Chapter 20

Ten Steps to Building a 3D World

In This Chapter

- Creating a virtual 3D world
- Populating your world with shapes
- Translating and rotating 3D objects
- Animating 3D objects
- Adding a light source for more realism

JavaFX has built-in support for realistic 3D modeling. In fact, the JavaFX scene graph is three-dimensional in nature. Most JavaFX programs work in just two dimensions, specifying just x- and y-coordinates. But all you have to do to step into the third dimension is specify z-coordinates to place the nodes of your scene graph in three-dimensional space.

JavaFX includes a rich set of classes that are dedicated to creating and visualizing 3D objects in 3D worlds. You can create three-dimensional shapes, such as cubes and cylinders. You can move the virtual camera around within the 3D space to look at your 3D objects from different angles and different perspectives. And you can even add lighting sources to carefully control the final appearance of your virtual worlds. In short, JavaFX is capable of producing astonishing 3D scenes.

In this chapter, I discuss in ten short steps how to create a relatively simple 3D program that displays the three-dimensional world shown in Figure 20-1. As you can see, this 3D space includes four shapes: a sphere, a cube, a cylinder, and a pyramid. This program also demonstrates several other key aspects of 3D programming: a perspective camera, a Phong material, a light source, and 3D animation.

Put on your Thinking Cap, as this chapter will get pretty technical at times, and many of the concepts presented in this chapter can be confusing, especially if this is your first experience with 3D programming.

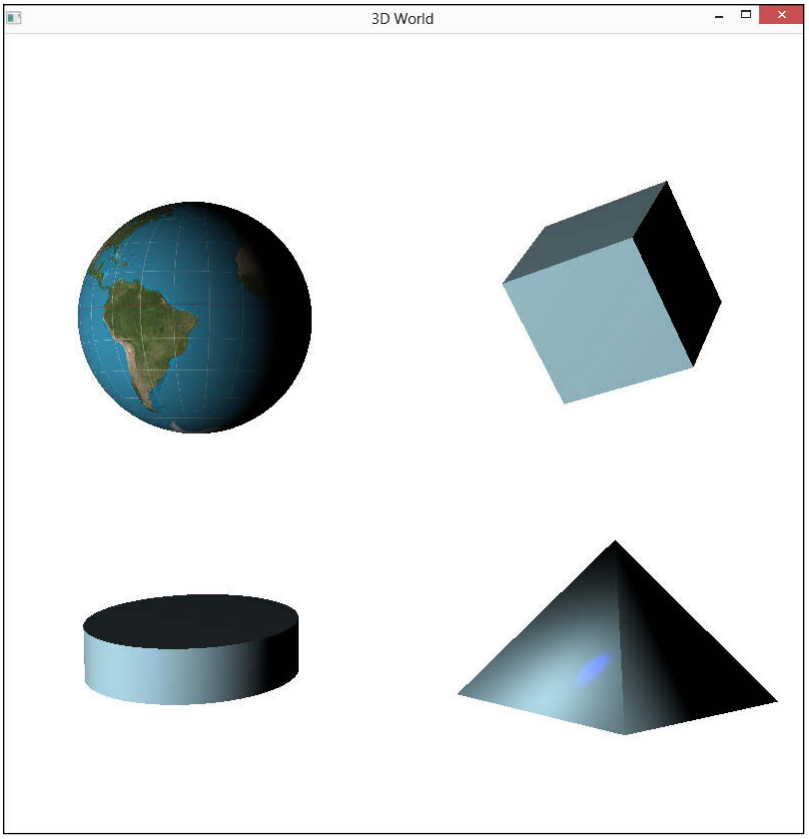

Figure 20-1: A sample 3D program.

Step One: Add a Perspective Camera

The first step in creating a three-dimensional JavaFX application is adding a camera to the scene graph. You do that by creating a `PerspectiveCamera` object, fiddling with its settings, and then calling the scene's `setCamera` method. Here's an example:

```
Group root = new Group();
Scene scene = new Scene(root, 800, 800);

PerspectiveCamera camera = new PerspectiveCamera(true);
camera.setTranslateZ(-1000);
camera.setNearClip(0.1);
camera.setFarClip(2000.0);
camera.setFieldOfView(35);
scene.setCamera(camera);
```

Getting used to JavaFX 3D coordinates

In two-dimensional scenes, coordinates are measured in the x-axis (horizontal) and the y-axis (vertical). Two-dimensional points are referenced by a pair of x- and y-coordinates. For example, the point (100, 200) represents the point 100 units to the right of zero on the right axis and 200 units below zero on the y-axis.

