denotes the remote light intensity distribution from the unit direc-

tion $\mathbf{l}(\theta, \phi) = \frac{\mathbf{L}}{L}$ as shown in Figure 20-2(a). Neglecting the effects of cast shadows and near-field illumination, the irradiance E is then a function of the unit surface normal $\mathbf{n}$ only and is calculated by integrating L over the upper hemisphere $\Omega(\mathbf{n})$:

$$E(\mathbf{n}) = c \times \int_{\Omega(\mathbf{n})} \mathbf{n} \cdot \mathbf{L}(\theta, \phi)\, d\phi d\theta \tag{20.12}$$

where c is a proportional constant. Then the radiosity of a surface, the rate at which energy leaves that surface (energy per unit time per unit area), can be found by multiplying E by ρ, the fraction of incident light that is reflected by the surface, and may depend on position p and be described by a texture:

$$B(p, \mathbf{n}) = \rho(p) E(\mathbf{n}) \tag{20.13}$$

The radiosity B corresponds directly to the brightness at a position of the scene. We want to find an analytic expression, which can be computed efficiently in real time, to approximate E. It turns out that spherical harmonics are good basis functions for this purpose.

20.4.2 Spherical Harmonics

Spherical harmonics, analogous to Fourier series which are a series of functions on a line or a circle, are a series of functions defined on the surface of a sphere, in terms of spherical coordinates. Denoted by $Y_l^m(\theta, \phi)$, spherical harmonics are the angular portion of the solution to *Laplace's equation* with the absence of azimuthal symmetry. In our notation, θ is the polar (colatitudinal) coordinate with $\theta \in [0, \pi]$, and ϕ the azimuthal (longitudinal) coordinate with $\phi \in [0, 2\pi)$. (See Figure 19-9 or Figure 20-2.) Spherical harmonics can be defined as

$$Y_l^m(\theta, \phi) = \left(\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!} \right)^{\frac{1}{2}} P_l^m(cos\theta) e^{im\phi} \tag{20.14}$$

where $P_l^m(x)$ is an *associated Legendre polynomial*, which are orthonormal:

$$P_l^{-m} = (-1)^m \frac{(l-m)!}{(l+m)!} P_l^m \tag{20.15}$$

and

$$\int_{\theta=0}^{\pi} \int_{\phi=0}^{2\pi} Y_l^m \overline{Y}_{l'}^{m'} (sin\theta) d\phi d\theta = \delta_{ll'} \delta_{mm'} \tag{20.16}$$

where $\overline{z}$ denotes the complex conjugate of z, and δ_{ij} is the Kronecker delta. (It equals 1 when $i = j$, and equals 0 when $i \neq j$.) One can also show that normalized spherical harmonic functions satisfy the relation:

$$\overline{Y}_l^m(\theta, \phi) = (-1)^m Y_l^{-m}(\theta, \phi) \tag{20.17}$$

The following list the first few analytic expressions of $Y_l^m(\theta, \phi)$:

$$\begin{aligned}
Y_0^0 &= \frac{1}{2}\left(\frac{1}{\pi}\right)^{\frac{1}{2}}, & Y_1^{-1} &= \frac{1}{2}\left(\frac{3}{2\pi}\right)^{\frac{1}{2}} sin\,\theta\, e^{-i\phi} \\
Y_1^0 &= \frac{1}{2}\left(\frac{3}{\pi}\right)^{\frac{1}{2}} cos\,\theta, & Y_1^1 &= -\frac{1}{2}\left(\frac{3}{2\pi}\right)^{\frac{1}{2}} sin\,\theta\, e^{i\phi} \\
Y_2^{-2} &= \frac{1}{4}\left(\frac{15}{2\pi}\right)^{\frac{1}{2}} sin^2\,\theta\, e^{-i2\phi}, & Y_2^{-1} &= \frac{1}{2}\left(\frac{15}{2\pi}\right)^{\frac{1}{2}} sin\,\theta\, cos\,\theta\, e^{-i\phi} \\
Y_2^0 &= \frac{1}{4}\left(\frac{5}{\pi}\right)^{\frac{1}{2}} (3cos^2\,\theta - 1), & Y_2^1 &= -\frac{1}{2}\left(\frac{15}{2\pi}\right)^{\frac{1}{2}} sin\,\theta\, cos\,\theta\, e^{i\phi} \\
Y_2^2 &= \frac{1}{4}\left(\frac{15}{2\pi}\right)^{\frac{1}{2}} sin^2\,\theta\, e^{i2\phi}, & Y_3^{-3} &= \frac{1}{8}\left(\frac{35}{\pi}\right)^{\frac{1}{2}} sin^3\,\theta\, e^{-i3\phi}
\end{aligned} \tag{20.18}$$

Note that in (20.14), when $m = 0$, the harmonics does not depend on the azimuthal coordinate ϕ. In SH lighting, we use a real basis of spherical harmonics denoted by Y_{lm}, which can be defined in terms of the complex basis denoted by Y_l^m:

$$Y_{lm} = \begin{cases} \sqrt{2}(-1)^m Im[Y_l^{|m|}] & \text{if } m < 0 \\ Y_l^0 & \text{if } m = 0 \\ \sqrt{2}(-1)^m Re[Y_l^m] & \text{if } m > 0 \end{cases} \tag{20.19}$$

The spherical coordinates and Cartesian coordinates are related by:

$$(x, y, z) = (\sin\theta\cos\phi,\ \sin\theta\cos\phi,\ \cos\theta) \tag{20.20}$$

Applying the formulas

$$\begin{aligned} e^{i\phi} &= cos\ \phi + i\ sin\ \phi \\ sin\ 2\phi &= 2sin\ \phi\ cos\ \phi \\ cos\ 2\phi &= cos^2\phi - sin^2\phi \end{aligned}$$

to (20.19) and evaluating the numeric constants, one can obtain the first 9 spherical harmonics in cartesian coordinates:

$$\begin{aligned} Y_{00} &= 0.282095 && \text{constant} \\ (Y_{1-1}; Y_{10}; Y_{11}) &= 0.488603(y; z; x) && \text{linear} \\ (Y_{2-2}; Y_{2-1}; Y_{21}) &= 1.092548(xy; yz; xz) && \text{quadratic} \\ Y_{20} &= 0.315392(3z^2 - 1) && \text{quadratic} \\ Y_{22} &= 0.546274(x^2 - y^2) && \text{quadratic} \end{aligned} \tag{20.21}$$

These 9 spherical harmonics are simply constant ($l = 0$), linear ($l = 1$), and quadratic ($l = 2$) polynomials of the cartesian components (x, y, z).

