

Light and Shadow

LIGHTING EFFECTS CAN GREATLY enhance the three-dimensional nature of a scene, as illustrated in Figure 6.1, which illustrates a light source and shaded objects, specular highlights, bump map textures that simulate surface detail, a postprocessing effect to simulate a glowing sun, and shadows cast from objects onto other objects.

When using lights, the colors on a surface may be brighter or dimmer, depending on the angle at which the light rays meet the surface. This effect is called *shading* and enables viewers to observe the 3D nature of a shape in a rendered image, without the need for vertex colors or textures, as illustrated in Figure 6.2.

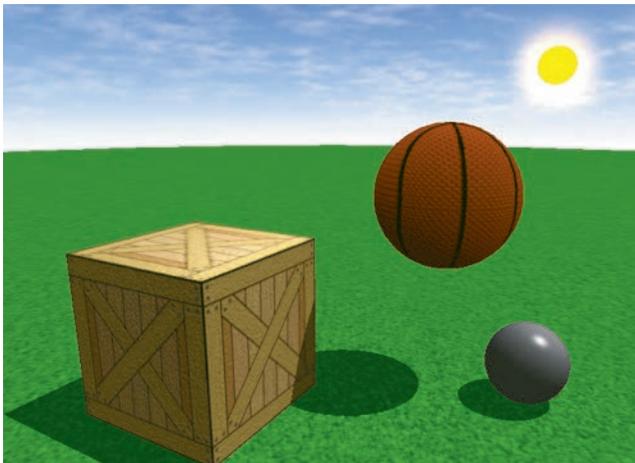


FIGURE 6.1 Rendered scene with lighting effects.

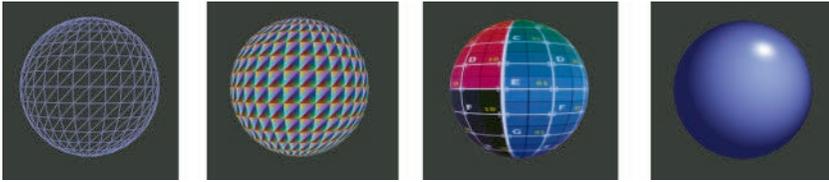


FIGURE 6.2 Four techniques to visualize a sphere: wireframe, vertex colors, texture, and shading.

In this chapter, you will learn how to calculate the effects of different types of light, such as ambient, diffuse, and specular, as well as how to simulate different light sources such as directional lights and point lights. Then, you will create new types of light objects to store this data, update various geometric classes to store additional related data, create a set of materials to process this data, and incorporate all these elements into the rendering process. Finally, you will learn about and implement the advanced topics illustrated by Figure 6.1, including bump mapping to simulate surface details, additional postprocessing techniques to generate light bloom and glow effects, and shadow mapping, which enables objects to cast shadows on other objects.

6.1 INTRODUCTION TO LIGHTING

There are many different types of lighting that may be used when rendering an object. *Ambient lighting* affects all points on all geometric surfaces in a scene by the same amount. Ambient light simulates light that has been reflected from other surfaces and ensures that objects or regions not directly lit by other types of lights remain at least partially visible. *Diffuse lighting* represents light that has been scattered and thus will appear lighter or darker in various regions, depending on the angle of the incoming light. *Specular lighting* creates bright spots and highlights on a surface to simulate *shininess*: the tendency of a surface to reflect light. These three types of lighting applied to a torus-shaped surface are illustrated in Figure 6.3. (A fourth type of lighting is *emissive lighting*, which is light emitted by an object that can be used to create a glow-like effect, but this will not be covered here.)

An *illumination model* is a combination of lighting types used to determine the color at each point on a surface. Two of the most commonly used illumination models are the Lambert model and the Phong model, illustrated in Figure 6.4. The *Lambert model* uses a combination of

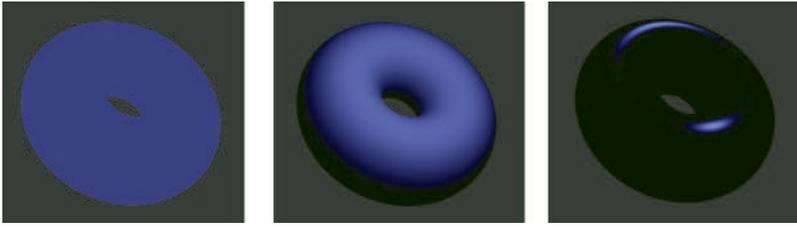


FIGURE 6.3 Ambient, diffuse, and specular lighting on a torus-shaped surface.

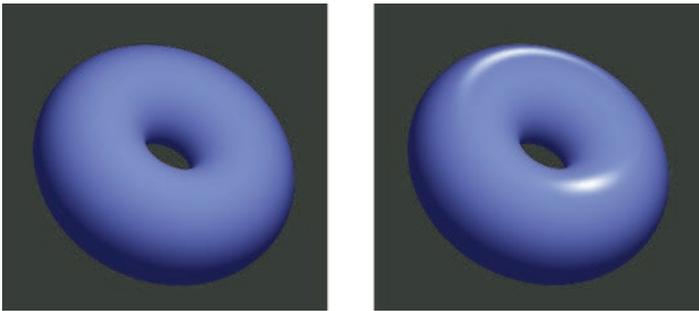


FIGURE 6.4 The Lambert and Phong illumination models on a torus-shaped surface.

ambient and diffuse lighting, particularly appropriate for simulating the appearance of matte or rough surfaces, where most of the light rays meeting the surface are scattered. The *Phong model* uses ambient, diffuse, and specular lighting, and is particularly appropriate for reflective or shiny surfaces. Due to the additional lighting data and calculations required, the Phong model is more computationally intensive than the Lambert model. The strength or sharpness of the shine in the Phong model can be adjusted by parameters stored in the corresponding material, and if the strength of the shine is set to zero, the Phong model generates results identical to the Lambert model.

The magnitude of the effect of a light source at a point depends on the angle at which a ray of light meets a surface. This angle is calculated as the angle between two vectors: the direction vector of the light ray, and a *normal vector*: a vector perpendicular to the surface. Figure 6.5 illustrates normal vectors for multiple surfaces as short line segments. When calculating normal vectors to a surface, one has the option of using either vertex normal vectors or face normal vectors. *Face normal vectors* are

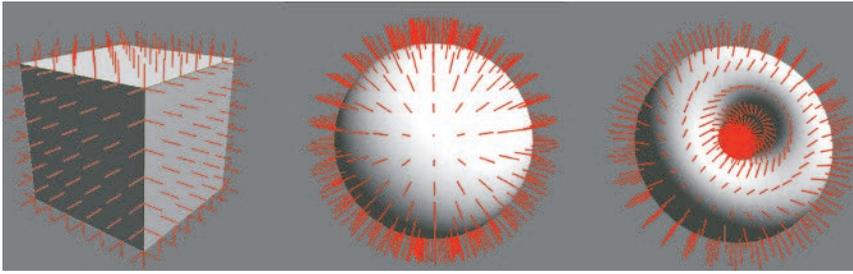


FIGURE 6.5 Normal vectors to a box, a sphere, and a torus.

perpendicular to the triangles in the mesh used to represent the surface. *Vertex normal vectors* are perpendicular to the geometric surface being approximated; the values of these vectors do not depend on the triangulation of the surface and can be calculated precisely when the parametric function defining the surface is known. For a surface defined by flat sides (such as a box or pyramid), there is no difference between vertex normals and face normals.

Different types of light objects will be used to simulate different sources of light, each of which emits light rays in different patterns. A *point light* simulates rays of light being emitted from a single point in all directions, similar to a lightbulb, and incorporates *attenuation*: a decrease in intensity as the distance between the light source and the surface increases. A *directional light* simulates a distant light source such as the sun, in which all the light rays are oriented along the same direction and there is no attenuation. (As a result, the position of a directional light has no effect on surfaces that it lights.) These light ray direction patterns are illustrated in Figure 6.6. For simplicity, in this framework, point lights and directional lights will only affect diffuse and specular values, and ambient light contributions

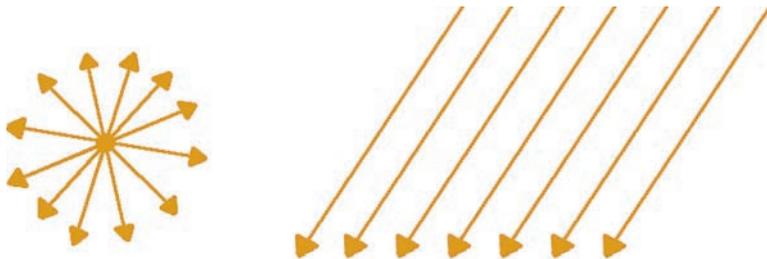


FIGURE 6.6 Directions of emitted light rays for point light (left) and directional light (right).

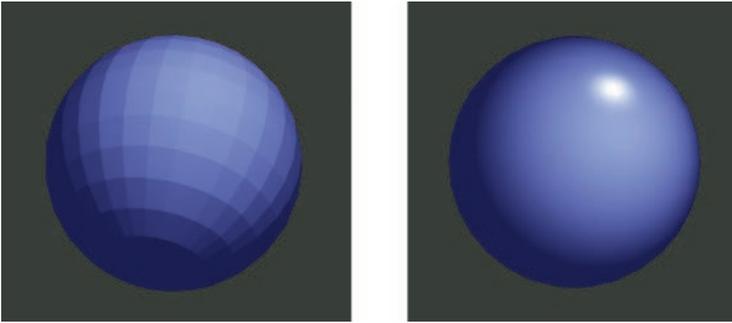


FIGURE 6.7 Rendering a sphere using flat shading (left) and Phong shading (right).

will be handled by a separate *ambient light* structure, although there is no physical analog for this object.

The choice of using face normal vectors or vertex normal vectors, and the part of the shader program in which light-related calculations appear define different *shading models*. The original and simplest is the *flat shading model*, in which face normal vectors are used, calculations take place in the vertex shader, and light contribution values are passed along to the fragment shader. The result is a faceted appearance, even on a smooth surface such as a sphere, as illustrated in Figure 6.7. The *Gouraud shading model* uses vertex normal vectors and calculates the effect of light at each vertex in the vertex shader; these values are passed through the graphics pipeline to the fragment shader, leading to interpolated values for each fragment and resulting in a smoother overall appearance (although there may be visual artifacts along the edges of some triangles). The most computationally intensive example that will be implemented in this framework, and the one that provides the smoothest and most realistic results, is the *Phong shading model* (not to be confused with the Phong illumination model), also illustrated in Figure 6.7. In this shading model, the normal vector data is passed from the vertex shader to the fragment shader, normal vectors are interpolated for each fragment, and the light calculations are performed at that stage.

6.2 LIGHT CLASSES

To incorporate lighting effects into the graphics framework, the first step is to create a series of light objects that store the associated data. For simplicity, a base **Light** class will be created that stores data that could be

needed by any type of light, including a constant that specifies the type of light (ambient, directional, or point). Some lights will require position or direction data, but since the `Light` class will extend the `Object3D` class, this information can be stored in and retrieved from the associated transformation matrix. Extensions of the `Light` class will represent the different types of lights, and their constructors will store values in the relevant fields defined by the base class.