When you work in three dimensions, a third axis — called the z-axis — is added. The z-axis is perpendicular to both the x- and y-axis. Imagine a line extending from your eyes into the computer monitor and through the monitor to the wall behind it. That's the z-axis.

Three-dimensional points are represented by three coordinates: x, y, and z. Thus, (100, 200, 300) represents a point 100 units to the right of zero on the x-axis, 200 units below 0 on the y-axis, and 300 units toward you from zero on the z-axis.

A crucial difference between 2D and 3D coordinates is that in a 2D scene, the origin (point 0,0) is located at the top-left corner of the screen. In a 3D scene, the origin (point 0,0,0) is located right at the middle of the space visualized by the 3D scene.

Also, keep in mind that in the JavaFX 3D world, z-coordinates decrease as they move toward you and increase as they move away from you. X- and y-coordinates behave exactly as they do in 2D: X-coordinates increase as they move to the right, and y-coordinates increase as they move down.

This example begins by creating a scene in the same manner as you'd create a scene for a 2D JavaFX application. Then, the example creates an instance of the `PerspectiveCamera` class and adjusts three properties of this class.

A *perspective camera* is an essential element in any 3D scene. A perspective camera represents the virtual camera that is used to render the three-dimensional world onto a flat surface. The camera is actually a part of the scene graph and has a position indicated by a set of x-, y-, z-coordinates, just like any other object in the 3D scene. The default position for the camera (and any other object you add to the scene) is the origin point (0,0,0). So, the first thing you want to do after you add a camera is move it to a location from which it can get a good view of the objects you'll be adding to the scene. In this example, I call the `setTranslateZ` method to back the camera away from the scene 1,000 units.

Next, I set the near and far clipping distances. These values mark the range within which the camera will render objects. The near clipping distance is typically set to a very small value (in this case, 0.1) and the far clipping distance to a value large enough to contain the objects you want to appear in the scene.

After setting the clipping distances, I adjusted the field of view of the camera. The field of view is given as an angle and is analogous to using a wide-angle or a telephoto lens in a real camera. The default value is 30, but for this application, I found that 35 gives a better look at the scene.

Finally, I designated the camera as the scene's active camera by calling the scene's `setCamera` method.

At this stage, you have created a three-dimensional world. However, that world is a pretty lonely place, as it has no inhabitants. In the next step, you add a basic 3D shape to the world.

Step Two: Add a Cylinder

In this step, you add a basic 3D object to your world. JavaFX provides three basic shapes you can add: cylinders, boxes, and spheres. Start by adding a cylinder:

```
Cylinder cylinder = new Cylinder(100,50);
root.getChildren().add(cylinder);
```

The `Cylinder` class constructor accepts two arguments: the radius of the cylinder and its height. This example creates a cylinder roughly the shape of a hockey puck, four times as wide as it is tall; then, it adds the cylinder to the scene's root node.

At this point, the cylinder exists in the world, but is not visible. Based on what you know of 2D shapes, you may be tempted to make it visible by adding a fill color (`setFill`) or a stroke color (`setStroke`). But that's not how 3D objects work. In the next step, you discover how to apply a material to the surface of the cylinder so that it will be visible in the scene.

Step Three: Create a Material

Rendering the faces of a 3D object is much more complicated than rendering flat, two-dimensional objects. For a 2D object, you just apply a `Paint` object via the `setFill` method. The paint can be a simple color, a gradient color, or an image.

For 3D objects, you don't apply paint. Instead, you apply a special object called a *Phong material*, represented by the `PhongMaterial` class. A Phong material (named after Bui Tuong Phong, a pioneering computer graphics expert in the 1970's) provides the means by which the faces of a 3D object are realistically rendered.

The following code creates a simple Phong material based on two shades of blue and then applies the material to the cylinder:

```
PhongMaterial blueStuff = new PhongMaterial();
blueStuff.setDiffuseColor(Color.LIGHTBLUE);
blueStuff.setSpecularColor(Color.BLUE);
cylinder.setMaterial(blueStuff);
```

After the Phong material has been applied to the cylinder, the cylinder will be visible within the scene, as shown in Figure 20-2.

Figure 20-2: The cylinder with a Phong material.