The spherical harmonics will be used as basis functions, which can be scaled and summed to produce an approximation to a specified function. The process of calculating how much of each basis function contributes to the approximation is referred to as *projection*.

We can decompose the light intensity distribution $L(\theta, \phi)$ and the irradiance $E(\theta, \phi)$ in terms of spherical harmonics with coefficients, L_{lm} and E_{lm}:

$$\begin{aligned} L(\theta, \phi) &= \sum_{l,m} L_{lm} Y_{lm}(\theta, \phi) \\ E(\theta, \phi) &= \sum_{l,m} E_{lm} Y_{lm}(\theta, \phi) \end{aligned} \tag{20.22}$$

We can also expand the dot product $a(\theta) = (\mathbf{n} \cdot \mathbf{l})$ with coefficients a_l. Since a does not depend on ϕ, $m = 0$, so we only use the index l:

$$a(\theta) = max(cos\ \theta, 0) = \sum_l a_l Y_{l0}(\theta) \tag{20.23}$$

One can show that

$$E(\theta, \phi) = \sum_{l,m} A_l L_{lm} Y_{lm}(\theta, \phi) \tag{20.24}$$

where

$$A_l = \left(\frac{4\pi}{2l+1}\right)^{\frac{1}{2}} a_l$$

In fact, $A_l = 0$ for odd values of $l > 1$, and for even values, it falls off rapidly as $l^{-5/2}$. With these, one can derive an analytic formula for A_l:

$$A_l = \begin{cases} \frac{2\pi}{3} & \text{if } l = 1 \\ 0 & \text{if } l > 1, odd \\ 2\pi \frac{(-1)^{\frac{1}{2}-1}}{(l+2)(l-1)} \frac{l!}{2^l \left(\frac{l}{2}!\right)^2} & \text{if } l \; even \end{cases} \tag{20.25}$$

The following are numerical values of the first few terms:

$$\begin{array}{llll} A_0 = 3.141593 & A_1 = 2.094395 & A_2 = 0.785398 & A_3 = 0 \\ A_4 = -0.130900 & A_5 = A_7 = 0 & A_6 = 0.049087 & A_8 = -0.024543 \end{array} \tag{20.26}$$

As one can see from these numerical values, A_l decays exponentially with l, and we just need to consider low-frequency lighting coefficients, of order $l \leq 2$. Practically, approximating the irradiance with only 9 coefficients (1 for $l = 0$, 3 for $l = 1, -1 \leq m \leq 1$, and 5 for $l = 2, -2 \leq m \leq 2$) gives very good results.

The lighting coefficients can be found by integration:

$$L_{lm} = \int_{\theta=0}^{\pi} \int_{\phi=0}^{2\pi} L(\theta, \phi) Y_{lm}(\theta, \phi) (\sin\theta) d\phi d\theta \tag{20.27}$$

The expressions for Y_{lm} are given in (20.21). In the case of environment mapping, the integrals are basically sums of the pixel values in the environment map L, weighted by the values of Y_{lm}, and this can be preprocessed.

20.4.3 Analytical Light Source

Some simple light sources such as directional, disc, and spherical sources can be represented by analytical expressions. For a directional light source, light comes from a single direction. Under this situation, integration against basis functions is reduced to evaluating basis functions in the light direction.

For a disc source, light is originated from a disc object. For instance, consider that the light is from a disc centered around the z-axis as shown in Figure 20-8 on the right. This could be a good approximation to illumination from the sun or the moon. The light intensity distribution $L(\theta, \phi)$ can be described by the equation:

$$L(\theta, \phi) = d_\alpha(\theta, \phi) = \begin{cases} 1 & \text{if } \theta \leq \alpha \\ 0 & \text{if } \theta > \alpha \end{cases} \tag{20.28}$$

This light distribution has rotational symmetry around the z-axis (i.e. $L(\theta, \phi)$ is independent of ϕ). Therefore, it only has non-zero coefficients for modes $m = 0$. We can evaluate numerically the first few non-zero terms using (20.21) and (20.27). For example, using $z = cos\ \theta$, and consider only $l = 1, m = 0$, we have,

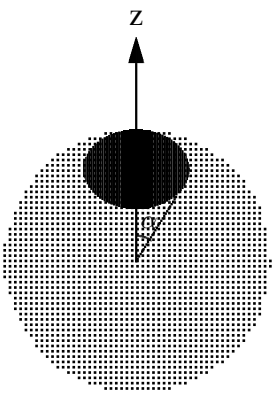


Figure 20-8 Disc Light Source

$$\begin{aligned} L_{10} = & \int_{\theta=0}^{\alpha}\int_{\phi=0}^{2\pi} Y_{10}(\theta,\phi)(\sin\theta)d\phi d\theta \\ = & \int_{\theta=0}^{\alpha}\int_{\phi=0}^{2\pi} 0.488603 \times \cos\theta(\sin\theta)d\phi d\theta \\ = & \ 0.488603 \times 2\pi \int_0^{\alpha} \cos\theta \sin\theta \, d\theta \\ = & \ 1.53499 \sin^2\alpha \end{aligned}$$

