Developing Graphics Frameworks with Python and OpenGL

Lee Stemkoski
Michael Pascale

CRC Press
Taylor & Francis Group

Developing Graphics Frameworks with Python and OpenGL

Developing Graphics Frameworks with Python and OpenGL

Lee Stemkoski
Michael Pascale

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

First edition published 2022
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

Library of Congress Cataloging-in-Publication Data
Names: Stemkoski, Lee, author. | Pascale, Michael, author.
Title: Developing graphics frameworks with Python and OpenGL /
Lee Stemkoski, Michael Pascale.
Description: First edition. | Boca Raton : CRC Press, 2021. |
Includes bibliographical references and index.
Identifiers: LCCN 2021002036 | ISBN 9780367721800 (hardback) |
ISBN 9781003181378 (ebook)
Subjects: LCSH: OpenGL. | Computer graphics—Computer programs. |
Python (Computer program language) | Computer graphics—Mathematics.
Classification: LCC T385 .S7549 2021 | DDC 006.6—dc23
LC record available at https://lccn.loc.gov/2021002036

ISBN: 978-0-367-72180-0 (hbk)
ISBN: 978-1-032-02146-1 (pbk)
ISBN: 978-1-003-18137-8 (ebk)

DOI: 10.1201/9781003181378

Typeset in Minion Pro
by codeMantra

Contents

Authors

Lee Stemkoski is a professor of mathematics and computer science. He earned his Ph.D. in mathematics from Dartmouth College in 2006 and has been teaching at the college level since. His specialties are computer graphics, video game development, and virtual and augmented reality programming.

Michael Pascale is a software engineer interested in the foundations of computer science, programming languages, and emerging technologies. He earned his B.S. in Computer Science from Adelphi University in 2019. He strongly supports open source software and open access educational resources.

CHAPTER 1

Introduction to Computer Graphics

The importance of computer graphics in modern society is illustrated by the great quantity and variety of applications and their impact on our daily lives. Computer graphics can be two-dimensional (2D) or three-dimensional (3D), animated, and interactive. They are used in data visualization to identify patterns and relationships, and also in scientific visualization, enabling researchers to model, explore, and understand natural phenomena. Computer graphics are used for medical applications, such as magnetic resonance imaging (MRI) and computed tomography (CT) scans, and architectural applications, such as creating blueprints or virtual models. They enable the creation of tools such as training simulators and software for computer-aided engineering and design. Many aspects of the entertainment industry make use of computer graphics to some extent: movies may use them for creating special effects, generating photorealistic characters, or rendering entire films, while video games are primarily interactive graphics-based experiences. Recent advances in computer graphics hardware and software have even helped virtual reality and augmented reality technology enter the consumer market.

The field of computer graphics is continuously advancing, finding new applications, and increasing in importance. For all these reasons, combined with the inherent appeal of working in a highly visual medium, the field of computer graphics is an exciting area to learn about, experiment with, and work in. In this book, you'll learn how to create a robust framework

DOI: 10.1201/9781003181378-1

capable of rendering and animating interactive three-dimensional scenes using modern graphics programming techniques.

Before diving into programming and code, you'll first need to learn about the core concepts and vocabulary in computer graphics. These ideas will be revisited repeatedly throughout this book, and so it may help to periodically review parts of this chapter to keep the overall process in mind. In the second half of this chapter, you'll learn how to install the necessary software and set up your development environment.

1.1 CORE CONCEPTS AND VOCABULARY

Our primary goal is to generate two-dimensional images of three-dimensional scenes; this process is called *rendering* the scene. Scenes may contain two- and three-dimensional objects, from simple geometric shapes such as boxes and spheres, to complex models representing real-world or imaginary objects such as teapots or alien lifeforms. These objects may simply appear to be a single color, or their appearance may be affected by textures (images applied to surfaces), light sources that result in *shading* (the darkness of an object not in direct light) and *shadows* (the silhouette of one object's shape on the surface of another object), or environmental properties such as fog. Scenes are rendered from the point of view of a virtual camera, whose relative position and orientation in the scene, together with its intrinsic properties such as angle of view and depth of field, determine which objects will be visible or partially obscured by other objects when the scene is rendered. A 3D scene containing multiple shaded objects and a virtual camera is illustrated in Figure 1.1. The region contained within the truncated pyramid shape outlined in white (called a *frustum*) indicates the space visible to the camera. In Figure 1.1, this region completely contains the red and green cubes, but only contains part of the blue sphere, and the yellow cylinder lies completely outside of this region. The results of rendering the scene in Figure 1.1 are shown in Figure 1.2.

From a more technical, lower-level perspective, rendering a scene produces a *raster*—an array of *pixels* (picture elements) which will be displayed on a screen, arranged in a two-dimensional grid. Pixels are typically extremely small; zooming in on an image can illustrate the presence of individual pixels, as shown in Figure 1.3.

On modern computer systems, pixels specify colors using triples of floating-point numbers between 0 and 1 to represent the amount of red, green, and blue light present in a color; a value of 0 represents no amount of that color is present, while a value of 1 represents that color is displayed

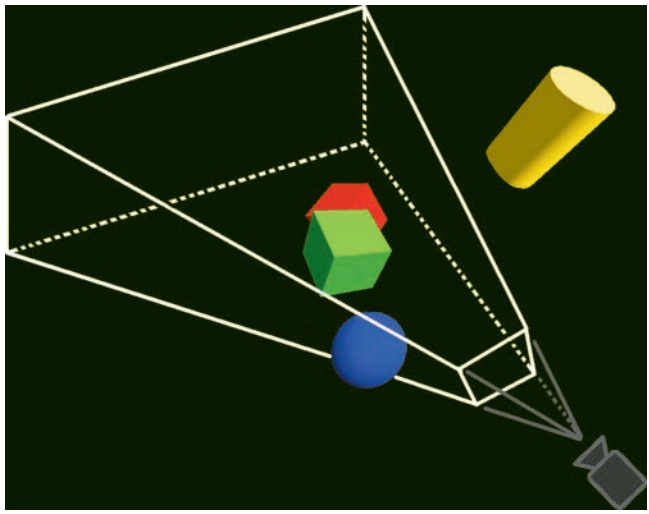

FIGURE 1.1 Three-dimensional scene with geometric objects, viewing region (white outline) and virtual camera (lower right).

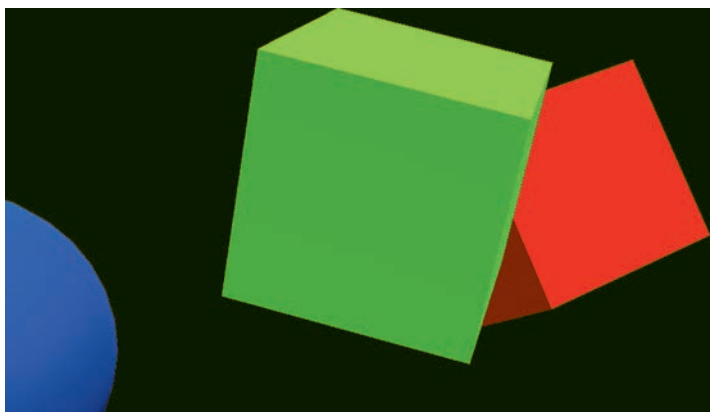

FIGURE 1.2 Results of rendering the scene from Figure 1.1

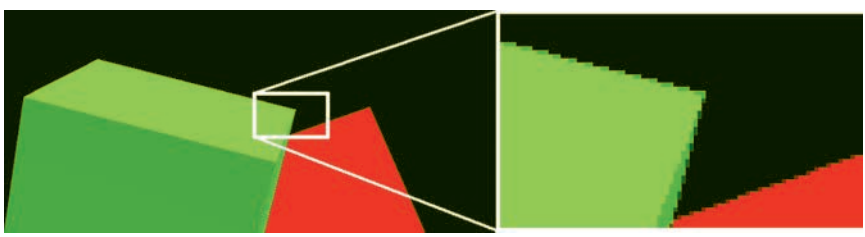

FIGURE 1.3 Zooming in on an image to illustrate individual pixels.

FIGURE 1.4 Various colors and their corresponding (R, G, B) values.

at full (100%) intensity. These three colors are typically used since photoreceptors in the human eye take in those particular colors. The triple (1, 0, 0) represents red, (0, 1, 0) represents green, and (0, 0, 1) represents blue. Black and white are represented by (0, 0, 0) and (1, 1, 1), respectively. Additional colors and their corresponding triples of values specifying the amounts of red, green, and blue (often called *RGB values*) are illustrated in Figure 1.4.

The quality of an image depends in part on its *resolution* (the number of pixels in the raster) and *precision* (the number of bits used for each pixel). As each bit has two possible values (0 or 1), the number of colors that can be expressed with N-bit precision is 2^N. For example, early video game consoles with 8-bit graphics were able to display $2^8 = 256$ different colors. Monochrome displays could be said to have 1-bit graphics, while modern displays often feature "high color" (16-bit, 65,536 color) or "true color" (24-bit, more than 16 million colors) graphics. Figure 1.5 illustrates the same image rendered with high precision but different resolutions, while Figure 1.6 illustrates the same image rendered with high resolution but different precision levels.

In computer science, a *buffer* (or *data buffer*, or *buffer memory*) is a part of a computer's memory that serves as temporary storage for data while it is being moved from one location to another. Pixel data is stored in a region of memory called the *framebuffer*. A framebuffer may contain multiple buffers that store different types of data for each pixel. At a minimum, the framebuffer must contain a *color buffer*, which stores RGB values. When rendering a 3D scene, the framebuffer must also contain a *depth buffer*, which stores distances from points on scene objects to the virtual camera. Depth values are used to determine whether the various points on each object are in front of or behind other objects (from the camera's perspective), and thus whether they will be visiblewhen the scene is rendered. If one scene object obscures another and a transparency effect is

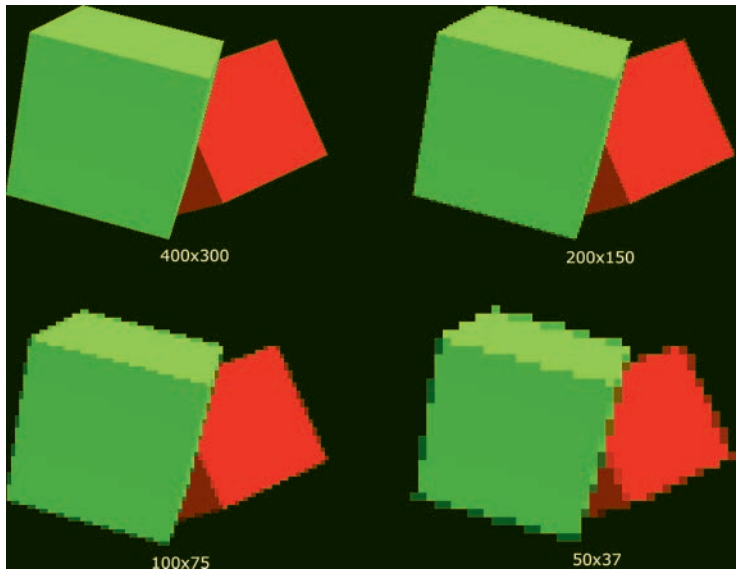

FIGURE 1.5 A single image rendered with different resolutions.

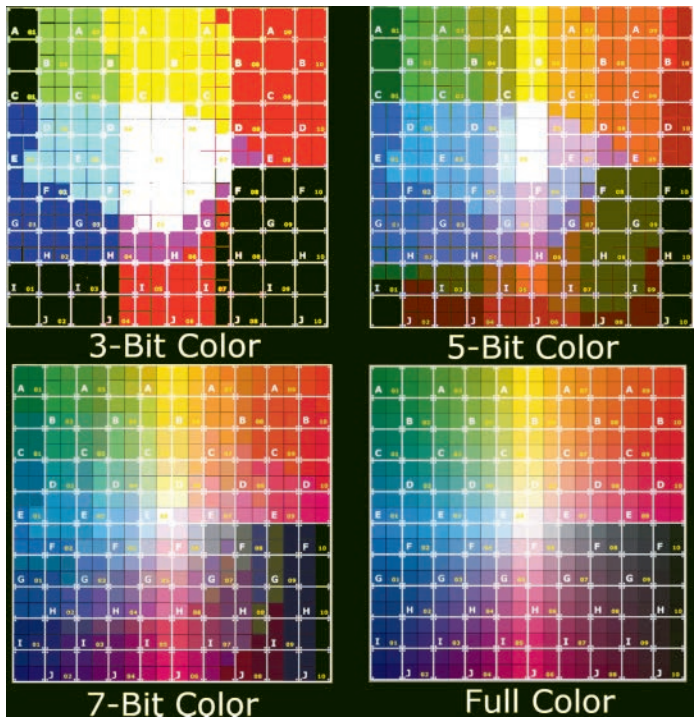

FIGURE 1.6 A single image rendered with different precisions.

desired, the renderer makes use of *alpha values*: floating-point numbers between 0 and 1 that specifies how overlapping colors should be blended together; the value 0 indicates a fully transparent color, while the value 1 indicates a fully opaque color. Alpha values are also stored in the color buffer along with RGB color values; the combined data is often referred to as RGBA color values. Finally, framebuffers may contain a buffer called a *stencil buffer*, which may be used to store values used in generating advanced effects, such as shadows, reflections, or portal rendering.

In addition to rendering three-dimensional scenes, another goal in computer graphics is to create animated scenes. Animations consist of a sequence of images displayed in quick enough succession that the viewer interprets the objects in the images to be continuously moving or changing in appearance. Each image that is displayed is called a *frame*. The speed at which these images appear is called the *frame rate* and is measured in *frames per second* (FPS). The standard frame rate for movies and television is 24 FPS. Computer monitors typically display graphics at 60 FPS. For virtual reality simulations, developers aim to attain 90 FPS, as lower frame rates may cause disorientation and other negative side effects in users. Since computer graphics must render these images in real time, often in response to user interaction, it is vital that computers be able to do so quickly.

In the early 1990s, computers relied on the *central processing unit* (CPU) circuitry to perform the calculations needed for graphics. As real-time 3D graphics became increasingly common in video game platforms (including arcades, gaming consoles, and personal computers), there was increased demand for specialized hardware for rendering these graphics. This led to the development of the *graphics processing unit* (GPU), a term coined by the Sony Corporation that referred to the circuitry in their PlayStation video game console, released in 1994. The Sony GPU performed graphics-related computational tasks including managing a framebuffer, drawing polygons with textures, and shading and transparency effects. The term *GPU* was popularized by the NVidia Corporation in 1999 with their release of the GeForce 256, a single-chip processor that performed geometric transformations and lighting calculations in addition to the rendering computations performed by earlier hardware implementations. NVidia was the first company to produce a GPU capable of being programmed by developers: each geometric vertex could be processed by a short program, as could every rendered pixel, before the resulting image was displayed on screen. This processor, the GeForce 3, was introduced in 2001 and was also used

in the Xbox video game console. In general, GPUs feature a highly parallel structure that enables them to be more efficient than CPUs for rendering computer graphics. As computer technology advances, so does the quality of the graphics that can be rendered; modern systems are able to produce real-time photorealistic graphics at high resolutions.

Programs that are run by GPUs are called *shaders*, initially so named because they were used for shading effects, but now used to perform many different computations required in the rendering process. Just as there are many high-level programming languages (such as Java, JavaScript, and Python) used to develop CPU-based applications, there are many shader programming languages. Each shader language implements an *application programming interface* (API), which defines a set of commands, functions, and protocols that can be used to interact with an external system—in this case, the GPU. Some APIs and their corresponding shader languages include

- The DirectX API and High-Level Shading Language (HLSL), used on Microsoft platforms, including the Xbox game console
- The Metal API and Metal Shading Language, which runs on modern Mac computers, iPhones, and iPads
- The OpenGL (Open Graphics Library) API and OpenGL Shading Language (GLSL), a cross-platform library.

This book will focus on OpenGL, as it is the most widely adopted graphics API. As a cross-platform library, visual results will be consistent on any supported operating system. Furthermore, OpenGL can be used in concert with a variety of high-level languages using *bindings*: software libraries that bridge two programming languages, enabling functions from one language to be used in another. For example, some bindings to OpenGL include

- JOGL (https://jogamp.org/jogl/www/) for Java
- WebGL (https://www.khronos.org/webgl/) for JavaScript
- PyOpenGL (http://pyopengl.sourceforge.net/) for Python

The initial version of OpenGL was released by Silicon Graphics, Inc. (SGI) in 1992 and has been managed by the Khronos Group since 2006. The Khronos Group is a non-profit technology consortium, whose members

include graphics card manufacturers and general technology companies. New versions of the OpenGL specification are released regularly to support new features and functions. In this book, you will learn about many of the OpenGL functions that allow you to take advantage of the graphics capabilities of the GPU and render some truly impressive three-dimensional scenes. The steps involved in this rendering process are described in detail in the sections that follow.

1.2 THE GRAPHICS PIPELINE

A *graphics pipeline* is an abstract model that describes a sequence of steps needed to render a three-dimensional scene. Pipelining allows a computational task to be split into subtasks, each of which can be worked on in parallel, similar to an assembly line for manufacturing products in a factory, which increases overall efficiency. Graphics pipelines increase the efficiency of the rendering process, enabling images to be displayed at faster rates. Multiple pipeline models are possible; the one described in this section is commonly used for rendering real-time graphics using OpenGL, which consists of four stages (illustrated by Figure 1.7):

- *Application Stage*: initializing the window where rendered graphics will be displayed; sending data to the GPU
- *Geometry Processing*: determining the position of each vertex of the geometric shapes to be rendered, implemented by a program called a *vertex shader*
- *Rasterization*: determining which pixels correspond to the geometric shapes to be rendered
- *Pixel Processing*: determining the color of each pixel in the rendered image, involving a program called a *fragment shader*

Each of these stages is described in more detail in the sections that follow; the next chapter contains code that will begin to implement many of the processes described here.

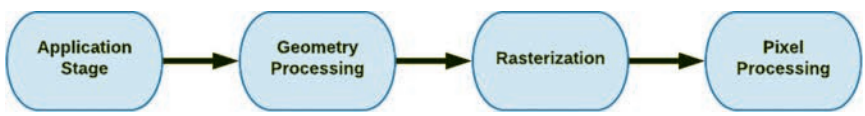

FIGURE 1.7 The graphics pipeline.

1.2.1 Application Stage

The application stage primarily involves processes that run on the CPU. One of the first tasks is to create a window where the rendered graphics will be displayed. When working with OpenGL, this can be accomplished using a variety of programming languages. The window (or a canvas-like object within the window) must be initialized so that the graphics are read from the GPU framebuffer. In the case of animated or interactive applications, the main application contains a loop that re-renders the scene repeatedly, typically aiming for a rate of 60 FPS. Other processes that may be handled by the CPU include monitoring hardware for user input events, or running algorithms for tasks such as physics simulation and collision detection.

Another class of tasks performed by the application includes reading data required for the rendering process and sending it to the GPU. This data may include vertex attributes (which describe the appearance of the geometric shapes being rendered), images that will be applied to surfaces, and source code for the vertex shader and fragment shader programs (which will be used later on during the graphics pipeline). OpenGL describes the functions that can be used to transmit this data to the GPU; these functions are accessed through the bindings of the programming language used to write the application. Vertex attribute data is stored in GPU memory buffers called *vertex buffer objects* (VBOs), while images that will be used as textures are stored in *texture buffers*. It is important to note that this stored data is not initially assigned to any particular program variables; these associations are specified later. Finally, source code for the vertex shader and fragment shader programs needs to be sent to the GPU, compiled, and loaded. If needed, buffer data can be updated during the application's main loop, and additional data can be sent to shader programs as well.

Once the necessary data has been sent to the GPU, before rendering can take place, the application needs to specify the associations between attribute data stored in VBOs and attribute variables in the vertex shader program. A single geometric shape may have multiple attributes for each vertex (such as position and color), and the corresponding data is streamed from buffers to variables in the shader during the rendering process. It is also frequently necessary to work with many sets of such associations: there may be multiple geometric shapes (with data stored in different buffers) that are rendered by the same shader program, or each shape may be rendered by a different shader program. These sets of associations can be

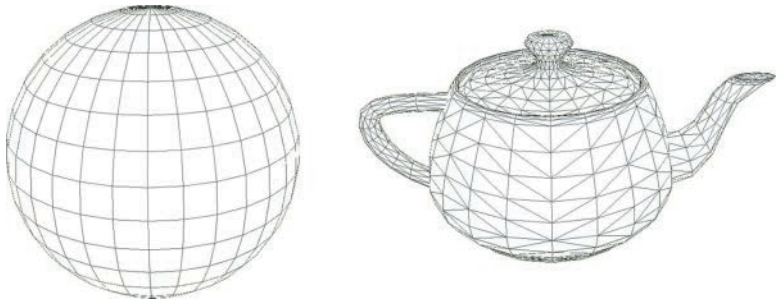

FIGURE 1.8 Wireframe meshes representing a sphere and a teapot.

conveniently managed by using *vertex array objects* (VAOs), which store this information and can be activated and deactivated as needed during the rendering process.

1.2.2 Geometry Processing

In computer graphics, the shape of a geometric object is defined by a *mesh*: a collection of points that are grouped into lines or triangles, as illustrated in Figure 1.8.

In addition to the overall shape of an object, additional information may be required to describe how the object should be rendered. The properties or attributes that are specific to rendering each individual point are grouped together into a data structure called a *vertex*. At a minimum, a vertex must contain the three-dimensional position of the corresponding point. Additional data contained by a vertex often includes

- a color to be used when rendering the point
- *texture coordinates* (or UV coordinates), which indicate a point in an image that is mapped to the vertex
- a *normal vector*, which indicates the direction perpendicular to a surface and is typically used in lighting calculations

Figure 1.9 illustrates different renderings of a sphere that make use of these attributes. Additional vertex attributes may be defined as needed.

During the geometry processing stage, the vertex shader is applied to each of the vertices; each attribute variable in the shader receives data from a buffer according to previously specified associations. The primary purpose of the vertex shader is to determine the final position of

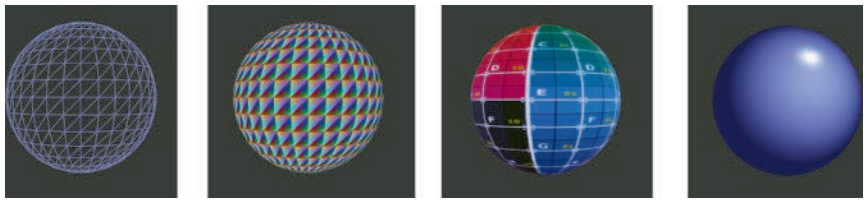

FIGURE 1.9 Different renderings of a sphere: wireframe, vertex colors, texture, and with lighting effects.

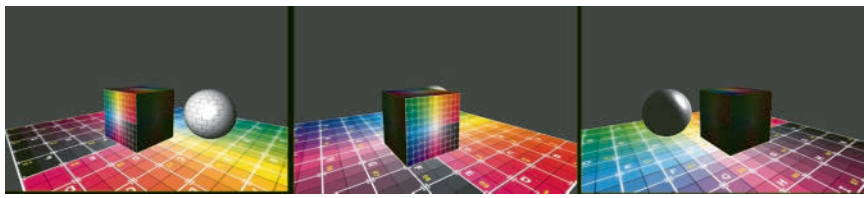

FIGURE 1.10 One scene rendered from multiple camera locations and angles.

each point being rendered, which is typically calculated from a series of transformations:

- the collection of points defining the intrinsic shape of an object may be translated, rotated, and scaled so that the object appears to have a particular location, orientation, and size with respect to a virtual three-dimensional world. This process is called the *model transformation*; coordinates expressed from this frame of reference are said to be in world space
- there may be a virtual camera with its own position and orientation in the virtual world. In order to render the world from the camera's point of view, the coordinates of each object in the world must be converted to a frame of reference relative to the camera itself. This process is called the *view transformation*, and coordinates in this context are said to be in *view space* (or *camera space*, or *eye space*). The effect of the placement of the virtual camera on the rendered image is illustrated in Figure 1.10
- the set of points in the world considered to be visible, occupying either a box-shaped or frustum-shaped region, must be scaled to and aligned with the space rendered by OpenGL: a cube-shaped region consisting of all points whose coordinates are between –1 and 1.

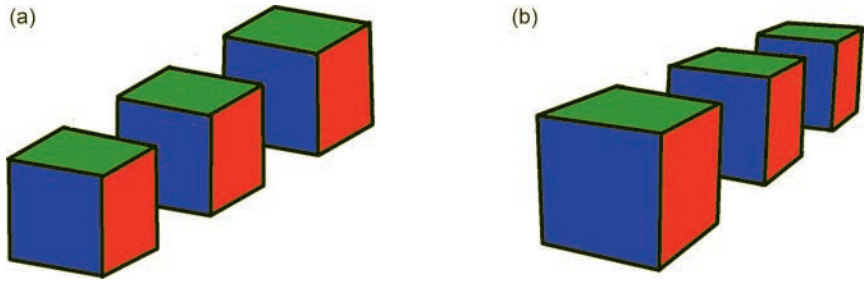

FIGURE 1.11 A series of cubes rendered with orthogonal projection (a) and perspective projection (b).

> The position of each point returned by the vertex shader is assumed to be expressed in this frame of reference. Any points outside this region are automatically discarded or *clipped* from the scene; coordinates expressed at this stage are said to be in *clip space*. This task is accomplished with a *projection transformation*. More specifically, it is called an *orthographic projection* or a *perspective projection*, depending on whether the shape of the visible world region is a box or a frustum. A perspective projection is generally considered to produce more realistic images, as objects that are farther away from the virtual camera will require greater compression by the transformation and thus appear smaller when the scene is rendered. The differences between the two types of projections are illustrated in Figure 1.11.

In addition to these transformation calculations, the vertex shader may perform additional calculations and send additional information to the fragment shader as needed.

1.2.3 Rasterization

Once the final positions of each vertex have been specified by the vertex shader, the rasterization stage begins. The points themselves must first be grouped into the desired type of *geometric primitive*: points, lines, or triangles, which consist of sets of 1, 2, or 3 points. In the case of lines or triangles, additional information must be specified. For example, consider an array of points [A, B, C, D, E, F] to be grouped into lines. They could be grouped in disjoint pairs, as in (A, B), (C, D), (E, F), resulting in a set of disconnected line segments. Alternatively, they could be grouped in overlapping pairs, as in (A, B), (B, C), (C, D), (D, E), (E, F), resulting in a set of connected line segments (called a line strip). The type of geometric

primitive and method for grouping points is specified using an OpenGL function parameter when the rendering process begins. The process of grouping points into geometric primitives is called *primitive assembly*.

Once the geometric primitives have been assembled, the next step is to determine which pixels correspond to the interior of each geometric primitive. Since pixels are discrete units, they will typically only approximate the continuous nature of a geometric shape, and a criterion must be given to clarify which pixels are in the interior. Three simple criteria could be

1. the entire pixel area is contained within the shape
2. the center point of the pixel is contained within the shape
3. any part of the pixel is contained within the shape

These effects of applying each of these criteria to a triangle are illustrated in Figure 1.12, where the original triangle appears outlined in blue, and pixels meeting the criteria are shaded gray.

For each pixel corresponding to the interior of a shape, a *fragment* is created: a collection of data used to determine the color of a single pixel in a rendered image. The data stored in a fragment always includes the *raster position*, also called *pixel coordinates*. When rendering a three-dimensional scene, fragments will also store a *depth* value, which is needed when points on different geometric objects would overlap from the perspective of the viewer. When this happens, the associated fragments would correspond to the same pixel, and the depth value determines which fragment's data should be used when rendering this pixel.

Additional data may be assigned to each vertex, such as a color, and passed along from the vertex shader to the fragment shader. In this case, a new data field is added to each fragment. The value assigned to this field at

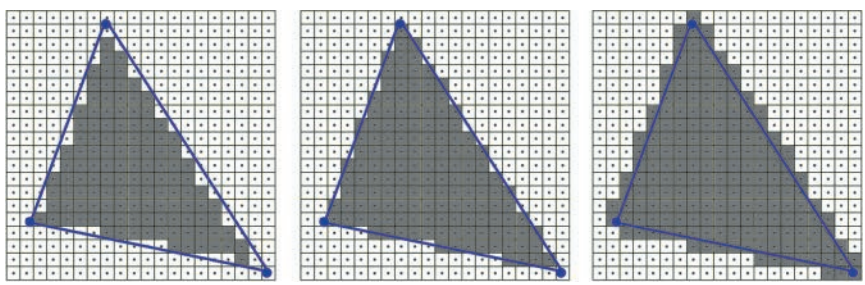

FIGURE 1.12 Different criteria for rasterizing a triangle.

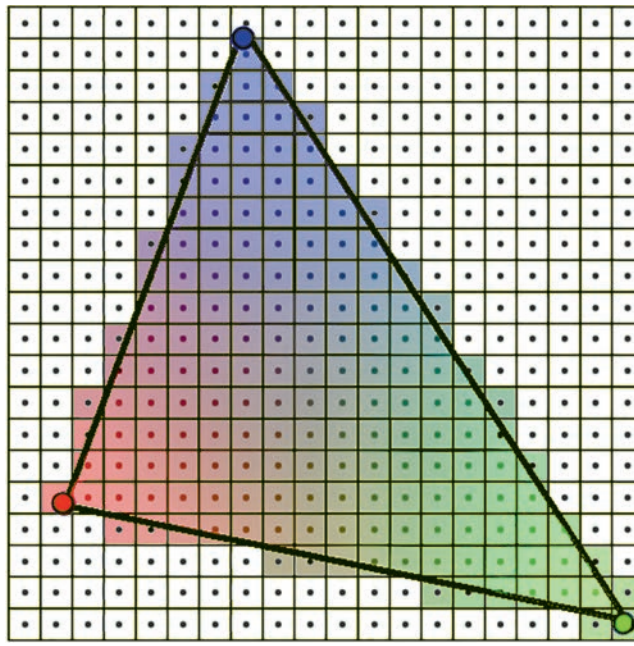

FIGURE 1.13 Interpolating color attributes.

each interior point is *interpolated* from the values at the vertices: calculated using a weighted average, depending on the distance from the interior point to each vertex. The closer an interior point is to a vertex, the greater the weight of that vertex's value when calculating the interpolated value. For example, if the vertices of a triangle are assigned the colors red, green, and blue, then each pixel corresponding to the interior of the triangle will be assigned a combination of these colors, as illustrated in Figure 1.13.

1.2.4 Pixel Processing

The primary purpose of this stage is to determine the final color of each pixel, storing this data in the color buffer within the framebuffer. During the first part of the pixel processing stage, a program called the fragment shader is applied to each of the fragments to calculate their final color. This calculation may involve a variety of data stored in each fragment, in combination with data globally available during rendering, such as

- a base color applied to the entire shape
- colors stored in each fragment (interpolated from vertex colors)

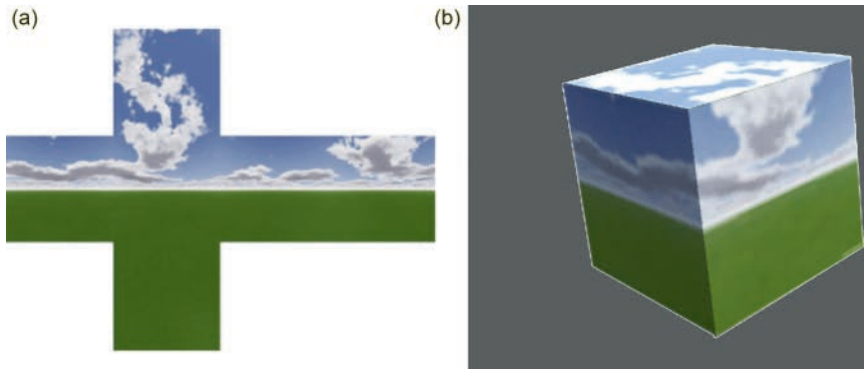

FIGURE 1.14 An image file (a) used as a texture for a 3D object (b).

- textures (images applied to the surface of the shape, illustrated by Figure 1.14), where colors are sampled from locations specified by texture coordinates
- light sources, whose relative position and/or orientation may lighten or darken the color, depending on the direction the surface is facing at a point, specified by normal vectors

Some aspects of the pixel processing stage are automatically handled by the GPU. For example, the depth values stored in each fragment are used in this stage to resolve visibility issues in a three-dimensional scene, determining which parts of objects are blocked from view by other objects. After the color of a fragment has been calculated, the fragment's depth value will be compared to the value currently stored in the depth buffer at the corresponding pixel coordinates. If the fragment's depth value is smaller than the depth buffer value, then the corresponding point is closer to the viewer than any that were previously processed, and the fragment's color will be used to overwrite the data currently stored in the color buffer at the corresponding pixel coordinates.

Transparency is also handled by the GPU, using the alpha values stored in the color of each fragment. The alpha value of a color is used to indicate how much of this color should be blended with another color. For example, when combining a color C1 with an alpha value of 0.6 with another color C2, the resulting color will be created by adding 60% of the value from each component of C1 to 40% of the value from each component of C2. Figure 1.15 illustrates a simple scene involving transparency.

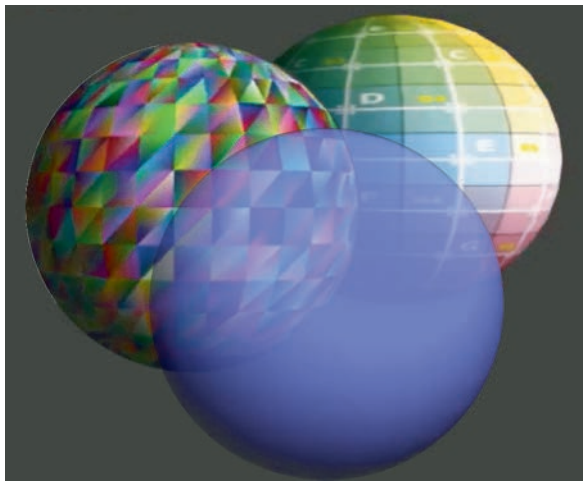

FIGURE 1.15 Rendered scene with transparency.

However, rendering transparent objects has some complex subtleties. These calculations occur at the same time that depth values are being resolved, and so scenes involving transparency must render objects in a particular order: all opaque objects must be rendered first (in any order), followed by transparent objects ordered from farthest to closest with respect to the virtual camera. Not following this order may cause transparency effects to fail. For example, consider a scene, such as that in Figure 1.15, containing a single transparent object close to the camera and multiple opaque objects farther from the camera that appear behind the transparent object. Assume that, contrary to the previously described rendering order, the transparent object is rendered first, followed by the opaque objects in some unknown order. When the fragments of the opaque objects are processed, their depth value will be greater than the value stored in the depth buffer (corresponding to the closer transparent object), and so the opaque fragments' color data will automatically be discarded, rather than blended with the currently stored color. Even attempting to use the alpha value of the transparent object stored in the color buffer in this example does not resolve the underlying issue, because when the fragments of each opaque object are being rendered, it is not possible at this point to determine if they may have been occluded from view by another opaque fragment (only the closest depth value, corresponding to the transparent object, is stored), and thus, it is unknown which opaque fragment's color values should be blended into the color buffer.

1.3 SETTING UP A DEVELOPMENT ENVIRONMENT

Most parts of the graphics pipeline discussed in the previous section—geometry processing, rasterization, and pixel processing—are handled by the GPU, and as mentioned previously, this book will use OpenGL for these tasks. For developing the application, there are many programming languages one could select from. In this book, you will be using Python to develop these applications, as well as a complete graphics framework to simplify the design and creation of interactive, animated, three-dimensional scenes.

1.3.1 Installing Python

To prepare your development environment, the first step is to download and install a recent version of Python (version 3.8 as of this writing) from http://www.python.org (Figure 1.16); installers are available for Windows, Mac OS X, and a variety of other platforms.

- When installing for Windows, check the box next to ***add to path***. Also, select the options ***custom installation*** and ***install for all users***; this simplifies setting up alternative development environments later.

The Python installer will also install IDLE, Python's Integrated Development and Learning Environment, which can be used for developing the graphics framework presented throughout this book. A more

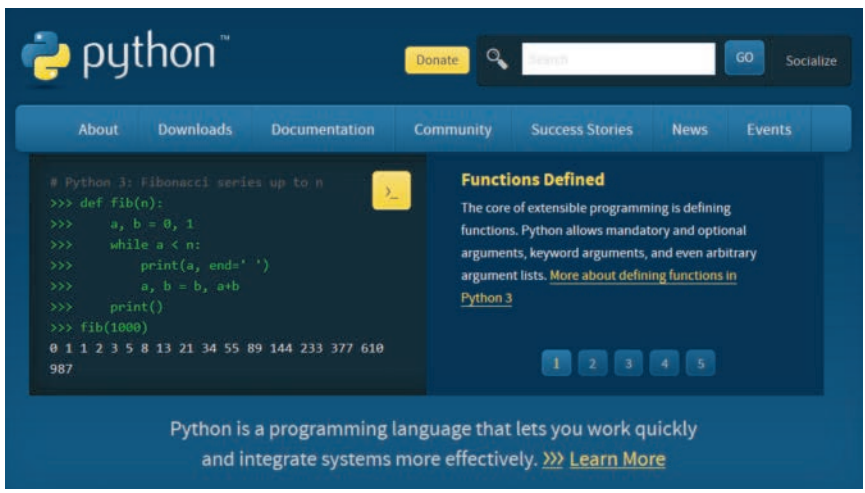

FIGURE 1.16 Python homepage: http://www.python.org.

sophisticated alternative is recommended, such as Sublime Text, which will be introduced later on in this chapter, and some of its advantages discussed. (If you are already familiar with an alternative Python development environment, you are of course also welcome to use that instead.)

IDLE has two main window types. The first window type, which automatically opens when you run IDLE, is the shell window, an interactive window that allows you to write Python code which is then immediately executed after pressing the Enter key. Figure 1.17 illustrates this window after entering the statements `123 + 214` and `print("Hello, world!")`. The second window type is the *editor window*, which functions as a text editor, allowing you to open, write, and save files containing Python code, which are called *modules* and typically use the.py file extension. An editor window can be opened from the shell window by selecting either ***File>New File*** or ***File>Open...*** from the menu bar. Programs may be run from the editor window by choosing ***Run>Run Module*** from the menu bar; this will display the output in a shell window (opening a new shell window if none are open). Figure 1.18 illustrates creating a file in the editor window containing the following code:

```
print("Hello, world!")
print("Have a nice day!")
```

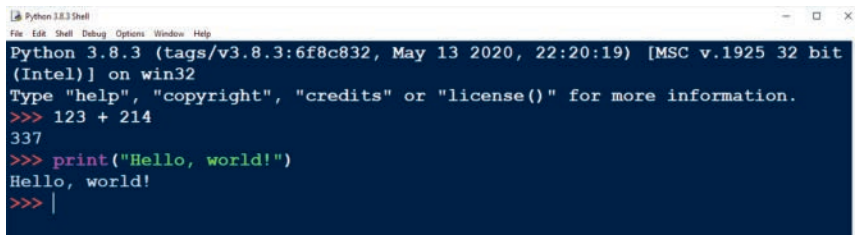

FIGURE 1.17 IDLE shell window.

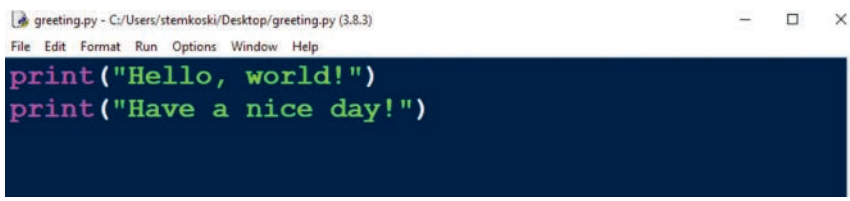

FIGURE 1.18 IDLE editor window.

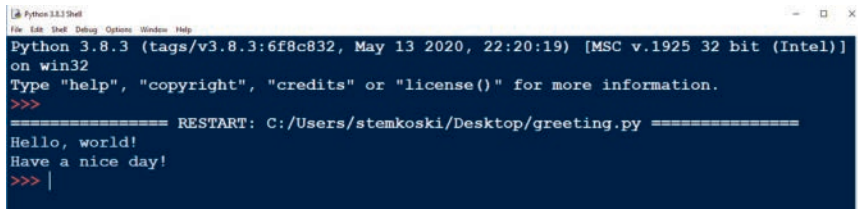

FIGURE 1.19 Results of running the Python program from Figure 1.18.

Figure 1.19 illustrates the results of running this code, which appear in a shell window.

1.3.2 Python Packages

Once Python has been successfully installed, your next step will be to install some *packages*, which are collections of related modules that provide additional functionality to Python. The easiest way to do this is by using *pip*, a software tool for package installation in Python. In particular, you will install

- Pygame (http://www.pygame.org), a package that can be used to easily create windows and handle user input
- Numpy (https://numpy.org/), a package for mathematics and scientific computing
- PyOpenGL and PyOpenGL_accelerate (http://pyopengl.sourceforge.net/), which provide a set of bindings from Python to OpenGL.

If you are using Windows, open Command Prompt or PowerShell (run with administrator privileges so that the packages are automatically available to all users) and enter the following command, which will install all of the packages described above:

```
py -m pip install pygame numpy PyOpenGL
PyOpenGL_accelerate
```

If you are using MacOS, the command is slightly different. Enter

```
python3-m pip install pygame numpy PyOpenGL
PyOpenGL_accelerate
```

To verify that these packages have been installed correctly, open a new IDLE shell window (restart IDLE if it was open before installation). To check Pygame, enter the following code, and press the Enter key:

```
import pygame
```

You should see a message that contains the number of the Pygame version that has been installed, such as "`pygame 1.9.6`", and a greeting message such as "`Hello from the pygame community`". If instead you see a message that contains the text `No module named 'pygame'`, then Pygame has not been correctly installed. Furthermore, it will be important to install a recent version of Pygame—at least a development version of Pygame 2.0.0. If an earlier version has been installed, return to the command prompt and in the install command above, change `pygame` to `pygame==2.0.0.dev10` to install a more recent version.

Similarly, to check the Numpy installation, instead use the code:

```
import numpy
```

In this case, if you see no message at all (just another input prompt), then the installation was successful. If you see a message that contains the text `No module named 'numpy'`, then Numpy has not been correctly installed. Finally, to check PyOpenGL, instead use the code:

```
import OpenGL
```

As was the case with testing the Numpy package, if there is no message displayed, then the installation was successful, but a message mentioning that the module is not installed will require you to try re-installing the package.

If you encounter difficulties installing any of these packages, there is additional help available online:

- Pygame: https://www.pygame.org/wiki/GettingStarted
- Numpy: https://numpy.org/install/
- PyOpenGL: at http://pyopengl.sourceforge.net/documentation/installation.html

1.3.3 Sublime Text

When working on a large project involving multiple files, you may want to install an alternative development environment, rather than restrict yourself to working with IDLE. The authors particularly recommend Sublime Text, which has the following advantages:

- lines are numbered for easy reference
- tabbed interface for working with multiple files in a single window
- editor supports multi-column layout to view and edit different files simultaneously
- directory view to easily navigate between project files in a project
- able to run scripts and display output in console area
- free, full-featured trial version available

To install the application, visit the Sublime Text website (https://www.sublimetext.com/), shown in Figure 1.20, and click on the "download" button (whose text may differ from the figure to reference the operating system you are using). Alternatively, you may click the download link in the navigation bar to view all downloadable versions. After downloading, you will need to run the installation program, which will require administrator-level privileges on your system. If unavailable, you may alternatively download a "portable version" of the software, which can

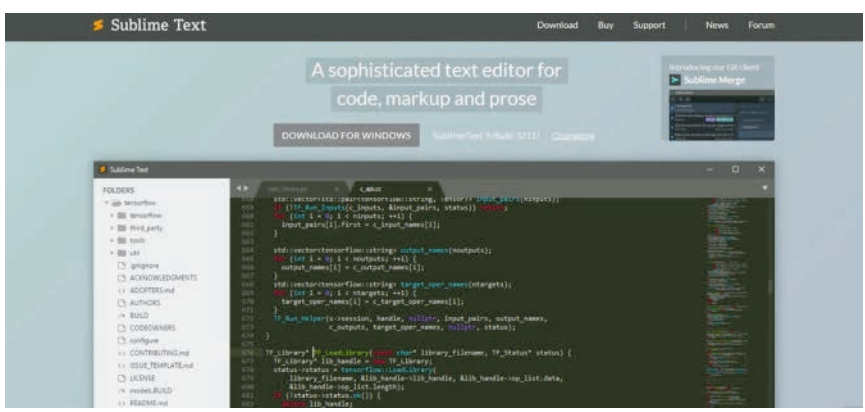

FIGURE 1.20 Sublime Text homepage

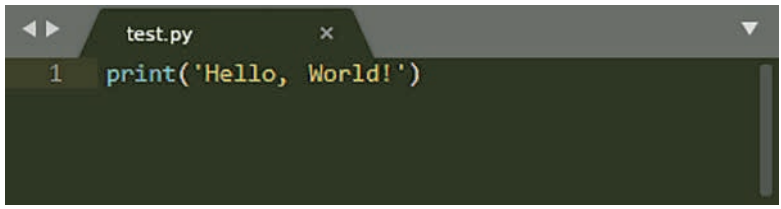

FIGURE 1.21 Sublime Text editor window.

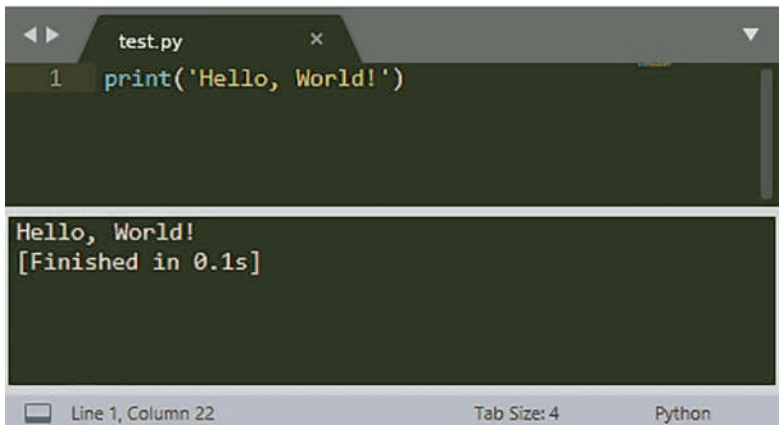

FIGURE 1.22 Output from Figure 1.21.

be found via the download link previously mentioned. While a free trial version is available, if you choose to use this software extensively, you are encouraged to purchase a license.

After installation, start the Sublime Text software. A new editor window will appear, containing an empty file. As previously mentioned, Sublime Text can be used to run Python scripts automatically, provided that Python has been installed for all users of your computer and it is included on the system path. To try out this feature, in the editor window, as shown in Figure 1.21, enter the text:

```
print("Hello, world!")
```

Next, save your file with the name `test.py`; the.`py` extension causes Sublime Text to recognize it as a Python script file, and syntax highlighting will be applied. Finally, from the menu bar, select `Tools > Build` or press the keyboard key combination `Ctrl + B` to build and run the application. The output will appear in the console area, as illustrated in Figure 1.22.

1.4 SUMMARY AND NEXT STEPS

In this chapter, you learned about the core concepts and vocabulary used in computer graphics, including rendering, buffers, GPUs, and shaders. Then, you learned about the four major stages in the graphics pipeline: the application stage, geometry processing, rasterization, and pixel processing; this section introduced additional terminology, including vertices, VBOs, VAOs, transformations, projections, fragments, and interpolation. Finally, you learned how to set up a Python development environment. In the next chapter, you will use Python to start implementing the graphics framework that will realize these theoretical principles.

CHAPTER 2

Introduction to Pygame and OpenGL

In this chapter, you will learn how to create windows with Pygame and how to draw graphics in these windows with OpenGL. You will start by rendering a point, followed by lines and triangles with a single color. Then, you will draw multiple shapes with multiple colors, create a series of animations involving movement and color transitions, and implement interactive applications with keyboard controlled movement.

2.1 CREATING WINDOWS WITH PYGAME

As indicated in the discussion of the graphics pipeline, the first step in rendering graphics is to develop an application where graphics will be displayed. This can be accomplished with a variety of programming languages; throughout this book, you will write windowed applications using Python and Pygame, a popular Python game development library.

As you write code, it is important to keep a number of software engineering principles in mind, including organization, reusability, and extensibility. To support these principles, the software developed in this book uses an object-oriented design approach. To begin, create a main folder where you will store your source code. Within this folder, you will store the main applications as well as your own packages: folders containing collections of related modules, which in this case will be Python files containing class definitions.

DOI: 10.1201/9781003181378-2

First, you will create a class called **`Base`** that initializes Pygame and displays a window. Anticipating that the applications created will eventually feature user interaction and animation, this class will be designed to handle the standard phases or "life cycle" of such an application:

- *Startup*: During this stage, objects are created, values are initialized, and any required external files are loaded.
- *The Main Loop*: This stage repeats continuously (typically 60 times per second), while the application is running and consists of the following three substages:
  - *Process Input*: Check if the user has performed any action that sends data to the computer, such as pressing keys on a keyboard or clicking buttons on a mouse.
  - *Update*: Changing values of variables and objects.
  - *Render*: Create graphics that are displayed on the screen.
- *Shutdown*: This stage typically begins when the user performs an action indicating that the program should stop running (for example, by clicking a button to quit the application). This stage may involve tasks such as signaling the application to stop checking for user input and closing any windows that were created by the application.

These phases are illustrated by the flowchart in Figure 2.1.

The **`Base`** class will be designed to be extended by the various applications throughout this book. In accordance with the principle of modularization, processing user input will be handled by a separate class named **`Input`** that you will create later.