Step Four: Translate the Cylinder

You undoubtedly noticed that the cylinder in Figure 20-2 doesn't look very three dimensional. That's because you're looking at it edge-on: The camera is pointing straight at the intersection of the x- and y-axes, and the cylinder is centered on that very spot.

To gain some perspective on the cylinder, you can move it to a different location in 3D space by translating the x-, y-, and z-coordinates. For example:

```
cylinder.setTranslateX(-200);
cylinder.setTranslateY(200);
cylinder.setTranslateZ(200);
```

Here, the cylinder is moved 200 units to the left, 200 units down, and 200 units away from the camera. The resulting view looks more like a cylinder, as you can see in Figure 20-3.

Figure 20-3: The translated cylinder.

In Figure 20-3, it looks as if the cylinder has been rotated forward so that you can see a bit of the top surface. This isn't the case, however. What has actually happened is that you're no longer looking at the cylinder edge-on. Instead, because the cylinder is below the camera, you're looking down on it. Thus, you can see a bit of the top face. You're also looking at it from the side, which explains why it appears just a tad tilted.

Step Five: Add a Box

In this step, I add a second object to the 3D world: In this case, a box, represented by the `Box` class. Here's the code:

```
Box box = new Box(100, 100, 100);
box.setMaterial(blueStuff);
box.setTranslateX(150);
box.setTranslateY(-100);
box.setTranslateZ(-100);
root.getChildren().add(box);
```

The `Box` constructor accepts three arguments representing the width, height, and depth of the box. In this example, all three are set to 100. Thus, the box will be drawn as a cube with each side measuring 100 units.

The box is given the same material as the cylinder; then, it is translated on all three axes so that you can have a perspective view of the box. Figure 20-4 shows how the box appears when rendered. As you can see, the left and bottom faces of the box are visible because you translated the position of the box up and to the right so that the camera can gain some perspective.

Figure 20-4: The box.

Step Six: Rotate the Box

In this step, I rotate the box to create an even more interesting perspective view. There are two ways to rotate a 3D object. The simplest is to call the object's `setRotate` method and supply a rotation angle:

```
box.setRotate(25);
```

By default, this will rotate the object on its z-axis. If this is difficult to visualize, imagine skewering the object with a long stick that is parallel to the z-axis. Then, spin the object on the skewer.

If you want to rotate the object along a different axis, first call the `setRotationAxis`. For example, to spin the object on its x-axis, use this sequence:

```
box.setRotationAxis(Rotate.X_AXIS);
box.setRotate(25);
```

Imagine running the skewer through the box with the skewer parallel to the x-axis and then spinning the box 25 degrees.

The only problem with using the `setRotate` method to rotate a 3D object is that it works only on one axis at a time. For example, suppose you want to rotate the box 25 degrees on both the z- and the x-axis. The following code will *not* accomplish this:

```
box.setRotationAxis(Rotate.X_AXIS);
box.setRotate(25);
box.setRotationAxis(Rotate.Z_AXIS);
box.setRotate(25);
```

When the `setRotate` method is called the second time to rotate the box on the z-axis, the x-axis rotation is reset.

To rotate on more than one axis, you must use the `Rotate` class instead. You create a separate `Rotate` instance for each axis you want to rotate the object on and then add all the `Rotate` instances to the object's `Transforms` collection via the `getTransforms().addAll` method, like this:

```
Rotate rxBox = new Rotate(0, 0, 0, 0, Rotate.X_AXIS);
Rotate ryBox = new Rotate(0, 0, 0, 0, Rotate.Y_AXIS);
Rotate rzBox = new Rotate(0, 0, 0, 0, Rotate.Z_AXIS);
rxBox.setAngle(30);
ryBox.setAngle(50);
rzBox.setAngle(30);
box.getTransforms().addAll(rxBox, ryBox, rzBox);
```

The `Rotate` constructor accepts four parameters. The first three are the x-, y-, and z-coordinates of the point within the object through which the rotation axis will pass. Typically, you specify zeros for these parameters to rotate the object around its center point. The fourth parameter specifies the rotation axis.

Figure 20-5 shows how the box appears after it's been rotated.