We can similarly obtain other terms. The first three non-zero terms are:

$$\begin{aligned} L_{00} = & \ 1.77245(1-\cos\alpha) \\ L_{10} = & \ 1.53499 \sin^2\alpha \\ L_{20} = & \ 1.98166 \ (\cos\alpha - \cos^3\alpha) \end{aligned} \tag{20.29}$$

The irradiance $E(\theta,\phi)$ of (20.24) can be approximated by,

$$\begin{aligned} E(\theta,\phi) = & \ A_0 L_{00} Y_{00} + A_1 L_{10} Y_{10} + A_2 L_{20} Y_{20} \\ = & \ 1.57079(1-cos\alpha) + 1.57080(sin^2\alpha)z + 0.490874(cos\alpha - cos^3\alpha)(3z^2-1) \end{aligned} \tag{20.30}$$

If α is 20^o, this is reduced to

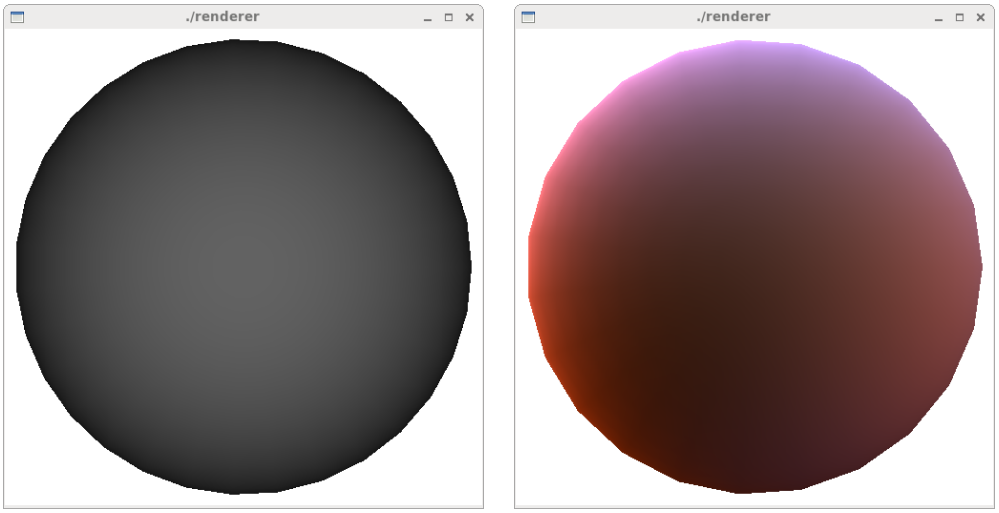
$$E(\theta,\phi) = \ 0.0947305 + 0.183748z + 0.0539584(3z^2-1) \tag{20.31}$$

The following shaders code shows an implementation of (20.30) using our *Sphere* class.

```
// Vertex shader for spherical harmonics lighting with disc source

uniform mat4 mvpMatrix;
attribute vec4 vPosition;
varying vec3  diffuseColor;

const float C0 = 1.57079;
const float C1 = 1.57080;
const float C2 = 0.490874;

void main(void)
{
   vec3 baseColor = vec3 (  1.0, 1.0, 1.0 );
   vec3 N = vPosition.xyz;   //normal for sphere
   N = normalize ( N );
   float z = N.z;

   float alpha = radians ( 20 );   // angle is 20 degrees
   float cs = cos ( alpha );
   float sn = sin ( alpha );
   float intensity = C0*(1.0 - cs) + C1*sn*sn*z +
                        C2 * cs * (1.0 - cs*cs) * (3.0*z*z - 1.0);
   diffuseColor = baseColor * intensity;

   gl_Position = mvpMatrix * vPosition;
}
-------------------------------------------------------------------------
// Fragment shader for spherical harmonics lighting with disc source

varying vec3  diffuseColor;
```

```
void main(void)
{
    gl_FragColor = vec4(diffuseColor, 1.0);
}
```

Figure 20-9(a) shows an output of this shader.



(a) (b)

Figure 20-9 Outputs of Spherical Harmonics Lighting

As we have mentioned, any function constructed using spherical harmonics has rotational symmetry around z-axis only has non-zero coefficients for modes $m = 0$. For the first 5 bands, only 5 out of the 25 coefficients are non-zero. This also makes the operation of rotating the light source more efficient.

For spherical light sources, we may approximate them as disc sources. Suppose the distance of a spherical light center to the origin is d and the sphere radius is r, then the size of the approximated disc can be determined by:

$$\sin \alpha = \frac{r}{d} \tag{20.32}$$

20.4.4 Environment Map Light Source

The lighting due to an environment map is approximated by 9 lighting coefficients, L_{lm} for $l \leq 2$, which can be found by integrating against the spherical harmonic basis functions according to (20.27). Each color channel is treated independently. So we have to do the integration 3 times, one for each of the RGB color values. The integrals are simply sums of the pixel values of the environment map $L(\theta, \phi)$, weighted by the basis functions Y_{lm}. The integrals can be computed using Monte-Carlo integration, a popular statistical numerical technique for evaluating integrals. The computation of the coefficients are preprocessed.

If the size (total number of pixels) of the environment map is S, the preprocessing computation takes $9S$ steps, one for each pixel, as 9 sets of coefficients are computed for each pixel color. This is significantly more efficient that traditional method of computing an irradiance environment map

texture that requires $T \times S$, where T is the number of texels in the irradiance environment map. Table 20-1 below shows the computed RGB values of the coefficients for a few environments, taken from the paper *An Efficient Representation for Irradiance Environment Maps* by Ravi Ramamoorthi and Pat Hanrahan of Stanford University.

