Chapter 5. Pygame and 3D

In our previous chapters, we developed our 2D games with Python modules that are built on top of a graphical user interface library, such as `Tkinter` and `Pyglet`. This allowed us to start coding our games without worrying about the lower-level details.

Now we will develop our first 3D game with Python, which will require an understanding of some basic principles of OpenGL, a popular multiplatform API for building 2D and 3D applications. You will learn how to integrate these programs with Pygame, a Python library commonly used to create sprite-based games.

In this chapter, we will cover the following topics:

- A steady approach to PyOpenGL and Pygame
- Initializing an OpenGL context
- Understanding the different modes that can be enabled with OpenGL
- How to render lights and simple shapes
- Integrating OpenGL with Pygame
- Drawing primitives and performance improvements

Installing packages

PyOpenGL is a package that offers Python bindings to OpenGL and related APIs, such as GLU and GLUT. It is available on the Python package Index, so you can easily install it via `pip`:

```
$ pip install PyOpenGL
```

However, we will need **freeglut** for our first examples, before we integrate OpenGL with Pygame. Freeglut is a third-party library that is not included if you install the package from PyPI.

On Windows, an alternative is to download and install the compiled binaries from http://www.lfd.uci.edu/~gohlke/pythonlibs/#pyopengl. Remember to install the version for Python 3.4.

Pygame is the other package that we will need in this chapter. It can be downloaded from the official website at http://www.pygame.org/download.shtml. You can install it from source if you want to; the compilation page contains the steps for building Pygame on different platforms.

Windows users can directly use the MSI for Python 3.2 or download *Unofficial Windows Binaries* from the Christoph Gohlke's website (http://www.lfd.uci.edu/~gohlke/pythonlibs/).

Macintosh users can find the instructions required to compile it from source on the Pygame website at http://pygame.org/wiki/macintosh.

Getting started with OpenGL

OpenGL is a broad topic in itself, and it is possible to find plenty of tutorials, books, and other resources, usually targeted at C or C++.

Since this chapter is not intended to be a comprehensive guide for this specification, we will take advantage of GLUT, which stands for **OpenGL Utility Toolkit**. It is widely used in small applications because of its simplicity and portability, and the bindings are implemented in PyOpenGL.

GLUT will help us perform some basic operations, such as creating windows and handling input events.

Tip

GLUT licensing

Unfortunately, GLUT is not in the public domain. The copyright is maintained by its author, Mark Kilgard, who wrote it for the sample programs included in *Red Book*, the official OpenGL programming guide.

This is the reason we are using `freeglut`, one of the open source alternatives that implement the GLUT API.

Initializing the window

The first lines of our script will be the `import` statements as well as the definition of our `App` class and its `__init__` method.

Apart from the OpenGL API and GLUT, we import the **OpenGL Utility Library** (**GLU**). GLU is usually distributed with the basic OpenGL package, and we will use a couple of functions offered by this library in our example:

```
import sys
import math

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *

class App(object):
    def __init__(self, width=800, height=600):
        self.title = b'OpenGL demo'
        self.width = width
        self.height = height
        self.angle = 0
        self.distance = 20
```

You may wonder what the `b` before the `'OpenGL demo'` string means. It represents a binary string, and it is one of the differences between Python 2 and 3. Therefore, if you find a GLUT program written in Python 2, remember that the string title of the window must be defined as a binary string in order to work with Python 3.

With these instance members, we can call our OpenGL initialization functions:

```
    def start(self):
        glutInit()
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH)
        glutInitWindowPosition(50, 50)
        glutInitWindowSize(self.width, self.height)
        glutCreateWindow(self.title)

        glEnable(GL_DEPTH_TEST)
        glEnable(GL_LIGHTING)
        glEnable(GL_LIGHT0)
```

Step by step, our `start` method performs the following operations:

- **glutInit()**: This initializes the GLUT library. While it is possible to pass parameters to this function, we will leave this call without any arguments.
- **glutInitDisplayMode()**: This sets the display mode of the top-level window that we will create. The mode is the bitwise OR of a few GLUT display mode masks. `GLUT_DOUBLE` is the mode for the double buffer, which creates separate front and back buffers. While one of these buffers is being displayed, the other one is being rendered. On the other hand, `GLUT_DEPTH` requests a depth buffer for the window. It stores the `z` coordinate of each generated pixel, and if the same pixel is rendered for a second time because two objects overlap, it determines which object is closer to the

camera, that is, reproducing the depth perception.

- **glutInitWindowPosition()** and **glutInitWindowsSize()**: These set the initial position of the window and its size. According to our width and height instance members, it indicates to create a window of 800 x 600 pixels with an offset of 50 pixels in the `x` and `y` axes from the top-left corner of the screen.
- **glutCreateWindow()**: This creates the top-level window of our application. The argument passed to this function is a binary string for use as the window title.
- **glEnable()**: This is the function used to enable the GL capabilities. In our app, we call it with the following values:
    - `GL_DEPTH_TEST`: This performs depth comparisons and updates the depth buffer.
    - `GL_LIGHTING`: This enables lighting.
    - `GL_LIGHT0`: This enables `Light0`. PyOpenGL defines a specific number of light constants—from `GL_LIGHT0` to `GL_LIGHT8`—but the particular implementation of OpenGL that you are running might allow more than this number.

Tip

Lighting and colors

When lighting is enabled, the colors are not determined by the `glColor` functions but by the combination of the lighting computation and the material colors set by `glMaterial`. To combine lighting with `glColor`, it is required that you enable `GL_COLOR_MATERIAL` first:

```
glEnable(GL_COLOR_MATERIAL)
# ...
glColor4f(r, g, b, a)
# Draw polygons
```

Once we have initialized GLUT and enabled the GL capabilities, we complete our `start()` method by specifying the clear color, setting the perspective, and starting the main loop:

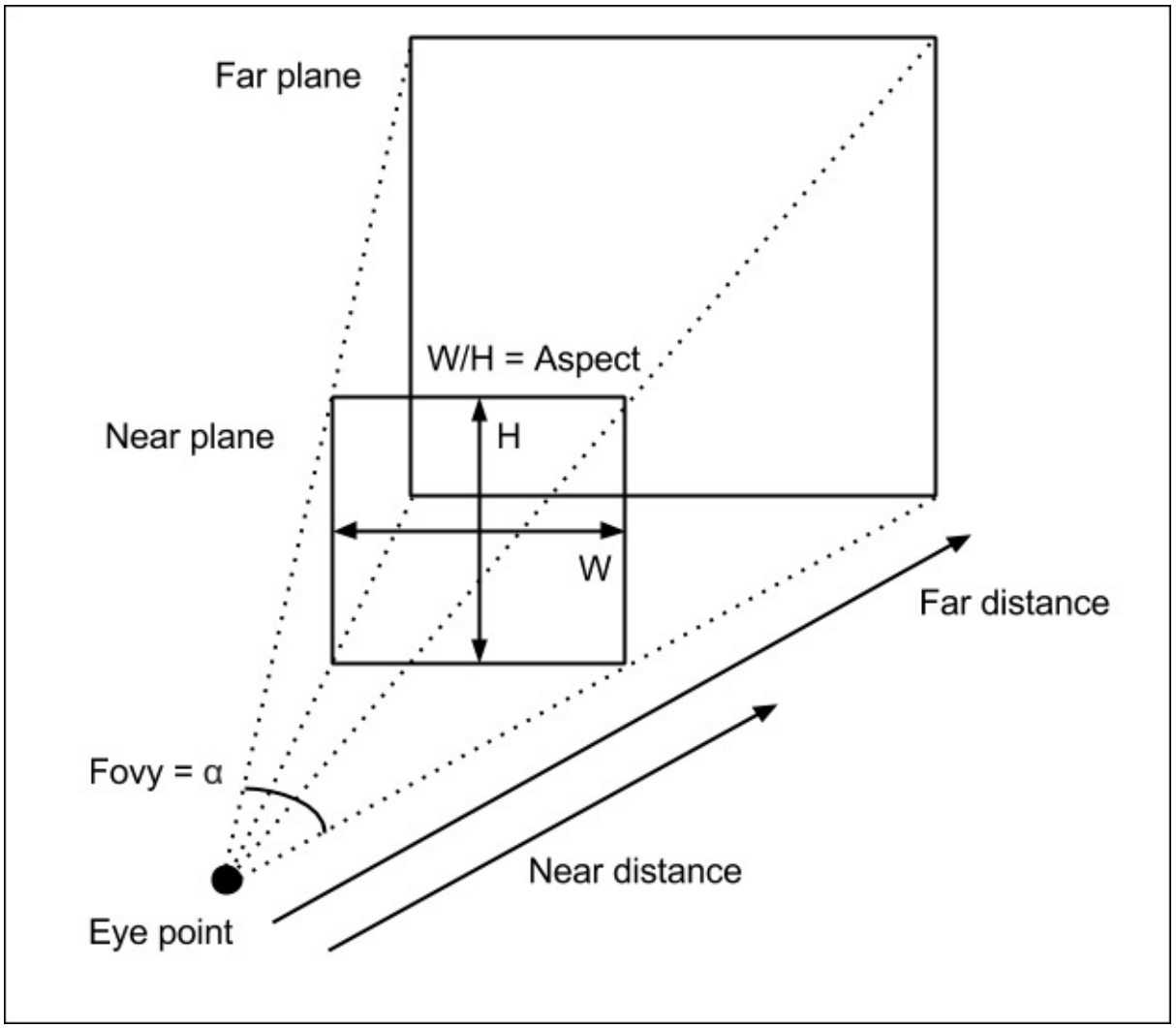
```
    def start(self):
        # ...
        glClearColor(.1, .1, .1, 1)
        glMatrixMode(GL_PROJECTION)
        aspect = self.width / self.height
        gluPerspective(40., aspect, 1., 40.)
        glMatrixMode(GL_MODELVIEW)

        glutDisplayFunc(self.display)
        glutSpecialFunc(self.keyboard)
        glutMainLoop()

    def keyboard(self, key, x, y):
        pass
```

These statements perform the following operations:

- **glClearColor()**: This defines the clear values for the color buffer; that is, each pixel will have this value if no other color is rendered in this pixel.
- **glMatrixMode()**: This sets the matrix stack mode for matrix operations, in this case

to the projection matrix stack. OpenGL concatenates matrix operations for hierarchical modes, making it easy to compose the transformation of a child object relative to its parent. With `GL_PROJECTION`, we set the matrix mode for the projection matrix stack.

- **gluPerspective()**: The previous statement sets the projection matrix stack as the current stack. With this function, we can generate the perspective projection matrix. The parameters that generate this matrix are as follows:
  - `fovy`: The view angle in degrees in the `y` direction.
  - `aspect`: This is the aspect ratio of the field of view. It is the ratio of the viewport width to the viewport height.
  - `zNear`: The distance from the viewer to the **Near plane**.
  - `zFar`: The distance from the viewer to the **Far plane**.



With `glMatrixMode(GL_MODELVIEW)`, we set the `modelview` matrix stack, which is the initial value, as the current matrix mode.

The last three GLUT calls do the following:

- **glutDisplayFunc()**: This receives the function that will be invoked to display the window.

- **glutSpecialFunc()**: This sets the keyboard callback for the current window. Note that this callback will be triggered only when the keys represented by the `GLUT_KEY_*` constants are pressed.
- **glutMainLoop()**: This starts the main loop of the application.

With the OpenGL context initialized, we are able to call the OpenGL functions that will render our scene.

Tip

The OpenGL and GLUT reference

As you may have already noticed, the OpenGL and GLUT specifications define a large number of functions. You can find the bindings of these APIs implemented by PyOpenGL on the official website at http://pyopengl.sourceforge.net/documentation/manual-3.0/index.html.

Drawing shapes

Our `display()` function performs the very common tasks of a main game loop.

It first clears the screen, then sets up a viewing transformation (we will see what this means after the snippet), and finally renders the light and draws the game objects:

```
    def display(self):
        x = math.sin(self.angle) * self.distance
        z = math.cos(self.angle) * self.distance

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glLoadIdentity()
        gluLookAt(x, 0, z,
                  0, 0, 0,
                  0, 1, 0)

        glLightfv(GL_LIGHT0, GL_POSITION, [15, 5, 15, 1])
        glLightfv(GL_LIGHT0, GL_DIFFUSE, [1., 1., 1., 1.])
        glLightfv(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.1)
        glLightfv(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.05)

        glPushMatrix()
        glMaterialfv(GL_FRONT, GL_DIFFUSE, [1., 1., 1., 1.])
        glutSolidSphere(2, 40, 40)
        glPopMatrix()

        glPushMatrix()
        glTranslatef(4, 2, 0)
        glMaterialfv(GL_FRONT, GL_DIFFUSE, [1.,0.4,0.4,1.0])
        glutSolidSphere(1, 40, 40)
        glPopMatrix()

        glutSwapBuffers()
```

These are the operations that `display()` performs:

- **glClear()**: With the `GL_COLOR_BUFFER_BIT` and `GL_DEPTH_BUFFER_BIT` masks, this clears the color and depth buffers.
- **glLoadIdentity()**: This loads the identity matrix as the current matrix. The identity matrix is a 4 x 4 matrix with ones in the main diagonal and zeros everywhere else. This makes the stack matrix start over at the origin, which is useful if you have previously applied some matrix transformations.
- **gluLookAt()**: This creates a viewing matrix. The first three parameters are the `x`, `y`, and `z` coordinates of the eye point. The next three parameters are the `x`, `y`, and `z` coordinates of the reference point, that is, the position the camera is looking at. Finally, the last three parameters specify the direction of the up vector (usually, it is 0, 1, 0).
- **glLightfv()**: This sets the parameters of light source 0 (`GL_LIGHT0`). The following parameters are specified in our example:
    - `GL_POSITION`: This defines the position of the light

    - `GL_DIFFUSE`: This sets the RGBA intensity of the light
    - `GL_CONSTANT_ATTENUATION`: This specifies the constant attenuation factor
    - `GL_LINEAR_ATTENUATION`: This specifies the linear attenuation factor

Once the lighting attributes are set, we can start rendering basic shapes with GLUT. If we draw the objects first, lighting will not be applied correctly:

- **glPushMatrix()**: This pushes a new matrix into the current matrix stack, identical to the one below it. While we do this, we can apply transformations such as `glTranslate` and `glRotate`, to this matrix. We will render our first sphere at the origin, but the second one will be transformed with `glTranslate`.
- **glTranslate()**: This multiplies the current matrix by the translation matrix. In our example, the translation values for the second sphere are `4` for the `x` axis, and `2` for the `y` axis.
- **glMaterialfv()**: This sets the material parameters of the front face, as it is called with `GL_FONT`. With `GL_DIFFUSE`, we specify that we are setting the RGBA reflectance of the material.
- **glutSolidSphere()**: Through GLUT, this routine allows us to easily draw a solid sphere. It receives the sphere's radius as the first argument, and the number of slices and stacks into which the sphere will be subdivided. The greater these values are, the rounder the sphere will be.
- **glPopMatrix()**: This pops the current matrix from the stack. If we did not do this, each new object rendered would be a child of the previous one.

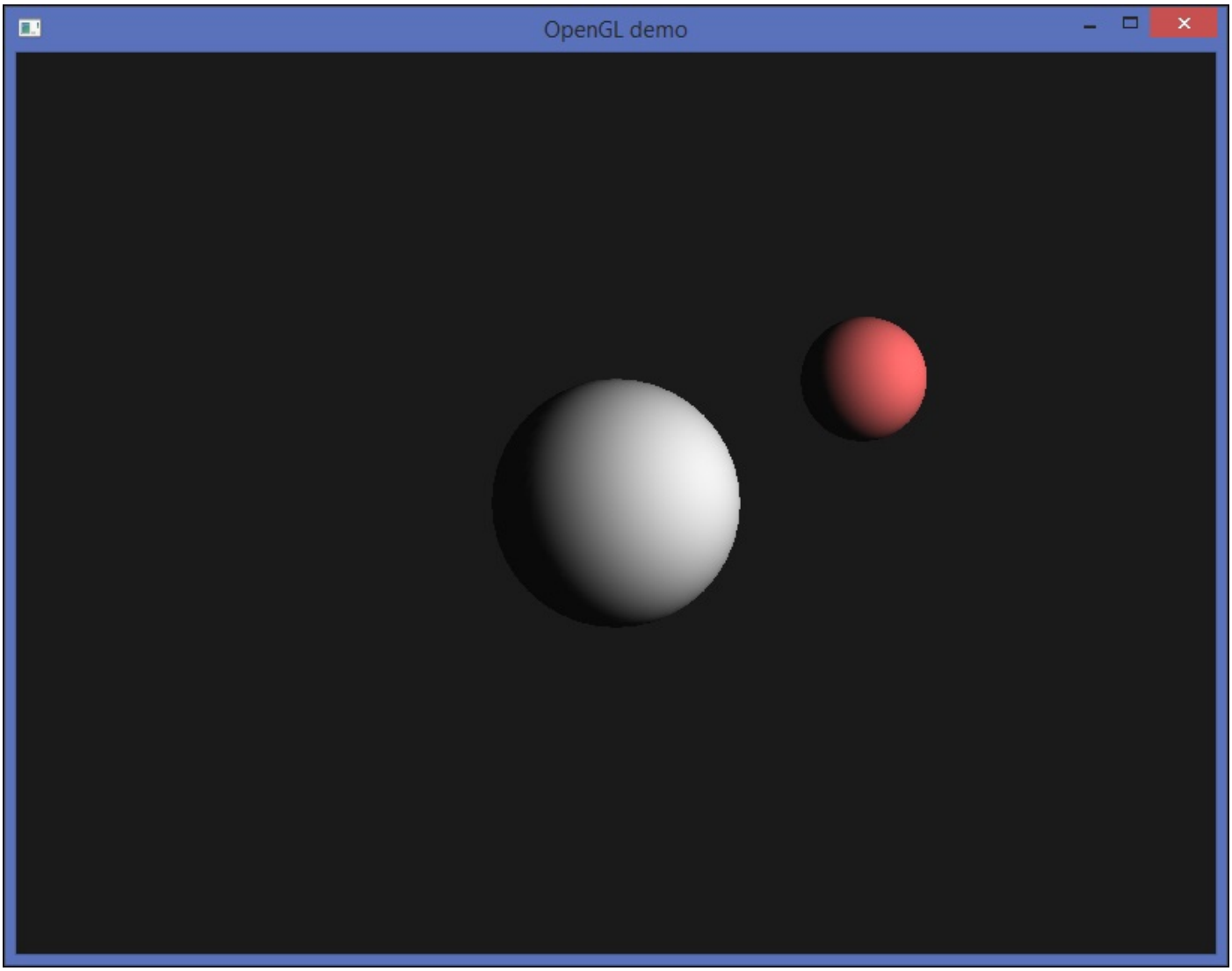
Finally, we switch the buffers with `glutSwapBuffers()`. If double buffering was not enabled, we should call the single buffer equivalent—`glFlush()`.

Running the demo

As usual, we check whether the module is the main script for starting the application:

```
if __name__ == '__main__':
    app = App()
    app.start()
```

If you run the complete application, the result will look like what is shown in the following screenshot:



Refactoring our OpenGL program

As you may have seen, this example uses enough OpenGL calls to grow out of control if we do not structure our code. That's why we are going to apply some object-oriented principles to achieve a better organization, without modifying the order of the calls or losing any functionality.

