CHAPTER 4

A Scene Graph Framework

In this chapter, you will begin to create the structure of a three-dimensional graphics framework in earnest. At the heart of the framework will be a *scene graph*: a data structure that organizes the contents of a 3D scene using a hierarchical or tree-like structure.

In the context of computer science, a *tree* is a collection of *node* objects, each of which stores a value and a list of zero or more nodes called *child* nodes. If a node *A* has a child node *B*, then node *A* is said to be the *parent* node of node *B*. In a tree, each node has exactly one parent, with the exception of a special node called the *root*, from which all other nodes can be reached by a sequence of child nodes. Starting with any node *N*, the set of nodes that can be reached from a sequence of child nodes are called the *descendants* of *N*, while the sequence of parent nodes from *N* up to and including the root node are called the *ancestors* of *N*. An abstract example of a tree is illustrated in Figure 4.1, where nodes are represented by ovals labeled with the letters from *A* through *G*, and arrows point from a node to its children. In the diagram, node *A* is the root and has child nodes *B*, *C*, and *D*; node *B* has child nodes *E* and *F*; node *D* has child node *G*. Nodes *E*, *F*, and *G* do not have any child nodes.

In a scene graph, each node represents a 3D object in the scene. As described previously, the current position, orientation, and scale of an object is stored in a matrix called the *model matrix*, which is calculated from the accumulated transformations that have been applied to the object.

DOI: 10.1201/9781003181378-4

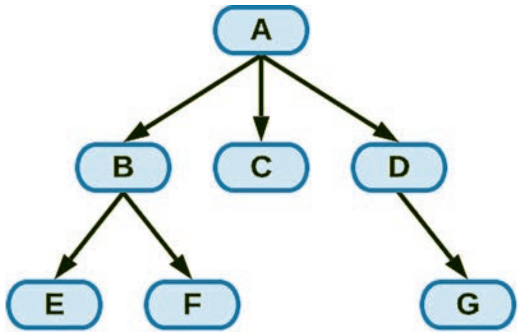

FIGURE 4.1 A tree with seven nodes.

For convenience, the position, orientation, and scale of an object will be collectively referred to as the *transform* of the object. The model matrix stores the transform of an object relative to its parent object in the scene graph. The transform of an object relative to the root of the scene graph, which is often called a *world transformation*, can be calculated from the product of the model matrix of the object and those of each of its ancestors. This structure enables complicated transformations to be expressed in terms of simpler ones. For example, the motion of the moon relative to the sun, illustrated in Figure 4.2 (gray dashed line), can be more simply expressed in terms of the combination of two circular motions: the moon relative to the Earth and the Earth relative to the sun (blue dotted line).

A scene graph structure also allows for simple geometric shapes to be grouped together into a compound object that can then be easily transformed as a single unit. For example, a simple model of a table may be created using a large, flat box shape for the top surface and four narrow, tall box shapes positioned underneath near the corners for the table legs, as illustrated by Figure 4.3. Let each of these objects be stored in a node, and all five nodes share the same parent node. Then, transforming the parent node affects the entire table object. (It is also worth noting that each of these boxes may reference the same vertex data; the different sizes and positions of each may be set with a model matrix.)

In the next section, you will learn about the overall structure of a scene graph-based framework, what the main classes will be, and how they encapsulate the necessary data and perform the required tasks to render a three-dimensional scene. Then, in the following sections, you will

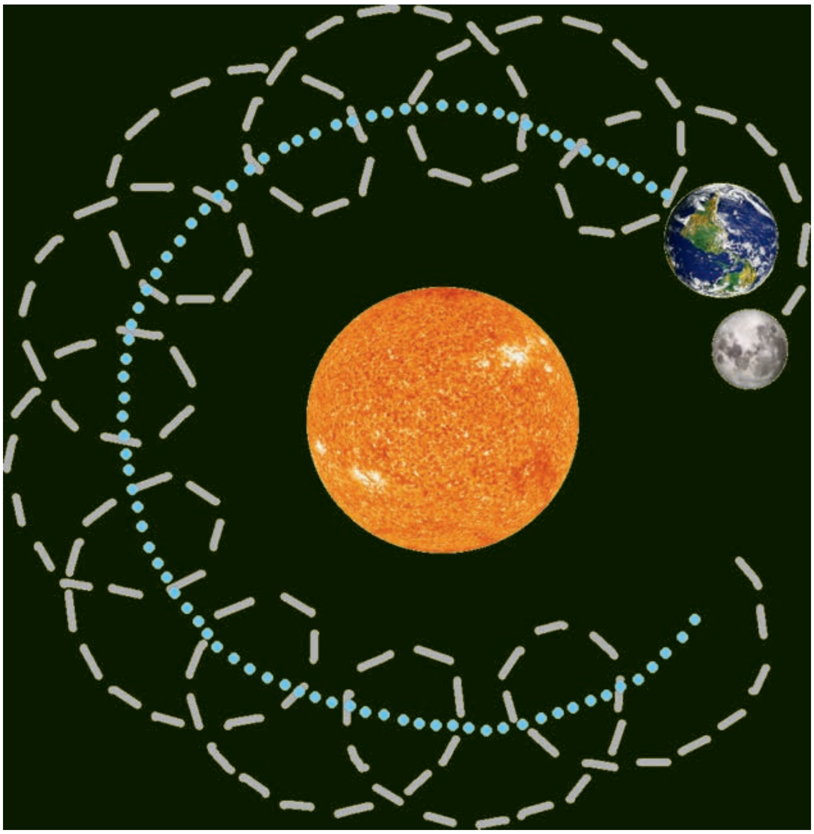

FIGURE 4.2 Motion of moon and Earth relative to sun.

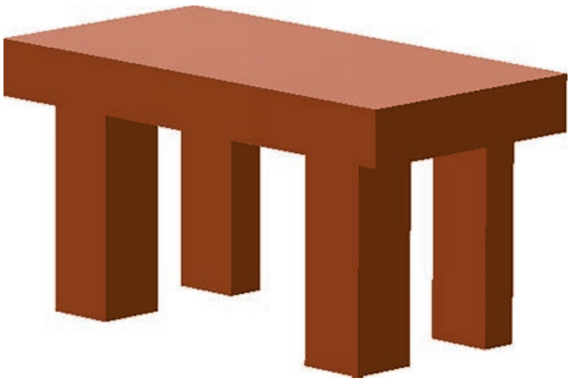

FIGURE 4.3 A table composed of five boxes.

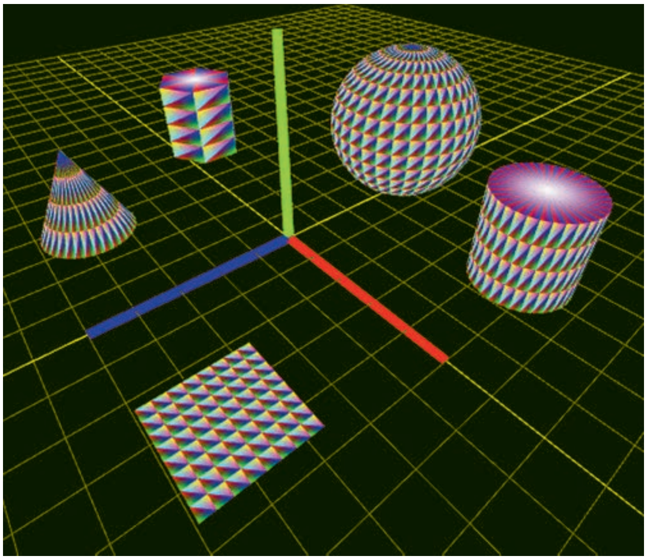
FIGURE 4.4 A scene containing multiple geometric shapes.

implement the classes for the framework, building on the knowledge and code from earlier chapters. The framework will enable you to rapidly create interactive scenes containing complex objects, such as the one illustrated in Figure 4.4.

4.1 OVERVIEW OF CLASS STRUCTURE

In a scene graph framework, the nodes represent objects located in a three-dimensional space. The corresponding class will be named **`Object3D`** and will contain three items:

1. a matrix to store its transform data
2. a list of references to child objects
3. a reference to a parent object

Many classes will extend the **`Object3D`** class, each with a different role in the framework. The root node will be represented by the **`Scene`** class. Interior nodes that are only used for grouping purposes will be represented by the **`Group`** class. Nodes corresponding to objects that can be rendered will be represented by the **`Mesh`** class. There are other objects with 3D characteristics that affect the appearance of the scene but are not themselves rendered. One such object is a virtual camera from whose point of view the scene will be rendered; this will be represented by the **`Camera`** class. Another such object is a virtual light source that affects shading and shadows; this will be represented by the **`Light`** class (but will not be created until Chapter 6).

To keep the framework code modular, each mesh will consist of a **`Geometry`** class object and a **`Material`** class object. The **`Geometry`** class will specify the general shape and other vertex-related properties, while the **`Material`** class will specify the general appearance of an object. Since each instance of a mesh stores a transformation matrix, multiple versions of a mesh (based on the same geometry and material data) can be rendered with different positions and orientations. Each mesh will also store a reference to a vertex array object, which associates vertex buffers (whose references will be stored by attribute objects stored in the geometry) to attribute variables (specified by shaders stored in the material). This will allow geometric objects to be reused and rendered with different materials in different meshes.

The **`Geometry`** class will mainly serve to store **`Attribute`** objects, which describe vertex properties, such as position and color, as seen in examples in prior chapters. In later chapters, geometric objects will also store texture coordinates, for applying images to shapes, and normal vectors, for use in lighting calculations. This class will calculate the total number of vertices, which is equal to the length of the data array stored in any attribute. Extensions of the **`Geometry`** class will be created to realize each particular shape. In some cases, such as rectangles and boxes, the data for each attribute will be listed directly. For other shapes, such as polygons, cylinders, and spheres, the attribute data will be calculated from mathematical formulas.

The **`Material`** class will serve as a repository for three types of information related to the rendering process and the appearance of an object: shader code (and the associated program reference), **`Uniform`** objects, and render settings: the properties which are set by calling OpenGL functions, such as the type of geometric primitive (points, lines, or triangles), point size, line width, and so forth. The base **`Material`** class will initialize dictionaries to store uniform objects and render setting data, and will define functions to perform tasks such as compiling the shader program code and locating uniform variable references. Extensions of this class will supply the actual shader code, a collection of uniform objects corresponding to uniform variables defined in the shaders, and a collection of render setting variables applicable to the type of geometric primitive being rendered.

A **Renderer** class will handle the general OpenGL initialization tasks as well as rendering the image. The rendering function will require a scene object and a camera object as parameters. For each mesh in the scene graph, the renderer will perform the tasks necessary before the **glDrawArrays** function is called, including activating the correct shader program, binding a vertex array object, configuring OpenGL render settings, and sending values to be used in uniform variables. Regarding uniform variables, there are three required by most shaders whose values are naturally stored outside the material: the transformation of a mesh, the transformation of the virtual camera used to view the scene, and the perspective transformation applied to all objects in the scene. While the uniform objects will be stored in the material for consistency, this matrix data will be copied into the corresponding uniform objects by the renderer before they send their values to the GPU.

Now that you have an idea of the initial classes that will be used by the framework, it is time to begin writing the code for each class.