To begin, in the main project folder, create a new folder named `light`. In this folder, create a new file named `light.py` with the following code:

```
from core.object3D import Object3D
class Light(Object3D):

    AMBIENT      = 1
    DIRECTIONAL  = 2
    POINT        = 3
    def __init__(self, lightType=0):
        super().__init__()
        self.lightType = lightType
        self.color      = [1, 1, 1]
        self.attenuation = [1, 0, 0]
```

As previously mentioned and alluded to by the constant values in the `Light` class, there will be three extensions of the class that represent different types of lights: ambient light, directional light, and point light. To implement the class representing ambient light, the simplest of the three, as it only uses the color data, in the `light` folder create a new file named `ambientLight.py` with the following code:

```
from light.light import Light
class AmbientLight(Light):

    def __init__(self, color=[1,1,1]):
        super().__init__(Light.AMBIENT)
        self.color = color
```

Next, you will implement the directional light class. However, you first need to add some functionality to the `Object3D` class that enables you to get and set the direction an object is facing, also called the *forward*

direction, which is defined by the orientation of its local negative z-axis. This concept was originally introduced in the discussion of the look-at matrix in Chapter 5. The `setDirection` function is effectively a local version of the `lookAt` function. To calculate the direction an object is facing, the `getDirection` function requires the rotation component of the mesh's transformation matrix, which is the top-left 3-by-3 submatrix; this functionality will be provided by a new function called `getRotationMatrix`. To proceed, in the file `object3D.py` in the `core` folder, add the following import statement:

```
import numpy
```

Then, add the following three functions to the class:

```
# returns 3x3 submatrix with rotation data
def getRotationMatrix(self):
    return numpy.array( [ self.transform[0] [0:3],
                        self.transform[1] [0:3],
                        self.transform[2] [0:3] ] )

def getDirection(self):
    forward = numpy.array([0,0,-1])
    return list( self.getRotationMatrix() @ forward )

def setDirection(self, direction):
    position = self.getPosition()
    targetPosition = [ position[0] + direction[0],
                      position[1] + direction[1],
                      position[2] + direction[2] ]
    self.lookAt( targetPosition )
```

With these additions, you are ready to implement the class that represents directional lights. In the `light` folder, create a new file named `directionalLight.py` with the following code:

```
from light.light import Light
class DirectionalLight(Light):

    def __init__(self, color=[1,1,1], direction=[0,
-1, 0]):
```

```

super().__init__(Light.DIRECTIONAL)
self.color = color
self.setDirection( direction )

```

Finally, you will implement the class that represents point lights. The variable **attenuation** stores a list of parameters that will be used when calculating the decrease in light intensity due to increased distance, which will be discussed in more detail later. In the **light** folder, create a new file named **pointLight.py** with the following code:

```

from light.light import Light
class PointLight(Light):

    def __init__(self, color=[1,1,1],
                 position=[0,0,0], attenuation=[1,0,0.1]):
        super().__init__(Light.POINT)
        self.color = color
        self.setPosition( position )
        self.attenuation = attenuation

```

6.3 NORMAL VECTORS

Just as working with textures required the addition of a new attribute (representing UV coordinates) to geometry classes, working with lights also requires new attributes, as discussed in the beginning of this chapter, representing vertex normal vectors and face normal vectors. The next step will be to add new attributes containing these two types of vectors to the previously created geometry classes: rectangles, boxes, polygons, and parametric surfaces. The shader code that will be created later in this chapter will access this data through shader variables named **vertexNormal** and **faceNormal**. In the case of rectangles, boxes, and polygons, since the sides of these shapes are flat, these two types of normal vectors are the same. Curved surfaces defined by parametric functions will require slightly more effort to calculate these two types of normal vectors.

6.3.1 Rectangles

Since a rectangle is a flat shape aligned with the *xy*-plane, the normal vectors at each vertex all point in the same direction: $\langle 0, 0, 1 \rangle$, aligned with the positive *z*-axis, as illustrated in Figure 6.8.

To implement normal vectors for rectangles, in the file **rectangleGeometry.py** in the **geometry** folder, add the following code:

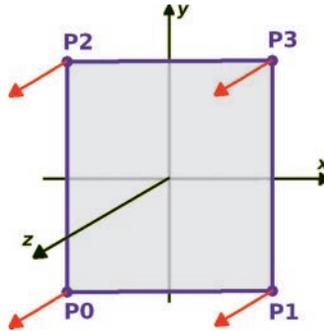


FIGURE 6.8 Normal vectors for a rectangle.

```

normalVector = [0, 0, 1]
normalData = [ normalVector ] * 6
self.addAttribute("vec3", "vertexNormal", normalData)
self.addAttribute("vec3", "faceNormal", normalData)

```

6.3.2 Boxes

To begin, recall the alignment of the vertices in a box geometry, illustrated in Figure 6.9.

Since a box has six flat sides, there will be six different normal vectors required for this shape. The right and left sides, as they are perpendicular to the x -axis, will have normal vectors $\langle 1, 0, 0 \rangle$ and $\langle -1, 0, 0 \rangle$. The top and bottom sides, perpendicular to the y -axis, will have normal vectors $\langle 0, 1, 0 \rangle$ and $\langle 0, -1, 0 \rangle$. The front and back sides, perpendicular to the z -axis, will have normal vectors $\langle 0, 0, 1 \rangle$ and $\langle 0, 0, -1 \rangle$. Observe that each corner point is part of three different sides; for example, point P6 is part of the left, top, and front sides. Thus, each corner of the cube may correspond to one of three normal vectors, depending on the triangle being generated in the

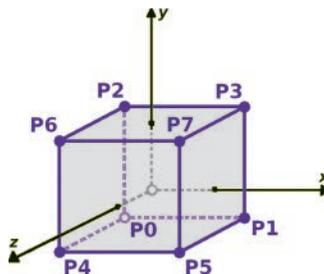


FIGURE 6.9 Vertices in a box geometry.

rendering process. To add normal vector data for this shape, in the file `boxGeometry.py` in the `geometry` folder, add the following code:

```
# normal vectors for x+, x-, y+, y-, z+, z-
N1, N2 = [1, 0, 0], [-1, 0, 0]
N3, N4 = [0, 1, 0], [0, -1, 0]
N5, N6 = [0, 0, 1], [0, 0, -1]
normalData = [N1]*6 + [N2]*6 + [N3]*6 + [N4]*6 +
              [N5]*6 + [N6]*6
self.addAttribute("vec3", "vertexNormal", normalData)
self.addAttribute("vec3", "faceNormal", normalData)
```

6.3.3 Polygons

Just as was the case for rectangles, since polygons are flat shapes aligned with the xy -plane, the normal vectors at each vertex are $\langle 0, 0, 1 \rangle$. To add normal vector data for polygons, in the file `polygonGeometry.py` in the `geometry` folder, you will need to add code in three different parts. First, before the `for` loop, add the following code:

```
normalData = []
normalVector = [0, 0, 1]
```

Within the `for` loop, the same line of code is repeated three times because each triangle has three vertices; as with the other attributes, three vectors are appended to the corresponding data array.

```
normalData.append( normalVector )
normalData.append( normalVector )
normalData.append( normalVector )
```

After the `for` loop, add the following code:

```
self.addAttribute("vec3", "vertexNormal", normalData)
self.addAttribute("vec3", "faceNormal", normalData)
```

This completes the necessary additions to the `Polygon` class.

6.3.4 Parametric Surfaces

Finally, you will add normal vector data for parametric surfaces. Unlike the previous cases, this will involve some calculations. To calculate the face normal vectors for each triangle, you will use the cross product operation,

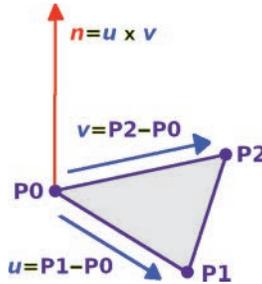


FIGURE 6.10 Calculating the normal vector for a triangle.

which takes two vectors as input and produces a vector perpendicular to both of the input vectors. Since each triangle is defined by three points P_0 , P_1 , and P_2 , one can subtract these points in pairs to create two vectors $\mathbf{v} = P_1 - P_0$ and $\mathbf{w} = P_2 - P_0$ aligned with the edges of the triangle. Then, the cross product of \mathbf{v} and \mathbf{w} results in the desired face normal vector \mathbf{n} . This calculation is illustrated in Figure 6.10.

To calculate the vertex normal vector at a point P_0 on the surface involves the same process, except that the points P_1 and P_2 used for this calculation are chosen to be very close to P_0 in order to more precisely approximate the exact normal to the surface. In particular, assume that the surface is defined by the parametric function S and that $P_0 = S(u, v)$. Let h be a small number, such as $h = 0.0001$. Then, define two additional points $P_1 = S(u+h, v)$ and $P_2 = S(u, v+h)$. With the three points P_0 , P_1 , and P_2 , one may then proceed exactly as before to obtain the desired vertex normal vector.

To implement these calculations, in the file `parametricGeometry.py` in the `geometry` directory, begin by adding the following import statement:

```
import numpy
```

Then, after the `for` loop that calculates the contents of the `uvs` list, add the following code, which implements a function to calculate a normal vector from three points as previously described, and populates a list with vertex normal vectors at the position of each vertex.

```
def calcNormal(P0, P1, P2):
    v1 = numpy.array(P1) - numpy.array(P0)
    v2 = numpy.array(P2) - numpy.array(P0)
```

```

normal = numpy.cross( v1, v2 )
normal = normal / numpy.linalg.norm(normal)
return normal

```

```

vertexNormals = []
for uIndex in range(uResolution+1):
    vArray = []
    for vIndex in range(vResolution+1):
        u = uStart + uIndex * deltaU
        v = vStart + vIndex * deltaV
        h = 0.0001
        P0 = surfaceFunction(u, v)
        P1 = surfaceFunction(u+h, v)
        P2 = surfaceFunction(u, v+h)
        normalVector = calcNormal(P0, P1, P2)
        vArray.append( normalVector )
    vertexNormals.append(vArray)

```

Then, immediately before the nested **for** loop that groups the vertex data into triangles, add the following code:

```

vertexNormalData = []
faceNormalData   = []

```

Within the nested **for** loop, after data is appended to the **uvData** list, add the following code:

```

# vertex normal vectors
nA = vertexNormals[xIndex+0][yIndex+0]
nB = vertexNormals[xIndex+1][yIndex+0]
nD = vertexNormals[xIndex+0][yIndex+1]
nC = vertexNormals[xIndex+1][yIndex+1]
vertexNormalData += [nA,nB,nC, nA,nC,nD]

# face normal vectors
fn0 = calcNormal(pA, pB, pC)
fn1 = calcNormal(pA, pC, pD)
faceNormalData += [fn0,fn0,fn0, fn1,fn1,fn1]

```

Finally, after the nested **for** loop, add the following two lines of code before the **countVertices** function is called:

```
self.addAttribute("vec3", "vertexNormal",
    vertexNormalData)
self.addAttribute("vec3", "faceNormal",
    faceNormalData)
```

Another related change that needs to be made is in the **Geometry** class **applyMatrix** function, which currently transforms only the position-related data stored in an attribute of a geometry. When transforming a geometry, the normal vector data should also be updated, but only by the rotational part of the transformation (as normal vectors are assumed to be in standard position, with initial point at the origin). This functionality is especially important for cylinder-based shapes, as they include both a parametric geometry component and one or two transformed polygon geometry components. To implement this, in the file **geometry.py** in the **geometry** folder, in the **applyMatrix** function, add the following code directly before the **uploadData** function is called.

```
# extract the rotation submatrix
rotationMatrix = numpy.array( [ matrix[0] [0:3],
                                matrix[1] [0:3],
                                matrix[2] [0:3] ] )

oldVertexNormalData = self.attributes["vertexNormal"].
    data
newVertexNormalData = []
for oldNormal in oldVertexNormalData:
    newNormal = oldNormal.copy()
    newNormal = rotationMatrix @ newNormal
    newVertexNormalData.append( newNormal )
self.attributes["vertexNormal"].data =
    newVertexNormalData

oldFaceNormalData = self.attributes["faceNormal"].
    data
newFaceNormalData = []
for oldNormal in oldFaceNormalData:
    newNormal = oldNormal.copy()
    newNormal = rotationMatrix @ newNormal
    newFaceNormalData.append( newNormal )
self.attributes["faceNormal"].data = newFaceNormalData
```

With these additions to the graphics framework, all geometric objects now include the vertex data that will be needed for lighting-based calculations.

6.4 USING LIGHTS IN SHADERS

The next major step in implementing lighting effects is to write a shader to perform the necessary calculations, which will require the data stored in the previously created **Light** class. Since scenes may feature multiple light objects, a natural way to proceed is to create a data structure within the shader to group this information together, analogous to the **Light** class itself. After learning how data is uploaded to a GLSL structure and updating the **Uniform** class as needed, the details of the light calculations will be explained. You will then implement three shaders: the flat shading model, the Lambert illumination model, and the Phong illumination model. While the first of these models uses face normal data in the vertex shader, the latter two models will use Phong shading, where vertex normal data will be interpolated and used in the fragment shader.

6.4.1 Structs and Uniforms

In GLSL, data structures are used to group together related data variables as a single unit, thus defining new types. These are created using the keyword **struct**, followed by a list of member variable types and names. For example, a structure to store light-related data will be defined as follows:

```
struct Light
{
    int lightType;
    vec3 color;
    vec3 direction;
    vec3 position;
    vec3 attenuation;
};
```

Following the definition of a struct, variables of this type may be defined in the shader. Fields within a struct are accessed using dot notation; for example, given a **Light** variable named **sun**, the information stored in the **direction** field can be accessed as **sun.direction**. The data for a uniform struct variable cannot all be uploaded by a single OpenGL function, so there will be a significant addition to the **Uniform**

class corresponding to light-type objects. When storing such an object, the **Uniform** class variable **variableRef** will not store a single uniform variable reference, but rather a dictionary object whose keys are the names of the struct fields and whose values are the corresponding variable references. When uploading data to the GPU, multiple **glUniform**-type functions will be called.