The first step will be to define a `Light` class. It will hold the attributes needed to render the light:

```
class Light(object):
    enabled = False
    colors = [(1.,1.,1.,1.), (1.,0.5,0.5,1.),
              (0.5,1.,0.5,1.), (0.5,0.5,1.,1.)]

    def __init__(self, light_id, position):
        self.light_id = light_id
        self.position = position
        self.current_color = 0
```

Besides, this modularization will help us implement a new functionality: changing the color of the light. We set the current color index to 0, and we will iterate over the different colors defined in `Light.colors` each time the `switch_color()` method is called.

The `render()` method respects the original implementation of lighting from our non-refactored version:

```
    def render(self):
        light_id = self.light_id
        color = Light.colors[self.current_color]
        glLightfv(light_id, GL_POSITION, self.position)
        glLightfv(light_id, GL_DIFFUSE, color)
        glLightfv(light_id, GL_CONSTANT_ATTENUATION, 0.1)
        glLightfv(light_id, GL_LINEAR_ATTENUATION, 0.05)

    def switch_color(self):
        self.current_color += 1
        self.current_color %= len(Light.colors)
```

Finally, we wrap the call to enable lighting with the `enable()` method and a `class` attribute:

```
    def enable(self):
        if not Light.enabled:
            glEnable(GL_LIGHTING)
            Light.enabled = True
        glEnable(self.light_id)
```

Another improvement is the creation of a `Sphere` class. This class will allow us to customize the radius, position, and color of each instance:

```
class Sphere(object):
    slices = 40
    stacks = 40
```

```
    def __init__(self, radius, position, color):
        self.radius = radius
        self.position = position
        self.color = color

    def render(self):
        glPushMatrix()
        glTranslatef(*self.position)
        glMaterialfv(GL_FRONT, GL_DIFFUSE, self.color)
        glutSolidSphere(self.radius, Sphere.slices, Sphere.stacks)
        glPopMatrix()
```

With these classes, we can adapt our `App` class and create the instances that we will render in the main loop:

```
class App(object):
    def __init__(self, width=800, height=600):
        # ...
        self.light = Light(GL_LIGHT0, (15, 5, 15, 1))
        self.sphere1 = Sphere(2, (0, 0, 0), (1, 1, 1, 1))
        self.sphere2 = Sphere(1, (4, 2, 0), (1, 0.4, 0.4, 1))
```

Remember that before rendering the `light` object, we need to enable OpenGL lighting through the `light.enable()` method:

```
    def start(self):
        # ...
        glEnable(GL_DEPTH_TEST)
        self.light.enable()
        # ...
```

Now the `display()` method becomes succinct and expressive, since the application delegates the OpenGL calls to the object instances:

```
    def display(self):
        x = math.sin(self.angle) * self.distance
        z = math.cos(self.angle) * self.distance
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glLoadIdentity()
        gluLookAt(x, 0, z,
                  0, 0, 0,
                  0, 1, 0)
        self.light.render()
        self.sphere1.render()
        self.sphere2.render()
        glutSwapBuffers()
```

To complete our first sample application, we will add input handling. It allows the player to rotate the camera around the spheres and move forward or away from the center of the scene.

Processing the user input

As we saw earlier, **glutSpecialFunc** takes a callback function that receives the pressed key as the first argument, and the `x` and `y` coordinates of the mouse when the key was pressed.

We will use the right and left arrow keys to move around the spheres, and the up and down arrow keys to approximate or move away from the spheres. Besides all of this, the color of the light will change if we press *F1*, and the application will be closed if the *Insert* key is pressed.

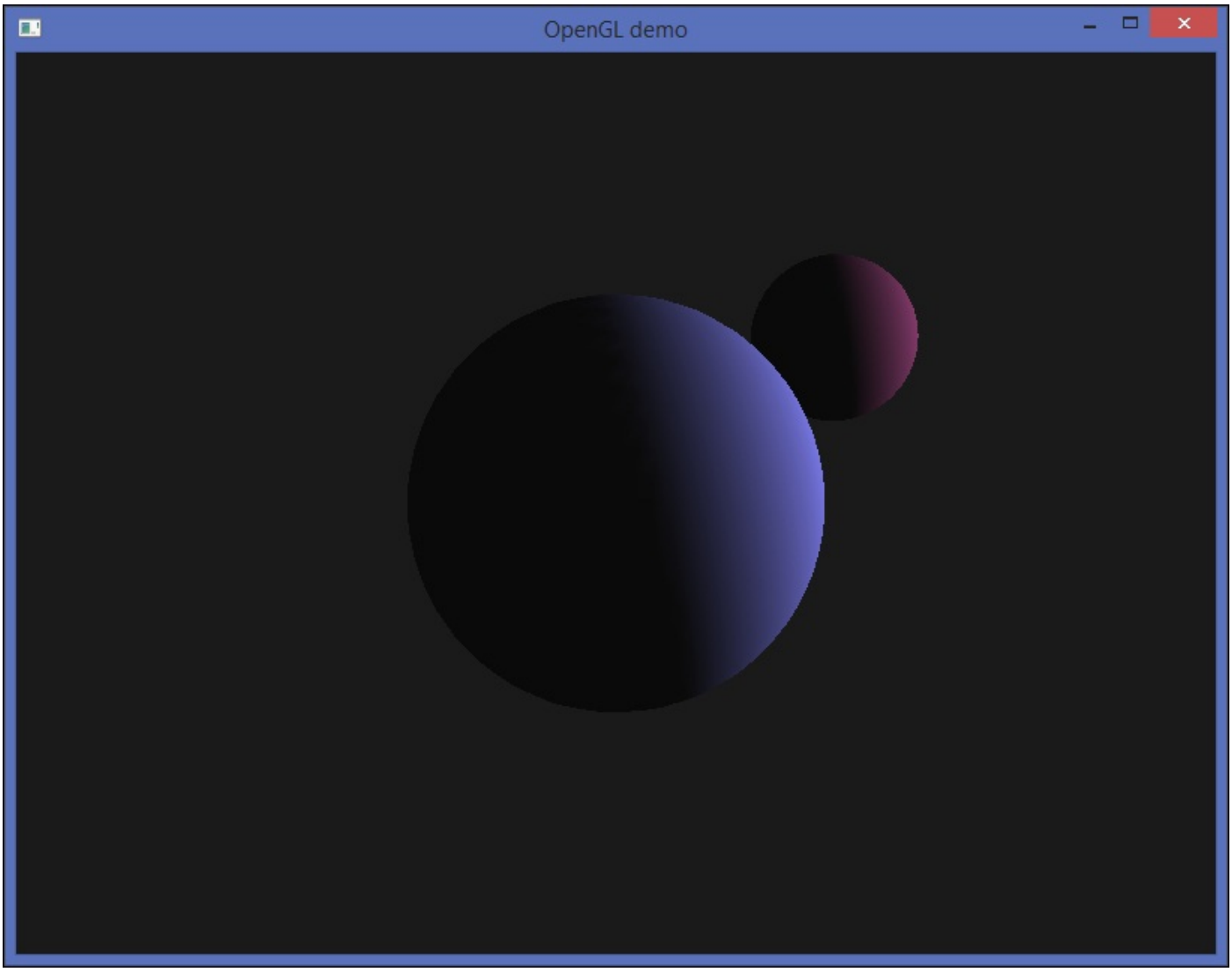
To do so, we will check the values of the key argument with the respective GLUT constants:

```
    def keyboard(self, key, x, y):
        if key == GLUT_KEY_INSERT:
            sys.exit()
        if key == GLUT_KEY_UP:
            self.distance -= 0.1
        if key == GLUT_KEY_DOWN:
            self.distance += 0.1
        if key == GLUT_KEY_LEFT:
            self.angle -= 0.05
        if key == GLUT_KEY_RIGHT:
            self.angle += 0.05
        if key == GLUT_KEY_F1:
            self.light.switch_color()
        self.distance = max(10, min(self.distance, 20))
        self.angle %= math.pi * 2
        glutPostRedisplay()
```