4.2 3D OBJECTS

The **Object3D** class represents a node in the scene graph tree structure, and as such, it will store a list of references to child objects and a parent object, as well as **add** and **remove** functions to update parent and child references when needed. In addition, each object stores transform data using a numpy matrix object and will have a function called **getWorldMatrix** to calculate the world transformation. When rendering the scene, the nodes in the tree will be collected into a list to simplify iterating over the set of nodes; this will be accomplished with a function called **getDescendantList**. To implement this, in the **core** folder, create a new file named **object3D.py** containing the following code:

```
from core.matrix import Matrix

class Object3D(object):

    def __init__(self):
        self.transform = Matrix.makeIdentity()
        self.parent = None
        self.children = []
    def add(self, child):
        self.children.append(child)
        child.parent = self

    def remove(self, child):
        self.children.remove(child)
        child.parent = None

    # calculate transformation of this Object3D relative
    #   to the root Object3D of the scene graph
    def getWorldMatrix(self):
        if self.parent == None:
            return self.transform
        else:
            return self.parent.getWorldMatrix() @
              self.transform

    # return a single list containing all descendants
    def getDescendantList(self):
        # master list of all descendant nodes
        descendants = []
        # nodes to be added to descendant list,
        #   and whose children will be added to this list
        nodesToProcess = [self]
        # continue processing nodes while any are left
        while len( nodesToProcess ) > 0:
            # remove first node from list
            node = nodesToProcess.pop(0)
            # add this node to descendant list
            descendants.append(node)
            # children of this node must also be
              processed
            nodesToProcess = node.children +
              nodesToProcess
        return descendants
```

It will also be convenient for this class to contain a set of functions that translate, rotate, and scale the object by creating and applying the corresponding matrices from the **`Matrix`** class to the model matrix. Recall that each of these transformations can be applied as either a local transformation or a global transformation, depending on the order in which the model matrix and the new transformation matrix are multiplied. (In this context, a global transformation means a transformation performed with respect to the coordinate axes of the parent object in the scene graph.) This distinction – whether a matrix should be applied as a local transformation – will be specified with an additional parameter. To incorporate this functionality, add the following code to the **`Object3D`** class:

```
# apply geometric transformations
def applyMatrix(self, matrix, localCoord=True):
    if localCoord:
        self.transform = self.transform @ matrix
    else:
        self.transform = matrix @ self.transform

def translate(self, x,y,z, localCoord=True):
    m = Matrix.makeTranslation(x,y,z)
    self.applyMatrix(m, localCoord)

def rotateX(self, angle, localCoord=True):
    m = Matrix.makeRotationX(angle)
    self.applyMatrix(m, localCoord)

def rotateY(self, angle, localCoord=True):
    m = Matrix.makeRotationY(angle)
    self.applyMatrix(m, localCoord)

def rotateZ(self, angle, localCoord=True):
    m = Matrix.makeRotationZ(angle)
    self.applyMatrix(m, localCoord)

def scale(self, s, localCoord=True):
    m = Matrix.makeScale(s)
    self.applyMatrix(m, localCoord)
```

Finally, the position of an object can be determined from entries in the last column of the transform matrix, as discussed in the previous chapter. Making use of this fact, functions to get and set the position of an object are implemented with the following code, which you should add to the **`Object3D`** class. Two functions are included to get the position of an object: one which returns its local position (with respect to its parent), and one which returns its global or world position, extracted from the world transform matrix previously discussed.

```
# get/set position components of transform
def getPosition(self):
    return [ self.transform.item((0,3)),
             self.transform.item((1,3)),
             self.transform.item((2,3)) ]

def getWorldPosition(self):
        worldTransform = self.getWorldMatrix()
        return [ worldTransform.item((0,3)),
                 worldTransform.item((1,3)),
                 worldTransform.item((2,3)) ]

def setPosition(self, position):
    self.transform.itemset((0,3), position[0])
    self.transform.itemset((1,3), position[1])
    self.transform.itemset((2,3), position[2])
```

The next few classes correspond to particular types of elements in the scene graph, and therefore, each will extend the **Object3D** class.

4.2.1 Scene and Group

The **Scene** and **Group** classes will both be used to represent nodes in the scene graph that do not correspond to visible objects in the scene. The **Scene** class represents the root node of the tree, while the **Group** class represents an interior node to which other nodes are attached to more easily transform them as a single unit. These classes do not add any functionality to the **Object3D** class; their primary purpose is to make the application code easier to understand.

In the **core** folder, create a new file named **scene.py** with the following code:

```
from core.object3D import Object3D


class Scene(Object3D):

    def __init__(self):
        super().__init__()
```

Then, create a new file named **`group.py`** with the following code.

```
from core.object3D import Object3D

class Group(Object3D):

    def __init__(self):
        super().__init__()
```

4.2.2 Camera

The **`Camera`** class represents the virtual camera used to view the scene. As with any 3D object, it has a position and orientation, and this information is stored in its transform matrix. The camera itself is not rendered, but its transform affects the apparent placement of the objects in the rendered image of the scene. Understanding this relationship is necessary to creating and using a **`Camera`** object. Fortunately, the key concept can be illustrated by a couple of examples.

Consider a scene containing multiple objects in front of the camera, and imagine that the camera shifts two units to the left. From the perspective of the viewer, all the objects in the scene would appear to have shifted two units to the right. In fact, these two transformations (shifting the camera left versus shifting all world objects right) are *equivalent*, in the sense that there is no way for the viewer to distinguish between them in the rendered image. As another example, imagine that the camera rotates 45° clockwise about its vertical axis. To the viewer, this appears equivalent to all objects in the world having rotated 45° counterclockwise around the camera. These examples illustrate the general notion that each transformation of the camera affects the scene objects in the opposite way. Mathematically, this relationship is captured by defining the *view matrix*, which describes the placement of objects in the scene with respect to the camera, as the inverse of the camera's transform matrix.

As cameras are used to define the position and orientation of the viewer, this class is also a natural place to store data describing the visible region of the scene, which is encapsulated by the projection matrix. Therefore, the **`Camera`** class will store both a view matrix and a projection matrix. The view matrix will be updated as needed, typically once during each iteration of the application main loop, before the meshes are drawn. To implement this class, in your **`core`** folder, create a new file named **`camera.py`** with the following code:

```
from core.object3D import Object3D
from core.matrix import Matrix
from numpy.linalg import inv

class Camera(Object3D):

    def __init__(self, angleOfView=60,
            aspectRatio=1, near=0.1, far=1000):
        super().__init__()
        self.projectionMatrix = Matrix.makePerspective
          (angleOfView, aspectRatio, near, far)
        self.viewMatrix = Matrix.makeIdentity()

    def updateViewMatrix(self):
        self.viewMatrix = inv( self.getWorldMatrix() )
```

4.2.3 Mesh

The **Mesh** class will represent the visible objects in the scene. It will contain geometric data that specifies vertex-related properties and material data that specifies the general appearance of the object. Since a vertex array object links data between these two components, the **Mesh** class is also a natural place to create and store this reference, and set up the associations between vertex buffers and shader variables. For convenience, this class will also store a boolean variable used to indicate whether or not the mesh should appear in the scene. To proceed, in your **core** folder, create a new file named **mesh.py** with the following code:

```
from core.object3D import Object3D
from OpenGL.GL import *


class Mesh(Object3D):


    def __init__(self, geometry, material):
        super().__init__()

        self.geometry = geometry
        self.material = material

        # should this object be rendered?
        self.visible = True
        # set up associations between
        #   attributes stored in geometry and
        #   shader program stored in material
        self.vaoRef = glGenVertexArrays(1)
        glBindVertexArray(self.vaoRef)
        for variableName, attributeObject
            in geometry.attributes.items():
             attributeObject.associateVariable(
              material.programRef, variableName)
        # unbind this vertex array object
        glBindVertexArray(0)
```

Now that the **Object3D** and the associated **Mesh** class have been created, the next step is to focus on the two main components of a mesh: the **Geometry** class and the **Material** class, and their various extensions.

4.3 GEOMETRY OBJECTS

Geometry objects will store attribute data and the total number of vertices. The base **Geometry** class will define a dictionary to store attributes, a function named **addAttribute** to simplify adding attributes, a variable to store the number of vertices, and a function named **countVertices** that can calculate this value (which is the length of any attribute object's data array). Classes that extend the base class will add attribute data and call the **countVertices** function after attributes have been added.

Since there will be many geometry-related classes, they will be organized into a separate folder. For this purpose, in your main folder, create a new folder called **geometry**. To create the base class, in the **geometry** folder, create a new file called **geometry.py** with the following code:

```
from core.attribute import Attribute

class Geometry(object):

    def __init__(self):

        # Store Attribute objects,
        #   indexed by name of associated variable in
              shader.
        # Shader variable associations set up later
        #   and stored in vertex array object in Mesh.
        self.attributes = {}

        # number of vertices
        self.vertexCount = None

    def addAttribute(self, dataType, variableName, data):
        self.attributes[variableName] = Attribute
            (dataType, data)

    def countVertices(self):
        # number of vertices may be calculated from
        #   the length of any Attribute object's array
               of data
        attrib = list( self.attributes.values() )[0]
        self.vertexCount = len( attrib.data )
```

The next step is to create a selection of classes that extend the **`Geometry`** class that contain the data for commonly used shapes. Many applications can make use of these basic shapes or combine basic shapes into compound shapes by virtue of the underlying structure of the scene graph.

In this chapter, these geometric objects will contain two attributes: vertex positions (which are needed for every vertex shader) and a default set of vertex colors. Until intermediate topics such as applying images to surfaces or lighting and shading are introduced in later chapters (along with their corresponding vertex attributes, texture coordinates, and normal vectors), vertex colors will be necessary to distinguish the faces of a three-dimensional object. For example, Figure 4.5 illustrates a cube with and without vertex colors applied; without these distinguishing features, a cube is indistinguishable from a hexagon. If desired, a developer can always change the default set of vertex colors in a geometric object by overwriting the array data in the corresponding attribute and calling its **`storeData`** function to resend the data to its buffer.

4.3.1 Rectangles

After a triangle, a rectangle is the simplest shape to render, as it is composed of four vertices grouped into two triangles. To provide flexibility when using this class, the constructor will take two parameters, the width and height of the rectangle, each with a default value of 1. Assuming that the

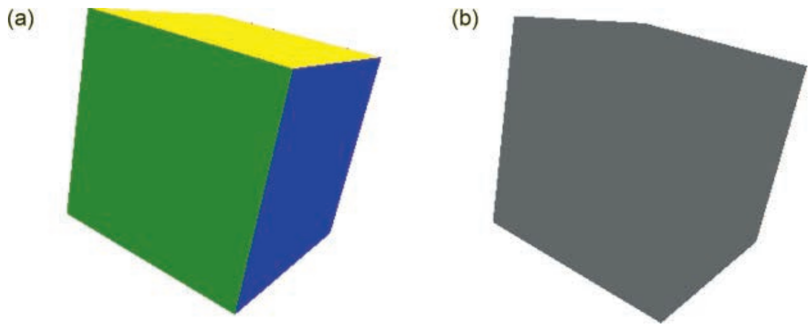

FIGURE 4.5 A cube rendered with (a) and without (b) vertex colors.

rectangle is centered at the origin, this means that the vertex x and y coordinates will be $\pm width / 2$ and $\pm height / 2$, as illustrated in Figure 4.6. (The z coordinates will be set to 0°.) Also, with the points denoted by **`P0`**, **`P1`**, **`P2`**, **`P3`** as shown in the diagram, they will be grouped into the triangles ($P0$, $P1$, $P3$) and ($P0$, $P3$, $P2$). Note that the vertices in each triangle are consistently listed in counterclockwise order, as OpenGL uses counterclockwise ordering by default to distinguish between the front side and back side of a triangle; back sides of shapes are frequently not rendered in order to improve rendering speed.

To implement this geometric shape, in the **`geometry`** folder, create a new file called **`rectangleGeometry.py`** containing the following code:

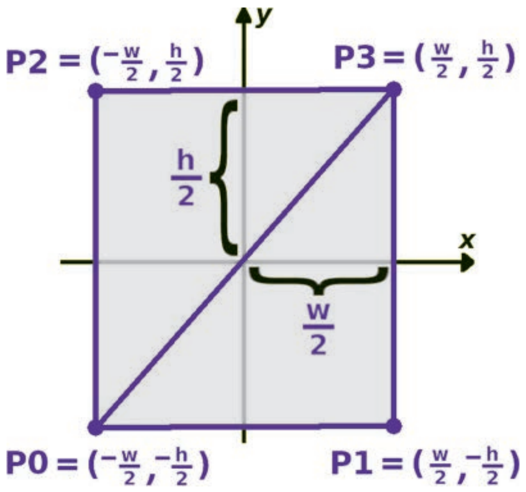
y
x
P2 = $(-\frac{w}{2}, \frac{h}{2})$
P3 = $(\frac{w}{2}, \frac{h}{2})$
$\frac{h}{2}$
$\frac{w}{2}$
P0 = $(-\frac{w}{2}, -\frac{h}{2})$
P1 = $(\frac{w}{2}, -\frac{h}{2})$

FIGURE 4.6 Vertex coordinates for a rectangle with width w and height h.

```
from geometry.geometry import Geometry

class RectangleGeometry(Geometry):

    def __init__(self, width=1, height=1):
        super().__init__()

        P0 = [-width/2, -height/2, 0]
        P1 = [ width/2, -height/2, 0]
        P2 = [-width/2,  height/2, 0]
        P3 = [ width/2,  height/2, 0]
        C0, C1, C2, C3 = [1,1,1], [1,0,0], [0,1,0],
           [0,0,1]
        positionData = [ P0,P1,P3, P0,P3,P2 ]
        colorData    = [ C0,C1,C3, C0,C3,C2 ]

        self.addAttribute("vec3", "vertexPosition",
          positionData)
        self.addAttribute("vec3", "vertexColor",
          colorData)
        self.countVertices()
```

Note that the colors corresponding to the vertices, denoted by `C0`, `C1`, `C2`, `C3`, are listed in precisely the same order as the positions; this will create a consistent gradient effect across the rectangle. Alternatively, to render each triangle with a single solid color, the color data array could have been entered as `[C0,C0,C0, C1,C1,C1]`, for example. Although you are not able to create an application to render this data yet, when it can eventually be rendered, it will appear as shown on the left side of Figure 4.7; the right side illustrates the alternative color data arrangement described in this paragraph.