Table 20-1 RGB Lighting Coefficients For a Few Environments

	Grace Cathedral			Eucalyptus Grove			St. Peters Basilica		
L_{00}	.79	.44	.54	.38	.43	.45	.36	.26	.23
$L_{1,-1}$	.39	.35	.60	.29	.36	.41	.18	.14	.13
L_{10}	-.34	-.18	-.27	.04	.03	.01	-.02	-.01	-.00
L_{11}	-.29	-.06	.01	-.10	-.10	-.09	.03	.02	.01
$L_{2,-2}$	-.11	-.05	-.12	-.06	-.06	-.04	.02	.01	.00
$L_{2,-1}$	-.26	-.22	-.47	.01	-.01	-.05	-.05	-.03	-.01
L_{20}	-.16	-.09	-.15	-.09	-.13	-.15	-.09	-.08	-.07
L_{21}	.56	.21	.14	-.06	-.05	-.04	.01	.00	.00
L_{22}	.21	-.05	-.30	.02	-.00	-.05	-.08	-.06	.00

We can calculate the irradiance due to an environment map using equation (20.24). We will combine the constants A_l of (20.26) and the numeric coefficients of Y_{lm} given by (20.21) and call the product constant c_i, where i is labeled in a convenient way to organize the terms. For example, c_4 is the product of A_0 and Y_{00}:

$$c_4 = A_0 \times Y_{00} = 3.141593 \times 0.282095 = 0.886227$$

With (20.21) and (20.26), we can express (20.24) as

$$\begin{aligned} E(\mathbf{n}) = \quad & c_1 L_{22}(x^2 - y^2) + c_3 L_{20} z^2 + c_4 L_{00} - c_5 L_{20} + \\ & 2c_1(L_{2,-2}xy + L_{21}xz + L_{2,-1}yz) + 2c_2(L_{11}x + L_{1,-1}y + L_{10}z) \end{aligned} \tag{20.33}$$

where

$$c_1 = 0.429043 \quad c_2 = 0.511664 \quad c_3 = 0.743125$$

$$c_4 = 0.886227 \quad c_5 = 0.247708$$

The following shaders code shows an implementation of the the lighting from the coefficients of *Grace Cathedral* shown in Table 20-1. Our *Sphere* is the illuminated object. Figure 20-9(b) shows an output of the shader.

```
// Vertex shader for SH lighting of Grace Cathedral

uniform mat4 mvpMatrix;
attribute vec4 vPosition;

varying vec3  diffuseColor;

const float c1 = 0.429043;
const float c2 = 0.511664;
const float c3 = 0.743125;
const float c4 = 0.886227;
const float c5 = 0.247708;

// Constants for Grace Cathedral lighting
const vec3 L00  = vec3( 0.79,  0.44,  0.54);
const vec3 L1_1 = vec3( 0.39,  0.35,  0.60);
const vec3 L10  = vec3(-0.34, -0.18, -0.27);
```

```
const vec3 L11  = vec3(-0.29, -0.06,  0.01);
const vec3 L2_2 = vec3(-0.11, -0.05, -0.12);
const vec3 L2_1 = vec3(-0.26, -0.22, -0.47);
const vec3 L20  = vec3(-0.16, -0.09, -0.15);
const vec3 L21  = vec3( 0.56,  0.21,  0.14);
const vec3 L22  = vec3( 0.21, -0.05, -0.30);


void main(void)
{
   vec3 N;
   N = vPosition.xyz;  //normal of  a point on sphere
   N = normalize( N ); //normalize

   float x = N.x, y = N.y, z = N.z;

   diffuseColor = c1 * L22 * (x*x - y*y) +
                  c3 * L20 * z*z +
                  c4 * L00 - c5 * L20 +
                  2.0 * c1 * (L2_2 * x*y + L21 * x*z + L2_1 * y*z) +
                  2.0 * c2 * (L11  * x + L1_1 * y + L10  * z);

     gl_Position = mvpMatrix * vPosition;
}
-----------------------------------------------------------------------
// Fragment shader for SH lighting of Grace Cathedral

varying vec3  diffuseColor;

void main(void)
{
    gl_FragColor = vec4(diffuseColor, 1.0);
}
```

The calculations are done in the vertex shader, which passes the color at each vertex via the **varying** variable *diffuseColor* to the fragment shader. The fragment shader simply takes the colors at vertices calculated by the vertex shader and renders it. For pixels not at any vertice of any ploygon, interpolation is used to obtain the color during rasterization. This is justified for diffuse reflection, which typically varies slowly across the scene.

20.5 Lighting of Cinematography

20.5.1 Introduction

Lighting on cinematography can contribute to the storytelling, mood, image composition, and special effects. For cinematography, physical light sources, such as desk or ceiling lamps, are rarely major contributors to the illumination. Instead, illumination effects are often created by placing various types of lamps and spotlights off-camera, and the lighting does not need to be restricted by physical laws. To achieve special effects, a lighting designer often employs whatever necessary tricks and cheats such as suspending a cloth in front of a light to soften shadows, positioning opaque cards or graded filters to shape a light, or focusing a narrow *tickler* light to get extra highlight.

Ronen Barzel of Pixar Animation Studios published a paper in *Journal of Graphics Tools* in 1997, presenting a lighting model, which was developed over several years in response to the needs of computer graphics film production, in particular for making the movie *Toy Story*. The model gives lighting designers control over the shape, placement, and texture of lights, so that their real-world cinematographic talent can be applied to computer images. The model does not emphasize on real physically simulating tools but on taking advantage of various sorts of imaginary art work available. The model became known as *überlight* model and its *RenderMan* implementation known as *überlight* shader.