Note that we trimmed the value of the `self.distance` member, so its value is always between `10` and `20`, and `self.angle` is also always between `0` and `2π`. To notify that the current window needs to be redisplayed, we call `glutPostRedisplay()`.

You can check out the `Chapter5_02.py` script, which contains this refactored version of our application.

When you run it, press the arrow keys to rotate around the spheres and *F1* to see how the lighting affects the spheres' materials.


OpenGL demo

Adding the Pygame library

With GLUT, we can write OpenGL programs quickly, primarily because it was aimed to provide routines that make learning OpenGL easier. However, the GLUT API was discontinued in 1998. Nonetheless, there are some popular substitutes in the Python ecosystem.

Pygame is one of these alternatives, and we will see that it can be seamlessly integrated with OpenGL, even simplifying the resulting code for the same program.

Pygame 101

Before we integrate Pygame into our OpenGL program, we will write a sample 2D application to get started with Pygame.

We will `import` Pygame and its locals module, which includes the constants that we will need in our application:

```
import sys
import pygame
from pygame.locals import *

class App(object):
    def __init__(self, width=400, height=300):
        self.title = 'Hello, Pygame!'
        self.fps = 100
        self.width = width
        self.height = height
        self.circle_pos = width/2, height/2
```

Pygame uses regular strings for the window title, so we will define the attribute without adding `b`. Another change is the number of **frames per second** (**FPS**), which we will later find out how to control via Pygame's clock:

```
    def start(self):
        pygame.init()
        size = (self.width, self.height)
        screen = pygame.display.set_mode(size, DOUBLEBUF)
        pygame.display.set_caption(self.title)
        clock = pygame.time.Clock()
        while True:
            dt = clock.tick(self.fps)
            for event in pygame.event.get():
                if event.type == QUIT:
                    pygame.quit()
                    sys.exit()
            pressed = pygame.key.get_pressed()
            x, y = self.circle_pos
            if pressed[K_UP]: y -= 0.5 * dt
            if pressed[K_DOWN]: y += 0.5 * dt
            if pressed[K_LEFT]: x -= 0.5 * dt
            if pressed[K_RIGHT]: x += 0.5 * dt
            self.circle_pos = x, y
            screen.fill((0, 0, 0))
            pygame.draw.circle(screen, (0, 250, 100),
                               (int(x), int(y)), 30)
            pygame.display.flip()
```

We initialize the Pygame modules with `pygame.init()`, and then we create a screen with a given width and height. The `DOUBLEBUF` flag is passed so as to enable double buffering, which has the benefits we mentioned previously.

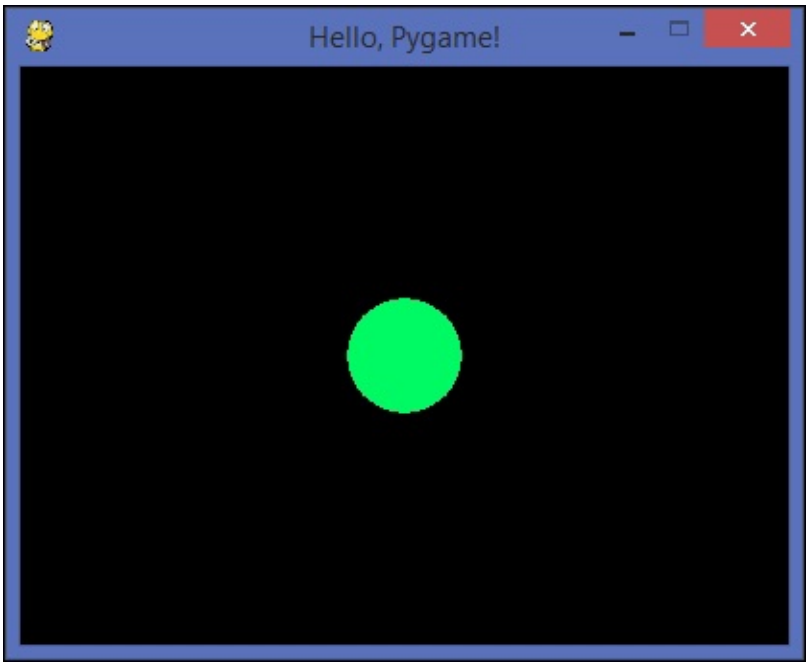
The main event loop is implemented with a while block, and with the `Clock` instance, we can control the frame rate and calculate the elapsed time between frames. This value will

be multiplied by the speed of movement, so the circle will move at the same speed if the FPS value changes.

With `pygame.event.get()`, we retrieve the event queue, and if a `QUIT` event occurs, the window is closed and the application finishes its execution.

The `pygame.key.get_pressed()` returns a list with the pressed keys, and with the key constants, we can check whether the arrow keys are pressed. If so, the circle's position is updated and it is drawn on the new coordinates.

Finally, `pygame.display.flip()` updates the screen's surface.



The `Chapter5_03.py` script contains the full code of this example.

Tip

The Pygame documentation

Since Pygame is divided into several modules, each one with various functions, classes, and constants, the official documentation is a useful reference.