To begin, in your main folder, create a new folder called **`core`**. For Python to recognize this (or any) folder as a package, within the folder, you need

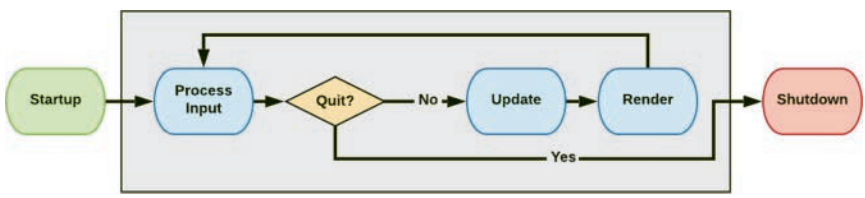

FIGURE 2.1 The phases of an interactive graphics-based application.

to create a new file named **__init__.py** (note the double underscore characters that occur before and after **init**). Any code in the **__init__.py** file will be run when modules from this package are imported into another program; leave this as an empty file. Next, also in the **core** folder, create a new file named **base.py**, and enter the following code (which contains some basic comments that will be explained more fully after):

```
import pygame
import sys

class Base(object):

    def __init__(self, screenSize=[512, 512]):

        # initialize all pygame modules
        pygame.init()
        # indicate rendering details
        displayFlags = pygame.DOUBLEBUF | pygame.
            OPENGL
        # initialize buffers to perform antialiasing
        pygame.display.gl_set_attribute(
            pygame.GL_MULTISAMPLEBUFFERS, 1)
        pygame.display.gl_set_attribute(
            pygame.GL_MULTISAMPLESAMPLES, 4)
        # use a core OpenGL profile for cross-platform
            compatibility
        pygame.display.gl_set_attribute(
            pygame.GL_CONTEXT_PROFILE_MASK,
            pygame.GL_CONTEXT_PROFILE_CORE)
        # create and display the window
        self.screen = pygame.display.set_mode(
            screenSize, displayFlags )
        # set the text that appears in the title bar
            of the window
        pygame.display.set_caption("Graphics Window")

        # determine if main loop is active
        self.running = True
        # manage time-related data and operations
        self.clock = pygame.time.Clock()

    # implement by extending class
```

```
    def initialize(self):
        pass

    # implement by extending class
    def update(self):
        pass

    def run(self):

        ## startup ##
        self.initialize()

        ## main loop ##
        while self.running:

            ## process input ##

            ## update ##
            self.update()

            ## render ##
            # display image on screen
            pygame.display.flip()

            # pause if necessary to achieve 60 FPS
            self.clock.tick(60)

        ## shutdown ##
        pygame.quit()
        sys.exit()
```

In addition to the comments throughout the code above, the following observations are noteworthy:

- The **`screenSize`** parameter can be changed as desired. At present, if the screen size is set to non-square dimensions, this will cause the rendered image to appear stretched along one direction. This issue will be addressed in Chapter 4 when discussing aspect ratios.
- The title of the window is set with the function **`pygame.display.set _ caption`** and can be changed as desired.

- The **displayFlags** variable is used to combine constants representing different display settings with the bitwise or operator '|'. Additional settings (such as allowing a resizable window) are described at https://www.pygame.org/docs/ref/display.html.

- The **pygame.DOUBLEBUF** constant indicates that a rendering technique called *double buffering* will be used, which employs two image buffers. The pixel data from one buffer is displayed on screen while new data is being written into a second buffer. When the new image is ready, the application switches which buffer is displayed on screen and which buffer will be written to; this is accomplished in Pygame with the statement **pygame.display.flip()**. The double buffering technique eliminates an unwanted visual artifact called *screen tearing*, in which the pixels from a partially updated buffer are displayed on screen, which happens when a single buffer is used and the rendering cycles are not synchronized with the display refresh rate.

- *Antialiasing* is a rendering technique used to remove the appearance of jagged, pixelated lines along edges of polygons in a rasterized image. The two lines of code beneath the antialiasing comment indicate that each pixel at the edge of a polygon will be sampled multiple times, and in each sample, a slight offset (smaller than the size of a pixel) is applied to all screen coordinates. The color samples are averaged, resulting in a smoother transition between pixel colors along polygon edges.

- Starting in OpenGL version 3.2 (introduced in 2009), *deprecation* was introduced: older functions were gradually replaced by more efficient versions, and future versions may no longer contain or support the older functions. This led to *core* and *compatibility profiles*: core profiles are only guaranteed to implement functions present in the current version of the API, while compatibility profiles will additionally support many functions that may have been deprecated. Each hardware vendor decides which versions of OpenGL will be supported by each profile. In recent versions of Mac OS X (10.7 and later) at the time of writing, the core profile supported is 3.2, while the compatibility profile supported is 2.1. Since some of the OpenGL features (such as vertex array objects or VAOs) that will be needed in constructing the graphics framework in this book were introduced in GLSL version 3.0, a core profile is specified for maximum

cross-platform compatibility. The corresponding line of code also requires at least Pygame version 2.0 to run.

- The function **`pygame.display.set _ mode`** sets the properties of the window and also makes the window appear on screen.
- The Clock object (initialized with **`pygame.time.Clock()`**) has many uses, such as keeping track of how much time has passed since a previous function call, or how many times a loop has run during the past second. Each iteration of the main loop results in an image being displayed, which can be considered a frame of an animation. Since the speed at which these images appear is the speed at which the main loop runs, both are measured in terms of frames per second (FPS). By default, the main loop will run as fast as possible—sometimes faster than 60 FPS, in which case the program may attempt to use nearly 100% of the CPU. Since most computer displays only update 60 times per second, there is no need to run faster than this, and the **`tick`** function called at the end of the main loop results in a short pause at the end of the loop that will bring the execution speed down to 60 FPS.
- The **`initialize`** and **`update`** functions are meant to be implemented by the applications that extend this class. Since every function must contain at least one statement, the **`pass`** statement is used here, which is a null statement—it does nothing.
- The **`run`** function contains all the phases of an interactive graphics-based application, as described previously; the corresponding code is indicated by comments beginning with **`##`**.

The next task we need to address is basic input processing; at a minimum, the user needs to be able to terminate the program, which will set the variable **`self.running`** to **`False`** in the code above. To this end, in the **`core`** folder, create a new file named **`input.py`** containing the following code:

```
import pygame

class Input(object):

    def __init__(self):
```

```
        # has the user quit the application?
        self.quit = False

    def update(self):
        # iterate over all user input events (such as
            keyboard or
        # mouse)that occurred since the last time
            events were checked
        # for event in pygame.event.get():
        # quit event occurs by clicking button to
             close window
            if event.type == pygame.QUIT:
                self.quit = True
```

At present, the **Input** class only monitors for quit-type events; in later sections, keyboard functionality will be added as well. For now, return to the **Base** class. After the import statements, add the code:

```
from core.input import Input
```

This will enable you to use the **Input** class from the **input** module in the **core** package. It should be noted that the import statements are written assuming that your application files (which will extend the **Base** class) will be stored in the main directory (which contains all the packages).

Next, at the end of the **init** function, add the code:

```
# manage user input
self.input = Input()
```

This will create and store an instance of the **Input** class when the **Base** class is created.

Finally, in the **run** function, after the comment **## process input ##**, add the code:

```
self.input.update()
if self.input.quit:
    self.running = False
```

This will enable the user to stop the application, as described prior to the code listing for the **Input** class.

You will next write an application that uses these classes to create a window. The general approach in this and similar applications will to extend the **Base** class, implement the **initialize** and **update** functions, and then, create an instance of the new class and call the **run** function. To proceed, in your main folder, create a new file named **test-2-1.py** with the code:

```
from core.base import Base

class Test(Base):

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

    def update(self):
        pass

# instantiate this class and run the program
Test().run()
```

In this program, a message is printed during initialization for illustrative purposes. However, no print statements are present in the **update** function, as attempting to print text 60 times per second would cause extreme slowdown in any program. Run this program, and you should see a blank window appear on screen (as illustrated in Figure 2.2) and the text "**Initializing program...**" will appear in the shell. When you click on the button to close the window, the window should close, as expected.

2.2 DRAWING A POINT

Now that you are able to create a windowed application, the next goal is to render a single point on the screen. You will accomplish this by writing the simplest possible vertex shader and fragment shader, using OpenGL Shading Language. You will then learn how to compile and link the shaders to create a graphics processing unit (GPU) program. Finally, you will extend the framework begun in the previous section to use GPU programs to display graphics in the Pygame application window.

2.2.1 OpenGL Shading Language

OpenGL Shading Language (GLSL) is a C-style language, and is both similar to and different from Python in a number of ways. Similar to

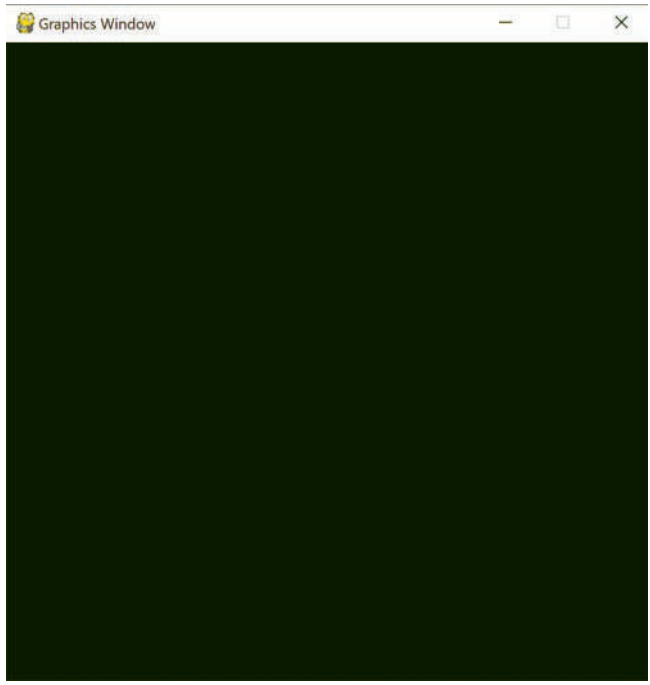

FIGURE 2.2 The Pygame window.

Python, there are "if statements" to process conditional statements, "for loops" to iterate a group of statements over a range of values, "while loops" to iterate a group of statements as long as a given condition is true, and functions that take a set of inputs and perform some computations (and optionally return an output value). Unlike Python, variables in GLSL must be declared with an assigned type, the end of each statement must be indicated with a semicolon, statements are grouped using braces (as opposed to indentation), comments are preceded by "**//**" (rather than "**#**"), and functions must specify the types of their input parameters and return value. The details of the differences in Python and GLSL syntax will be illustrated and indicated as the features are introduced in the examples throughout this book.

The basic data types in GLSL are boolean, integer, and floating point values, indicated by **`bool`**, **`int`**, and **`float`**, respectively. GLSL has vector data types, which are often used to store values indicating positions, colors, and texture coordinates. Vectors may have two, three, or four components, indicated by **`vec2`**, **`vec3`**, and **`vec4`** (for vectors consisting of floats). As a C-like language, GLSL provides arrays and *structs*: user-defined data

types. To facilitate graphics-related computations, GLSL also features matrix data types, which often store transformations (translation, rotation, scaling, and projections), and sampler data types, which represent textures; these will be introduced in later chapters.

The components of a vector data type can be accessed in multiple ways. For example, once a **`vec4`** named **`v`** has been initialized, its components can be accessed using array notation (**`v[0]`**, **`v[1]`**, **`v[2]`**, **`v[3]`**), or using dot notation with any of the following three systems: (**`v.x`**, **`v.y`**, **`v.z`**, **`v.w`**) or (**`v.r`**, **`v.g`**, **`v.b`**, **`v.a`**) or (**`v.s`**, **`v.t`**, **`v.p`**, **`v.q`**). While all these systems are interchangeable, programmers typically choose a system related to the context for increased readability: (x, y, z, w) are used for positions, (r, g, b, a) are used for colors (red, green, blue, alpha), and (s, t, p, q) are used for texture coordinates.

Every shader must contain a function named **`main`**, similar to the C programming language. No values are returned (which is indicated by the keyword **`void`**), and there are no parameters required by the main function; thus, every shader has the general following structure:

```
void main()
{
    // code statements here
}
```

In the description of the graphics pipeline from the previous chapter, it was mentioned that a vertex shader will receive data about the geometric shapes being rendered via buffers. At this point, you may be wondering how vertex attribute data is sent from buffers to a vertex shader if the **`main`** function takes no parameters. In general, data is passed in and out of shaders via variables that are declared with certain *type qualifiers*: additional keywords that modify the properties of a variable. For example, many programming languages have a qualifier to indicate that the value of a variable will remain constant; in GLSL, this is indicated by the keyword **`const`**. Additionally, when working with shaders, the keyword **`in`** indicates that the value of a variable will be supplied by the previous stage of the graphics pipeline, while the keyword **`out`** indicates that a value will be passed along to the next stage of the graphics pipeline. More specifically, in the context of a vertex shader, **`in`** indicates that values will be supplied from a buffer, while **`out`** indicates that values will be passed to the fragment shader. In the context of a fragment shader, **`in`** indicates that values

will be supplied from the vertex shader (interpolated during the rasterization stage), while **out** indicates values will be stored in one of the various buffers (color, depth, or stencil).

There are two particular **out** variables that are required when writing shader code for a GPU program. First, recall that the ultimate goal of the vertex shader is to calculate the position of a point. OpenGL uses the built-in variable **gl _ Position** to store this value; a value must be assigned to this variable by the vertex shader. Second, recall that the ultimate goal of the fragment shader is to calculate the color of a pixel. Early versions of OpenGL used a built-in variable called **gl _ FragColor** to store this value, and each fragment shader was required to assign a value to this variable. Later versions require fragment shader code to explicitly declare an **out** variable for this purpose. Finally, it should be mentioned that both of these variables are **vec4** type variables. For storing color data, this makes sense as red, green, blue, and alpha (transparency) values are required. For storing position data, this is less intuitive, as a position in three-dimensional space can be specified using only *x*, *y*, and *z* coordinates. By including a fourth coordinate (commonly called *w* and set to the value 1), this makes it possible for geometric transformations (such as translation, rotation, scaling, and projection) to be represented by and calculated using a single matrix, which will be discussed in detail in Chapter 3.

As indicated at the beginning of this section, the current goal is to write a vertex shader and a fragment shader that will render a single point on the screen. The code presented will avoid the use of buffers and exclusively use built-in variables. (You do not need to create any new files or enter any code at this time.) The vertex shader will consist of the following code:

```
void main()
{
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
}
```

In early versions of OpenGL, the simplest possible fragment shader could have consisted of the following code:

```
void main()
{
    gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0);
}
```

For more modern versions of OpenGL, where you need to declare a variable for the output color, you can use the following code for the fragment shader:

```
out vec4 fragColor;
void main()
{
    fragColor = vec4(1.0, 1.0, 0.0, 1.0);
}
```

Taken together, the vertex shader and the fragment shader produce a program that renders a point in the center of the screen, colored yellow. If desired, these values can be altered within certain bounds. The *x*, *y*, and *z* components of the position vector may be changed to any value between –1.0 and 1.0, and the point will remain visible; any values outside this range place the point outside of the region rendered by OpenGL and will result in an empty image being rendered. Changing the *z* coordinate (within this range) will have no visible effect at this time, since no perspective transformations are being applied. Similarly, the *r*, *g*, and *b* components of the color vector may be changed as desired, although dark colors may be difficult to distinguish on the default black background color. It should also be noted that the number types **`int`** and **`float`** are not interchangeable; entering just 1 rather than 1.0 may cause shader compilation errors.

2.2.2 Compiling GPU Programs

Now that you've learned the basics about writing shader code, the next step is to learn how to compile and link the vertex and fragment shaders to create a GPU program. To continue with the goal of creating a reusable framework, you will create a utility class that will perform these tasks. In this section and those that follow, many of the functions from PyOpenGL will be introduced and described in the following style:

> **functionName**(*parameter1* , *parameter2* , …)
>
> Description of function and parameters.

Many of these functions will have syntax identical to that presented in the official OpenGL reference pages maintained by the Khronos Group at https://www.khronos.org/registry/OpenGL-Refpages/. However, there will be a few differences, because the OpenGL Shading Language (GLSL)

is a C-style language, and PyOpenGL is a Python binding. In particular, arrays are handled differently in these two programming languages, which is often reflected in the Python functions requiring fewer arguments.

The first step towards compiling GPU programs centers on the individual shaders. Shader objects must be created to store shader source code, the source code must be sent to these objects, and then the shaders must be compiled. This is accomplished using the following three functions:

glCreateShader(*shaderType*)

Creates an empty shader object, which is used to store the source code of a shader, and returns a value by which it can be referenced. The type of shader (such as a vertex shader or a fragment shader) is specified with the *shaderType* parameter, whose value will be an OpenGL constant such as GL_VERTEX_SHADER or GL_FRAGMENT_SHADER.

glShaderSource(*shaderRef*, *shaderCode*)

Stores the source code in the string parameter *shaderCode* in the shader object referenced by the parameter *shaderRef*.

glCompileShader(*shaderRef*)

Compiles the source code stored in the shader object referenced by the parameter *shaderRef*.

Since mistakes may be made when writing shader code, compiling a shader may or may not succeed. Unlike application compilation errors, which are typically automatically displayed to the programmer, shader compilation errors need to be checked for specifically. This process is typically handled in multiple steps: checking if compilation was successful, and if not, retrieving the error message, and deleting the shader object to free up memory. This is handled with the following functions:

glGetShaderiv(*shaderRef, shaderInfo*)

Returns information from the shader referenced by the parameter *shaderRef*. The type of information retrieved is specified with the *shaderInfo* parameter, whose value will be an OpenGL constant such as GL_SHADER_TYPE (to determine the type of shader) or GL_COMPILE_STATUS (to determine if compilation was successful).

glGetShaderInfoLog(*shaderRef*)

Returns information about the compilation process (such as errors and warnings) from the shader referenced by the parameter *shaderRef*.

glDeleteShader(*shaderRef*)

Frees the memory used by the shader referenced by the parameter *shaderRef*, and makes the reference available for future shaders that are created.

With an understanding of these functions, coding in Python can begin. The Python binding PyOpenGL provides access to the needed functions and constants through the OpenGL package and its GL namespace. To begin, in the **core** folder, create a new file named **openGLUtils.py** with the following code:

```
from OpenGL.GL import *

# static methods to load and compile OpenGL shaders
    and link to create programs
class OpenGLUtils(object):

    @staticmethod
    def initializeShader(shaderCode, shaderType):

    # specify required OpenGL/GLSL version
    shaderCode = '#version 330\n' + shaderCode

    # create empty shader object and return reference
      value
    shaderRef = glCreateShader(shaderType)
    # stores the source code in the shader
    glShaderSource(shaderRef, shaderCode)
    # compiles source code previously stored in the
      shader object
    glCompileShader(shaderRef)

    # queries whether shader compile was successful
    compileSuccess = glGetShaderiv(shaderRef,
      GL_COMPILE_STATUS)
```

```
    if not compileSuccess:
        # retrieve error message
        errorMessage = glGetShaderInfoLog(shaderRef)
        # free memory used to store shader program
        glDeleteShader(shaderRef)
        # convert byte string to character string
        errorMessage = '\n' + errorMessage.
          decode('utf-8')
        # raise exception: halt program and print
          error message
        raise Exception( errorMessage )

    # compilation was successful; return shader
        reference value
    return shaderRef
```

Note that in the code above, **`initializeShader`** is declared to be static so that it may be called directly from the **`OpenGLUtils`** class rather than requiring an instance of the class to be created.

Next, a program object must be created and the compiled shaders must be attached and linked together. These tasks require the use of the following functions:

glCreateProgram( )

Creates an empty program object, to which shader objects can be attached, and returns a value by which it can be referenced.

glAttachShader(*programRef*, *shaderRef* **)**

Attaches a shader object specified by the parameter *shaderRef* to the program object specified by the parameter *programRef*.

glLinkProgram(*programRef*)

Links the vertex and fragment shaders previously attached to the program object specified by the parameter *programRef*. Among other things, this process verifies that any variables used to send data from the vertex shader to the fragment shader are declared in both shaders consistently.

As was the case with checking for shader compilation errors, program linking errors need to be checked for manually. This process is completely analogous and uses the following functions:

glGetProgramiv(*programRef* , *programInfo*)

Returns information from the program referenced by the parameter *programRef*. The type of information retrieved is specified with the programInfo parameter, whose value will be an OpenGL constant such as GL_LINK_STATUS (to determine if linking was successful).

glGetProgramInfoLog(*programRef*)

Returns information about the linking process (such as errors and warnings) from the program referenced by the parameter *programRef*.

glDeleteProgram(*programRef*)

Frees the memory used by the program referenced by the parameter *programRef*, and makes the reference available for future programs that are created.

Next, return to the **openGLUtils.py** file, and add the following function:

```
@staticmethod
def initializeProgram(vertexShaderCode,
   fragmentShaderCode):

    vertexShaderRef = OpenGLUtils.initializeShader(
                vertexShaderCode, GL_VERTEX_SHADER)
    fragmentShaderRef = OpenGLUtils.initializeShader(
                fragmentShaderCode, GL_FRAGMENT_SHADER)

    # create empty program object and store reference
      to it
    programRef = glCreateProgram()

    # attach previously compiled shader programs
    glAttachShader(programRef, vertexShaderRef)
    glAttachShader(programRef, fragmentShaderRef)

    # link vertex shader to fragment shader
    glLinkProgram(programRef)
```

```
    # queries whether program link was successful
    linkSuccess = glGetProgramiv(programRef,
      GL_LINK_STATUS)
    if not linkSuccess:
        # retrieve error message
        errorMessage = glGetProgramInfoLog(programRef)
        # free memory used to store program
        glDeleteProgram(programRef)
        # convert byte string to character string
        errorMessage = '\n' + errorMessage.
          decode('utf-8')
        # raise exception: halt application and print
          error message
        raise Exception( errorMessage )


    # linking was successful; return program reference
      value
    return programRef
```

There is one additional OpenGL function that can be useful when debugging to determine what version of OpenGL/GLSL your computer supports:

glGetString(*name*)

Returns a string describing some aspect of the currently active OpenGL implementation, specified by the parameter *name*, whose value is one of the OpenGL constants: GL_VENDOR, GL_RENDERER, GL_VERSION, or GL_SHADING_VERSION.

In the **`OpenGLUtils`** class, add the following function. (Note that the **`decode`** function is used to convert the byte string returned by **`glGetString`** to a standard Python string.)

```
@staticmethod
def printSystemInfo():
      print("  Vendor: " + glGetString(GL_VENDOR).
        decode('utf-8') )
     print("Renderer: " + glGetString(GL_RENDERER).
       decode('utf-8') )
```

```
print("OpenGL version supported: " +
   glGetString(GL_VERSION).decode('utf-8') )
print("  GLSL version supported: " +
    glGetString(GL_SHADING_LANGUAGE_VERSION).
      decode('utf-8') )
```

In the next section, you will learn how to use the functions from this class to compile shader code in an application.

2.2.3 Rendering in the Application

Next, you will create another application (once again, extending the **`Base`** class) that contains the shader code previously discussed and uses the functions from the utility class to compile the shaders and create the program. In the application itself, there are a few more tasks that must be performed that also require OpenGL functions.

As mentioned in the discussion of the graphics pipeline in the previous chapter, VAOs will be used to manage vertex related data stored in vertex buffers. Even though this first example does not use any buffers, many implementations of OpenGL require a VAO to be created and bound in order for the application to work correctly. The two functions you will need are as follows:

glGenVertexArrays(*vaoCount*)

Returns a set of available VAO references. The number of references returned is specified by the integer parameter *vaoCount*.

glBindVertexArray(*vaoRef*)

Binds a VAO referenced by the parameter *vaoRef*, after first creating the object if no such object currently exists. Unbinds any VAO that was previously bound.

Next, in order to specify the program to be used when rendering, and to start the rendering process, you will need to use the following two functions:

glUseProgram(*programRef*)

Specifies that the program to be used during the rendering process is the one referenced by the parameter *programRef*.

glDrawArrays(drawMode , firstIndex , indexCount)

Renders geometric primitives (points, lines, or triangles) using the program previously specified by glUseProgram. The program's vertex shader uses data from arrays stored in enabled vertex buffers, beginning at the array index specified by the parameter *firstIndex*. The total number of array elements used is specified by the parameter *indexCount*. The type of geometric primitive rendered and the way in which the vertices are grouped is specified by the parameter *drawMode*, whose value is an OpenGL constant such as GL_POINTS, GL_LINES, GL_LINE_LOOP, GL_TRIANGLES, or GL_TRIANGLE_FAN.

Finally, an aesthetic consideration: depending on your computer display, it may be difficult to observe a single point, which is rendered as a single pixel by default. To make this point easier to see, you will use the following function to increase the size of the rendered point.

glPointSize(*size*)

Specifies that points should be rendered with diameter (in pixels) equal to the integer parameter *size*. (If not specified, the default value is 1.)

You are now prepared to write the application. In your main folder, create a new file named **`test-2-2.py`** with the following code:

```
from core.base import Base
from core.openGLUtils import OpenGLUtils
from OpenGL.GL import *

# render a single point
class Test(Base):

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

        ### initialize program ###

        # vertex shader code
        vsCode = """
        void main()
```

```
        {
            gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
        }
        """


        # fragment shader code
        fsCode = """
        out vec4 fragColor;
        void main()
        {
            fragColor = vec4(1.0, 1.0, 0.0, 1.0);
        }
        """


        # send code to GPU and compile; store program
          reference
        self.programRef = OpenGLUtils.initializeProgram(
           vsCode, fsCode)

        ### set up vertex array object ###
        vaoRef = glGenVertexArrays(1)
        glBindVertexArray(vaoRef)

        ### render settings (optional) ###

        # set point width and height
        glPointSize( 10 )

    def update(self):

        # select program to use when rendering
        glUseProgram( self.programRef )

        # renders geometric objects using selected
          program
        glDrawArrays( GL_POINTS, 0, 1 )

# instantiate this class and run the program
Test().run()
```

In this program, note that the source code for the vertex shader and the fragment shader are written using multiline strings, which are enclosed with triple quotation marks, and the diameter of the rendered point is set

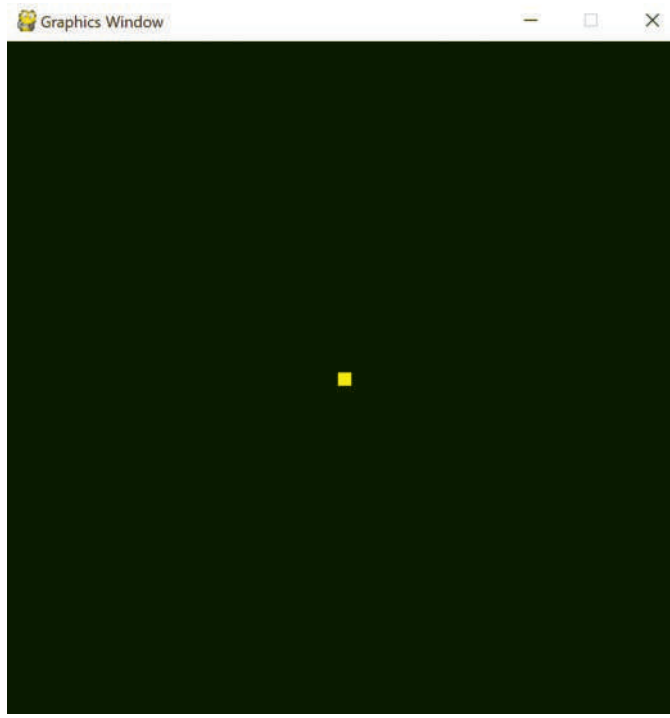

FIGURE 2.3 Rendering a single point.

to be 10 pixels in diameter. When you run this program, a window should appear, similar to that shown in Figure 2.3.

Once you have successfully run this program and viewed the rendered point, you should experiment with changing the shader code. As mentioned previously, experiment with changing the *x*, *y*, and *z* components of the position vector, or the *r*, *g*, and *b* components of the color vector. Try changing the point size specified in the main application. You may even want to purposefully introduce errors in the shader code to learn what error messages are displayed in each case. Some common errors include

- not including a semicolon at the end of a statement
- using an **`int`** instead of a **`float`**
- assigning the wrong data type to a variable (such as assigning a **`vec3`** to the **`vec4`** type variable **`gl _ Position`**)
- using the wrong number of values in a vector

Try introducing each of the errors into your shader code one at a time, and familiarize yourself with the corresponding error messages. Once you are finished experimenting with the code, the next topic you will learn about is using vertex buffers to draw two-dimensional shapes.

2.3 DRAWING SHAPES

Every vertex shader that you write (with the single exception of the one-point example from the previous section) will use arrays of data stored in vertex buffers. Multiple points are needed to specify the shape of any two-dimensional or three-dimensional object. In what follows, you will learn about the OpenGL functions that enable applications to work with vertex buffers, create a reusable class to perform these tasks, and then write a series of applications that use this class to create one or more shapes with one or more colors.

2.3.1 Using Vertex Buffers

Most of the tasks involving vertex buffers take place when an application is initialized, and so it may be helpful to review the discussion of the application stage in the graphics pipeline from the previous chapter. In brief, the relevant steps in this stage are creating buffers, storing data in buffers, and specifying associations between vertex buffers and shader variables. These tasks are handled by the six OpenGL functions described in this section, starting with

glGenBuffers(*bufferCount*)

Returns a set of available vertex buffer references. The number of references returned is specified by the integer parameter *bufferCount*.

Note that in contrast to the OpenGL functions glCreateShader and glCreateProgram, the function glGenBuffers does not actually create an object; this is handled by the next function.

glBindBuffer(*bindTarget* , *bufferRef* **)**

Creates a buffer object referenced by the parameter *bufferRef* if no such object exists. The buffer object is bound to the target specified by the parameter *bindTarget*, whose value is an OpenGL constant such as GL_ARRAY_BUFFER (for vertex attributes) or GL_TEXTURE_BUFFER (for texture data).

Binding a buffer to a target is a necessary precursor for much of what happens next. The OpenGL functions that follow—those that relate to a buffer in some way—do not contain a parameter that directly references a buffer. Instead, some functions (such as **`glBufferData`**, discussed next) include a parameter like *bindTarget* and thus affect whichever buffer is currently bound to *bindTarget*. In other cases, the function may only be applicable to a particular type of buffer, in which case, *bindTarget* is implicit and thus not included. For example, functions that only apply to vertex attribute data (such as **`glVertexAttribPointer`**, discussed later) only affect vertex buffers and thus automatically affect the buffer bound to GL_ARRAY_BUFFER.

glBufferData(*bindTarget , bufferData , bufferUsage* **)**

Allocates storage for the buffer object currently bound to the target specified by the parameter *bindTarget* and stores the information from the parameter *bufferData*. (Any data that may have been previously stored in the associated buffer is deleted.) The parameter *bufferUsage* indicates how the data will most likely be accessed, and this information may be used by the GL implementation to improve performance. The value of *bufferUsage* is an OpenGL constant such as GL_STATIC_DRAW (which indicates the buffer contents will be modified once) or GL_DYNAMIC_DRAW (which indicates the buffer contents will be modified many times).

In order to set up an association between a vertex buffer and a shader variable, the reference to the shader variable in a given program must be obtained. This can be accomplished by using the following function:

glGetAttribLocation(*programRef , variableName* **)**

Returns a value used to reference an attribute variable (indicated by the type qualifier **in**) with name indicated by the parameter *variableName* and declared in the vertex shader of the program referenced by the parameter *programRef*. If the variable is not declared or not used in the specified program, the value -1 is returned.

Once the reference for an attribute variable has been obtained and the corresponding vertex buffer is bound, the association between the buffer and variable can be established using the function presented next:

glVertexAttribPointer(*variableRef* , *size* , *baseType* , *normalize* , *stride* , *offset* **)**

Specifies that the attribute variable indicated by the parameter *variableRef* will receive data from the array stored in the vertex buffer currently bound to GL_ARRAY_BUFFER. The basic data type is specified by the parameter *baseType*, whose value is an OpenGL constant such as GL_INT or GL_FLOAT. The number of components per attribute is specified by the parameter *size*, which is the integer 1, 2, 3, or 4 depending on if the attribute is a basic data type or a vector with 2, 3, or 4 components. The flexibility offered by the last three parameters will not be needed in this book, but in brief: the parameter *normalize* is a boolean value that specifies if vector attributes should be rescaled to have length 1, while the parameters *stride* and *offset* are used to specify how this attribute's data should be read from the associated buffer (it is possible to pack and interleave data for multiple attributes into a single buffer, which can be useful in reducing total calls to the GPU).

So, for example, if the shader variable has type `int`, then you would use the parameters *size* = 1 and *baseType* = GL_INT, while if the shader variable has type `vec3` (a vector containing three `float` values), then you would use the parameters *size* = 3 and *baseType* = GL_FLOAT (The final three parameters will not be covered in this book, and they will always be assigned their default values of *normalize* = False, *stride* = 0, and *offset* = None.).

Even after an association has been specified, the corresponding buffer will not be accessed during rendering unless explicitly enabled with the following function:

glEnableVertexAttribArray(*variableRef*)

Specifies that the values in the vertex buffer bound to the shader variable referenced by the parameter *variableRef* will be accessed and used during the rendering process.

Finally, as a reminder, vertex array objects (VAOs) will be used to manage the vertex-related data specified by these functions. While a VAO is bound, it stores information including the associations between vertex buffers and attribute variables in a program, how data is arranged within each buffer, and which associations have been enabled.

2.3.2 An Attribute Class

With the previously described OpenGL functions, you are now ready to create a Python class to manage attribute data and related tasks. The information managed by the class will include the type of data (**`float`**, **`vec2`**, **`vec3`**, or **`vec4`**), an array containing the data, and the reference of the vertex buffer where the data is stored. The two main tasks handled by the class will be storing the array of data in a vertex buffer, and associating the vertex buffer to a shader variable in a given program; each of these tasks will be implemented with a function. To begin, in the **`core`** folder, create a new file named **`attribute.py`** with the following code:

```
from OpenGL.GL import *
import numpy

class Attribute(object):

    def __init__(self, dataType, data):

        # type of elements in data array:
        #   int | float | vec2 | vec3 | vec4
        self.dataType = dataType

        # array of data to be stored in buffer
        self.data = data

        # reference of available buffer from GPU
        self.bufferRef = glGenBuffers(1)

        # upload data immediately
        self.uploadData()

    # upload this data to a GPU buffer
    def uploadData(self):

        # convert data to numpy array format;
        #   convert numbers to 32 bit floats
        data = numpy.array( self.data ).astype( numpy.
          float32 )

        # select buffer used by the following
          functions
```

```
        glBindBuffer(GL_ARRAY_BUFFER, self.bufferRef)

        # store data in currently bound buffer
        glBufferData(GL_ARRAY_BUFFER, data.ravel(),
          GL_STATIC_DRAW)

    # associate variable in program with this buffer
    def associateVariable(self, programRef,
      variableName):

        # get reference for program variable with
          given name
        variableRef = glGetAttribLocation(programRef,
          variableName)

        # if the program does not reference the
          variable, then exit
        if variableRef == -1:
            return

        # select buffer used by the following
          functions
        glBindBuffer(GL_ARRAY_BUFFER, self.bufferRef)

        # specify how data will be read
        #   from the currently bound buffer into the
            specified
        #   variable
        if self.dataType == "int":
            glVertexAttribPointer(
                variableRef, 1, GL_INT, False, 0, None)
        elif self.dataType == "float":
            glVertexAttribPointer(
                variableRef, 1, GL_FLOAT, False, 0,
                  None)
        elif self.dataType == "vec2":
            glVertexAttribPointer(
                variableRef, 2, GL_FLOAT, False, 0,
                  None)
        elif self.dataType == "vec3":
            glVertexAttribPointer (variableRef, 3,
               GL_FLOAT, False, 0, None)
```

```
elif self.dataType == "vec4":
    glVertexAttribPointer
        (variableRef, 4, GL_FLOAT, False, 0,
           None)
else:
    raise Exception("Attribute " +
      variableName +
                        " has unknown type "
                            + self.dataType)

# indicate that data will be streamed to this
  variable
glEnableVertexAttribArray( variableRef )
```

In addition to the comments throughout the code above, note the following design considerations:

- The code from the **uploadData** function could have been part of the class initialization. However, this functionality is separated in case the values in the variable **data** need to be changed and the corresponding buffer updated later, in which case the function can be called again.
- The numpy library is used to convert the Python list into a compatible format and ensure the elements of the array are of the correct data type.
- This class does not store the name of the variable or the reference to the program that will access the buffer, because in practice, there may be more than one program that does so, and the variables in each program may have different names.
- It may not appear that the information from **glVertexAttribPointer**—in particular, the associations between vertex buffers and program variables—is being stored anywhere. However, this information is in fact stored by whichever VAO was bound prior to these function calls.

2.3.3 Hexagons, Triangles, and Squares

Finally, considering aesthetics once again: since you will be rendering lines with the following program, you may wish to increase their width to make them easier to see (the default line width is a single pixel). This can be accomplished with the following function:

glLineWidth(*width*)

Specifies that lines should be rendered with width (in pixels) equal to the integer parameter *width*. (If not specified, the default value is 1.) Some OpenGL implementations only support lines of width 1, in which case attempting to set the width to a value larger than 1 will cause an error.

With this knowledge, you are now ready to create applications that render one or more shapes. This next application renders an outlined hexagon. In your main folder, create a new file named **test-2-3.py** with the following code:

```
from core.base import Base
from core.openGLUtils import OpenGLUtils
from core.attribute import Attribute
from OpenGL.GL import *

# render six points in a hexagon arrangement
class Test(Base):

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

        ### initialize program ###
        vsCode = """
        in vec3 position;
        void main()
        {
            gl_Position = vec4(
               position.x, position.y, position.z, 1.0);
        }
        """

        fsCode = """
        out vec4 fragColor;
        void main()
        {
            fragColor = vec4(1.0, 1.0, 0.0, 1.0);
        }
        """

        self.programRef = OpenGLUtils.
          initializeProgram(vsCode, fsCode)
```

```
        ### render settings (optional) ###
        glLineWidth( 4 )

        ### set up vertex array object ###
        vaoRef = glGenVertexArrays(1)
        glBindVertexArray(vaoRef)

        ### set up vertex attribute ###
        positionData = [ [ 0.8, 0.0, 0.0], [ 0.4, 0.6,
                           0.0],
                         [-0.4, 0.6, 0.0], [-0.8, 0.0,
                          0.0],
                         [-0.4, -0.6, 0.0], [0.4,
                          -0.6, 0.0] ]
        self.vertexCount = len(positionData)
        positionAttribute = Attribute( "vec3",
          positionData )
        positionAttribute.associateVariable(
           self.programRef, "position" )

    def update(self):
        glUseProgram( self.programRef )
        glDrawArrays( GL_LINE_LOOP , 0 , self.
          vertexCount )

# instantiate this class and run the program
Test().run()
```

There are many similarities between this application and the previous application (the file **test-2-2.py**, which rendered a single point): the fragment shader code is identical (pixels are colored yellow), and both use the functions **initializeProgram**, **glUseProgram**, and **glDrawArrays**. The major differences are

- The vertex shader declares a **vec3** variable named **position** with the type qualifier **in**. This indicates that **position** is an attribute variable, and thus, it will receive data from a vertex buffer.
- The vertex attribute is set up and configured with three lines of code, corresponding to the following tasks: create an array of position data, create an Attribute object (which also stores data in a buffer), and set up the association with the attribute variable **position**.

- A VAO is created to store information related to the vertex buffer. The VAO reference is not stored in a class variable, because there is only one set of associations to manage and so the VAO remains bound the entire time. (When there are multiple sets of associations to manage, the VAO references will be stored in class variables so that they can be bound in the **`update`** function when needed.)
- The **`glDrawArrays`** function parameter GL_LINE_LOOP indicates that lines will be rendered from each point to the next, and the last point will also be connected by the first; in addition, the number of elements in the vertex buffer is stored in the variable **`vertexCount`**, for convenience and readability.

When you run this program, a window should appear similar to that shown in Figure 2.4.

This is an opportune time to experiment with draw mode parameters. Two additional line rendering configurations are possible. If you want to draw a line segment from each point to the next but not connect the last point to the first, this is accomplished with the parameter GL_LINE_STRIP. Alternatively, if you want to draw a series of disconnected line

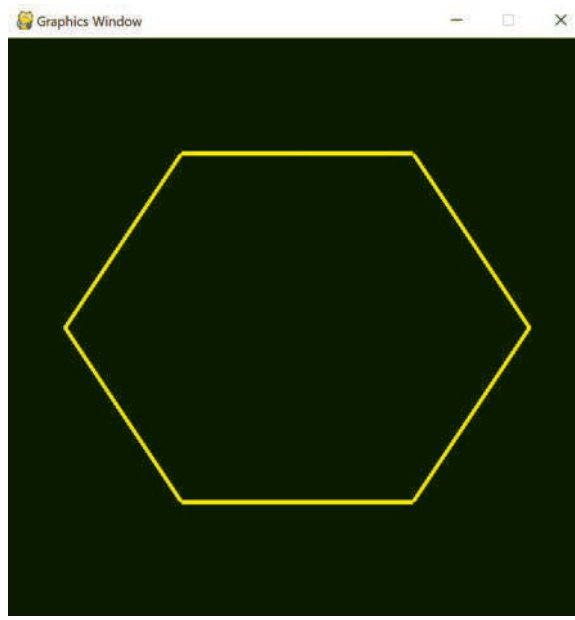

FIGURE 2.4 Rendering six points with GL_LINE_LOOP.

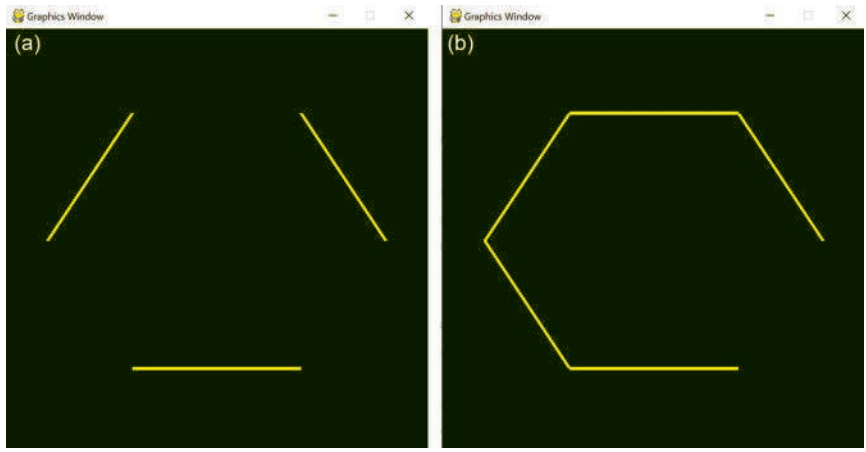

FIGURE 2.5 Rendering six points with GL_LINES (a) and GL_LINE_STRIP (b).

segments, then you would use GL_LINES, which connects each point with an even index in the array to the following point in the array. If the array of points is denoted by [A, B, C, D, E, F], then GL_LINES groups the points into the disjoint pairs (A, B), (C, D), (E, F) and draws a line segment for each pair. (If the number of points to render was odd, then the last point would be ignored in this case.) These cases are illustrated in Figure 2.5; note that A represents the rightmost point and the points in the array are listed in counterclockwise order.

It is also possible to render two-dimensional shapes with filled-in triangles. As is the case with rending lines, there are multiple parameters that result in different groupings of points. The most basic is GL_TRIANGLES, which groups points into disjoint triples, analogous to how GL_LINES groups points into disjoint pairs; continuing the previous notation, the groups are (A, B, C) and (D, E, F). Also useful in this example is GL_TRIANGLE_FAN, which draws a series of connected triangles where all triangles share the initial point in the array, and each triangle shares an edge with the next, resulting in a fan-like arrangement. For this example, the groups are (A, B, C), (A, C, D), (A, D, E), and (A, E, F), which results in a filled-in hexagon (and would similarly fill any polygon where the points are specified in order around the circumference). These cases are illustrated in Figure 2.6.

Now that you have seen how to use a single buffer in an application, you will next render two different shapes (a triangle and a square), whose vertex positions are stored in two separate buffers with different sizes. No

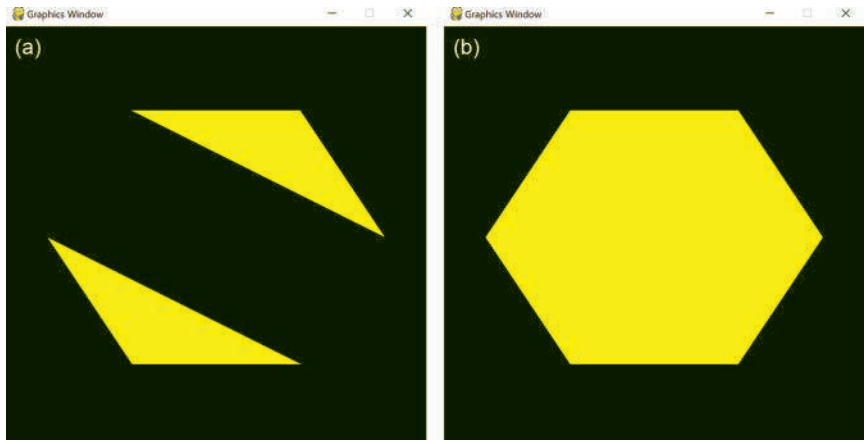

FIGURE 2.6 Rendering six points with GL_TRIANGLES (a) and GL_TRIANGLE_FAN (b).

new concepts or functions are required, and there are no changes to the shader code from the previous example. The main difference is that the **glDrawArrays** function will need to be called twice (once to render each shape), and prior to each call, the correct buffer needs to be associated to the attribute variable **position**. This will require two VAOs to be created and bound when needed (before associating variables and before drawing shapes), and therefore, the VAO references will be stored in variables with class-level scope. To proceed, in your main folder, create a new file named **test-2-4.py** with the following code:

```
from core.base import Base
from core.openGLUtils import OpenGLUtils
from core.attribute import Attribute
from OpenGL.GL import *

# render two shapes
class Test(Base):

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

        ### initialize program ###
        vsCode = """
        in vec3 position;
        void main()
```

```
{
    gl_Position = vec4(
       position.x, position.y, position.z, 1.0);
}
"""


fsCode = """
out vec4 fragColor;
void main()
{
    fragColor = vec4(1.0, 1.0, 0.0, 1.0);
}
"""


self.programRef = OpenGLUtils.
  initializeProgram(vsCode, fsCode)

### render settings ###
glLineWidth(4)


### set up vertex array object - triangle ###
self.vaoTri = glGenVertexArrays(1)
glBindVertexArray(self.vaoTri)
positionDataTri = [[-0.5, 0.8, 0.0], [-0.2,
  0.2, 0.0],
   [-0.8, 0.2, 0.0]]
self.vertexCountTri = len(positionDataTri)
positionAttributeTri = Attribute("vec3",
  positionDataTri)
positionAttributeTri.associateVariable(
   self.programRef, "position" )

### set up vertex array object - square ###
self.vaoSquare = glGenVertexArrays(1)
glBindVertexArray(self.vaoSquare)
positionDataSquare = [[0.8, 0.8, 0.0], [0.8,
                       0.2, 0.0],
                      [0.2, 0.2, 0.0], [0.2,
                        0.8, 0.0]]
self.vertexCountSquare =
  len(positionDataSquare)
```

```
        positionAttributeSquare = Attribute(
            "vec3", positionDataSquare)
        positionAttributeSquare.associateVariable(
            self.programRef, "position" )

    def update(self):
        # using same program to render both shapes
        glUseProgram( self.programRef )

        # draw the triangle
        glBindVertexArray( self.vaoTri )
        glDrawArrays( GL_LINE_LOOP , 0 , self.
          vertexCountTri )
        # draw the square
        glBindVertexArray( self.vaoSquare )
        glDrawArrays( GL_LINE_LOOP , 0 , self.
          vertexCountSquare )

# instantiate this class and run the program
Test().run()
```

The result of running this application is illustrated in Figure 2.7.

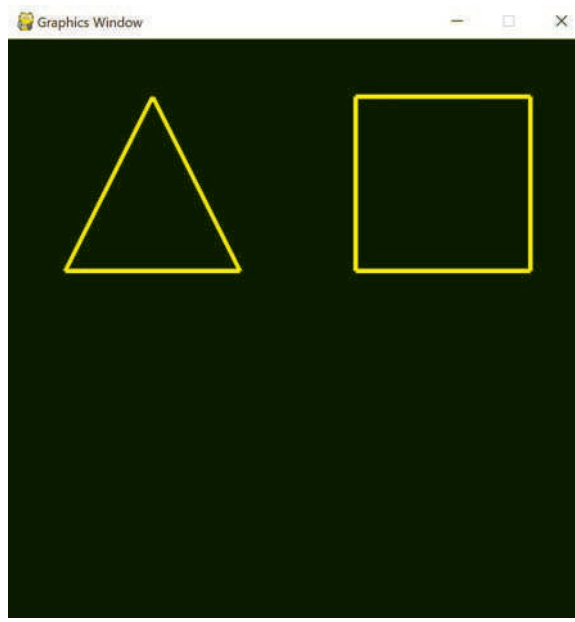

FIGURE 2.7 Rendering a triangle and a square.

Now that you have learned how to work with multiple buffers, you may want to use them for purposes other than positions; for example, as previously mentioned, buffers can be used to store vertex colors. Since attribute data is passed into the vertex shader, but the color data needs to be used in the fragment shader, this requires you to pass data from the vertex shader to the fragment shader, which is the topic of the next section.

2.3.4 Passing Data between Shaders

In this section, you will create an application that renders six points (once again, arranged in a hexagonal pattern) in six different colors. This requires data (in particular, the color data) to be sent from the vertex shader to the fragment shader. Recall from the introductory explanation of OpenGL Shading Language that *type qualifiers* are keywords that modify the properties of a variable, and in particular

- In a vertex shader, the keyword **`in`** indicates that values will be supplied from a buffer, while the keyword **`out`** indicates that values will be passed to the fragment shader.
- In a fragment shader, the keyword **`in`** indicates that values will be supplied from the vertex shader, after having been interpolated during the rasterization stage.