20.5.2 Superellipse

The $\ddot{u}$berlight model uses a pair of superellipses to define the shape of light and to model a gradual dropoff of light at the edge of the shape from full to zero intensity.

A superellipse, first discussed by $Lam\dot{e}$ in 1818, is a curve described by the Cartesian equation:

$$\left(\frac{x}{a}\right)^{\frac{2}{d}} + \left(\frac{y}{b}\right)^{\frac{2}{d}} = 1 \tag{20.34}$$

where d, a, and b are positive numbers. This formula describes a closed curve contained in the rectangle $-a \leq x \leq a$ and $-b \leq y \leq b$. The variable d can be regarded as a "roundness" parameter varying the shape from pure ellipse when $d = 1$ to a pure rectangle as $d \to 0$. Figure 20-10 shows superellipses for various d values.

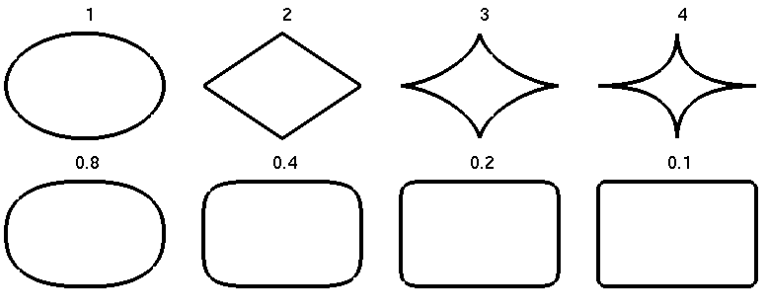


Figure 20-10 Superellipses with Various d values

A superellipse may be described parametrically by:

$$\left.\begin{aligned} x(\theta) &= \pm a \cos^d\theta \\ y(\theta) &= \pm b \sin^d\theta \end{aligned}\right\} \quad 0 \leq \theta < \frac{\pi}{2} \tag{20.35}$$

We can use two nested superellipses with radii a, b and A, B as shown in Figuure 20-11 to specify a shape, at which light intensity transitions smoothly. Given a point P, we compute a clip factor, which has a value of 1 if P lies within the inner superellipse, and 0 if lies outside the outer superellipse, and varies smoothly from 0 to 1 for points between the superellipses.

20.5.3 Step Transitions

To achieve soft-cliping, we need a step function to provide control values between 0 and 1 for a variable s. The function varies slowly from 0 to 1 when s changes from c to d. It returns 0 if $s < c$ and 1 if $s > d$. Then 1 minus the function gives us the desired clipping factor.

A linear step function gives these effects and is described by:

$$\text{linearstep}(c,d,s) = \begin{cases} 0 & s \le c \\ \frac{s-c}{c-d} & c \le s \le d \\ 1 & s \ge d \end{cases} \tag{20.36}$$

The function **linearstep** given in (20.36) and shown on the left of Figure 20-12, is simple and fast to compute. However, it is not C^1-continuous as it changes abruptly at the boundaries. It can causing Mach banding, a physical illusion named after the physicist Ernest Mach. It exaggerates contrast between edges of slightly different shades of gray.

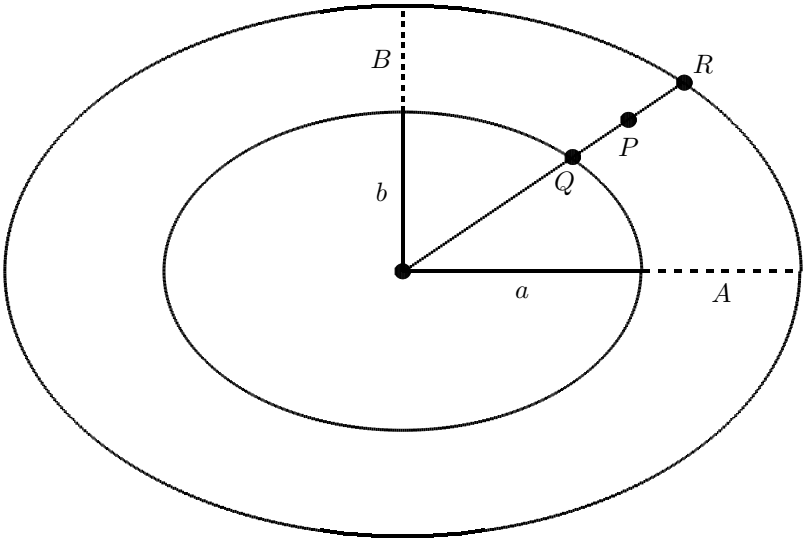


Figure 20-11 Nested Superellipses

A better function to use is a smooth step function, where the interpreting function is defined by a cubic curve so that the tangent is horizontal at the start and the end, and the curve is C^1-continuous. (i.e. Its first derivative at every point exists and is continuous.) A commonly used curved is a special Hermite Spline, a cubic spline defined by two interpolating points and two tangents. Such a step function is shown on the right of Figure 20-12 and can be described by:

$$\text{smoothstep}(c,d,s) = \begin{cases} 0 & s \le c \\ 3t^2 - 2t^3 & c \le s \le d \\ 1 & s \ge d \end{cases} \tag{20.37}$$

where

$$t = \frac{s-c}{c-d} \tag{20.38}$$

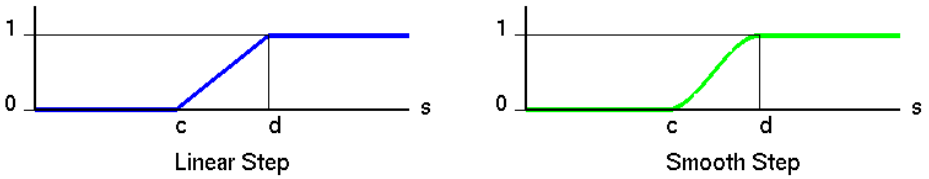


Figure 20-12 Linear and Smooth Step Functions

The **smoothstep** function is supported by the OpenGL shading language and is also in the *RenderMan* liabrary. The glsl function **smoothstep**() works exactly as described in (20.37).