We are using some functions from the `key` module; you can find further information about it at https://www.pygame.org/docs/ref/key.html. The same applies for the `display` and `time` modules.

Pygame integration

Let's see how it is possible to implement the same functionality with Pygame. The first step is to replace the `OpenGL.GLUT` import with the ones we used in our previous example:

```
import sys
import math

import pygame
from pygame.locals import *

from OpenGL.GL import *
from OpenGL.GLU import *
```

The `title` string is now a regular string, and the `FPS` attribute can be added as well:

```
class App(object):
    def __init__(self, width=800, height=600):
        self.title = 'OpenGL demo'
        self.fps = 60
        self.width = width
        self.height = height
        # ...
```

We remove the GLUT calls from our `start()` method, and they are replaced by the Pygame initialization. Apart from `DOUBLEBUF`, we will add the `OPENGL` flag to create an OpenGL context:

```
    def start(self):
        pygame.init()
        pygame.display.set_mode((self.width, self.height),
                                OPENGL | DOUBLEBUF)
        pygame.display.set_caption(self.title)

        glEnable(GL_CULL_FACE)
        # ...
        glMatrixMode(GL_MODELVIEW)

        clock = pygame.time.Clock()
        while True:
            dt = clock.tick(self.fps)
            self.process_input(dt)
            self.display()
```

The new `process_input()` method updates the scene and the `instance` attributes by retrieving the events from the event queue and processing the pressed keys.

If a `QUIT` event occurs or the *Esc* key is pressed, the `Pygame` program is executed. Otherwise, the camera position is updated with the distance and angle of rotation, controlled by the arrow keys:

```
    def process_input(self, dt):
        for event in pygame.event.get():
            if event.type == QUIT:
                self.quit()
```

```
            if event.type == KEYDOWN:
                if event.key == K_ESCAPE:
                    self.quit()
                if event.key == K_F1:
                    self.light.switch_color()

        pressed = pygame.key.get_pressed()
        if pressed[K_UP]:
            self.distance -= 0.01 * dt
        if pressed[K_DOWN]:
            self.distance += 0.01 * dt
        if pressed[K_LEFT]:
            self.angle -= 0.005 * dt
        if pressed[K_RIGHT]:
            self.angle += 0.005 * dt

        self.distance = max(10, min(self.distance, 20))
        self.angle %= math.pi * 2
```

The `glutSwapBuffers()` is replaced by `pygame.display.flip()`, and the new `quit()` method quits Pygame and exits Python gracefully:

```
    def display(self):
        # ...
        self.light.render()
        self.sphere1.render()
        self.sphere2.render()
        pygame.display.flip()

    def quit(self):
        pygame.quit()
        sys.exit()
```

Another consequence of removing GLUT is that we cannot use `glutSolidSphere` to render our spheres.

Fortunately, we can substitute it with the `gluSphere` GLU function. The only difference is that we need to create a `GLUquadratic` object first, and then call the function with this argument and the usual radius, number of slices, and number of stacks into which the sphere is divided:

```
class Sphere(object):
    slices = 40
    stacks = 40

    def __init__(self, radius, position, color):
        self.radius = radius
        self.position = position
        self.color = color
        self.quadratic = gluNewQuadric()

    def render(self):
        glPushMatrix()
        glTranslatef(*self.position)
        glMaterialfv(GL_FRONT, GL_DIFFUSE, self.color)
        gluSphere(self.quadratic, self.radius,
```

```
                Sphere.slices, Sphere.stacks)
    glPopMatrix()
```

With these changes, the GLUT API is now completely replaced by Pygame. Check out the `chapter5_04.py` script to see the complete implementation.

Tip

OpenGL and SDL

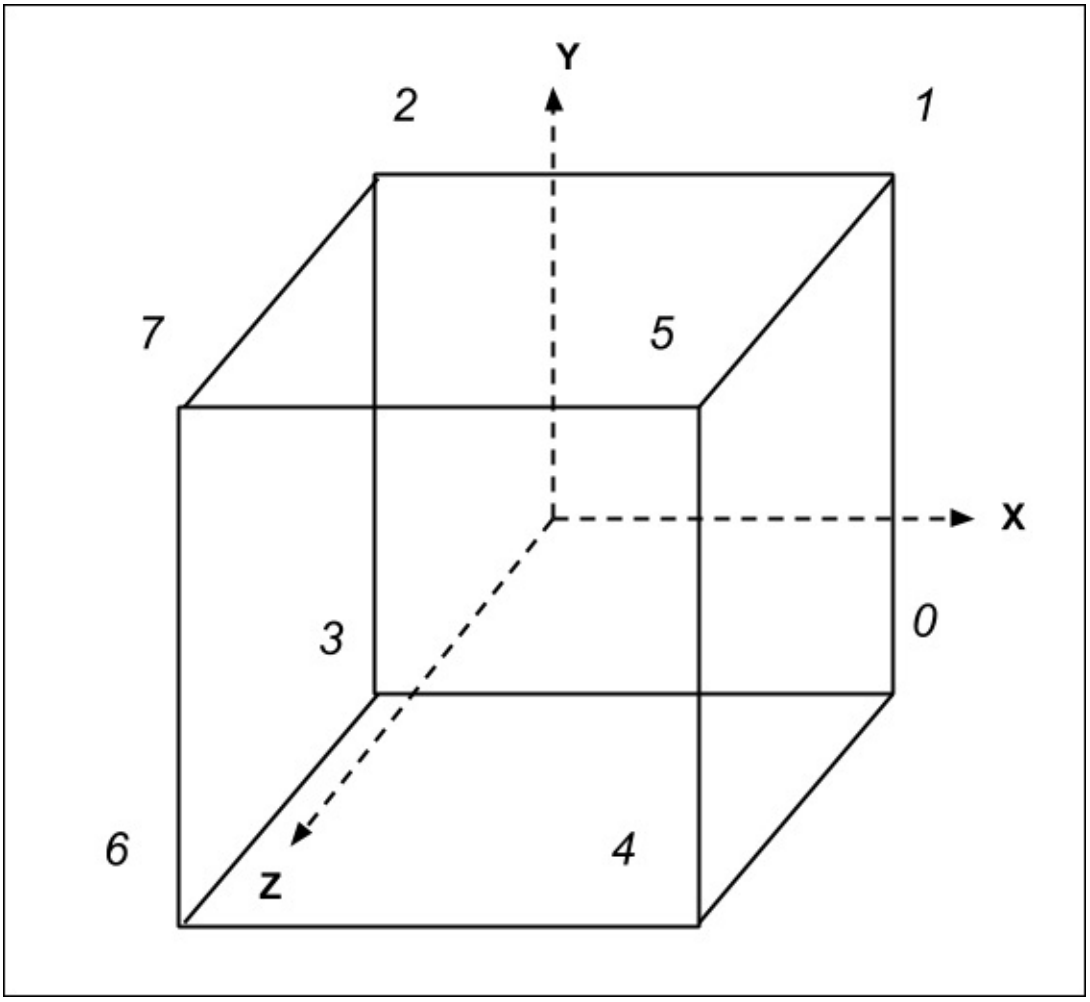
By including Pygame, we replace the GLUT API with **Simple DirectMedia Layer** (**SDL**), which is the library that Pygame is built over. Like `freeglut`, it is another cross-platform alternative to GLUT.