To add this functionality, in the file **uniform.py** in the **core** folder, change the **locateVariable** function to the following:

```
# get and store reference(s) for program variable with
  given name
def locateVariable(self, programRef, variableName):
    if self.dataType == "Light":
        self.variableRef = {}
        self.variableRef["lightType"] =
            glGetUniformLocation(programRef,
                variableName + ".lightType")
        self.variableRef["color"] =
            glGetUniformLocation(programRef,
                variableName + ".color")
        self.variableRef["direction"] =
            glGetUniformLocation(programRef,
                variableName + ".direction")
        self.variableRef["position"] =
            glGetUniformLocation(programRef,
                variableName + ".position")
        self.variableRef["attenuation"] =
            glGetUniformLocation(programRef,
                variableName + ".attenuation")
    else:
        self.variableRef = glGetUniformLocation
            (programRef, variableName)
```

Also in the **Uniform** class, in the **uploadData** function **if-else** block, add the following code:

```
elif self.dataType == "Light":
    glUniform1i( self.variableRef["lightType"], self.
        data.lightType )
    glUniform3f( self.variableRef["color"],
```

```

        self.data.color[0], self.data.color[1], self.
            data.color[2] )
    direction = self.data.getDirection()
    glUniform3f( self.variableRef["direction"],
        direction[0], direction[1], direction[2] )
    position = self.data.getPosition()
    glUniform3f( self.variableRef["position"],
        position[0], position[1], position[2] )
    glUniform3f( self.variableRef["attenuation"],
        self.data.attenuation[0],
        self.data.attenuation[1],
        self.data.attenuation[2] )

```

6.4.2 Light-Based Materials

In each of the three materials that will be created in this section, key features will be the **Light** struct previously discussed, the declaration of four uniform **Light** variables (this will be the maximum supported by this graphics framework), and a function (named **lightCalc**) to calculate the effect of light sources at a point. In the flat shading material, these elements will be added to the vertex shader, while in the Lambert and Phong materials, these elements will be added to the fragment shader instead.

To begin, you will create the flat shader material. In the **material** directory, create a new file called **flatMaterial.py** containing the following code; the code for the vertex and fragment shaders and for adding uniform objects will be added later.

```

from material.material import Material
from OpenGL.GL import *
class FlatMaterial(Material):

    def __init__(self, texture=None, properties={}):

        vertexShaderCode = """
        // (vertex shader code to be added)
        """

        fragmentShaderCode = """
        // (fragment shader code to be added)
        """

```

```

super().__init__(vertexShaderCode,
                 fragmentShaderCode)
// (uniforms to be added)
self.locateUniforms()

# render both sides?
self.settings["doubleSide"] = True
# render triangles as wireframe?
self.settings["wireframe"] = False
# line thickness for wireframe rendering
self.settings["lineWidth"] = 1

self.setProperties(properties)

def updateRenderSettings(self):

    if self.settings["doubleSide"]:
        glDisable(GL_CULL_FACE)
    else:
        glEnable(GL_CULL_FACE)

    if self.settings["wireframe"]:
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)
    else:
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)

    glLineWidth(self.settings["lineWidth"])

```

In the flat shading model, lights are processed in the vertex shader. Thus, in the vertex shader code area, add the following code:

```

struct Light
{
    // 1 = AMBIENT, 2 = DIRECTIONAL, 3 = POINT
    int lightType;
    // used by all lights
    vec3 color;
    // used by directional lights
    vec3 direction;
    // used by point lights
    vec3 position;
}

```

```

        vec3 attenuation;
    };

uniform Light light0;
uniform Light light1;
uniform Light light2;
uniform Light light3;

```

Next, you will implement the `lightCalc` function. The function will be designed so that it may calculate the contributions from a combination of ambient, diffuse, and specular light. In the flat shading and Lambert materials, only ambient and diffuse light contributions are considered, and thus, the only parameters required by the `lightCalc` function are the light source itself, and the position and normal vector for a point on the surface. Values for the contributions from each type of light are stored in the variables `ambient`, `diffuse`, and `specular`, each of which is initially set to zero and then modified as necessary according to the light type.

The calculations for the diffuse component of a directional light and a point light are quite similar. One difference is in the calculation of the light direction vector: for a directional light, this is constant, but for a point light, this is dependent on the position of the light and the position of the point on the surface. Once the light direction vector is known, the contribution of the light source at a point can be calculated. The value of the contribution depends on the angle between the light direction vector and the normal vector to the surface. When this angle is small (close to zero), the contribution of the light source is close to 100%. As the angle approaches 90°, the contribution of the light source approaches 0%. This models the observation that light rays meeting a surface at large angles are scattered, which reduces the intensity of the reflected light. Fortunately, this mathematical relationship is easily captured by the cosine function, as $\cos(0^\circ)=1$ and $\cos(90^\circ)=0$. Furthermore, it can be proven that the cosine of the angle between two unit vectors (vectors with length 1) is equal to the dot product of the vectors, which can be calculated with the GLSL function `dot`. If the value of the cosine is negative, this indicates that the surface is inclined at an angle away from the light source, in which case the contribution should be set to zero; this will be accomplished with the GLSL function `max`, as you will see.



FIGURE 6.11 Attenuation of a light source as a function of distance.

Finally, point lights also incorporate attenuation effects: the intensity of the light should decrease as the distance d between the light source and the surface increases. This effect is modeled mathematically by multiplying the diffuse component by the factor $1/(a + b \cdot d + c \cdot d^2)$, where the coefficients a , b , c are used to adjust the rate at which the light effect decreases. Figure 6.11 displays a graph of this function for the default attenuation coefficients $a=1$, $b=0$, $c=0.1$, which results in the function $1/(1 + 0 \cdot d + 0.1 \cdot d^2)$. Observe that in the graph, when the distance is at a minimum ($d=0$), the attenuation factor is 1, and the attenuation is 50% approximately when $d=3.2$.

To implement the `lightCalc` function, in the vertex shader code, add the following after the declaration of the uniform light variables:

```
vec3 lightCalc(Light light, vec3 pointPosition, vec3
    pointNormal)
{
    float ambient = 0;
    float diffuse = 0;
    float specular = 0;
    float attenuation = 1;
    vec3 lightDirection = vec3(0,0,0);

    if ( light.lightType == 1 ) // ambient light
    {
        ambient = 1;
    }
    else if ( light.lightType == 2 ) // directional
        light
    {
        lightDirection = normalize(light.direction);
    }
    else if ( light.lightType == 3 ) // point light
    {
```

```

        lightDirection = normalize(pointPosition -
            light.position);
        float distance = length(light.position -
            pointPosition);
        attenuation = 1.0 / (light.attenuation[0] +
            light.attenuation[1] *
                distance +
            light.attenuation[2] *
                distance * distance);
    }
    if ( light.lightType > 1 ) // directional or point
        light
    {
        pointNormal = normalize(pointNormal);
        diffuse = max( dot(pointNormal,
            -lightDirection), 0.0 );
        diffuse *= attenuation;
    }

    return light.color * (ambient + diffuse +
        specular);
}

```

With these additions in place, you are ready to complete the vertex shader for the flat shading material. In addition to the standard calculations involving the vertex position and UV coordinates, you also need to calculate the total contribution from all of the lights. If data for a light variable has not been set, then the light's **lightType** variable defaults to zero, in which case the value returned by **lightCalc** is also zero. Before being used in the **lightCalc** function, the model matrix needs to be applied to the position data, and the rotational part of the model matrix needs to be applied to the normal data. The total light contribution is passed from the vertex shader to the fragment shader for use in determining the final color of each fragment. In the vertex shader code, add the following code after the body of the **lightCalc** function:

```

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;
in vec3 vertexPosition;
in vec2 vertexUV;

```

```

in vec3 faceNormal;
out vec2 UV;
out vec3 light;

void main()
{
    gl_Position = projectionMatrix * viewMatrix *
                  modelMatrix
                  * vec4(vertexPosition, 1);
    UV = vertexUV;
    // calculate total effect of lights on color
    vec3 position = vec3( modelMatrix *
                          vec4(vertexPosition, 1) );
    vec3 normal = normalize( mat3(modelMatrix) *
                             faceNormal );
    light = vec3(0,0,0);
    light += lightCalc( light0, position, normal );
    light += lightCalc( light1, position, normal );
    light += lightCalc( light2, position, normal );
    light += lightCalc( light3, position, normal );
}

```

Next, you need to add the code for the flat material fragment shader. In addition to the standard elements of past fragment shaders, this new shader also uses the light value calculated in the vertex shader when determining the final color of a fragment. This material (as well as the two that follow) will include an optional **texture** parameter that can be set. If a texture is passed into the material, it will also cause a shader variable **useTexture** to be set to true, in which case a color sampled from the supplied texture will be combined with the material's base color. To implement this, set the fragment shader code in the flat material to be the following:

```

uniform vec3 baseColor;
uniform bool useTexture;
uniform sampler2D texture;
in vec2 UV;
in vec3 light;
out vec4 fragColor;
void main()
{
    vec4 color = vec4(baseColor, 1.0);
    if ( useTexture )

```

```

        color *= texture2D( texture, UV );
    color *= vec4( light, 1 );
    fragColor = color;
}

```

Finally, you must add the necessary uniform objects to the material using the `addUniform` function. To proceed, after the fragment shader code and before the function `locateUniforms` is called, add the following code. (The data for the light objects will be supplied by the `Renderer` class, handled similarly to the model, view, and projection matrix data.)

```

self.addUniform("vec3", "baseColor", [1.0, 1.0, 1.0])
self.addUniform("Light", "light0", None )
self.addUniform("Light", "light1", None )
self.addUniform("Light", "light2", None )
self.addUniform("Light", "light3", None )
self.addUniform("bool", "useTexture", 0)
if texture == None:
    self.addUniform("bool", "useTexture", False)
else:
    self.addUniform("bool", "useTexture", True)
    self.addUniform("sampler2D", "texture", [texture.
        textureRef, 1])

```

This completes the code required for the `FlatMaterial` class.

Next, to create the Lambert material, make a copy of the file `flatMaterial.py` and name the copy `lambertMaterial.py`. Within this file, change the name of the class `LambertMaterial`. Since the Phong shading model will be used in this material (as opposed to the Gouraud shading model), the light-based calculations will occur in the fragment shader. Therefore, move the code involving the definition of the `Light` struct, the declaration of the four `Light` variables, and the function `lightCalc` from the vertex shader to the beginning of the fragment shader. Then, since vertex normals will be used instead of face normals, and since the position and normal data must be transmitted to the fragment shader for use in the `lightCalc` function there, change the code for the vertex shader to the following:

```

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;

```

```

uniform mat4 modelMatrix;
in vec3 vertexPosition;
in vec2 vertexUV;
in vec3 vertexNormal;
out vec3 position;
out vec2 UV;
out vec3 normal;
void main()
{
    gl_Position = projectionMatrix * viewMatrix *
        modelMatrix * vec4(vertexPosition, 1);
    position = vec3( modelMatrix *
        vec4(vertexPosition, 1) );
    UV = vertexUV;
    normal = normalize( mat3(modelMatrix) *
        vertexNormal );
}

```

In the Lambert material fragment shader, the light calculations will take place and be combined with the base color (and optionally, texture color data). Replace the fragment shader code following the declaration of the `lightCalc` function with the following:

```

uniform vec3 baseColor;
uniform bool useTexture;
uniform sampler2D texture;
in vec3 position;
in vec2 UV;
in vec3 normal;
out vec4 fragColor;

void main()
{
    vec4 color = vec4(baseColor, 1.0);
    if ( useTexture )
        color *= texture2D( texture, UV );
    // calculate total effect of lights on color
    vec3 total = vec3(0,0,0);
    total += lightCalc( light0, position, normal );
    total += lightCalc( light1, position, normal );
    total += lightCalc( light2, position, normal );
    total += lightCalc( light3, position, normal );
}

```

```

    color *= vec4( total, 1 );
    fragColor = color;
}

```

This completes the required code for the **LambertMaterial** class.

Finally, you will create the Phong material, which includes specular light contributions. This calculation is similar to the diffuse light calculation, involving the angle between two vectors (which can be calculated using a dot product), but in this case, the vectors of interest are the reflection of the light direction vector and the vector from the viewer or virtual camera to the surface point. These elements are illustrated in Figure 6.12. First, the light direction vector d impacts the surface at a point. Then, the vector d is reflected around the normal vector n , producing the reflection vector r . The vector from the virtual camera to the surface is indicated by v . The angle of interest is indicated by a ; it is the angle between the vector r and the vector v .

The impact of specular light on an object is typically adjusted by two parameters: *strength*, a multiplicative factor which can be used to make the overall specular light effect appear brighter or dimmer, and *shininess*, which causes the highlighted region to be more blurry or more sharply defined, which corresponds to how reflective the surface will be perceived. Figure 6.13 illustrates the effects of increasing the shininess value by a factor of 4 in each image from the left to the right.

To implement these changes, make a copy of the file **lambertMaterial.py** and name the copy **phongMaterial.py**. Within this file,

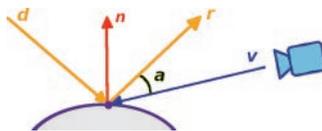


FIGURE 6.12 The vectors used in the calculation of specular highlights.