To evaluate the step function, we need to first compute the parameters c, d, and s of (20.37). Referring to Figure 20-11, given a point P, we express the ray through P, and originating from the origin $O = (0, 0)$ as $L(s) = O + s(P - O) = sP$. Suppose the ray intersects the inner superellipse at Q, and the outer at R. To find Q, and R, we can express the points as:

$$P = pP, \quad Q = qP, \quad R = rR \tag{20.39}$$

Trivially, $p = 1$. Assuming $P = (x, y)$, we can compute q by:

$$\begin{aligned} 1 &= \left(\frac{qx}{a}\right)^{\frac{2}{d}} + \left(\frac{qy}{b}\right)^{\frac{2}{d}} \\ &= q^{\frac{2}{d}} \left(\left(\frac{x}{a}\right)^{\frac{2}{d}} + \left(\frac{y}{b}\right)^{\frac{2}{d}} \right) \end{aligned} \tag{20.40}$$

from which, we can solve for q as:

$$q = ab \left((bx)^{\frac{2}{d}} + (ay)^{\frac{2}{d}} \right)^{\frac{-d}{2}} \tag{20.41}$$

We can similarly obtain an expression for r by replacing a and b in (20.41) by A and B respectively:

$$r = AB \left((Bx)^{\frac{2}{d}} + (Ay)^{\frac{2}{d}} \right)^{\frac{-d}{2}} \tag{20.42}$$

So the final clip factor is given by $1 - smoothstep(q, r, 1)$.

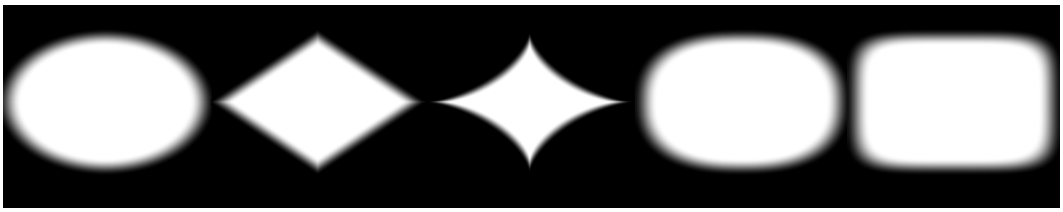


Figure 20-13 Nested Superellipses Shading with Smooth Step (left to right: $d = 1, 2, 3, 0.8, 0.4$)

If the center of the light beam is on the z-axis and the beam is aimimg along the z-axis, the illumination across a cross section of the beam can be implemented using (20.41) and (20.42) as follows in a fragment shader:

```
uniform float a;              //inner superellipse width
uniform float b;              //inner superellipse height
uniform float A;              //outer superellipse width
uniform float B;              //outer superellipse hight
uniform float d;              //roundness of shape

float superEllipseShape(vec3 pos)
{
  // Project point onto z = 1.0 plane; consider absolute values
  float x = abs(pos.x/pos.z);
  float y = abs(pos.y/pos.z);
  float e1 = 2.0 / d, e2 = -d / 2.0;
```

```
    float q = a * b * pow(pow(b * x, e1) + pow(a * y, e1), e2);
    float r = A * B * pow(pow(B * x, e1) + pow(A * y, e1), e2);

    return 1.0 - smoothstep(q, r, 1.0);
}
```

To do such an implementation, we need to consider the relative position and orientation of the light source with respect to the vertex to be shone.

Suppose $P = (x, y, z)$ is the vertex and the light source center is at the point $Q = (x_l, y_l, z_l)$ as shown in Figure 20-14 below. We want to move the origin O of the world coordinate system to the point Q; this is simply a translation by the vector $\mathbf{L} = Q - O = (x_l, y_l, z_l)$. To simplify the calculations, we align the light vector $\mathbf{L} = Q - O$ with the z-axis, so that the shape formed on the object at P is simply the projection of the light source on the rotated $x - y$ plane. To rotate L onto the z-axis, we need to

1. rotate **L** about the x-axis to put **L** on the xz-plane,
2. rotate **L** about the y-axis to align it with the z-axis.

Suppose $\mathbf{L} = (L_x, L_y, L_z)$ is normalized (i.e. $|L| = 1$). The components L_x, L_y, and L_z are the direction cosines of x, y, and z axes respectively. That is, if the angles between **L** and the x, y, and z axes are A, B, and C respectively, then $L_x = cos\ A$, $L_y = cos\ B$, and $L_z = cos\ C$.

For the rotation about the x-axis, we need to find cos α and sin α, where α is the angle between the projection of **L** (in the yz-plane) and the z-axis. Let $d = (L_y^2 + L_z^2)^{\frac{1}{2}} = (1 - L_x^2)^{\frac{1}{2}} = (1 - cos^2 A)^{\frac{1}{2}} = sin\ A$, which is the length of the projection. Then

$$\cos\alpha = \frac{L_z}{d}, \quad \sin\alpha = \frac{L_y}{d} \tag{20.43}$$

(See Figure 20-15 the projection of **L** on y-z plane.)