This application described will use the shaders that follow. First, the vertex shader code:

```
in vec3 position;
in vec3 vertexColor;
out vec3 color;
void main()
{
    gl_Position = vec4(position.x, position.y,
       position.z, 1.0);
    color = vertexColor;
}
```

Note the presence of two **`in`** variables; this is because there are two vertex attributes, **`position`** and **`vertexColor`**. The data arrays for each of these attributes will be stored in a separate buffer and associated with the corresponding variable. The arrays should contain the same number of elements; the length of the arrays is the number of vertices. For a given

vertex, the same array index will be used to retrieve data from each of the attribute arrays. For example, the vertex that uses the value from index N in the position data array will also use the value from index N in the vertex color data array.

In addition, the vertex shader contains an **out** variable, **color**, which will be used to transmit the data to the fragment shader; a value must be assigned to this (and in general, any **out** variable) within the vertex shader. In this case, the value in **vertexColor** just "passes through" to the variable **color**; no computations are performed on this value.

Next, the fragment shader code:

```
in vec3 color;
out vec4 fragColor;
void main()
{
    fragColor = vec4(color.r, color.g, color.b, 1.0);
}
```

Note here the presence of an **in** variable, which must have the same name as the corresponding **out** variable from the vertex shader. (Checking for pairs of **out**/**in** variables with consistent names between the vertex and fragment shaders is one of the tasks performed when a program is linked.) The components of the vector are accessed using the (r, g, b) naming system for readability, as they correspond to color data in this context.

With an understanding of these new shaders, you are ready to create the application. It uses separate **Attribute** objects to manage the attribute data, but only one VAO needs to be created since these vertex buffers are associated with different variables in the program. To proceed, in your main folder, create a new file named **test-2-5.py** with the following code:

```
from core.base import Base
from core.openGLUtils import OpenGLUtils
from core.attribute import Attribute
from OpenGL.GL import *

# render shapes with vertex colors
class Test(Base):
```

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

    ### initialize program ###
    vsCode = """
    in vec3 position;
    in vec3 vertexColor;
    out vec3 color;
    void main()
    {
        gl_Position = vec4(position.x, position.y,
           position.z, 1.0);
        color = vertexColor;
    }
    """

    fsCode = """
    in vec3 color;
    out vec4 fragColor;
    void main()
    {
        fragColor = vec4(color.r, color.g,
        color.b, 1.0);
    }
    """

    self.programRef = OpenGLUtils.
      initializeProgram(vsCode, fsCode)

    ### render settings (optional) ###
    glPointSize( 10 )
    glLineWidth( 4 )

    ### set up vertex array object ###
    vaoRef = glGenVertexArrays(1)
    glBindVertexArray(vaoRef)

    ### set up vertex attributes ###
    positionData = [ [0.8, 0.0, 0.0], [0.4, 0.6,
                        0.0],
       [-0.4, 0.6, 0.0],  [-0.8, 0.0, 0.0], [-0.4,
         -0.6, 0.0], [0.4, -0.6, 0.0] ]
```

```
        self.vertexCount = len(positionData)


        positionAttribute = Attribute("vec3",
          positionData)
        positionAttribute.associateVariable(
           self.programRef, "position" )


        colorData = [ [1.0, 0.0, 0.0], [1.0, 0.5,
          0.0],
           [1.0, 1.0, 0.0], [0.0, 1.0, 0.0], [0.0,
              0.0, 1.0], [0.5, 0.0, 1.0] ]

        colorAttribute = Attribute("vec3", colorData)
        colorAttribute.associateVariable(
           self.programRef, "vertexColor" )


    def update(self):
        glUseProgram( self.programRef )
        glDrawArrays( GL_POINTS, 0, self.vertexCount )

# instantiate this class and run the program
Test().run()
```

The result of running this application is illustrated in Figure 2.8.

Recall from the previous chapter that during the graphics pipeline, between the geometry processing stage (which involves the vertex shader) and the pixel processing stage (which involves the fragment shader) is the rasterization stage. At this point, the programs are sufficiently complex to illustrate the interpolation of vertex attributes that occurs during this stage.

During the rasterization stage, points are grouped into *geometric primitives*: points, lines, or triangles, depending on the draw mode specified. Then, the GPU determines which pixels correspond to the space occupied by each geometric primitive. For each of these pixels, the GPU generates a fragment, which stores the data needed to determine the final color of each pixel. Any vertex data passed from the vertex shader via an **`out`** variable will be interpolated for each of these fragments; a weighted average of the vertex values is automatically calculated, depending on the distances from

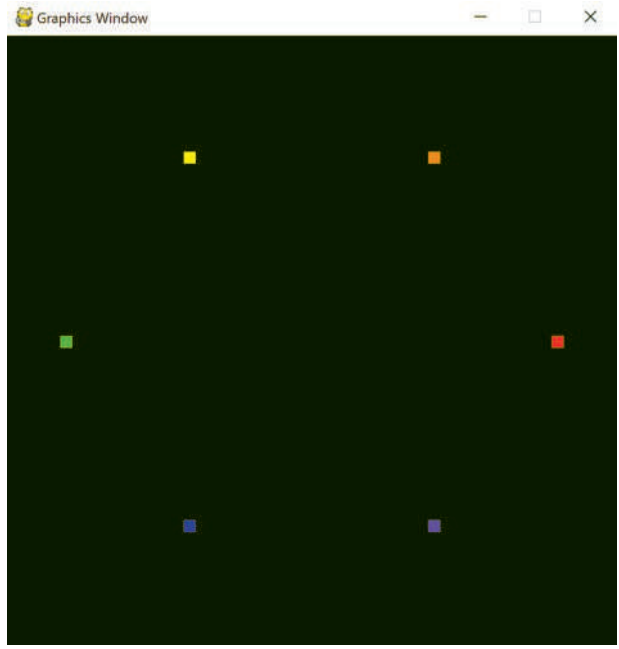

FIGURE 2.8 Rendering six points with vertex colors, using GL_POINTS.

the interior point to the original vertices. It is this interpolated value that is sent to the corresponding **`in`** variable in the fragment shader.

For a numerical example of interpolation, suppose a line segment is to be drawn between a point $P1$ with color $C1 = [1, 0, 0]$ (red) and a point $P2$ with color $C2 = [0, 0, 1]$ (blue). All points along the segment will be colored according to some combination of the values in $C1$ and $C2$; points closer to $P1$ will have colors closer to $C1$ (more red) while points closer to $P2$ will have colors closer to $C2$ (more blue). For example, the point on the segment exactly halfway between $P1$ and $P2$ will be colored $0.5 \cdot C1 + 0.5 \cdot C2 = [0.5, 0, 0.5]$, a shade of purple. For another example, a point that is a 25% of the distance along the segment from $P1$ to $P2$ will be colored $0.75 \cdot C1 + 0.25 \cdot C2 = [0.75, 0, 0.25]$, a reddish-purple. Similarly, colors of points within a triangle are weighted averages of the colors of its three vertices; the color of an interior point depends on the distance from the interior point to each of these three vertices. To see examples of color interpolation within lines and triangles, you can run the previous program but changing the *drawMode* parameter of **`glDrawArrays`** to either GL_LINE_LOOP or GL_TRIANGLE_FAN; the results are illustrated in Figure 2.9.

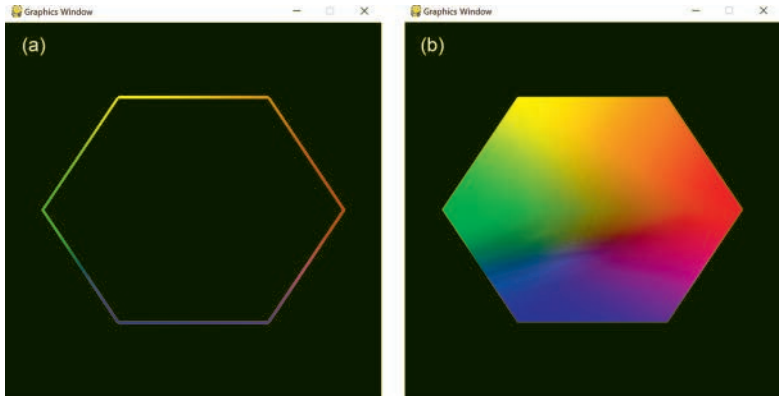

FIGURE 2.9 Rendering six points with vertex colors, using GL_LINE_LOOP (a) and GL_TRIANGLE_FAN (b).

Many times you may want to draw a shape that is a single color. This is technically possible using the previous shaders: if your shape has N vertices, you could create an array that contains the same color N times. However, there is a much less redundant method to accomplish this task using uniform variables, which are introduced in the next section.

2.4 WORKING WITH UNIFORM DATA

There are many scenarios in computer graphics where you may want to repeatedly use the same information in shader programming. For example, you may want to translate all the vertices that define a shape by the same amount, or you may want to draw all the pixels corresponding to a shape with the same color. The most flexible and efficient way to accomplish such tasks is by using *uniform variables*: global variables that can be accessed by both the vertex shader and the fragment shader, and whose values are constant while each shape is being drawn (but can be changed between draw function calls).

2.4.1 Introduction to Uniforms

Uniform shader variables, declared with the type qualifier **`uniform`**, provide a mechanism to send data directly from variables in a CPU application to variables in a GPU program. In contrast to attribute data, uniform data is not stored in GPU buffers and is not managed with VAOs. Similar to attribute data, a reference to the location of the uniform variable must be obtained before data may be sent to it. This is accomplished with the following OpenGL function:

glGetUniformLocation(*programRef*, *variableName* **)**

Returns a value used to reference a uniform variable (indicated by the type qualifier **`uniform`**) with name indicated by the parameter *variableName* and declared in the program referenced by the parameter *programRef*. If the variable is not declared or not used in the specified program, the value -1 is returned.

The other OpenGL functions required to work with uniforms are used to store data in uniform variables. However, unlike the function **`glVertexAttribPointer`** which specified the data type with its parameters *baseType* and *size*, specifying different types of uniform data is handled by different functions. The notation {*a*|*b*|*c*|…} used below (similar to that used in the official OpenGL documentation provided by the Khronos group) indicates that one of the values *a*, *b*, *c* … in the set should be each chosen to specify the name of a function. In other words, the notation below indicates that the possible function names are **`glUniform1f`**, **`glUniform2f`**, **`glUniform3f`**, **`glUniform4f`**, **`glUniform1i`**, **`glUniform2i`**, **`glUniform3i`**, and **`glUniform4i`**.

glUniform{ 1 | 2 | 3 | 4 }{ f | i }(*variableRef*, *value1*, … **)**

Specify the value of the uniform variable referenced by the parameter *variableRef* in the currently bound program. The number chosen in the function name refers to the number of values sent, while the letter (**f** or **i**) refers to the data type (float or integer).

For example, to specify a single integer value, the form of the function you need to use is **`glUniform1i(variableRef, value1)`**, while to specify the values for a **`vec3`** (a vector consisting of 3 floats), you would instead use **`glUniform3f(variableRef, value1, value2, value3)`**. The boolean values **`true`** and **`false`** correspond to the integers 1 and 0, respectively, and **`glUniform1i`** is used to transmit this data.

2.4.2 A Uniform Class

Just as you previously created an **`Attribute`** class to simplify working with attributes, in this section you will create a **`Uniform`** class to simplify working with uniforms, using the two OpenGL functions discussed in the previous section. Each uniform object will store a value and the name of the value type. One of the tasks of each uniform object will be to locate the reference to the

shader variable that will receive the value stored in the object. The variable reference will be stored in the uniform object, since it is not stored by a VAO, as was the case when working with attributes. The other task will be to send the value in the uniform object to the associated shader variable. This will need to be done repeatedly, as there will typically be multiple geometric objects, each using different values in the uniform variables, which need to be sent to the shader program before the corresponding geometric object is rendered.

With these design considerations in mind, you are ready to create the **Uniform** class. In the **core** folder, create a new file named **uniform.py** with the following code:

```
from OpenGL.GL import *

class Uniform(object):

    def __init__(self, dataType, data):

        # type of data:
        #   int | bool | float | vec2 | vec3 | vec4
        self.dataType = dataType

        # data to be sent to uniform variable
        self.data = data

        # reference for variable location in program
        self.variableRef = None

     # get and store reference for program variable
       with given name
    def locateVariable(self, programRef,
      variableName):
        self.variableRef = glGetUniformLocation(
           programRef, variableName)

     # store data in uniform variable previously
       located
         def uploadData(self):

        # if the program does not reference the
          variable, then exit
        if self.variableRef == -1:
            return
```

```
        if self.dataType == "int":
            glUniform1i(self.variableRef, self.data)
        elif self.dataType == "bool":
            glUniform1i(self.variableRef, self.data)
        elif self.dataType == "float":
            glUniform1f(self.variableRef, self.data)
        elif self.dataType == "vec2":
            glUniform2f(self.variableRef, self.
              data[0], self.data[1])
        elif self.dataType == "vec3":
            glUniform3f(self.variableRef, self.
              data[0], self.data[1],
                          self.data[2])
        elif self.dataType == "vec4":
            glUniform4f(self.variableRef, self.
              data[0], self.data[1],
                          self.data[2], self.data[3])
```

2.4.3 Applications and Animations

In this section, you will first use the **`Uniform`** class to help render an image containing two triangles with the same shape, but in different locations and with different (solid) colors. The code to accomplish this will include the following features:

- a single vertex buffer, used to store the positions of the vertices of a triangle (centered at the origin)
- a single GPU program containing two uniform variables:
  - one used by the vertex shader to translate the position of the triangle vertices
  - one used by the fragment shader to specify the color of each pixel in the triangle
- two **`Uniform`** objects (one to store the position, one to store the color) for each triangle

To continue, in your main folder, create a new file named **`test-2-6.py`** with the following code:

```
from core.base import Base
from core.openGLUtils import OpenGLUtils
from core.attribute import Attribute
```

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


# render two triangles with different positions and
    colors
class Test(Base):

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


        ### initialize program ###
        vsCode = """
        in vec3 position;
        uniform vec3 translation;
        void main()
        {
            vec3 pos = position + translation;
           gl_Position = vec4(pos.x, pos.y, pos.z, 1.0);
        }
        """


        fsCode = """
        uniform vec3 baseColor;
        out vec4 fragColor;
        void main()
        {
            fragColor = vec4(
                baseColor.r, baseColor.g, baseColor.b,
                  1.0);
        }
        """


        self.programRef = OpenGLUtils.
          initializeProgram(vsCode, fsCode)

        ### set up vertex array object ###
        vaoRef = glGenVertexArrays(1)
        glBindVertexArray(vaoRef)
```

```
        ### set up vertex attribute ###
        positionData = [ [0.0, 0.2, 0.0], [0.2, -0.2,
          0.0], [-0.2, -0.2, 0.0] ]
        self.vertexCount = len(positionData)
        positionAttribute = Attribute("vec3",
          positionData)
        positionAttribute.associateVariable(
           self.programRef, "position" )


        ### set up uniforms ###
        self.translation1 = Uniform("vec3", [-0.5,
          0.0, 0.0])
        self.translation1.locateVariable(
           self.programRef, "translation" )


        self.translation2 = Uniform("vec3", [0.5, 0.0,
          0.0])
        self.translation2.locateVariable(
           self.programRef, "translation" )

        self.baseColor1 = Uniform("vec3", [1.0, 0.0,
          0.0])
        self.baseColor1.locateVariable( self.
          programRef, "baseColor" )

        self.baseColor2 = Uniform("vec3", [0.0, 0.0,
          1.0])
        self.baseColor2.locateVariable( self.
          programRef, "baseColor" )



    def update(self):
        glUseProgram( self.programRef )

        # draw the first triangle
        self.translation1.uploadData()
        self.baseColor1.uploadData()
        glDrawArrays( GL_TRIANGLES , 0 , self.
          vertexCount )
```

```
        # draw the second triangle
        self.translation2.uploadData()
        self.baseColor2.uploadData()
        glDrawArrays( GL_TRIANGLES , 0 , self.
          vertexCount )

# instantiate this class and run the program
Test().run()
```

The result of running this application is illustrated in Figure 2.10.

Next, you will learn how to create animated effects by continuously changing the values stored in uniform variables.

In the previous application code examples, you may have wondered why the **`glDrawArrays`** function was called within the **`update`** function—since the objects being drawn were not changing, the result was that the same image was rendered 60 times per second. In those examples, the rendering code technically could have been placed in the **`initialize`** function instead without affecting the resulting image (although it would have been rendered only once). The goal was to adhere to the "life cycle" structure of an application discussed at the beginning of this chapter, which

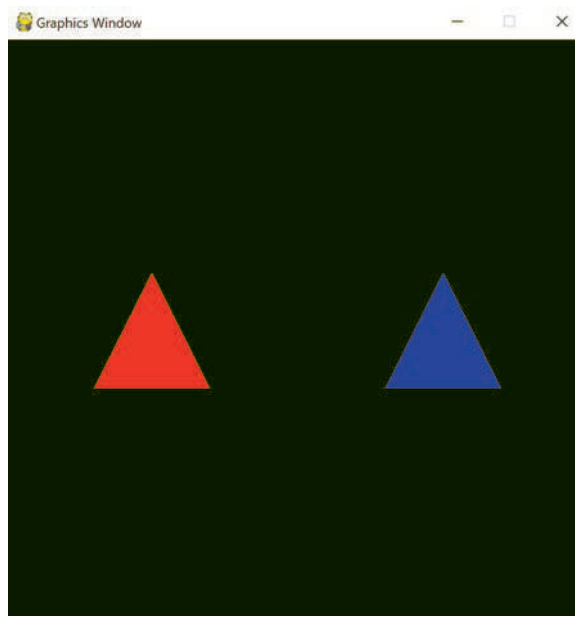

FIGURE 2.10 Two similar triangles rendered using uniform variables.

becomes necessary at this point. Starting in this section, the appearance of the objects will change over time, and thus, the image must be rendered repeatedly.

Since geometric shapes will appear in different positions in subsequent renders, it is important to refresh the drawing area, resetting all the pixels in the color buffer to some default color. If this step is not performed, then in each animation frame, the rendered shapes will be drawn superimposed on the previous frame, resulting in a smeared appearance. To specify the color used when clearing (which effectively becomes the background color), and to actually perform the clearing process, the following two OpenGL functions will be used:

glClearColor(*red* , *green* , *blue* , *alpha* **)**

Specify the color used when clearing the color buffer; the color components are specified by the parameters *red*, *green*, *blue*, and *alpha*. Each of these parameters is a float between 0 and 1; the default values are all 0.

glClear(*mask*)

Reset the contents of the buffer(s) indicated by the parameter *mask* to their default values. The value of mask can be one of the OpenGL constants such as GL_COLOR_BUFFER_BIT (to clear the color buffer), GL_DEPTH_BUFFER_BIT (to clear the depth buffer), GL_STENCIL_BUFFER_BIT (to clear the stencil buffer), or any combination of the constants, combined with the bitwise or operator '|'.

In the next application, you will draw a triangle that continuously moves to the right, and once it moves completely past the right edge of the screen, it will reappear on the left side. (This behavior is sometimes called a *wrap-around effect*, popularized by various 1980s arcade games.) The same shader code will be used as before, and **`Uniform`** objects will be created for the triangle. The **`Uniform`** corresponding to the translation variable will have its value changed (the first component, representing the x-coordinate, will be incremented) during the **`update`** function, prior to clearing the color buffer and rendering the triangle. The wrap-around effect is accomplished by checking if the translation x-coordinate is greater than a certain positive value, and if so, setting it to a certain negative value (these values depending on the size of the triangle).

In your main folder, create a new file named **test-2-7.py** with the following code:

```
from core.base import Base
from core.openGLUtils import OpenGLUtils
from core.attribute import Attribute
from core.uniform import Uniform
from OpenGL.GL import *


# animate triangle moving across screen
class Test(Base):

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


        ### initialize program ###
        vsCode = """
        in vec3 position;
        uniform vec3 translation;
        void main()
        {
            vec3 pos = position + translation;
            gl_Position = vec4(pos.x, pos.y, pos.z,
              1.0);
        }
        """

        fsCode = """
        uniform vec3 baseColor;
        out vec4 fragColor;
        void main()
        {
            fragColor = vec4(
                baseColor.r, baseColor.g, baseColor.b,
                  1.0);
        }
        """

        self.programRef = OpenGLUtils.
           initializeProgram(vsCode, fsCode)

        ### render settings (optional) ###
```

```
        # specify color used when clearly
        glClearColor(0.0, 0.0, 0.0, 1.0)

        ### set up vertex array object ###
        vaoRef = glGenVertexArrays(1)
        glBindVertexArray(vaoRef)

        ### set up vertex attribute ###
        positionData = [ [0.0, 0.2, 0.0], [0.2, -0.2,
          0.0],
           [-0.2, -0.2, 0.0] ]
        self.vertexCount = len(positionData)

        positionAttribute = Attribute("vec3",
          positionData)
        positionAttribute.associateVariable(
           self.programRef, "position" )

        ### set up uniforms ###
        self.translation = Uniform("vec3", [-0.5, 0.0,
          0.0])
        self.translation.locateVariable(
           self.programRef, "translation" )

        self.baseColor = Uniform("vec3", [1.0, 0.0,
          0.0])
        self.baseColor.locateVariable( self.
          programRef, "baseColor" )


    def update(self):

        ### update data ###

        # increase x coordinate of translation
        self.translation.data[0] += 0.01
        # if triangle passes off-screen on the right,
        #   change translation so it reappears on the
           left
        if self.translation.data[0] > 1.2:
            self.translation.data[0] = -1.2

        ### render scene ###
```

```
        # reset color buffer with specified color
        glClear(GL_COLOR_BUFFER_BIT)

        glUseProgram( self.programRef )
        self.translation.uploadData()
        self.baseColor.uploadData()
        glDrawArrays( GL_TRIANGLES , 0 , self.
          vertexCount )

# instantiate this class and run the program
Test().run()
```

It can be difficult to convey an animation with a series of static images, but to this end, Figure 2.11 shows a series of images captured with short time intervals between them.

In the previous application, the movement of the triangle was specified relative to whatever its current position was (it continuously moved to the right of its starting position). The next application will feature time-based movement: the position will be calculated from equations expressed in terms of a time variable.

Since keeping track of time will be useful in many applications, you will begin by adding related functionality to the **`Base`** class. The variable **`time`** will keep track of the number of seconds the application has been running, which will be used to calculate time-based movement. The variable **`deltaTime`** will keep track of the number of seconds that have elapsed during the previous iteration of the run loop (typically 1/60 or approximately 0.017 seconds), which will be useful for future applications. Open the file **`base.py`** from the **`core`** folder, and at the end of the **`__init__`** function, add the following code:

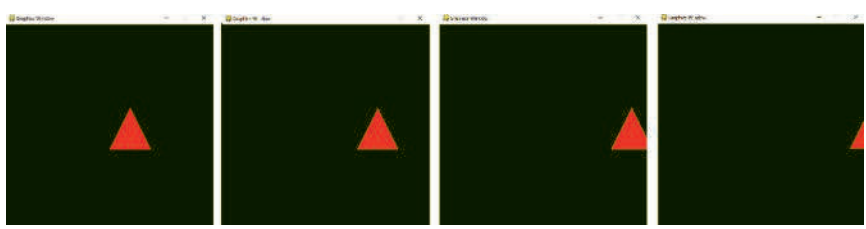

FIGURE 2.11 Frames from an animation of a triangle moving to the right.

```
# number of seconds application has been running
self.time = 0
```

Then, in the **run** function, directly before the statement **self.update()**, add the following code:

```
# seconds since iteration of run loop
self.deltaTime = self.clock.get_time() / 1000
# increment time application has been running
self.time += self.deltaTime
```

In the next application, a triangle will move along a circular path. This is most easily accomplished by using the trigonometric functions sine (sin) and cosine (cos), both of which smoothly oscillate between the values -1 and 1. Positions (x, y) along a circular path can be expressed with the following equations:

$$x = \cos(t),\ y = \sin(t)$$

In your application, the variable t will be the total elapsed time. This particular circular path is centered at the origin and has radius 1; for a path centered at (a, b) with radius r, you can use the following equations:

$$x = r \cdot \cos(t) + a,\ y = r \cdot \sin(t) + b$$

To keep the entire triangle visible as it moves along this path, you will need to use a radius value less than 1.

The code for this application is nearly identical to the previous application, and so to begin, in the main folder, make a copy of the file **test-2-7.py** and name the copy **test-2-8.py**. In this new file, at the top, add the import statement:

```
from math import sin, cos
```

Next, delete the code between the comments **### update data ###** and **### render scene ###**, and in its place, add the following code:

```
self.translation.data[0] = 0.75 * cos(self.time)
self.translation.data[1] = 0.75 * sin(self.time)
```

With these additions, the code for this application is complete.

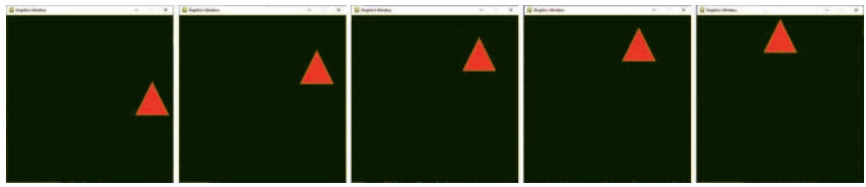

FIGURE 2.12 Frames from an animation of a triangle moving along a circular path.

When you run this application, you will see a triangle moving in a circular path around the origin, in a counterclockwise direction. The **sin** and **cos** functions have a *period* of 2**π** units, meaning that they repeat their pattern of values during each interval of this length, and thus, the triangle will complete a complete revolution around the origin every 2**π** (approximately 6.283) seconds. As before, part of this animation is illustrated with a sequence of images, which are displayed in Figure 2.12.

In the final example of this section, instead of an animation based on changing position, you will create an animation where the triangle color shifts back and forth between red and black. This will be accomplished by changing the value of the red component of the color variable. As before, you will use a sine function to generate oscillating values. Since the components that specify color components range between 0 and 1, while a sine function ranges between –1 and 1, the output values from the sine function will have to be modified. Since the function $f(t)=\sin(t)$ ranges from –1 to 1, the function $f(t)=\sin(t)+1$ shifts the output values to the range 0–2. Next, you must scale the size of the range; the function $f(t)=(\sin(t)+1)/2$ ranges from 0 to 1. Further adjustments are also possible: for example, given a constant *c*, the function $\sin(c \cdot t)$ will traverse its range of output values *c* times faster than $\sin(t)$. The final version of the equation you will use is $f(t)=(\sin(3 \cdot t)+1)/2$

To create this example, start by making a copy of the file **test-2-8.py** from the main folder and name the copy **test-2-9.py**. As before, in this new file, delete the code between the comments **### update data ###** and **### render scene ###**, and in its place, add the following code:

```
self.baseColor.data[0] = (sin(3 * (self.time)) + 1) / 2
```

Running this application, you should see the color of the triangle shifting between red and black every few seconds. This animation is illustrated with a sequence of images, which are displayed in Figure 2.13.

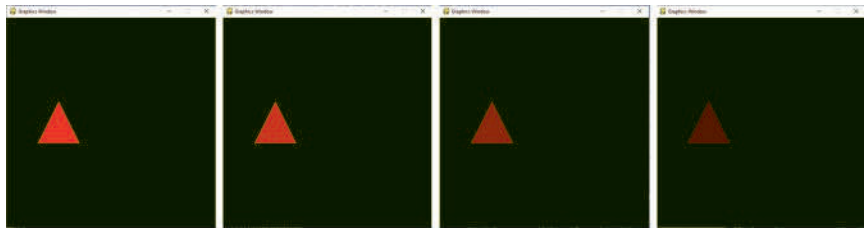

FIGURE 2.13 Frames from an animation of a triangle shifting color.

If desired, you could even make the triangle's color shift along a greater range of colors by also modifying the green and blue components of the color variable. In this case, the sine waves will also need to be shifted, so that each component reaches its peak value at different times. Mathematically, this can be accomplished by adding values to the *t* variable within the sine function. This can be accomplished, for example, by replacing the code added above with the following:

```
self.baseColor.data[0] = (sin(self.time) + 1) / 2
self.baseColor.data[1] = (sin(self.time + 2.1) + 1) / 2
self.baseColor.data[2] = (sin(self.time + 4.2) + 1) / 2
```

Feel free to experiment with and combine these different animations to see what you can produce!

2.5 ADDING INTERACTIVITY

In this section, you will add the ability for the user to interact with an application using the keyboard. The first step will be to add supporting data and functions to the **`Input`** class, and testing it with a text-based application. After establishing that keyboard input works as expected, you will then use this functionality into a graphics-based application, where the user can move a triangle around the screen using the arrow keys.

2.5.1 Keyboard Input with Pygame

The Pygame library provides support for working with user input events; previously, in the **`Input`** class update function, you wrote code to handle quit events. Pygame also detects keydown events, which occur the moment a key is initially pressed, and keyup events, which occur the moment the key is released. The Input class will store the names of these keys in lists that can be checked later (such as in the update function of the main application).

Keys that are designated as down or up should only remain so for a single iteration of the main application loop, and so the contents of these two lists must be cleared before checking the Pygame event list for new events during the next iteration.

An event or action is said to be *discrete* if it happens once at an isolated point in time, or *continuous* if it continues happening during an interval of time. The keydown and keyup events are discrete, but many applications also feature continuous actions (such as moving an object on screen) that occur for as long as a key is being held down. Here, the term *pressed* will be used to refer to the state of a key between the keydown and keyup events (although it should be noted that there is no standard terminology for these three states). You will keep track of the names of pressed keys in a third list. Finally, for readability, the contents of each list can be queried using functions you will add to the **Input** class.

To make the modifications, open the **input.py** file in the **core** folder. In the **Input** class **__init__** function, add the following code:

```
# lists to store key states
#   down, up: discrete event; lasts for one iteration
#   pressed: continuous event, between down and up
      events
self.keyDownList     = []
self.keyPressedList  = []
self.keyUpList       = []
```

Next, in the **update** function, add the following code before the **for** loop:

```
# reset discrete key states
self.keyDownList = []
self.keyUpList = []
```

Within the **for** loop, add the following code:

```
# check for keydown and keyup events;
#   get name of key from event
#   and append to or remove from corresponding lists
if event.type == pygame.KEYDOWN:
    keyName = pygame.key.name( event.key )
    self.keyDownList.append( keyName )
    self.keyPressedList.append( keyName )
if event.type == pygame.KEYUP:
```

```
    keyName = pygame.key.name( event.key )
    self.keyPressedList.remove( keyName )
    self.keyUpList.append( keyName )
```

Finally, add the following three functions to the **Input** class:

```
# functions to check key states
def isKeyDown(self, keyCode):
    return keyCode in self.keyDownList
def isKeyPressed(self, keyCode):
    return keyCode in self.keyPressedList
def isKeyUp(self, keyCode):
    return keyCode in self.keyUpList
```

As previously indicated, you will now create a text-based application to verify that these modifications work as expected, and to illustrate how the class will be used in practice. In your main folder, create a new file named **test-2-10.py** containing the following code:

```
from core.base import Base

# check input
class Test(Base):

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

    def update(self):

        # debug printing
        if len(self.input.keyDownList) > 0:
            print( "Keys down:", self.input.
              keyDownList )

        if len(self.input.keyPressedList) > 0:
            print( "Keys pressed:", self.input.
              keyPressedList )

        if len(self.input.keyUpList) > 0:
            print( "Keys up:", self.input.keyUpList )

# instantiate this class and run the program
Test().run()
```

When you run this program, pressing any keys on the keyboard should cause messages to be printed that show the contents of the non-empty key lists: the names of any keys that are down, pressed, or up. After verifying that this code works as expected, either comment out or delete the code in the application's **update** function and replace it with the following code, which contains examples of how the functions are typically used.

```
# typical usage
if self.input.isKeyDown("space"):
    print( "The 'space' key was just pressed down.")

if self.input.isKeyPressed("right"):
    print( "The 'right' key is currently being
      pressed.")
```

When you run this program, pressing the space bar key should cause a single message to appear each time it is pressed, regardless of how long it is held down. In contrast, pressing the right arrow key should cause a series of messages to continue to appear until the key is released. It is possible that on different operating systems, different names or symbols may be used for particular keys (such as the arrow keys); this can be investigated by running the application with the previous code.

2.5.2 Incorporating with Graphics Programs

Now that the framework is able to process discrete and continuous keyboard input, you will create the graphics-based application described earlier that enables the user to move a triangle using the arrow keys. As this application is similar to many of the animation examples previously discussed, begin by making a copy of the file **test-2-7.py** from the main folder and name the copy **test-2-11.py**. In this new file, at the end of the **initialize** function, add the following code:

```
# triangle speed, units per second
self.speed = 0.5
```

The **speed** variable specifies how quickly the triangle will move across the screen. Recalling that the horizontal or x-axis values displayed on screen range from –1 to 1, a speed of 0.5 indicates that the triangle will be able to move fully across the screen in 4 seconds.

Next, in the **update** function, delete the code between the comments **### update data ###** and **### render scene ###**, and in its place, add the following code:

```
distance = self.speed * self.deltaTime
if self.input.isKeyPressed("left"):
    self.translation.data[0] -= distance
if self.input.isKeyPressed("right"):
    self.translation.data[0] += distance
if self.input.isKeyPressed("down"):
    self.translation.data[1] -= distance
if self.input.isKeyPressed("up"):
    self.translation.data[1] += distance
```

Depending on your operating system, you may need to change the strings checked by the **isKeyPressed** function; for maximum cross-platform compatibility, you may want to use the frequently used set of letters w/a/s/d in place of up/left/down/right, respectively.

Note that this segment of code begins by calculating how far the triangle should be moved across the screen (the distance traveled), taking into account the elapsed time since the previous render, which is stored in the variable **deltaTime** that you added to the **Base** class in Section 2.4.3. The calculation itself is based on the physics formula *speed=distance/time*, which is equivalent to *distance=speed * time*. Also note that a sequence of **if** statements are used, rather than **if-else** statements, which allows the user to press multiple keys simultaneously and move in diagonal directions, or even press keys indicating opposite directions, whose effects will cancel each other out. You may have noticed that translations of the z component (the forward/backward direction) were not included; this is because such a movement will not change the appearance of the shape on screen, since perspective transformations have not yet been introduced into the framework.

Once you have finished adding this code, run the application and try pressing the arrow keys to move the triangle around the screen, and congratulations on creating your first interactive GPU-based graphics program!

2.6 SUMMARY AND NEXT STEPS

In this chapter, you learned how to create animations and interactive applications. Each of these examples involved polygon shapes (the *z*-coordinate of each point was always set to zero), and the movement was

limited to translations (of the x and y coordinates). A natural next goal is to transform these shapes in more advanced ways, such as combining rotations with translations. Perspective transformations also need to be introduced so that translations in the z direction and rotations around the x-axis and y-axis of the scene will appear as expected. In the next chapter, you will learn the mathematical foundations required to create these transformations, and in the process, create truly three-dimensional scenes.

CHAPTER 3

Matrix Algebra and Transformations

In this chapter, you will learn about some mathematical objects—vectors and matrices—that are essential to rendering three-dimensional scenes. After learning some theoretical background, you will apply this knowledge to create a **`Matrix`** class that will be fundamental in manipulating the position and orientation of geometric objects. Finally, you will learn how to incorporate matrix objects into the rendering of an interactive 3D scene.

3.1 INTRODUCTION TO VECTORS AND MATRICES

When creating animated and interactive graphics applications, you will frequently want to transform sets of points defining the shape of geometric objects. You may want to translate the object to a new position, rotate the object to a new orientation, or scale the object to a new size. Additionally, you will want to project the viewable region of the scene into the space that is rendered by OpenGL. This will frequently be a perspective projection, where objects appear smaller the further away they are from the virtual camera. These ideas were first introduced in Chapter 1, in the discussion of the graphics pipeline and geometry processing; these calculations take place in a vertex shader. In Chapter 2, you learned how to work with points (using the vector data types **`vec3`** and **`vec4`**), and you implemented a transformation (two-dimensional translation). In this chapter, you will learn about a data structure called a *matrix*—a rectangular or two-dimensional

DOI: 10.1201/9781003181378-3

array of numbers—that is capable of representing all of the different types of transformations you will need in three-dimensional graphics. For these matrix-based transformations, you will learn how to

- apply a transformation to a point
- combine multiple transformations into a single transformation
- create a matrix corresponding to a given description of a transformation

3.1.1 Vector Definitions and Operations

In the previous chapter, you worked with vector data types, such as **`vec3`**, which are data structures whose components are floating-point numbers. In this section, you will learn about vectors from a mathematical point of view. For simplicity, the topics in this section will be introduced in a two-dimensional context. In a later section, you will learn how each of the concepts is generalized to higher-dimensional settings.

A *coordinate system* is defined by an origin point and the orientation and scale of a set of coordinate axes. A *point* $P = (x, y)$ refers to a location in space, specified relative to a coordinate system. A point is typically drawn as a dot, as illustrated by Figure 3.1. Note that in the diagram, the axes are oriented so that the *x*-axis is pointing to the right, and the *y*-axis is pointing upward; the direction of the arrow represents the direction in which the values increase. The orientation of the coordinate axes is somewhat arbitrary, although having the *x*-axis point to the right is a standard choice. In most two-dimensional mathematics diagrams, as well as OpenGL, the *y*-axis points upward. However, in many two-dimensional

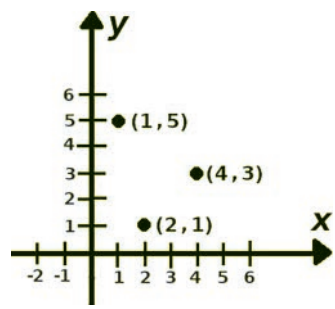

FIGURE 3.1 A collection of points.

computer graphics applications, the origin is placed in the top-left corner of the screen or drawing area, and the y-axis points downward. It is always important to know the orientation of the coordinate system you are working in!

A *vector* $\boldsymbol{v} = \langle m, n \rangle$ refers to a *displacement*—an amount of change in each coordinate—and is typically drawn as an arrow pointing along the direction of displacement, as illustrated by Figure 3.2. The point where the arrow begins is called the *initial point* or *tail*; the point where the arrow ends is called the *terminal point* or *head*, and indicates the result when the displacement has been applied to the initial point. The distance between the initial and terminal points of the vector is called its *length* or *magnitude*, and can be calculated from the components of the vector. Vectors are not associated with any particular location in space; the same vector may exist at different locations. A vector whose initial point is located at the origin (when a coordinate system is specified) is said to be in *standard position*.

To further emphasize the difference between location and displacement, consider navigating in a city whose roads are arranged as lines in a grid. If you were to ask someone to meet you at the intersection of 42nd Street and 5th Avenue, this refers to a particular location and corresponds to a point. If you were to ask someone to travel five blocks north and three blocks east from their current position, this refers to a displacement (a change in their position) and corresponds to a vector.

Each of these mathematical objects—points and vectors—are represented by a list of numbers, but as explained above, they have different geometric interpretations. In this book, notation will be used to quickly distinguish these objects. When referring to vectors, bold lowercase letters will be used for variables, and angle brackets will be used when listing components, as in $\boldsymbol{v} = \langle 1, 4 \rangle$. In contrast, when referring to points, regular (that is, non-bold) uppercase letters will be used for variables,

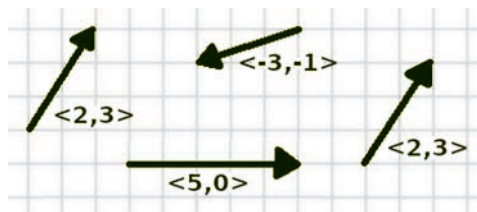

FIGURE 3.2 A collection of vectors.

and standard parentheses will be used when listing components, as in $P = (3, 2)$. Individual numbers (that are not part of a point or vector) are often called *scalars* in this context (to clearly distinguish them from points and vectors) and will be represented with regular lowercase letters, as in $x = 5$. Additionally, subscripted variables will sometimes be used when writing the components of a point or vector, and these subscripts may be letters or numbers, as in $P = (p_x, p_y)$ or $P = (p_1, p_2)$ for points, and $\boldsymbol{v} = \langle v_x, v_y \rangle$ or $\boldsymbol{v} = \langle v_1, v_2 \rangle$ for vectors.

While many algebraic operations can be defined on combinations of points and vectors, those with a geometric interpretation will be most significant in what follows. The first such operation is vector addition, which combines two vectors $\boldsymbol{v} = \langle v_1, v_2 \rangle$ and $\boldsymbol{w} = \langle w_1, w_2 \rangle$ and produces a new vector according to the following formula:

$$\boldsymbol{v} + \boldsymbol{w} = \langle v_1, v_2 \rangle + \langle w_1, w_2 \rangle = \langle v_1 + w_1, v_2 + w_2 \rangle$$

For example, $\langle 2, 5 \rangle + \langle 4, -2 \rangle = \langle 6, 3 \rangle$. Geometrically, this corresponds to displacement along the vector $\boldsymbol{v}$, followed by displacement along the vector $\boldsymbol{w}$; this can be visualized by aligning the terminal point of $\boldsymbol{v}$ with the initial point of $\boldsymbol{w}$. The result is a new vector $\boldsymbol{u} = \boldsymbol{v} + \boldsymbol{w}$ that shares the initial point of $\boldsymbol{v}$ and the terminal point of $\boldsymbol{w}$, as illustrated in Figure 3.3.

There is also a geometric interpretation for adding a vector $\boldsymbol{v}$ to a point P with the same algebraic formula; this corresponds to translating a point to a new location, which yields a new point. Align the vector $\boldsymbol{v}$ so its initial point is at location P, and the result is the point Q located at the terminal point of $\boldsymbol{v}$; this is expressed by the equation $P + \boldsymbol{v} = Q$. For example, $(2, 2) + \langle 1, 3 \rangle = (3, 5)$, as illustrated in Figure 3.4.

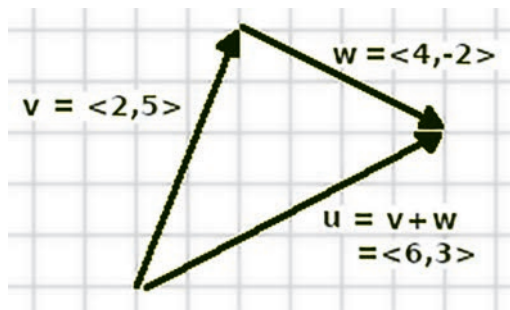

FIGURE 3.3 Vector addition.

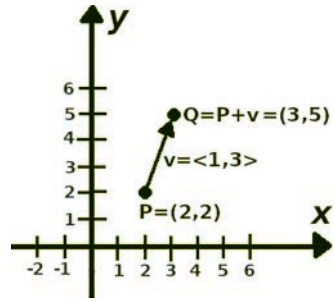

FIGURE 3.4 Adding a vector to a point.

The sum of two points does not have any clear geometric interpretation, but the difference of two points does. Rearranging the equation $P+\boldsymbol{v}=Q$ yields the equation $\boldsymbol{v}=Q-P$, which can be thought of as calculating the displacement vector between two points by subtracting their coordinates.

The componentwise product of two points or two vectors does not have any clear geometric interpretation. However, multiplying a vector $\boldsymbol{v}$ by a scalar c does. This operation is called *scalar multiplication* and is defined by

$$c\cdot\boldsymbol{v}=c\cdot\langle v_1, v_2\rangle=\langle c\cdot v_1, c\cdot v_2\rangle$$

For example, $2\cdot\langle 3, 2\rangle=\langle 6, 4\rangle$. Geometrically, this corresponds to scaling the vector; the length of $\boldsymbol{v}$ is multiplied by a factor of $|c|$ (the absolute value of c), and the direction is reversed when $c<0$. Figure 3.5 illustrates scaling a vector $\boldsymbol{v}$ by various amounts.

The operations of vector addition and scalar multiplication will be particularly important in what follows. Along these lines, consider the vectors

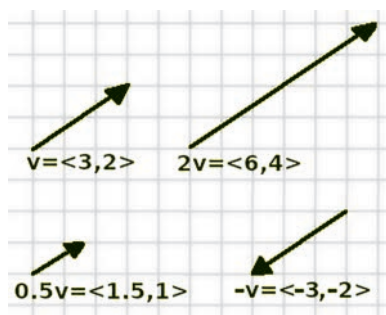

FIGURE 3.5 Scalar multiplication.

$\boldsymbol{i} = \langle 1, 0 \rangle$ and $\boldsymbol{j} = \langle 0, 1 \rangle$. Any other vector $\boldsymbol{v} = \langle x, y \rangle$ can be written in terms of $\boldsymbol{i}$ and $\boldsymbol{j}$ using vector multiplication and scalar multiplication in exactly one way, as follows:

$$\boldsymbol{v} = \langle x, y \rangle = \langle x, 0 \rangle + \langle 0, y \rangle = x \cdot \langle 1, 0 \rangle + y \cdot \langle 0, 1 \rangle = x \cdot \boldsymbol{i} + y \cdot \boldsymbol{j}$$

Any set of vectors with these properties is called a *basis*. There are many sets of basis vectors, but since $\boldsymbol{i}$ and $\boldsymbol{j}$ are in a sense the simplest such vectors, they are called the *standard basis* (for two-dimensional space).

3.1.2 Linear Transformations and Matrices

The main goal of this chapter is to design and create functions that transform sets of points or vectors in certain geometric ways. These are often called vector functions, to distinguish them from functions that have scalar valued input and output. They may also be called transformations to emphasize their geometric interpretation. These functions may be written as $F\big(\,(p_1, p_2)\,\big) = (q_1, q_2)$, or $F\big(\langle v_1, v_2 \rangle\big) = \langle w_1, w_2 \rangle$, or occasionally vectors will be written in column form, as

$$F\left(\begin{bmatrix} v_1 \\ v_2 \end{bmatrix}\right) = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

The latter of these three expressions is most commonly used in a traditional mathematical presentation, but the alternative expressions are often used for writing mathematics within sentences.

Vector functions with differing levels of complexity are easy to write. As simple examples, one may consider the *zero function*, where the output is always the zero vector:

$$F\big(\langle v_1, v_2 \rangle\big) = \langle 0, 0 \rangle$$

There is also the *identity function*, where the output is always equal to the input:

$$F\big(\langle v_1, v_2 \rangle\big) = \langle v_1, v_2 \rangle$$

At the other extreme, one could invent all manner of arbitrary complicated expressions, such as

$$F\left(\langle v_1, v_2\rangle\right)=\left\langle v_1-3\cdot\cos(3v_2),\,(v_1)^7+\ln\left(|v_2|\right)+\pi\right\rangle$$

Most of which lack any geometrical significance whatsoever.

The key is to find a relatively simple class of vector functions which can perform the types of geometric transformations described at the beginning of this chapter. It will be particularly helpful to choose to work with vector functions that can be simplified in some useful way. One such set of functions are *linear functions* or *linear transformations*, which are functions F that satisfy the following two equations relating to scalar multiplication and vector addition: for any scalar c and vectors $\boldsymbol{v}$ and $\boldsymbol{w}$,

$$F(c\cdot\boldsymbol{v})=c\cdot F(\boldsymbol{v})$$

$$F(\boldsymbol{v}+\boldsymbol{w})=F(\boldsymbol{v})+F(\boldsymbol{w})$$

One advantage to working with such functions involves the standard basis vectors $\boldsymbol{i}$ and $\boldsymbol{j}$: if F is a linear function and the values of $F(\boldsymbol{i})$ and $F(\boldsymbol{j})$ are known, then it is possible to calculate the value of $F(\boldsymbol{v})$ for any vector $\boldsymbol{v}$, even when a general formula for the function F is not given. For example, assume that F is a linear function, $F(\boldsymbol{i})=\langle 1,2\rangle$ and $F(\boldsymbol{j})=\langle 3,1\rangle$. Then by using the equations that linear functions satisfy, the value of $F\left(\langle 4,5\rangle\right)$ can be calculated as follows:

$$\begin{aligned}
F\left(\langle 4,5\rangle\right) &= F\left(\langle 4,0\rangle+\langle 0,5\rangle\right)\\
&= F\left(\langle 4,0\rangle\right)+F\left(\langle 0,5\rangle\right)\\
&= F\left(4\cdot\langle 1,0\rangle\right)+F\left(5\cdot\langle 0,1\rangle\right)\\
&= 4\cdot F\left(\langle 1,0\rangle\right)+5\cdot F\left(\langle 0,1\rangle\right)\\
&= 4\cdot F(\boldsymbol{i})+5\cdot F(\boldsymbol{j})\\
&= 4\cdot\langle 1,2\rangle+5\cdot\langle 3,1\rangle\\
&= \langle 4,8\rangle+\langle 15,5\rangle\\
&= \langle 19,13\rangle
\end{aligned}$$

In a similar way, the general formula for $F\left(\langle x\ \ y\rangle\right)$ for this example can be calculated as follows:

$$
\begin{aligned}
F(\langle x \;\; y\rangle) &= F(\langle x, 0\rangle) + (\langle 0, y\rangle) \\
&= F(\langle x, 0\rangle) + F(\langle 0, y\rangle) \\
&= F(x \cdot \langle 1, 0\rangle) + F(y \cdot \langle 0, 1\rangle) \\
&= x \cdot F(\langle 1, 0\rangle) + y \cdot F(\langle 0, 1\rangle) \\
&= x \cdot F(\boldsymbol{i}) + y \cdot F(\boldsymbol{j}) \\
&= x \cdot \langle 1, 2\rangle + y \cdot \langle 3, 1\rangle \\
&= \langle x, 2x\rangle + \langle 3y, y\rangle \\
&= \langle x + 3y, 2x + y\rangle
\end{aligned}
$$

In fact, the same line of reasoning establishes the most general case: if F is a linear function, where $F(\boldsymbol{i}) = a, c$ and $F(\boldsymbol{j}) = b,\ d$, then the formula for the function F is $F(\langle x, y\rangle) = \langle a \cdot x + b \cdot y, c \cdot x + d \cdot y\rangle$, since

$$
\begin{aligned}
F(\langle x \;\; y\rangle) &= F(\langle x, 0\rangle + \langle 0, y\rangle) \\
&= F(\langle x, 0\rangle) + F(\langle 0, y\rangle) \\
&= F(x \cdot \langle 1, 0\rangle) + F(y \cdot \langle 0, 1\rangle) \\
&= x \cdot F(\langle 1, 0\rangle) + y \cdot F(\langle 0, 1\rangle) \\
&= x \cdot F(\boldsymbol{i}) + y \cdot F(\boldsymbol{j}) \\
&= x \cdot \langle a, c\rangle + y \cdot \langle b, d\rangle \\
&= \langle a \cdot x, c \cdot x\rangle + \langle b \cdot y, d \cdot y\rangle \\
&= \langle a \cdot x + b \cdot y, c \cdot x + d \cdot y\rangle
\end{aligned}
$$

In addition to these useful algebraic properties, it is possible to visualize the geometric effect of a linear function on the entire space. To begin, consider a unit square consisting of the points $\langle u, v\rangle = u \cdot \boldsymbol{i} + v \cdot \boldsymbol{j}$, where $0 \le u \le 1$ and $0 \le v \le 1$, the dot-shaded square labeled as S on the left side of Figure 3.6.

