Figure 7.9 Spherical harmonics lighting
(Lighting using the coefficients from Table 7.1. From the left: Old Town Square, Grace Cathedral, Galileo's Tomb, Campus Sunset, and St. Peter's Basilica. (3Dlabs, Inc.))

The trade-offs in using image-based lighting versus procedurally defined lights are similar to the trade-offs between using stored textures versus procedural textures. Image-based lighting techniques can capture and re-create complex lighting environments relatively easily. It would be exceedingly difficult to simulate such an environment with a large number of procedural light sources. On the other hand, procedurally defined light sources do not use up texture memory and can easily be modified and animated.

Shadow Mapping

Recent advances in computer graphics have produced a plethora of techniques for rendering realistic lighting and shadows. OpenGL can be used to implement almost any of them. In this section, we will cover one technique known as *shadow mapping*, which uses a *depth texture* to determine whether a point is lit or not.

Shadow mapping is a multipass technique that uses depth textures to provide a solution to rendering shadows. A key pass is to view the scene from the shadow-casting light source rather than from the final viewpoint. By moving the viewpoint to the position of the light source, you will notice that everything seen from that location is lit—there are no shadows from the perspective of the light. By rendering the scene's depth from the point of view of the light into a depth buffer, we can obtain a map of the shadowed and unshadowed points in the scene—a *shadow map*. Those points visible to the light will be rendered, and those points hidden from the light (those in shadow) will be culled away by the depth test. The resulting depth buffer then contains the distance from the light to the closest point to the light for each pixel. It contains nothing for anything in shadow.

The condensed two-pass description is as follows:

- Render the scene from the point of view of the light source. It doesn't matter what the scene looks like; you only want the depth values. Create a shadow map by attaching a depth texture to a framebuffer object and rendering depth directly into it.
- Render the scene from the point of view of the viewer. Project the surface coordinates into the light's reference frame and compare their depths to the depth recorded into the light's depth texture. Fragments that are further from the light than the recorded depth value were not visible to the light, and hence in shadow.

The following sections provide a more detailed discussion, along with sample code illustrating each of the steps.

Creating a Shadow Map

The first step is to create a texture map of depth values as seen from the light's point of view. You create this by rendering the scene with the viewpoint located at the light's position. Before we can render depth into a depth texture, we need to create the depth texture and attach it to a framebuffer object. Example 7.15 shows how to do this. This code is included in the initialization sequence for the application.

Example 7.15 Creating a Framebuffer Object with a Depth Attachment

```
// Create a depth texture
glGenTextures(1, &depth_texture);
glBindTexture(GL_TEXTURE_2D, depth_texture);
// Allocate storage for the texture data
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
             DEPTH_TEXTURE_SIZE, DEPTH_TEXTURE_SIZE,
             0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
// Set the default filtering modes
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Set up depth comparison mode
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
// Set up wrapping modes
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glBindTexture(GL_TEXTURE_2D, 0);

// Create FBO to render depth into
glGenFramebuffers(1, &depth_fbo);
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);
```

```
// Attach the depth texture to it
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
                     depth_texture, 0);
// Disable color rendering as there are no color attachments
glDrawBuffer(GL_NONE);
```

In Example 7.15, the depth texture is created and allocated using the GL_DEPTH_COMPONENT32 internal format. This creates a texture that is capable of being used as the depth buffer for rendering and as a texture that can be used later for reading from. Notice also how we set the *texture comparison mode*. This allows us to leverage *shadow textures*—a feature of OpenGL that allows the comparison between a reference value and a value stored in the texture to be performed by the texture hardware rather than explicitly in the shader. In the example, `DEPTH_TEXTURE_SIZE` has previously been defined to be the desired size for the shadow map. This should generally be at least as big as the default framebuffer (your OpenGL window); otherwise, aliasing and sampling artifacts could be present in the resulting images. However, making the depth texture unnecessarily large will waste lots of memory and bandwidth and adversely affect the performance of your program.

The next step is to render the scene from the point of view of the light. To do this, we create a view-transformation matrix for the light source using the provided `lookat` function. We also need to set the light's projection matrix. As world and eye coordinates for the light's viewpoint, we can multiply these matrices together to provide a single view-projection matrix. In this simple example we can also bake the scene's model matrix into the same matrix (providing a model-view-projection matrix to the light shader). The code to perform these steps is shown in Example 7.16.