In the next few subsections, classes for geometric shapes of increasing complexity will be developed. At this point, you may choose to skip ahead to Section 4.4, or you may continue creating as many of the geometric classes below as you wish before proceeding.

4.3.2 Boxes

A box is a particularly simple three-dimensional shape to render. Although some other three-dimensional shapes (such as some pyramids) may have fewer vertices, the familiarity of the shape and the symmetries in the positions of its vertices make it a natural choice for a first three-dimensional

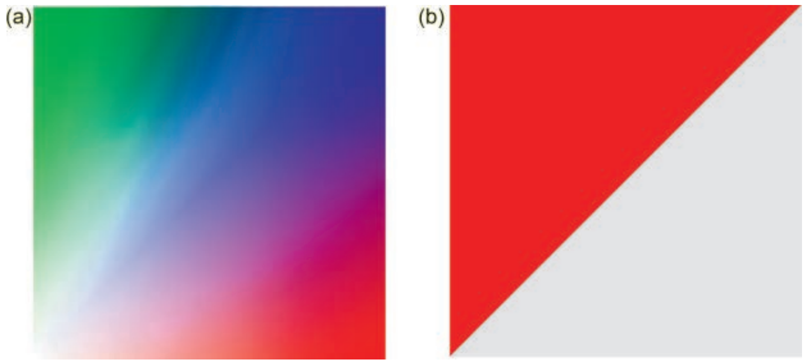

FIGURE 4.7 Rendering RectangleGeometry with gradient coloring (a) and solid coloring (b).

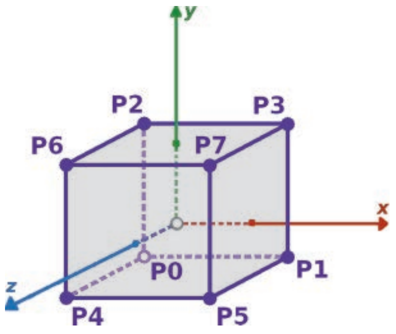

FIGURE 4.8 Vertices of a cube.

shape to implement. A box has 8 vertices and 6 sides composed of 2 triangles each, for a total of 12 triangles. Since each triangle is specified with three vertices, the data arrays for each attribute will contain 36 elements. Similar to the **`Rectangle`** class just created, the constructor of the **`Box`** class will take three parameters: the width, height, and depth of the box, referring to lengths of the box edges parallel to the *x*-, *y*-, and *z*-axes, respectively. As before, the parameters will each have a default value of 1, and the box will be centered at the origin. The points will be denoted P0 through P7, as illustrated in Figure 4.8, where the dashed lines indicate parts of the lines which are obscured from view by the box. To more easily visualize the arrangement of the triangles in this shape, Figure 4.9 depicts an “unfolded” box lying in a flat plane, sometimes called a *net diagram*. For each face of the box, the vertices of the corresponding triangles will be ordered in the same sequence as they were in the **`Rectangle`** class.

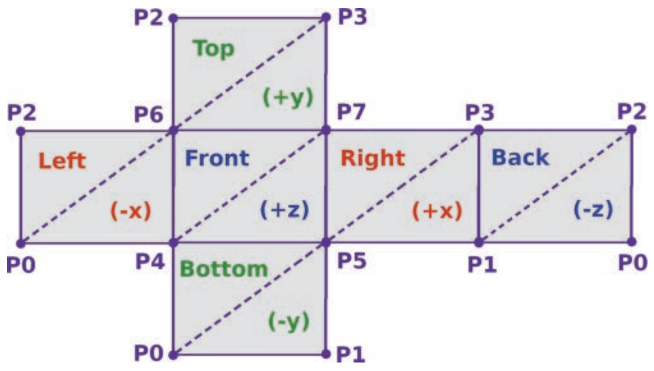

FIGURE 4.9 Vertex arrangement of an unfolded cube.

To aid with visualization, the vertices will be assigned colors (denoted C1–C6) depending on the corresponding face. The faces perpendicular to the *x*-axis, *y*-axis, and *z*-axis will be tinted shades of red, green, and blue, respectively. Note that each of the vertices is present on three different faces: for instance, the vertex with position P7 is part of the right (*x*+) face, the top (*y*+) face, and the front (*z*+) face, and thus, each point will be associated with multiple colors, in contrast to the **Rectangle** class. To create this class, in the **geometry** folder, create a new file called **box-Geometry.py** containing the following code:

```
from geometry.geometry import Geometry

class BoxGeometry(Geometry):

    def __init__(self, width=1, height=1, depth=1):
        super().__init__()

        P0 = [-width/2, -height/2, -depth/2]
        P1 = [ width/2, -height/2, -depth/2]
        P2 = [-width/2,  height/2, -depth/2]
        P3 = [ width/2,  height/2, -depth/2]
        P4 = [-width/2, -height/2,  depth/2]
        P5 = [ width/2, -height/2,  depth/2]
        P6 = [-width/2,  height/2,  depth/2]
        P7 = [ width/2,  height/2,  depth/2]
        # colors for faces in order: x+, x-, y+, y-,
            z+, z-
        C1, C2 = [1, 0.5, 0.5], [0.5, 0, 0]
```

```
C3, C4 = [0.5, 1, 0.5], [0, 0.5, 0]
C5, C6 = [0.5, 0.5, 1], [0, 0, 0.5]

positionData = [ P5,P1,P3,P5,P3,P7, P0,P4,P6,P0,
                 P6,P2,P6,P7,P3,P6,P3,P2,
                 P0,P1,P5,P0,P5,P4,P4,P5,P7,
                 P4,P7,P6, P1,P0,P2,P1,P2,P3 ]

colorData = [C1]*6 + [C2]*6 + [C3]*6 +
            [C4]*6 + [C5]*6 + [C6]*6

self.addAttribute("vec3", "vertexPosition",
  positionData)
self.addAttribute("vec3", "vertexColor",
  colorData)
self.countVertices()
```

Note the use of the list operators * to duplicate an array a given number of times and + to concatenate lists. Figure 4.10 illustrates how this box will appear from multiple perspectives once you are able to render it later in this chapter.

4.3.3 Polygons

Polygons (technically, *regular polygons*) are two-dimensional shapes such that all sides have the same length and all angles have equal measure, such as equilateral triangles, squares, pentagons, hexagons, and so forth. The corresponding class will be designed so that it may produce a polygon with any number of sides (three or greater). The coordinates of the vertices can be calculated by using equally spaced points on the circumference of a circle. A circle with radius R can be expressed with the

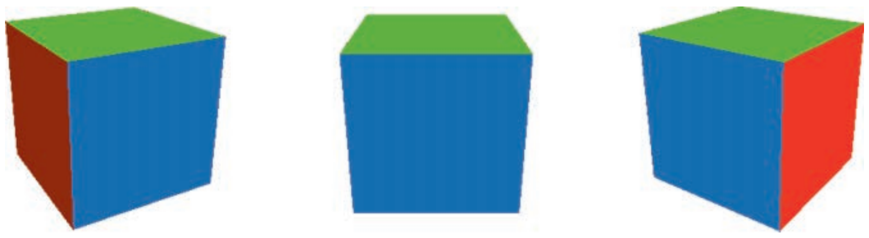
FIGURE 4.10 Rendering BoxGeometry from multiple perspectives.

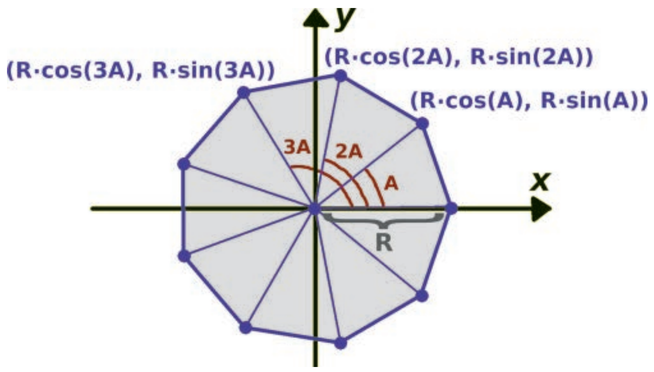

FIGURE 4.11 Calculating the vertices of a regular polygon.

parametric equations $x = R \cdot \cos(t)$ and $y = R \cdot \sin(t)$. Note that these parametric equations also satisfy the implicit equation of a circle of radius R, $x^2 + y^2 = R^2$, which can be verified with the use of the trigonometric identity $\sin^2(t) + \cos^2(t) = 1$. The key is to find the required values of the angle t that correspond to these equally spaced points. This in turn is calculated using multiples of a base angle A, equal to 2π divided by the number of sides of the polygon being generated, as illustrated with a nonagon in Figure 4.11.

Once it is understood how the vertices of a polygon can be calculated, one must also consider how the vertices will be grouped into triangles. In this case, each triangle will have one vertex at the origin (the center of the polygon) and two adjacent vertices on the circumference of the polygon, ordered counterclockwise, as usual. In addition, the same three vertex colors will be repeated in each triangle for simplicity. To proceed, in the **geometry** folder, create a new file called **polygonGeometry.py** with the following code:

```
from geometry.geometry import Geometry
from math import sin, cos, pi

class PolygonGeometry(Geometry):

    def __init__(self, sides=3, radius=1):
        super().__init__()

                A = 2 * pi / sides
        positionData = []
        colorData    = []

        for n in range(sides):
            positionData.append( [0, 0, 0] )
            positionData.append(
               [radius*cos(n*A),  radius*sin(n*A),  0] )
            positionData.append(
              [radius*cos((n+1)*A), radius*sin((n+1)*A),
                 0] )
            colorData.append( [1, 1, 1] )
            colorData.append( [1, 0, 0] )
            colorData.append( [0, 0, 1] )

        self.addAttribute("vec3", "vertexPosition",
          positionData)
        self.addAttribute("vec3", "vertexColor",
          colorData)
        self.countVertices()
```

Figure 4.12 illustrates a few different polygons that you will eventually be able to render with this class, with 3, 8, and 32 sides. Note that with sufficiently many sides, the polygon closely approximates a circle. In fact, due to the discrete nature of computer graphics, it is not possible to render a perfect circle, and so this is how circular shapes are implemented in practice.

For convenience, you may decide to extend the **Polygon** class to generate particular polygons with preset numbers of sides (or even a circle, as previously discussed), while still allowing the developer to specify a value for the radius, which will be passed along to the base class. For example, you could optionally create a **Hexagon** class with a file in the **geometry** folder named **hexagon.py** containing the following code:

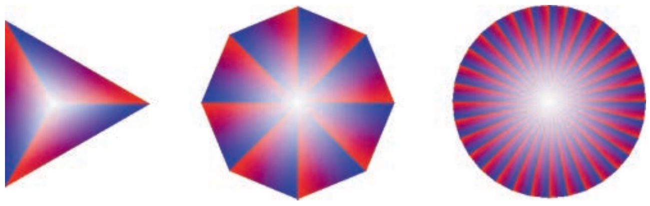

FIGURE 4.12 Polygons with 3 sides, 8 sides, and 32 sides.

```
from geometry.polygonGeometry import PolygonGeometry
class HexagonGeometry(PolygonGeometry):
    def __init__(self, radius=1):
        super().__init__( sides=6, radius=radius )
```

4.3.4 Parametric Surfaces and Planes

Similar to the two-dimensional polygons just presented, there are a variety of surfaces in three dimensions that can be expressed with mathematical functions. The simplest type of surface arises from a function of the form $z = f(x, y)$, but this is too restrictive to express many common surfaces, such as cylinders and spheres. Instead, each of the coordinates *x*, *y*, and *z* will be expressed by a function of two independent variables *u* and *v*. Symbolically,

$$x = f(u,v), \quad y = g(u,v), \quad z = h(u,v)$$

or, written in a different format,

$$(x,y,z) = (\ f(u,v), g(u,v), h(u,v)\ ) = S(u,v)$$

Generally, the variables *u* and *v* are limited to a rectangular domain such as $0 \le u \le 1$ and $0 \le v \le 1$, and thus, the function *S* can be thought of as transforming a two-dimensional square or rectangular region, embedding it in three-dimensional space. The function *S* is called a *parametric function*. Graphing the set of output values (*x*, *y*, *z*) yields a surface that is said to be *parameterized* by the function *S*. Figure 4.13 depicts a rectangular region (subdivided into triangles) and the result of transforming it into the surface of a sphere or a cylinder.