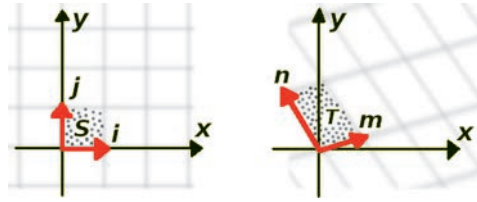

FIGURE 3.6 The geometric effects of a linear transformation.

Assume F is a linear function with $F(\boldsymbol{i})=\boldsymbol{m}$ and $F(\boldsymbol{j})=\boldsymbol{n}$. Then, the set of points in S is transformed to the set of points that can be written as

$$F(u\cdot\boldsymbol{i}+v\cdot\boldsymbol{j})=u\cdot F(\boldsymbol{i})+v\cdot F\langle\boldsymbol{j}\rangle=u\cdot\boldsymbol{m}+v\cdot\boldsymbol{n}$$

This area is indicated by the dot-shaded parallelogram labeled as T on the right side of Figure 3.6. Similarly, the function F transforms each square region on the left of Figure 3.6 into a parallelogram shaped region on the right of Figure 3.6.

The formula for a linear function F can be written in column form as

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)=\begin{bmatrix} a\cdot x+b\cdot y \\ c\cdot x+d\cdot y \end{bmatrix}$$

It is useful to think of the vector $\langle x, y\rangle$ as being operated on by the set of numbers a, b, c, d, which naturally leads us to a particular mathematical notation: these numbers can be grouped into a rectangular array of numbers called a *matrix*, typically enclosed within square brackets, and the function F can be rewritten as

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)=\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}$$

In accordance with this notation, the *product* of a matrix and a vector is defined by the following equation:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}=\begin{bmatrix} a\cdot x+b\cdot y \\ c\cdot x+d\cdot y \end{bmatrix}$$

For example, consider the following linear function:

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)=\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}$$

Then, $F(\langle 5,6\rangle)$ can be calculated as follows:

$$F\left(\begin{bmatrix} 5 \\ 6 \end{bmatrix}\right)=\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix}\begin{bmatrix} 5 \\ 6 \end{bmatrix}=\begin{bmatrix} 2\cdot 5+3\cdot 6 \\ 4\cdot 5+1\cdot 6 \end{bmatrix}=\begin{bmatrix} 28 \\ 26 \end{bmatrix}$$

Similarly, to calculate $F(\langle -3,1\rangle)$,

$$F\left(\begin{bmatrix} -3 \\ 1 \end{bmatrix}\right)=\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix}\begin{bmatrix} -3 \\ 1 \end{bmatrix}=\begin{bmatrix} 2\cdot(-3)+3\cdot 1 \\ 4\cdot(-3)+1\cdot 1 \end{bmatrix}=\begin{bmatrix} -3 \\ -11 \end{bmatrix}$$

Once again, it will be helpful to use notation to distinguish between matrices and other types of mathematical objects (scalars, points, and vectors). When referring to a matrix, bold uppercase letters will be used for variables, and square brackets will be used to enclose the grid of numbers or variables. This notation can be used to briefly summarize the previous observation: if F is a linear function, then F can be written in the form $F(\boldsymbol{v})=\boldsymbol{A}\cdot\boldsymbol{v}$ for some matrix $\boldsymbol{A}$. Conversely, one can also show that if F is a vector function defined by matrix multiplication, that is, if $F(\boldsymbol{v})=\boldsymbol{A}\cdot\boldsymbol{v}$, then F also satisfies the equations that define a linear function; this can be verified by straightforward algebraic calculations. Therefore, these two descriptions of vector functions—those that are linear and those defined by matrix multiplication—are equivalent; they define precisely the same set of functions.

Two of the vector functions previously mentioned can be represented using matrix multiplication: the zero function can be written as

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)=\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}=\begin{bmatrix} 0\cdot x+0\cdot y \\ 0\cdot x+0\cdot y \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

while the identity function can be written as

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)=\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}=\begin{bmatrix} 1\cdot x+0\cdot y \\ 0\cdot x+1\cdot y \end{bmatrix}=\begin{bmatrix} x \\ y \end{bmatrix}$$

The matrix in the definition of the identity function is called the *identity matrix* and appears in a variety of contexts, as you will see.

When introducing notation for vectors, it was indicated that subscripted variables will sometimes be used for the components of a vector. When working with a matrix, *double* subscripted variables will sometimes be used for its components; the subscripts will indicate the position (row and column) of the component in the matrix. For example, the variable a_{12} will refer to the entry in row 1 and column 2 of the matrix $\boldsymbol{A}$, and in general, a_{mn} will refer to the entry in row m and column n of the matrix $\boldsymbol{A}$. The contents of the entire matrix $\boldsymbol{A}$ can be written as

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

At this point, you know how to apply a linear function (written in matrix notation) to a point or a vector. In many contexts (and in computer graphics in particular), you will want to apply multiple functions to a set of points. Given two functions F and G, a new function H can be created by defining $H(\boldsymbol{v}) = F\big(G(\boldsymbol{v})\big)$; the function H is called the *composition* of F and G. As it turns out, if F and G are linear functions, then H will be a linear function as well. This can be verified by checking that the two linearity equations hold for H, making use of the fact that the linearity equations are true for the functions F and G by assumption. The derivation relating to vector addition is as follows:

$$\begin{aligned} H(\boldsymbol{v}+\boldsymbol{w}) &= F\big(G(\boldsymbol{v}+\boldsymbol{w})\big) \\ &= F\big(G(\boldsymbol{v})+G(\boldsymbol{w})\big) \\ &= F\big(G(\boldsymbol{v})\big)+F\big(G(\boldsymbol{w})\big) \\ &= H(\boldsymbol{v})+H(\boldsymbol{w}) \end{aligned}$$

The derivation relating to scalar multiplication is as follows:

$$\begin{aligned} H(c\cdot\boldsymbol{v}) &= F\big(G(c\cdot\boldsymbol{v})\big) \\ &= F\big(c\cdot G(\boldsymbol{v})\big) \\ &= c\cdot F\big(G(\boldsymbol{v})\big) \\ &= c\cdot H(\boldsymbol{v}) \end{aligned}$$

The reason this is significant is that given two linear functions—each of which can be represented by a matrix, as previously observed—their composition can be represented by a *single* matrix, since the composition is also a linear function. By repeatedly applying this reasoning, it follows that the composition of *any number* of linear functions can be represented by a *single* matrix. This immediately leads to the question: how can the matrix corresponding to the composition be calculated? Algebraically, given two transformations $F(\boldsymbol{v}) = \boldsymbol{A} \cdot \boldsymbol{v}$ and $G(\boldsymbol{v}) = \boldsymbol{B} \cdot \boldsymbol{v}$, the goal is to find a matrix $\boldsymbol{C}$ such that the transformation $H(\boldsymbol{v}) = \boldsymbol{C} \cdot \boldsymbol{v}$ is equal to $F(G(\boldsymbol{v})) = \boldsymbol{A} \cdot (\boldsymbol{B} \cdot \boldsymbol{v})$ for all vectors $\boldsymbol{v}$. In this case, one writes $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$. This operation is referred to as *matrix multiplication*, and $\boldsymbol{C}$ is called the *product* of the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$.

The formula for matrix multiplication may be deduced by computing both sides of the equation $\boldsymbol{C} \cdot \boldsymbol{v} = \boldsymbol{A} \cdot (\boldsymbol{B} \cdot \boldsymbol{v})$ and equating the coefficients of x and y on either side of the equation. Expanding $\boldsymbol{C} \cdot \boldsymbol{v}$ yields

$$\boldsymbol{C} \cdot \boldsymbol{v} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_{11} \cdot x + c_{12} \cdot y \\ c_{21} \cdot x + c_{22} \cdot y \end{bmatrix}$$

Similarly, expanding $\boldsymbol{A} \cdot (\boldsymbol{B} \cdot \boldsymbol{v})$ yields

$$\begin{aligned} \boldsymbol{A} \cdot (\boldsymbol{B} \cdot \boldsymbol{v}) &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \left(\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \right) \\ &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} \cdot x + b_{12} \cdot y \\ b_{21} \cdot x + b_{22} \cdot y \end{bmatrix} \\ &= \begin{bmatrix} a_{11}(b_{11} \cdot x + b_{12} \cdot y) + a_{12}(b_{21} \cdot x + b_{22} \cdot y) \\ a_{21}(b_{11} \cdot x + b_{12} \cdot y) + a_{22}(b_{21} \cdot x + b_{22} \cdot y) \end{bmatrix} \\ &= \begin{bmatrix} (a_{11} \cdot b_{11} + a_{12} \cdot b_{21}) \cdot x + (a_{11} \cdot b_{12} + a_{12} \cdot b_{22}) \cdot y \\ (a_{21} \cdot b_{11} + a_{22} \cdot b_{21}) \cdot x + (a_{21} \cdot b_{12} + a_{22} \cdot b_{22}) \cdot y \end{bmatrix} \end{aligned}$$

Thus, matrix multiplication $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ is defined as

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} (a_{11} \cdot b_{11} + a_{12} \cdot b_{21}) + (a_{11} \cdot b_{12} + a_{12} \cdot b_{22}) \\ (a_{21} \cdot b_{11} + a_{22} \cdot b_{21}) + (a_{21} \cdot b_{12} + a_{22} \cdot b_{22}) \end{bmatrix}$$

This formula can be written more simply using a vector operation called the *dot product*. Given vectors $\boldsymbol{v} = \langle v_1 , v_2 \rangle$ and $\boldsymbol{w} = \langle w_1 , w_2 \rangle$, the dot product $d = \boldsymbol{v} \bullet \boldsymbol{w}$ is a (scalar) number, defined by

$$d = \boldsymbol{v} \bullet \boldsymbol{w} = \langle v_1, v_2 \rangle \bullet \langle w_1 , w_2 \rangle = v_1 \cdot w_1 + v_2 \cdot w_2$$

As an example of a dot product calculation, consider

$$\langle 3, 4 \rangle \bullet \langle 7, 5 \rangle = 3 \cdot 7 + 4 \cdot 5 = 41$$

To restate the definition of matrix multiplication: partition the matrix $\boldsymbol{A}$ into row vectors and the matrix $\boldsymbol{B}$ into column vectors. The entry c_{mn} of the product matrix $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ is equal to the dot product of row vector m from matrix $\boldsymbol{A}$ (denoted by $\boldsymbol{a_m}$) and column vector n from matrix $\boldsymbol{B}$ (denoted by $\boldsymbol{b_n}$), as illustrated in the following formula, where partitions are indicated by dashed lines.

$$\begin{aligned}
\boldsymbol{A} \cdot \boldsymbol{B} &= \left[\begin{array}{cc} a_{11} & a_{12} \\ \hdashline a_{21} & a_{22} \end{array}\right] \cdot \left[\begin{array}{c:c} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array}\right] \\
&= \left[\begin{array}{c} \boldsymbol{a}_1 \\ \hdashline \boldsymbol{a}_2 \end{array}\right] \cdot \left[\begin{array}{c:c} \boldsymbol{b}_1 & \boldsymbol{b}_2 \end{array}\right] \\
&= \begin{bmatrix} \boldsymbol{a}_1 \bullet \boldsymbol{b}_1 & \boldsymbol{a}_1 \bullet \boldsymbol{b}_2 \\ \boldsymbol{a}_2 \bullet \boldsymbol{b}_1 & \boldsymbol{a}_2 \bullet \boldsymbol{b}_2 \end{bmatrix} \\
&= \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}
\end{aligned}$$

As an example of a matrix multiplication computation, consider

$$\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} 9 & 8 \\ 7 & 6 \end{bmatrix} = \begin{bmatrix} \langle 2, 3 \rangle \bullet \langle 9, 7 \rangle & \langle 2, 3 \rangle \bullet \langle 8, 6 \rangle \\ \langle 4, 5 \rangle \bullet \langle 9, 7 \rangle & \langle 4, 5 \rangle \bullet \langle 8, 6 \rangle \end{bmatrix}$$

$$= \begin{bmatrix} (2 \cdot 9 + 3 \cdot 7) & (2 \cdot 8 + 3 \cdot 6) \\ (4 \cdot 9 + 5 \cdot 7) & (4 \cdot 8 + 5 \cdot 6) \end{bmatrix}$$

$$= \begin{bmatrix} 39 & 34 \\ 71 & 62 \end{bmatrix}$$

In general, matrix multiplication can quickly become tedious, and so a software package is typically used to handle these and other matrix-related calculations.

It is important to note that, in general, matrix multiplication is not a commutative operation. Given matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, the product $\boldsymbol{A}\cdot\boldsymbol{B}$ is usually not equal to the product $\boldsymbol{B}\cdot\boldsymbol{A}$. For example, calculating the product of the matrices from the previous example in the opposite order yields

$$\begin{bmatrix} 9 & 8 \\ 7 & 6 \end{bmatrix}\cdot\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}=\begin{bmatrix} \langle 9,8\rangle\bullet\langle 2,4\rangle & \langle 9,8\rangle\bullet\langle 2,5\rangle \\ \langle 7,2\rangle\bullet\langle 6,4\rangle & \langle 7,6\rangle\bullet\langle 3,5\rangle \end{bmatrix}$$

$$=\begin{bmatrix} (9\cdot 2+8\cdot 4) & (9\cdot 3+8\cdot 5) \\ (7\cdot 2+6\cdot 4) & (7\cdot 3+6\cdot 5) \end{bmatrix}$$

$$=\begin{bmatrix} 50 & 67 \\ 38 & 51 \end{bmatrix}$$

This fact has a corresponding geometric interpretation as well: the order in which geometric transformations are performed makes a difference. For example, let T represent translation by $\langle 1, 0\rangle$, and let R represent rotation around the origin by 90°. If P denotes the point $P=(2, 0)$, then $R(T(P))=R(3,0)=(0,3)$, while $T(R(P))=T(0,2)=(1,2)$, as illustrated in Figure 3.7. Thus, $R(T(P))$ does not equal $T(R(P))$.

The identity matrix $\boldsymbol{I}$, previously mentioned, is the matrix

$$\boldsymbol{I}=\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The identity matrix has multiplication properties similar to those of the number 1 in ordinary multiplication. Just as for any number x, it is true that $1\cdot x=x$ and $x\cdot 1=x$, for any matrix $\boldsymbol{A}$ it can be shown with algebra that $\boldsymbol{I}\cdot\boldsymbol{A}=\boldsymbol{A}$ and $\boldsymbol{A}\cdot\boldsymbol{I}=\boldsymbol{A}$. Because of these properties, both 1 and $\boldsymbol{I}$ are called *identity elements* in their corresponding mathematical contexts. Similarly, the identity function is the identity element in the context of function composition. Thinking of vector functions as geometric transformations, the identity function does not change the location of any points; the geometric transformations translation by $\langle 0, 0\rangle$, rotation around the origin by

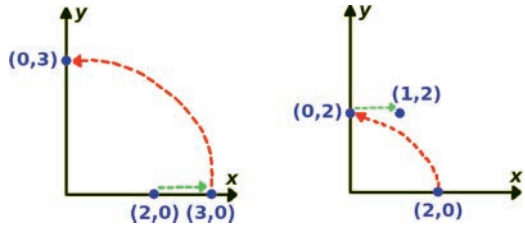

FIGURE 3.7 Geometric transformations (translation and rotation) are not commutative.

0 degrees, and scaling all components by a factor of 1 are all equivalent to the identity function.

The concept of identity elements leads to the concept of *inverse* elements. In a given mathematical context, combining an object with its inverse yields the identity element. For example, a number x multiplied by its inverse equals 1; a function composed with its inverse function yields the identity function. Analogously, a matrix multiplied by its inverse matrix results in the identity matrix. Symbolically, the inverse of a matrix $\boldsymbol{A}$ is a matrix $\boldsymbol{M}$ such that $\boldsymbol{A}\cdot\boldsymbol{M}=\boldsymbol{I}$ and $\boldsymbol{M}\cdot\boldsymbol{A}=\boldsymbol{I}$. The inverse of the matrix $\boldsymbol{A}$ is typically written using the notation $\boldsymbol{A}^{-1}$. Using a fair amount of algebra, one can find a formula for the inverse of the matrix $\boldsymbol{A}$ by solving the equation $\boldsymbol{A}\cdot\boldsymbol{M}=\boldsymbol{I}$ for the entries of $\boldsymbol{M}$:

$$\boldsymbol{A}\cdot\boldsymbol{M}=\begin{bmatrix} a & b \\ c & d \end{bmatrix}\cdot\begin{bmatrix} m & n \\ p & q \end{bmatrix}=\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}=\boldsymbol{I}$$

Solving this equation first involves calculating the product on the left-hand side of the equation and setting each entry of the resulting matrix equal to the corresponding entry in the identity matrix. This yields a system of four equations with four unknowns (the entries of $\boldsymbol{M}$). Solving these four equations yields the following formula for the inverse of a 2-by-2 matrix:

$$\boldsymbol{M}=\boldsymbol{A}^{-1}=\begin{bmatrix} d/(ad-bc) & -b/(ad-bc) \\ -c/(ad-bc) & a/(ad-bc) \end{bmatrix}$$

The value $(a\cdot d-b\cdot c)$ appearing in the denominator of each entry of the inverse matrix is called the *determinant* of the matrix $\boldsymbol{A}$. If this value is

equal to 0, then the fractions are all undefined, and the inverse of the matrix $\boldsymbol{A}$ does not exist. Analogous situations, in which certain elements do not have inverse elements, arise in other mathematical contexts. For example, in ordinary arithmetic, the number $x = 0$ does not have a multiplicative inverse, as nothing times 0 equals the identity element 1.

As may be expected, if an invertible vector function F can be represented with matrix multiplication as $F(\boldsymbol{v}) = \boldsymbol{A} \cdot \boldsymbol{v}$, then the inverse function G can be represented with matrix multiplication by the inverse of $\boldsymbol{A}$, as $G(\boldsymbol{v}) = \boldsymbol{A}^{-1} \cdot \boldsymbol{v}$, since

$$F\big(G(v)\big) = \boldsymbol{A} \cdot \boldsymbol{A}^{-1} \cdot \boldsymbol{v} = \boldsymbol{I} \cdot \boldsymbol{v} = \boldsymbol{v}$$

$$G\big(F(v)\big) = \boldsymbol{A}^{-1} \cdot \boldsymbol{A} \cdot \boldsymbol{v} = \boldsymbol{I} \cdot \boldsymbol{v} = \boldsymbol{v}$$

Once again, thinking of vector functions as geometric transformations, the inverse of a function performs a transformation that is in some sense the "opposite" or "reverse" transformation. For example, the inverse of translation by $\langle m, n \rangle$ is translation by $\langle -m, -n \rangle$; the inverse of clockwise rotation by an angle a is counterclockwise rotation by an angle a (which is equivalent to clockwise rotation by an angle $-a$); the inverse of scaling the components of a vector by the values r and s is scaling the components by the values $1/r$ and $1/s$.

3.1.3 Vectors and Matrices in Higher Dimensions

All of these vector and matrix concepts can be generalized to three, four, and even higher dimensions. In this section, these concepts will be restated in a three-dimensional context. The generalization to four-dimensional space follows the same algebraic pattern. Four-dimensional vectors and matrices are used quite frequently in computer graphics, for reasons discussed later in this chapter.

Three-dimensional coordinate systems are drawn using *xyz*-axes, where each axis is perpendicular to the other two. Assuming that the axes are oriented as in Figure 3.1, so that the plane spanned by the x and y axes are aligned with the window used to display graphics, there are two possible directions for the (positive) z-axis: either pointing towards the viewer or away from the viewer. These two systems are called *right-handed* and *left-handed coordinate systems*, respectively, so named due to the hand-based mnemonic rule used to remember the orientation. To visualize the relative orientation of the axes in a right-handed coordinate system, using

FIGURE 3.8 Using a right hand to determine the orientation of *xyz*-axes.

your right hand, imagine your index finger pointing along the *x*-axis and your middle finger perpendicular to this (in the direction the palm of your hand is facing) pointing along the *y*-axis. Then, your extended thumb will be pointing in the direction of the *z*-axis; this is illustrated in Figure 3.8. If the *z*-axis were pointing in the opposite direction, this would correspond to a left-handed coordinate system, and indeed, this would be the orientation indicated by carrying out the steps above with your left hand. Some descriptions of the right-hand rule, instead of indicating the directions of the *x* and *y* axes with extended fingers, will suggest curling the fingers of your hand in the direction from the *x*-axis to the *y*-axis; your extended thumb still indicates the direction of the *z*-axis, and the two descriptions have the same result.

In mathematics, physics, and computer graphics, it is standard practice to use a right-handed coordinate system, as shown on the left side of Figure 3.9. In computer graphics, the positive *z*-axis points directly at the viewer. Although the three axes are perpendicular to each other, when illustrated in this way, at first it may be difficult to see that the *z*-axis is perpendicular to the other axes. In this case, it may aid with visualization to imagine the *xyz*-axes as aligned with the edges of a cube hidden from view, illustrated with dashed lines as shown on the right side of Figure 3.9.

In three-dimensional space, points are written as $P = \left(p_x, p_y, p_z\right)$ or $P = \left(p_1, p_2, p_3\right)$, and vectors are written as $\boldsymbol{v} = \left\langle v_x, v_y, v_z \right\rangle$ or $\boldsymbol{v} = \langle v_1, v_2, v_3 \rangle$. Sometimes, for clarity, the components may be written as *x*, *y*, and *z* (and in four-dimensional space, the fourth component may be written as *w*). Vector addition is defined by

$$\boldsymbol{v} + \boldsymbol{w} = \langle v_1 , v_2, v_3 \rangle + \langle w_1, w_2, w_3 \rangle = \langle v_1 + w_1, v_2 + w_2, v_3 + w_3 \rangle$$

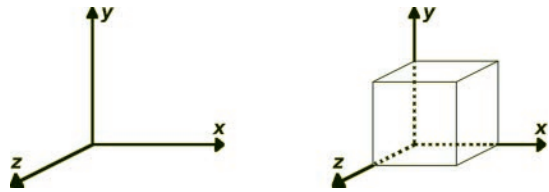

FIGURE 3.9 Coordinate axes in three dimensions.

while scalar multiplication is defined by

$$c \cdot \boldsymbol{v} = c \cdot \langle v_1, v_2, v_3 \rangle = \langle c \cdot v_1, c \cdot v_2, c \cdot v_3 \rangle$$

The standard basis for three-dimensional space consists of the vectors $\boldsymbol{i} = \langle 1, 0, 0 \rangle$, $\boldsymbol{j} = \langle 0, 1, 0 \rangle$, and $\boldsymbol{k} = \langle 0, 0, 1 \rangle$. Every vector $\boldsymbol{v} = \langle x, y, z \rangle$ can be written as a linear combination of these three vectors as follows:

$$\begin{aligned} v &= \langle x, y, z \rangle \\ &= \langle x, 0, 0 \rangle + \langle 0, y, 0 \rangle + \langle 0, 0, z \rangle \\ &= x \cdot \langle 1, 0, 0 \rangle + y \cdot \langle 0, 1, 0 \rangle + z \cdot \langle 0, 0, 1 \rangle \\ &= x \cdot \boldsymbol{i} + y \cdot \boldsymbol{j} + z \cdot \boldsymbol{k} \end{aligned}$$

The definition of a linear function is identical for vectors of any dimension, as it only involves the operations of vector addition and scalar multiplication; it does not reference the number of components of a vector at all:

$$F(c \cdot \boldsymbol{v}) = c \cdot F(\boldsymbol{v})$$

$$F(\boldsymbol{v} + \boldsymbol{w}) = F(\boldsymbol{v}) + F(\boldsymbol{w})$$

The values of a three-dimensional linear function can be calculated for any vector if the values of $F(\boldsymbol{i})$, $F(\boldsymbol{j})$, and $F(\boldsymbol{k})$ are known, and in general, such a function can be written in the following form:

$$F\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} a_{11} \cdot x + a_{12} \cdot y + a_{13} \cdot z \\ a_{21} \cdot x + a_{22} \cdot y + a_{23} \cdot z \\ a_{31} \cdot x + a_{32} \cdot y + a_{33} \cdot z \end{bmatrix}$$

As before, the coefficients of x, y, and z in the formula are typically grouped into a 3-by-3 matrix:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The matrix-vector product $\boldsymbol{A}\cdot\boldsymbol{v}$ is then defined as

$$\boldsymbol{A}\cdot\boldsymbol{v} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_{11}\cdot x + a_{12}\cdot y + a_{13}\cdot z \\ a_{21}\cdot x + a_{22}\cdot y + a_{23}\cdot z \\ a_{31}\cdot x + a_{32}\cdot y + a_{33}\cdot z \end{bmatrix}$$

Matrix multiplication is most clearly described using the dot product, which for three-dimensional vectors is defined as

$$d = \boldsymbol{v}\bullet\boldsymbol{w} = \langle v_1, y_2 \;\; y_3\rangle \cdot \langle w_1 \;\; w_2, w_3\rangle = v_1\cdot w_1 + v_2\cdot w_2 + v_3\cdot w_3$$

Matrix multiplication $\boldsymbol{A}\cdot\boldsymbol{B}$ can be calculated from partitioning the entries of the two matrices into vectors: each row of the first matrix ($\boldsymbol{A}$) is written as a vector, and each column of the second matrix ($\boldsymbol{B}$) is written as a vector. Then, the value in row m and column n of the product is equal to the dot product of row vector m from matrix $\boldsymbol{A}$ (denoted by $\boldsymbol{a}_{\mathbf{m}}$) and column vector n from matrix $\boldsymbol{B}$ (denoted by $\boldsymbol{b}_n$), as illustrated below.

$$\boldsymbol{A}\cdot\boldsymbol{B} = \left[\begin{array}{ccc} a_{11} & a_{12} & a_{13} \\ \hline a_{21} & a_{22} & a_{23} \\ \hline a_{33} & a_{32} & a_{33} \end{array}\right] \cdot \left[\begin{array}{c:c:c} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{33} & b_{32} & b_{33} \end{array}\right]$$

$$= \left[\begin{array}{c} \boldsymbol{a}_1 \\ \hline \boldsymbol{a}_2 \\ \hline \boldsymbol{a}_3 \end{array}\right] \cdot \left[\begin{array}{c:c:c} \boldsymbol{b}_1 & \boldsymbol{b}_2 & \boldsymbol{b}_3 \end{array}\right]$$

$$= \begin{bmatrix} \boldsymbol{a}_1\bullet\boldsymbol{b}_1 & \boldsymbol{a}_1\bullet\boldsymbol{b}_2 & \boldsymbol{a}_1\bullet\boldsymbol{b}_3 \\ \boldsymbol{a}_2\bullet\boldsymbol{b}_1 & \boldsymbol{a}_2\bullet\boldsymbol{b}_2 & \boldsymbol{a}_2\bullet\boldsymbol{b}_3 \\ \boldsymbol{a}_3\bullet\boldsymbol{b}_1 & \boldsymbol{a}_3\bullet\boldsymbol{b}_2 & \boldsymbol{a}_3\bullet\boldsymbol{b}_3 \end{bmatrix}$$

$$= \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The 3-by-3 identity matrix $\boldsymbol{I}$ has the following form:

$$\boldsymbol{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

As before, this identity matrix is defined by the equations $\boldsymbol{I} \cdot \boldsymbol{A} = \boldsymbol{A}$ and $\boldsymbol{A} \cdot \boldsymbol{I} = \boldsymbol{A}$ for any matrix $\boldsymbol{A}$. Similarly, the inverse of a matrix $\boldsymbol{A}$ (if it exists) is a matrix denoted by $\boldsymbol{A}^{-1}$, defined by the equations $\boldsymbol{A} \cdot \boldsymbol{A}^{-1} = \boldsymbol{I}$ and $\boldsymbol{A}^{-1} \cdot \boldsymbol{A} = \boldsymbol{I}$. The formula for the inverse of a 3-by-3 matrix in terms of its entries is quite tedious to write down, and as mentioned previously, a software package will be used to handle these calculations.

3.2 GEOMETRIC TRANSFORMATIONS

The previous section introduced linear functions: vector functions that satisfy the linearity equations, functions which can be written with matrix multiplication. It remains to show that the geometric transformations needed in computer graphics (translation, rotation, scaling, and perspective projections) are linear functions, and then, formulas must be found for the corresponding matrices. In particular, it must be possible to determine the entries of a matrix corresponding to a description of a transformation, such as "translate along the x direction by 3 units" or "rotate around the z-axis by 45°." Formulas for each type of transformation (in both two and three dimensions) will be derived next, in increasing order of difficulty: scaling, rotation, translation, and perspective projection. In the following sections, vectors will be drawn in standard position (the initial point of each vector will be at the origin), and vectors can be identified with their terminal points.

3.2.1 Scaling

A scaling transformation multiplies each component of a vector by a constant. For two-dimensional vectors, this has the following form (where r and s are constants):

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} r \cdot x \\ s \cdot y \end{bmatrix}$$

It can quickly be deduced and verified that this transformation can be expressed with matrix multiplication as follows:

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} r \cdot x \\ s \cdot y \end{bmatrix} = \begin{bmatrix} r & 0 \\ 0 & s \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

Similarly, the three-dimensional version of this transformation, where the z-component of a vector is scaled by a constant value t, is:

$$F\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} r \cdot x \\ s \cdot y \\ t \cdot z \end{bmatrix} = \begin{bmatrix} r & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & t \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Observe that if all the scaling constants are equal to 1, then the formula for the scaling matrix results in the identity matrix. This corresponds to the following pair of related statements: scaling the components of a vector by 1 does not change the value of the vector, just as multiplying a vector by the identity matrix does not change the value of the vector.

3.2.2 Rotation

In two dimensions, a rotation transformation rotates vectors by a constant amount around the origin point. Unlike the case with the scaling transformation, it is not immediately clear how to write a formula for a rotation function $F(\boldsymbol{v})$ or whether rotation transformations can even be calculated with matrix multiplication. To establish this fact, it suffices to show that rotation is a linear transformation that it satisfies the two linearity equations. An informal geometric argument will be presented for each equation.

To see that $F(c \cdot \boldsymbol{v}) = c \cdot F(\boldsymbol{v})$, begin by considering the endpoint of the vector $\boldsymbol{v}$, and assume that this point is at a distance d from the origin. When multiplying $\boldsymbol{v}$ by c, the resulting vector has the same direction, and the endpoint is now at a distance $c \cdot d$ from the origin. Note that rotation transformations fix the origin point and do not change the distance of a point from the origin. Applying the rotation transformation F to the

vectors $\boldsymbol{v}$ and $c \cdot \boldsymbol{v}$ yields the vectors $F(\boldsymbol{v})$ and $F(c \cdot \boldsymbol{v})$; these vectors have the same direction, and their endpoints have the same distances from the origin: d and $c \cdot d$, respectively. However, the vector $c \cdot F(\boldsymbol{v})$ is also aligned with $F(\boldsymbol{v})$ and its endpoint has distance $c \cdot d$ from the origin. Therefore, the endpoints of $F(c \cdot \boldsymbol{v})$ and $c \cdot F(\boldsymbol{v})$ must be at the same position, and thus, $F(c \cdot \boldsymbol{v}) = c \cdot F(\boldsymbol{v})$. This is illustrated in Figure 3.10.

To see that $F(\boldsymbol{v} + \boldsymbol{w}) = F(\boldsymbol{v}) + F(\boldsymbol{w})$, begin by defining $\boldsymbol{u} = \boldsymbol{v} + \boldsymbol{w}$, and let $\boldsymbol{o}$ represent the origin. Due to the nature of vector addition, the endpoints of $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$, together with $\boldsymbol{o}$, form the vertices of a parallelogram. Applying the rotation transformation F to this parallelogram yields a parallelogram M whose vertices are $\boldsymbol{o}$ and the endpoints of the vectors $F(\boldsymbol{u})$, $F(\boldsymbol{v})$, and $F(\boldsymbol{w})$. Again, due to the nature of vector addition, the endpoints of $F(\boldsymbol{v})$, $F(\boldsymbol{w})$, and $F(\boldsymbol{v}) + F(\boldsymbol{w})$, together with $\boldsymbol{o}$, form the vertices of a parallelogram N. Since parallelograms M and N have three vertices in common, their fourth vertex must also coincide, from which it follows that $F(\boldsymbol{v}) + F(\boldsymbol{w}) = F(\boldsymbol{u}) = F(\boldsymbol{v} + \boldsymbol{w})$. This is illustrated in Figure 3.11.

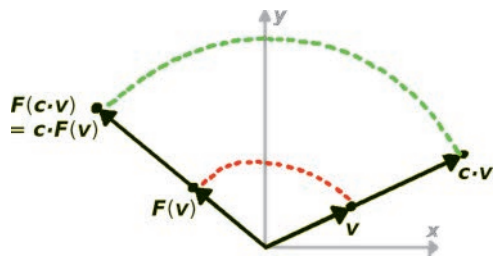

FIGURE 3.10 Illustrating that rotation transformations are linear (scalar multiplication).

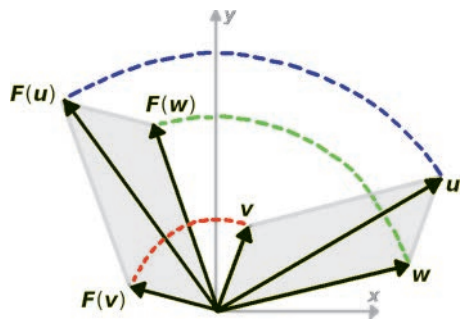

FIGURE 3.11 Illustrating that rotation transformations are linear (vector addition).

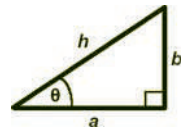

FIGURE 3.12 Right triangle with angle θ, indicating adjacent (a), opposite (b), and hypotenuse (h) side lengths.

Given that rotation is a linear transformation, the previous theoretical discussion of linear functions provides a practical method for calculating a formula for the associated matrix. The values of the function at the standard basis vectors—in two dimensions, $F(\boldsymbol{i})$ and $F(\boldsymbol{j})$—are the columns of the associated matrix. Thus, the next step is to calculate the result of rotating the vectors $\boldsymbol{i}=\langle 1, 0\rangle$ and $\boldsymbol{j}=\langle 0, 1\rangle$ around the origin (counterclockwise) by an angle θ.

This calculation requires basic knowledge of trigonometric functions. Given a right triangle with angle θ, adjacent side length a, opposite side length b, and hypotenuse length h, as illustrated in Figure 3.12, then the trigonometric functions are defined as ratios of these lengths: the sine function is defined by $\sin(\theta)=b/h$, the cosine function is defined by $\cos(\theta)=a/h$, and the tangent function is defined by $\tan(\theta)=b/a$.

As illustrated in Figure 3.13, rotating the vector $\boldsymbol{i}$ by an angle θ yields a new vector $F(\boldsymbol{i})$, which can be viewed as the hypotenuse of a right triangle. Since rotation does not change lengths of vectors, the hypotenuse has length $h=1$, which implies $\sin(\theta)=b$ and $\cos(\theta)=a$, from which it follows that $F(\boldsymbol{i})=\cos(\theta),\sin(\theta)$. This vector represents the first column of the rotation matrix.

As illustrated in Figure 3.14, rotating the vector $\boldsymbol{j}$ by an angle θ yields a new vector $F(\boldsymbol{j})$, which can once again be viewed as the hypotenuse of a right triangle with $h=1$, and as before, $\sin(\theta)=b$ and $\cos(\theta)=a$. Note that since the horizontal displacement of the vector $F(\boldsymbol{j})$ is towards the

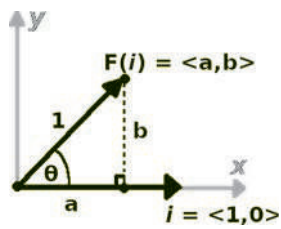

FIGURE 3.13 Rotating the basis vector $\boldsymbol{i}$ by an angle θ.

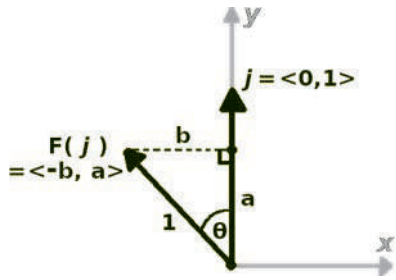

FIGURE 3.14 Rotating the basis vector $\boldsymbol{j}$ by an angle θ.

negative x direction, the value of the first vector component is $-b$, the negative of the length of the side opposite angle θ. From this, it follows that $F(\boldsymbol{j}) = \langle -\sin(\theta), \cos(\theta) \rangle$, yielding the second column of the rotation matrix.

Based on these calculations, the matrix corresponding to rotation around the origin by an angle θ in two-dimensional space is given by the matrix:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

To conclude the discussion of two-dimensional rotations, consider the following computational example. Assume that one wants to rotate the point (7, 5) around the origin by $\theta = 30°$ (or equivalently, $\theta = \pi/6$ radians). The new location of the point can be calculated as follows:

$$\begin{bmatrix} \cos(30°) & -\sin(30°) \\ \sin(30°) & \cos(30°) \end{bmatrix} \begin{bmatrix} 7 \\ 5 \end{bmatrix} = \begin{bmatrix} \sqrt{3}/2 & -1/2 \\ 1/2 & \sqrt{3}/2 \end{bmatrix} \begin{bmatrix} 7 \\ 5 \end{bmatrix}$$

$$= \begin{bmatrix} (7\sqrt{3} - 5)/2 \\ (7 + 5\sqrt{3})/2 \end{bmatrix} \approx \begin{bmatrix} 3.56 \\ 7.83 \end{bmatrix}$$

In three dimensions, rotations are performed around a line, rather than a point. In theory, it is possible to rotate around any line in three-dimensional space. For simplicity, only the formulas for rotation around each of the three axes, as illustrated in Figure 3.15, will be derived in this section.

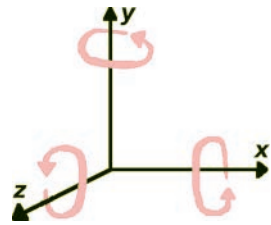

FIGURE 3.15 Rotations around the axes in three-dimensional space.

Note that the rotations appear counterclockwise when looking along each axis from positive values to the origin.

If the *xy*-plane of two-dimensional space is thought of as the set of points in three-dimensional space where $z = 0$, then the two-dimensional rotation previously discussed corresponds to rotation around the *z*-axis. Analogous to the observation that in two dimensions, rotating around a point does not move the point, in three dimensions, rotating around an axis does not move that axis. By the same reasoning as before, rotation (around an axis) is a linear transformation. Therefore, calculating the matrix corresponding to a rotation transformation *F* can be accomplished by finding the values of *F* at $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$ (the standard basis vectors in three dimensions); the vectors $F(\boldsymbol{i})$, $F(\boldsymbol{j})$, and $F(\boldsymbol{k})$ the results will be the columns of the matrix.

To begin, let *F* denote rotation around the *z*-axis, a transformation which extends the previously discussed two-dimensional rotation around a point in the *xy*-plane to three-dimensional space. Since this transformation fixes the *z*-axis and therefore all *z* coordinates, based on previous work calculating $F(\boldsymbol{i})$ and $F(\boldsymbol{j})$, it follows that $F(\boldsymbol{i}) = \langle \cos(\theta), \sin(\theta), 0 \rangle$, $F(\boldsymbol{j}) = \langle -\sin(\theta), \cos(\theta), 0 \rangle$, and $F(\boldsymbol{k}) = \langle 0, 0, 1 \rangle$, and therefore, the matrix for this transformation is

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Next, let F denote rotation around the *x*-axis. Evaluating the values of this function requires similar reasoning. This transformation fixes the *x*-axis, so $F(\boldsymbol{i}) = \langle 1, 0, 0 \rangle$ is the first column of the matrix. Since the transformation fixes all *x* coordinates, two-dimensional diagrams such as those

in Figures 3.13 and 3.14, featuring the *yz*-axes, can be used to analyze $F(\boldsymbol{j})$ and $F(\boldsymbol{k})$, this is illustrated in Figure 3.16, where the *x*-coordinate is excluded for simplicity. One finds that $F(\boldsymbol{j}) = \langle 0, \cos(\theta), \sin(\theta) \rangle$ and $F(\boldsymbol{k}) = \langle 0, -\sin(\theta), \cos(\theta) \rangle$, and thus, the corresponding matrix is

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Finally, let F denote the rotation around the *y*-axis. As before, the calculations use the same logic. However, the orientation of the axes illustrated in Figure 3.15 must be kept in mind. Drawing a diagram analogous to Figure 3.16, aligning the *x*-axis horizontally and the *z*-axis vertically (and excluding the *y*-coordinate), a counterclockwise rotation around the *y*-axis in three-dimensional space will appear as a clockwise rotation in the diagram; this is illustrated in Figure 3.17. One may then

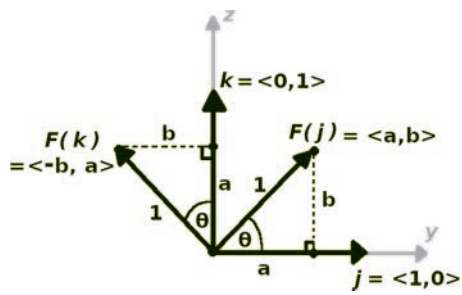

FIGURE 3.16 Calculating rotation around the *x*-axis.

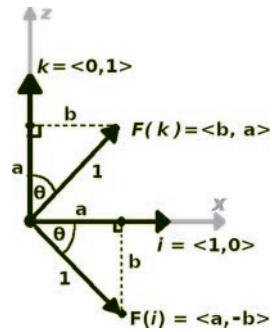

FIGURE 3.17 Calculating rotation around the *y*-axis.

calculate that $F(\boldsymbol{i}) = \langle \cos(\theta), 0, -\sin(\theta) \rangle$ and $F(\boldsymbol{k}) = \langle \sin(\theta), 0, \cos(\theta) \rangle$, and $F(\boldsymbol{j}) = \langle 0, 1, 0 \rangle$ since the *y*-axis is fixed. Therefore, the matrix for rotation around the *y*-axis is shown in Figure 3.17.

This completes the analysis of rotations in three-dimensional space; you now have formulas for generating a matrix corresponding to counterclockwise rotation by an angle θ around each of the axes. As a final note, observe that if the angle of rotation is $\theta = 0$, then since $\cos(0) = 1$ and $\sin(0) = 1$, each of the rotation matrix formulas yields an identity matrix.

3.2.3 Translation

A translation transformation adds constant values to each component of a vector. For two-dimensional vectors, this has the following form (where *m* and *n* are constants):

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} x+m \\ y+n \end{bmatrix}$$

It can quickly be established that this transformation **cannot** be represented with a 2-by-2 matrix. For example, consider translation by ⟨2, 0⟩. If this could be represented as a matrix transformation, then it would be possible to solve the following equation for the constants *a*, *b*, *c*, and *d*:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a \cdot x + b \cdot y \\ c \cdot x + d \cdot y \end{bmatrix} = \begin{bmatrix} x+2 \\ y \end{bmatrix}$$

Matching coefficients of *x* and *y* leads to $a = 1$, $c = 0$, $d = 1$, and the unavoidable expression $b = 2/y$, which is not a constant value for *b*. If the value $b = 2$ were chosen, the resulting matrix would not produce a translation—it would correspond to a *shear transformation*:

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + 2 \cdot y \\ y \end{bmatrix}$$

This particular shear transformation is illustrated in Figure 3.18, where the dot-shaded square on the left side is transformed into the dot-shaded parallelogram on the right side.

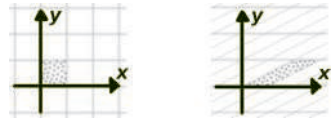

FIGURE 3.18 A shear transformation along the x direction.

Observe that, in Figure 3.18, the points along each horizontal line are being translated by a constant amount that depends on the y-coordinate of the points on the line; this is the defining characteristic of any shear transformation. In this particular example, the points along the line $y = 1$ are translated 2 units to the right, the points along $y = 2$ are translated 4 units to the right, and so forth; the points along $y = p$ are translated $2p$ units to the right.

The goal is to find a matrix that performs a constant translation on the *complete* set of points in a space; a shear transformation performs a constant translation on a *subset* of the points in a space. This observation is the key to finding the desired matrix and requires a new way of thinking about points. Consider a one-dimensional space, which would consist of only an x-axis. A translation on this space would consist of adding a constant number m to each x value. To realize this transformation as a matrix, consider a copy of the one-dimensional space embedded in two-dimensional space along the line $y = 1$; symbolically, identifying the one-dimensional point x with the two-dimensional point $(x, 1)$. Then, one-dimensional translation by m corresponds to the matrix calculation:

$$\begin{bmatrix} 1 & m \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix} = \begin{bmatrix} x+m \\ 1 \end{bmatrix}$$

Analogously, to perform a two-dimensional translation by $\langle m, n \rangle$, identify each point (x, y) with the point $(x, y, 1)$ and perform the following matrix calculation:

$$\begin{bmatrix} 1 & 0 & m \\ 0 & 1 & n \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+m \\ y+n \\ 1 \end{bmatrix}$$

Finally, to perform a three-dimensional translation by $\langle m, n, p \rangle$, identify each point (x, y, z) with the point $(x, y, z, 1)$ and perform the following matrix calculation:

$$\begin{bmatrix} 1 & 0 & 0 & m \\ 0 & 1 & 0 & n \\ 0 & 0 & 1 & p \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+m \\ y+n \\ z+p \\ 1 \end{bmatrix}$$

In other words, to represent translation as a matrix transformation, the space being translated is identified with a subset of a higher dimensional space with the additional coordinate set equal to 1. This is the reason that four-dimensional vectors and matrices are used in three-dimensional computer graphics. Even though there is no intuitive way to visualize four spatial dimensions, performing algebraic calculations on four-dimensional vectors is a straightforward process. This system of representing three-dimensional points with four-dimensional points (or representing n-dimensional points with $(n+1)$-dimensional points in general) is called *homogeneous coordinates*. As previously mentioned, each point (x, y, z) is identified with $(x, y, z, 1)$; conversely, each four-dimensional point (x, y, z, w) is associated with the three-dimensional point $(x/w, y/w, z/w)$. This operation is called *perspective division* and aligns with the previous correspondence when $w=1$. There are additional uses for perspective division, which will be discussed further in Section 3.2.4.

It is also important to verify that the transformations previously discussed are compatible with the homogeneous coordinate system. In two dimensions, the transformation $F(\langle x, y \rangle) = \langle a \cdot x + b \cdot y, c \cdot x + d \cdot y \rangle$ becomes $F(\langle x, y, 1 \rangle) = \langle a \cdot x + b \cdot y, c \cdot x + d \cdot y, 1 \rangle$, which corresponds to the matrix multiplication:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a \cdot x + b \cdot y \\ c \cdot x + d \cdot y \\ 1 \end{bmatrix}$$

Therefore (when using homogeneous coordinates), all the geometric transformations of interest—translation, rotation, and scaling, collectively referred to as *affine transformations*—can be represented by multiplying by a matrix of the following form:

$$\begin{bmatrix} a_{11} & a_{12} & m_1 \\ a_{21} & a_{22} & m_2 \\ 0 & 0 & 1 \end{bmatrix}$$

where the 2-by-2 submatrix in the upper left represents the rotation and/or scaling part of the transformation (or is the identity matrix when there is no rotation or scaling), and the two-component vector $\langle m_1, m_2 \rangle$ within the rightmost column represents the translation part of the transformation (or is the zero vector if there is no translation).

Similarly, in three dimensions (using homogeneous coordinates), affine transformations can be represented by multiplying by a matrix of the following form:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & m_1 \\ a_{21} & a_{22} & a_{23} & m_2 \\ a_{31} & a_{32} & a_{33} & m_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the 3-by-3 submatrix in the upper left represents the rotation and/or scaling part of the transformation (or is the identity matrix if there is no rotation or scaling), and the three-component vector $\langle m_1, m_2, m_3 \rangle$ within the rightmost column represents the translation part of the transformation (or is the zero vector if there is no translation).

3.2.4 Projections

In this section, the goal is to derive a formula for a perspective projection transformation. At the beginning of Chapter 1, and also in Section 1.2.2 on geometry processing, some of the core ideas were illustrated and informally introduced. To review, the viewable region in the scene needs to be mapped to the region rendered by OpenGL, a cube where the x, y, and z coordinates are all between –1 and +1, also referred to as *clip space*. In a perspective projection, the shape of the viewable region is a *frustum* or truncated pyramid. The pyramid is oriented so that it is "lying on its side": its central axis is aligned with the negative z-axis, as illustrated in Figure 3.19, and the viewer or virtual camera is positioned at the origin of the scene, which aligns with the point that was the tip of the original pyramid. The smaller rectangular end of the frustum is nearest to the origin, and the larger rectangular end is farthest from the origin. When the frustum is compressed into a cube, the larger end must be compressed more. This causes objects in the rendered image of the scene to appear smaller the farther they are from the viewer.