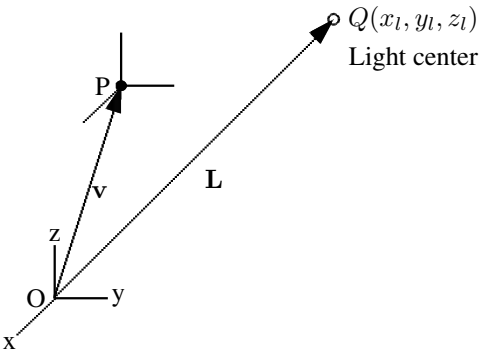


Figure 20-14 Lighting Coordinates

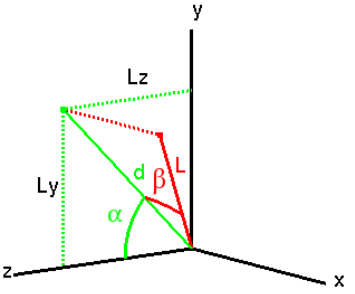


Figure 20-15 Aligning **L** with z-axis

Using (3.32) of Chapter 3, the rotation matrix about x-axis is:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & L_z/d & -L_y/d & 0 \\ 0 & L_y/d & L_z/d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{20.44}$$

For the rotation about the y-axis, we need to rotate the vector for an angle $\beta = -(90^o - A)$. So sin β = - cos $A = -L_x$. Also, cos β = sin A = d. Using (3.30) of Chapter 3, we obtain the rotation

matrix:

$$R_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} d & 0 & -L_x & 0 \\ 0 & 1 & 0 & 0 \\ L_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{20.45}$$

The composite matrix $R = R_y R_x$ will bring the unit vector $\mathbf{L}$ to coincide with the z axis.

Using our *Sphere* class as an example, the following is the complete code of a vertex shader that implements these features with some parameters hard-coded. The rotation matrix can be easily calculated evenly if the light source position is changing with time.

```
-------------------------------------------------------------------------
// Vertex Shader of uberlight-like model
uniform mat4 mvpMatrix;
attribute vec4 vPosition;
uniform vec4 lightPos;      // light position in world coordinates
uniform vec4 eyePos;        // view position in world space

varying vec3 vPosLC;        // vertex position in light coordinates
varying vec3 normLC;        // normal in light coordinates
varying vec3 eyeLC;         // view point in light coordinates

void main()
{
    vec3 L = normalize ( lightPos.xyz );
    float d = sqrt (L.y*L.y + L.z*L.z);
    mat3 Rx, Ry, R;    // rotation matrices

    Rx = mat3 ( 1, 0,     0,          //1st column
                0, L.z/d, L.y/d,      //2nd column
                0, -L.y/d, L.z/d );   //3rd column
    Ry = mat3 ( d,  0, L.x,           //1st column
                0,   1, 0,            //2nd column
                -L.x, 0, d );         //3rd column

    R = Ry * Rx;                  //the composite matrix
    vPosLC = R * (vPosition.xyz - lightPos.xyz);
    eyeLC  = R * (eyePos.xyz - lightPos.xyz);
    normLC = R * vPosition.xyz;

    gl_Position = mvpMatrix * vPosition;
}
-------------------------------------------------------------------------
```

20.5.3 Distance Falloff

Besides the intensity falloff across a beam, which is modeled by nested superellipses, the light intensity also drops off with distance from the light source. To simplify the controls, people often employ near and far distances, also known as cuton and cutoff values to define regions illuminated by the beam as shown in Figure 20-16. Smooth transitions between zones are desired and they can be implemented using the glsl **smoothstep**() function.

This kind of lighting control does not follow any physical laws. It is introduced for lighting designers to create special imaginary effects in cinematograhy.

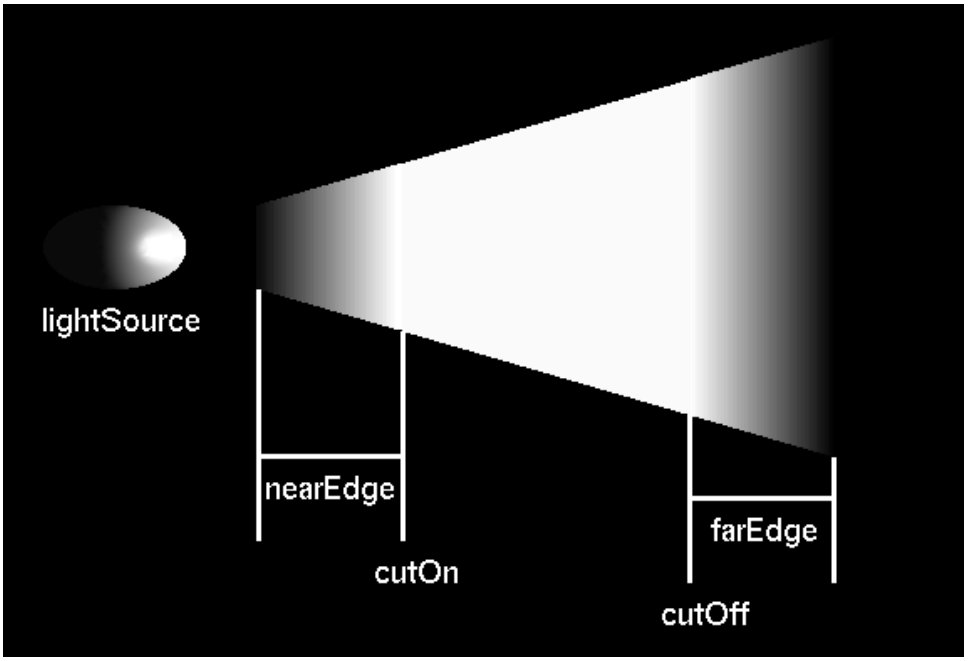


Figure 20-16 Distance Effects of Illumination

Besides zone transitions, the light intensity falls off with distance and is often expressed using the inverse power formula:

$$I(d) = K\left(\frac{d_0}{d}\right)^{\alpha} \tag{20.46}$$

where d is the distance from the light source, α is an attenuation exponent, and K is the desired intensity at the canonical distance d_0. In the expression, I goes to infinity as d approaches 0. A simple solution is to clamp the intensity to a maximum value, but this creates discontinuities, which may cause Mach banding. A better method is use a Guassian-like curve for a distance smaller than the canonical distance:

$$I(d) = \begin{cases} Me^{s\left(\frac{d_0}{d}\right)^{\beta}} & d < d_0 \\ K\left(\frac{d_0}{d}\right)^{\alpha} & d > d_0 \end{cases} \tag{20.47}$$

where $s = ln\left(\frac{K}{M}\right)$ and $\beta = -\alpha/s$ are chosen so that the two expressions for $d < d_0$ and $d > d_0$ give the same value, $I(d_0) = K$ at $d = d_0$ and give the same slope $I'(d) = -K\alpha/d_0$. If $K = 1$, and $\alpha = 1$, then $s = ln(1/M)$ and $\beta = -1/s$.