To incorporate this into the graphics framework you are creating, the first step is to create a class that takes as inputs a parametric function $S(u, v)$ that defines a surface, bounds for *u* and *v*, and the resolution—in this context, the number of sample values to be used between the *u*

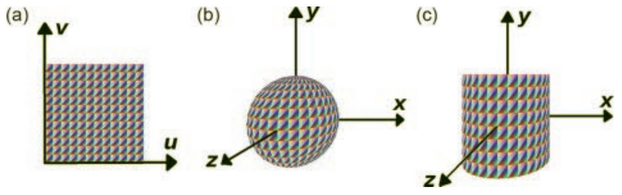

FIGURE 4.13 A rectangular region (a), transformed into a sphere (b) and a cylinder (c).

and *v* bounds. With this data, the space between *u* and *v* coordinates (traditionally called **deltaU** and **deltaV**) can be calculated, and a set of points on the surface can be calculated and stored in a two-dimensional array (called **positions**) for convenience. Finally, the vertex positions (and related vertex data, such as colors) must be grouped into triangles and stored in a dictionary of **Attribute** objects for use in the **Geometry** class. To accomplish this task, in your **geometry** folder, create a new file named **parametricGeometry.py** containing the following code:

```
from geometry.geometry import Geometry

class ParametricGeometry(Geometry):
    def __init__(self, uStart, uEnd, uResolution,
    vStart, vEnd, vResolution, surfaceFunction):

        # generate set of points on function
        deltaU = (uEnd - uStart) / uResolution
        deltaV = (vEnd - vStart) / vResolution
        positions = []

        for uIndex in range(uResolution+1):
            vArray = []
            for vIndex in range(vResolution+1):
                u = uStart + uIndex * deltaU
                v = vStart + vIndex * deltaV
                vArray.append( surfaceFunction(u,v) )
            positions.append(vArray)

        # store vertex data
        positionData = []
        colorData    = []

        # default vertex colors
        C1, C2, C3 = [1,0,0], [0,1,0], [0,0,1]
        C4, C5, C6 = [0,1,1], [1,0,1], [1,1,0]

        # group vertex data into triangles
        # note: .copy() is necessary to avoid storing
            references
        for xIndex in range(uResolution):
            for yIndex in range(vResolution):
                # position data
                pA = positions[xIndex+0][yIndex+0]
                pB = positions[xIndex+1][yIndex+0]
                pD = positions[xIndex+0][yIndex+1]
                pC = positions[xIndex+1][yIndex+1]
                positionData += [ pA.copy(), pB.copy(),
                  pC.copy(), pA.copy(), pC.copy(),
                               pD.copy() ]

                # color data
                colorData += [C1,C2,C3, C4,C5,C6]

        self.addAttribute("vec3", "vertexPosition",
          positionData)
        self.addAttribute("vec3", "vertexColor",
          colorData)
        self.countVertices()
```

The **ParametricGeometry** class should be thought of as an abstract class: it will not be instantiated directly; instead, it will be extended by other classes that supply specific functions and variable bounds that yield different surfaces. The simplest case is a *plane*, a flat surface that can be thought of as a subdivided rectangle, similar to the **Rectangle** class previously developed. The equation for a plane (extending along the x and y directions, and where z is always 0) is

$$S(u, v) = (u, v, 0)$$

As was the case with the **Rectangle** class, the plane will be centered at the origin, and parameters will be included in the constructor to specify the width and height of the plane. Additional parameters will be included to allow the user to specify the resolution for the u and v variables, but given the more relevant variable names **widthResolution** and **heightResolution**. To create this class, create a new file named **planeGeometry.py** in the **geometry** folder, containing the following code:

```
from geometry.parametricGeometry import
  ParametricGeometry

class PlaneGeometry(ParametricGeometry):
    def __init__(self, width=1, height=1,
      widthSegments=8, heightSegments=8):

        def S(u,v):
            return [u, v, 0]

        super().__init__( -width/2,  width/2,
          widthSegments, -height/2, height/2,
                                   heightSegments, S )
```

A plane geometry with the default parameter values above will appear as shown in Figure 4.14.

4.3.5 Spheres and Related Surfaces

Along with boxes, spheres are one of the most familiar three-dimensional shapes, illustrated on the left side of Figure 4.15. In order to render a sphere in this framework, you will need to know the parametric equations of a sphere. For simplicity, assume that the sphere is centered at the origin and has radius 1. The starting point for deriving this formula is the parametric equation of a circle of radius R, since the cross-sections of a sphere are circles. Assuming that cross-sections will be analyzed along the y-axis, let $z = R \cdot \cos(u)$ and $x = R \cdot \sin(u)$, where $0 \le u \le 2\pi$. The radius R of the cross-section will depend on the value of y. For example, in the central cross-section, when $y = 0$, the radius is $R = 1$. At the top and bottom of the sphere (where $y = 1$ and $y = -1$), the cross-sections are single points, which can be considered as $R = 0$. Since the equations for x, y, and z must also satisfy the implicit equation of a unit sphere, $x^2 + y^2 + z^2 = 1$, you can substitute the formulas for x and z into this equation and simplify to get the equation $R^2 + y^2 = 1$. Rather than solve for R as a function of y, it is more productive to once again use the parametric equations for a circle, letting $R = \cos(v)$ and $y = \sin(v)$. For R and y to have the values previously

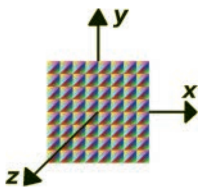

FIGURE 4.14 Plane geometry.

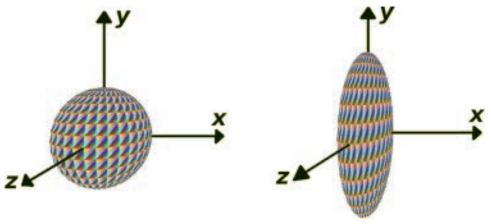

FIGURE 4.15 Sphere and ellipsoid.

described, the values of v must range from $-\pi/2$ to $\pi/2$. This yields the full parameterization of the unit sphere:

$$(x, y, z) = (\sin(u)\cdot\cos(v), \sin(v), \cos(u)\cdot\cos(v))$$

For additional flexibility, you may scale the parametric equations for x, y, and z by different amounts, resulting in a shape called an *ellipsoid*, illustrated on the right side of Figure 4.15. Then, a sphere can be considered as a special case of an ellipsoid, where the scaling amounts are equal along each direction.

To implement these shapes, you will start with an ellipsoid. The size parameters will be called *width*, *height*, and *depth*, and used in the same way as the corresponding parameters that define the size of a box. In the **geometry** folder, create a new file named **ellipsoidGeometry.py**, containing the following code:

```
from geometry.parametricGeometry import
  ParametricGeometry
from math import sin, cos, pi


class EllipsoidGeometry(ParametricGeometry):

    def __init__(self, width=1, height=1, depth=1,
                 radiusSegments=32, heightSegments=16):
        def S(u,v):
            return [  width/2 * sin(u) * cos(v),
                     height/2 * sin(v),
                      depth/2 * cos(u) * cos(v) ]

        super().__init__( 0, 2*pi, radiusSegments,
                          -pi/2, pi/2, heightSegments, S )
```

Next, you will extend this class to create a sphere. In the **geometry** folder, create a new file named **sphereGeometry.py**, containing the following code:

```
from geometry.ellipsoidGeometry import
  EllipsoidGeometry
from math import sin, cos, pi

class SphereGeometry(EllipsoidGeometry):

    def __init__(self, radius=1,
                radiusSegments=32, heightSegments=16):

        super().__init__( 2*radius, 2*radius, 2*radius,
                          radiusSegments,
                            heightSegments )
```

4.3.6 Cylinders and Related Surfaces

As was the case for spheres, the starting point for deriving the equation of a cylinder (illustrated in Figure 4.16) is the parametric equation of a circle, since the cross-sections of a cylinder are also circles. For the central axis of the cylinder to be aligned with the y-axis, as illustrated in Figure 4.16, let $z = R \cdot \cos(u)$ and $x = R \cdot \sin(u)$, where $0 \le u \le 2\pi$. Furthermore, for the cylinder to have height h and be centered at the origin, you will use the parameterization:

$$y = h \cdot (v - 1/2), \text{ where } 0 \le v \le 1.$$

This parameterization yields an "open-ended" cylinder or tube; the parameterization does not include top or bottom sides. The data for these sides can be added from polygon geometries, modified so that the circles

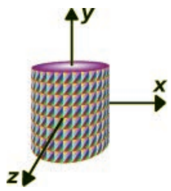

FIGURE 4.16 Cylinder.

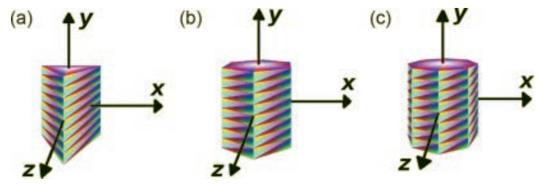

FIGURE 4.17 Triangular, hexagonal, and octagonal prisms.

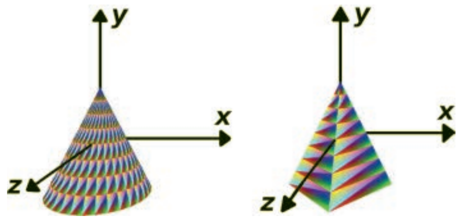

FIGURE 4.18 Cone and pyramid.

are perpendicular to the *y*-axis and centered at the top and bottom of the cylinder. This requires additional code which will be described later.

To approximate a circular cross-section, a large number of radial segments will typically be used. Choosing a significantly smaller number of radial segments will result in a solid whose cross-sections are clearly polygons: these three-dimensional shapes are called *prisms*, three of which are illustrated in Figure 4.17. Note that a square prism has the shape of a box, although due to the way the class is structured, it will contain more triangles and is aligned differently: in the **`BoxGeometry`** class, the coordinates were chosen so that the sides were perpendicular to the coordinate axes; a square prism will appear to have been rotated by 45° (around the *y*-axis) from this orientation.

By generalizing the cylinder equations a bit more, you gain the ability to produce more three-dimensional shapes, as illustrated in Figure 4.18. For example, cones are similar to cylinders in that their cross-sections are circles, with the difference that the radius of each circle becomes smaller the closer the cross-section is to the top of the cylinder; the top is a single point, a circle with radius zero. Furthermore, by replacing the circular cross-sections of a cone with polygon cross-sections, the result is a pyramid. Square pyramids may come to mind most readily, but one may consider triangular pyramids, pentagonal pyramids, hexagonal pyramids, and so on. To provide maximum generality, the base class for all of these shapes will include parameters where the radius at the top and the radius at the bottom can be specified, and the radius of each cross-section will be linearly interpolated from these two values. In theory, this would even enable frustum (truncated pyramid) shapes to be created.

