3D Programming is an incredibly wide and complex field. This chapter explores some topics that are the absolute minimum requirement to write a simple 3D game:

- We’ll revisit our friend the vector and attach one more coordinate.
- Lighting is a vital part of any 3D game. We’ll look at how to perform simple lighting with OpenGL ES.
- Defining objects programmatically is cumbersome. We’ll look at a simple 3D file format so that we can load and render 3D models created with 3D modeling software.
- In Chapter 8, we discussed object representation and collision detection. We’ll look at how to do the same in 3D.
- We’ll also briefly revisit some of the physics concepts that we explored in Chapter 8—this time in a 3D context.

Let’s start with 3D vectors.

Before We Begin

As always, we’ll create a couple of simple example programs in this chapter. To do that, we just create a new project and copy over all of the source code of the framework we’ve developed so far.

As in previous chapters, we’ll have a single test starter activity that presents us with the tests in the form of a list. We’ll call it GLAdvancedStarter and make it our default activity. Simply copy over the GL3DBasicsStarter, and replace the class names of the tests. We also need to add each of the test activities to the manifest with a proper <activity> element.

Each of the tests will extend GLGame as usual; the actual code will be implemented as a GLScreen instance that we’ll hook up with the GLGame instance. As in Chapter 10, to conserve space, we’ll present only the relevant portions of the GLScreen implementations. All of the tests and the
 starter activity reside within the package com.badlogic.androidgames.gladvanced. Some of the classes will be part of our framework and will go into the respective framework packages.

**Vectors in 3D**

In Chapter 8, we discussed vectors and their interpretation in 2D. As you might have guessed, all of the things we discussed there still hold in 3D space as well. All we need to do is to add one more coordinate to our vector, namely the z coordinate.

The operations we looked at with vectors in 2D can be easily transferred to 3D space. We specify vectors in 3D with a statement like the following:

\[ v = (x, y, z) \]

Addition in 3D is carried out as follows:

\[ c = a + b = (a_x, a_y, b_z) + (b_x, b_y, b_z) = (a_x + b_x, a_y + b_y, a_z + b_z) \]

Subtraction works exactly the same way:

\[ c = a - b = (a_x, a_y, b_z) - (b_x, b_y, b_z) = (a_x - b_x, a_y - b_y, a_z - b_z) \]

Multiplying a vector by a scalar works like this:

\[ a' = a \times \text{scalar} = (a_x \times \text{scalar}, a_y \times \text{scalar}, a_z \times \text{scalar}) \]

Measuring the length of a vector in 3D is also quite simple; we just add the z coordinate to the Pythagorean equation:

\[ |a| = \sqrt{a_x \times a_x + a_y \times a_y + a_z \times a_z} \]

Based on this, we can also normalize our vectors to unit length again:

\[ a' = \frac{a_x}{|a|}, \frac{a_y}{|a|}, \frac{a_z}{|a|} \]

All of the interpretations of vectors we talked about in Chapter 8 hold in 3D as well:

- Positions are just denoted by a normal vector’s x, y, and z coordinates.
- Velocities and accelerations can also be represented as 3D vectors. Each component then represents a certain quantity of the attribute on one axis, such as meters per second (m/s) in the case of velocity or meters per second per second (m/s²) for acceleration.
- We can represent directions (or axes) as simple 3D unit vectors. We did that in Chapter 8 when we used the rotation facilities of OpenGL ES.
- We can measure distances by subtracting the starting vector from the end vector and measuring the resulting vector’s length.

One more operation that can be rather useful is rotation of a 3D vector around a 3D axis. We used this principle earlier via the OpenGL ES glRotatef() method. However, we can’t use it to rotate one of the vectors that we’ll use to store positions or directions of our game objects, because it only works on vertices that we submit to the GPU. Luckily, there’s a Matrix class in the Android API that allows us to emulate what OpenGL ES does on the GPU. Let’s write a Vector3 class that implements all of these features. Listing 11-1 shows the code, which we’ll again explain along the way.