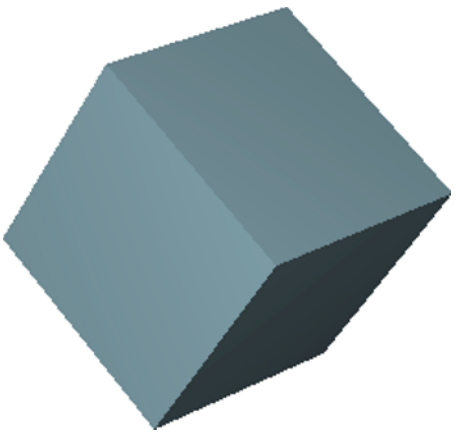

Figure 20-5: The box after it's been rotated.

Step Seven: Add a Sphere

In this step, I add a sphere, represented by the `Sphere` class. The `Sphere` constructor accepts just a single parameter, which specifies the radius of the sphere. For example, these lines create a sphere whose radius is 100, and then translates it to move it off the center point of your virtual world:

```
Sphere sphere = new Sphere(100);
sphere.setTranslateX(-180);
sphere.setTranslateY(-100);
sphere.setTranslateZ(100);
root.getChildren().add(sphere);
```

Rather than apply the same blue Phong material to the sphere, I decided to do something more interesting: I apply a Phong material constructed from an image of a cylindrical projection of the earth using this code:

```
Image earthImage = new Image("file:earth.jpg");
PhongMaterial earthPhong = new PhongMaterial();
earthPhong.setDiffuseMap(earthImage);
sphere.setMaterial(earthPhong);
```

Figure 20-6 shows the resulting sphere.

Figure 20-6: A sphere with a cylindrical projection of the earth applied as the Phong material.

You can wrap any image around a sphere (or any other 3D object, for that matter) using this technique. I obtained the image I used for this program from Wikipedia. Just search for *Behrmann Projection* and then download the file. (I used Windows Paint to crop the edges of the image a bit because the image available on Wikipedia has a small border around the edges.)

Step Eight: Add a Mesh Object

The three objects you've added to your virtual world so far have been created using the three built-in 3D shape classes that come with JavaFX: `Cylinder`, `Box`, and `Sphere`. For more complex objects, you must use the `TriangleMesh` class to create the object based on a connected series of triangles.

In this step, I create one of the simplest of all mesh objects: the four-sided pyramid pictured in Figure 20-7. Visually, a four-sided pyramid has a total of five *faces:* the four triangular side faces and the square base. But in a JavaFX triangle mesh, squares are not allowed, only triangles. So the pyramid actually consists of six faces: the four triangular side faces and two adjacent triangles that make up the face.

This section presents probably the most conceptually difficult information in this entire book. If you haven't studied meshes in a Computer Graphics class, be prepared to read through the following paragraphs several times before it starts to make sense. If it still doesn't make sense, grab a latte, pull out a sheet of graph paper, and start doodling. Drawing the pyramid with your own hand will help your understanding. I recommend you use a pencil.

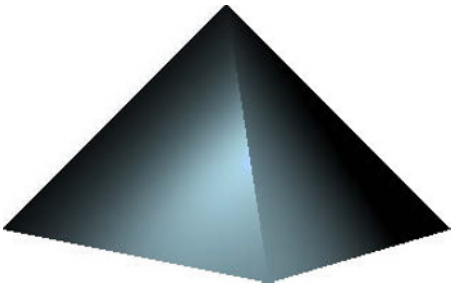

Figure 20-7: A square pyramid.

To get the pyramid started, call the `TriangleMesh` constructor like this:

```
TriangleMesh pyramidMesh = new TriangleMesh();
```

To complete the pyramid, you need to populate three collections that define the geometry of the mesh. These collections hold the points, the faces, and the texture coordinates that define the shape.

I start with the texture coordinate collection, because you can pretty much ignore it for this simple pyramid. Texture coordinates are useful when you're using a material that contains an image that should be stretched in a specific way over the framework of the mesh. They allow you to associate a specific x-, y-coordinate in the image with each corner of each face.

Unfortunately, you can't simply leave out the texture coordinates even if you don't need them, so you must load at least one coordinate. Do that with this line of code:

```
pyramidMesh.getTexCoords().addAll(0,0);
```

Now I move on to the other two collections. The next is a collection of the *vertices* (that is, corners) that defines the shape. Your square pyramid has five vertices, which you can envision as the top, the front corner (the point nearest you), the left corner, the back corner, and the right corner. These vertices are numbered 0, 1, 2, 3, and 4.