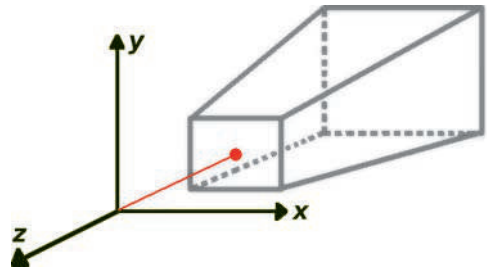

FIGURE 3.19 The frustum for a perspective transformation.

The shape of a frustum is defined by four parameters: the *near distance*, the *far distance*, the (vertical) *angle of view*, and the *aspect ratio*. The near and far distances are the most straightforward to explain: they refer to distances from the viewer (along the z-axis), and they set absolute bounds on what could potentially be seen by the viewer—any points outside of this range will not be rendered. However, not everything between these bounds will be visible. The *angle of view* is a measure of how much of the scene is visible to the viewer, and is defined as the angle between the top and bottom planes of the frustum (as oriented in Figure 3.19) if those planes were extended to the origin. Figure 3.20 shows two different frustums (shaded regions) as viewed from the side (along the x-axis). The figure also illustrates the fact that for fixed near and far distances, larger angles of view correspond to larger frustums.

In order for the dimensions of the visible part of the near plane to be proportional to the dimensions of the rendered image, the *aspect ratio* (defined as width divided by height) of the rendered image is the final

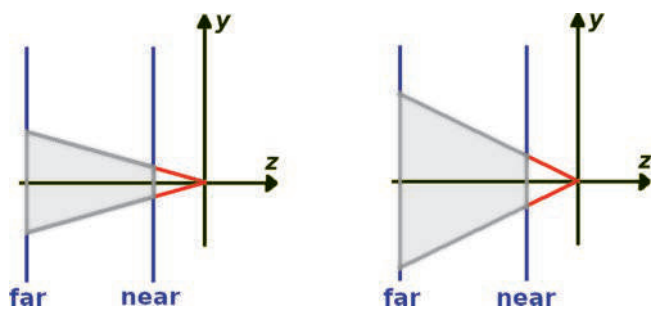

FIGURE 3.20 The effect of the angle of view on the size of a frustum.

value used to specify the shape of the frustum, illustrated in Figure 3.21, which depicts the frustum as viewed from the front (along the negative z-axis). In theory, a horizontal angle of view could be used to specify the size of the frustum instead of the aspect ratio, but in practice, determining the aspect ratio is simpler.

In a perspective projection, points in space (within the frustum) are mapped to points in the *projection window*: a flat rectangular region in space corresponding to the rendered image that will be displayed on the computer screen. The projection window corresponds to the smaller rectangular side of the frustum, the side nearest to the origin. To visualize how a point P is transformed in a perspective projection, draw a line from P to the origin, and the intersection of the line with the projection window is the result. Figure 3.22 illustrates the results of projecting three different points within the frustum onto the projection window.

To derive the formula for a perspective projection, the first step is to adjust the position of the projection window so that points in the frustum are projected to y-coordinates in the range from –1 to 1. Then, the formula for projecting y-coordinates will be derived, incorporating both matrix multiplication and the perspective division that occurs with homogeneous coordinates:

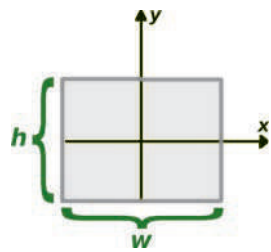

FIGURE 3.21 The aspect ratio $r = w/h$ of the frustum.

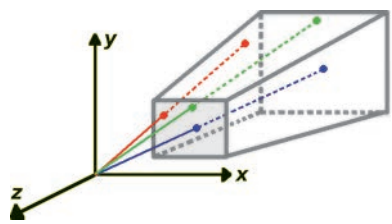

FIGURE 3.22 Projecting points from the frustum to the projection window.

converting (x, y, z, w) to $(x/w, y/w, z/w)$. Next, the formula for projecting x-coordinates will be derived, which is completely analogous to the formula for y-coordinates except that the aspect ratio needs to be taken into consideration. Finally, the z-coordinates of points in the frustum, which are bounded by the near distance and far distance, will be converted into the range from -1 to 1 and once again will require taking perspective division into account.

To begin, it will help to represent the parameters that define the shape of the frustum—the near distance, far distance, (vertical) angle of view, and aspect ratio—by n, f, a, and r, respectively. Consider adjusting the projection window so that the y-coordinates range from -1 to 1, while preserving the angle of view a, as illustrated on the left side of Figure 3.23 (viewed from the side, along the x-axis). This will change the distance d from the origin to the projection window. The value of d can be calculated using trigonometry on the corresponding right triangle, illustrated on the right side of Figure 3.23. By the definition of the tangent function, $\tan(a/2) = 1/d$, from which it follows that $d = 1/\tan(a/2)$. Therefore, all points on the projection window have their z coordinate equal to $-d$.

Next, ignoring x-coordinates for a moment, consider a point $P = (P_y, P_z)$ in the frustum that will be projected onto this new projection window. Drawing a line from P to the origin, let $Q = (Q_y, Q_z)$ be the intersection of the line with the projection window, as illustrated in Figure 3.24. Adding a perpendicular line from P to the z-axis, it becomes clear that we have two similar right triangles. Note that since the bases of the triangles are located on the negative z-axis, but lengths of sides are positive, the lengths of the sides are the negatives of the z-coordinates. The sides of similar triangles are proportional to each other, so we know that $P_y / (-P_z) = Q_y / (-Q_z)$.

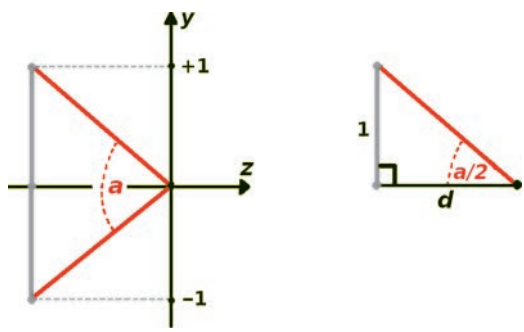

FIGURE 3.23 Adjusting the projection window.

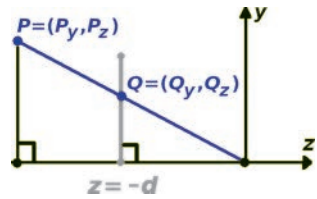

FIGURE 3.24 Calculating *y*-components of projected points.

Since Q is a point on the adjusted projection window, we also know that $Q_z = -d$. This allows us to write a formula for the y-coordinate of the projection: $Q_y = d \cdot P_y / (-P_z)$.

At first glance, this may appear to be incompatible with our matrix-based approach, as this formula is not a linear transformation, due to the division by the z coordinate. Fortunately, the fact that we are working in homogeneous coordinates (x, y, z, w) will enable us to resolve this problem. Since this point will be converted to (x/w, y/w, z/w) by perspective division (which is automatically performed by the GPU after the vertex shader is complete), the "trick" is to use a matrix to change the value of the w-component (which is typically equal to 1) to the value of the z-component (or more precisely, the negative of the z-component). This can be accomplished with the following matrix transformation (where $*$ indicates an as-yet unknown value):

$$\begin{bmatrix} * & * & * & * \\ 0 & d & 0 & 0 \\ * & * & * & * \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} * \\ d \cdot P_y \\ * \\ -P_z \end{bmatrix}$$

Then, after performing the division transformation, the homogeneous point $(*, d \cdot P_y, *, -P_z)$ is transformed into $(*, d \cdot P_y / (-P_z), *)$, as desired.

The next step is to derive a formula for the x-component of projected points; the calculations are similar to those for the y-component. The corresponding diagram is illustrated in Figure 3.25, viewed from along the y-axis. Again there is a pair of similar triangles, and therefore, the ratios of corresponding sides are equal, from which we obtain the equation $P_x / (-P_z) = Q_x / (-Q_z)$, and therefore, $Q_x = d \cdot P_x / (-Q_z)$.

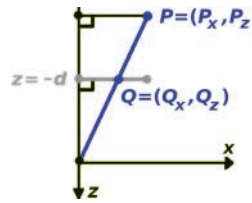

FIGURE 3.25 Calculating *x*-components of projected points.

However, one additional factor must be taken into account: the aspect ratio *r*. We have previously considered the *y* values to be in the range from −1 to 1; in accordance with the aspect ratio, the set of points that should be included in the rendered image have *x* values in the range from −*r* to *r*. This range needs to be scaled into clip space, from −1 to 1, and therefore, the formula for the x coordinate must also be divided by *r*. This leads us to the formula $Q_x = (d/r) \cdot P_x / (-P_z)$, which can be accomplished with the following matrix transformation (again, ∗ indicating undetermined values):

$$\begin{bmatrix} d/r & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ * & * & * & * \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} (d/r) \cdot P_x \\ d \cdot P_y \\ * \\ -P_z \end{bmatrix}$$

The values in the third row of the matrix remain to be determined and will affect the *z* component of the point. The *z* value is used in depth calculations to determine which points will be visible, as specified by the near distance and far distance values. The values of the *x* and *y* components of the point are not needed for this calculation, and so the first two values in the row should be 0; refer to the remaining unknown values as *b* and *c*. Then, we have the following matrix transformation:

$$\begin{bmatrix} d/r & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & b & c \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} (d/r) \cdot P_x \\ d \cdot P_y \\ b \cdot P_z + c \\ -P_z \end{bmatrix}$$

After perspective division, the third coordinate $b \cdot P_z + c$ becomes $(b \cdot P_z + c)/(-P_z) = -b - c/P_z$. The values of the constants b and c will be determined soon, after two important points are clarified. First, the near distance n and the far distance f are typically given as positive values, even though the visible region frustum lies along the negative z-axis in world space, and thus, the nearest visible point P satisfies $P_z = -n$, while the farthest visible point P satisfies $P_z = -f$. Second, we know that we must convert the z coordinates of visible points into clip space coordinates (the range from –1 to 1), and it might seem as though the z coordinates of the nearest points to the viewer should be mapped to the value 1, as the positive z axis points directly at the viewer in our coordinate system. However, this is not the case! When OpenGL performs depth testing, it determines if one point is closer (to the viewer) than another by comparing their z coordinates. The type of comparison can be specified by the OpenGL function **`glDepthFunc`**, which has the default value GL_LESS, an OpenGL constant which indicates that a point should be considered closer to the viewer if its z coordinate is *less*. This means that in clip space, the *negative* z axis points directly at the viewer; this space uses a *left-handed* coordinate system. Combining these two points, we now know that if $P_z = -n$, then $-b - c/P_z$ should equal –1, and if $P_z = -f$, then $-b - c/P_z$ should equal 1. This corresponds to the following system of equations:

$$-b + \frac{c}{n} = -1$$

$$-b + \frac{c}{f} = 1$$

There are a variety of approaches to solve this system; one of the simplest is to multiply the first equation by –1 and then add it to the second equation. This eliminates b; solving for c yields $c = 2 \cdot n \cdot f / (n - f)$. Substituting this value for c into the first equation and solving for b yield $b = (n + f)/(n - f)$.

With this calculation finished, the matrix is completely determined. To summarize, the perspective projection transformation for a frustum-shaped region defined by near distance n, far distance f, (vertical) angle of view a, and aspect ratio r, when working with homogeneous coordinates, can be achieved with the following matrix:

$$\begin{bmatrix} \frac{1}{r \cdot \tan(a/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(a/2)} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2 \cdot n \cdot f}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

3.2.5 Local Transformations

At this point, you are able to produce the matrices corresponding to translation, rotation, and scaling transformations. For example, consider an object in two-dimensional space, such as the turtle on the left side of Figure 3.26, whose shape is defined by some set of points *S*. Let ***T*** be the matrix corresponding to translation by ⟨1, 0⟩, and let ***R*** be the matrix corresponding to rotation around the origin by 45°. To move the turtle around you could, for example, multiply all the points in the set *S* by ***T***, and then by ***R***, and then by ***T*** again, which would result in the sequence of images illustrated in the remaining parts of Figure 3.26. All these transformations are relative to an external, fixed coordinate system called *world coordinates* or *global coordinates*, and this aspect is emphasized by the more specific term *global transformations*.

The internal or local coordinate system used to define the vertices of a geometric object is somewhat arbitrary—the origin point and the orientation and scale of the local coordinate axes are typically chosen for convenience, without reference to the global coordinate system. For example, the local origin is frequently chosen to correspond to a location on the object that is in some sense the "center" of the object. Locations specified

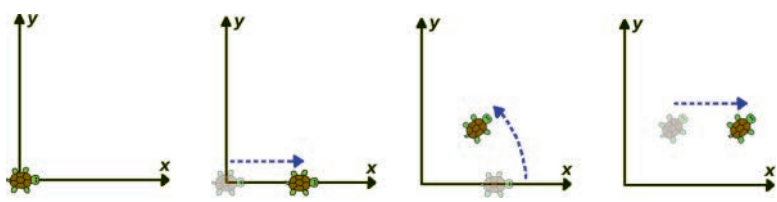

FIGURE 3.26 A sequence of global transformations.

relative to this coordinate system are called *object coordinates* or *local coordinates*. After an object is added to a three-dimensional scene, the object can then be repositioned, oriented, and resized as needed using geometric transformations.

Of particular interest in this section are *local transformations*: transformations relative to the local coordinate system of an object, and how they may be implemented with matrix multiplication. Initially, the local coordinate axes of an object are aligned with the global coordinate axes. As an object is transformed, its local coordinate axes undergo the same transformations. Figure 3.27 illustrates multiple copies of the turtle object together with their local coordinate axes after various transformations have been applied.

Figure 3.28 illustrates several examples of local transformations. Assuming the turtle starts at the state with position and orientation shown

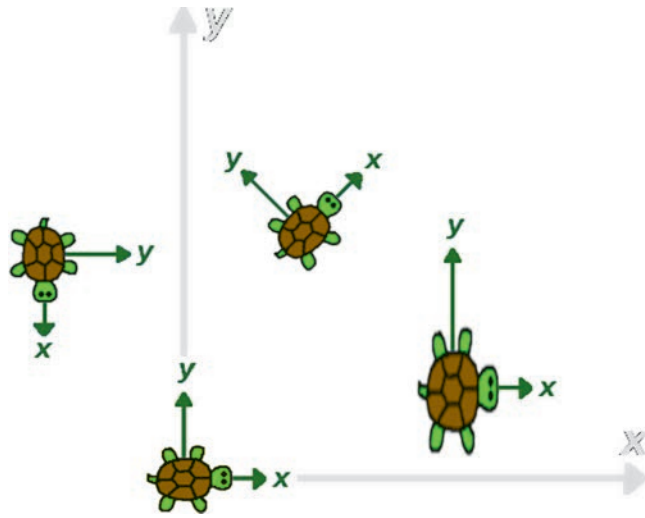

FIGURE 3.27 Transforming local coordinate axes.

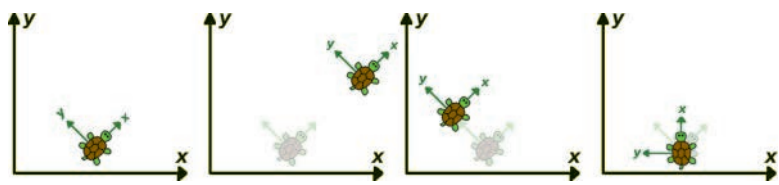

FIGURE 3.28 Various local transformations.

in the leftmost image, the remaining images show local translation by $\langle 2, 0\rangle$, local translation by $\langle 0, 1\rangle$, and local rotation by 45°, each applied to the starting state. Observe in particular that a local rotation takes place around the center of an object (the origin of its local coordinate system), rather than around the world or global origin.

The concepts of local and global transformation are reflected in everyday language. For example, walking forwards, backwards, left, or right, are examples of local translations that depend on the current orientation of a person's coordinate system: in other words, it matters which way the person is currently facing. If two people are facing in different directions, and they are both asked to step forward, they will move in different directions. If a global translation is specified, which is typically done by referencing the compass directions (north, south, east, and west), then you can be assured that people will walk in the same direction, regardless of which way they may have been facing at first.

The question remains: how can matrix multiplication be used to perform local transformations (assuming that it is possible)? Before addressing this question, it will help to introduce a new concept. When transforming a set of points with a matrix, the points are multiplied by the matrix in the vertex shader and the new coordinates are passed along to the fragment shader, but the new coordinates of the points are not permanently stored. Instead, the accumulated transformations that have been applied to an object are stored as the product of the corresponding matrices, which is a single matrix called the *model matrix* of the object. The model matrix effectively stores the current location, orientation, and scale of an object (although it is slightly complicated to extract some of the information from the entries of the matrix).

Given an object whose shape is defined by a set of points S, assume that a sequence of transformations have been applied and let $\boldsymbol{M}$ denote the current model matrix of the object. Thus, the current location of the points of the object can be calculated by $\boldsymbol{M}\cdot P$, where P ranges over the points in the set S. Let $\boldsymbol{T}$ be the matrix corresponding to a transformation. If you were to apply this matrix as a global transformation, as described in previous sections, the new model matrix would be $\boldsymbol{T}\cdot\boldsymbol{M}$, since each new matrix that is applied becomes the leftmost element in the product (just as functions are ordered in function composition). In order for the matrix $\boldsymbol{T}$ to have the effect of a local transformation on an object, the local coordinate axes of the object would have to be aligned with the global coordinate axes, which suggests the following three-step strategy:

1. Align the two sets of axes by applying $\boldsymbol{M}^{-1}$, the inverse of the model matrix.
2. Apply $\boldsymbol{T}$, since local and global transformations are the same when the axes are aligned.
3. Apply $\boldsymbol{M}$, the original model matrix, which returns the object to its previous state while taking the transformation $\boldsymbol{T}$ into account. (In a sense, it is as if the matrix $\boldsymbol{M}$ has been applied to transformation $\boldsymbol{T}$, converting it from a global transformation to a local transformation.)

This sequence of transformations is illustrated in Figure 3.29, where the images show an object with model matrix $\boldsymbol{M}$ (in this example, translation and then rotation by 45° was applied), the result of applying $\boldsymbol{M}^{-1}$ (reversing the rotation and then reversing the translation), the result of applying $\boldsymbol{T}$ (in this example, translation), and the result of applying $\boldsymbol{M}$ again (translating and rotating again). The last image also shows the outline of the object in its original placement for comparison.

At the end of this process, combining all these transformations, the model matrix has become $\boldsymbol{M} \cdot \boldsymbol{T} \cdot \boldsymbol{M}^{-1} \cdot \boldsymbol{M}$ (recall that matrices are applied to a point from the right to the left). Since $\boldsymbol{M}^{-1} \cdot \boldsymbol{M} = \boldsymbol{I}$ (the identity matrix), this expression simplifies to $\boldsymbol{M} \cdot \boldsymbol{T}$: the original model matrix $\boldsymbol{M}$ multiplied on the *right* by $\boldsymbol{T}$. This is the answer to the question of interest. To summarize, given an object with model matrix $\boldsymbol{M}$ and a transformation with matrix $\boldsymbol{T}$:

- To apply $\boldsymbol{T}$ as a global transformation, let the new model matrix equal $\boldsymbol{T} \cdot \boldsymbol{M}$.
- To apply $\boldsymbol{T}$ as a local transformation, let the new model matrix equal $\boldsymbol{M} \cdot \boldsymbol{T}$.

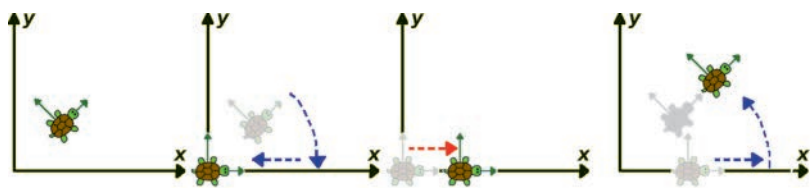

FIGURE 3.29 A sequence of global transformations equivalent to a local translation.

As you will see in later sections, being able to use local transformations will be useful when creating interactive three-dimensional scenes. For example, navigating within a scene feels more intuitive and immersive if one is able to move the virtual camera with local transformations. The viewer might not always be aware of the direction of the *z*-axis, which could lead to unexpected motions when moving in global directions, but the viewer is always able to see what is in front of them, which makes moving forward more natural.

3.3 A MATRIX CLASS

Now that you have learned how to derive the various types of matrices that will be needed, the next step will be to create a **`Matrix`** class containing static methods to generate matrices (with the numpy library) corresponding to each of the previously discussed geometric transformations: identity, translation, rotation (around each axis), scaling, and projection. To proceed, in the **`core`** folder, create a new file named **`matrix.py`** containing the following code:

```
import numpy
from math import sin, cos, tan, pi

class Matrix(object):

    @staticmethod
    def makeIdentity():
        return numpy.array( [[1, 0, 0, 0],
                             [0, 1, 0, 0],
                             [0, 0, 1, 0],
                             [0, 0, 0, 1]] ).
                             astype(float)

    @staticmethod
    def makeTranslation(x, y, z):
        return numpy.array([[1, 0, 0, x],
                            [0, 1, 0, y],
                            [0, 0, 1, z],
                            [0, 0, 0, 1]]).
                            astype(float)

    @staticmethod
    def makeRotationX(angle):
```

```
        c = cos(angle)
        s = sin(angle)
        return numpy.array([[1, 0,  0, 0],
                            [0, c, -s, 0],
                            [0, s,  c, 0],
                            [0, 0,  0, 1]]).
                            astype(float)

    @staticmethod
    def makeRotationY(angle):
        c = cos(angle)
        s = sin(angle)
        return numpy.array([[ c, 0, s, 0],
                            [ 0, 1, 0, 0],
                            [-s, 0, c, 0],
                            [ 0, 0, 0, 1]]).
                            astype(float)

    @staticmethod
    def makeRotationZ(angle):
        c = cos(angle)
        s = sin(angle)
        return numpy.array([[c, -s, 0, 0],
                            [s,  c, 0, 0],
                            [0,  0, 1, 0],
                            [0,  0, 0, 1]]).
                            astype(float)

    @staticmethod
    def makeScale(s):
        return numpy.array([[s, 0, 0, 0],
                            [0, s, 0, 0],
                            [0, 0, s, 0],
                            [0, 0, 0, 1]]).
                            astype(float)

    @staticmethod
    def makePerspective(angleOfView=60,
aspectRatio=1, near=0.1, far=1000):
        a = angleOfView * pi/180.0
        d = 1.0 / tan(a/2)
        r = aspectRatio
```

```
        b = (far + near) / (near - far)
        c = 2*far*near / (near - far)
        return numpy.array([[d/r, 0,  0, 0],
                            [0,   d,  0, 0],
                            [0,   0,  b, c],
                            [0,   0, -1, 0]]).
                            astype(float)
```

With this class completed, you are nearly ready to incorporate matrix-based transformations into your scenes.

3.4 INCORPORATING WITH GRAPHICS PROGRAMS

Before creating the main application, the **`Uniform`** class needs to be updated to be able to store the matrices generated by the **`Matrix`** class. In GLSL, 4-by-4 matrices correspond to the data type **`mat4`**, and the corresponding uniform data can be sent to the GPU with the following OpenGL command:

glUniformMatrix4fv(*variableRef*, *matrixCount*, *transpose*, *value*)

Specify the value of the uniform variable referenced by the parameter *variableRef* in the currently bound program. The number of matrices is specified by the parameter *matrixCount*. The matrix data is stored as an array of vectors in the parameter *value*. OpenGL expects matrix data to be stored as an array of column vectors; if this is not the case (if data is stored as an array of row vectors), then the boolean parameter *transpose* should be set to the OpenGL constant GL_TRUE (which causes the data to be re-interpreted as rows) and GL_FALSE otherwise.

In the file **`uniform.py`** located in the **`core`** folder, add the following else-if condition at the end of the block of **`elif`** statements in the **`uploadData`** function:

```
elif self.dataType == "mat4":
    glUniformMatrix4fv(self.variableRef, 1, GL_TRUE,
      self.data)
```

Since you will now be creating three-dimensional scenes, you will activate a render setting that performs depth testing (in case you later

choose to add objects which may obscure other objects). This (and other render settings) can be configured by using the following two OpenGL functions:

glEnable(*setting*)

Enables an OpenGL capability specified by an OpenGL constant specified by the parameter *setting*. For example, possible constants include GL_DEPTH_TEST to activate depth testing, GL_POINT_SMOOTH to draw rounded points instead of square points, or GL_BLEND to enable blending colors in the color buffer based on alpha values.

glDisable(*setting*)

Disables an OpenGL capability specified by an OpenGL constant specified by the parameter *setting*.

Finally, for completeness, we include the OpenGL function that allows you to configure depth testing, previously mentioned in Section 3.2.4 on perspective projection. However, you will not change the function from its default setting, and this function will not be used in what follows.

glDepthFunc(compareFunction)

Specify the function used to compare each pixel depth with the depth value present in the depth buffer. If a pixel passes the comparison test, it is considered to be currently the closest to the viewer, and its values overwrite the current values in the color and depth buffers. The function used is specified by the OpenGL constant *compareFunction*, some of whose possible values include the default setting GL_LESS (indicating that a pixel is closer to the viewer if the depth value is less) and GL_GREATER (indicating that a pixel is closer if the depth value is greater). If depth testing has not been enabled, the depth test always passes, and pixels are rendered on top of each other according to the order in which they are processed.

Now you are ready to create an interactive scene. The first new component will be the vertex shader code: there will be two uniform **`mat4`** variables. One will store the model transformation matrix, which will be used to translate and rotate a geometric object. The other will store the perspective transformation matrix, which will make objects appear smaller as

they move further away. To begin creating this scene, in your main directory, create a file named **test-3.py**, containing the following code (the **import** statements will be needed later on):

```
from core.base import Base
from core.openGLUtils import OpenGLUtils
from core.attribute import Attribute
from core.uniform import Uniform
from core.matrix import Matrix
from OpenGL.GL import *
from math import pi


# move a triangle around the screen
class Test(Base):

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


        ### initialize program ###
        vsCode = """
        in vec3 position;
        uniform mat4 projectionMatrix;
        uniform mat4 modelMatrix;
        void main()
        {
            gl_Position = projectionMatrix *
                  modelMatrix * vec4(position, 1.0);
        }
        """


        fsCode = """
        out vec4 fragColor;
        void main()
        {
            fragColor = vec4(1.0, 1.0, 0.0, 1.0);
        }
        """


        self.programRef = OpenGLUtils.initializeProgram(
            vsCode, fsCode)
```

Next, you will initialize attribute data for the vertices of an isosceles triangle, uniforms for the model and projection matrices, and variables to store the movement and rotation speed that will be applied to the triangle in the **update** function. To continue, return to the file named **test-3.py**, and add the following code to the **initialize** function:

```
### render settings ###
glClearColor(0.0, 0.0, 0.0, 1.0)
glEnable(GL_DEPTH_TEST)

### set up vertex array object ###
vaoRef = glGenVertexArrays(1)
glBindVertexArray(vaoRef)

### set up vertex attribute ###
positionData = [ [0.0, 0.2, 0.0], [0.1, -0.2, 0.0],
   [-0.1, -0.2, 0.0] ]
self.vertexCount = len(positionData)
positionAttribute = Attribute("vec3", positionData)
positionAttribute.associateVariable( self.programRef,
   "position" )

### set up uniforms ###
mMatrix = Matrix.makeTranslation(0, 0, -1)
self.modelMatrix = Uniform("mat4", mMatrix)
self.modelMatrix.locateVariable( self.programRef,
   "modelMatrix" )

pMatrix = Matrix.makePerspective()
self.projectionMatrix = Uniform("mat4", pMatrix)
self.projectionMatrix.locateVariable( self.programRef,
   "projectionMatrix" )

# movement speed, units per second
self.moveSpeed = 0.5
# rotation speed, radians per second
self.turnSpeed = 90 * (pi / 180)
```

Next, you will turn your attention to creating an **update** function. The first step will be to calculate the actual amounts of movement that may be applied, based on the previously set base speed and the time elapsed since

the last frame (which is stored in the **Base** class variable **deltaTime**). In the file **test-3.py**, add the following code:

```
def update(self):
    # update data
    moveAmount = self.moveSpeed * self.deltaTime
    turnAmount = self.turnSpeed * self.deltaTime
```

To illustrate the versatility of using matrices to transform objects, both global and local movement will be implemented. Next, there will be a large number of conditional statements, each of which follow the same pattern: check if a particular key is being pressed, and if so, create the corresponding matrix **m** and multiply the model matrix by **m** in the correct order (**m** on the left for global transformations and **m** on the right for local transformations). Note that while shaders use the operator '*' for matrix multiplication, numpy uses the operator '**@**' for matrix multiplication. For global translations, you will use the keys W/A/S/D for the up/left/down/right directions and the keys Z/X for the forward/backward directions. In the file **test-3.py**, in the **update** function, add the following code:

```
# global translation
if self.input.isKeyPressed("w"):
    m = Matrix.makeTranslation(0, moveAmount, 0)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("s"):
    m = Matrix.makeTranslation(0, -moveAmount, 0)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("a"):
    m = Matrix.makeTranslation(-moveAmount, 0, 0)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("d"):
    m = Matrix.makeTranslation(moveAmount, 0, 0)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("z"):
    m = Matrix.makeTranslation(0, 0, moveAmount)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("x"):
    m = Matrix.makeTranslation(0, 0, -moveAmount)
    self.modelMatrix.data = m @ self.modelMatrix.data
```

Rotation that makes the object appear to rotate left and right from the viewer's perspective is really a rotation around the *z*-axis in three-dimensional space. Since the keys A/D move the object left/right, you will use the keys Q/E to rotate the object left/right, as they lay in the row directly above A/D. Since these keys will be used for a global rotation, they will cause the triangle to rotate around the origin (0,0,0) of the three-dimensional world. Continuing on in the file **`test-3.py`**, in the **`update`** function, add the following code:

```
# global rotation (around the origin)
if self.input.isKeyPressed("q"):
    m = Matrix.makeRotationZ(turnAmount)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("e"):
    m = Matrix.makeRotationZ(-turnAmount)
    self.modelMatrix.data = m @ self.modelMatrix.data
```

Next, to incorporate local translation, you will use the keys I/J/K/L for the directions up/left/down/right, as they are arranged in a similar layout to the W/A/S/D keys. Continue by adding the following code to the **`update`** function:

```
# local translation
if self.input.isKeyPressed("i"):
    m = Matrix.makeTranslation(0, moveAmount, 0)
    self.modelMatrix.data = self.modelMatrix.data @ m
if self.input.isKeyPressed("k"):
    m = Matrix.makeTranslation(0, -moveAmount, 0)
    self.modelMatrix.data = self.modelMatrix.data @ m
if self.input.isKeyPressed("j"):
    m = Matrix.makeTranslation(-moveAmount, 0, 0)
    self.modelMatrix.data = self.modelMatrix.data @ m
if self.input.isKeyPressed("l"):
    m = Matrix.makeTranslation(moveAmount, 0, 0)
    self.modelMatrix.data = self.modelMatrix.data @ m
```

You will use the keys U/O for local rotation left/right, as they are in the row above the keys used local movement left/right (J/L), analogous to the key layout for global transformations. Since these keys will refer to a local rotation, they will rotate the triangle around its center (where the world origin was located when the triangle was in its original position).

```
# local rotation (around object center)
if self.input.isKeyPressed("u"):
    m = Matrix.makeRotationZ(turnAmount)
    self.modelMatrix.data = self.modelMatrix.data @ m
if self.input.isKeyPressed("o"):
    m = Matrix.makeRotationZ(-turnAmount)
    self.modelMatrix.data = self.modelMatrix.data @ m
```

After processing user input, the buffers need to be cleared before the image is rendered. In addition to clearing the color buffer, since depth testing is now being performed, the depth buffer should also be cleared. Uniform values need to be stored in their corresponding variables, and the **glDrawArrays** function needs to be called to render the triangle. To accomplish these tasks, at the end of the **update** function, add the following code:

```
### render scene ###
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
glUseProgram( self.programRef )
self.projectionMatrix.uploadData()
self.modelMatrix.uploadData()
glDrawArrays( GL_TRIANGLES , 0 , self.vertexCount )
```

Finally, to run this application, the last lines of code need to instantiate the **Test** class and call the **run** function. At the end of the **test-3.py** file, add the following code (with no indentation, as it is not part of the class or the **update** function):

```
# instantiate this class and run the program
Test().run()
```

At this point, the application class is complete! Run the file and you should see an isosceles triangle in the center of your screen. Press the keyboard keys as described previously to experience the difference between global and local transformations of an object. While the object is located at the origin, local and global rotations will appear identical, but when the object is located far away from the origin, the difference in rotation is more easily seen. Similarly, while the object is in its original orientation (its local right direction aligned with the positive x-axis), local and global translations will appear identical, but after a rotation, the difference in the translations becomes apparent.

3.5 SUMMARY AND NEXT STEPS

In this chapter, you learned about the mathematical foundations involved in geometric transformations. You learned about vectors and the operations of vector addition and scalar multiplication, and how any vector can be written as a combination of standard basis vectors using these operations. Then, you learned about a particular type of vector function called a linear transformation, which naturally led to the definition of a matrix and matrix multiplication. Then, you learned how to create matrices representing the different types of geometric transformations (scaling, rotation, translation, and perspective projection) used in creating animated, interactive three-dimensional graphics applications. You also learned how a model matrix stores the accumulated transformations that have been applied to an object, and how this structure enables you to use matrices for both global and local transformations. Finally, all of this was incorporated into the framework being developed in this book.

In the next chapter, you will turn your attention to automating many of these steps as you begin to create the graphics framework classes in earnest.

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

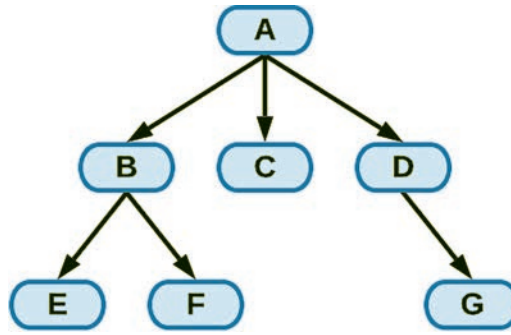

FIGURE 4.1 A tree with seven nodes.

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

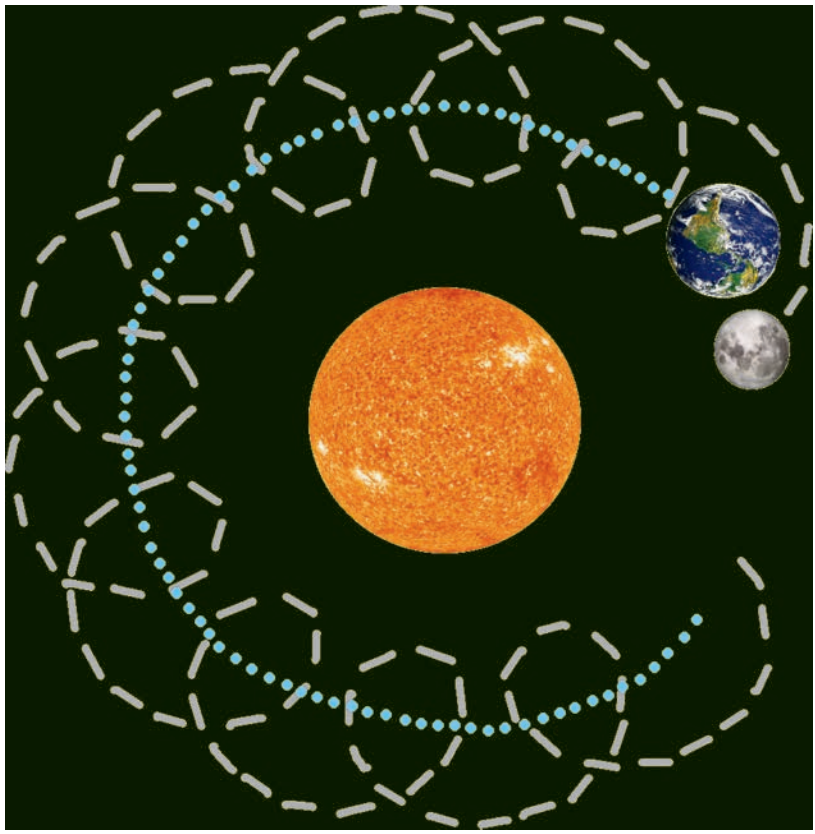

FIGURE 4.2 Motion of moon and Earth relative to sun.

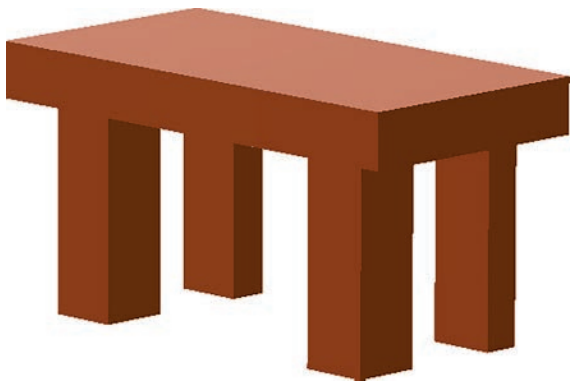

FIGURE 4.3 A table composed of five boxes.

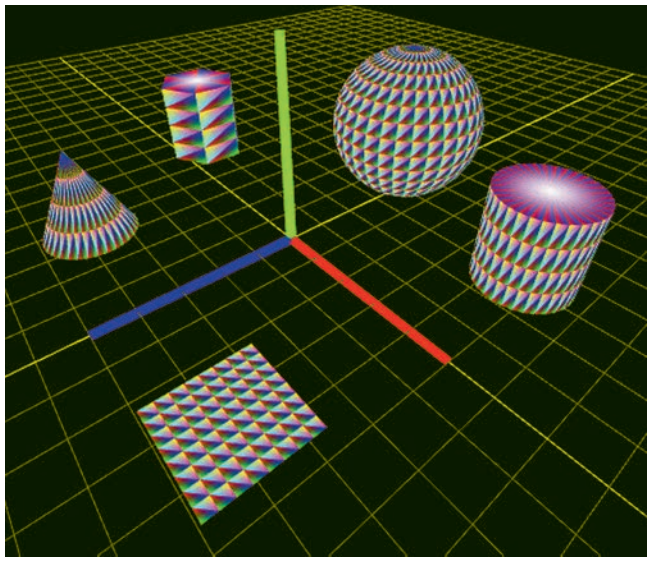

FIGURE 4.4 A scene containing multiple geometric shapes.

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

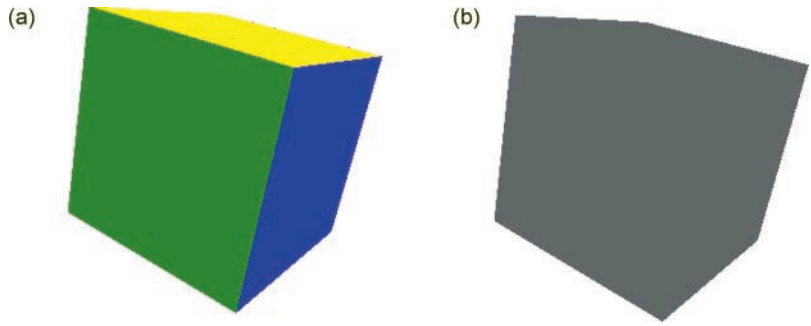

FIGURE 4.5 A cube rendered with (a) and without (b) vertex colors.

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 (*P*0, *P*1, *P*3) and (*P*0, *P*3, *P*2). 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:

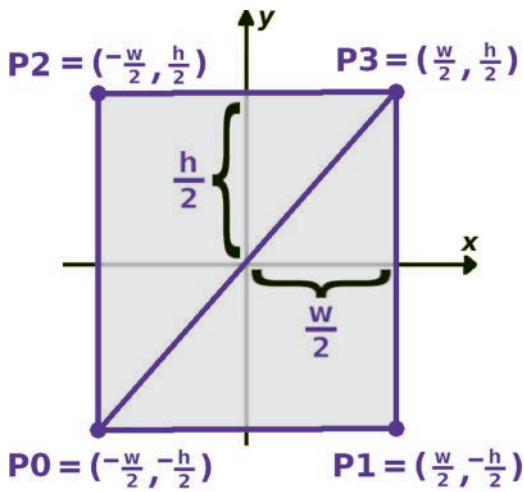

FIGURE 4.6 Vertex coordinates for a rectangle with width w and height h.

```
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

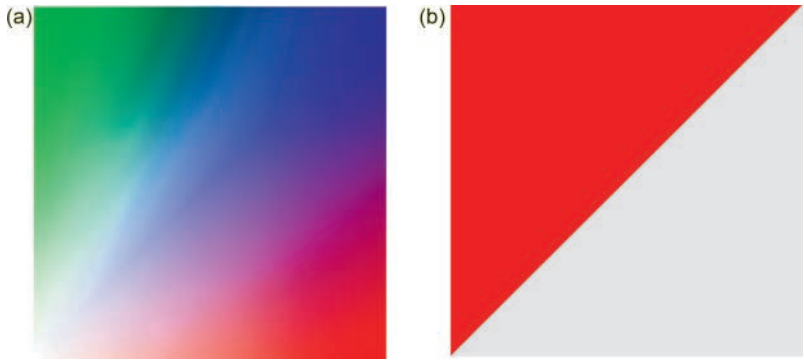

FIGURE 4.7 Rendering RectangleGeometry with gradient coloring (a) and solid coloring (b).

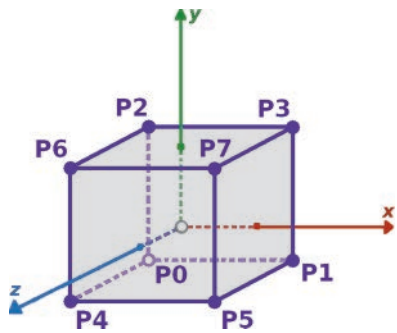

FIGURE 4.8 Vertices of a cube.

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.

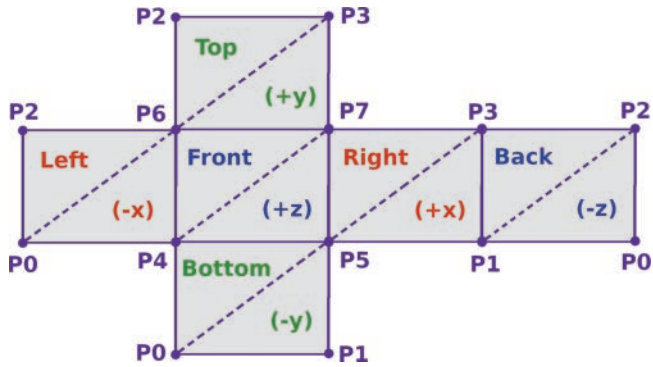

FIGURE 4.9 Vertex arrangement of an unfolded cube.

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 **`boxGeometry.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.

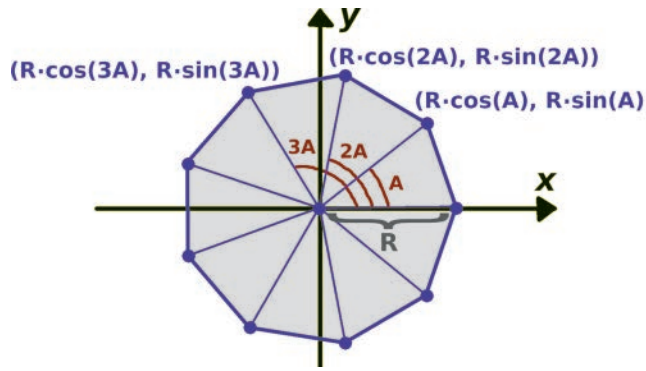

FIGURE 4.11 Calculating the vertices of a regular polygon.

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:

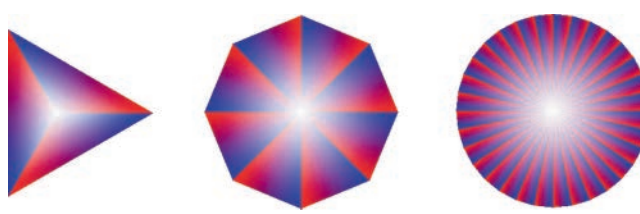

FIGURE 4.12 Polygons with 3 sides, 8 sides, and 32 sides.

```
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 $0 \leq u \leq 1$ and $0 \leq v \leq 1$, 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

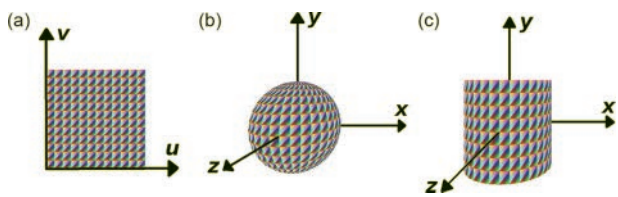

FIGURE 4.13 A rectangular region (a), transformed into a sphere (b) and a cylinder (c).

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 $0 \leq u \leq 2\pi$. 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

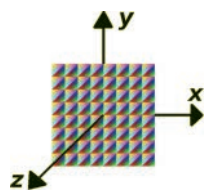

FIGURE 4.14 Plane geometry.

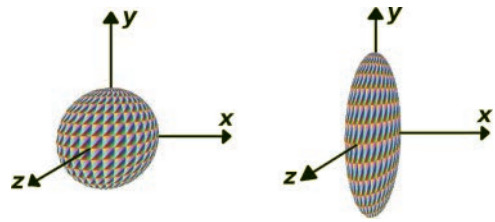

FIGURE 4.15 Sphere and ellipsoid.

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 $0 \leq u \leq 2\pi$. Furthermore, for the cylinder to have height h and be centered at the origin, you will use the parameterization:

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

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

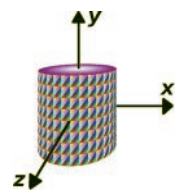

FIGURE 4.16 Cylinder.

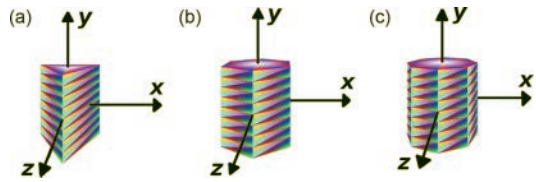

FIGURE 4.17 Triangular, hexagonal, and octagonal prisms.

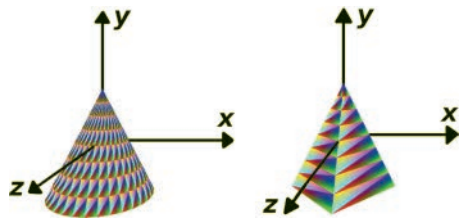

FIGURE 4.18 Cone and pyramid.

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

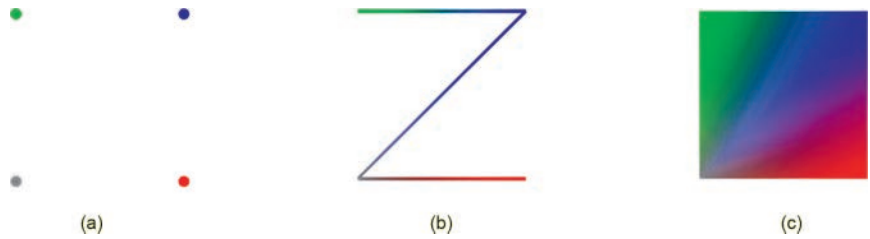

FIGURE 4.19 Rendering the six vertices of a Rectangle Geometry with a point material (a), line material (b), and surface material (c).

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
```

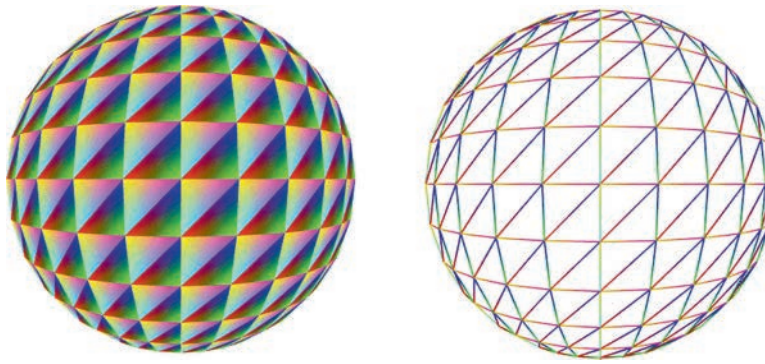

FIGURE 4.20 Rendering a sphere with triangles and as a wireframe.

```
        # 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 **getDescendantList** 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:

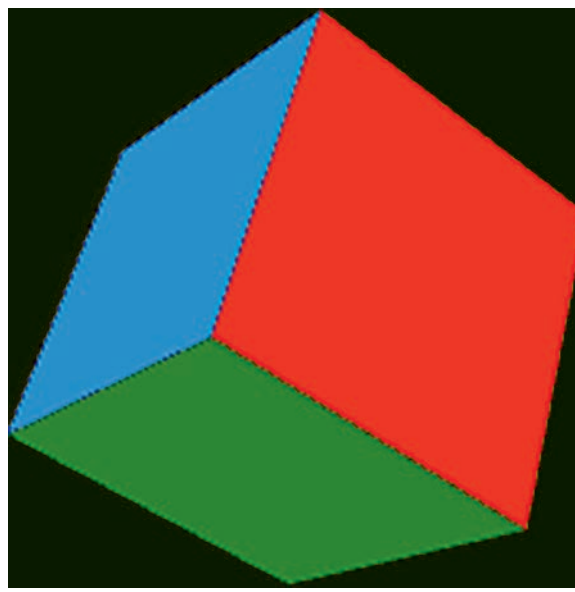

FIGURE 4.21 Rendering a spinning cube with the graphics framework classes.