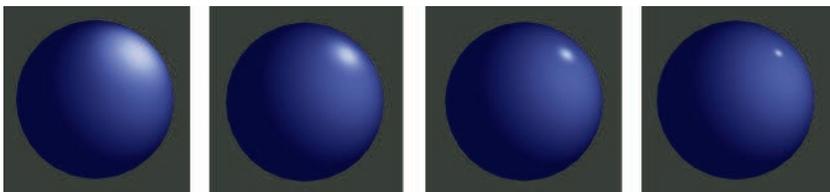


FIGURE 6.13 The effect of increasing shininess in specular lighting.

change the name of the class **PhongMaterial**. To perform the necessary calculations in the **lightCalc** function, it must take in additional data. Before the declaration of the **lightCalc** function, add the following uniform declarations (so that this function can access the associated values):

```
uniform vec3 viewPosition;
uniform float specularStrength;
uniform float shininess;
```

Then, within the **lightCalc** function, in the block of code corresponding to the condition **light.lightType > 1**, after the diffuse value is calculated, add the following code that will calculate the specular component when needed (when there is also a nonzero diffuse component):

```
if (diffuse > 0)
{
    vec3 viewDirection = normalize(viewPosition -
        pointPosition);
    vec3 reflectDirection = reflect(lightDirection,
        pointNormal);
    specular = max( dot(viewDirection,
        reflectDirection), 0.0 );
    specular = specularStrength * pow(specular,
        shininess);
}
```

Finally, in the section of the material code where the uniform data is added, add the following two lines of code, which supplies default values for the specular lighting parameters.

```
self.addUniform("vec3", "viewPosition", [0,0,0])
self.addUniform("float", "specularStrength", 1)
self.addUniform("float", "shininess", 32)
```

This completes the required code for the **PhongMaterial** class.

6.5 RENDERING SCENES WITH LIGHTS

In this section, you will update the **Renderer** class to extract the list of light objects that have been added to a scene, and supply that information to the corresponding uniforms. Following that, you will create a scene featuring

all three types of lights and all three types of materials. To begin, in the file `renderer.py` in the `core` folder, add the following import statement:

```
from light.light import Light
```

Next, during the rendering process, a list of lights must be extracted from the scene graph structure. This will be accomplished in the same way that the list of mesh objects is extracted: by creating a filter function and applying it to the list of descendents of the root of the scene graph. Furthermore, since data for four lights is expected by the shader, if less than four lights are present, then default `Light` objects will be created (which result in no contribution to the overall lighting of the scene) and added to the list. To proceed, in the `render` function, after the `meshList` variable is created, add the following code:

```
lightFilter = lambda x : isinstance(x, Light)
lightList = list( filter( lightFilter,
    descendentList ) )
# scenes support 4 lights; precisely 4 must be present
while len(lightList) < 4:
    lightList.append( Light() )
```

Next, for all light-based materials, you must set the data for the four uniform objects referencing lights; these materials can be identified during the rendering stage by checking if there is a uniform object stored with the key `"light0"`. Additionally, for the Phong material, you must set the data for the camera position; this case can be identified by checking for a uniform under the key `"viewPosition"`. This can be implemented by adding the following code in the `for` loop that iterates over `meshList`, directly after the matrix uniform data is set.

```
# if material uses light data, add lights from list
if "light0" in mesh.material.uniforms.keys():
    for lightNumber in range(4):
        lightName = "light" + str(lightNumber)
        lightObject = lightList[lightNumber]
        mesh.material.uniforms[lightName].data =
            lightObject
# add camera position if needed (specular lighting)
```

```
if "viewPosition" in mesh.material.uniforms.keys():
    mesh.material.uniforms["viewPosition"].data =
        camera.getWorldPosition()
```

With these additions to the graphics framework, you are ready to create an example. To fully test all the classes you have created so far in this chapter, you will create a scene that includes all the light types (ambient, directional, and point), as well as all the material types (flat, Lambert, and Phong). When you are finished, you will see a scene containing three spheres, similar to that in Figure 6.14. From left to right is a sphere with a red flat-shaded material, a sphere with a textured Lambert material, and a sphere with a blue-gray Phong material. The scene also includes a dark gray ambient light, a white directional light, and a red point light. The latter two light types and their colors may be guessed by the specular light colors on the third sphere and the amount of the sphere that is lit by each of the lights; the point light illuminates less of the sphere due to its nearness to the sphere.

To begin, you will first make some additions to the `test-template.py` file for future convenience. In this file, add the following import statements:

```
from geometry.sphereGeometry import SphereGeometry
from light.ambientLight import AmbientLight
from light.directionalLight import DirectionalLight
from light.pointLight import PointLight
from material.flatMaterial import FlatMaterial
from material.lambertMaterial import LambertMaterial
from material.phongMaterial import PhongMaterial
```



FIGURE 6.14 Rendered scene with all light types and material types.

Next, make a copy of the template file and name it `test-6-1.py`. In this new file, in the `initialize` function, replace the code in that function, starting from the line where the camera object is created, with the following code, which will set up the main scene:

```
self.camera = Camera( aspectRatio=800/600 )
self.camera.setPosition( [0,0,6] )

ambient = AmbientLight( color=[0.1, 0.1, 0.1] )
self.scene.add( ambient )
directional = DirectionalLight(
    color=[0.8, 0.8, 0.8], direction=[-1, -1, -2] )
self.scene.add( directional )
point = PointLight(
    color=[0.9, 0, 0], position=[1, 1, 0.8] )
self.scene.add( point )
sphereGeometry = SphereGeometry()
flatMaterial = FlatMaterial(
    properties={ "baseColor" : [0.6, 0.2, 0.2] } )
grid = Texture("images/grid.png")
lambertMaterial = LambertMaterial( texture=grid )
phongMaterial = PhongMaterial(
    properties={ "baseColor" : [0.5, 0.5, 1] } )
sphere1 = Mesh(sphereGeometry, flatMaterial)
sphere1.setPosition( [-2.2, 0, 0] )
self.scene.add( sphere1 )
sphere2 = Mesh(sphereGeometry, lambertMaterial)
sphere2.setPosition( [0, 0, 0] )
self.scene.add( sphere2 )
sphere3 = Mesh(sphereGeometry, phongMaterial)
sphere3.setPosition( [2.2, 0, 0] )
self.scene.add( sphere3 )
```

Finally, change the last line of code in the file to the following, to make the window large enough to easily see all three spheres simultaneously:

```
Test( screenSize=[800,600] ).run()
```

When you run this program, you should see an image similar to that in Figure 6.14.

6.6 EXTRA COMPONENTS

In Chapter 4, the **AxesHelper** and **GridHelper** classes were introduced to help provide a sense of orientation and scale within a scene by creating simple meshes. In the same spirit, in this section you will create two additional classes, **PointLightHelper** and **DirectionalLightHelper**, to visualize the position of point lights and the direction of directional lights, respectively. Figure 6.15 illustrates the two helpers added to the scene from the previous test example. Note that a wireframe diamond shape is present at the location of the point light, and a small wireframe grid with a perpendicular ray illustrates the direction of the directional light. Furthermore, in both cases, the color of the helper objects is equal to the color of the associated lights.

First you will implement the directional light helper, which is a grid helper object with an additional line segment added. To proceed, in the **extras** folder, create a new file named **directionalLightHelper.py** containing the following code:

```
from extras.gridHelper import GridHelper
class DirectionalLightHelper(GridHelper):

    def __init__(self, directionalLight):
        color = directionalLight.color
        super().__init__(size=1, divisions=4,
```

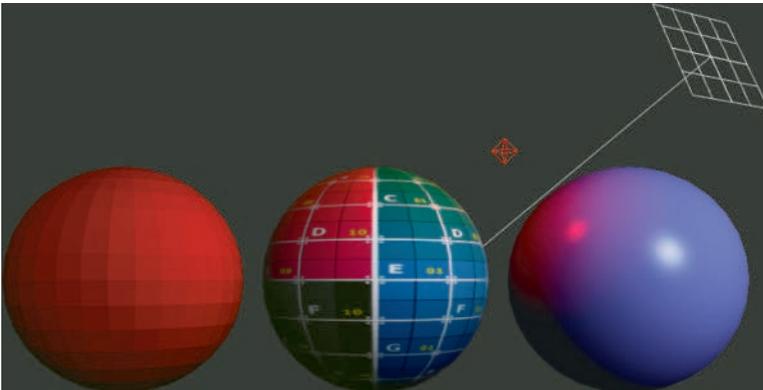


FIGURE 6.15 Objects illustrating the properties of point lights and directional lights.

```

        gridColor=color, centerColor=[1,1,1])
self.geometry.attributes["vertexPosition"].
    data += [[0,0,0], [0,0,-10]]
self.geometry.attributes["vertexColor"].data
    += [color, color]
self.geometry.attributes["vertexPosition"].
    uploadData()
self.geometry.attributes["vertexColor"].
    uploadData()
self.geometry.countVertices()

```

Next, you will implement the point light helper, which is a wireframe sphere geometry whose resolution parameters are small enough that the sphere becomes an octahedron. To proceed, in the `extras` folder, create a new file named `pointLightHelper.py` containing the following code:

```

from geometry.sphereGeometry import SphereGeometry
from material.surfaceMaterial import SurfaceMaterial
from core.mesh import Mesh

class PointLightHelper(Mesh):

    def __init__(self, pointLight, size=0.1,
                 lineWidth=1):
        color = pointLight.color
        geometry = SphereGeometry(radius=size,
                                   radiusSegments=4, heightSegments=2)
        material = SurfaceMaterial({
            "baseColor": color,
            "wireframe": True,
            "doubleSide": True,
            "lineWidth": linewidth
        })
        super().__init__(geometry, material)

```

Next, you will test these helper objects out by adding them to the previous test example. As an extra feature, you will also illustrate the dynamic lighting effects of the graphics framework and learn how to keep the light

source and helper objects in sync. In the file `test-6-1.py`, add the following import statements:

```
from extras.directionalLightHelper import
DirectionalLightHelper
from extras.pointLightHelper import PointLightHelper
from math import sin
```

Next, change the code where the directional and point lights are created to the following; the variable declarations are changed so that the lights can be accessed in the `update` function later.

```
self.directional = DirectionalLight(
    color=[0.8, 0.8, 0.8], direction=[-1, -1, -2] )
self.scene.add( self.directional )
self.point = PointLight(
    color=[0.9, 0, 0], position=[1, 1, 0.8] )
self.scene.add( self.point )
```

Then, also in the `initialize` function after the code that you just modified, add the following. Note that the helper objects are added to their corresponding lights rather than directly to the scene. This approach takes advantage of the scene graph structure and guarantees that the transformations of each pair will stay synchronized. In addition, the position of the directional light has been set. This causes no change in the effects of the directional light; it has been included to position the directional light helper object at a convenient location that does not obscure the other objects in the scene.

```
directHelper = DirectionalLightHelper(self.
    directional)
self.directional.setPosition( [3,2,0] )
self.directional.add( directHelper )
pointHelper = PointLightHelper( self.point )
self.point.add( pointHelper )
```

Finally, you will add some code that makes the point light move up and down while the directional light tilts from left to right. In the `update` function, add the following code:

```

self.directional.setDirection( [ -1, sin(0.7*self.
    time), -2] )
self.point.setPosition( [1, sin(self.time), 0.8] )

```

When you run the program, you should see the helper objects move as described, and the lighting on the spheres in the scene will also change accordingly.

6.7 BUMP MAPPING

Another effect that can be accomplished with the addition of lights is *bump mapping*: a technique for simulating details on the surface of an object by altering the normal vectors and using the adjusted vectors in lighting calculations. This additional normal vector detail is stored in a texture called a *bump map* or a *normal map*, in which the (r, g, b) values at each point correspond to the (x, y, z) values of a normal vector. This concept is illustrated in Figure 6.16. The left image in the figure shows a colored texture of a brick wall. The middle image in the figure shows the associated grayscale *height map*, in which light colors represent a large amount of displacement in the perpendicular direction, while dark colors represent a small amount. In this example, the white regions correspond to the bricks, which extrude slightly from the wall, while the darker regions correspond to the mortar between the bricks, which here appears pressed into the wall to a greater extent. The right image in the figure represents the normal map. Points that are shades of red represent normal vectors mainly oriented towards the positive x -axis, while green and blue amounts correspond to the y -axis and z -axis directions, respectively. Observe that in the normal map for this example, the top edge of each brick appears to be light green, while the right edge appears to be pink.

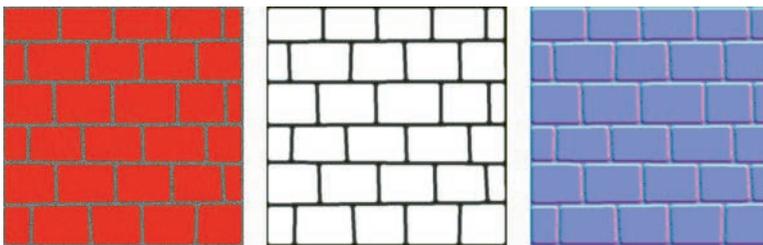


FIGURE 6.16 A color texture, height map texture, and normal map texture for a brick wall.