Given the height *h* and the length *s* of each side of the pyramid, you can calculate the x-, y-, and z-coordinates for each vertex using the following formulas:

Vertex	***Corner***	***X***	***Y***	***Z***
0	Top	0	0	0
1	Front	0	*h*	$-s/2$
2	Left	$-s/2$	*h*	0
3	Back	$s/2$	*h*	0
4	Right	0	*h*	$s/2$

With all that as background, here's the code to create the Points collection:

```
float h = 150;                    // Height
float s = 300;                    // Side
pyramidMesh.getPoints().addAll(
        0,    0,    0,            // Point 0 - Top
        0,    h,    -s/2,         // Point 1 - Front
        -s/2, h,    0,            // Point 2 - Left
        s/2,  h,    0,            // Point 3 - Back
        0,    h,    s/2           // Point 4 - Right
    );
```

The final collection defines the faces. The faces are defined by specifying the index of each vertex that makes up each face. For example, the front left face is a triangle whose three vertices are the top, the front, and the left. The indexes for these three vertices are 0, 2, and 1.

There are a total of six triangles in the pyramid, and their faces are defined by the following points:

Face	*Point 1*	*Point 2*	*Point 3*
Front left	0	2	1
Front right	0	1	3
Back right	0	3	4
Back left	0	4	2
Bottom rear	4	1	2
Bottom front	4	3	2

Although it may not be evident from this table, the order in which the faces appear is critical to the success of the mesh. In general, the faces are listed in a counter-clockwise and downward order. Thus, the four side faces wrap around the pyramid in counter-clockwise order. They're followed by the two bottom faces.

Each face in the Faces collection is represented by three pairs of numbers, each of which represents the index of one of the vertices of the triangle and the index of the corresponding texture coordinate. Because you have only one item in the Texture Coordinate collection, the second number in each pair will always be zero. Thus, the sequence 0, 0, 2, 0, 1, 0 defines the front left face: The vertex indexes are 0, 2, and 1, and the texture coordinate indexes are all 0.

Here's the code to load the Faces collection:

```
pyramidMesh.getFaces().addAll(
        0,0,  2,0,  1,0,          // Front left face
        0,0,  1,0,  3,0,          // Front right face
        0,0,  3,0,  4,0,          // Back right face
        0,0,  4,0,  2,0,          // Back left face
        4,0,  1,0,  2,0,          // Bottom rear face
        4,0,  3,0,  1,0           // Bottom front face
    );
```

After the three collections of the mesh are ready, the rest of the code fleshes out the pyramid by adding a Phong material, translates the pyramid to get it off the center of the scene, and adds the pyramid to the root:

```
MeshView pyramid = new MeshView(pyramidMesh);
pyramid.setDrawMode(DrawMode.FILL);
pyramid.setMaterial(blueStuff);
pyramid.setTranslateX(200);
pyramid.setTranslateY(100);
pyramid.setTranslateZ(200);
root.getChildren().add(pyramid);
```

Step Nine: Animate the Objects

Whew! Your 3D virtual world now has four objects: a sphere that looks like the earth, a cubic box, a cylinder that looks like a hockey puck, and a pyramid.

In this step, I add an animation to all four objects to get them spinning. Each object gets a simple `RotationTransition` animation. First, the box:

```
RotateTransition rt1 = new RotateTransition();
rt1.setNode(box);
rt1.setDuration(Duration.millis(3000));
rt1.setAxis(Rotate.Z_AXIS);
rt1.setByAngle(360);
rt1.setCycleCount(Animation.INDEFINITE);
rt1.setInterpolator(Interpolator.LINEAR);
rt1.play();
```

After the `play` method is called, the box starts spinning, making one complete turn around its z-axis every three seconds.

The other three animations are similar; the only differences are the node to be rotated, the axis of rotation, and the speed. For the cylinder, the rotation is on the x-axis. The sphere rotates around the y-axis, creating the impression that the world is revolving. For the sphere, the speed is set to one revolution every 10 seconds. And finally, the pyramid rotates on the y-axis.

Step Ten: Add a Light Source

The last step into this foray into the world of 3D programming is to add a light source. The light source will change the whole look of the scene, as shown in Figure 20-8.