Drawing with OpenGL

Until now, we have always rendered our objects with a utility routine, but most OpenGL applications require the use of some drawing primitives.

The Cube class

We will define a new class to render cubes, and we will use the following representation to better understand the vertices' positions. From 0 to 7, the vertices are enumerated and represented in a 3D space.



The sides of a cube can now be represented as tuples: the back face is `(0, 1, 2, 3)`, the right face is `(4, 5, 1, 0)`, and so on.

Note that we arrange the vertices in counterclockwise order. As we will learn later, this will help us enable an optimization called **face culling**, which consists of drawing only the visible faces of a polygon:

```
class Cube(object):
    sides = ((0,1,2,3), (3,2,7,6), (6,7,5,4),
             (4,5,1,0), (1,5,7,2), (4,0,3,6))
```

The `__init__` method will store the values of the `position` and `color`, as well as the vertex coordinates with respect to the center position of the cube:

```
    def __init__(self, position, size, color):
        self.position = position
        self.color = color
        x, y, z = map(lambda i: i/2, size)
```

```
        self.vertices = (
            (x, -y, -z), (x,  y, -z),
            (-x, y, -z), (-x, -y, -z),
            (x, -y, z), (x,  y, z),
            (-x, -y, z), (-x, y, z))
```

The `render()` method pushes a new matrix, transforms it according to its current position, and calls `glVertex3fv()` for each vertex of the six faces of cube.

The `glVertex3fv` takes a list of three float values that specify the vertex position. This function is executed between the `glBegin()` and `glEnd()` calls. They delimit the vertices that define a primitive. The `GL_QUADS` mode treats each group of four vertices as an independent quadrilateral.

The last statement pops the current matrix from the matrix stack:

```
    def render(self):
        glPushMatrix()
        glTranslatef(*self.position)
        glBegin(GL_QUADS)
        glMaterialfv(GL_FRONT, GL_DIFFUSE, self.color)
        for side in Cube.sides:
            for v in side:
                glVertex3fv(self.vertices[v])
        glEnd()
        glPopMatrix()
```

Enabling face culling

Even though a cube has six faces, we can see a maximum of only three faces at once, and only two or one from certain angles. Therefore, if we discard the faces that are not going to be visible, we can avoid rendering at least 50 percent of the faces of our cubes.

By enabling face culling, OpenGL checks which faces are facing the viewer and discards the faces that are facing backwards. The only requirement is to draw the faces of the cube in the counterclockwise order of the vertices, which is the default front face in OpenGL. The implementation part is easy; we add the following line to our `glEnable` calls:

```
# ...
glEnable(GL_LIGHTING)
glEnable(GL_CULL_FACE)
# ...
```

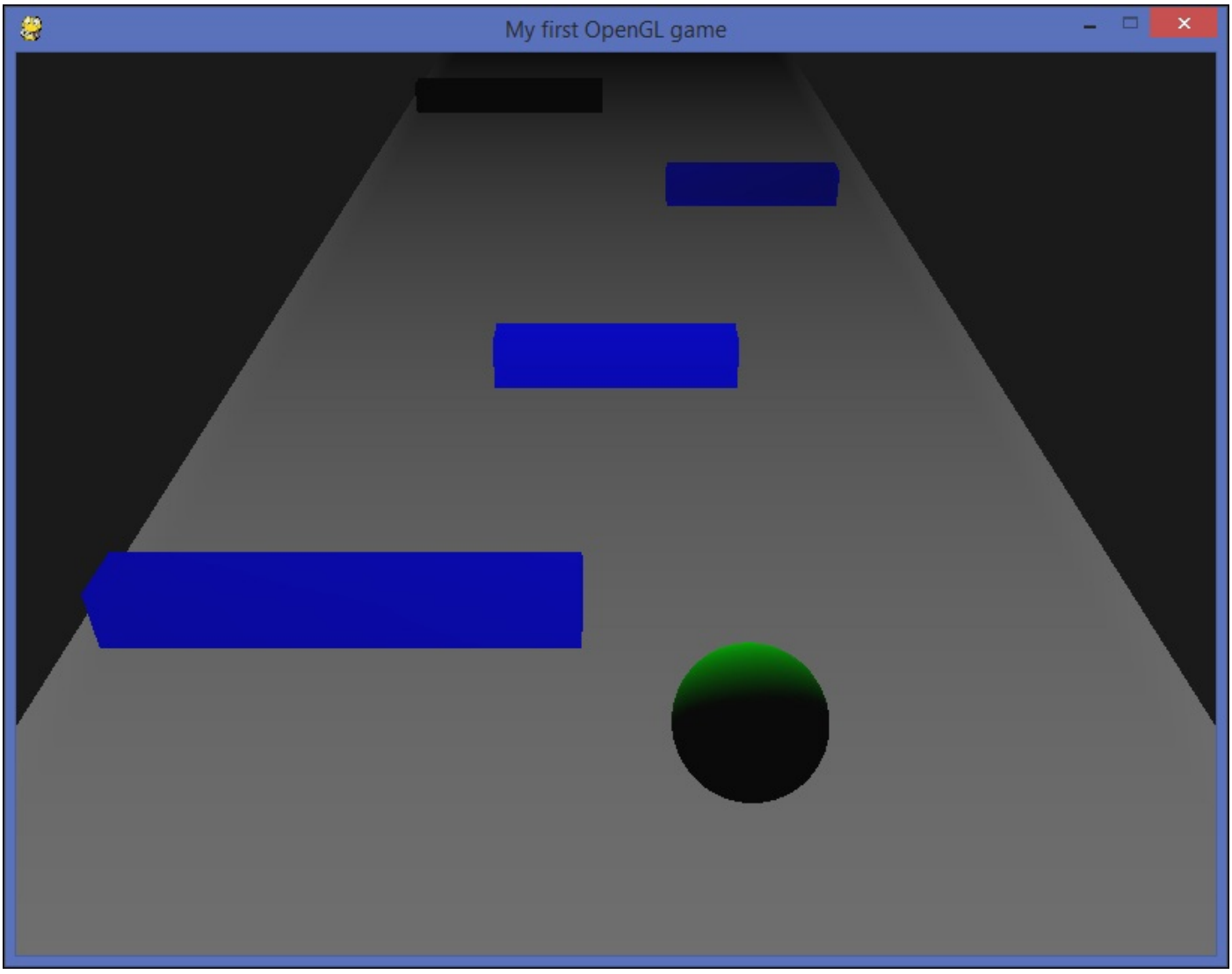
In our next application, we will add some cubes and enable face culling to see this optimization in practice.