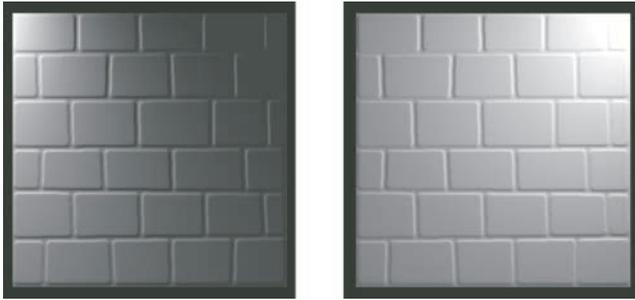


FIGURE 6.17 A normal map applied to a rectangle, with point light source on the upper-left (left) and upper right (right).

When normal map data is combined with normal data in the fragment shader, and the result is used in light calculations, the object in the resulting scene will appear to have geometric features that are not actually present in the vertex data. This is illustrated in Figure 6.17, where the bump map from Figure 6.16 has been applied to a rectangle geometry, and is lit by a point light from two different positions. The light and dark regions surrounding each brick create an illusion of depth even though the mesh itself is perfectly flat.

When a color texture and a bump map are used in combination, the results are subtle but significantly increase the realism of the scene. This is particularly evident in an interactive scene with dynamic lighting – a scene containing lights whose position, direction, or other properties change.

To implement bump mapping is fairly straightforward. The following modifications should be carried out for both the files `lambertMaterial.py` and `phongMaterial.py` in the `material` directory.

In the `__init__` function, add the parameter and default value `bumpTexture=None`.

In the fragment shader, before the `main` function, add the following uniform variable declarations.

```
uniform bool useBumpTexture;
uniform sampler2D bumpTexture;
uniform float bumpStrength;
```

Similar to the way the optional `texture` parameter works, if the `bumpTexture` parameter is set, then `useBumpTexture` will be set to `True`.

In this case, the normal data encoded within the bump texture will be multiplied by the strength parameter and added to the normal vector to produce a new normal vector. To implement this, in the `main` function in the fragment shader, change the lines of code involving the variable `total` (used for calculating the total light contribution) to the following:

```
vec3 bNormal = normal;
if (useBumpTexture)
    bNormal += bumpStrength * vec3(texture2D(
bumpTexture, UV ));
vec3 total = vec3(0,0,0);
total += lightCalc( light0, position, bNormal );
total += lightCalc( light1, position, bNormal );
total += lightCalc( light2, position, bNormal );
total += lightCalc( light3, position, bNormal );
color *= vec4( total, 1 );
```

Finally, you need to add the corresponding uniform data, which parallels the structure for the texture variable. Since texture slot 1 is already in use by the shader, texture slot 2 will be reserved for the bump texture. After the fragment shader code and before the function `locateUniforms` is called, add the following code:

```
if bumpTexture == None:
    self.addUniform("bool", "useBumpTexture", False)
else:
    self.addUniform("bool", "useBumpTexture", True)
    self.addUniform("sampler2D", "bumpTexture",
[bumpTexture.textureRef, 2])
    self.addUniform("float", "bumpStrength", 1.0)
```

With these additions, the graphics framework can now support bump mapping. To create an example, you can use the color and normal map image files provided with this library; alternatively, bump maps or normal maps may be easily found with an image-based internet search, or produced from height maps using graphics editing software such as the GNU Image Manipulation Program (GIMP), freely available at <http://gimp.org>.

To proceed, make a copy of the test template file and name it `test-6-2.py`. In this new file, in the `initialize` function, replace the code

in that function, after the line where the camera object is created, with the following code:

```
self.camera.setPosition( [0,0,2.5] )
ambientLight = AmbientLight( color=[0.3, 0.3, 0.3] )
self.scene.add( ambientLight )
pointLight = PointLight(
    color=[1,1,1], position=[1.2, 1.2, 0.3])
self.scene.add( pointLight )

colorTex = Texture("images/brick-color.png")
bumpTex = Texture("images/brick-bump.png")

geometry = RectangleGeometry(width=2, height=2)
bumpMaterial = LambertMaterial(
    texture=colorTex,
    bumpTexture=bumpTex,
    properties={"bumpStrength": 1}
)
mesh = Mesh(geometry, bumpMaterial)
self.scene.add(mesh)
```

When you run the application, you should see a scene similar to that in Figure 6.18. To fully explore the effects of bump mapping, you may want to add a movement rig, animate the position of the light along a path (similar to the previous example) or apply the material to a different geometric surface, such as a sphere.

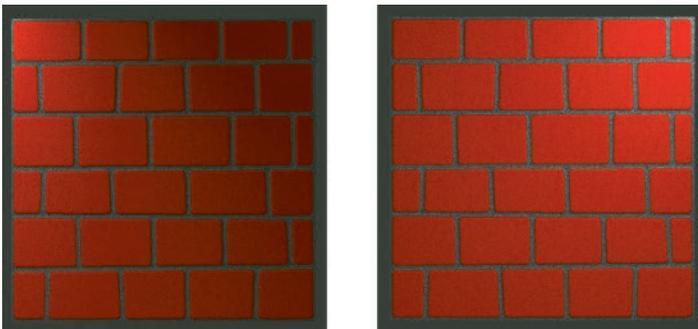


FIGURE 6.18 Combining color texture with normal map texture.

6.8 BLOOM AND GLOW EFFECTS

In this section, you will learn how to implement light-inspired postprocessing effects. The first of these is called *light bloom*, or just *bloom*, which simulates the effect of an extremely bright light overloading the sensors in a real-world camera, causing the edges of bright regions to blur beyond their natural borders. Figure 6.19 illustrates a scene containing a number of crates in front of a simulated light source. The left side of the figure shows the scene without a bloom effect, and the right side shows the scene with the bloom effect, creating the illusion of very bright lights.

A similar combination of postprocessing filters can be used to create a glow effect, in which objects appear to radiate a given color. Figure 6.20

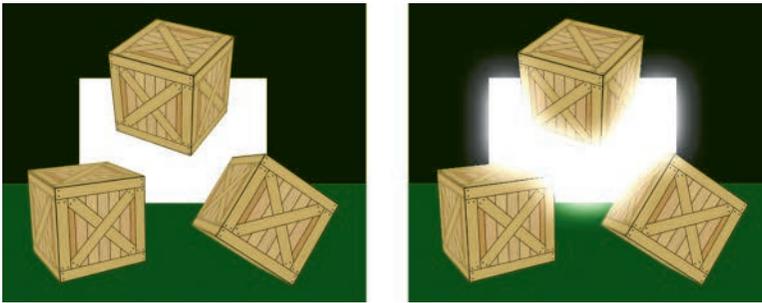


FIGURE 6.19 Light bloom postprocessing effect.

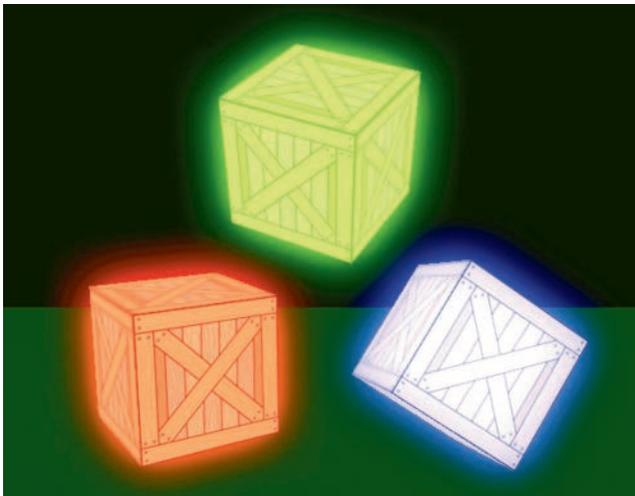


FIGURE 6.20 Glow postprocessing effect.

illustrates a scene similar to Figure 6.19, but with the background light source removed, and each of the three crates appears to be glowing a different color. Most notably, in contrast to the bloom effect, the colors used for glow do not need to appear in the original scene.

These techniques require three new postprocessing effects: brightness filtering, blurring, and additive blending. These effects will be created in the same style used in Section 5.9 on postprocessing in Chapter 5, starting from the template effect file. To prepare for testing these effects, make a copy of the file `test-5-12.py` and name it `test-6-3.py`. Make sure that in the line of code where the renderer is created, the clear color is set to black by including the parameter `clearColor=[0,0,0]`; any other clear color may cause unexpected effects in the rendered results. After writing the code for each effect, you can test it in the application by adding the import statement corresponding to the effect and changing the effect that is added to the postprocessor. Recall that if no effects are added to the postprocessor, then the original scene is rendered, illustrated in Figure 6.21; this will serve as a baseline for visual comparison with the postprocessing effects that follow.

First, you will create the brightness filter effect, while only renders the pixels with a certain brightness, as illustrated in Figure 6.22.

This effect is accomplished by only rendering a fragment if the sum of the red, green, and blue components is greater than a given threshold value; otherwise, the fragment is discarded. You will add the threshold value as

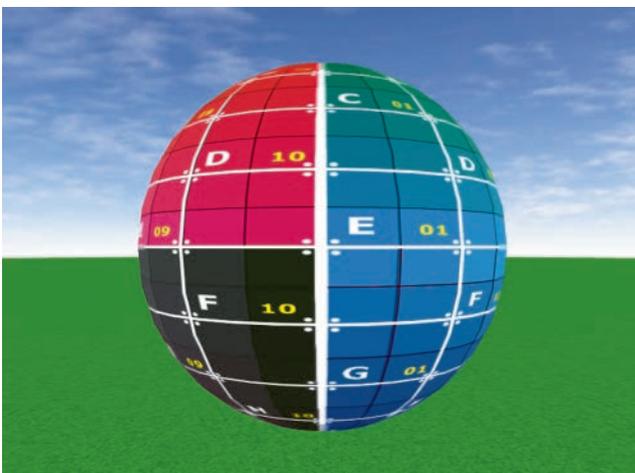


FIGURE 6.21 Default scene with no postprocessing effects.

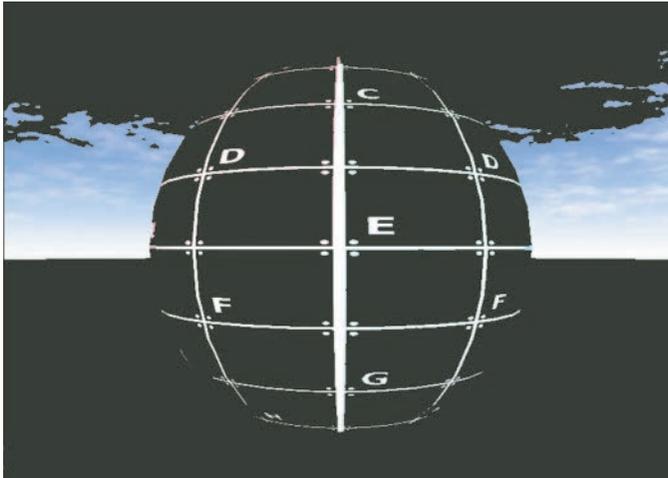


FIGURE 6.22 Brightness filter postprocessing effect.

a parameter in the class constructor, and create a corresponding uniform variable in the shader and uniform object in the class. To implement this, in the `effects` folder, make a copy of the `templateEffect.py` file and name it `brightFilterEffect.py`. In the new file, change the name of the class to `BrightFilterEffect`, and change the initialization function declaration to the following:

```
def __init__(self, threshold=2.4):
```

Next, change the fragment shader code to the following:

```
in vec2 UV;
uniform sampler2D texture;
uniform float threshold;
out vec4 fragColor;

void main()
{
    vec4 color = texture2D(texture, UV);
    if (color.r + color.g + color.b < threshold)
        discard;
    fragColor = color;
}
```

Finally, add the following line of code near the end of the file, before the `locateUniforms` function is called.

```
self.addUniform("float", "threshold", threshold)
```

This completes the code for the brightness filter effect; recall that you may use the file `test-6-3.py` to test this effect, in which case you should see a result similar to Figure 6.22.

The next effect you will create is a blur effect, which blends the colors of adjacent pixels within a given radius. For computational efficiency, this is typically performed in two passes: first, a weighted average of pixel colors is performed along the horizontal direction (called a *horizontal blur*), and then, these results are passed into a second shader where a weighted average of pixel colors is performed along the vertical direction (called a *vertical blur*). Figure 6.23 shows the results of applying the horizontal and vertical blur effects separately to the base scene, while Figure 6.24 shows the results of applying these effects in sequence, resulting in blur in all directions.