To efficiently code this set of shapes, the most general class will be named **CylindricalGeometry**, and the classes that extend it will be named **CylinderGeometry**, **PrismGeometry**, **ConeGeometry**, and **PyramidGeometry**. (To create a less common shape such as a frustum, you can use the **CylindricalGeometry** class directly.) To begin, in the **geometry** folder, create a new file named **cylindricalGeometry.py**, containing the following code:

```
from geometry.parametricGeometry import
  ParametricGeometry
from math import sin, cos, pi


class CylindricalGeometry(ParametricGeometry):

    def __init__(self, radiusTop=1, radiusBottom=1,
        height=1,
                 radialSegments=32, heightSegments=4,
                  closedTop=True, closedBottom=True):

        def S(u,v):
            return [ (v*radiusTop + (1-v)*radiusBottom)
               * sin(u), height * (v - 0.5),
                      (v*radiusTop + (1-v)*radiusBottom)
                        * cos(u) ]

        super().__init__( 0, 2*pi, radialSegments,
                          0, 1, heightSegments, S )
```

The most natural way to create a top and bottom for the cylinder is to use the data generated by the **PolygonGeometry** class. For the polygons to be correctly aligned with the top and bottom of the cylinder, there needs to be a way to transform the vertex position data of a polygon. Furthermore, once the data has been transformed, all the attribute data from the polygon objects will need to be merged into the attribute data for the cylindrical object. Since these operations may be useful in multiple situations, functions to perform these tasks will be implemented in the

Geometry class. In the file **geometry.py** in the **geometry** folder, add the following two functions:

```
# transform the data in an attribute using a matrix
    def applyMatrix(self, matrix,
    variableName="vertexPosition"):


    oldPositionData = self.attributes[variableName].data
    newPositionData = []

    for oldPos in oldPositionData:
        # avoid changing list references
        newPos = oldPos.copy()
        # add homogeneous fourth coordinate
        newPos.append(1)
        # multiply by matrix
        newPos = matrix @ newPos
        # remove homogeneous coordinate
        newPos = list( newPos[0:3] )
        # add to new data list
        newPositionData.append( newPos )

    self.attributes[variableName].data =
      newPositionData
    # new data must be uploaded
    self.attributes[variableName].uploadData()

# merge data from attributes of other geometry into
    this object;
#   requires both geometries to have attributes with
      same names
def merge(self, otherGeometry):

    for variableName, attributeObject in self.
       attributes.items():
        attributeObject.data +=
             otherGeometry.attributes[variableName].data
        # new data must be uploaded
        attributeObject.uploadData()

    # update the number of vertices
    self.countVertices()
```

With these additions to the **Geometry** class, you can now use these functions as described above. In the file **cylindricalGeometry.py**, add the following code to the initialization function, after which the **CylindricalGeometry** class will be complete.

```
if closedTop:
    topGeometry = PolygonGeometry(radialSegments,
      radiusTop)
    transform = Matrix.makeTranslation(0, height/2, 0) @
          Matrix.makeRotationY(-pi/2) @ Matrix.
            makeRotationX(-pi/2)
    topGeometry.applyMatrix( transform )
    self.merge( topGeometry )

if closedBottom:
   bottomGeometry = PolygonGeometry(radialSegments,
     radiusBottom)
   transform = Matrix.makeTranslation(0, -height/2, 0) @
         Matrix.makeRotationY(-pi/2) @ Matrix.
           makeRotationX(pi/2)
    bottomGeometry.applyMatrix( transform )
    self.merge( bottomGeometry )
```

To create cylinders, the same radius is used for the top and bottom, and the top and bottom sides will both be closed (present) or not. In the **geometry** folder, create a new file named **cylinderGeometry.py**, containing the following code:

```
from geometry.cylindricalGeometry import
  CylindricalGeometry
class CylinderGeometry(CylindricalGeometry):

    def __init__(self, radius=1, height=1,
                 radialSegments=32, heightSegments=4,
                   closed=True):

        super().__init__(radius, radius, height,
                         radialSegments, heightSegments,
                         closed, closed)
```

To create prisms, the parameter **radialSegments** is replaced by **sides** for clarity in this context. In the **geometry** folder, create a new file named **prismGeometry.py**, containing the following code:

```
from geometry.cylindricalGeometry import
CylindricalGeometry
class PrismGeometry(CylindricalGeometry):

    def __init__(self, radius=1, height=1,
             sides=6, heightSegments=4, closed=True):

        super().__init__(radius, radius, height,
                        sides, heightSegments,
                          closed, closed)
```

To create cones, the top radius will always be zero, and the top polygon side never needs to be rendered. In the **geometry** folder, create a new file named **coneGeometry.py**, containing the following code:

```
from geometry.cylindricalGeometry import
CylindricalGeometry
class ConeGeometry(CylindricalGeometry):

    def __init__(self, radius=1, height=1,
                 radialSegments=32, heightSegments=4,
                   closed=True):

        super().__init__(0, radius, height,
                        radialSegments, heightSegments,
                        False, closed)
```

Finally, creating pyramids is similar to creating cones, and as was the case for prisms, the parameter **radialSegments** is replaced by **sides** for clarity in this context. In the **geometry** folder, create a new file named **pyramidGeometry.py**, containing the following code:

```
from geometry.cylindricalGeometry import
CylindricalGeometry
class PyramidGeometry(CylindricalGeometry):

    def __init__(self, radius=1, height=1,
                 sides=4, heightSegments=4,
closed=True):

        super().__init__(0, radius, height,
                        sides, heightSegments, False,
                        closed)
```

4.4 MATERIAL OBJECTS

Material objects will store three types of data related to rendering: shader program references, **`Uniform`** objects, and OpenGL render settings. As was the case with the base **`Geometry`** class, there will be many extensions of the base **`Material`** class. For example, different materials will exist for rendering geometric data as a collection of points, as a set of lines, or as a surface. Some basic materials will implement vertex colors or uniform base colors, while advanced materials (developed in later chapters) will implement texture mapping, lighting, and other effects. The framework will also enable developers to easily write customized shaders in applications.

The tasks handled by the base **`Material`** class will include

- compiling the shader code and initializing the program
- initializing dictionaries to store uniforms and render settings
- defining uniforms corresponding to the model, view, and projection matrices, whose values are stored outside the material (in mesh and camera objects)
- define a method named **`addUniform`** to simplify creating and adding **`Uniform`** objects
- defining a method named **`locateUniforms`** that determines and stores all the uniform variable references in the shaders
- defining a method named **`setProperties`** that can be used to set multiple uniform and render setting values simultaneously from a dictionary (for convenience).

Classes that extend this class will

- contain the actual shader code
- add any extra uniform objects required by the shaders
- call the **`locateUniforms`** method once all uniform objects have been added
- add OpenGL render settings (as Python variables) to the settings dictionary
- implement a method named **`updateRenderSettings`**, which will call the OpenGL functions needed to configure the render settings previously specified.

4.4.1 Base Class

Since there will be many extensions of this class, all the material-related classes will be organized into a separate folder. To this end, in your main folder, create a new folder called **`material`**. To create the base class, in the **`material`** folder, create a new file called **`material.py`** with the following code:

```
from core.openGLUtils import OpenGLUtils
from core.uniform import Uniform
from OpenGL.GL import *

class Material(object):

    def __init__(self, vertexShaderCode,
      fragmentShaderCode):

        self.programRef = OpenGLUtils.
          initializeProgram(vertexShaderCode,
                                  fragmentShaderCode)

        # Store Uniform objects,
        #   indexed by name of associated variable in
              shader.
        self.uniforms = {}

        # Each shader typically contains these
            uniforms;
        #   values will be set during render process
              from Mesh/Camera.
        # Additional uniforms added by extending
             classes.
        self.uniforms["modelMatrix"]      =
          Uniform("mat4", None)
        self.uniforms["viewMatrix"]       =
          Uniform("mat4", None)
        self.uniforms["projectionMatrix"] =
          Uniform("mat4", None)

        # Store OpenGL render settings,
        #   indexed by variable name.
        # Additional settings added by extending
            classes.
        self.settings = {}
        self.settings["drawStyle"] = GL_TRIANGLES

    def addUniform(self, dataType, variableName, data):
        self.uniforms[variableName] =
          Uniform(dataType, data)


    # initialize all uniform variable references
    def locateUniforms(self):
        for variableName, uniformObject in self.
          uniforms.items():
            uniformObject.locateVariable(
                       self.programRef, variableName )

    # configure OpenGL with render settings
    def updateRenderSettings(self):
        pass

    # convenience method for setting multiple material
        "properties"
    #   (uniform and render setting values) from a
           dictionary
    def setProperties(self, properties):
        for name, data in properties.items():
            # update uniforms
            if name in self.uniforms.keys():
                self.uniforms[name].data = data
            # update render settings
            elif name in self.settings.keys():
                self.settings[name] = data
            # unknown property type
            else:
                raise Exception(
            "Material has no property named: " + name)
```

With this class completed, you will next turn your attention to creating extensions of this class.

4.4.2 Basic Materials

In this section, you will create an extension of the **Material** class, called **BasicMaterial**, which contains shader code and a set of

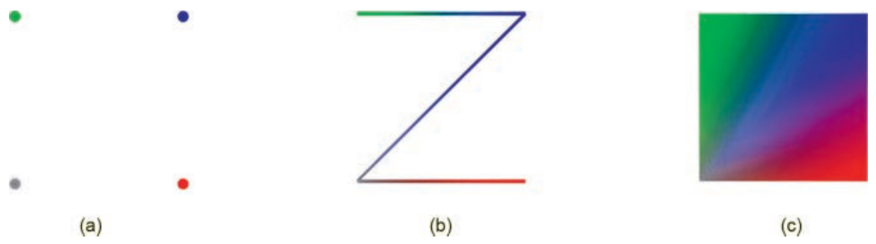

FIGURE 4.19 Rendering the six vertices of a Rectangle Geometry with a point material (a), line material (b), and surface material (c).

corresponding uniforms. The shaders can be used to render points, lines, or surfaces. Keeping modular design principles in mind, this class will in turn be extended into classes called **`PointMaterial`**, **`LineMaterial`**, and **`SurfaceMaterial`**, each of which will contain the relevant OpenGL render settings for the corresponding type of geometric primitive. Figure 4.19 illustrates the results of rendering the six vertices of a **`Rectangle`** object with each of these types of materials using vertex colors. Note that since the bottom-left vertex color has been changed to gray, it is visible against a white background and that the line material groups points into pairs and thus does not produce a full wireframe (although this will be possible with the surface material settings).

The shaders for the basic material will use two attributes: vertex positions and vertex colors. As before, attribute variables are designated with the type qualifier **`in`**. The vertex color data will be sent from the vertex shader to the fragment shader using the variable **`color`**. The uniform variables used by the vertex shader will include the model, view, and projection matrices, as usual, which are used to calculate the final position of each vertex. The two main options for coloring fragments are either to use interpolated vertex colors or to apply a single color to all vertices. To this end, there will be two additional uniform variables used by this shader. The first variable, **`baseColor`**, will be a **`vec3`** containing a color applied to all vertices, with the default value (1,1,1), corresponding to white. The second variable, **`useVertexColors`**, will be a boolean value that determines whether the data stored in the vertex color attribute will be applied to the base color. You do not need to include a boolean variable specifying whether base color should be used (in other words, there is no **`useBaseColor`** variable), because if the base color is left at its default value of (1,1,1), then combining this with other colors (by multiplication) will have no effect.

To implement this basic material, in the **`material`** folder, create a new file called **`basicMaterial.py`** with the following code:

```
from material.material import Material
from core.uniform import Uniform

class BasicMaterial(Material):

    def __init__(self):

        vertexShaderCode = """
        uniform mat4 projectionMatrix;
        uniform mat4 viewMatrix;
        uniform mat4 modelMatrix;
        in vec3 vertexPosition;
        in vec3 vertexColor;
        out vec3 color;

        void main()
        {
            gl_Position = projectionMatrix *
              viewMatrix * modelMatrix
              * vec4(vertexPosition, 1.0);
            color = vertexColor;
        }
        """

        fragmentShaderCode = """
        uniform vec3 baseColor;
        uniform bool useVertexColors;
        in vec3 color;
        out vec4 fragColor;

        void main()
        {
            vec4 tempColor = vec4(baseColor, 1.0);

            if ( useVertexColors )
                tempColor *= vec4(color, 1.0);

            fragColor = tempColor;
        }
        """
        super().__init__(vertexShaderCode,
          fragmentShaderCode)
        self.addUniform("vec3", "baseColor", [1.0,
          1.0, 1.0])
        self.addUniform("bool", "useVertexColors",
          False)
        self.locateUniforms()
```

Next, the render settings (such as **drawStyle**) need to be specified, and the **updateRenderSettings** function needs to be implemented. As previously mentioned, this will be accomplished with three classes that extend the **BasicMaterial** class.

