Super Jumper worked out rather well with the 2D OpenGL ES rendering engine. Now it’s time to go full 3D. You’ve already worked in a 3D space when you defined your view frustum and the vertices of your sprites. In the latter case, the z-coordinate of each vertex was simply set to zero by default. The difference from 2D rendering really isn’t all that big:

- Vertices not only have x- and y-coordinates, but they also have a z-coordinate.

- Instead of an orthographic projection, a perspective projection is used. Objects further away from the camera will appear smaller.

- Transformations, such as rotations, translations, and scales, have more degrees of freedom in 3D. Instead of just moving the vertices in the x-y plane, they can now be moved around freely on all 3 axes.

- A camera is defined with an arbitrary position and orientation in 3D space.

- The order in which you render the triangles of your objects is now important. Objects further away from the camera must be overlapped by objects that are closer to the camera.

The best thing is that you have already laid the groundwork for all of this in your framework. You just need to adjust a couple classes slightly to go 3D.

Before We Begin

As always, you’ll write a couple of examples in this chapter. For this, you’ll follow the same route as before by having a starter activity showing a list of examples. You’ll reuse the entire framework created over the last couple chapters, including the GLGame, GLScreen, Texture, and Vertices classes.

The starter activity of this chapter is called GL3DBasicsStarter. You can reuse the code of the GLBasicsStarter from Chapter 6; you’ll just change the package name for the
example classes that are going to run to com.badlogic.androidgames.gl3d. You must also add each of the tests to the manifest in the form of <activity> elements again. All the tests will be run in fixed landscape orientation, which will be specified per <activity> element.

Each of the tests is an instance of the GLGame abstract class, and the actual test logic is implemented in the form of a GLScreen contained in the GLGame implementation of the test, as seen in previous chapters. You will only present the relevant portions of the GLScreen to conserve some pages. The naming conventions are again XXXTest and XXXScreen for the GLGame and GLScreen implementation of each test.

**Vertices in 3D**

In Chapter 7, you learned that a vertex has a few attributes:

- Position
- Color (optional)
- Texture coordinates (optional)
- You created a helper class called Vertices, which handles all the dirty details for you. You limited the vertex positions to have only x- and y-coordinates. All you need to do to go 3D is modify the Vertices class so that it supports 3D vertex positions.

**Vertices3: Storing 3D Positions**

Let's write a new class called Vertices3 to handle 3D vertices based on your original Vertices class. Listing 10–1 shows the code.

**Listing 10–1. Vertices3.java, Now with More Coordinates.**

```java
package com.badlogic.androidgames.framework.gl;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.IntBuffer;
import java.nio.ShortBuffer;
import javax.microedition.khronos.opengles.GL10;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class Vertices3 {
    final GLGraphics glGraphics;
    final boolean hasColor;
    final boolean hasTexCoords;
    final int vertexSize;
    final IntBuffer vertices;
```