To create the horizontal blur effect, you will sample the pixels within the bounds specified by a parameter named `blurRadius`. Since textures are sampled using UV coordinates, the shader will also need the dimensions of the rendered image (a parameter named `textureSize`) to calculate the pixel-to-UV coordinate conversion factor (which is $1/\text{textureSize}$). Then, within the fragment shader, a `for` loop will calculate a weighted average of colors along this line, with the greatest weight applied to the pixel at the original UV coordinates, and the weights decreasing linearly

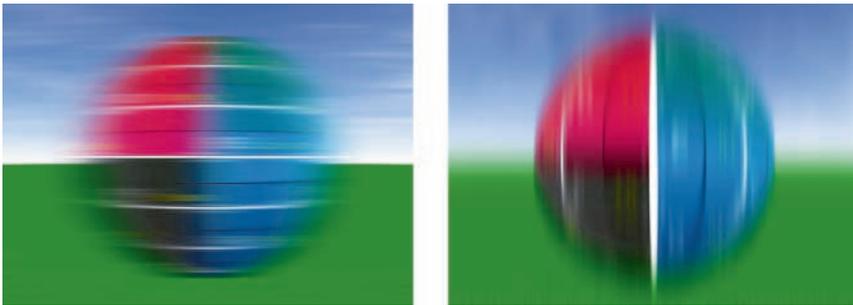


FIGURE 6.23 Horizontal blur (left) and vertical blur (right).

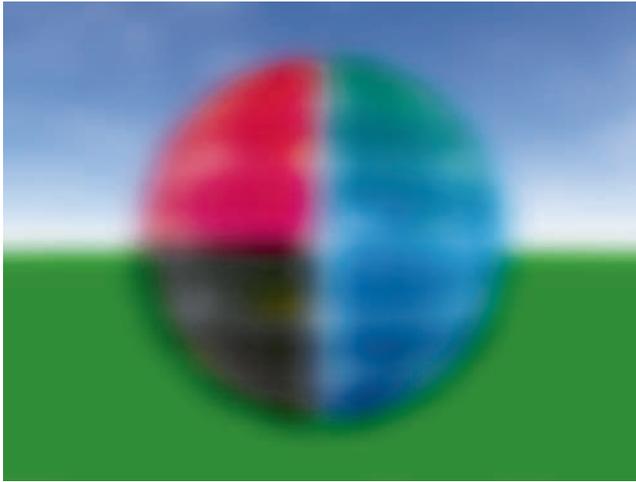


FIGURE 6.24 A combined two-pass blur postprocessing effect.

towards the ends of the sample region. The sum of these colors is normalized by dividing by the sum of the weights, which is equal to the alpha component of the sum (since the original alpha component at each point equals 1). To implement this, in the `effects` folder, make a copy of the `templateEffect.py` file and name it `horizontalBlurEffect.py`. In the new file, change the name of the class to `HorizontalBlurEffect`, and change the initialization function declaration to the following:

```
def __init__(self, textureSize=[512,512],
             blurRadius=20):
```

Next, change the fragment shader code to the following:

```
in vec2 UV;
uniform sampler2D texture;
uniform vec2 textureSize;
uniform int blurRadius;
out vec4 fragColor;
void main()
{
    vec2 pixelToTextureCoords = 1 / textureSize;
    vec4 averageColor = vec4(0,0,0,0);
    for (int offsetX = -blurRadius; offsetX <=
         blurRadius; offsetX++)
```

```

    {
        float weight = blurRadius - abs(offsetX) + 1;
        vec2 offsetUV = vec2(offsetX, 0) *
            pixelToTextureCoords;
        averageColor += texture2D(texture, UV +
            offsetUV) * weight;
    }
    averageColor /= averageColor.a;
    fragColor = averageColor;
}

```

Finally, add the following code near the end of the file, before the `locateUniforms` function is called.

```

self.addUniform("vec2", "textureSize", textureSize)
self.addUniform("int", "blurRadius", blurRadius)

```

This completes the code for the horizontal blur effect. The vertical blur effect works in the same way, except that the texture is sampled along vertical lines. To create this effect, make a copy of the `horizontalBlurEffect.py` file and name it `verticalBlurEffect.py`. In the new file, change the name of the class to `VerticalBlurEffect`. The only change that needs to be made is the for loop in the fragment shader, which should be changed to the following:

```

for (int offsetY = -blurRadius; offsetY <=
    blurRadius; offsetY++)
{
    float weight = blurRadius - abs(offsetY) + 1;
    vec2 offsetUV = vec2(0, offsetY) *
        pixelToTextureCoords;
    averageColor += texture2D(texture, UV + offsetUV)
        * weight;
}

```

At this point, the blur shader effects are complete and can be applied individually or in sequence to create images with effects similar to those seen in Figures 6.23 and 6.24.

The next effect you will create is an additive blend effect, where an additional texture is overlaid on a rendered scene, using a weighted sum of the individual pixel colors. In some applications, color values are multiplied

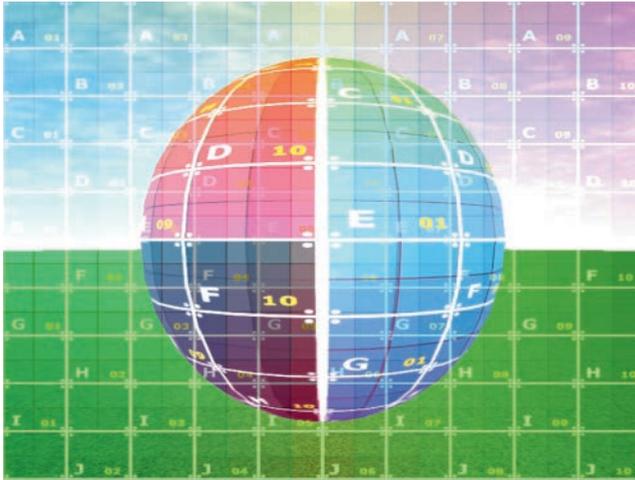


FIGURE 6.25 Additive blending postprocessing effect.

together; this is particularly useful for shading, as the component values of a color are between 0 and 1, and multiplying by values in this range decreases values and darkens the associated colors. Alternatively, adding color components increases values and brightens the associated colors, which is particularly appropriate when simulating light-based effects. This effect is illustrated in Figure 6.25, where the original scene is additively blended with the grid texture.

To implement this, in the `effects` folder, make a copy of the `templateEffect.py` file and name it `additiveBlendEffect.py`. In the new file, change the name of the class to `AdditiveBlendEffect`, and change the initialization function declaration to the following:

```
def __init__(self, blendTexture=None,
             originalStrength=1, blendStrength=1):
```

Next, change the fragment shader code to the following:

```
in vec2 UV;
uniform sampler2D texture;
uniform sampler2D blendTexture;
uniform float originalStrength;
uniform float blendStrength;
out vec4 fragColor;
```

```

void main()
{
    vec4 originalColor = texture2D(texture, UV);
    vec4 blendColor = texture2D(blendTexture, UV);
    vec4 color = originalStrength * originalColor +
                blendStrength * blendColor;
    fragColor = color;
}

```

Finally, add the following code near the end of the file, before the `locateUniforms` function is called. Note that since texture slot 1 is used for the original texture, texture slot 2 will be reserved for the blended texture.

```

self.addUniform("sampler2D", "blendTexture",
               [blendTexture.textureRef, 2])
self.addUniform("float", "originalStrength",
               originalStrength)
self.addUniform("float", "blendStrength",
               blendStrength)

```

With these additions, the additive blend shader effect is complete.

To combine these effects to create a light bloom effect, assuming that all the necessary imports have been added to `test-6-3.py`, add effects to the postprocessor object as follows:

```

# combined effects to create light bloom
self.postprocessor.addEffect( BrightFilterEffect(2.4) )
self.postprocessor.addEffect( HorizontalBlurEffect(
    textureSize=[800,600], blurRadius=50) )
self.postprocessor.addEffect( VerticalBlurEffect(
    textureSize=[800,600], blurRadius=50) )
mainScene = self.postprocessor.renderTargetList[0].
    texture
self.postprocessor.addEffect( AdditiveBlendEffect(
    mainScene, originalStrength=2, blendStrength=1) )

```

Note that the results of the first render pass (the original scene) are accessed through the postprocessor object and blended with the results of the bright filtered light after the light has been blurred. Running this application will result in an image similar to that shown in Figure 6.26.

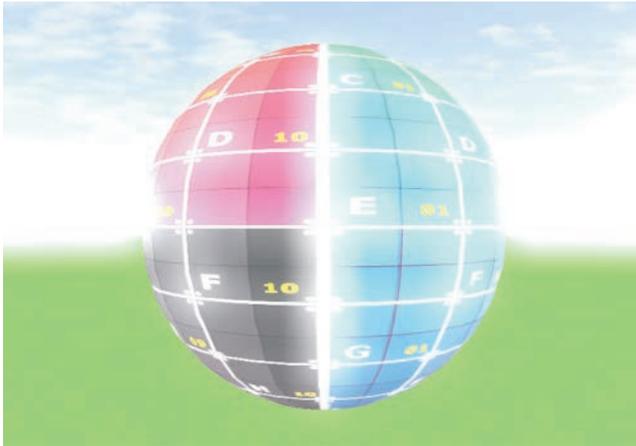


FIGURE 6.26 Light bloom postprocessing effect.

Next, you will use these shaders to create a glow effect. One method to implement glow is to use a second scene, referred to here as the glow scene, containing only the objects that should glow. (This is analogous to the brightness filter step used in creating the light bloom effect.) The objects in the glow scene should use the same geometry data and transform matrices as their counterparts in the original scene, but in the glow scene, they will be rendered with a solid colored material corresponding to the desired glow color. Two postprocessing objects are then used to accomplish the glow effect. The first renders the glow scene, applies a blur filter, and stores the result in a render target (accomplished by setting the **finalRenderTarget** parameter). The second renders the original scene and then applies an additive blend effect using the results from the first postprocessor. Creating a red glow effect applied to the sphere in the main scene in this section will produce an image similar to Figure 6.27.

To implement this example, make a copy of the file **test-6-3.py** and name it **test-6-4.py**. Add the following import statements:

```
from material.surfaceMaterial import SurfaceMaterial
from core.renderTarget import RenderTarget
```

In the **initialize** function, make sure that the line of code that initializes the renderer is as follows (otherwise, your scene may appear oversaturated).

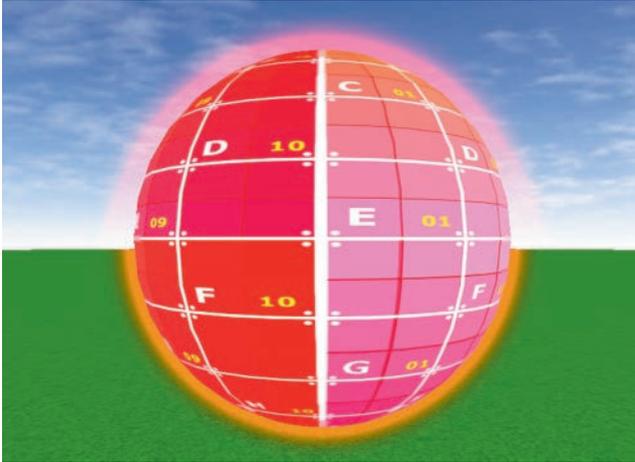


FIGURE 6.27 Glow postprocessing effect.

```
self.renderer = Renderer( clearColor=[0,0,0] )
```

Next, in the `initialize` function, replace all the code involving postprocessing with the following:

```
# glow scene
self.glowScene = Scene()
redMaterial = SurfaceMaterial({"baseColor": [1,0,0]})
glowSphere = Mesh(sphereGeometry, redMaterial)
glowSphere.transform = self.sphere.transform
self.glowScene.add( glowSphere )

# glow postprocessing
glowTarget = RenderTarget( resolution=[800,600] )
self.glowPass = Postprocessor(self.renderer,
    self.glowScene, self.camera, glowTarget)
self.glowPass.addEffect( HorizontalBlurEffect(
    textureSize=[800,600], blurRadius=50) )
self.glowPass.addEffect( VerticalBlurEffect(
    textureSize=[800,600], blurRadius=50) )

# combining results of glow effect with main scene
self.comboPass = Postprocessor(self.renderer,
    self.scene, self.camera)
```

```
self.comboPass.addEffect( AdditiveBlendEffect(
    glowTarget.texture, originalStrength=1,
    blendStrength=3) )
```

Finally, replace the code in the update function with the following:

```
self.glowPass.render()
self.comboPass.render()
```

Running this application should produce a result similar to Figure 6.27. If desired, the intensity of the glow can be changed by altering the value of the `blendStrength` parameter in the additive blend effect.

6.9 SHADOWS

In this section, you will add shadow rendering functionality to the graphics framework. In addition to adding further realism to a scene, shadows can also be fundamental for estimating relative positions between objects. For example, the left side of Figure 6.28 shows a scene containing a ground and a number of crates; it is difficult to determine whether the crates are resting on the ground. The right side of Figure 6.28 adds shadow effects, which provide visual cues to the viewer so that they may gain a better understanding of the arrangement of the objects in the scene.