The first extension will be the **PointMaterial** class, which renders vertices as points. Recall that render setting values are stored in the dictionary object named **settings** with various keys: strings, such as **"drawStyle"**. The draw style is the OpenGL constant GL_POINTS. The size of the points is stored with the key **"pointSize"**. The points may be drawn in a rounded style by setting the boolean variable with the key **"roundedPoints"** to **True**. Finally, the class constructor contains an optional dictionary object named **properties** that can be used to easily change the default values of any of these render settings or the previously discussed uniform values, using the function **setProperties**. To implement this class, in the **material** folder, create a new file called **pointMaterial.py** with the following code:

```
from material.basicMaterial import BasicMaterial
from OpenGL.GL import *

class PointMaterial(BasicMaterial):

    def __init__(self, properties={}):
        super().__init__()

        # render vertices as points
        self.settings["drawStyle"] = GL_POINTS
        # width and height of points, in pixels
        self.settings["pointSize"] = 8
        # draw points as rounded
        self.settings["roundedPoints"] = False

        self.setProperties(properties)
    def updateRenderSettings(self):

        glPointSize(self.settings["pointSize"])

        if self.settings["roundedPoints"]:
            glEnable(GL_POINT_SMOOTH)
        else:
            glDisable(GL_POINT_SMOOTH)
```

The second extension will be the **LineMaterial** class, which renders vertices as lines. In this case, there are three different ways to group vertices: as a connected set of points, a loop (additionally connecting the last point to the first), and as a disjoint set of line segments. These are specified by the OpenGL constants GL_LINE_STRIP, GL_LINE_LOOP, and GL_LINES, respectively, but for readability will be stored under the settings dictionary key **"lineType"** with the string values **"connected"**, **"loop"**, or **"segments"**. The other render setting is the thickness or width of the lines, stored with the key **"lineWidth"**. To implement this class, in the **material** folder, create a new file called **lineMaterial.py** with the following code:

```
from material.basicMaterial import BasicMaterial
from OpenGL.GL import *

class LineMaterial(BasicMaterial):

    def __init__(self, properties={}):
        super().__init__()

        # render vertices as continuous line by
            default
        self.settings["drawStyle"] = GL_LINE_STRIP
        # line thickness
        self.settings["lineWidth"] = 1
        # line type: "connected" | "loop" | "segments"
        self.settings["lineType"] = "connected"

        self.setProperties(properties)

    def updateRenderSettings(self):

        glLineWidth(self.settings["lineWidth"])
        if self.settings["lineType"] == "connected":
            self.settings["drawStyle"] =
              GL_LINE_STRIP
        elif self.settings["lineType"] == "loop":
            self.settings["drawStyle"] = GL_LINE_LOOP
        elif self.settings["lineType"] == "segments":
            self.settings["drawStyle"] = GL_LINES
        else:
            raise Exception("Unknown LineMaterial draw
              style.")
```

The third extension will be the **SurfaceMaterial** class, which renders vertices as a surface. In this case, the draw style is specified by the OpenGL constant GL_TRIANGLES. For rendering efficiency, OpenGL only renders the front side of triangles by default; the front side is defined to be the side from which the vertices appear to be listed in counterclockwise order. Both sides of each triangle can be rendered by changing the value stored with the key **"doubleSide"** to **True**. A surface can be rendered in wireframe style by changing the value stored with the key **"wireframe"** to **True**, in which case the thickness of the lines may also be set as with line-based materials with the dictionary key **"lineWidth"**. The results of rendering a shape in wireframe style (with double-sided rendering set to False) are illustrated in Figure 4.20. To implement this class, in the **material** folder, create a new file called **surfaceMaterial.py** with the following code:

```
from material.basicMaterial import BasicMaterial
from OpenGL.GL import *

class SurfaceMaterial(BasicMaterial):

    def __init__(self, properties={}):
        super().__init__()

        # render vertices as surface
        self.settings["drawStyle"] = GL_TRIANGLES
        # render both sides? default: front side only
        #   (vertices ordered counterclockwise)
        self.settings["doubleSide"] = False
        # render triangles as wireframe?
        self.settings["wireframe"] = False
```

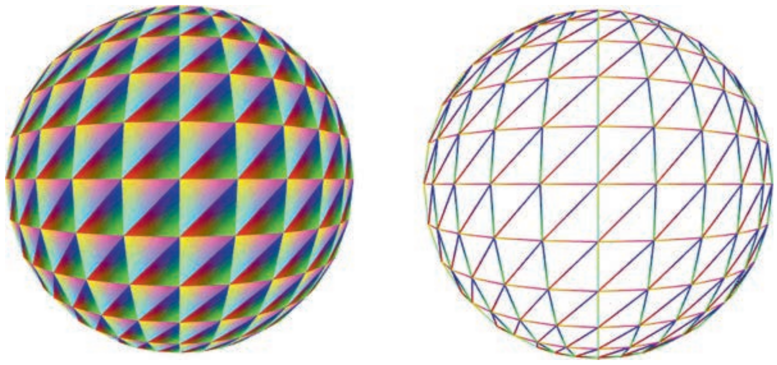

FIGURE 4.20 Rendering a sphere with triangles and as a wireframe.

```
        # 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"])
```

At this point, you have completed many geometry and material classes, which store all the information required to render an object. In the next section, you will create a class that uses this information in the process of rendering mesh objects.

4.5 RENDERING SCENES WITH THE FRAMEWORK

The final class required in the framework at this stage is the **`Renderer`** class. When initialized, this class will perform general rendering tasks, including enabling depth testing, antialiasing, and setting the color used when clearing the color buffer (the default background color). A function named **render** will take a **Scene** and a **Camera** object as input, and performs all of the rendering related tasks that you have seen in earlier examples. The color and depth buffers are cleared, and the camera's view matrix is updated. Next, a list of all the **Mesh** objects in the scene is created by first extracting all elements in the scene using the **get-DescendantList** function and then filtering this list using the Python functions **filter** and **isinstance**. Then, for each mesh that is visible, the following tasks need to be performed:

- the shader program being used must be specified
- the vertex array object that specifies the associations between vertex buffers and shader variables must be bound
- the values corresponding to the model, view, and projection matrices (stored in the mesh and camera) must be stored in the corresponding uniform objects
- the values in all uniform objects must be uploaded to the GPU
- render settings are applied via OpenGL functions as specified in the **updateRenderSettings** function
- the **glDrawArrays** function is called, specifying the correct draw mode and the number of vertices to be rendered.

To continue, in the **core** folder, create a new file called **renderer.py** with the following code:

```
from OpenGL.GL import *
from core.mesh import Mesh

class Renderer(object):

    def __init__(self, clearColor=[0,0,0]):

        glEnable( GL_DEPTH_TEST )
        # required for antialiasing
        glEnable( GL_MULTISAMPLE )
        glClearColor(clearColor[0], clearColor[1],
          clearColor[2], 1)

    def render(self, scene, camera):
        # clear color and depth buffers
        glClear(GL_COLOR_BUFFER_BIT |
          GL_DEPTH_BUFFER_BIT)

        # update camera view (calculate inverse)
        camera.updateViewMatrix()

        # extract list of all Mesh objects in scene
        descendantList = scene.getDescendantList()
        meshFilter = lambda x : isinstance(x, Mesh)
        meshList = list( filter( meshFilter,
          descendantList ) )

        for mesh in meshList:

            # if this object is not visible,
            #   continue to next object in list
            if not mesh.visible:
                continue

            glUseProgram( mesh.material.programRef )

            # bind VAO
            glBindVertexArray( mesh.vaoRef )

            # update uniform values stored outside of
                material
            mesh.material.uniforms["modelMatrix"].data =
                                 mesh.getWorldMatrix()
             mesh.material.uniforms["viewMatrix"].data =
                                  camera.viewMatrix
             mesh.material.
               uniforms["projectionMatrix"].data =
                               camera.projectionMatrix

            # update uniforms stored in material
            for variableName, uniformObject in
                     mesh.material.uniforms.items():
                uniformObject.uploadData()

            # update render settings
            mesh.material.updateRenderSettings()
            glDrawArrays( mesh.material.
              settings["drawStyle"], 0,
                         mesh.geometry.vertexCount )
```

At this point, you are now ready to create an application using the graphics framework! Most applications will require at least seven classes to be imported: **Base**, **Renderer**, **Scene**, **Camera**, **Mesh**, and at least one geometry and one material class to be used in the mesh. This example also illustrates how a scene can be rendered in a non-square window without distortion by setting the aspect ratio of the camera. (If using the default window size, this parameter is not necessary.) To create the application that consists of a spinning cube, in your main project folder, create a new file named **test-4-1.py**, containing the following code:

```
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 geometry.boxGeometry     import BoxGeometry
from material.surfaceMaterial import SurfaceMaterial


# render a basic scene
class Test(Base):

    def initialize(self):
        print("Initializing program...")

        self.renderer = Renderer()
        self.scene = Scene()
        self.camera   = Camera( aspectRatio=800/600 )
        self.camera.setPosition( [0, 0, 4] )

        geometry = BoxGeometry()
        material = SurfaceMaterial(
          {"useVertexColors": True} )
        self.mesh = Mesh( geometry, material )
        self.scene.add( self.mesh )

    def update(self):
        self.mesh.rotateY( 0.0514 )
        self.mesh.rotateX( 0.0337 )
        self.renderer.render( self.scene, self.camera
)

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

Running this code should produce a result similar to that illustrated in Figure 4.21, where the dark background is due to the default clear color in the renderer being used.

Hopefully, the first thing you noticed about the application code was that it is quite short, and focuses on high-level concepts. This is thanks to all the work that went into writing the framework classes in this chapter. At this point, you should try displaying the other geometric shapes that you have implemented to confirm that they appear as expected. In addition, you should also try out the other materials and experiment with changing the default uniform values and render settings. When using a dictionary to set more than one of these properties, using multiline formatting might make your code easier to read. For example, you could configure the material in the previous example using the following code:

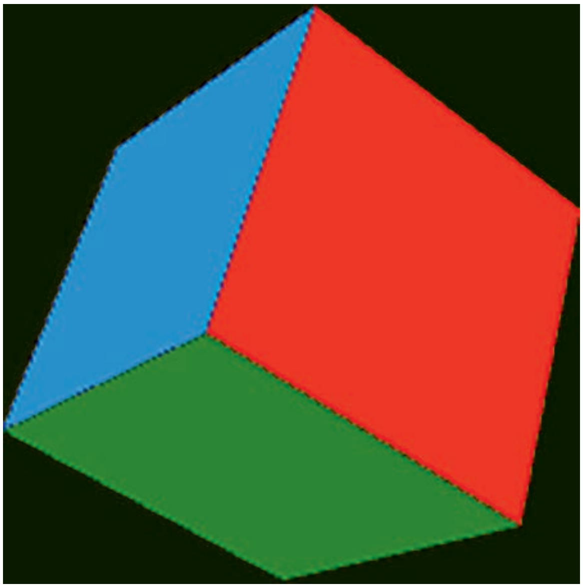

FIGURE 4.21 Rendering a spinning cube with the graphics framework classes.

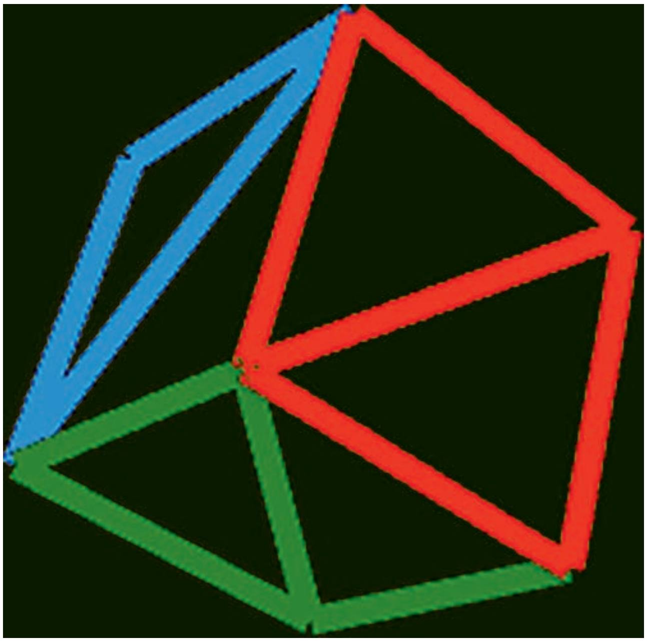

FIGURE 4.22 Rendering a cube with alternate material properties.