To add the light source, I use the following code:

```
PointLight light = new PointLight(Color.WHITE);
light.setTranslateX(-1000);
light.setTranslateY(100);
light.setTranslateZ(-1000);
root.getChildren().add(light);
```

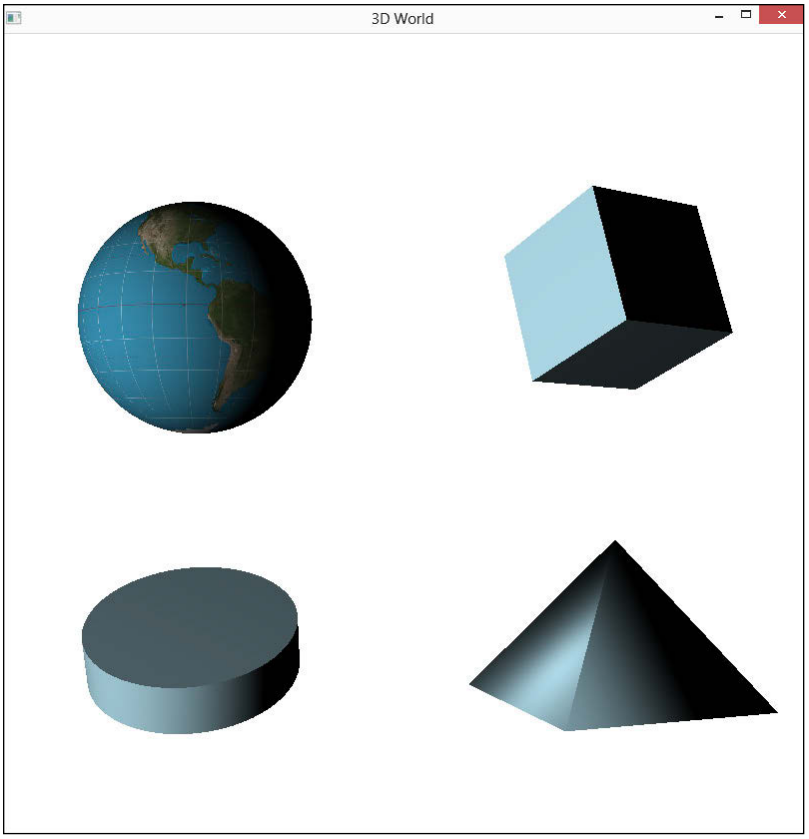

Figure 20-8: Your 3D world with a light source.

The `PointLight` class defines a light source that originates from a specific point in the scene and projects light of the given color (in this case, good ol' white). To create the lighting effect I want, I relocate the light by translating its coordinates 1,000 to the left, 100 down, and 1,000 units toward the user. The result casts nice shadows on the backsides of the spinning objects.

Putting It All Together: The Complete 3D World Program

Now that you've seen all the pieces, Listing 20-1 shows the entire program. Comments within the program make it clear which sections of the program correspond to the steps outlined in this chapter.

With this as a starting point, you're well on your way to creating virtual 3D worlds of your own. Have fun!

Listing 20-1: The 3D World Program

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.shape.*;
import javafx.scene.paint.*;
import javafx.animation.*;
import javafx.util.*;
import javafx.scene.transform.*;
import javafx.scene.image.*;

public class ThreeDWorld extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }

    @Override public void start(Stage primaryStage)
    {
        Group root = new Group();
        Scene scene = new Scene(root, 800, 800);


        // STEP ONE:  ADD A CAMERA

        PerspectiveCamera camera = new PerspectiveCamera(true);
        camera.setTranslateZ(-1000);
        camera.setNearClip(0.1);
        camera.setFarClip(2000.0);
        camera.setFieldOfView(35);
        scene.setCamera(camera);


        // STEP TWO: ADD A CYLINDER

        Cylinder cylinder = new Cylinder(100,50);
        root.getChildren().add(cylinder);


        // STEP THREE: CREATE A MATERIAL

        PhongMaterial blueStuff = new PhongMaterial();
        blueStuff.setDiffuseColor(Color.LIGHTBLUE);
        blueStuff.setSpecularColor(Color.BLUE);
        cylinder.setMaterial(blueStuff);
```

(continued)

Listing 20-1 ***(continued)***

```
        // STEP FOUR: TRANSLATE THE CYLINDER

        cylinder.setTranslateX(-200);
        cylinder.setTranslateY(200);
        cylinder.setTranslateZ(200);


        // STEP FIVE: ADD A BOX

        Box box = new Box(100, 100, 100);
        box.setMaterial(blueStuff);
        box.setTranslateX(150);
        box.setTranslateY(-100);
        box.setTranslateZ(-100);
        root.getChildren().add(box);