6.9.1 Theoretical Background

In this graphics framework, shadows will be based on a single directional light. By definition, the light rays emitted by a directional light have a constant direction and are not affected by distance; these qualities will also be present in the corresponding shadows. A point will be considered to be

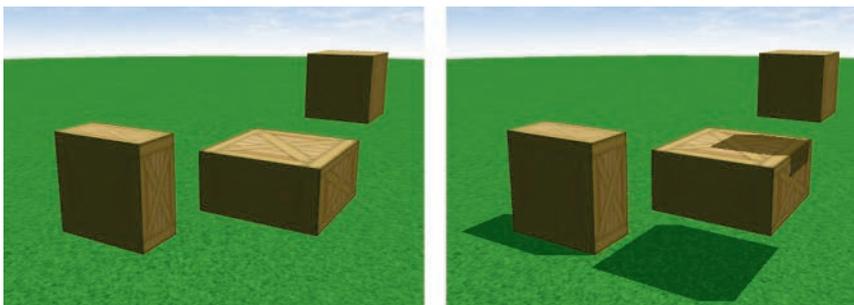


FIGURE 6.28 A scene without shadows (left) and with shadows (right).

“in shadow” (or more precisely, in the shadow of another object) when there is another point along the direction of the light ray that is closer to the light source. In this case, the closer point is also said to have “cast a shadow” on the more distant point. The colors corresponding to a point in shadow will be darkened during the rendering process, which simulates a reduced amount of light impacting the surface at that point.

The data required for creating shadows will be gathered in a rendering pass called a *shadow pass*, performed before the main scene is rendered. The purpose of the shadow pass is to render the scene from the position and direction of the light source to determine what points are “visible” to the light. The visible points are considered to be illuminated by light rays, and no further shading will be applied. The points that are not visible from the light are considered to be in shadow and will be darkened.

Recall that during the pixel processing stage of the graphics pipeline, for each pixel in the rendered image, the depth buffer stores the distance from the viewer to the corresponding point in the scene. This information is used during the rendering process when a fragment would correspond to the same pixel as a previously processed fragment, in order to determine whether the new fragment is closer to the viewer, in which case the new fragment’s data overwrites the data currently stored in the color and depth buffers. In the final rendered image of a scene, each pixel corresponds to a fragment that is the shortest distance from the camera, as compared to all other fragments that would correspond to the same pixel. This “shortest distance” information from the depth buffer can be used to generate shadows according to the following algorithm (called *shadow mapping*):

- Render the scene from the point of view of the directional light and store the depth buffer values.
- When rendering the main scene, calculate the distance from each fragment to the light source. If this distance is greater than the corresponding stored depth value, then the fragment is not closest to the light source, and therefore, it is in shadow.

The depth buffer values that are generated during the shadow pass will be stored in a texture called the *depth texture*, which will contain grayscale colors at each pixel based on the corresponding depth value. Due to the default configuration of the depth test function, depth values near 0 (corresponding to dark colors) represent nearby points. Conversely, depth

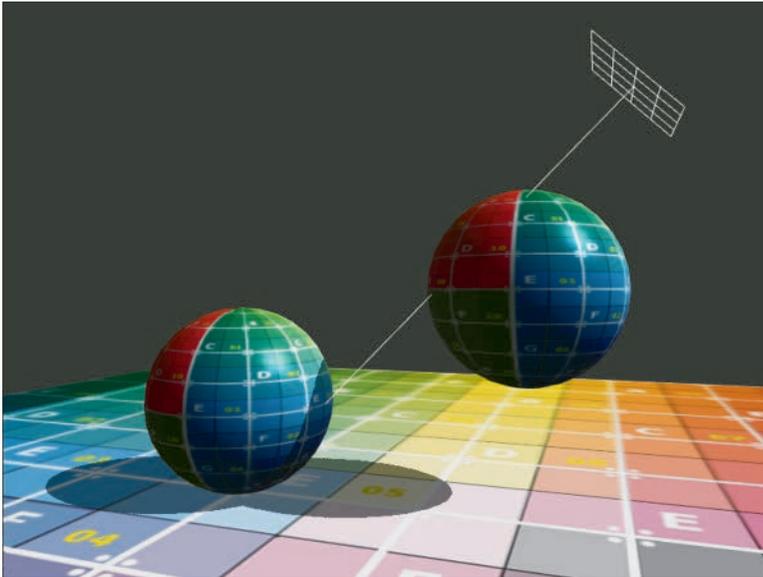


FIGURE 6.29 A scene including a directional light and shadows.

values near 1 (corresponding to light colors) represent more distant points. To help visualize these concepts, Figure 6.29 depicts a scene, including shadows cast by a directional light, rendered from the point of view of a perspective camera. The position and direction of the directional light are indicated by a directional light helper object. On the left side of Figure 6.30 we see the same scene, rendered by a camera using an orthographic projection, from the point of view of the directional light. For convenience, this secondary camera will be referred to as the *shadow camera*. Note that no shadows are visible from this point of view; indeed, the defining characteristic of shadows is that they are exactly the set of points *not* visible from the shadow camera. The right side of Figure 6.30 shows the corresponding grayscale depth texture that will be produced during the shadow pass.

When rendering shadows with this approach, there are some limitations and constraints that should be kept in mind when setting up a scene. First, the appearance of the shadows is affected by the resolution of the depth texture; low resolutions will lead to shadows that appear pixelated, as illustrated in Figure 6.31. Second, the points in the scene that may cast shadows or be in shadow are precisely those points contained in the volume rendered by the shadow camera. One could increase the viewing

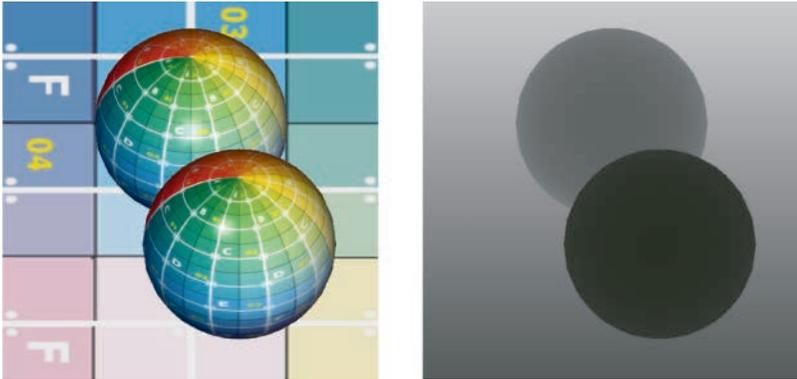


FIGURE 6.30 A scene viewed from a directional light (left) and the corresponding depth texture (right).



FIGURE 6.31 Shadow pixelation artifacts.

bounds of the shadow camera to encompass a larger area, but unless the resolution of the depth texture is increased proportionally, there will be fewer pixels corresponding to each unit of world space, and the pixelation of the shadows will increase. To reduce shadow pixelation, one typically adjusts the shadow camera bounding parameters to closely fit the region of the scene involving shadows.

A point will be considered to be in shadow when two conditions are true: the surface must be facing the light at the point in question, and the distance from the point to the light must be greater than the value stored in the depth texture (indicating that a closer point exists and is casting a shadow on this point). The first of these conditions can be checked by examining the angle between the light direction and the normal vector to the surface at that point. If the cosine of that angle is greater than 0, then the angle is less than 90° , indicating that the surface is indeed facing the

light at that point. (Since this calculation requires interpolated normal vectors for each fragment, the shadow calculations will only be implemented for the Lambert and Phong materials in this framework.)

To check the second condition, depth information must be extracted from the depth texture. Selecting the correct point from the depth texture requires us to know where a particular fragment would appear if it were being rendered during the shadow pass. Calculating this information during the normal rendering pass requires access to the information stored by the shadow camera object (its projection matrix and view matrix). Position calculations are most efficiently handled in a vertex shader, where the standard model/view/projection matrix multiplications are typically performed. The result will be in clip space coordinates, where the coordinates of points in the visible region are each in the range from -1 to 1 . Once this calculation is performed on the vertex position, this value (which will be called the *shadow position*, as it is derived from shadow camera data) will be transmitted to the fragment shader and interpolated for each fragment.

In the fragment shader, the coordinates of the shadow position variable can be used to calculate and recover the required depth values. The x and y coordinates can be used to derive the coordinates of the corresponding pixel in the rendered image (as viewed from the shadow camera), or more importantly, the UV coordinates corresponding to that pixel, which are needed to retrieve a value stored in the depth texture. Since clip space coordinates range from -1 to 1 , while UV coordinates range from 0 to 1 , the shadow position coordinates need to be transformed accordingly before sampling the texture. Recall that the value retrieved from the depth texture represents the distance to the closest point (relative to the shadow camera frame of reference) and is in the range from 0 to 1 . The distance from the fragment being processed to the shadow camera is stored in the z component of the shadow position variable. After converting this value to the range from 0 to 1 , you can compare the distance from the fragment to the shadow camera with the closest stored distance to the shadow camera. If the fragment distance is greater, then a different point is closer to the shadow camera along this direction, in which case the fragment is considered to be in shadow and its color will be adjusted accordingly.

With this knowledge, you are now prepared to add shadow effects to the graphics framework.

6.9.2 Adding Shadows to the Framework

The steps involved in adding shadow casting functionality to the graphics framework are similar to the steps involved in adding the lighting effects at the beginning of this chapter. First, a special material called **DepthMaterial** will be created to generate the depth texture during the shadow pass. Then, a **Shadow** class will be created to store the objects necessary for shadow calculations (including a reference to the directional light, the shadow camera, and a render target to be used during the shadow pass). In the **LambertMaterial** and **PhongMaterial** classes, a shadow struct will be defined to group related variables used in the shadow calculations. Then, the vertex and fragment shaders of both these materials will be updated with additional uniform objects and code. The **Uniform** class will be extended to store **Shadow** objects and upload data to the corresponding fields in a uniform shadow variable in the shaders. Finally, the **Renderer** class will be updated with a new `enableShadows` function, which will generate a **Shadow** object and cause a shadow pass to be performed by the `render` function before the main scene is rendered. As usual, after the framework classes have been updated, you will create an interactive scene to verify that everything works as expected.

To begin, you will first create the **DepthMaterial** class. In the `material` folder, create a new file named `depthMaterial.py` containing the following code. Note the use of the built-in GLSL variable `gl_FragCoord`, whose *z* coordinate stores the depth value. Since the fragment shader is using grayscale colors to encode depth values in the texture, any component of the texture color may be used later on to retrieve this “closest distance to light” information.

```
from material.material import Material

class DepthMaterial(Material):

    def __init__(self):
        # vertex shader code
        vertexShaderCode = """
        in vec3 vertexPosition;
        uniform mat4 projectionMatrix;
        uniform mat4 viewMatrix;
        uniform mat4 modelMatrix;
```

```

void main()
{
    gl_Position = projectionMatrix *
        viewMatrix *
        modelMatrix * vec4(vertexPosition, 1);
}
"""

# fragment shader code
fragmentShaderCode = """
out vec4 fragColor;
void main()
{
    float z = gl_FragCoord.z;
    fragColor = vec4(z, z, z, 1);
}
"""

# initialize shaders
super().__init__(vertexShaderCode,
    fragmentShaderCode)
self.locateUniforms()

```

Next, you will create a **Shadow** class to store the objects necessary for the shadow mapping algorithm previously described. For points in the scene that do not map to pixels within the depth texture, the pixel should be colored white, which will prevent shadows from being generated at that point. For this reason, it is important to use the texture parameter wrap setting `CLAMP_TO_BORDER`; the default texture border color was already set to white in the **Texture** class. In the **light** folder, create a new file named **shadow.py** with the following code:

```

from core.camera import Camera
from core.renderTarget import RenderTarget
from material.depthMaterial import DepthMaterial
from OpenGL.GL import *

class Shadow(object):

    def __init__(self, lightSource, strength=0.5,
        resolution=[512,512],

```

```

        cameraBounds=[-5,5, -5,5, 0,20],
        bias=0.01):

    super().__init__()

    # must be directional light
    self.lightSource = lightSource

    # camera used to render scene from perspective
    # of light
    self.camera = Camera()
    left, right, bottom, top, near, far =
        cameraBounds
    self.camera.setOrthographic(left, right, bottom,
                               top, near, far)
    self.lightSource.add( self.camera )
    # target used during the shadow pass,
    # contains depth texture
    self.renderTarget = RenderTarget( resolution,
        properties={"wrap": GL_CLAMP_TO_BORDER} )
    # render only depth data to target texture
    self.material = DepthMaterial()
    # controls darkness of shadow
    self.strength = strength
    # used to avoid visual artifacts
    # due to rounding/sampling precision issues
    self.bias = bias

    def updateInternal(self):
        self.camera.updateViewMatrix()
        self.material.uniforms["viewMatrix"].data =
            self.camera.viewMatrix
        self.material.uniforms["projectionMatrix"].
            data = self.camera.projectionMatrix

```

Next, you will need to update both the Lambert and Phong materials to support shadow effects. The following modifications should be made to both the files `lambertMaterial.py` and `phongMaterial.py` in the `material` directory.