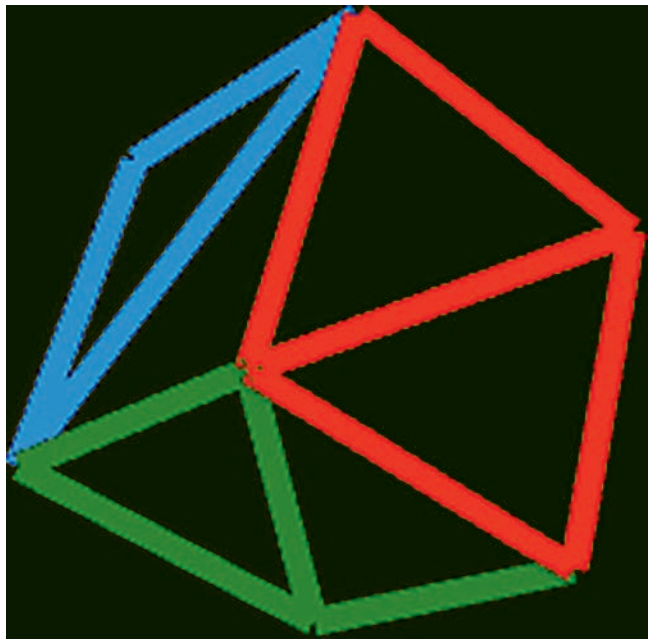

FIGURE 4.22 Rendering a cube with alternate material properties.

```
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.

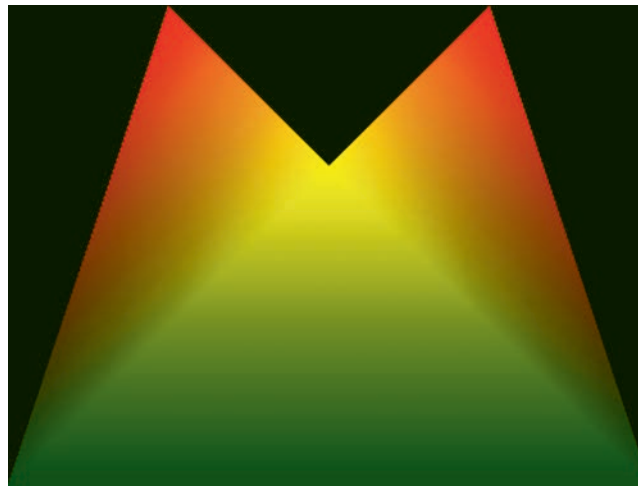

FIGURE 4.23 A custom geometry.

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()
```

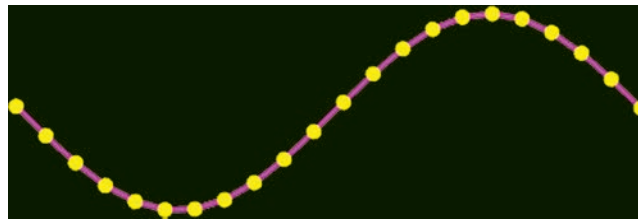

FIGURE 4.24 A custom geometry with data generated from a function.

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.

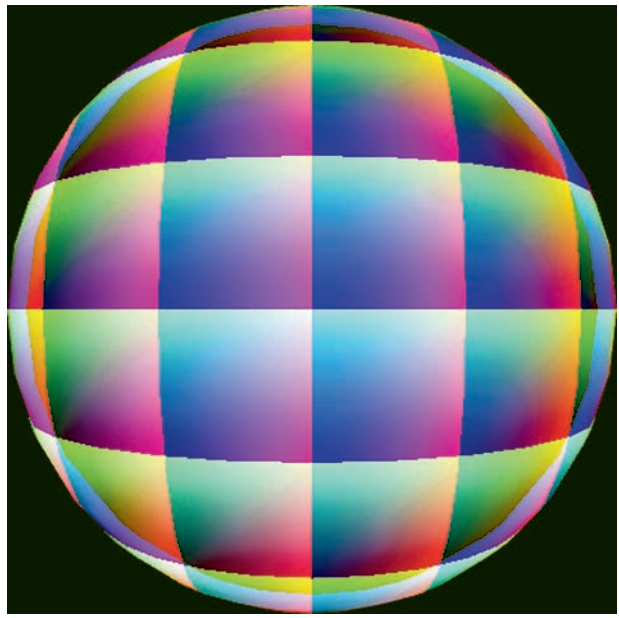

FIGURE 4.25 Coloring the surface of a sphere based on point coordinates.

```
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.

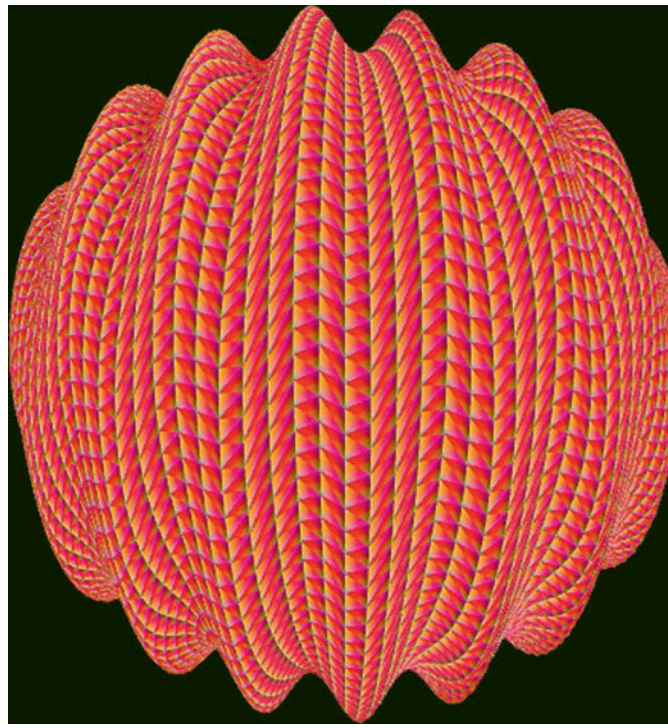

FIGURE 4.26 A sphere with periodic displacement and color shifting.

```
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()
```

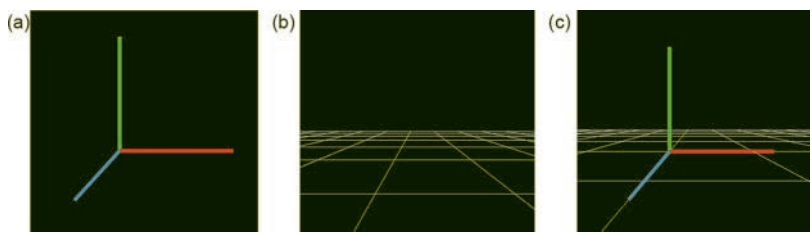

FIGURE 4.27 Coordinate axes (a), a grid (b), and in combination (c).

```
        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 )
```

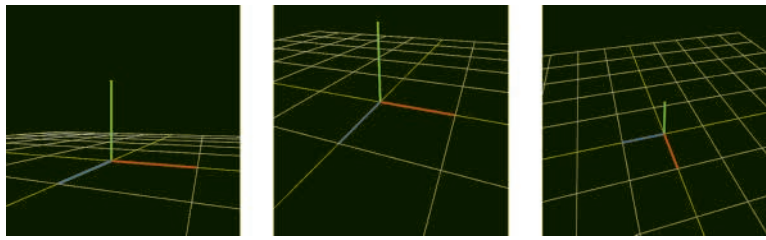

FIGURE 4.28 Multiple views of the coordinate axes and grid.

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.

CHAPTER 5

Textures

In many computer graphics applications, you will want to use *textures*: images applied to surfaces of geometric shapes in three-dimensional scenes, such as Figure 5.1, which shows an image applied to a box to create the appearance of a wooden crate. Textures and some related concepts (such as UV coordinates) were briefly mentioned in Chapter 1.

FIGURE 5.1 A wooden crate.

DOI: 10.1201/9781003181378-5

In this chapter, you will learn all the details necessary to use textures in your scenes, including creating a **`Texture`** class that handles uploading pixel data to the graphics processing unit (GPU), assigning UV coordinates to geometric objects, associating textures to uniform variables, and sampling pixel data from a texture in a shader. Once you have learned the basics, you will learn how to create animated effects such as blending and distortion. In the last part of this chapter, there are some optional sections that introduce advanced techniques, such as procedurally generating textures, using text in scenes, rendering the scene to a texture, and applying postprocessing effects such as pixelation and vignette shading. In theory, textures may also be one-dimensional and three-dimensional, but the discussion in this chapter will be limited to two-dimensional textures.

5.1 A TEXTURE CLASS

In OpenGL, a *texture object* is a data structure that stores pixel data from an image. In addition, a texture object stores information about which algorithms to use to determine the colors of the surface the texture is being applied to; this requires some type of computation whenever the texture and the surface are different sizes. In this section, you will create a **`Texture`** class to more easily manage this information and perform related tasks, similar to the motivation for creating an **`Attribute`** class in Chapter 2. Working with textures involves some additional OpenGL functions, introduced and discussed in this section.

Similar to the process for working with vertex buffers, you must first identify an available texture name or reference, and bind it to a particular type of target. OpenGL functions that affect a texture in some way do not directly reference a particular texture object; they also contain a target parameter and affect the texture object currently bound to that target, similar to the use of **`glBindBuffer`** and GL_ARRAY_BUFFER when working with attributes.

glGenTextures(*textureCount*)

Returns a set of nonzero integers representing available texture references. The number of references returned is specified by the integer parameter *textureCount*.

glBindTexture(*bindTarget, textureRef*)

The texture referenced by the parameter *textureRef* is bound to the target specified by the parameter *bindTarget*, whose value is an OpenGL

constant such as GL_TEXTURE_1D or GL_TEXTURE_2D (for one-dimensional and two-dimensional textures, respectively). Future OpenGL operations affecting the same *bindTarget* will be applied to the referenced texture.

Another necessary task is to upload the corresponding pixel data to the GPU. In addition to the raw pixel data itself, there is additional information required to parse this data, including the resolution (width and height) and precision (number of bits used in the components of each color) of the image. Some image formats (such as JPEG) do not support transparency, and thus, each pixel only stores three-component RGB values. Image formats that do support transparency (such as PNG) also store alpha values, and thus, each pixel stores four-component RGBA values. All of this information (and more) must be transmitted to the GPU along with the pixel data in order for it to be parsed properly; this is handled by the following OpenGL function:

glTexImage2D(*bindTarget, level, internalFormat, width, height, border, format, type, pixelData*)

Create storage and upload *pixelData* for the texture currently bound to *bindTarget*. The dimensions of the image are given by the integers *width* and *height*. The parameter *border* is usually 0, indicating that the texture has no border. The parameter *level* is usually 0, indicating this is the base image level in the associated mipmap image. The other parameters describe how the image data is stored in memory. The format of the data stored in *pixelData* and the desired format that should be used by the GPU are defined by the parameters *format* and *internalFormat*, respectively, and are specified using OpenGL constants such as GL_RGB or GL_RGBA. The parameter *type* indicates the precision used for the color data and is often GL_UNSIGNED_BYTE, indicating that 8 bits are used for each component.

When applying a texture to a surface, the pixels of the texture (referred to as *texels*) are usually not precisely the same size as the pixels in the rendered image of the scene. If the texture is smaller than the area of the surface it is being applied to, then the texture needs to be scaled up, a process called *magnification*. This requires an algorithm to determine a color for each pixel on the surface based on the colors of the texels. The simplest

FIGURE 5.2 Original image (left), magnified using nearest neighbor (center) and bilinear (right) filters.

algorithm is *nearest neighbor filtering*, which assigns the pixel the color of the closest texel; this will result in a blocky or pixelated appearance, as illustrated in Figure 5.2. Another algorithm is *bilinear filtering*, which interpolates the colors of the four closest texels (calculating a weighted average, similar to the rasterization process described in Chapters 1 and 2); the result is a smoother appearance, also illustrated in Figure 5.2.

If the texture is larger than the area of the surface it is being applied to, then the texture needs to be scaled down, a process called *minification*. After minification, the texels will be smaller than the pixels; multiple texels will overlap each pixel. As before, an algorithm is required to determine the color of each pixel from the texture data. A nearest neighbor filter can be used, where the pixel color is set to the color of the texel whose center is closest to the center of the pixel, but this may cause unwanted visual artifacts. A linear filter can be used, where the pixel color is calculated from a weighted average of the colors of the texels overlapping the pixel, but if a very large number of texels correspond to each pixel, calculating this average can take a relatively large amount of time. A computer graphics technique called *mipmapping* provides a more efficient approach to this calculation.

A *mipmap* is a sequence of images, each of which is half the width and height of the previous image; the final image in the sequence consists of a single pixel. A sample mipmap is illustrated in Figure 5.3. The images in the sequence only need to be calculated once, when the original image is uploaded to the GPU. Then, when a texture needs to be minified, the GPU can select the image from the corresponding mipmap that is closest to the desired size, and perform linear filtering on that image, a much more efficient calculation as it will involve only four texels.

Mipmap images are generated using the following OpenGL function:

glGenerateMipmap(*bindTarget*)

Generates a sequence of mipmap images for the texture currently bound to *bindTarget*.

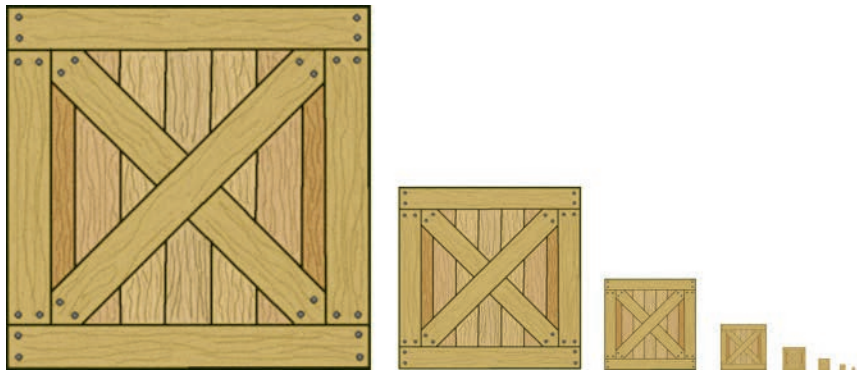

FIGURE 5.3 Example of a mipmap.

The next function enables you to select the filters used for magnification and minification, as well as configure other texture-related settings.

glTexParameteri(*bindTarget, parameterName, parameterValue*)

This is a general-purpose function used to set parameter values for the texture currently bound to *bindTarget*. In particular, the parameter with the symbolic name specified by the OpenGL constant *parameterName* is assigned the value specified by the OpenGL constant *parameterValue*.

For example, the magnification and minification filters are specified by the OpenGL constants GL_TEXTURE_MAG_FILTER and GL_TEXTURE_MIN_FILTER, respectively. Possible values for either include

- GL_NEAREST, for nearest-neighbor filtering
- GL_LINEAR, for linear filtering

When performing minification, mipmap-based filter options are also available, such as

- GL_NEAREST_MIPMAP_NEAREST, to select the closest mipmap and use nearest-neighbor filtering on it
- GL_NEAREST_MIPMAP_LINEAR, to select the closest mipmap and use linear filtering on it

- GL_LINEAR_MIPMAP_LINEAR, to select the two closest mipmaps, use linear filtering on each, and then linearly interpolate between the two results

Another use of the OpenGL function **glTexParameteri** relates to texture coordinates. Similar to the red, green, blue, and alpha components of a color, the components in texture coordinates range from 0 to 1. You can specify how to handle values outside this range by setting the parameters referenced by the OpenGL constants GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T. (You can think of (*s*, *t*) as an alternative notation for texture coordinates, which are typically written using the variables (*u*, *v*), although some distinguish (*s*, *t*) as *surface coordinates* that may extend beyond the range of texture coordinate values and must be projected into this range.) Some of the possible values for the wrap parameters are

- GL_REPEAT: only the fractional part of the values are used (the integer parts are ignored), equivalent to reducing the values modulo 1, creating a periodic, repeating pattern; this is the default setting.
- GL_CLAMP_TO_EDGE: clamps values to the range from 0 to 1 while avoiding sampling from the border color set for the texture.
- GL_CLAMP_TO_BORDER: when values are outside the range from 0 to 1, the value returned by sampling from a texture will be the border color set for the texture.

The effects of these wrap settings are illustrated in Figure 5.4.

Finally, before creating the **`Texture`** class, it is important to understand how the Pygame package handles image data. In Pygame, an object called a *surface* represents images and stores pixel data. Surfaces can be created

FIGURE 5.4 Texture wrap settings: repeat, clamp to edge, clamp to border.

by loading an image from a file or by manipulating pixel data directly. There is also a useful surface method called **tostring**, which converts surface pixel data to a string buffer that can be used to upload data to the GPU. In this framework, you will always convert the surface data to a string buffer containing RGBA data, and therefore, the *format* and *internalFormat* parameters of the **glTexImage2D** function can always be set to GL_RGBA.

With this knowledge, you are ready to implement the **Texture** class. In the **core** folder, create a new file called **texture.py** with the following code:

```
import pygame
from OpenGL.GL import *

class Texture(object):

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

        # pygame object for storing pixel data;
        #   can load from image or manipulate directly
        self.surface = None

        # reference of available texture from GPU
        self.textureRef = glGenTextures(1)

        # default property values
        self.properties = {
            "magFilter" : GL_LINEAR,
            "minFilter" : GL_LINEAR_MIPMAP_LINEAR,
            "wrap"      : GL_REPEAT
        }

        # overwrite default property values
        self.setProperties(properties)

        if fileName is not None:
            self.loadImage(fileName)
            self.uploadData()

    # load image from file
    def loadImage(self, fileName):
        self.surface = pygame.image.load(fileName)
```

```
    # set property values
    def setProperties(self, props):
        for name, data in props.items():
            if name in self.properties.keys():
                self.properties[name] = data
            else: # unknown property type
                raise Exception(
                      "Texture has no property with
                         name: " + name)

    # upload pixel data to GPU
    def uploadData(self):

        # store image dimensions
        width = self.surface.get_width()
        height = self.surface.get_height()


        # convert image data to string buffer
        pixelData = pygame.image.tostring(self.
             surface, "RGBA", 1)

        # specify texture used by the following
             functions
        glBindTexture(GL_TEXTURE_2D, self.textureRef)

        # send pixel data to texture buffer
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
                     width, height, 0, GL_RGBA,
                     GL_UNSIGNED_BYTE, pixelData)

        # generate mipmap image from uploaded pixel
            data
        glGenerateMipmap(GL_TEXTURE_2D)

        # specify technique for magnifying/minifying
            textures
        glTexParameteri(GL_TEXTURE_2D,
             GL_TEXTURE_MAG_FILTER,
                         self.properties["magFilter"] )
        glTexParameteri(GL_TEXTURE_2D,
             GL_TEXTURE_MIN_FILTER,
                         self.properties["minFilter"] )
```

```
# specify what happens to texture coordinates
# outside range [0, 1]
glTexParameteri(GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_S,
                  self.properties["wrap"] )
glTexParameteri(GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_T,
                  self.properties["wrap"] )

# set default border color to white;
#   important for rendering shadows
glTexParameterfv(GL_TEXTURE_2D,
   GL_TEXTURE_BORDER_COLOR, [1,1,1,1])
```

Note that the constructor for the class is designed so that creating a surface object from an image file is optional; this allows for the possibility of creating a surface object by other means and assigning it to a texture object directly, which will be explored later in this chapter.

5.2 TEXTURE COORDINATES

Texture coordinates (also known as UV coordinates) are used to specify which points of an image correspond to which vertices of a geometric object. Each coordinate ranges from 0 to 1, with the point (0,0) corresponding to the lower-left corner of the image, and the point (1,1) corresponding to the upper-right corner of the image. Typically, these values are passed from the vertex shader to the fragment shader (automatically interpolated for each fragment), and then, the UV coordinates are used to select a point on the texture from which the color of the fragment will be generated. Figure 5.5 shows a sample image and two of the many different

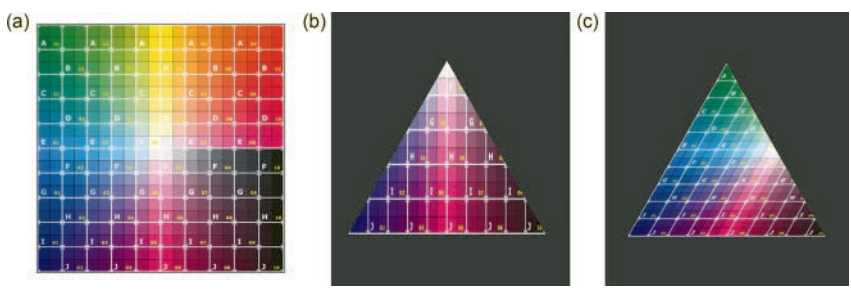

FIGURE 5.5 Texture coordinates used to apply an image (a) to a triangle (b, c).

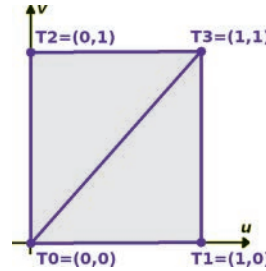

FIGURE 5.6 Texture coordinates for a rectangle shape.

ways that UV coordinates can be applied to the vertices of a triangle, in order to map different parts of the image to the triangle.

The next step will be to add a new attribute containing UV coordinates to the previously created geometry classes: rectangles, boxes, polygons, and parametric surfaces. The shader code that will be created later in this chapter will access this data through a shader variable named **`vertexUV`**.

5.2.1 Rectangles

The simplest class to add UV coordinates to is the **`Rectangle`** class. The four vertices of the rectangle correspond to the four corners of a texture; coordinates will be assigned as shown in Figure 5.6.

It will be important to keep in mind that UV coordinates for each vertex need to be stored in a list in the same order as the vertex positions. In the file **`rectangleGeometry.py`** in the **`geometry`** folder, add the following code to the initialization function:

```
# texture coordinates
T0, T1, T2, T3 = [0,0], [1,0], [0,1], [1,1]
uvData = [ T0,T1,T3, T0,T3,T2 ]
self.addAttribute("vec2", "vertexUV", uvData)
```

At this point, this code cannot yet be tested. If you wish to test this code as soon as possible, you may skip ahead to Section 5.3 on using textures in shaders, or you may continue implementing UV coordinates for other geometric shapes in what follows.

5.2.2 Boxes

Since each of the six sides of a box is a rectangle, and the vertices were ordered in the same way, the code for adding UV coordinates to the

BoxGeometry class is straightforward. In the file **boxGeometry.py** in the **geometry** folder, add the following code to the initialization function:

```
# texture coordinates
T0, T1, T2, T3 = [0,0], [1,0], [0,1], [1,1]
uvData = [ T0,T1,T3, T0,T3,T2 ] * 6
self.addAttribute("vec2", "vertexUV", uvData)
```

When you are able to render this shape with a grid texture later on in this chapter, it will appear similar to Figure 5.7.

5.2.3 Polygons

To apply a texture to a polygon without distorting the texture, you will use UV coordinates corresponding to the arrangement of the vertices of the polygon, effectively appearing to "cut out" a portion of the texture, as illustrated for various polygons in Figure 5.8.

Recall that the vertices of the polygon are arranged around the circumference of a circle centered at the origin with a user-specified radius, and are

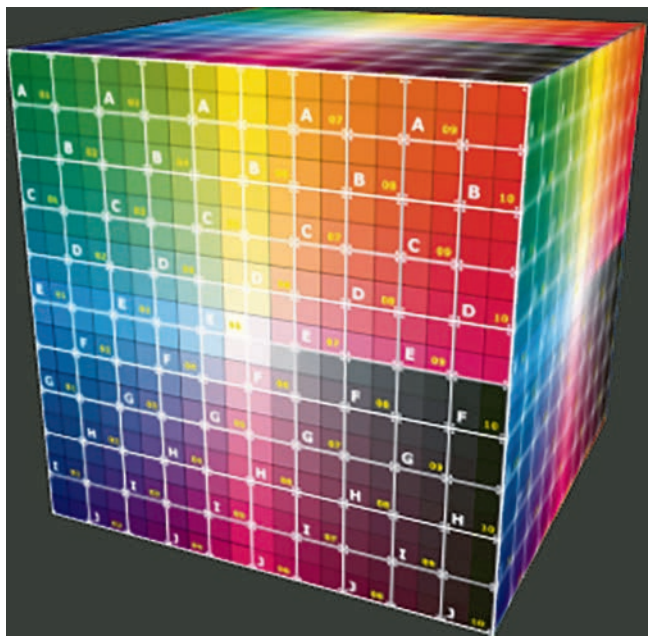

FIGURE 5.7 Box geometry with grid texture applied.

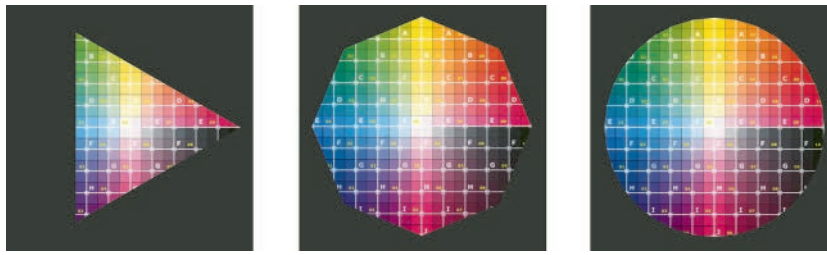

FIGURE 5.8 Textured polygons with 3 sides, 8 sides, and 32 sides.

parameterized by the sine and cosine functions. To stay within the range of UV coordinates (from 0 to 1), you will use a circle of radius 0.5, centered at the point (0.5, 0.5). To implement this, in the file **polygonGeometry.py** in the **geometry** folder, add the following code immediately before the **for** loop:

```
uvData   = []
uvCenter = [0.5, 0.5]
```

Within the **for** loop, add the following code:

```
uvData.append( uvCenter )
uvData.append( [     cos(n*A)*0.5 + 0.5,
    sin(n*A)*0.5 + 0.5 ] )
uvData.append( [ cos((n+1)*A)*0.5 + 0.5,
    sin((n+1)*A)*0.5 + 0.5 ] )
```

Finally, following the for loop, add the following single line:

```
self.addAttribute("vec2", "vertexUV", uvData)
```

5.2.4 Parametric Surfaces

The positions of vertices on a parametric surface are calculated using a parametric function, which maps a two-dimensional rectangular region onto a surface in three-dimensional space. Therefore, UV coordinates can be generated by rescaling points in the domain of the parametric function to values in the domain of texture coordinates: from 0 to 1. This will be accomplished in a sequence of steps, similar to the process by which vertex positions were generated. The first step will be to generate UV coordinates for each point in the domain; these will be stored in a two-dimensional

list called **uvs**. Then, this data will be grouped into triangles and stored in a two-dimensional list called **uvData**, which will be used for the corresponding attribute.

To begin, in the file **parametricGeometry.py** in the **geometry** folder, add the following code after the **for** loop that generates data for the list named **positions**. Note that dividing each of the u and v index values by the corresponding resolution values produces a sequence of evenly spaced values between 0 and 1.

```
uvs = []
uvData = []

for uIndex in range(uResolution+1):
    vArray = []
    for vIndex in range(vResolution+1):
        u = uIndex/uResolution
        v = vIndex/vResolution
        vArray.append( [u, v] )
    uvs.append(vArray)
```

Next, to group this data into triangles, add the following code in the nested **for** loop that follows, after the line of code where data is added to the **colorData** list.

```
# uv coordinates
uvA = uvs[xIndex+0][yIndex+0]
uvB = uvs[xIndex+1][yIndex+0]
uvD = uvs[xIndex+0][yIndex+1]
uvC = uvs[xIndex+1][yIndex+1]
uvData += [uvA,uvB,uvC, uvA,uvC,uvD]
```

Finally, after the code where the other attributes are created, add the following line of code:

```
self.addAttribute("vec2", "vertexUV", uvData)
```

After you are able to test this code, you will be able to create textured parametric surfaces, like those shown in Figure 5.9. No special modifications need to be made to the **CylindricalGeometry** class, since the **merge** function used by that class copies all attributes from the **PolygonGeometry** objects used for the top and bottom sides, including the recently added UV coordinates.

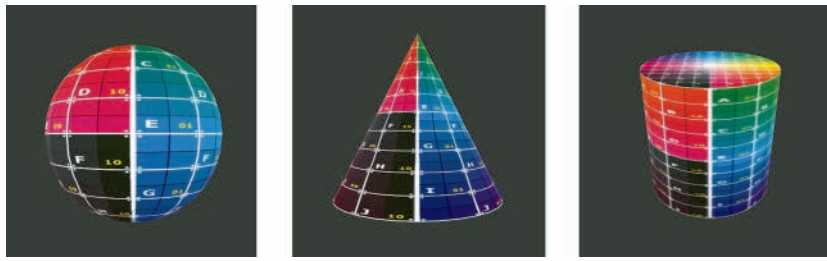

FIGURE 5.9 A sphere, cone, and cylinder with grid image textures applied.

5.3 USING TEXTURES IN SHADERS

Once you have implemented UV coordinates for at least one geometric shape, the next step is to focus on adding shader code that supports this new functionality. *Sampling* refers to the process of calculating a color based on the information stored in a texture object; this calculation is performed by a *texture unit*, to which a texture object is assigned. To perform sampling in a shader program, you will use a uniform variable of type **`sampler2D`**, which stores a reference to a texture unit. You will also use the function **`texture2D`**, which takes a sampler2D and a vector (containing UV coordinates) as inputs, and performs the sampling calculation.

There are a limited number of texture units available for use by a shader at any given time. In OpenGL 3.0 and above, there are a minimum of 16 texture units available. This number may be greater for some systems depending on the implementation of OpenGL, but only 16 texture units are guaranteed to exist. It may be possible to create many more texture objects than this, depending on the size of the image data in each texture object and the storage capacity of the GPU. However, when rendering any particular geometric primitive, you may be limited to using the data from at most 16 of these texture objects, as they are accessed through the texture units.

In order to work with multiple texture units (as you will later on in this chapter), you will need to use the following OpenGL function:

glActiveTexture(*textureUnitRef*)

Activates a texture unit specified by the parameter *textureUnitRef*; any texture objects that are bound using the function glBindTexture are assigned to the active texture unit. The value of *textureUnitRef*

is an OpenGL constant GL_TEXTURE0, GL_TEXTURE1, GL_TEXTURE2, etc., with maximum value depending on the number of texture units supported by the particular OpenGL implementation. These constants form a consecutive set of integers, so that for any positive integer *n*, GL_TEXTURE*n* is equal to GL_TEXTURE0+*n*.

Due to the intermediate step of assigning the texture object to a texture unit that whose reference is stored in a shader variable, each **Uniform** object that represents a sampler2D will store a list in its **data** field containing the reference for the texture object, followed by the number of the texture unit to be used. Unlike texture object references, which must be generated by the OpenGL function **glGenTextures**, texture units may be chosen by the developer. In addition, you will need to use the previously discussed OpenGL function **glUniform1i** to upload the number of the texture unit (not the OpenGL constant) to the sampler2D variable. In other words, using the previously established notation, if the active texture unit is GL_TEXTURE*n*, then the integer *n* should be uploaded to the sampler2D variable.

With this knowledge, you are ready to modify the **Uniform** class. In the file **uniform.py** in the **core** directory, add the following code to the **if** statement block in the **uploadData** function:

```
elif self.dataType == "sampler2D":
    textureObjectRef, textureUnitRef = self.data
    # activate texture unit
    glActiveTexture( GL_TEXTURE0 + textureUnitRef )
    # associate texture object reference to currently
    #   active texture unit
    glBindTexture( GL_TEXTURE_2D, textureObjectRef )
    # upload texture unit number (0...15) to
    #   uniform variable in shader
    glUniform1i( self.variableRef, textureUnitRef )
```

Since certain image formats (such as PNG images) use transparency, you will next make a few additions to the **Renderer** class to support this feature. Blending colors in the color buffer based on alpha values is enabled with the OpenGL function **glEnable**. The formula used when blending colors needs to be specified as well, with the use of the following OpenGL function:

glBlendFunc(*sourceFactor, destinationFactor*)

Specifies values to be used in the formula applied when blending an incoming "source" color with a "destination" color already present in the color buffer. The components of the source and destination colors are multiplied by values specified by the corresponding parameters, *sourceFactor* and *destinationFactor*. The values of these parameters are calculated based on the specified OpenGL constant, some possible values including GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA, GL_ZERO, and GL_ONE.

The formula that will be used for blending will be a weighted average based on the alpha value of the source color. To implement this, in the file **`renderer.py`** in the **`core`** folder, add the following code to the initialization function:

```
glEnable(GL_BLEND)
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

When using alpha blending, careful attention must be paid to the render order, as described in Chapter 1, in Section 1.2.4 on pixel processing. Figure 5.10 illustrates an image of a circle applied to two rectangle geometries; the pixels of the image outside the circle are completely transparent. The top-left circle is nearer to the camera than the bottom-right circle. The left side of the figure shows the circles rendered correctly. The right side of the figure shows the result of rendering the front circle before the back circle: the transparent pixels of the front image are blended with

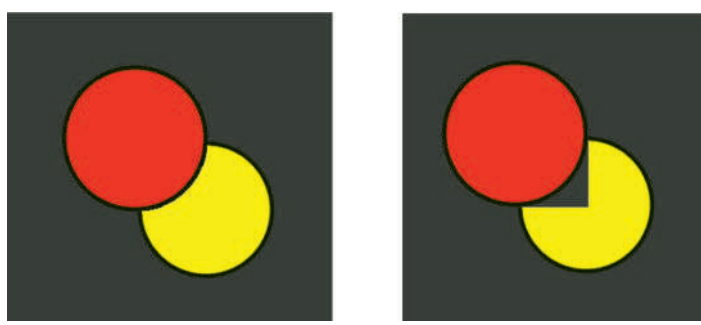

FIGURE 5.10 Transparency rendering issues caused by render order.

the background color and stored in the color buffer, and then, when the back image is processed later, the fragments that appear behind the front circle—including those behind transparent pixels—are automatically discarded, as they are a greater distance away from the camera. This causes part of the back circle to be incorrectly removed from the final rendered image.

The next step is to create an extension of the **`Material`** class that incorporates UV coordinates, a uniform sampler2D variable, and the GLSL function **`texture2D`**. Unlike the basic materials created in the previous chapter, which included point-based and line-based versions, the only material developed here will be a surface-based version. Therefore, there only needs to be a single class that will contain the shader code, add the **`Uniform`** objects, initialize the render settings, and implement the **`updateRenderSettings`** function. In addition, this material will not make use of the vertex color attribute, but will keep the base color uniform, which can be used to tint textures by a particular color. This shader will also include uniform variables **`repeatUV`** and **`offsetUV`**, which can be used to scale and translate UV coordinates, allowing textures to appear multiple times on a surface as illustrated by Figure 5.11. Finally, the

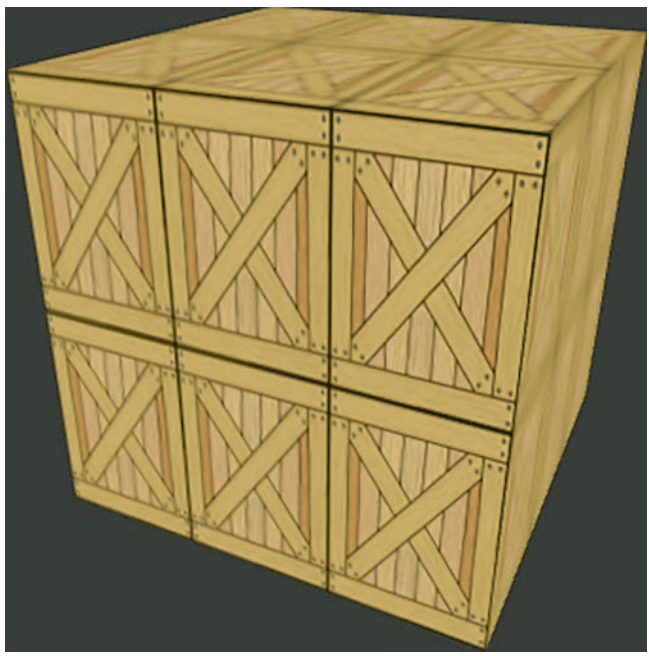

FIGURE 5.11 Applying a repeated crate texture to a box geometry.

fragment shader will make use of the GLSL command **discard**, which will be used to stop processing fragments with small alpha values. This will automatically resolve some of the rendering issues previously described when the pixels are transparent, but not when the pixels are translucent (partially transparent).

To implement this texture-supporting material, in the **material** folder, create a new file named **textureMaterial.py** and add the following code:

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

    def __init__(self, texture, properties={}):

        vertexShaderCode = """
        uniform mat4 projectionMatrix;
        uniform mat4 viewMatrix;
        uniform mat4 modelMatrix;
        in vec3 vertexPosition;
        in vec2 vertexUV;
        uniform vec2 repeatUV;
        uniform vec2 offsetUV;
        out vec2 UV;

        void main()
        {
            gl_Position = projectionMatrix *
                viewMatrix *
                modelMatrix * vec4(vertexPosition,
                  1.0);
            UV = vertexUV * repeatUV + offsetUV;
        }
        """

        fragmentShaderCode = """
        uniform vec3 baseColor;
        uniform sampler2D texture;
        in vec2 UV;
        out vec4 fragColor;

        void main()
        {
```

```
            vec4 color = vec4(baseColor, 1.0) *
                 texture2D(texture, UV);
            if (color.a < 0.10)
                discard;

            fragColor = color;
        }
        """

        super().__init__(vertexShaderCode,
            fragmentShaderCode)

        self.addUniform("vec3", "baseColor", [1.0,
           1.0, 1.0])
        self.addUniform("sampler2D", "texture",
           [texture.textureRef, 1])
        self.addUniform("vec2", "repeatUV", [1.0,
           1.0])
        self.addUniform("vec2", "offsetUV", [0.0,
           0.0])
        self.locateUniforms()

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

        self.setProperties(properties)

    def updateRenderSettings(self):

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

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

        glLineWidth(self.settings["lineWidth"])
```

With this class finished, the graphics framework now supports texture rendering.

5.4 RENDERING SCENES WITH TEXTURES

At this point, you can now create applications where textures are applied to surfaces. For convenience, in the **test-template.py** file you created in the last chapter, add the following two import statements. These correspond to the new classes that you have created in this chapter, and will be used in many of the examples that follow.

```
from core.texture import Texture
from material.textureMaterial import TextureMaterial
```

Your first texture-based application will be a simple example—applying a grid image to a rectangle shape—to verify that the code entered up to this point works correctly. To continue to keep your codebase organized, in your main project folder, add a new folder named **images**; this is where all image files will be stored. The image files used in this book will be available to download from an online project repository. Alternatively, you may replace the provided images with image files of your own choosing.

In your main project folder, make a copy of the file **test-template.py** and name it **test-5-1.py**. In this new file, change the position of the camera to (0, 0, 2) and replace the lines of code where the geometry and material variables are defined with the following code:

```
geometry = RectangleGeometry()
grid = Texture("images/grid.png")
material = TextureMaterial(grid)
```

When you run this application, you should see a result similar to that of Figure 5.12.

As you have seen earlier in this chapter, the same image used in Figure 5.12 can be applied to other surfaces, such as Spheres, Cones, and Cylinders, as seen in Figure 5.9; the result is that the image will appear stretched in some locations, and compressed in other locations. A *spherical texture* is an image that appears correctly proportioned when applied to a sphere; such an image will appear distorted in some areas (such as being stretched out near the top and bottom) when viewed as a rectangular image. One well-known example is a map of the Earth, illustrated on the left side of Figure 5.13 and applied to a sphere on the right side of the figure.

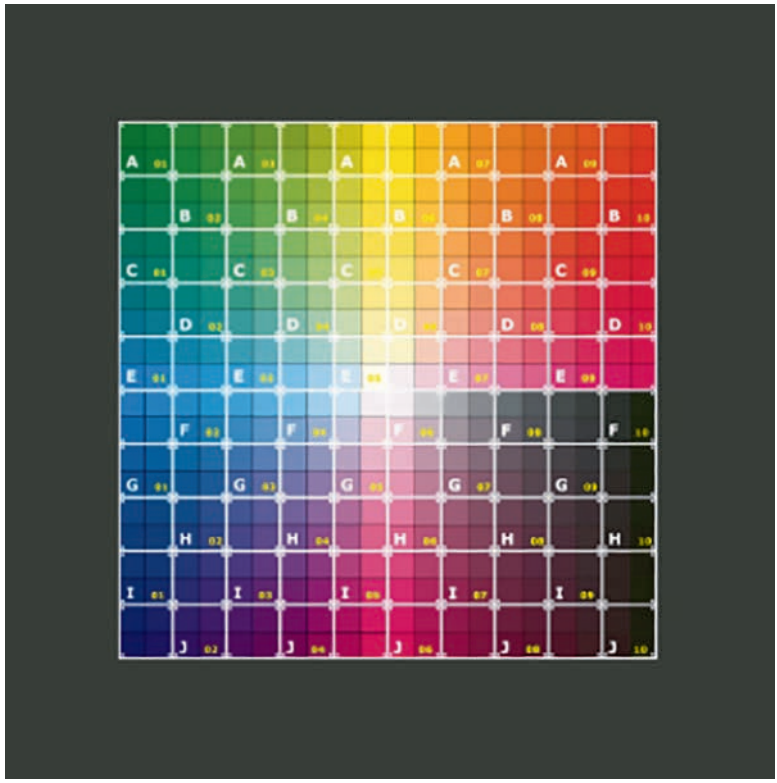

FIGURE 5.12 Texture applied to a rectangle.

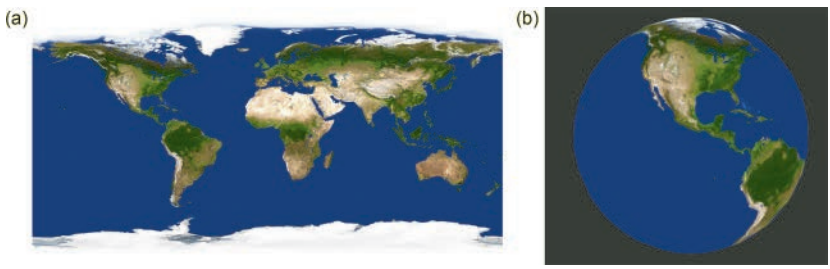

FIGURE 5.13 A spherical texture of the Earth's surface (a), applied to a sphere (b).

Spherical textures can be particularly useful in generating virtual environments. A *skysphere* is a spherical texture used to create a background in a three-dimensional scene. As the name suggests, the texture is applied to a large sphere that surrounds the virtual camera; the back sides of the triangles must be set to render. The same concept can be used

FIGURE 5.14 Multiple views of a scene featuring a skysphere texture.

to create backgrounds from textures based on other shapes; *skybox* and *skydome* textures are also frequently used in practice together with the corresponding shape. The next application illustrates how to use a skysphere texture, combined with a large plane containing a repeating grass texture, and adds a **MovementRig** object to enable the viewer to look around the scene. Figure 5.14 shows the resulting scene, viewed from multiple angles: from left to right, the camera rotates right and tilts upward.

To implement this scene, make a copy of the file **test-template.py** and name it **test-5-2.py**. Add the following import statements:

```
from geometry.rectangleGeometry import
RectangleGeometry
from geometry.sphereGeometry import SphereGeometry
from extras.movementRig import MovementRig
```

In the initialize function, replace the code in that function starting from the line where the camera position is set, with the following code:

```
self.rig = MovementRig()
self.rig.add( self.camera )
self.scene.add( self.rig )
self.rig.setPosition( [0, 1, 4] )
skyGeometry = SphereGeometry(radius=50)
skyMaterial = TextureMaterial( Texture("images/
   sky-earth.jpg") )
sky = Mesh( skyGeometry, skyMaterial )
self.scene.add( sky )
grassGeometry = RectangleGeometry(width=100,
   height=100)
```

```
grassMaterial = TextureMaterial( Texture("images/
   grass.jpg"),
                                    {"repeatUV":
                                        [50,50]} )
grass = Mesh( grassGeometry, grassMaterial )
grass.rotateX(-3.14/2)
self.scene.add( grass )
```

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

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

When you run this application, you can use the keyboard to navigate around the scene and see results similar to those in Figure 5.14.

5.5 ANIMATED EFFECTS WITH CUSTOM SHADERS

In this section, you will write some custom shaders and materials to create animated effects involving textures. In the first example, you will create a rippling effect in a texture, by adding a sine-based displacement to the V component of the UV coordinates in the fragment shader. The overall structure of the application will be similar to the applications involving custom materials from the previous chapter. Besides the use of the **sin** function, the other significant addition to the code is the inclusion of a uniform float variable that stores the time that has elapsed since the application began; this value will be incremented in the **update** function.

To create this effect, make a copy of the file **test-template.py** and name it **test-5-3.py**. Add the following import statements:

```
from geometry.rectangleGeometry import
RectangleGeometry
from material.material import Material
```

Then, in the **initialize** function, replace the code in that function, starting from the line where the camera position is set, with the following code:

```
self.camera.setPosition( [0, 0, 1.5] )
vertexShaderCode = """
uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;
```

```
in vec3 vertexPosition;
in vec2 vertexUV;
out vec2 UV;

void main()
{
    gl_Position = projectionMatrix * viewMatrix *
       modelMatrix *
                    vec4(vertexPosition, 1.0);
    UV = vertexUV;
}
"""
fragmentShaderCode = """
uniform sampler2D texture;
in vec2 UV;
uniform float time;
out vec4 fragColor;

void main()
{
    vec2 shiftUV = UV + vec2(0, 0.2 * sin(6.0*UV.x +
       time));
    fragColor = texture2D(texture, shiftUV);
}
"""
gridTex = Texture("images/grid.png")
self.waveMaterial = Material(vertexShaderCode,
   fragmentShaderCode)
self.waveMaterial.addUniform("sampler2D", "texture",
      [gridTex.textureRef, 1])
self.waveMaterial.addUniform("float", "time", 0.0)
   self.waveMaterial.locateUniforms()
geometry = SphereGeometry(radius=0.5)
self.mesh = Mesh( geometry, self.waveMaterial )
self.scene.add( self.mesh )
```

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

```
self.waveMaterial.uniforms["time"].data += self.
deltaTime
```

When you run this application, you should see an animated effect as illustrated in Figure 5.15. Note that in this and the following examples, the customized material is applied to a rectangle, but can just as easily be applied to other geometric shapes, with visually interesting results.

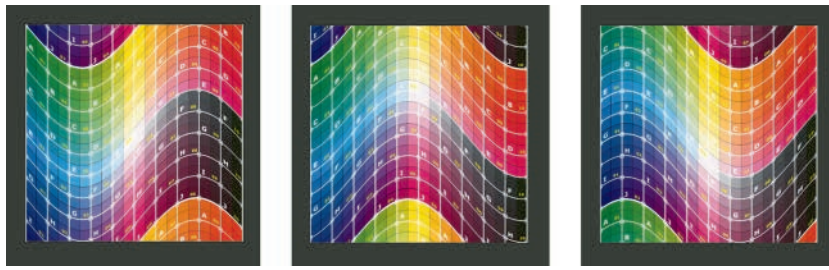

FIGURE 5.15 Images from an animated ripple effect on a grid texture.

In the next example, you will cyclically blend between two different textures. This example also illustrates the importance of assigning texture objects to different texture units. The main idea of this shader is to sample colors from both textures at each fragment, and then, linearly interpolate between these colors to determine the fragment color. The amount of interpolation varies periodically between 0 and 1, calculated using the absolute value of the sine of the time that has elapsed since the application was started.

Since this example is similar in structure to the previous example, make a copy of the file **test-5-3.py** and name it **test-5-4.py**. In the **initialize** function, replace the code in that function, starting from the line where the fragment shader code is created, with the following code:

```
fragmentShaderCode = """
uniform sampler2D texture1;
uniform sampler2D texture2;
in vec2 UV;
uniform float time;
out vec4 fragColor;

void main()
{
    vec4 color1 = texture2D(texture1, UV);
    vec4 color2 = texture2D(texture2, UV);
    float s = abs(sin(time));
    fragColor = s * color1 + (1.0 - s) * color2;
}
"""
gridTex  = Texture("images/grid.png")
crateTex = Texture("images/crate.png")
```

```
self.blendMaterial = Material(vertexShaderCode,
   fragmentShaderCode)
self.blendMaterial.addUniform("sampler2D", "texture1",
        [gridTex.textureRef, 1])
self.blendMaterial.addUniform("sampler2D", "texture2",
         [crateTex.textureRef, 2])
self.blendMaterial.addUniform("float", "time", 0.0)
self.blendMaterial.locateUniforms()

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

Finally, in the **update** function, replace the line of code referencing the previous wave material with the following:

```
self.blendMaterial.uniforms["time"].data += self.
deltaTime
```

When you run this application, you should see an animated effect as illustrated in Figure 5.16.

In the final example in this section, you will again use two textures. One of these textures, shown in Figure 5.17, will be used to produce pseudo-random values (also called *noise*) to distort a texture over time, as shown in Figure 5.18. These values are generated by sampling red, green, or blue values from the noise texture, whose colors appear to be a random pattern, but whose components change continuously throughout the image. In this shader, the distortion effect is created by continuously shifting the UV coordinates, using them to obtain values from the noise texture, and then, the final fragment color is sampled from the image texture at the original UV coordinates offset by the noise value.

FIGURE 5.16 Images from an animated blend effect between a crate texture and a grid texture.

FIGURE 5.17 A "noise" texture used to generate pseudo-random values.

FIGURE 5.18 Images from an animated distortion effect applied to a grid texture.