```
material = SurfaceMaterial({
    "useVertexColors": True,
    "wireframe":       True,
    "lineWidth":       8
})
```

This would produce a result similar to that shown in Figure 4.22.

Before proceeding, it will be very helpful to create a template file containing most of this code. To this end, in your main project folder, save a copy of the file named **`test-4-1.py`** as a new file named **`test-template.py`**, and comment out the two lines of code in the **`update`** function that rotate the mesh.

The remaining examples in this section will illustrate how to create custom geometry and custom material objects in an application.

4.6 CUSTOM GEOMETRY AND MATERIAL OBJECTS

The first example will demonstrate how to create a custom geometry object by explicitly listing the vertex data, similar to the geometry classes representing rectangles and boxes; the result will be as shown in Figure 4.23.

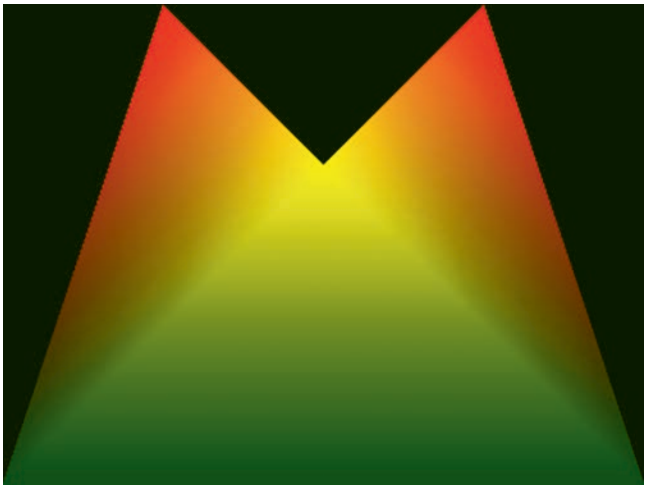

FIGURE 4.23 A custom geometry.

To begin, create a copy of the file **`test-template.py`**, save it as **`test-4-2.py`**. Whenever you want to create your own customized geometry, you will need to import the **`Geometry`** class. Therefore, add the following import statement at the top of the file:

```
from geometry.geometry import Geometry
```

Next, replace the line of code where the geometry object is initialized with the following block of code:

```
geometry = Geometry()
P0 = [-0.1,  0.1, 0.0]
P1 = [ 0.0,  0.0, 0.0]
P2 = [ 0.1,  0.1, 0.0]
P3 = [-0.2, -0.2, 0.0]
P4 = [ 0.2, -0.2, 0.0]
posData = [P0,P3,P1, P1,P3,P4, P1,P4,P2]
geometry.addAttribute("vec3", "vertexPosition",
posData)
R = [1, 0, 0]
Y = [1, 1, 0]
G = [0, 0.25, 0]
colData = [R,G,Y, Y,G,G, Y,G,R]
geometry.addAttribute("vec3", "vertexColor", colData)
geometry.countVertices()
```

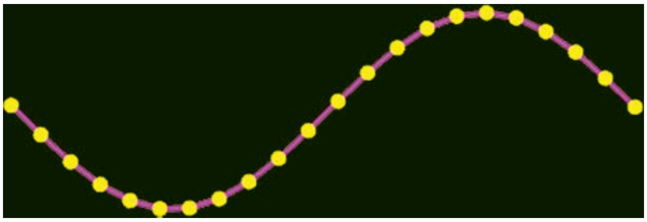

FIGURE 4.24 A custom geometry with data generated from a function.

With these changes, save and run your file, and you should see a result similar to that in Figure 4.23.

For all but the simplest models, listing the vertices by hand can be a tedious process, and so you may wish to generate vertex data using functions. The next example generates the image from Figure 4.24 from vertices generated along the graph of a sine function. This particular appearance is generated by drawing the same geometric data twice: once using a point-based material and once using a line-based material.

To begin, create a copy of the file **test-template.py** and save it as **test-4-3.py**. As before, you will need to import the Geometry and Attribute classes; in addition, you will need the **sin** function from the **math** package, the **arange** function from **numpy** (to generate a range of decimal values), and the point-based and line-based basic material classes. Therefore, add the following import statements at the top of the file:

```
from geometry.geometry import Geometry
from math import sin
from numpy import arange
from material.pointMaterial import PointMaterial
from material.lineMaterial import LineMaterial
```

Next, in the **initialize** function, delete the code in that function that occurs after the camera position is set, replacing it with the following:

```
geometry = Geometry()
posData = []
for x in arange(-3.2, 3.2, 0.3):
    posData.append([x, sin(x), 0])
geometry.addAttribute("vec3", "vertexPosition",
  posData)
geometry.countVertices()
```

```
pointMaterial = PointMaterial(
     {"baseColor": [1,1,0], "pointSize": 10} )
pointMesh = Mesh( geometry, pointMaterial )

lineMaterial = LineMaterial( {"baseColor": [1,0,1],
  "lineWidth": 4} )
lineMesh = Mesh( geometry, lineMaterial )

self.scene.add( pointMesh )
self.scene.add( lineMesh )
```

Note that vertex color data does not need to be generated, since the material's base color is used when rendering. Save and run this file, and the result will be similar to Figure 4.24.

Next, you will turn your attention to customized materials, where the shader code, uniforms, and render settings are part of the application code. In the next example, you will color the surface of an object based on the coordinates of each point on the surface. In particular, you will take the fractional part of the *x*, *y*, and *z* coordinates of each point and use these for the red, green, and blue components of the color. The fractional part is used because this is a value between 0 and 1, which is the range of color components. Figure 4.25 shows the effect of applying this shader to a sphere of radius 3.

As before, create a copy of the file **`test-template.py`**, this time saving it with the file name **`test-4-4.py`**. Whenever you want to create your own customized material, you will need to import the **`Material`** class, and possibly also the OpenGL functions and constants. Therefore, add the following import statements at the top of your new application:

```
from geometry.sphereGeometry import SphereGeometry
from material.material import Material
```

Next, in the **`initialize`** function, delete the code in that function that occurs after the camera object is initialized, and replace it with the following code. Note that there are **`out`** and **`in`** variables named **`position`**, which are used to transmit position data from the vertex shader to the fragment shader (which, as usual, is interpolated for each fragment). Additionally, to obtain the fractional part of each coordinate, the values are reduced modulo 1 using the GLSL function **`mod`**. In this example, uniform objects do not need to be created for the matrices, as this is handled by the **`Mesh`** class.

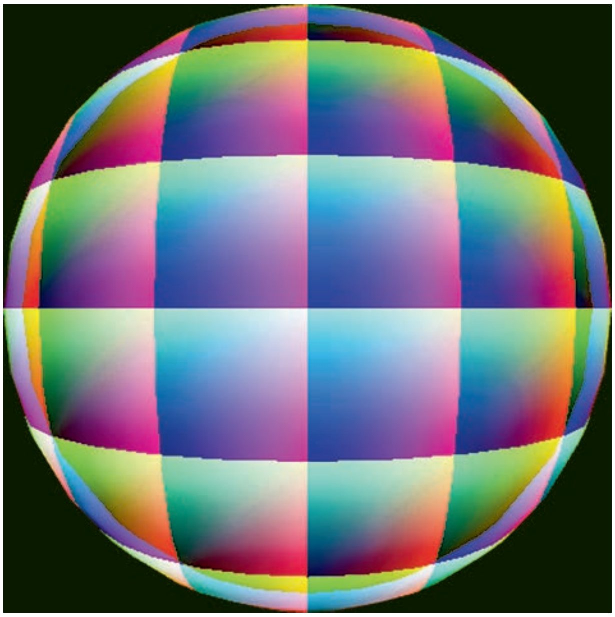

FIGURE 4.25 Coloring the surface of a sphere based on point coordinates.

```
self.camera.setPosition( [0, 0, 7] )

geometry = SphereGeometry(radius=3)

vsCode = """
in vec3 vertexPosition;
out vec3 position;
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;
void main()
{
    vec4 pos = vec4(vertexPosition, 1.0);
    gl_Position = projectionMatrix * viewMatrix *
modelMatrix * pos;
    position = vertexPosition;
}
"""

fsCode = """
in vec3 position;
```

```
out vec4 fragColor;
void main()
{
    vec3 color = mod(position, 1.0);
    fragColor = vec4(color, 1.0);
}
"""

material = Material(vsCode, fsCode)
material.locateUniforms()

self.mesh = Mesh( geometry, material )
self.scene.add( self.mesh )
```

It is easy to see the red and green color gradients on the rendered sphere, but not the blue gradient, due to the orientation of the sphere and the position of the camera (looking along the *z*-axis). If desired, you may add code to the **`update`** function that will rotate this mesh around the y-axis, to get a fuller understanding of how the colors are applied across the surface.

The final example in this section will illustrate how to create animated effects in both the vertex shader and the fragment shader, using a custom material. Once again, you will use a spherical shape for the geometry. In the material's vertex shader, you will add an offset to the *y*-coordinate, based on the sine of the *x*-coordinate, and shift the displacement over time. In the material's fragment shader, you will shift between the geometry's vertex colors and a shade of red in a periodic manner. A still image from this animation is shown in Figure 4.26.

Create a copy of the file **`test-template.py`**, and save it with the file name **`test-4-5.py`**. Add the same import statements at the top of your new application as before:

```
from geometry.sphereGeometry import SphereGeometry
from material.material import Material
```

In the **`initialize`** function, delete the code in that function that occurs after the camera object is initialized, and replace it with the following code. Note that in this example, there is a uniform variable called **`time`** present in both the vertex shader and the fragment shader, for which a Uniform object will need to be created. Also note the creation of the Python variable **`self.time`**, which will be used to supply the value to the uniform later.

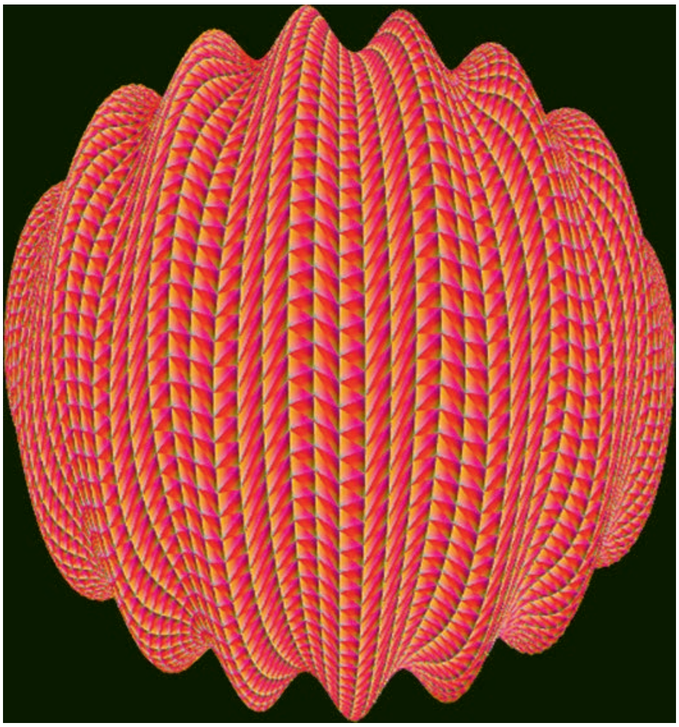

FIGURE 4.26 A sphere with periodic displacement and color shifting.

```
self.camera.setPosition( [0, 0, 7] )

geometry = SphereGeometry(radius=3,
radiusSegments=128, heightSegments=64)

vsCode = """
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;
in vec3 vertexPosition;
in vec3 vertexColor;
out vec3 color;
uniform float time;
void main()
{
    float offset = 0.2 * sin(8.0 * vertexPosition.x +
      time);
    vec3 pos = vertexPosition + vec3(0.0, offset, 0.0);
    gl_Position = projectionMatrix * viewMatrix *
         modelMatrix *
                vec4(pos, 1);
    color = vertexColor;
}
"""

fsCode = """
in vec3 color;
uniform float time;
out vec4 fragColor;
void main()
{
    float r = abs(sin(time));
    vec4 c = vec4(r, -0.5*r, -0.5*r, 0.0);
    fragColor = vec4(color, 1.0) + c;
}
"""

material = Material(vsCode, fsCode)
material.addUniform("float", "time", 0)
material.locateUniforms()

self.time = 0;

self.mesh = Mesh( geometry, material )
self.scene.add( self.mesh )
```

Finally, to produce the animated effect, you must increment and update the value of the **`time`** variable. In the **`update`** function, add the following code before the **`render`** function is called:

```
self.time += 1/60
self.mesh.material.uniforms["time"].data = self.time
```

With these additions, this example is complete. Run the code and you should see an animated rippling effect on the sphere, as the color shifts back and forth from the red end of the spectrum.

4.7 EXTRA COMPONENTS

Now that you are familiar with writing customized geometric objects, there are a number of useful, reusable classes you will add to the framework: axes and grids, to more easily orient the viewer. Following this, you will create a movement rig, enabling you to more easily create interactive scenes by moving the camera or objects in the scene in an intuitive way.

4.7.1 Axes and Grids

At present, there is no easy way to determine one's orientation relative to the scene, or a sense of scale, within a three-dimensional scene built in this framework. One approach that can partially alleviate these issues is to create three-dimensional axis and grid objects, illustrated separately and together in Figure 4.27.

For convenience, each of these objects will extend the Mesh class, and set up their own Geometry and Material within the class. Since they are not really of core importance to the framework, in order to keep the file system organized, in your main project folder, create a new folder called **`extras`**.

First, you will implement the object representing the (positive) coordinate axes. By default, the *x*, *y*, and *z* axes will have length 1 and be rendered with red, green, and blue lines, using a basic line material, although these parameters will be able to be adjusted in the constructor. In the **`extras`** folder, create a new file named **`axesHelper.py`** with the following code:

```
from core.mesh import Mesh
from geometry.geometry import Geometry
from material.lineMaterial import LineMaterial

class AxesHelper(Mesh):

    def __init__(self, axisLength=1, lineWidth=4,
                 axisColors=[[1,0,0],[0,1,0],[0,0,1]]
):

        geo = Geometry()
```

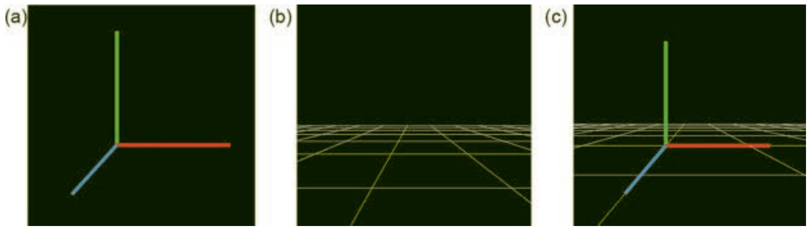

FIGURE 4.27 Coordinate axes (a), a grid (b), and in combination (c).