Basic collision detection game

With all of these ingredients, you are now able to write a simple game that detects simple collisions between shapes.

The scene consists of an infinite lane. Blocks appear randomly at the end of the lane and move towards the player, represented as the sphere in the following screenshot. He or she must avoid hitting the blocks by moving the sphere from right to left in the horizontal axis.

The game is over when the player's character collides with one of the blocks.



This gameplay is direct and uncomplicated, and it will allow us to develop a 3D game without worrying too much about more complicated physics calculations.

Since we are going to reuse the `Light`, `Cube`, and `Sphere` classes, we need to define a new class only to represent our game blocks:

```
class Block(Cube):
    color = (0, 0, 1, 1)
    speed = 0.01
```

```
    def __init__(self, position, size):
        super().__init__(position, (size, 1, 1), Block.color)
        self.size = size

    def update(self, dt):
        x, y, z = self.position
        z += Block.speed * dt
        self.position = x, y, z
```

Its `update()` method simply moves the block towards the player by updating its `z` coordinate with uniform speed.

Our `App` class sets the initial values of the attributes that we will need during the execution of our game, and creates the `Light` and the `game` object instances as in our previous examples:

```
class App(object):
    def __init__(self, width=800, height=600):
        # ...
        self.game_over = False
        self.random_dt = 0
        self.blocks = []
        self.light = Light(GL_LIGHT0, (0, 15, -25, 1))
        self.player = Sphere(1, position=(0, 0, 0),
                             color=(0, 1, 0, 1))
        self.ground = Cube(position=(0, -1, -20),
                           size=(16, 1, 60),
                           color=(1, 1, 1, 1))
```

The `start()` method has small variations, only adding `glEnable(GL_CULL_FACE)`, as we mentioned previously:

```
    def start(self):
        pygame.init()
        # ...
        glMatrixMode(GL_MODELVIEW)
        glEnable(GL_CULL_FACE)
        self.main_loop()
```

The `main_loop()` method is now a separate method and includes the random generation of blocks, collision detection, as well as the updating of the positions of the blocks:

```
    def main_loop(self):
        clock = pygame.time.Clock()
        while True:
            for event in pygame.event.get():
                if event.type == QUIT:
                    pygame.quit()
                    sys.exit()
            if not self.game_over:
                self.display()
                dt = clock.tick(self.fps)
                for block in self.blocks:
                    block.update(dt)
                self.clear_past_blocks()
                self.add_random_block(dt)
```

```
            self.check_collisions()
            self.process_input(dt)
```

We will implement collision detection by comparing the boundaries of the closest blocks with the extremes of the sphere. Since the sphere's width is smaller than the block size, if one of these extremes is between the right and left boundaries of a block, it will be considered as a collision:

```
    def check_collisions(self):
        blocks = filter(lambda x: 0 < x.position[2] < 1,
                        self.blocks)
        x = self.player.position[0]
        r = self.player.radius
        for block in blocks:
            x1 = block.position[0]
            s = block.size / 2
            if x1-s < x-r < x1+s or x1-s < x+r < x1+s:
                self.game_over = True
                print("Game over!")
```

To prevent the spawning of too many blocks, we defined a counter called **random_dt**. It accumulates the elapsed time in milliseconds between frames, and it will try to spawn a new block only if the sum is greater than `800` milliseconds:

```
    def add_random_block(self, dt):
        self.random_dt += dt
        if self.random_dt >= 800:
            r = random.random()
            if r < 0.1:
                self.random_dt = 0
                self.generate_block(r)

    def generate_block(self, r):
        size = 7 if r < 0.03 else 5
        offset = random.choice([-4, 0, 4])
        self.blocks.append(Block((offset, 0, -40), size))
```

If the generated random number is lower than `0.1`, a new block is added to the block list and the `random_dt` counter is reset to `0`. In this way, the minimum elapsed time between two blocks can be `0.8` seconds, giving enough time to leave a tolerable distance from one block to another.

Another operation that the main loop performs is removing the blocks that are located behind the cameras' viewing area, avoiding the creation of too many `Block` instances:

```
    def clear_past_blocks(self):
        blocks = filter(lambda x: x.position[2] > 5,
                        self.blocks)
        for block in blocks:
            self.blocks.remove(block)
            del block
```

The code for displaying the game objects stays as succinct as usual, thanks to the transfer of the drawing primitives to the respective `render()` methods:

```
    def display(self):
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glLoadIdentity()
        gluLookAt(0, 10, 10,
                  0, 0, -5,
                  0, 1, 0)
        self.light.render()
        for block in self.blocks:
            block.render()
        self.player.render()
        self.ground.render()
        pygame.display.flip()
```

To finish our game, we will modify the input handling of our program. This change is straightforward, since we only need to update the `x` component of the character's position and trim it so that it cannot move out of the lane:

```
    def process_input(self, dt):
        pressed = pygame.key.get_pressed()
        x, y, z = self.player.position
        if pressed[K_LEFT]:
            x -= 0.01 * dt
        if pressed[K_RIGHT]:
            x += 0.01 * dt
        x = max(min(x, 7), -7)
        self.player.position = (x, y, z)
```

In the `chapter5_05.py` script, you can find the full implementation of the game. Run it and feel free to modify and improve it! You can add pickup items and keep track of the score, or give the player a number of lives before the game is over.

Summary

In this chapter, you learned how it is possible to work with Python and OpenGL, and with basic knowledge about OpenGL APIs, we were able to develop a simple 3D game.

We saw two cross-platform alternatives for creating an OpenGL context: GLUT and Pygame. You can decide which one better suits your 3D games, depending on the trade-offs of each option. Keep this in mind: an advantage of using both is that you may adapt existing examples from one library to the other!

With these foundations of 3D covered, in the next chapter, we will see how to develop a 3D platformer based on these technologies.