        // STEP SIX: ROTATE THE BOX

        Rotate rxBox = new Rotate(0, 0, 0, 0, Rotate.X_AXIS);
        Rotate ryBox = new Rotate(0, 0, 0, 0, Rotate.Y_AXIS);
        Rotate rzBox = new Rotate(0, 0, 0, 0, Rotate.Z_AXIS);
        rxBox.setAngle(30);
        ryBox.setAngle(50);
        rzBox.setAngle(30);
        box.getTransforms().addAll(rxBox, ryBox, rzBox);


        // STEP SEVEN: ADD A SPHERE

        Sphere sphere = new Sphere(100);
        sphere.setTranslateX(-180);
        sphere.setTranslateY(-100);
        sphere.setTranslateZ(100);
        root.getChildren().add(sphere);

        Image earthImage = new Image("file:earth.jpg");
        PhongMaterial earthPhong = new PhongMaterial();
        earthPhong.setDiffuseMap(earthImage);
        sphere.setMaterial(earthPhong);


        //STEP EIGHT: ADD A MESH OBJECT

        TriangleMesh pyramidMesh = new TriangleMesh();

        pyramidMesh.getTexCoords().addAll(0,0);

        float h = 150;                    // Height
```

```
float s = 300;                    // Side
pyramidMesh.getPoints().addAll(
        0,    0,    0,            // Point 0 - Top
        0,    h,    -s/2,       // Point 1 - Front
        -s/2, h,    0,            // Point 2 - Left
        s/2,  h,    0,            // Point 3 - Back
        0,    h,    s/2            // Point 4 - Right
    );

pyramidMesh.getFaces().addAll(
        0,0,  2,0,  1,0,        // Front left face
        0,0,  1,0,  3,0,        // Front right face
        0,0,  3,0,  4,0,        // Back right face
        0,0,  4,0,  2,0,        // Back left face
        4,0,  1,0,  2,0,        // Bottom rear face
        4,0,  3,0,  1,0           // Bottom front face
    );

MeshView pyramid = new MeshView(pyramidMesh);
pyramid.setDrawMode(DrawMode.FILL);
pyramid.setMaterial(blueStuff);
pyramid.setTranslateX(200);
pyramid.setTranslateY(100);
pyramid.setTranslateZ(200);
root.getChildren().add(pyramid);


// STEP NINE: ANIMATE THE OBJECTS

RotateTransition rt1 = new RotateTransition();
rt1.setNode(box);
rt1.setDuration(Duration.millis(3000));
rt1.setAxis(Rotate.Z_AXIS);
rt1.setByAngle(360);
rt1.setCycleCount(Animation.INDEFINITE);
rt1.setInterpolator(Interpolator.LINEAR);
rt1.play();

RotateTransition rt2 = new RotateTransition();
rt2.setNode(cylinder);
rt2.setDuration(Duration.millis(3000));
rt2.setAxis(Rotate.X_AXIS);
rt2.setByAngle(360);
rt2.setCycleCount(Animation.INDEFINITE);
rt2.setInterpolator(Interpolator.LINEAR);
rt2.play();

RotateTransition rt3 = new RotateTransition();
rt3.setNode(pyramid);
rt3.setDuration(Duration.millis(3000));
```

(continued)

Listing 20-1 ***(continued)***

```
        rt3.setAxis(Rotate.Y_AXIS);
        rt3.setByAngle(360);
        rt3.setCycleCount(Animation.INDEFINITE);
        rt3.setInterpolator(Interpolator.LINEAR);
        rt3.play();

        RotateTransition rt4 = new RotateTransition();
        rt4.setNode(sphere);
        rt4.setDuration(Duration.millis(9000));
        rt4.setAxis(Rotate.Y_AXIS);
        rt4.setByAngle(360);
        rt4.setCycleCount(Animation.INDEFINITE);
        rt4.setInterpolator(Interpolator.LINEAR);
        rt4.play();


        // STEP TEN: ADD A LIGHT SOURCE

        PointLight light = new PointLight(Color.WHITE);
        light.setTranslateX(-1000);
        light.setTranslateY(100);
        light.setTranslateZ(-1000);
        root.getChildren().add(light);


        // Finalize and show the stage

        primaryStage.setScene(scene);
        primaryStage.setTitle("3D World");
        primaryStage.show();
    }
}
```