Once again, this example is similar in structure to the previous examples. To begin, make a copy of the file **`test-5-3.py`** and name it **`test-5-5.py`**. In the **`initialize`** function, replace the code in that function, starting from the line where the fragment shader code is created, with the following code:

```
fragmentShaderCode = """
uniform sampler2D noise;
uniform sampler2D image;
```

```
in vec2 UV;
uniform float time;
out vec4 fragColor;

void main()
{
    vec2 uvShift = UV + vec2( -0.033, 0.07 ) * time;
    vec4 noiseValues = texture2D( noise, uvShift );
    vec2 uvNoise = UV + 0.4 * noiseValues.rg;
    fragColor = texture2D( image, uvNoise );
}
"""

noiseTex  = Texture("images/noise.png")
gridTex = Texture("images/grid.png")

self.distortMaterial = Material(vertexShaderCode,
   fragmentShaderCode)
self.distortMaterial.addUniform("sampler2D", "noise",
        [noiseTex.textureRef, 1])
self.distortMaterial.addUniform("sampler2D", "image",
        [gridTex.textureRef, 2])
self.distortMaterial.addUniform("float", "time", 0.0)
self.distortMaterial.locateUniforms()

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

Finally, in the **update** function, replace the line of code referencing the previous wave material with the following:

```
self.distortMaterial.uniforms["time"].data += self.
deltaTime
```

When you run this application, you should see an animated effect as illustrated in Figure 5.17. A shader such as this can be used to add realistic dynamic elements to an interactive three-dimensional scene. For example, by applying this shader to a texture such as the water or lava textures shown in Figure 5.19, one can create a fluid-like appearance.

5.6 PROCEDURALLY GENERATED TEXTURES

A *procedurally generated texture* is a texture that is created using a mathematical algorithm, rather than using an array of pixels stored in an image file. Procedural textures can yield noise textures (similar to Figure 5.17) and simulations of naturally occurring patterns such as clouds, water, wood, marble, and other patterns (similar to those in Figure 5.19).

The first step in generating a noise texture is to create a function that can produce pseudo-random values in a shader. This can be accomplished by taking advantage of the limited precision of the fractional part of floating-point numbers, which can be obtained using the GLSL function **`fract`**. While a function such as $\sin(x)$ is perfectly predictable, the output of the function $\text{fract}\left(235{,}711 \cdot \sin(x)\right)$ is effectively random for generating images. With regard to the number that $\sin(x)$ is being multiplied by, only the magnitude (and not the particular digits) is important. To produce random values across a two-dimensional region, one could define a similar function, such as $\text{fract}\left(235{,}711 \cdot \sin\left(14.337 \cdot x + 42.418 \cdot y\right)\right)$.

Next, you will create a sample application with a custom material that illustrates how this function can be used in a shader. To begin, make a copy of the file **`test-template.py`** and name it **`test-5-6.py`**. Add the following import statements:

```
from geometry.rectangleGeometry import
   RectangleGeometry
from material.material import Material
```

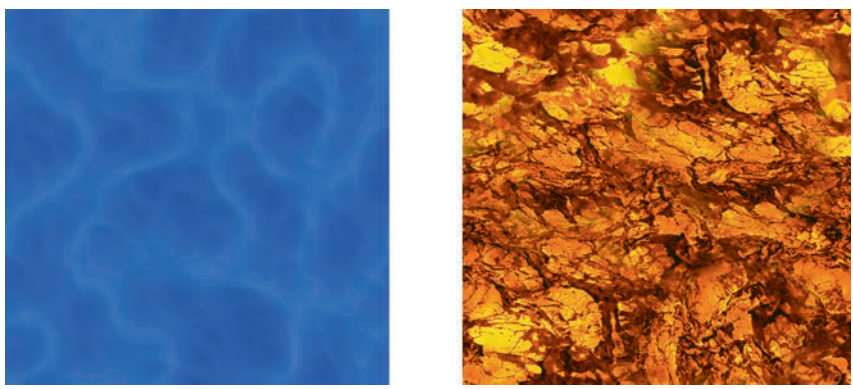

FIGURE 5.19 Water and lava textures.

Then, in the **initialize** function, replace the code in that function, starting from the line where the camera position is set, with the following code. Note that this code contains the first example of a function defined in GLSL, in this case the **random** function defined in the fragment shader.

```
self.camera.setPosition( [0, 0, 1.5] )

vsCode = """
uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;
in vec3 vertexPosition;
in vec2 vertexUV;
out vec2 UV;

void main()
{
    vec4 pos = vec4(vertexPosition, 1.0);
    gl_Position = projectionMatrix * viewMatrix *
       modelMatrix * pos;
    UV = vertexUV;
}
"""

fsCode = """
// return a random value in [0, 1]
float random(vec2 UV)
{
    return fract(235711.0 * sin(14.337*UV.x +
       42.418*UV.y));
}

in vec2 UV;
out vec4 fragColor;
void main()
{
    float r = random(UV);
    fragColor = vec4(r, r, r, 1);
}
"""

material = Material(vsCode, fsCode)
```

```
material.locateUniforms()

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

When you run this example, it will produce an image similar to the one illustrated in Figure 5.20, where the random value generated from the UV coordinates at each fragment are used for the red, green, and blue components of the fragment color, yielding a shade of gray. The overall image resembles the static pattern seen on older analog televisions resulting from weak broadcasting signals.

For the applications that follow, this texture is actually *too* random. The next step will be to scale up and round the UV coordinates used as inputs for the random function, which will produce a result similar to the left side of Figure 5.21, in which the corners of each square correspond to points whose scaled UV coordinates are integers, and the color of each square corresponds to the random value at the lower-left corner. Next, the random value (and thus the color) at each point will be replaced with

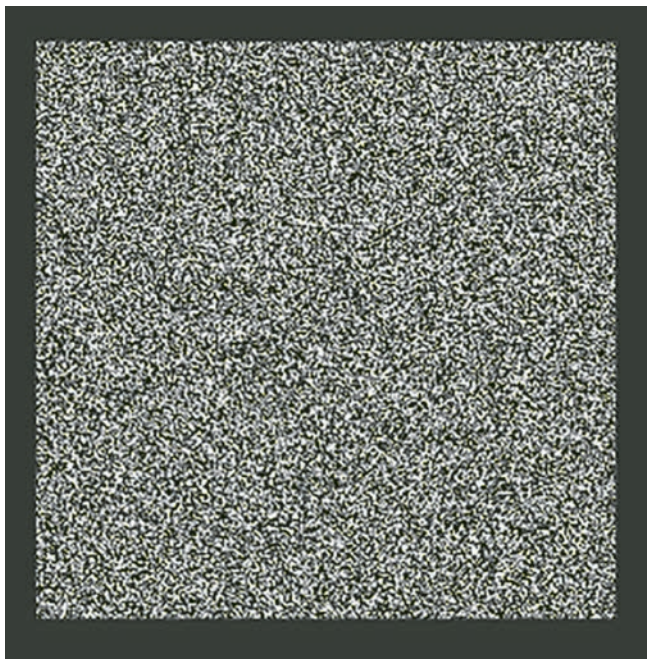

FIGURE 5.20 A randomly generated texture.

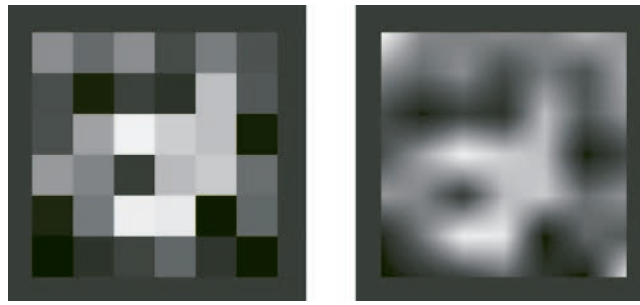

FIGURE 5.21 Scaled and smoothed random textures.

a weighted average of the random values at the vertices of the square in which the point is contained, thus producing a smoothed version of the left side of Figure 5.21, illustrated on the right side of Figure 5.21. The GLSL function **mix** will be used to linearly interpolate the two values at the bottom corners of each square, followed by interpolating the values at the top corners of each square and then interpolating these results.

To produce these images, first add the following functions to the code in the fragment shader:

```
float boxRandom(vec2 UV, float scale)
{
    vec2 iScaleUV = floor(scale * UV);
    return random(iScaleUV);
}

float smoothRandom(vec2 UV, float scale)
{
    vec2 iScaleUV = floor(scale * UV);
    vec2 fScaleUV = fract(scale * UV);
    float a = random(iScaleUV);
    float b = random(round(iScaleUV + vec2(1, 0)));
    float c = random(round(iScaleUV + vec2(0, 1)));
    float d = random(round(iScaleUV + vec2(1, 1)));
    return mix( mix(a, b, fScaleUV.x),
                mix(c, d, fScaleUV.x),
                fScaleUV.y );
}
```

Then, to produce the image on the left of Figure 5.21, in the **main** function in the fragment shader, replace the line of code where the variable **r** is declared with the following.

```
float r = boxRandom(UV, 6);
```

To produce the image on the right side of Figure 5.21, replace this line of code with the following.

```
float r = smoothRandom(UV, 6);
```

To produce a grayscale noise texture similar to that in Figure 5.17, you will combine a sequence of images similar to those shown on the right side of Figure 5.21; such a sequence is illustrated in Figure 5.22. The scale of the UV coordinates in each image in the sequence are doubled, resulting in square regions whose dimensions are half those from the previous image. The randomly generated values will be scaled by half at each stage as well, so that the finer details will contribute a proportional amount to the accumulated total. As this process is reminiscent of fractals—shapes that contain the same pattern at different scales—this function for generating random values will be called **`fractalRandom`**. The final texture generated by this function is illustrated in Figure 5.23.

FIGURE 5.22 A sequence of scaled and smoothed random textures.

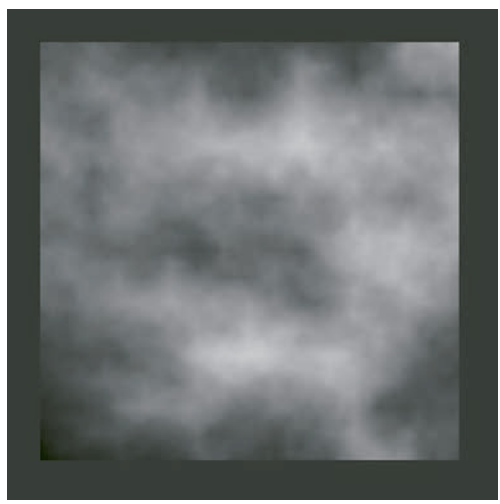

FIGURE 5.23 Fractal noise produced from combining images from Figure 5.22.

To produce the image in Figure 5.23, first add the following function to the code in the fragment shader:

```
// add smooth random values at different scales
//   weighted (amplitudes) so that sum is
       approximately 1.0
float fractalRandom(vec2 UV, float scale)
{
    float value = 0.0;
    float amplitude = 0.5;

    for (int i = 0; i < 6; i++)
    {
        value += amplitude * smoothRandom(UV, scale);
        scale *= 2.0;
        amplitude *= 0.5;
    }

    return value;
}
```

Then, in the **main** function in the fragment shader, replace the line of code where the variable **r** is declared with the following.

```
float r = fractalRandom(UV, 4);
```

While many other methods exist for producing random values, such as the Perlin noise and cellular noise algorithms, the functions here will be sufficient for our purposes. The next step will be to use these functions to produce random images simulating textures found in nature, such as those illustrated in Figure 5.24, which are meant to resemble clouds, lava, marble, and wood.

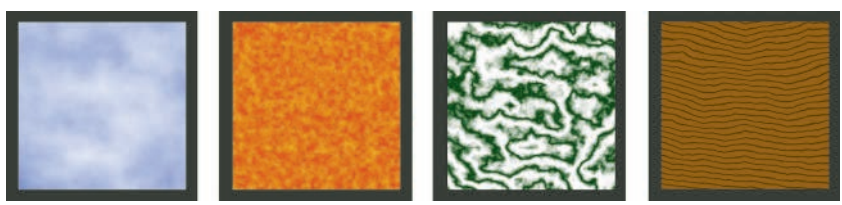

FIGURE 5.24 Procedurally generated textures: clouds, lava, marble, wood.

All of the examples from Figure 5.24 are generated by mixing two different colors, where the amount of interpolation is determined in part or in whole by the **fractalRandom** function. To create the cloud image, replace the code in the fragment shader with the following:

```
// clouds
float r = fractalRandom(UV, 5);
vec4 color1 = vec4(0.5, 0.5, 1, 1);
vec4 color2 = vec4(1, 1, 1, 1);
fragColor = mix( color1, color2, r );
```

To create the lava image, replace the code in the fragment shader with the following. Notice that the finer level of detail is created by using a larger scale value in the fractalRandom function.

```
// lava
float r = fractalRandom(UV, 40);
vec4 color1 = vec4(1, 0.8, 0, 1);
vec4 color2 = vec4(0.8, 0, 0, 1);
fragColor = mix( color1, color2, r );
```

To create the marble image, replace the code in the fragment shader with the following. As you will see, this code differs by the scaling of the random value and the application of the **abs** and **sin** functions, which creates a greater contrast between the differently colored regions of the image.

```
// marble
float t = fractalRandom(UV, 4);
float r = abs(sin(20 * t));
vec4 color1 = vec4(0.0, 0.2, 0.0, 1.0);
vec4 color2 = vec4(1.0, 1.0, 1.0, 1.0);
fragColor = mix( color1, color2, r );
```

Finally, to create the wood image, replace the code in the fragment shader with the following. In this code, the addition of the **UV.y** component in the calculation produces horizontal lines that are then randomly distorted. In addition, the modifications in the calculation of the **r** variable result in lines that are more sharply defined in the image.

```
// wood grain
float t = 80 * UV.y + 20 * fractalRandom(UV, 2);
```

```
float r = clamp( 2 * abs(sin(t)), 0, 1 );
vec4 color1 = vec4(0.3, 0.2, 0.0, 1.0);
vec4 color2 = vec4(0.6, 0.4, 0.2, 1.0);
fragColor = mix( color1, color2, r );
```

5.7 USING TEXT IN SCENES

In this section, you will explore a number of different ways that images of text can be added to three-dimensional scenes. Many of these approaches will naturally lead to additions to the graphics framework, such as a sprite material and an orthogonal camera that may be useful in other contexts as well.

5.7.1 Rendering Text Images

The first goal is to render text to a surface, which the Pygame library uses to store pixel data. Fortunately, this is a straightforward process, thanks to the built-in functionality present in Pygame. In this section, you will create a new class called **`TextTexture`**, which extends the **`Texture`** class and renders an image of text, whose appearance will have the following customizations available:

- the text to be rendered
- the font to be used: either the name of a system font that is already installed on the computer, or the file name of a font file to be loaded
- the size of the font
- the color of the font and the color of the background; due to Pygame conventions, the values of the red, green, and blue components of the colors must be specified by integers in the range from 0 to 255
- whether or not to use a transparent background (if **`True`**, overrides the background color)
- width and height of the rendered image; if not set, the image size will exactly fit the text
- horizontal and vertical alignment: if the size of the image is larger than the size of the rendered text, the values of these parameters can be used to align the text horizontally (left, center, right) or vertically (top, middle, bottom)
- the width and color of a border around the image; if border width is not specified or set to zero, no border will be rendered

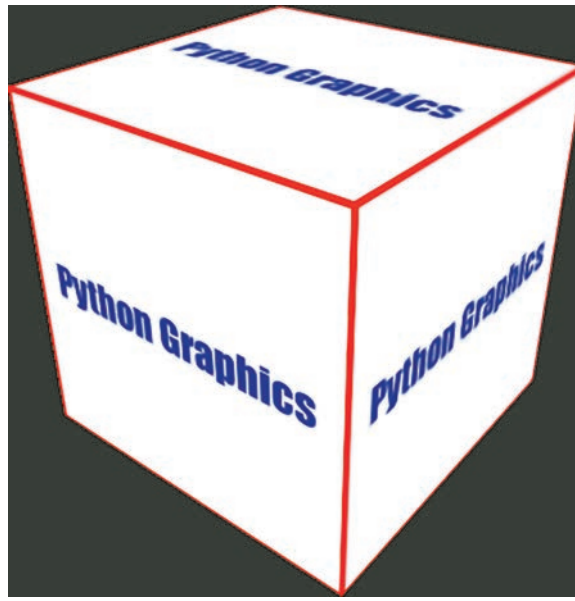

FIGURE 5.25 Text rendered to image and applied to a cube.

With these settings, you can generate images such as the one seen in Figure 5.25, which has been applied to a cube. The image shown is created from the Impact font, size 32. The text is rendered in blue on a white background with a red border. The image size is 256 by 256 pixels, and the text is aligned to the center of the image. This particular image will be generated in an example that follows.

The first step will be to create the **TextTexture** class. In the **extras** folder, create a new file named **textTexture.py** containing the following code, which is explained by the comments present throughout.

```
from core.texture import Texture
import pygame
class TextTexture(Texture):
    def __init__( self, text="Hello, world!",
                  systemFontName="Arial",
                    fontFileName=None,
                  fontSize=24,
                  fontColor=[0,0,0],
                      backgroundColor=[255,255,255],
                  transparent=False,
                  imageWidth=None, imageHeight=None,
```

```
            alignHorizontal=0.0,
               alignVertical=0.0,
            imageBorderWidth=0,
               imageBorderColor=[0,0,0]):
super().__init__()
# default font
font = pygame.font.SysFont(systemFontName,
   fontSize)
# can override by loading font file
if fontFileName is not None:
    font = pygame.font.Font(fontFileName,
       fontSize)
# render text to (antialiased) surface
fontSurface = font.render(text, True,
   fontColor)
# determine size of rendered text for
    alignment purposes
(textWidth, textHeight) = font.size(text)
# if image dimensions are not specified,
#   use font surface size as default
if imageWidth is None:
    imageWidth = textWidth
if imageHeight is None:
    imageHeight = textHeight

# create surface to store image of text
#   (with transparency channel by default)
self.surface = pygame.Surface( (imageWidth,
   imageHeight),
                                   pygame.
                                     SRCALPHA )

# background color used when not transparent
if not transparent:
    self.surface.fill( backgroundColor )
# alignHorizontal, alignVertical are
    percentages,
#   measured from top-left corner
cornerPoint = ( alignHorizontal *
   (imageWidth-textWidth),
                alignVertical * (imageHeight-
                  textHeight) )
destinationRectangle = fontSurface.get_rect(
```

```
                                topleft=cornerPoint )
        # optional: add border
        if imageBorderWidth > 0:
            pygame.draw.rect( self.surface,
                imageBorderColor,
                [0,0, imageWidth,imageHeight],
                    imageBorderWidth )

        # apply fontSurface to correct position on
            final surface
        self.surface.blit( fontSurface,
             destinationRectangle )
        self.uploadData()
```

To demonstrate the use of this class in an application in the graphics framework, in the main folder, make a copy of the file **test-template.py** and name it **test-5-6.py**. Add the following import statements:

```
from geometry.rectangleGeometry import
RectangleGeometry
from extras.textTexture import TextTexture
```

Then, in the **initialize** function, replace the code in that function, starting from the line where the camera position is set, with the following code:

```
self.camera.setPosition( [0, 0, 1.5] )
geometry = RectangleGeometry()
message = TextTexture(text="Python Graphics",
                      systemFontName="Impact",
                         fontSize=32,
                      fontColor=[0,0,200],
                      imageWidth=256,
                         imageHeight=256,
                      alignHorizontal=0.5,
                         alignVertical=0.5,
                      imageBorderWidth=4,
                      imageBorderColor=[255,0,0])
material = TextureMaterial(message)
self.mesh = Mesh( geometry, material )
self.scene.add( self.mesh )
```

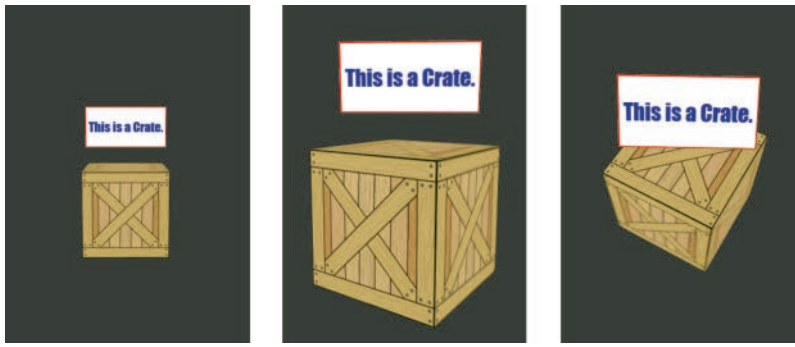

FIGURE 5.26 Billboarding a label above a crate.

When you run this code, you should see a rectangle object displaying the texture seen on the sides of the cube in Figure 5.25.

5.7.2 Billboarding

One major use of text is to convey information to the viewer, as in a label. One way to incorporate a label in a scene is to apply it to an in-scene object, such as a sign or a marquee. The approach has the drawback that the viewer might not be able to read the label from all angles and inadvertently miss some useful information. One remedy for this issue is *billboarding*: orienting an object so that it always faces the camera, as illustrated in Figure 5.26. Two methods for implementing billboarding are using a special matrix transformation and using a custom material, each of which are discussed in detail and implemented in what follows.

5.7.2.1 Look-At Matrix

One method for accomplishing billboarding is using a transformation called a *look-at matrix*, which, as the name implies, orients one object towards another object called the *target*. The *look direction* or *forward direction* of a 3D object is its local negative z-axis, just as is the case when working with camera objects. Similarly, in this context, the positive local x axis and y axis are sometimes referred to as the *right direction* and the *up direction* of the object, respectively. The look direction is aligned with the vector from the position of the object to the position of the target. To avoid ambiguity, an up direction must be specified when calculating this matrix, as there are many ways to look along the look direction: the object could be tilted (rotated) to the left or the right by any amount with respect to this direction.

To determine the rotation components of a look-at matrix (the upper 3-by-3 submatrix), you can apply the same mathematical approach used in Chapter 3 to derive the matrices representing rotation around each of the coordinate axes: determine the results of applying the look-at transformation to the standard basis vectors $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$. If F represents this transformation, then in line with the vocabulary introduced above, $F(\boldsymbol{i})$ will equal the right direction, $F(\boldsymbol{j})$ will equal the up direction, and $-F(\boldsymbol{k})$ will equal the forward direction. These results will be the columns for the rotational component of the transformation matrix. The fourth column of the transformation matrix corresponds to the position of the object, which should not be changed in this matrix.

Before proceeding with the necessary calculations, a new vector operation must be introduced: the cross product. The *cross product* $\boldsymbol{v} \times \boldsymbol{w}$ of two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ in three-dimensional space produces a third vector $\boldsymbol{u}$ which is perpendicular to both $\boldsymbol{v}$ and $\boldsymbol{w}$. The orientation of $\boldsymbol{u}$ is given by the right-hand convention discussed in Chapter 3. For example, if $\boldsymbol{v}$ and $\boldsymbol{w}$ align with the x and y axes, respectively, then $\boldsymbol{u}$ will align with the z axis. The cross product operation is not commutative: switching the order in the product will reverse the orientation of the result; symbolically, $\boldsymbol{v} \times \boldsymbol{w} = -(\boldsymbol{w} \times \boldsymbol{v})$. If $\boldsymbol{v}$ and $\boldsymbol{w}$ are parallel—aligned along the same direction—then the calculation of a perpendicular vector is ambiguous and the cross product returns the zero vector. The cross product will be used in calculating the right, up, and forward vectors, as they all must be perpendicular to each other. The actual formula for the cross product will not be discussed here, as it can be easily looked up in any vector algebra text, and you will use the Python package numpy to perform this calculation.

Since the forward vector points from the object position to the target position, this vector can be calculated by subtracting these positions. The object's up vector should lie in the plane spanned by the world up vector $\langle 0, 1, 0 \rangle$ and the object forward vector, and therefore, the right vector will be perpendicular to these vectors. Thus, the right vector can be calculated from the cross product of the world up vector and the forward vector. (If the world up vector and the forward vector are pointing in the same direction, then the world up vector can be perturbed by a small offset to avoid a zero vector as the result.) Finally, the object's up vector can be calculated from the cross product of the right vector and the forward vector. Finally, all these vectors should have length 1, so each vector can be scaled by dividing it by its length (calculated with the `norm` function in numpy). The natural place to implement these calculations is in the `Matrix` class. In the file `matrix.py` in the `core` folder, add the following import statements:

```
from numpy import subtract, divide, cross
from numpy.linalg import norm
```

Then, add the following static function, which will be used to generate the look-at matrix:

```
@staticmethod
def makeLookAt(position, target):
    worldUp = [0, 1, 0]
    forward = subtract( target, position )
    right = cross( forward, worldUp )

    # if forward and worldUp vectors are parallel,
    # right vector is zero;
    #   fix by perturbing worldUp vector a bit
    if norm(right) < 0.001:
        offset = numpy.array( [0.001, 0, 0] )
        right = cross( forward, worldUp + offset )

    up = cross( right, forward )

    # all vectors should have length 1
    forward = divide( forward, norm(forward) )
    right = divide( right, norm(right) )
    up = divide( up, norm(up) )

    return numpy.array( [[right[0], up[0],
        -forward[0], position[0]],
                         [right[1], up[1],
                            -forward[1], position[1]],
                         [right[2], up[2],
                            -forward[2], position[2]],
                         [       0,     0,
                          0,           1]]
                         )
```

Finally, in the file **object3D.py** in the **core** folder, add the following function, which will be used to apply the look-at matrix to an object, retaining the object's position while replacing its orientation so that the object faces the target.

```
def lookAt(self, targetPosition):
```

```
        self.transform = Matrix.makeLookAt( self.
            getWorldPosition(),

          targetPosition )
```

With these additions to the graphics framework, you can now recreate the scene illustrated in Figure 5.26. Since the side of the rectangle that faces forward (the local negative *z* direction) is the side on which the image is rendered backwards, a 180° rotation is applied to the geometry before creating the mesh so that the image will appear correctly when the look-at matrix is applied to the mesh. The scene will include a movement rig so that you can see the orientation of the label change in response to the movement of the camera. In the main project folder, create a copy of the file **test-template.py** and name it **test-5-7.py**. In this new file, add the following import statements:

```
from core.matrix import Matrix
from extras.textTexture import TextTexture
from extras.movementRig import MovementRig
from geometry.rectangleGeometry import
RectangleGeometry
from geometry.boxGeometry       import BoxGeometry
```

Then, in the **initialize** function, replace the code in that function, starting from the line where the camera position is set, with the following code:

```
self.rig = MovementRig()
self.rig.add( self.camera )
self.rig.setPosition( [0, 1, 5] )
self.scene.add( self.rig )

labelTexture = TextTexture(text=" This is a Crate. ",
                    systemFontName="Arial Bold",
                       fontSize=40,
                    fontColor=[0,0,200],
                    imageWidth=256, imageHeight=128,
                    alignHorizontal=0.5,
                       alignVertical=0.5,
                    imageBorderWidth=4,
imageBorderColor=[255,0,0])
```

```
labelMaterial = TextureMaterial(labelTexture)
labelGeometry = RectangleGeometry(width=1,
   height=0.5)
labelGeometry.applyMatrix( Matrix.makeRotationY(3.14) )
self.label = Mesh(labelGeometry, labelMaterial)
self.label.setPosition( [0, 1, 0] )
self.scene.add(self.label)


crateGeometry = BoxGeometry()
crateTexture = Texture("images/crate.png")
crateMaterial = TextureMaterial(crateTexture)
crate = Mesh( crateGeometry, crateMaterial )
self.scene.add( crate )
```

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

```
self.rig.update( self.input, self.deltaTime )
self.label.lookAt( self.camera.getWorldPosition() )
```

With these additions, this new example is complete. When you run the application, you should be able to recreate the images from the perspectives illustrated in Figure 5.26.

5.7.2.2 Sprite Material

Another approach to billboarding is by using a customized vertex shader that discards any rotation information from the model and view matrices (only retaining position related data) before applying the projection matrix. In computer graphics, a two-dimensional image used in a three-dimensional scene in this way is often referred to as a *sprite*, and accordingly, the material you will create to implement this shader will be called **SpriteMaterial**.

Such a material is also a natural place to build in support for tilesets. A *tileset* (sometimes also called a *spritesheet*) is a grid of rectangular images (called *tiles*) combined into a single image for convenience and efficiency. When rendering the scene, the UV coordinates of the object can be transformed so the desired tile is displayed from the tileset image. One situation where tilesets are useful is *texture packing*: combining multiple images into a single image for the purpose of reducing the number of textures required by an application. This may be particularly useful in an application where the text or image displayed on a particular surface will change

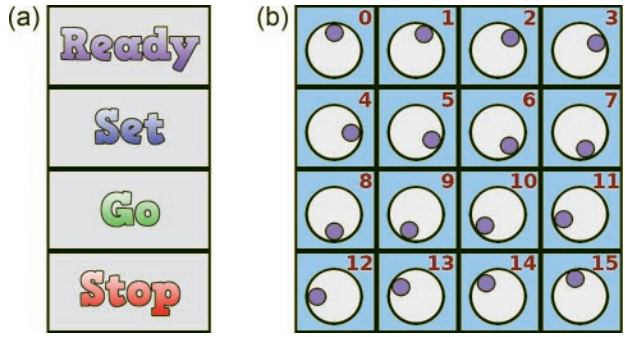

FIGURE 5.27 Tilesets used for texture packing (a) and spritesheet animation (b).

periodically. The left side of Figure 5.27 shows an example of such an image. Another application is *spritesheet animation*: displaying a sequence of images in rapid succession to create the illusion of motion or continuous change. The right side of Figure 5.27 contains a 4-by-4 tileset; when the images are displayed in the order indicated, a small circle will appear to be rolling around within a larger circle.

The vertex shader in the **`SpriteMaterial`** class will contain the same uniform matrices and attribute variables as previous materials. One new addition will be a boolean variable named **`billboard`**; if set to true, it will enable a billboard effect as previously described by replacing the rotation component of the combined model and view matrices with the identity matrix. In addition, to support tilesets, there will be a variable named **`tileNumber`**. If **`tileNumber`** is set to a number greater than –1, then the vertex shader will use the information stored in **`tileCount`** (the number of columns and rows in the tileset) to transform the UV coordinates so that the desired tile is rendered on the object.

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

```
from material.material import Material
from OpenGL.GL import *

class SpriteMaterial(Material):

    def __init__(self, texture, properties={}):

        vertexShaderCode = """
        uniform mat4 projectionMatrix;
```

```
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;
uniform bool billboard;
uniform float tileNumber;
uniform vec2 tileCount;
in vec3 vertexPosition;
in vec2 vertexUV;
out vec2 UV;

void main()
{
    mat4 mvMatrix = viewMatrix * modelMatrix;
    if ( billboard )
    {
        mvMatrix[0][0] = 1;
        mvMatrix[0][1] = 0;
        mvMatrix[0][2] = 0;
        mvMatrix[1][0] = 0;
        mvMatrix[1][1] = 1;
        mvMatrix[1][2] = 0;
        mvMatrix[2][0] = 0;
        mvMatrix[2][1] = 0;
        mvMatrix[2][2] = 1;
    }
   gl_Position = projectionMatrix * mvMatrix *
                 vec4(vertexPosition, 1.0);
   UV = vertexUV;
   if (tileNumber > -1.0)
   {
       vec2 tileSize = 1.0 / tileCount;
       float columnIndex = mod(tileNumber,
           tileCount[0]);
       float rowIndex = floor(tileNumber /
           tileCount[0]);
       vec2 tileOffset = vec2( columnIndex/
          tileCount[0],
        1.0 - (rowIndex + 1.0)/tileCount[1] );
       UV = UV * tileSize + tileOffset;
   }
}
"""

fragmentShaderCode = """
```

```
        uniform vec3 baseColor;
        uniform sampler2D texture;
        in vec2 UV;
        out vec4 fragColor;

        void main()
        {
            vec4 color = vec4(baseColor, 1) *
               texture2D(texture, UV);
            if (color.a < 0.1)
                discard;

            fragColor = color;
        }
        """

        super().__init__(vertexShaderCode,
            fragmentShaderCode)
        self.addUniform("vec3", "baseColor", [1.0,
            1.0, 1.0])
        self.addUniform("sampler2D", "texture",
                    [texture.textureRef, 1])
        self.addUniform("bool",  "billboard", False)
        self.addUniform("float", "tileNumber", -1)
        self.addUniform("vec2",  "tileCount",  [1, 1])
        self.locateUniforms()

        # render both sides?
        self.settings["doubleSide"] = True
        self.setProperties(properties)

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

Next, you will make an application to test this material, using both the billboarding and spritesheet animation features. You will also include a movement rig to move the camera around the scene, and a grid for a fixed plane of reference as you move around the scene. To begin, create a copy of the file **`test-template.py`** named **`test-5-8.py`** and add the following import files.

```
from geometry.rectangleGeometry import
RectangleGeometry
from material.spriteMaterial import SpriteMaterial
from extras.movementRig import MovementRig
from extras.gridHelper import GridHelper
from math import floor
```

Then, in the **initialize** function, replace the code in that function, starting from the line where the camera position is set, with the following code:

```
self.rig = MovementRig()
self.rig.add( self.camera )
self.rig.setPosition( [0, 0.5, 3] )
self.scene.add( self.rig )
geometry = RectangleGeometry()
tileSet = Texture("images/rolling-ball.png")
spriteMaterial = SpriteMaterial(tileSet, {
    "billboard"  : 1,
    "tileCount"  : [4,4],
    "tileNumber" : 0
})
self.tilesPerSecond = 8

self.sprite = Mesh( geometry, spriteMaterial )
self.scene.add( self.sprite )

grid = GridHelper()
grid.rotateX(-3.14/2)
self.scene.add( grid )
```

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

```
tileNumber = floor(self.time * self.tilesPerSecond)
self.sprite.material.uniforms["tileNumber"].data =
   tileNumber
self.rig.update( self.input, self.deltaTime )
```

When you run this application, using the spritesheet on the right side of Figure 5.27, you should see the animation of the small circle rolling around the large circle, in a scene similar to that in Figure 5.28. Furthermore, as you move around the scene, the rectangle should always be facing the camera.

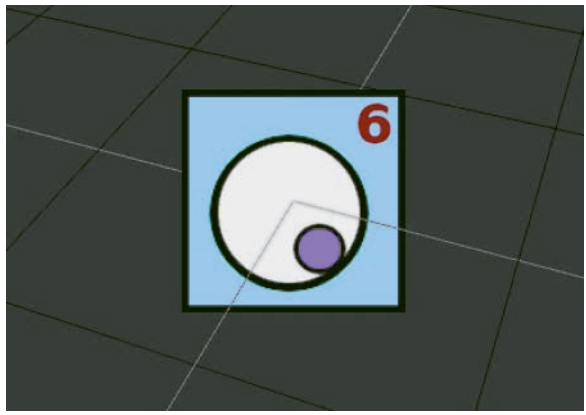

FIGURE 5.28 A frame from a spritesheet animation in a scene.

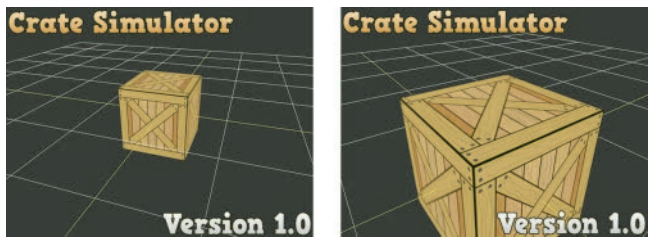

FIGURE 5.29 A heads-up display containing fixed text.

5.7.3 Heads-Up Displays and Orthogonal Cameras

The methods for viewing text implemented in the previous section are useful when the text only needs to be visible in a particular area in the scene. For text or graphics that should be visible to the viewer at all times, the most common approach is to use a *heads-up display* (HUD): a transparent layer containing these elements, rendered after the scene, and therefore appearing on top, as illustrated in Figure 5.29. Implementing this functionality will involve additions or modifications to multiple classes: **`Matrix`**, **`Camera`**, **`Renderer`**, and **`Rectangle`**.

The objects to be included in the heads-up display will be added to a second scene, sometimes called the *HUD layer*. In contrast to the primary three-dimensional scene, where the scale is somewhat arbitrary and a perspective projection is used, the natural unit of measurement in the HUD layer is pixels and an orthographic projection is used. The viewable region of space in the HUD layer will be a rectangular box, whose width and

height will be equal to the dimensions of the screen, and whose depth is arbitrary. In general, z-coordinates in the HUD layer are only important if one object should be rendered above another. When used to determine stack order in this way, the z coordinate of an object is also called its *z-index*.

The viewable region can be specified by six constants: *left*, *right*, *bottom*, *top*, *near*, and *far*, denoted by the variables l, r, b, t, n, and f, respectively; the points (x, y, z) in the region satisfy the conditions $l \leq x \leq r$, $b \leq y \leq t$, and $n \leq -z \leq f$ (the negation of z due to the reversal of the z direction in clip space, discussed in Chapter 3). This region in turn must be projected into the cubical volume rendered by OpenGL, the set of points (x, y, z) where all components are bounded by −1 and +1. The necessary transformations in each coordinate are each affine transformations, involving both scaling and translation. For example, the transformation of the x coordinate is a function of the form $F(x) = p \cdot x + q$, and given that $F(l) = -1$ and $F(r) = +1$, it is possible to algebraically solve for the values of p and q in terms of l and r. In particular, $p = 2/(r - l)$ and $q = -(r + l)/(r - l)$. Similar calculations can be used to derive the transformation needed for the y coordinate and the z coordinate. Since homogeneous coordinates are used throughout this graphics framework, these transformations can be combined into the following matrix:

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The next step is to add a function to the **`Matrix`** class that generates this matrix. To implement this, in the **`matrix.py`** file in the **`core`** folder, add the following function:

```
@staticmethod
def makeOrthographic(left=-1, right=1, bottom=-1,
    top=1,
                         near=-1, far=1):
```

```
    return numpy.array([[2/(right-left), 0, 0,
                          -(right+left)/(right-left)],
                        [0, 2/(top-bottom), 0,
                          -(top+bottom)/(top-bottom)],
                        [0, 0, -2/(far-near),
                          -(far+near)/(far-near)],
                        [0, 0, 0, 1]])
```

Next, the **Camera** class needs to be updated so that an orthographic matrix can be used as the projection matrix. This will be accomplished by adding a **setOrthographic** function containing the necessary parameters. For symmetry, a **setPerspective** function will be added as well. To implement these, in the **camera.py** file in the **core** folder, add the following functions:

```
def setPerspective(self, angleOfView=50,
                         aspectRatio=1, near=0.1,
                            far=1000):
    self.projectionMatrix = Matrix.
       makePerspective(angleOfView,
                          aspectRatio, near, far)

def setOrthographic(self, left=-1, right=1,
    bottom=-1, top=1,
                        near=-1, far=1):
    self.projectionMatrix = Matrix.
       makeOrthographic(left, right,
                             bottom, top, near, far)
```

At present, whenever a scene is rendered by the **Renderer** class, the color and depth buffers are cleared. In order for the HUD layer to be rendered on top of the main three-dimensional scene and have the main scene still be visible, the color buffer needs to *not* be cleared between these two renders. However, the depth buffer should still be cleared, because otherwise fragments from the HUD layer might not be rendered due to residual depth values from rendering the main scene. To this end, in the **renderer.py** file in the **core** folder, change the declaration of the **render** function to the following:

```
def render(self, scene, camera, clearColor=True,
clearDepth=True):
```

Then, replace the line of code containing the **glClear** function with the following block of code:

```
 # clear color and depth buffers
if clearColor:
    glClear(GL_COLOR_BUFFER_BIT)
if clearDepth:
    glClear(GL_DEPTH_BUFFER_BIT)
```

Finally, to simplify aligning these objects within the HUD layer, a few changes will be made to the **Rectangle** class, enabling any point to be assigned to any location within the rectangle. As the **Rectangle** class was originally written, the origin (0, 0) corresponds to the center of the rectangle. You will add two parameters, **position** and **alignment** (each a list of two numbers) which will affect the way the vertex position data is generated. The **alignment** components should be thought of as percentages (values between 0.00 and 1.00), corresponding to offsets along the width and height, used to indicate which point in the rectangle will correspond to **position**. For example, if **alignment[0]** has a value of 0.00, 0.50, or 1.00, then the rectangle will be left-aligned, centered, or right-aligned, respectively, with respect to the *x* component of **position**. If **alignment[1]** has a value of 0.00, 0.50, or 1.00, then the rectangle will be bottom-aligned, centered, or top-aligned, respectively, with respect to the *y* component of position.

To implement the changes, in the file **rectangleGeometry.py** in the **geometry** folder, change the declaration of the **__init__**function to the following:

```
def __init__(self, width=1, height=1, position=[0, 0],
alignment=[0.5, 0.5]):
```

Then, change the block of code that assigns values to **P0**, **P1**, **P2**, and **P3**, to the following:

```
x, y = position
a, b = alignment
P0 = [ x +  (-a)*width, y +  (-b)*height, 0 ]
P1 = [ x + (1-a)*width, y +  (-b)*height, 0 ]
P2 = [ x +  (-a)*width, y + (1-b)*height, 0 ]
P3 = [ x + (1-a)*width, y + (1-b)*height, 0 ]
```

This completes all the changes necessary to be able to render the scene shown in Figure 5.24. To create the application, in the main folder, make a copy of the file **test-template.py** and name it **test-5-9.py**. Add the following import statements:

```
from geometry.rectangleGeometry import
RectangleGeometry
from geometry.boxGeometry import BoxGeometry
from material.textureMaterial import TextureMaterial
from extras.movementRig import MovementRig
from extras.gridHelper import GridHelper
```

Then, in the **initialize** function, replace the code in that function, starting from the line where the camera object is created, with the following code, which will set up the main scene.

```
self.camera = Camera( aspectRatio=800/600 )
self.rig = MovementRig()
self.rig.add( self.camera )
self.rig.setPosition( [0, 0.5, 3] )
self.scene.add( self.rig )

crateGeometry = BoxGeometry()
crateMaterial = TextureMaterial( Texture("images/
   crate.png") )
crate = Mesh( crateGeometry, crateMaterial )
self.scene.add( crate )

grid = GridHelper( gridColor=[1,1,1],
centerColor=[1,1,0] )
grid.rotateX( -3.14/2 )
self.scene.add( grid )
```

The next step is to set up the HUD layer. When creating a rectangle to display an image in the HUD, the width and height of the rectangle should be set to the width and height of the image (although these values may also be scaled proportionally if desired). The HUD layer illustrated in Figure 5.29 contains two labels; the code that follows will create these two objects. The **position** and **alignment** parameter values for the first label are chosen to align the top-left corner of the rectangle with the point in the top-left of the screen, while the parameters for the second label align

its bottom-right corner with the bottom-right of the screen. To continue, add the following code at the end of the **initialize** function.

```
self.hudScene = Scene()
self.hudCamera = Camera()
self.hudCamera.setOrthographic(0,800, 0,600, 1,-1)
labelGeo1 = RectangleGeometry(width=600, height=80,
                              position=[0,600],
                                 alignment=[0,1])
labelMat1 = TextureMaterial( Texture("images/crate-
   sim.png"))
label1 = Mesh(labelGeo1, labelMat1)
self.hudScene.add( label1 )
labelGeo2 = RectangleGeometry(width=400, height=80,
                              position=[800,0],
                                alignment=[1,0])
labelMat2 = TextureMaterial( Texture("images/
version-1.png"))
label2 = Mesh(labelGeo2, labelMat2)
self.hudScene.add( label2 )
```

Next, replace the code in the **update** function with the following code, which renders the scenes in the correct order and prevents the color buffer from being cleared as needed so that the content from each scene is visible.

```
self.rig.update( self.input, self.deltaTime )
self.renderer.render( self.scene, self.camera )
self.renderer.render( self.hudScene, self.hudCamera,
   clearColor=False)
```

Finally, change the end of the program to the following, which increases the size of the graphics window so that it is better able to fit the HUD layer content.

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

At this point, when you run the application, you will be able to move the camera around the scene using the standard movement rig controls and produce images such as those illustrated in Figure 5.29.

5.8 RENDERING SCENES TO TEXTURES

The next goal is to render scenes to textures, which can be used for a variety of purposes. Figure 5.30 illustrates a scene based on the skysphere example shown in Figure 5.14, which also includes a sphere with a grid texture applied, and a dark box with a rectangle positioned in front, representing a television screen. The texture used for the rectangle is a rendering of the scene from a second camera positioned above these objects and oriented downward. As you will see when you implement this scene, the sphere is also rotating, and the image in the rectangle constantly updates and shows this.

Framebuffers, introduced in Chapter 1, are of key importance when rendering to a texture. When OpenGL is initialized, a framebuffer is automatically generated, containing a color buffer and a depth buffer, and is attached to the window that displays graphics. When a scene is rendered, the image seen in the window corresponds to the data in the color buffer. To render an image to a texture, you will need to manually set up a framebuffer and configure the attached buffers (color and depth). These tasks will require the use of many OpenGL functions, which are discussed in what follows.

The first steps in working with framebuffers are to generate a reference and bind it to a target, analogous to the first steps when working with vertex buffers or textures. In the case of framebuffers, this is accomplished with the following two OpenGL functions:

glGenFramebuffers(*bufferCount*)

Returns a set of nonzero integers representing available framebuffer references. The number of references returned is specified by the integer parameter *bufferCount*.

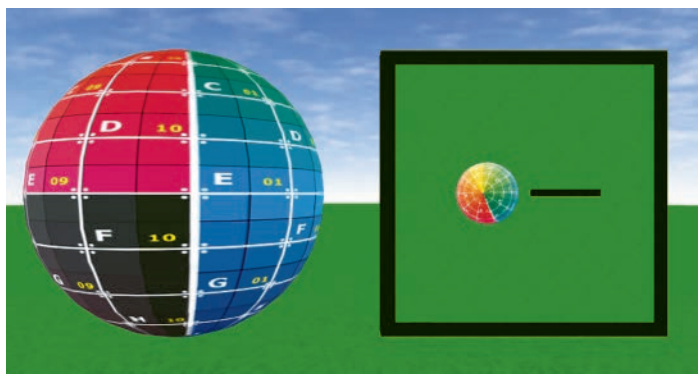

FIGURE 5.30 Rendering a scene to a texture within the scene.

glBindFramebuffer(*bindTarget, framebufferRef)*

The framebuffer referred to by the parameter *framebufferRef* is bound to the target specified by the parameter *bindTarget*, whose value is an OpenGL constant such as GL_FRAMEBUFFER or GL_DRAW_FRAMEBUFFER. Future OpenGL operations affecting the same *bindTarget* will be applied to the referenced framebuffer.

Next, the buffers used by the framebuffer need to be configured. To use a texture for a buffer (as you will for the color buffer), you use the following function:

glFramebufferTexture(*bindTarget, attachment, textureRef, level*)

Attaches a texture object specified by *textureRef* as the type of buffer specified by *attachment* to the framebuffer currently bound to *bindTarget*. The parameter *attachment* is an OpenGL constant such as GL_COLOR_ATTACHMENT*n* (for an integer *n*), GL_DEPTH_ATTACHMENT, or GL_STENCIL_ATTACHMENT. The parameter *level* is usually 0, indicating this is the base image level in the associated mipmap image.

Similar to textures, *renderbuffers* are OpenGL objects that store image data, but specifically used with and optimized for framebuffers. As usual, the first steps in working with these objects are to generate a reference and bind it to a target, which uses the following OpenGL functions:

glGenRenderbuffers(*bufferCount*)

Returns a set of nonzero integers representing available renderbuffer references. The number of references returned is specified by the integer parameter *bufferCount*.

glBindRenderbuffer(*bindTarget, renderbufferRef*)

The renderbuffer referred to by the parameter *renderbufferRef* is bound to the target specified by the parameter *bindTarget*, whose value must be the OpenGL constant GL_RENDERBUFFER. Future OpenGL operations affecting the same *bindTarget* will be applied to the referenced renderbuffer.

Once a render buffer is bound, storage is allocated with the following function:

glRenderbufferStorage(*bindTarget, format, width, height*)

Allocate storage for the renderbuffer currently bound to the target *bindTarget*, whose value must be the OpenGL constant GL_RENDERBUFFER. The data stored in the renderbuffer will have the type specified by the parameter *format*, whose value is an OpenGL constant such as GL_RGB, GL_RGBA, GL_DEPTH_COMPONENT, or GL_STENCIL. The dimensions of the buffer are specified by *width* and *height*.

The next OpenGL function is analogous in purpose to glFramebufferTexture, but is used instead when a renderbuffer will be used to store data instead of a texture.

glFramebufferRenderbuffer(*framebufferTarget, attachment, renderbufferTarget, renderbufferRef*)

Attaches a renderbuffer object specified by *renderbufferRef* as the type of buffer specified by *attachment* to the framebuffer currently bound to *framebufferTarget*. The parameter *attachment* is an OpenGL constant such as GL_COLOR_ATTACHMENT*n* (for an integer *n*), GL_DEPTH_ATTACHMENT, or GL_STENCIL_ATTACHMENT. The parameter *renderbufferTarget* must be the OpenGL constant GL_RENDERBUFFER.

Finally, to verify that the framebuffer has been configured correctly, you can use the following function:

glCheckFramebufferStatus(*bindTarget*)

Check if the framebuffer currently bound to *bindTarget* is complete: at least one color attachment has been added and all attachments have been correctly initialized.

With a knowledge of these OpenGL functions, you are now prepared to create the **`RenderTarget`** class. In the **`core`** folder, create a new file named **`renderTarget.py`** containing the following code. Note that if a texture is not supplied as a parameter, an empty texture is automatically generated.