```
        positionData = [[0,0,0], [axisLength,0,0],
                        [0,0,0], [0,axisLength,0],
                        [0,0,0], [0,0,axisLength]]


        colorData = [axisColors[0], axisColors[0],
                     axisColors[1], axisColors[1],
                     axisColors[2], axisColors[2]]



        geo.addAttribute("vec3", "vertexPosition",
          positionData)
        geo.addAttribute("vec3", "vertexColor",
          colorData)
        geo.countVertices()

        mat = LineMaterial({
            "useVertexColors": True,
            "lineWidth":       lineWidth,
            "lineType":        "segments"
        })

        # initialize the mesh
        super().__init__(geo, mat)
```

Next, you will create a (square) grid object. Settings that you will be able to customize will include the dimensions of the grid, the number of divisions on each side, the color of the grid lines, and a separate color for the central grid line. In the **extras** folder, create a new file named **gridHelper.py** containing the following code:

```
from core.mesh import Mesh
from geometry.geometry import Geometry
from material.lineMaterial import LineMaterial

class GridHelper(Mesh):

    def __init__(self, size=10, divisions=10,
      gridColor=[0,0,0], centerColor=[0.5,0.5,0.5],
                          lineWidth=1):

        geo = Geometry()
```

```
positionData = []
colorData = []

# create range of values
values = []
deltaSize = size/divisions
for n in range(divisions+1):
    values.append( -size/2 + n * deltaSize )

# add vertical lines
for x in values:
    positionData.append( [x, -size/2, 0] )
    positionData.append( [x,  size/2, 0] )
    if x == 0:
        colorData.append(centerColor)
        colorData.append(centerColor)
    else:
        colorData.append(gridColor)
        colorData.append(gridColor)


# add horizontal lines
for y in values:
    positionData.append( [-size/2, y, 0] )
    positionData.append( [ size/2, y, 0] )
    if y == 0:
        colorData.append(centerColor)
        colorData.append(centerColor)
    else:
        colorData.append(gridColor)
        colorData.append(gridColor)

geo.addAttribute("vec3", "vertexPosition",
  positionData)
geo.addAttribute("vec3", "vertexColor",
  colorData)
geo.countVertices()

mat = LineMaterial({
    "useVertexColors": 1,
    "lineWidth":       lineWidth,
    "lineType":        "segments"
})
```

```
        # initialize the mesh
        super().__init__(geo, mat)
```

Note that the grid will by default by parallel to the *xy*-plane. For it to appear horizontal, as in Figure 4.27, you could rotate it by 90° around the *x*-axis. In order to see how these classes are used in code, to produce an image like the right side of Figure 4.27, make a copy of the file **test-template.py**, and save it as **test-4-6.py**. First, in the beginning of this new file, add the following import statements:

```
from extras.axesHelper import AxesHelper
from extras.gridHelper import GridHelper
from math import pi
```

Then, in the **initialize** function, delete the code in that function that occurs after the camera object is initialized, and replace it with the following code, which adds coordinate axes and a grid to the scene, and demonstrates use of some of the available customization parameters.

```
self.camera.setPosition( [0.5, 1, 5] )

axes = AxesHelper(axisLength=2)
self.scene.add( axes )

grid = GridHelper(size=20, gridColor=[1,1,1],
centerColor=[1,1,0])
grid.rotateX(-pi/2)
self.scene.add(grid)
```

When running this test application, you should see axes and a grid as previously described.

4.7.2 Movement Rig

As the final topic in this chapter, you will learn how to create a movement rig: an object with a built-in control system that can be used to move the camera or other attached objects within a scene in a natural way, similar to the way person might move around a field: moving forwards and backwards, left and right (all local translations), as well as turning left and right, and looking up and down. Here, the use of the verb "look" indicates that even if a person's point of view tilts up or down, their movement is still aligned with the horizontal plane. The only unrealistic movement feature that will be incorporated will be that the attached object will also be able to move up and down along the direction of the *y*-axis (perpendicular to the horizontal plane).

To begin, this class, which will be called **`MovementRig`**, naturally extends the **`Object3D`** class. In addition, to support the "look" feature, it will take advantage of the scene graph structure, by way of including a child Object3D; the move and turn motions will be applied to the base Object3D, while the look motions will be applied to the child Object3D (and thus the orientation resulting from the current look angle will have no effect on the move and turn motions). However, in order to properly attach objects to the movement rig (to the child object within the rig) will require the Object3D functions **`add`** and **`remove`** to be overridden. For convenience, the rate of each motion will be able to be specified. To begin, in the **`extras`** folder, create a new file named **`movementRig.py`** with the following code:

```
from core.object3D import Object3D

class MovementRig(Object3D):

    def __init__(self, unitsPerSecond=1,
      degreesPerSecond=60):

        # initialize base Object3D; controls movement
        # and turn left/right
        super().__init__()

        # initialize attached Object3D; controls look
            up/down
        self.lookAttachment = Object3D()
        self.children = [ self.lookAttachment ]
        self.lookAttachment.parent = self

        # control rate of movement
        self.unitsPerSecond   = unitsPerSecond
        self.degreesPerSecond = degreesPerSecond

    # adding and removing objects applies to look
        attachment;
    #   override functions from Object3D class
    def add(self, child):
        self.lookAttachment.add(child)

    def remove(self, child):
        self.lookAttachment.remove(child)
```

Next, in order to conveniently handle movement controls, this class will have an **update** function that takes an **Input** object as a parameter, and if certain keys are pressed, transforms the movement rig correspondingly. In order to provide the developer the ability to easily configure the keys being used, they will be assigned to variables in the class, and in theory, one could even disable certain types of motion by assigning the value None to any of these motions. The default controls will follow the standard practice of using the "w" / "a" / "s" / "d" keys for movement forwards / left / backwards / right. The letters "q" and "e" will be used for turning left and right, as they are positioned above the keys for moving left and right. Movement up and down will be assigned to the keys “r” and “f”, which can be remembered with the mnemonic words "rise" and "fall", and "r" is positioned in the row above “f”. Finally, looking up and down will be assigned to the keys "t" and "g", as they are positioned adjacent to the keys for moving up and down. To implement this, in the—**init**—function, add the following code:

```
# customizable key mappings
#   defaults: WASDRF (move), QE (turn), TG (look)
self.KEY_MOVE_FORWARDS  = "w"
self.KEY_MOVE_BACKWARDS = "s"
self.KEY_MOVE_LEFT      = "a"
self.KEY_MOVE_RIGHT     = "d"
self.KEY_MOVE_UP        = "r"
self.KEY_MOVE_DOWN      = "f"
self.KEY_TURN_LEFT      = "q"
self.KEY_TURN_RIGHT     = "e"
self.KEY_LOOK_UP        = "t"
self.KEY_LOOK_DOWN      = "g"
```

Finally, in the **MovementRig** class, add the following function, which also calculates the amount of motion that should occur based on **deltaTime**: the amount of time that has elapsed since the previous update.

```
def update(self, inputObject, deltaTime):
    moveAmount = self.unitsPerSecond * deltaTime
    rotateAmount = self.degreesPerSecond *
                   (3.1415926 / 180) * deltaTime


    if inputObject.isKeyPressed(self.
       KEY_MOVE_FORWARDS):
        self.translate( 0, 0, -moveAmount )
    if inputObject.isKeyPressed(self.
       KEY_MOVE_BACKWARDS):
        self.translate( 0, 0, moveAmount )
    if inputObject.isKeyPressed(self.KEY_MOVE_LEFT):
        self.translate( -moveAmount, 0, 0 )
    if inputObject.isKeyPressed(self.KEY_MOVE_RIGHT):
        self.translate( moveAmount, 0, 0 )
    if inputObject.isKeyPressed(self.KEY_MOVE_UP):
        self.translate( 0, moveAmount, 0 )
    if inputObject.isKeyPressed(self.KEY_MOVE_DOWN):
        self.translate( 0, -moveAmount, 0 )

    if inputObject.isKeyPressed(self.KEY_TURN_RIGHT):
        self.rotateY( -rotateAmount )
    if inputObject.isKeyPressed(self.KEY_TURN_LEFT):
        self.rotateY( rotateAmount )

    if inputObject.isKeyPressed(self.KEY_LOOK_UP):
        self.lookAttachment.rotateX( rotateAmount )
    if inputObject.isKeyPressed(self.KEY_LOOK_DOWN):
        self.lookAttachment.rotateX( -rotateAmount )
```

To see one way to use this class, in the previous application file (**test-4-6.py**), add the following import statement:

```
from extras.movementRig import MovementRig
```

Then, in the **initialize** function, delete the line of code that sets the position of the camera, and add the following code instead:

```
self.rig = MovementRig()
self.rig.add( self.camera )
self.rig.setPosition( [0.5, 1, 5] )
self.scene.add( self.rig )
```

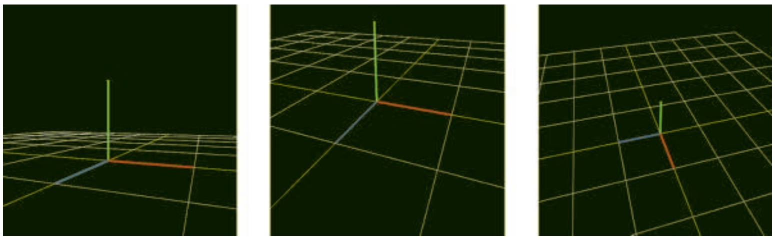

FIGURE 4.28 Multiple views of the coordinate axes and grid.

Finally, in the **update** function, add the following line of code:

```
self.rig.update( self.input, self.deltaTime )
```

When you run this application, it will initially appear similar to the right side of Figure 4.27. However, by pressing the motion keys as previously indicated, you should easily be able to view the scene from many different perspectives, some of which are illustrated in Figure 4.28.

Another way the **MovementRig** class may be used is by adding a cube or other geometric object to the rig, instead of a camera. While the view from the camera will remain fixed, this approach will enable you to move an object around the scene in a natural way.

4.8 SUMMARY AND NEXT STEPS

Building on your knowledge and work from previous chapters, in this chapter, you have seen the graphics framework truly start to take shape. You learned about the advantages of a scene graph framework and began by developing classes corresponding to the nodes: Scene, Group, Camera, and Mesh. Then, you created many classes that generate geometric data corresponding to many different shapes you may want to render: rectangles, boxes, polygons, spheres, cylinders, and more. You also created classes that enabled these objects to be rendered as collections of points, lines, or triangulated surfaces. After learning how to render objects in this new framework, you also learned how customized geometry or material objects can be created. Finally, you created some extra classes representing coordinate axes and grids, to help the viewer to have a sense of orientation and scale within the scene, and a movement rig class, to help the viewer interact with the scene, by moving the camera or other objects with a natural control scheme.

In the next chapter, you will move beyond vertex colors and learn about textures: images applied to surfaces of objects, which can add realism and sophistication to your three-dimensional scenes.