Example 7.16 Setting up the Matrices for Shadow Map Generation

```
// Time varying light position
vec3 light_position = vec3(
   sinf(t * 6.0f * 3.141592f) * 300.0f,
   200.0f,
   cosf(t * 4.0f * 3.141592f) * 100.0f + 250.0f);

// Matrices for rendering the scene
mat4 scene_model_matrix = rotate(t * 720.0f, Y);

// Matrices used when rendering from the light's position
mat4 light_view_matrix = lookat(light_position, vec3(0.0f), Y);
mat4 light_projection_matrix(frustum(-1.0f, 1.0f, -1.0f, 1.0f,
                                     1.0f, FRUSTUM_DEPTH));

// Now we render from the light's position into the depth buffer.
// Select the appropriate program
glUseProgram(render_light_prog);
```



```
glUniformMatrix4fv(render_light_uniforms.MVPMatrix,
                   1, GL_FALSE,
                   light_projection_matrix *
                   light_view_matrix *
                   scene_model_matrix);
```

In Example 7.16, we set the light's position using a function of time (`t`) and point it towards the origin. This will cause the shadows to move around. `FRUSTUM_DEPTH` is set to the maximum depth over which the light will influence and represents the far plan of the light's frustum. The near plane is set to `1.0f`, but ideally the ratio of far plane to near plane distance should be as small as possible (i.e., the near plane should be as far as possible from the light and the far plane should be as close as possible to the light) to maximize the precision of the depth buffer.

The shaders used to generate the depth buffer from the light's position are trivial. The vertex shader simply transforms the incoming position by the provided model-view-projection matrix. The fragment shader writes a constant into a dummy output and is only present because OpenGL requires it.[1] The vertex and fragment shaders used to render depth from the light's point of view are shown in Example 7.17.

Example 7.17 Simple Shader for Shadow Map Generation

```
---------------------- Vertex Shader ----------------------
// Vertex shader for shadow map generation

#version 330 core

uniform mat4 MVPMatrix;

layout (location = 0) in vec4 position;

void main(void)
{
    gl_Position = MVPMatrix * position;
}

------------------- Fragment Shader ---------------------
// Fragment shader for shadow map generation

#version 330 core

layout (location = 0) out vec4 color;

void main(void)
{
    color = vec4(1.0);
}
```

1. The results of rasterization are undefined in OpenGL if no fragment shader is present. It is legal to have no fragment shader when rasterization is turned off, but here we *do* want to rasterize so that we can generate depth values for the scene.

At this point we are ready to render the scene into the depth texture we created earlier. We need to bind the framebuffer object with the depth texture attachment and set the viewport to the depth texture size. Then we clear the depth buffer (which is actually our depth texture now) and draw the scene. Example 7.18 contains the code to do this.

Example 7.18 Rendering the Scene From the Light's Point of View

```
// Bind the "depth only" FBO and set the viewport to the size
// of the depth texture
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);
glViewport(0, 0, DEPTH_TEXTURE_SIZE, DEPTH_TEXTURE_SIZE);

// Clear
glClearDepth(1.0f);
glClear(GL_DEPTH_BUFFER_BIT);

// Enable polygon offset to resolve depth-fighting isuses
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(2.0f, 4.0f);
// Draw from the light's point of view
DrawScene(true);
glDisable(GL_POLYGON_OFFSET_FILL);
```

Notice that we're using *polygon offset* here. This pushes the generated depth values away from the viewer (the light, in this case) by a small amount. In this application, we want the depth test to be conservative, insofar as when there is doubt about whether a point is in shadow or not, we want to light it. If we did not do this, we would end up with *depth fighting* in the rendered image due to precision issues with the floating-point depth buffer. Figure 7.10 shows the resulting depth map of our scene as seen from the light's position.