In the `__init__` function, add the parameter and default value `useShadow=False`, which will need to be set to `True` when the material is created if shadow effects are desired. Since shadow-related calculations will take place in both the vertex and fragment shader, in the code for each, add the following struct definition before the `main` function:

```
struct Shadow
{
    // direction of light that casts shadow
    vec3 lightDirection;

    // data from camera that produces depth texture
    mat4 projectionMatrix;
    mat4 viewMatrix;

    // texture that stores depth values from shadow
    // camera
    sampler2D depthTexture;

    // regions in shadow multiplied by (1-strength)
    float strength;

    // reduces unwanted visual artifacts
    float bias;
};
```

After the `Shadow` struct definition in the vertex shader, add the following variables:

```
uniform bool useShadow;
uniform Shadow shadow0;
out vec3 shadowPosition0;
```

In the vertex shader `main` function, add the following code, which calculates the position of the vertex relative to the shadow camera.

```
if (useShadow)
{
    vec4 temp0 = shadow0.projectionMatrix * shadow0.
        viewMatrix *
        modelMatrix * vec4(vertexPosition, 1);
```

```

    shadowPosition0 = vec3( temp0 );
}

```

After the Shadow struct definition in the fragment shader, add the following variables:

```

uniform bool useShadow;
uniform Shadow shadow0;
in vec3 shadowPosition0;

```

In the fragment shader **main** function, before the value of **frag-Color** is set, add the following code, which determines if the surface is facing towards the light direction, and determines if the fragment is in the shadow of another object. When both of these conditions are true, the fragment color is darkened by multiplying the **color** variable by a value based on the shadow strength parameter.

```

if (useShadow)
{
    // determine if surface is facing towards light
    // direction
    float cosAngle = dot( normalize(normal),
        -normalize(shadow0.lightDirection) );
    bool facingLight = (cosAngle > 0.01);

    // convert range [-1, 1] to range [0, 1]
    // for UV coordinate and depth information
    vec3 shadowCoord = ( shadowPosition0.xyz + 1.0 )
        / 2.0;
    float closestDistanceToLight = texture2D(
        shadow0.depthTexture, shadowCoord.xy).r;
    float fragmentDistanceToLight =
        clamp(shadowCoord.z, 0, 1);
    // determine if fragment lies in shadow of another
    // object
    bool inShadow = ( fragmentDistanceToLight >
        closestDistanceToLight + shadow0.bias );

    if (facingLight && inShadow)
    {
        float s = 1.0 - shadow0.strength;

```

```

        color *= vec4(s, s, s, 1);
    }
}

```

Finally, in the part of the class where uniform data is added and before the `locateUniforms` function is called, add the following code. Similar to code for adding Light uniforms, the data for the Shadow uniform will be supplied by the `Renderer` class.

```

if not useShadow:
    self.addUniform("bool", "useShadow", False)
else:
    self.addUniform("bool", "useShadow", True)
    self.addUniform("Shadow", "shadow0", None)

```

Now that the contents of the `Shadow` class and the related struct are understood, you will update the `Uniform` class to upload this data as needed. In the file `uniform.py` in the `core` folder, in the `locateVariable` function, add the following code as a new case within the `if-else` block:

```

elif self.dataType == "Shadow":
    self.variableRef = {}
    self.variableRef["lightDirection"] =
        glGetUniformLocation(programRef,
            variableName + ".lightDirection")
    self.variableRef["projectionMatrix"] =
        glGetUniformLocation(programRef,
            variableName + ".projectionMatrix")
    self.variableRef["viewMatrix"] =
        glGetUniformLocation(programRef, variableName +
            ".viewMatrix")
    self.variableRef["depthTexture"] =
        glGetUniformLocation(programRef,
            variableName + ".depthTexture")
    self.variableRef["strength"] =
        glGetUniformLocation(programRef, variableName +
            ".strength")
    self.variableRef["bias"] =
        glGetUniformLocation(programRef, variableName +
            ".bias")

```

Then, in the **Uniform** class **uploadData** function, add the following code as a new case within the **if-else** block. (The texture unit reference value was chosen to be a value not typically used by other textures.)

```
elif self.dataType == "Shadow":

    direction = self.data.lightSource.getDirection()
    glUniform3f( self.variableRef["lightDirection"],
                 direction[0], direction[1], direction[2] )

    glUniformMatrix4fv( self.
                        variableRef["projectionMatrix"],
                        1, GL_TRUE, self.data.camera.projectionMatrix )
    glUniformMatrix4fv( self.
                        variableRef["viewMatrix"],
                        1, GL_TRUE, self.data.camera.viewMatrix )

    # configure depth texture
    textureObjectRef = self.data.renderTarget.texture.
        textureRef
    textureUnitRef = 15
    glActiveTexture( GL_TEXTURE0 + textureUnitRef )
    glBindTexture( GL_TEXTURE_2D, textureObjectRef )
    glUniform1i( self.variableRef["depthTexture"],
                 textureUnitRef )

    glUniform1f( self.variableRef["strength"], self.
                 data.strength )
    glUniform1f( self.variableRef["bias"], self.data.
                 bias )
```

The final set of changes and additions involve the **Renderer** class. To begin, add the following import statement:

```
from light.shadow import Shadow
In the __init__ function, add the following line of
code:
self.shadowsEnabled = False
```

After the `__init__` function, add the following function which will enable the shadow pass that will soon be added to the **render** function.

The only required parameter is `shadowLight`, the directional light that will be used to cast shadows.

```
def enableShadows(self, shadowLight, strength=0.5,
resolution=[512,512]):
    self.shadowsEnabled = True
    self.shadowObject = Shadow(shadowLight,
strength=strength, resolution=resolution)
```

The main addition to the `Renderer` class is the shadow pass. The collection of mesh objects in the scene must be gathered into a list before the shadow pass, therefore the corresponding block of code will be moved, as shown in what follows.

During the shadow pass, the framebuffer stored in the shadow render target will be used, and buffers cleared as normal. If there are no objects in the scene to generate a color for a particular pixel in the depth texture, then that pixel should be colored white, which will prevent a shadow from being generated at that location, and therefore, this is used as the clear color for the shadow pass. Much of the remaining code that follows is a simplified version of a standard render pass, as only one material (the depth material) will be used, and only triangle-based meshes need to be included at this stage. To proceed, at the beginning of the `render` function, add the following code:

```
# filter descendents
descendentList = scene.getDescendentList()
meshFilter = lambda x : isinstance(x, Mesh)
meshList = list( filter( meshFilter, descendentList )
)

# shadow pass
if self.shadowsEnabled:
    # set render target properties
    glBindFramebuffer(GL_FRAMEBUFFER,
self.shadowObject.renderTarget.framebufferRef)
    glViewport(0,0, self.shadowObject.renderTarget.
width, self.shadowObject.renderTarget.height)

    # set default color to white,
    # used when no objects present to cast shadows
    glClearColor(1,1,1,1)
```

```

glClear(GL_COLOR_BUFFER_BIT)
glClear(GL_DEPTH_BUFFER_BIT)

# everything in the scene gets rendered with
    depthMaterial
#   so only need to call glUseProgram & set
    matrices once
glUseProgram( self.shadowObject.material.
    programRef )
self.shadowObject.updateInternal()

for mesh in meshList:
    # skip invisible meshes
    if not mesh.visible:
        continue

    # only triangle-based meshes cast shadows
    if mesh.material.settings["drawStyle"] !=
        GL_TRIANGLES:
        continue

    # bind VAO
    glBindVertexArray( mesh.vaoRef )

    # update transform data
    self.shadowObject.material.
        uniforms["modelMatrix"].data =
            mesh.getWorldMatrix()

    # update uniforms (matrix data) stored in
        shadow material
    for varName, unifObj in
        self.shadowObject.material.
            uniforms.items():
        unifObj.uploadData()
glDrawArrays( GL_TRIANGLES, 0, mesh.geometry.
    vertexCount )

```

Finally, in the standard rendering part of the **render** function, the **Shadow** object needs to be copied into the data variable of the corresponding **Uniform** object, if shadow rendering has been enabled and if

such a uniform exists in the material of the mesh being drawn at that stage. To implement this, in the final **for** loop that iterates over **meshList**, in the section where mesh material uniform data is set and before the **uploadData** function is called, add the following code:

```
# add shadow data if enabled and used by shader
if self.shadowsEnabled and "shadow0" in mesh.material.
    uniforms.keys():
    mesh.material.uniforms["shadow0"].data = self.
        shadowObject
```

This completes the additions to the **Renderer** class in particular and support for rendering shadows in the graphics framework in general. You are now ready to create an example to produce a scene similar to that illustrated in Figure 6.28.

In your main project direction, create a new file named **test-6-5.py** containing the following code, presented here in its entirety for simplicity. Note the inclusion of commented out code, which can be used to illustrate the dynamic capabilities of shadow rendering, render the scene from the shadow camera perspective, or display the depth texture on a mesh within the scene.

```
from core.base      import Base
from core.renderer  import Renderer
from core.scene     import Scene
from core.camera    import Camera
from core.mesh      import Mesh
from core.texture   import Texture
from lights.ambientLight      import AmbientLight
from lights.directionalLight  import DirectionalLight
from material.phongMaterial    import PhongMaterial
from geometry.rectangleGeometry import
    RectangleGeometry
from geometry.sphereGeometry  import SphereGeometry
from extras.movementRig       import MovementRig
from extras.directionalLightHelper import
    DirectionalLightHelper
# testing shadows
class Test(Base):

    def initialize(self):
```

```

self.renderer = Renderer([0.2, 0.2, 0.2])
self.scene    = Scene()
self.camera   = Camera( aspectRatio=800/600 )
self.rig      = MovementRig()
self.rig.add( self.camera )
self.rig.setPosition( [0,2,5] )

ambLight = AmbientLight( color=[0.2, 0.2, 0.2] )
self.scene.add( ambLight )

self.dirLight = DirectionalLight(
    direction=[-1,-1,0] )
self.dirLight.setPosition( [2,4,0] )
self.scene.add( self.dirLight )
directHelper = DirectionalLightHelper(self.
    dirLight)
self.dirLight.add( directHelper )

sphereGeometry = SphereGeometry()
phongMaterial = PhongMaterial(
    texture=Texture("images/grid.png"),
    useShadow=True )

sphere1 = Mesh(sphereGeometry, phongMaterial)
sphere1.setPosition( [-2, 1, 0] )
self.scene.add( sphere1 )

sphere2 = Mesh(sphereGeometry, phongMaterial)
sphere2.setPosition( [ 1, 2.2, -0.5] )
self.scene.add( sphere2 )

self.renderer.enableShadows( self.dirLight )

# optional: render depth texture to mesh in
# scene
# depthTexture =
#     self.renderer.shadowObject.
#         renderTarget.texture
# shadowDisplay = Mesh( RectangleGeometry(),
#     TextureMaterial
#         (depthTexture) )
# shadowDisplay.setPosition([-1,3,0])

```

```

    # self.scene.add( shadowDisplay )

    floor = Mesh( RectangleGeometry(width=20,
        height=20), phongMaterial)
    floor.rotateX(-3.14/2)
    self.scene.add(floor)
def update(self):

    # view dynamic shadows -- need to increase
        shadow camera range
    # self.dirLight.rotateY(0.01337, False)

    self.rig.update( self.input, self.deltaTime )
    self.renderer.render( self.scene, self.camera )

    # render scene from shadow camera
    # shadowCam = self.renderer.shadowObject.
        camera
    # self.renderer.render( self.scene, shadowCam )

# instantiate this class and run the program
Test( screenSize=[800,600] ).run()

```

At this point, you can now easily include shadows in your scenes. If desired, you can also include them together with the other lighting-based effects implemented throughout this chapter, to create scenes showcasing a variety of techniques as illustrated in Figure 6.1.

6.10 SUMMARY AND NEXT STEPS

In this chapter, you learned about different types of lighting, light sources, illumination models, and shading models. After creating ambient, directional, and point light objects, you added normal vector data to geometric objects and used them in conjunction with light data to implement lighting with flat-shaded, Lambert, and Phong materials. You used normal vector data encoded in bump map textures to add the illusion of surface detail to geometric objects with light. Then, you extended the set of postprocessing effects, enabling the creation of light bloom and glow effects. Finally, you added shadow rendering capabilities to the framework, which built on all the concepts you have learned throughout this chapter.

While this may be the end of this book, hopefully it is just the beginning of your journey into computer graphics. As you have seen, Python and

OpenGL can be used together to create a framework that enables you to rapidly build applications featuring interactive, animated three-dimensional scenes with highly sophisticated graphics. The goal of this book has been to guide you through the creation of this framework, providing you with a complete understanding of the theoretical underpinnings and practical coding techniques involved, so that you can not only make use of this framework, but also further extend it to create any three-dimensional scene you can imagine. Good luck to you in your future endeavors!