```
from OpenGL.GL import *
import pygame
```

```
from core.texture import Texture

class RenderTarget(object):

    def __init__(self, resolution=[512, 512],
        texture=None,
                 properties={}):

        # values should equal texture dimensions
        self.width, self.height = resolution

        if texture is not None:
            self.texture = texture
        else:
            self.texture = Texture( None, {
                "magFilter" : GL_LINEAR,
                "minFilter" : GL_LINEAR,
                "wrap"      : GL_CLAMP_TO_EDGE
            })
            self.texture.setProperties( properties )
            self.texture.surface = pygame.Surface(
               resolution )
            self.texture.uploadData()

        # create a framebuffer
        self.framebufferRef = glGenFramebuffers(1)
        glBindFramebuffer(GL_FRAMEBUFFER, self.
           framebufferRef)
        # configure color buffer to use this texture
        glFramebufferTexture(GL_FRAMEBUFFER,
           GL_COLOR_ATTACHMENT0,
        self.texture.textureRef, 0)
        # generate a buffer to store depth information
        depthBufferRef = glGenRenderbuffers(1)
        glBindRenderbuffer(GL_RENDERBUFFER,
           depthBufferRef)
        glRenderbufferStorage(GL_RENDERBUFFER,
           GL_DEPTH_COMPONENT,
        self.width, self.height)
        glFramebufferRenderbuffer(GL_FRAMEBUFFER,
           GL_DEPTH_ATTACHMENT,
        GL_RENDERBUFFER, depthBufferRef);
```

```
        # check framebuffer status
        if (glCheckFramebufferStatus(GL_FRAMEBUFFER)
                != GL_FRAMEBUFFER_COMPLETE):
            raise Exception("Framebuffer status error")
```

In order to use a manually created framebuffer as the target when rendering (rather than the framebuffer attached to the window), you must also make a few additions to the **Renderer** class. First, because the texture to which you will be rendering may not be the same size as the window, you need to be able to specify how clip space coordinates map to window coordinates. In particular, you need to specify the size and location of the *viewport*: the rectangular area in the window used to display the color data from the currently active framebuffer. This rectangular area is specified by the coordinates of the lower-left corner and its width and height, which is set by the following OpenGL function:

glViewport(*x, y, width, height*)

Specify the coordinates (*x, y*) of the lower-left corner and the *width* and *height* of the viewport rectangle. When a GL context is first created, (*x, y*) is initialized to (0, 0) and the *width* and *height* are initialized to the dimensions of the window.

When a **Renderer** object is initialized, you will store the size of the window for later use when setting the viewport. To begin, in the file **renderer.py** in the **core** folder, add the following import statement:

```
import pygame
```

Then, in the **__init__**function, add the following line of code:

```
self.windowSize = pygame.display.get_surface().
   get_size()
```

The **render** function should take an additional parameter in case a scene is to be rendered to a **RenderTarget** object, so change the parameter list as follows:

```
def render(self, scene, camera, clearColor=True,
    clearDepth=True,
                                    renderTarget=None):
```

Next, at the very beginning of the **render** function, add the following code, which binds the correct framebuffer (the value 0 indicating the framebuffer attached to the window is the render target), and sets the viewport size accordingly.

```
# activate render target
if (renderTarget == None):
    # set render target to window
    glBindFramebuffer(GL_FRAMEBUFFER, 0)
    glViewport(0,0, self.windowSize[0], self.
       windowSize[1])
else:
    # set render target properties
    glBindFramebuffer(GL_FRAMEBUFFER, renderTarget.
       framebufferRef)
    glViewport(0,0, renderTarget.width, renderTarget.
       height)
```

At this point, you are ready to begin creating this example. Start by making a copy of the file **test-5-2.py** (the skysphere example) and rename it as **test-5-11.py**. Add the following import statements:

```
from core.renderTarget import RenderTarget
from geometry.boxGeometry import BoxGeometry
from material.surfaceMaterial import SurfaceMaterial
```

To demonstrate that viewports work as expected, you will use an 800 by 600 size window and a 512 by 512 size texture. To change the size of the window from the default, change the last line of the code in the application to the following:

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

So that the scene does not appear stretched, you need to set the aspect ratio of the camera accordingly. Thus, change the line of code where the camera object is created to the following:

```
self.camera = Camera(aspectRatio=800/600)
```

Next, in the **initialize** function, add the following code to add new meshes to the scene.

```
sphereGeometry = SphereGeometry()
sphereMaterial = TextureMaterial( Texture("images/
   grid.png") )
self.sphere = Mesh( sphereGeometry, sphereMaterial )
self.sphere.setPosition( [-1.2, 1, 0] )
self.scene.add( self.sphere )


boxGeometry = BoxGeometry(width=2, height=2,
   depth=0.2)
boxMaterial = SurfaceMaterial({"baseColor":[0,0,0]} )
box = Mesh( boxGeometry, boxMaterial )
box.setPosition( [1.2, 1, 0] )
self.scene.add( box )
```

To create the "television screen"—the rectangular mesh whose material will use the texture from a render target—also add the following code to the **initialize** function:

```
self.renderTarget = RenderTarget( resolution=[512,
  512] )

screenGeometry = RectangleGeometry(width=1.8,
   height=1.8)
screenMaterial = TextureMaterial( self.renderTarget.
   texture )
screen = Mesh( screenGeometry, screenMaterial )
screen.setPosition( [1.2, 1, 0.11] )
self.scene.add( screen )
```

To create the second camera that will be used when rendering to the texture, also add the following code to the **initialize** function. Note that the aspect ratio of the sky camera is derived from the dimensions of the previously created render target and that the recently introduced look-at functionality is used to orient the sky camera in the desired direction.

```
self.skyCamera = Camera( aspectRatio=512/512 )
self.skyCamera.setPosition( [0, 10, 0.1] )
self.skyCamera.lookAt( [0,0,0] )
self.scene.add( self.skyCamera )
```

Finally, change the code in the **update** function to the following, which causes the sphere to spin, updates the movement rig, renders the scene to the render target (using the new sky camera), and then renders the scene to the window (using the original camera).

```
self.sphere.rotateY( 0.01337 )
self.rig.update( self.input, self.deltaTime )
self.renderer.render( self.scene, self.skyCamera,
                      renderTarget=self.renderTarget )
self.renderer.render( self.scene, self.camera )
```

With these additions, the example is complete. When you run the example, you should see a scene similar to that in Figure 5.30. Another application of rendering to a texture is to help orient a player moving a character around a large, complex environment by creating a "minimap": a texture similar to the one created in example in this section, but the sky camera moves in sync with the player's character and stays oriented towards the character at all times, and the result is rendered to a small rectangle in a HUD layer so that it stays fixed on screen and is visible to the player at all times.

A frequently used technique in computer graphics enabled by rendering to textures is postprocessing, which is described in detail and implemented in the next section.

5.9 POSTPROCESSING

In computer graphics, postprocessing is the application of additional visual effects to the image of a rendered scene. Figure 5.31 illustrates three such effects applied separately to a scene: *vignette* (reduced brightness towards the edges of an image), color inversion, and pixelation.

In addition, it is also desirable in many situations to chain these effects—apply them one after the other in sequence—to create a compound effect. For example, to replicate the appearance of older handheld video game

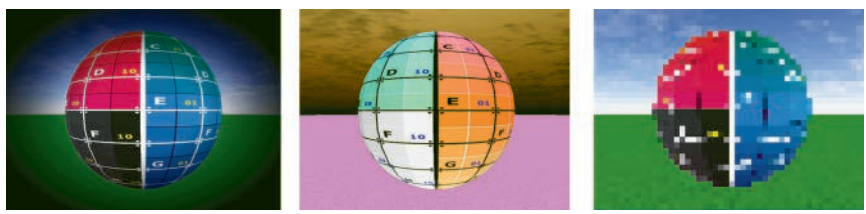

FIGURE 5.31 Postprocessing effects: vignette, color inversion, pixilation.

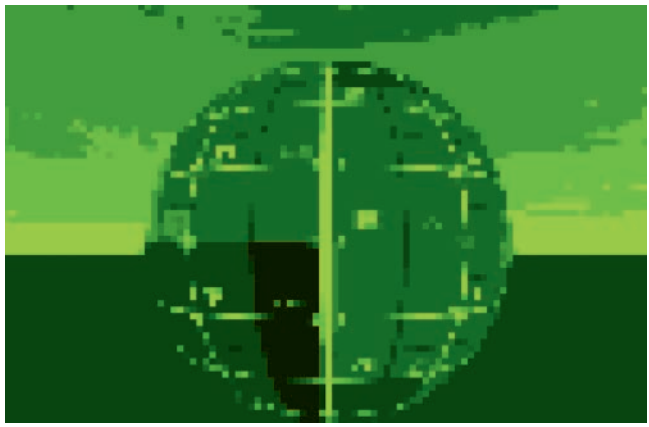

FIGURE 5.32 A compound postprocessing effect to simulate older video game system graphics.

system graphics, one could take the same base scene used in Figure 5.31 and apply a green tint effect followed by a pixelation effect, resulting in the image illustrated in Figure 5.32.

With the addition of the **`RenderTarget`** class from the previous section, incorporating postprocessing into the graphics framework is a straightforward task. Each effect will be implemented with a simple shader incorporated into a material. There will be a sequence of render passes, starting with the base scene, each of which is applied to a render target. The texture in each render target in the sequence will be used in a material implementing an effect for the next render pass in the sequence. The last render pass in the sequence will use a specified final render target; if none is specified, the window will be the target and the final result of the postprocessing effect sequence will be displayed.

To implement postprocessing functionality, a **`Postprocessor`** class will be created. This class will maintain lists of **`Scene`**, **`Camera`**, and **`RenderTarget`** objects. The original scene and camera will be the first elements in their respective lists. All subsequent scenes will consist of a single rectangle; its vertices will be aligned with clip space to eliminate the need for any matrix transformations in the vertex shader. In order to use the **`Renderer`** class, a camera must be supplied, and so a camera using the default orthographic projection (aligned to clip space) will be reused for the additional render passes. To begin, in the **`extras`** folder, create a new file name **`postprocessor.py`** with the following code:

```
from core.renderer import Renderer
from core.scene    import Scene
from core.camera   import Camera
from core.mesh     import Mesh
from core.renderTarget import RenderTarget
from geometry.geometry import Geometry


class Postprocessor(object):

    def __init__(self, renderer, scene, camera,
            finalRenderTarget=None):

        self.renderer = renderer

        self.sceneList  = [ scene ]
        self.cameraList = [ camera ]
        self.renderTargetList = [ finalRenderTarget ]
        self.finalRenderTarget = finalRenderTarget
        self.orthoCamera = Camera()
        self.orthoCamera.setOrthographic() # aligned
           with clip space
                                            # by default

        # generate a rectangle already aligned with
            clip space;
        #   no matrix transformations will be applied
        self.rectangleGeo = Geometry()
        P0, P1, P2, P3 = [-1,-1], [1,-1], [-1,1],
            [1,1]
        T0, T1, T2, T3 = [ 0, 0], [1, 0], [ 0,1],
            [1,1]
        positionData = [ P0,P1,P3, P0,P3,P2 ]
        uvData       = [ T0,T1,T3, T0,T3,T2 ]
        self.rectangleGeo.addAttribute("vec2",
                       "vertexPosition", positionData)
        self.rectangleGeo.addAttribute("vec2",
           "vertexUV", uvData)
        self.rectangleGeo.countVertices()

    def addEffect(self, effect):

        postScene = Scene()
```

```
        resolution = self.renderer.windowSize
        target = RenderTarget( resolution )

        # change the previous entry in the render
            target list
        #   to this newly created render target
        self.renderTargetList[-1] = target

        # the effect in this render pass will use
        #   the texture that was written to
        #   in the previous render pass
        effect.uniforms["texture"].data[0] = target.
           texture.textureRef

        mesh = Mesh( self.rectangleGeo, effect )
        postScene.add( mesh )

        self.sceneList.append( postScene )
        self.cameraList.append( self.orthoCamera )
        self.renderTargetList.append( self.
           finalRenderTarget )

    def render(self):
        passes = len(self.sceneList)
        for n in range( passes ):
            scene  = self.sceneList[n]
            camera = self.cameraList[n]
            target = self.renderTargetList[n]
            self.renderer.render( scene, camera,
               renderTarget=target )
```

As previously mentioned, the postprocessing effects (the parameter in the **addEffect** function) are materials containing simple shaders, designed to work with the geometric object created by the **Postprocessor** class. Next, you will create a template material that only incorporates the data needed for postprocessing effects. To begin, in your project directory, create a new folder called **effects**. Within that folder, create a new file called **templateEffect.py** that contains the following code:

```
from material.material import Material
class TemplateEffect(Material):

    def __init__(self):
```

```
vertexShaderCode = """
in vec2 vertexPosition;
in vec2 vertexUV;
out vec2 UV;
void main()
{
    gl_Position = vec4(vertexPosition, 0.0, 1.0);
     UV = vertexUV;
}
"""

fragmentShaderCode = """
in vec2 UV;
uniform sampler2D texture;
out vec4 fragColor;
void main()
{
     vec4 color = texture2D(texture, UV);
     fragColor = color;
}
"""

super().__init__(vertexShaderCode,
   fragmentShaderCode)
self.addUniform("sampler2D", "texture", [None,
   1])
self.locateUniforms()
```

Observe that, in the **`TemplateEffect`** class, the texture is rendered without any change; this is sometimes called a *pass-through* shader. The texture data stored in the **`Uniform`** object is set within the **`addEffect`** function in the **`Postprocessor`** class.

The next step is to create some basic effects. Each will involve some modifications to the fragment shader code in the template class created above. You will begin by creating one of the simplest postprocessing effects: color tinting, illustrated with a red tint applied in Figure 5.33.

This effect is accomplished by averaging the red, green, and blue components of each pixel, and then multiplying it by the tint color. You will add the tint color as a parameter in the class constructor, and create a corresponding uniform variable in the shader and uniform object in the class. To implement this, in the **`effects`** folder, make a copy of

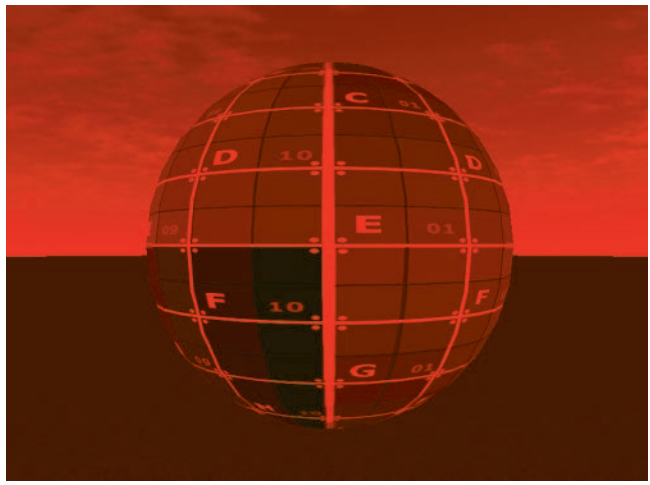

FIGURE 5.33 Color tint postprocessing effect.

the **templateEffect.py** file and name it **tintEffect.py**. In the new file, change the name of the class to **TintEffect**, and change the initialization function declaration to the following:

```
def __init__(self, tintColor=[1,0,0]):
```

Next, change the fragment shader code to the following:

```
in vec2 UV;
uniform vec3 tintColor;
uniform sampler2D texture;
out vec4 fragColor;

void main()
{
    vec4 color = texture2D(texture, UV);
    float gray = (color.r + color.g + color.b) / 3.0;
    fragColor = vec4(gray * tintColor, 1.0);
}
```

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

```
self.addUniform("vec3", "tintColor", tintColor)
```

Now that you have a basic effect to work with, you can create an application that uses it together with the **Postprocessor** class. In your main project folder, start by making a copy of the file **test-5-2.py** (the sky-sphere example) and rename it as **test-5-12.py**. Add the following import statements:

```
from extras.postprocessor import Postprocessor
from effect.tintEffect import TintEffect
```

Optionally, in the **initialize** function, you may add the following code to include a textured sphere, so that the resulting scene more closely resembles those in this section:

```
sphereGeometry = SphereGeometry()
sphereMaterial = TextureMaterial(Texture("images/grid.
   png"))
self.sphere = Mesh(sphereGeometry, sphereMaterial)
self.sphere.setPosition([0,1,0])
self.scene.add(self.sphere)
```

Next, in the **initialize** function, add the following code to set up postprocessing:

```
self.postprocessor = Postprocessor(self.renderer,
     self.scene, self.camera)
self.postprocessor.addEffect( TintEffect(
     tintColor=[1,0,0] ) )
```

Finally, replace the line of code in the **update** function referencing the **renderer** object to the following:

```
self.postprocessor.render()
```

When you run this example, you will see a scene similar to Figure 5.33. In the remainder of this section, you will create the effects illustrated earlier. After writing the code for each one, you can test it using the file **test-5-12.py** by adding the corresponding import statement and changing the effect that is added to the postprocessor.

To create the color inversion effect from Figure 5.31, make a copy of the **templateEffect.py** file and name it **invertEffect.py**. In the new file, change the name of the class to **InvertEffect**, and change the fragment shader code to the following:

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

void main()
{
    vec4 color = texture2D(texture, UV);
    vec4 invert = vec4(1 - color.r, 1 - color.g, 1 -
        color.b, 1);
    fragColor = invert;
}
```

To create the pixelation effect from Figure 5.31, when determining the color of each fragment, you will round the UV coordinates to a level of precision determined by the dimensions of the texture (indicated by the parameter **`resolution`**) and the desired size of each of the constant colored boxes in the pixelation (indicated by the variable **`pixelSize`**). To implement this, make a copy of the **`templateEffect.py`** file and name it **`pixelateEffect.py`**. In the new file, change the name of the class to **`PixelateEffect`**, and change the initialization function declaration to the following:

```
def __init__(self, pixelSize=8,
resolution=[512,512]):
```

Next, change the fragment shader code to the following:

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

void main()
{
    vec2 factor = resolution / pixelSize;
    vec2 newUV = floor( UV * factor ) / factor;
    vec4 color = texture2D(texture, newUV);
    fragColor = color;
}
```

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

```
self.addUniform("float", "pixelSize", pixelSize)
self.addUniform("vec2", "resolution", resolution)
```

To create the vignette effect from Figure 5.31, you will use the UV coordinates to recalculate the clip space coordinates, which are (0, 0) at the center and extend between –1 and 1 on each axis. The color of each fragment will be interpolated between the texture color and a specified dimming color; the color is typically black (which is the reason for the "dimming" terminology), but other colors may be used just as well. The amount of interpolation will depend on the distance between the fragment position and the origin. The distance at which dimming begins is specified by the parameter **`dimStart`**, while the distance at which the pixel matches the dimming color is specified by the parameter **`dimEnd`**. To implement this, make a copy of the **`templateEffect.py`** file and name it **`vignetteEffect.py`**. In the new file, change the name of the class to **`VignetteEffect`**, and change the initialization function declaration to the following:

```
def __init__(self, dimStart=0.4, dimEnd=1.0,
   dimColor=[0,0,0]):
```

Next, change the fragment shader code to the following:

```
in vec2 UV;
uniform sampler2D texture;
uniform float dimStart;
uniform float dimEnd;
uniform vec3 dimColor;
out vec4 fragColor;

void main()
{
    vec4 color = texture2D(texture, UV);

    // calculate position in clip space from UV
         coordinates
    vec2 position = 2 * UV - vec2(1,1);
    // calculate distance (d) from center, which
      affects brightness
    float d = length(position);
    // calculate brightness (b) factor:
```

```
    //   when d=dimStart, b=1; when d=dimEnd, b=0.
    float b = (d - dimEnd)/(dimStart - dimEnd);
    // prevent oversaturation
    b = clamp(b, 0, 1);
    // mix the texture color and dim color
    fragColor = vec4( b * color.rgb + (1-b) *
      dimColor, 1 );
}
```

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

```
self.addUniform("float", "dimStart", dimStart)
self.addUniform("float", "dimEnd", dimEnd)
self.addUniform("vec3", "dimColor", dimColor)
```

The last effect that will be introduced will be reducing the color precision in an image, illustrated in Figure 5.34, which was used to produce the composite effect illustrated in Figure 5.32.

The method for creating this effect is similar to that from the pixelation effect. In this case, the texture colors are rounded to a particular level of precision determined by the parameter **levels**. To implement this, make a copy of the **templateEffect.py** file and name

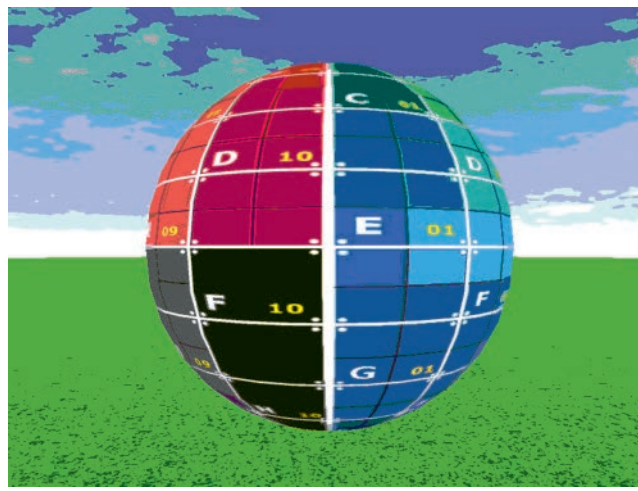

FIGURE 5.34 Color reduction postprocessing effect.

it **colorReduceEffect.py**. In the new file, change the name of the class to **ColorReduceEffect**, and change the initialization function declaration to the following:

```
def __init__(self, levels=4):
```

Next, change the fragment shader code to the following:

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

void main()
{
    vec4 color = texture2D(texture, UV);
    vec4 reduced = round(color * levels) / levels;
    reduced.a = 1.0;
    fragColor = reduced;
}
```

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

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

With all these effects are your disposal, you can also experiment with different parameter settings and combinations of effects. In particular, to recreate the compound effect shown in Figure 5.32, in your application, after all necessary import statements have been added, replace the code involving the **addEffect** function with the following code, to set up a chain of postprocessing effects.

```
self.postprocessor.addEffect(
     TintEffect(tintColor=[0,1,0]) )
self.postprocessor.addEffect(
     ColorReduceEffect(levels=5) )
self.postprocessor.addEffect( PixelateEffect
   (resolution=[800,600] ) )
```

5.10 SUMMARY AND NEXT STEPS

In this chapter, you extended the graphics framework, adding the ability to apply textures to the surfaces of geometric shapes. You created multiple animated effects, used algorithms to generate textures involving randomness and inspired by natural phenomena, generated textures involving text, and set up a postprocessing render system with a collection of special effects. This required additions to many of the geometry classes, modifications of multiple core classes (including **`Uniform`**, **`Matrix`**, **`Object3D`**, **`Camera`**, and **`Renderer`**), and the introduction of new material classes.

In the next chapter, you will add even more realism and sophistication to your three-dimensional scenes by learning how to add lights, shading, and shadows to the graphics framework.

CHAPTER 6

Light and Shadow

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

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

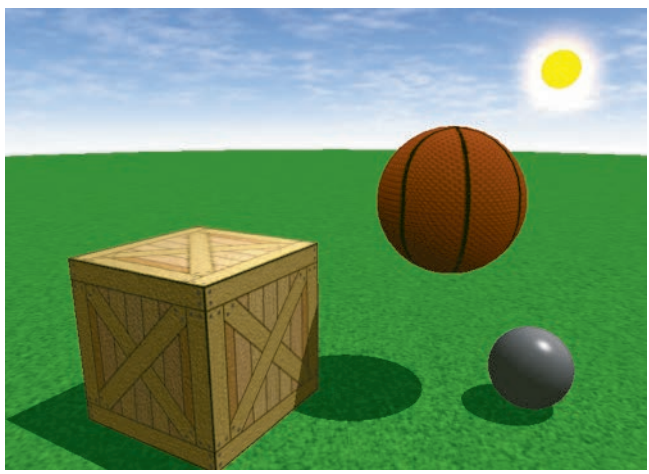

FIGURE 6.1 Rendered scene with lighting effects.

DOI: 10.1201/9781003181378-6

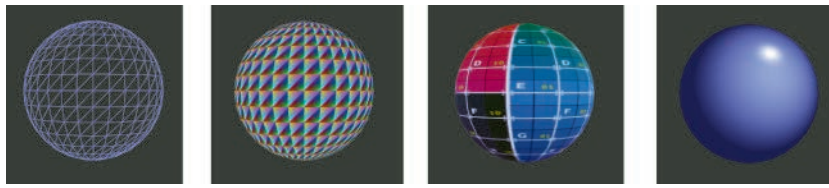

FIGURE 6.2 Four techniques to visualize a sphere: wireframe, vertex colors, texture, and shading.

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

6.1 INTRODUCTION TO LIGHTING

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

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

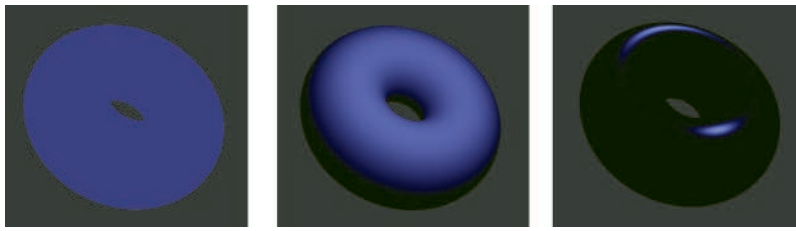

FIGURE 6.3 Ambient, diffuse, and specular lighting on a torus-shaped surface.

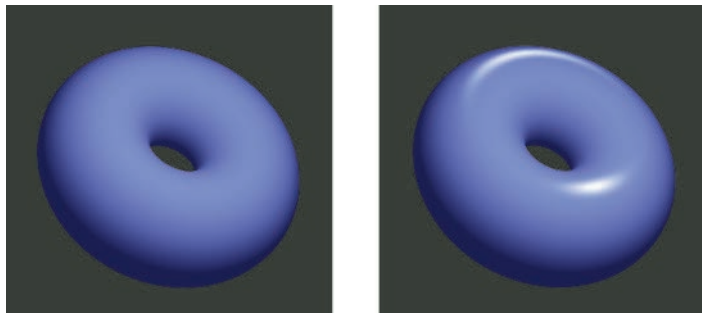

FIGURE 6.4 The Lambert and Phong illumination models on a torus-shaped surface.

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

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

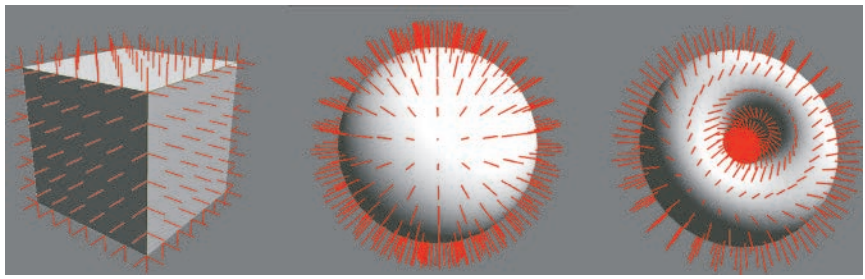

FIGURE 6.5 Normal vectors to a box, a sphere, and a torus.

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

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

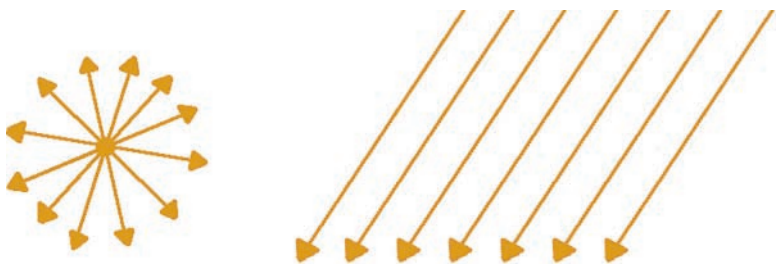

FIGURE 6.6 Directions of emitted light rays for point light (left) and directional light (right).

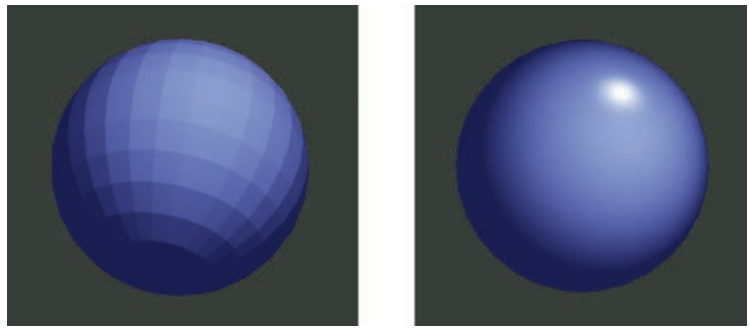

FIGURE 6.7 Rendering a sphere using flat shading (left) and Phong shading (right).

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

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

6.2 LIGHT CLASSES

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

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

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

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

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

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

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

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

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

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

```
import numpy
```

Then, add the following three functions to the class:

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

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

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

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

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

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

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

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

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

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

6.3 NORMAL VECTORS

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

6.3.1 Rectangles

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

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

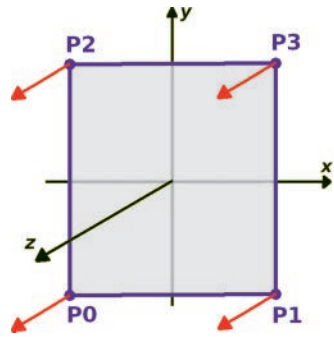

FIGURE 6.8 Normal vectors for a rectangle.

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

6.3.2 Boxes

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

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

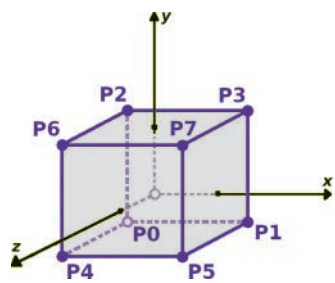

FIGURE 6.9 Vertices in a box geometry.

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

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

6.3.3 Polygons

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

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

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

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

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

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

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

6.3.4 Parametric Surfaces

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

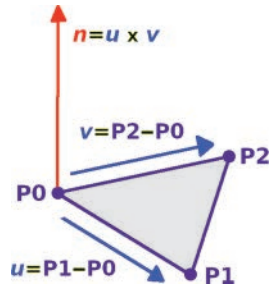

FIGURE 6.10 Calculating the normal vector for a triangle.

which takes two vectors as input and produces a vector perpendicular to both of the input vectors. Since each triangle is defined by three points P0, P1, and P2, one can subtract these points in pairs to create two vectors $\boldsymbol{v} = \text{P1} - \text{P0}$ and $\boldsymbol{w} = \text{P2} - \text{P0}$ aligned with the edges of the triangle. Then, the cross product of $\boldsymbol{v}$ and $\boldsymbol{w}$ results in the desired face normal vector $\boldsymbol{n}$. This calculation is illustrated in Figure 6.10.

To calculate the vertex normal vector at a point P0 on the surface involves the same process, except that the points P1 and P2 used for this calculation are chosen to be very close to P0 in order to more precisely approximate the exact normal to the surface. In particular, assume that the surface is defined by the parametric function S and that $\text{P0} = S(u,v)$. Let h be a small number, such as $h = 0.0001$. Then, define two additional points $\text{P1} = S(u+h,v)$ and $\text{P2} = S(u,v+h)$. With the three points P0, P1, and P2, one may then proceed exactly as before to obtain the desired vertex normal vector.

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

```
import numpy
```

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

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

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

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

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

```
vertexNormalData = []
faceNormalData   = []
```

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

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

# face normal vectorsfn0 = calcNormal(pA, pB, pC)
fn1 = calcNormal(pA, pC, pD)
faceNormalData += [fn0,fn0,fn0, fn1,fn1,fn1]
```

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

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

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

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

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

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

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

6.4 USING LIGHTS IN SHADERS

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

6.4.1 Structs and Uniforms

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

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

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

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

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

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

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

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

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

6.4.2 Light-Based Materials

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

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

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

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

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

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

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

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

        self.setProperties(properties)

    def updateRenderSettings(self):

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

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

        glLineWidth(self.settings["lineWidth"])
```

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

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

```
    vec3 attenuation;
};

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

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

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

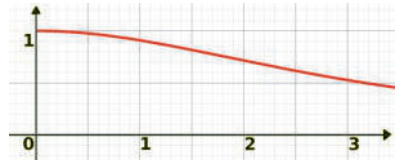

FIGURE 6.11 Attenuation of a light source as a function of distance.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
```

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

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

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

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

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

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

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

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

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

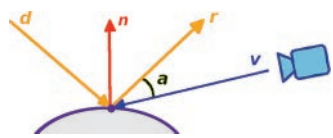

FIGURE 6.12 The vectors used in the calculation of specular highlights.

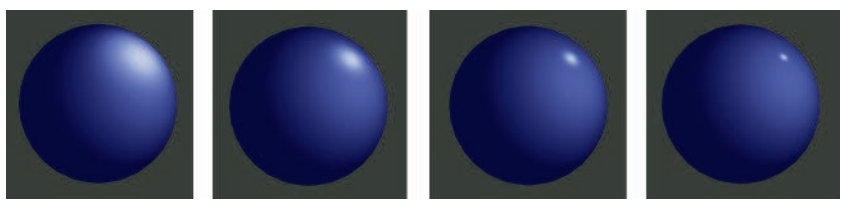

FIGURE 6.13 The effect of increasing shininess in specular lighting.

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

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

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

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

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

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

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

6.5 RENDERING SCENES WITH LIGHTS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.6 EXTRA COMPONENTS

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

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

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

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

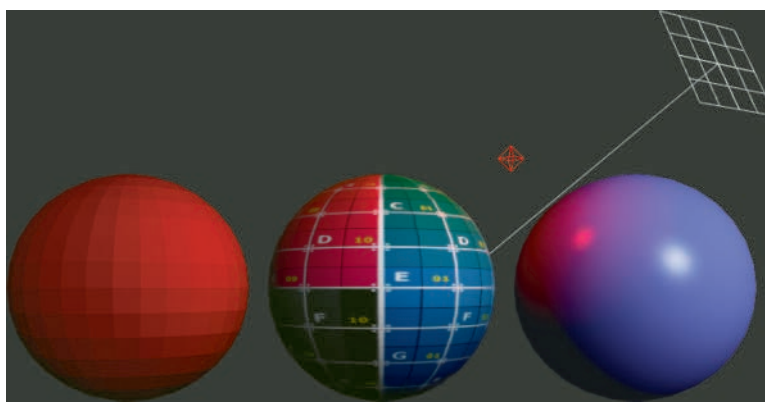

FIGURE 6.15 Objects illustrating the properties of point lights and directional lights.

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

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

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

class PointLightHelper(Mesh):

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

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

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

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

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

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

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

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

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

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

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

6.7 BUMP MAPPING

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

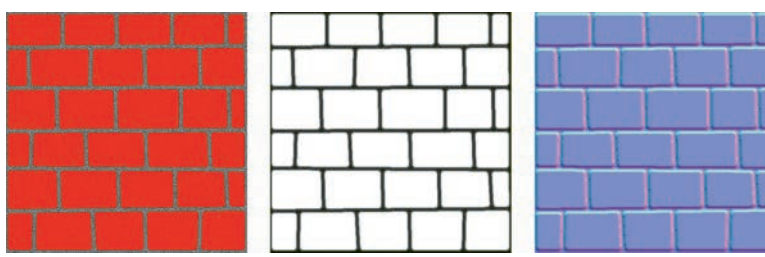

FIGURE 6.16 A color texture, height map texture, and normal map texture for a brick wall.

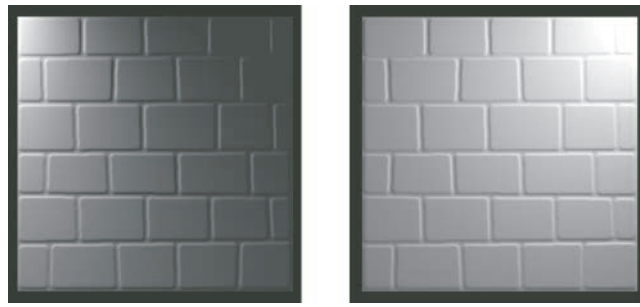

FIGURE 6.17 A normal map applied to a rectangle, with point light source on the upper-left (left) and upper right (right).

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

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

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

In the—**init**—function, add the parameter and default value **bumpTexture=None**.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

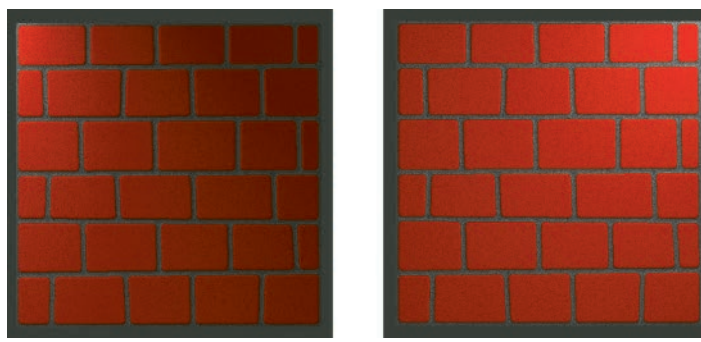

FIGURE 6.18 Combining color texture with normal map texture.

6.8 BLOOM AND GLOW EFFECTS

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

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

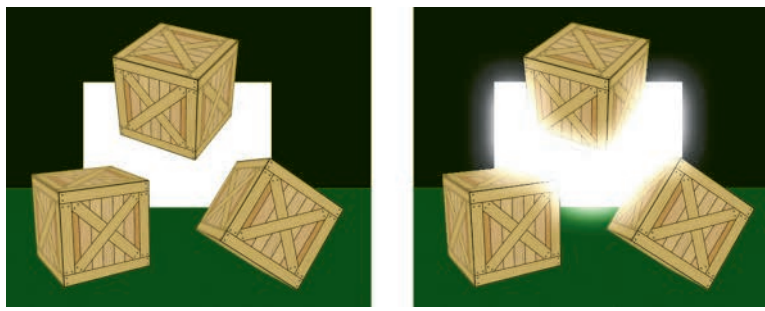

FIGURE 6.19 Light bloom postprocessing effect.

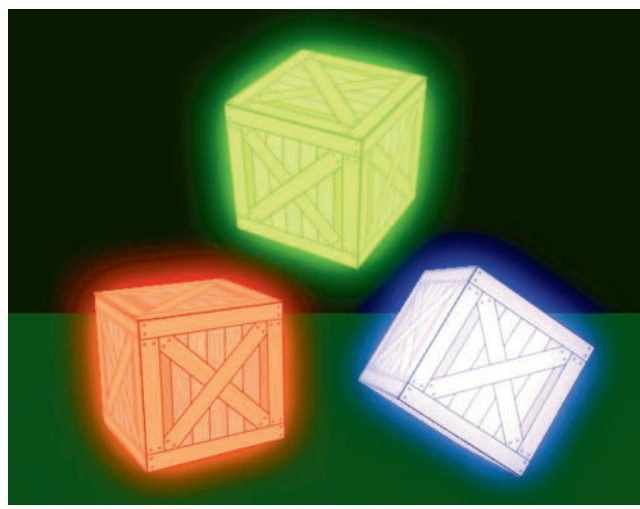

FIGURE 6.20 Glow postprocessing effect.

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

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

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

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

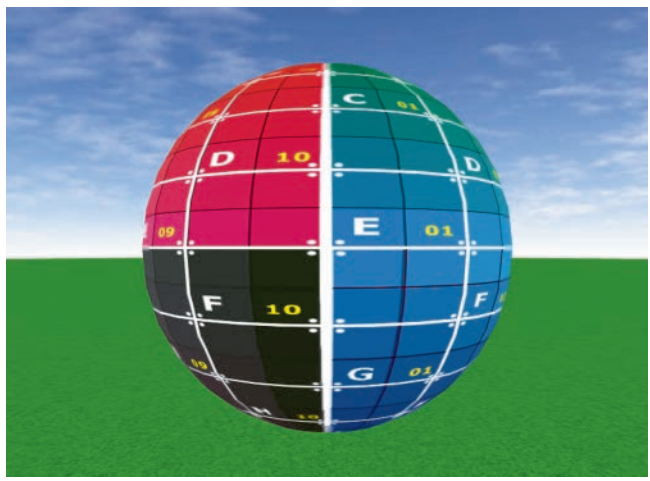

FIGURE 6.21 Default scene with no postprocessing effects.

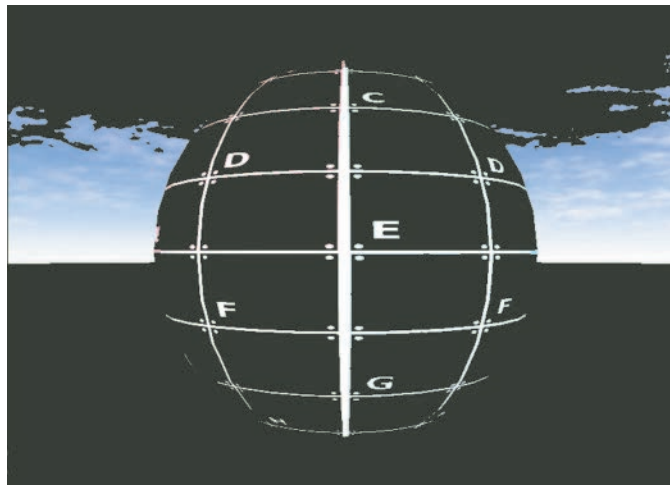

FIGURE 6.22 Brightness filter postprocessing effect.

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

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

Next, change the fragment shader code to the following:

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

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

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

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

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

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

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

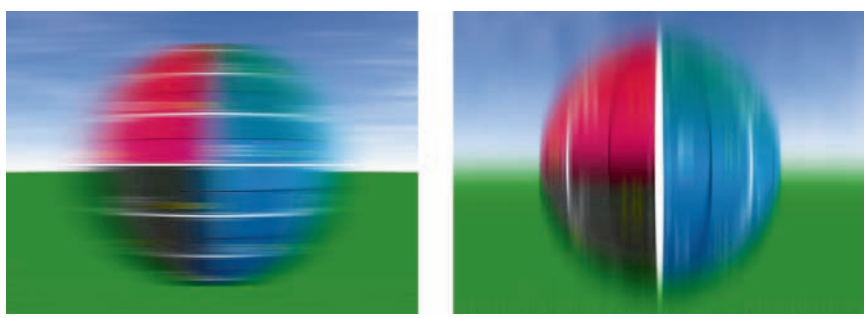

FIGURE 6.23 Horizontal blur (left) and vertical blur (right).

FIGURE 6.24 A combined two-pass blur postprocessing effect.

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

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

Next, change the fragment shader code to the following:

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

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

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

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

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

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

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

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

FIGURE 6.25 Additive blending postprocessing effect.

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

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

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

Next, change the fragment shader code to the following:

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

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

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

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

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

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

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

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

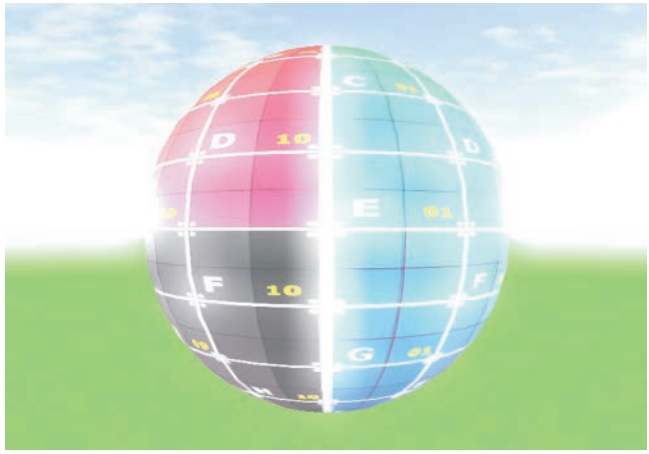

FIGURE 6.26 Light bloom postprocessing effect.

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

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

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

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

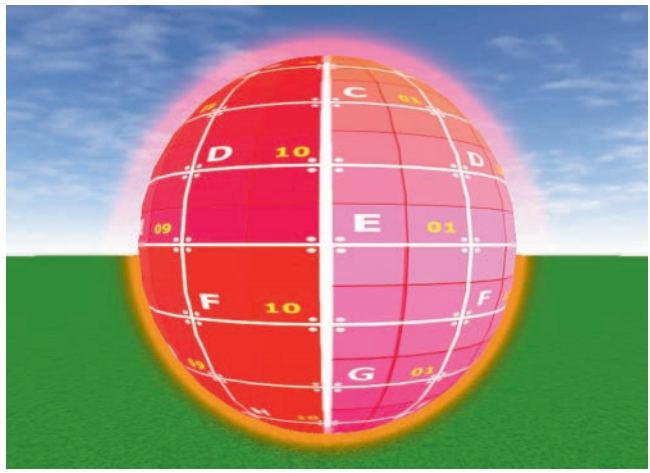

FIGURE 6.27 Glow postprocessing effect.

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

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

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

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

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

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

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

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

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

6.9 SHADOWS

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

6.9.1 Theoretical Background

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

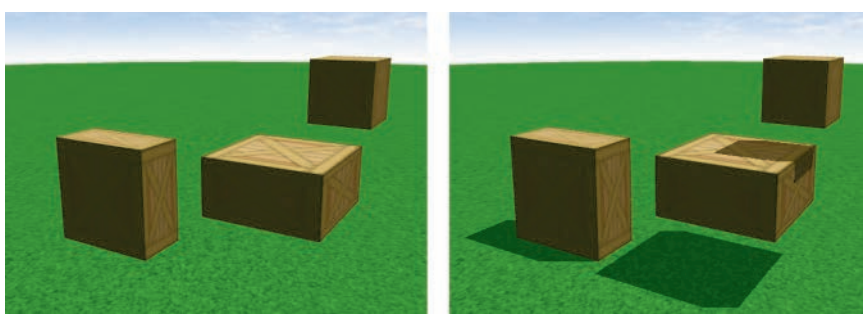

FIGURE 6.28 A scene without shadows (left) and with shadows (right).

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

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

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

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

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

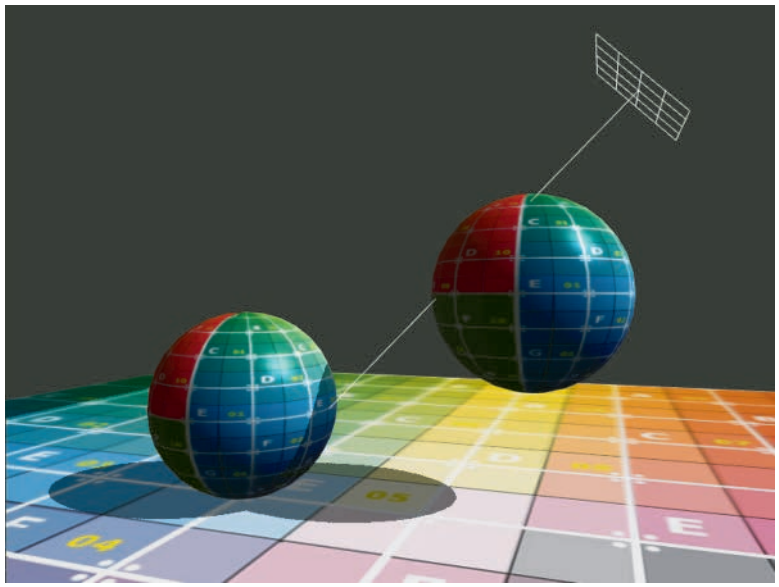

FIGURE 6.29 A scene including a directional light and shadows.

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

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

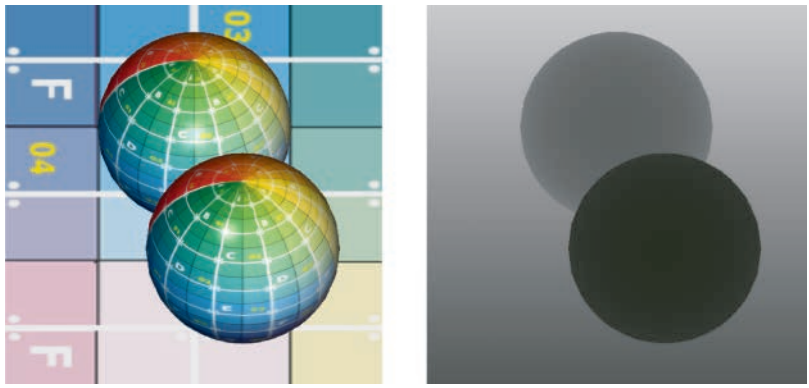

FIGURE 6.30 A scene viewed from a directional light (left) and the corresponding depth texture (right).

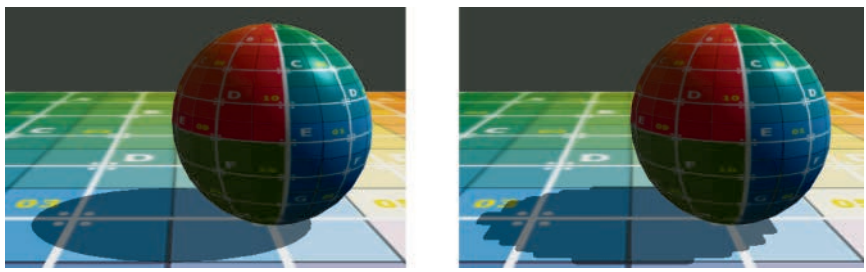

FIGURE 6.31 Shadow pixelation artifacts.

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

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

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

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

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

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

6.9.2 Adding Shadows to the Framework

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

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

```
from material.material import Material

class DepthMaterial(Material):

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

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

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

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

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

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

class Shadow(object):

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

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

        super().__init__()

        # must be directional light
        self.lightSource = lightSource

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    shadowPosition0 = vec3( temp0 );
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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


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


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


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

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

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

```
from light.shadow import Shadow
```

In the **__init__** function, add the following line of code:

```
self.shadowsEnabled = False
```

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

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

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

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

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

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

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

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

```
glClear(GL_COLOR_BUFFER_BIT)
glClear(GL_DEPTH_BUFFER_BIT)

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

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

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

    # bind VAO
    glBindVertexArray( mesh.vaoRef )

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

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

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

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

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

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

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

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

    def initialize(self):
```

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

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


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

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

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

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

self.renderer.enableShadows( self.dirLight )

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

```
        # self.scene.add( shadowDisplay )

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

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

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

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

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

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

6.10 SUMMARY AND NEXT STEPS

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

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

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

Index