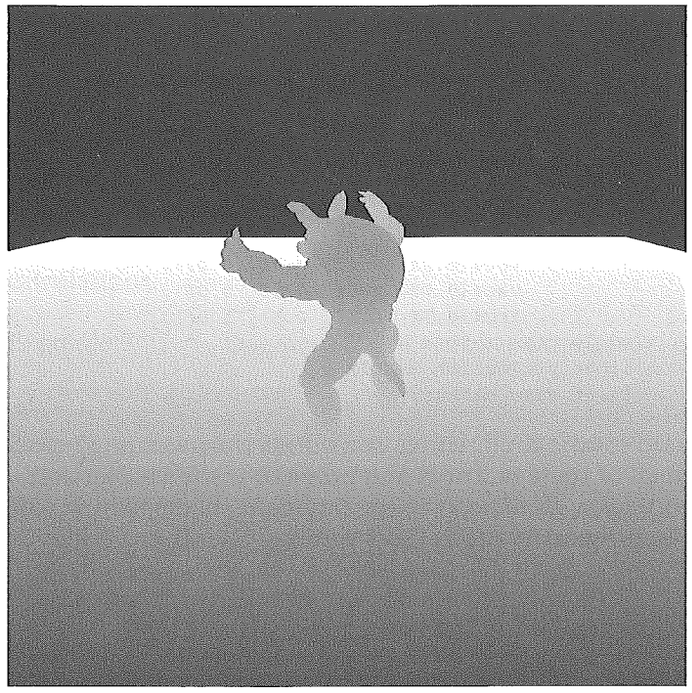

Figure 7.10 Depth rendering
(Depths are rendered from the light's position. Within rendered objects, closer points have smaller depths and show up darker.)

Using a Shadow Map

Now that we have the depth for the scene rendered from the light's point of view we can render the scene with our regular shaders and use the resulting depth texture to produce shadows as part of our lighting calculations. This is where the meat of the algorithm is. First, we need to set up the matrices for rendering the scene from the viewer's position. The matrices we'll need are the *model matrix*, *view matrix* (which transforms vertices for classic lighting), and the *projection matrix* (which transforms coordinates to projective space for rasterization). Also, we'll need a *shadow matrix*. This matrix transforms world coordinates into the light's projective space and simultaneously applies a scale and bias to the resulting depth values. The transformation to the light's eye space is performed by transforming the world space vertex coordinates through the light's view matrix followed by the light's projection matrix (which we calculated earlier). The scale and bias matrix maps depth values in projection space (which lie between -1.0 and $+1.0$) into the range 0.0 to 1.0.

The code to set all these matrices up is given in Example 7.19.

Example 7.19 Matrix Calculations for Shadow Map Rendering

```
mat4 scene_model_matrix = rotate(t * 720.0f, Y);
mat4 scene_view_matrix = translate(0.0f, 0.0f, -300.0f);
mat4 scene_projection_matrix = frustum(-1.0f, 1.0f, -aspect, aspect,
                                       1.0f, FRUSTUM_DEPTH);
mat4 scale_bias_matrix = mat4(vec4(0.5f, 0.0f, 0.0f, 0.0f),
                              vec4(0.0f, 0.5f, 0.0f, 0.0f),
                              vec4(0.0f, 0.0f, 0.5f, 0.0f),
                              vec4(0.5f, 0.5f, 0.5f, 1.0f));
mat4 shadow_matrix = scale_bias_matrix *
                     light_projection_matrix *
                     light_view_matrix;
```

The vertex shader used for the final render transforms the incoming vertex coordinates through all of these matrices and provides world coordinates, eye coordinates, and *shadow coordinates* to the fragment shader, which will perform the actual lighting calculations. This vertex shader is given in Example 7.20.

Example 7.20 Vertex Shader for Rendering from Shadow Maps

```
#version 330 core

uniform mat4 model_matrix;
uniform mat4 view_matrix;
uniform mat4 projection_matrix;

uniform mat4 shadow_matrix;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

out VS_FS_INTERFACE
{
    vec4 shadow_coord;
    vec3 world_coord;
    vec3 eye_coord;
    vec3 normal;
} vertex;

void main(void)
{
    vec4 world_pos = model_matrix * position;
    vec4 eye_pos = view_matrix * world_pos;
    vec4 clip_pos = projection_matrix * eye_pos;

    vertex.world_coord = world_pos.xyz;
    vertex.eye_coord = eye_pos.xyz;
    vertex.shadow_coord = shadow_matrix * world_pos;
    vertex.normal = mat3(view_matrix * model_matrix) * normal;

    gl_Position = clip_pos;
}
```

Finally, the fragment shader performs lighting calculations for the scene. If the point is considered to be illuminated by the light, the light's contribution is included in the final lighting calculation, otherwise only ambient light is applied. The shader given in Example 7.21 performs these calculations.