The following is the fragment shader code that works with the vertex shader presented above; it implements these lighting features. For simplicity and clarity of presentation, we hard-code a number of parameters.

```
-------------------------------------------------------------------------
// Fragment Shader of uberlight-like model

// Super ellipse shaping parameters
uniform float a;             //inner superellipse width
uniform float b;             //inner superellipse height
```

```
uniform float A;             //outer superellipse width
uniform float B;             //outer superellipse hight
uniform float d;             //roundness of shape

// Distance attenuation parameters
uniform float zNear;
uniform float zFar;
uniform float nearEdge;
uniform float farEdge;

varying vec3 vPosLC;         // vertex position in light coordinates
varying vec3 normLC;         // normal in light coordinates
varying vec3 eyeLC;          // view position in light coordinates

float superEllipseShape(vec3 pos)
{
   // Project point onto z = 1.0 plane, and consider absolute values
   float x = abs(pos.x/pos.z);
   float y = abs(pos.y/pos.z);

   float e1 = 2.0 / d, e2 = -d / 2.0;

   float q = a * b * pow(pow(b * x, e1) + pow(a * y, e1), e2);
   float r = A * B * pow(pow(B * x, e1) + pow(A * y, e1), e2);

   return 1.0 - smoothstep(q, r, 1.0);
}

float distanceAttenuation(vec3 pos,float maxIntensity,float d0,float alpha)
{
   float z = abs(pos.z);

   // attenuation in various light zones (Figure 20-16)
   float a =  smoothstep(zNear - nearEdge, zNear, z) *
              (1.0 - smoothstep(zFar, zFar + farEdge, z));

   // attenuation due to distance
   if ( z > d0 )
     a *= pow ( d0/z, alpha );
   else {
     float s = log ( 1.0 / maxIntensity );
     float beta = -alpha / s;
     a *=  maxIntensity * exp ( s * pow(z/d0, beta) );
   }

   return a;
}

void main()
{
   float attenuation;
   attenuation = superEllipseShape(vPosLC) *
                 distanceAttenuation(vPosLC, 2.0, 1.0, 1.0 );

   vec3 N = normalize(normLC);
```

```
    vec3 L = -normalize(vPosLC);
    vec3 V = normalize(eyeLC-vPosLC);
    vec3 H = normalize(L + V);    // half-vector

    vec3 baseColor = vec3(1.0, 1.0, 1.0);
    float shinness = 4.0;
    float NdotL = dot(N, L);
    float NdotH = dot(N, H);

    float diff = max(NdotL, 0.0);
    float spec =  max(NdotH, 0.0);

    vec3 ambient = vec3 (0.5, 0.2, 0.2);
    vec3 diffuse = attenuation * baseColor  * diff;
    vec3 specular = attenuation * baseColor * pow(spec, shinness);

    gl_FragColor = vec4(diffuse + specular + ambient, 1.0);
}
-------------------------------------------------------------------------
```

Figure 20-17 shows an output of this shader with the following parameters provided by the main OpenGL application:

lightPos[]:	{1, 1, 2, 1}	eyePos[]:	0, 0, 6.3, 1
a, b, A, B, d:	0.2, 0.3, 0.3, 0.4, 0.8	zNear, zFar:	1.0, 2.0
nearEdge:	0.4	farEdge:	0.8

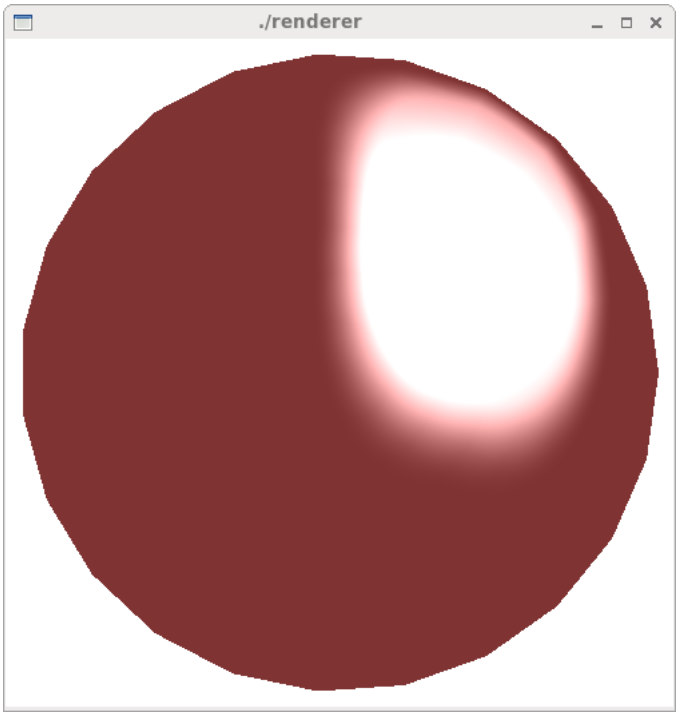


Figure 20-17 Output of uberlight-like shader