Example 7.21 Fragment Shader for Rendering from Shadow Maps

```
#version 330 core

uniform sampler2DShadow depth_texture;
uniform vec3 light_position;

uniform vec3 material_ambient;
uniform vec3 material_diffuse;
uniform vec3 material_specular;
uniform float material_specular_power;

layout (location = 0) out vec4 color;

in VS_FS_INTERFACE
{
    vec4 shadow_coord;
    vec3 world_coord;
    vec3 eye_coord;
    vec3 normal;
} fragment;

void main(void)
{
    vec3 N = fragment.normal;
    vec3 L = normalize(light_position - fragment.world_coord);
    vec3 R = reflect(-L, N);
    vec3 E = normalize(fragment.eye_coord);
    float NdotL = dot(N, L);
    float EdotR = dot(-E, R);

    float diffuse = max(NdotL, 0.0);
    float specular = max(pow(EdotR, material_specular_power), 0.0);

    float f = textureProj(depth_texture, fragment.shadow_coord);

    color = vec4(material_ambient +
                 f * (material_diffuse * diffuse +
                      material_specular * specular), 1.0);
}
```

Don't worry about the complexity of the lighting calculations in this shader. The important part of the algorithm is the use of the `sampler2DShadow` sampler type and the `textureProj` function. The `sampler2DShadow` sampler is a special type of 2D texture that, when sampled, will return either 1.0 if the sampled texture satisfies the comparison test for the texture, and 0.0 if it does not. The texture comparison mode for the depth texture was set earlier in Example 7.15 by calling **glTexParameteri()** with the GL_TEXTURE_COMPARE_MODE parameter name and GL_COMPARE_REF_TO_TEXTURE parameter value. When the depth comparison mode for the texture is configured like this, the texel values will be compared against the reference value that is supplied in the third component of `fragment.shadow_coord`—which is the z component of the scaled and biased projective-space coordinate of the fragment as viewed from the light. The depth comparison function is set to GL_LEQUAL, which causes the test to pass if the reference value is less than or equal to the value in the texture. When multiple texels are sampled (e.g., when the texture mode is linear), the result of reading from the texture is the average of all the 0.0s and 1.0s for the samples making up the final texel. That is, near the edge of a shadow, the returned value might be 0.25, or 0.5, and so on, rather than just 0.0 or 1.0. We scale the lighting calculations by this result to take light visibility into account during shading.

The `textureProj` function is a *projective texturing* function. It divides the incoming texture coordinate (in this case `fragment.shadow_coord` by its own last component (`fragment.shadow_coord.w`) to transform it into normalized device coordinates, which is exactly what the perspective transformation performed by OpenGL before rasterization does.

The result of rendering our scene with this shader is shown in Figure 7.11.

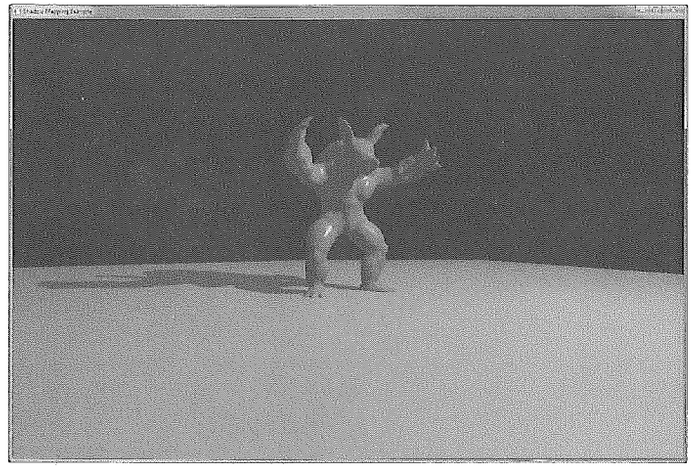

Figure 7.11 Final rendering of shadow map

That wraps up *shadow mapping*. There are many other techniques, including enhancements to shadow mapping, and we encourage you to explore on your own.