The Java3D Programming Model

by George Crawford III

Introduction

In the previous *Objective Viewpoint* article, we learned the steps required to develop a simple Java3D application. This article explains how to create a scene graph for a Java3D application using the Simple3D source code from our previous tutorial to derive the graph.

This tutorial is presented in the following way: first, we briefly explain the Java3D programming model and show the scene graph and source code for the `Simple3D.java` program developed in the last article. Then we step through the source code and show how that code relates to the scene graph. In this way, you should be able to see how the two are related and have have a better understanding of how to begin developing complex Java3D programs from a design perspective.

Java3D Architecture

A Java3D virtual universe is specified using a non-cyclic, directed scene graph. A directed scene graph is a tree structure that describes the scene to be rendered. The nodes in the graph form a parent-child relationship, where a parent node has one or more child nodes. Figure 1 depicts the scene graph that represents the Java3D program structure constructed in the preceding article.

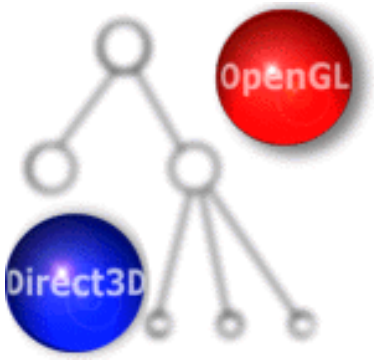

Figure 1: *Simple3D.java* program scene graph

Scene Graph Components

Table 1 lists the components that comprise the scene graph; there is no need to fully understand each element at this point.

Graph Element	Description
VirtualUniverse	Represents a virtual environment.
Locale	High-resolution 3D coordinate that provides a frame of reference for objects within the universe.

Table 1

The Virtual Universe

A **VirtualUniverse** object represents all the virtual objects that can exist within a particular 3D environment. There can be more than one VirtualUniverse in a Java3D application. However, VirtualUniverses are mutually exclusive (objects in one VirtualUniverse cannot exist in another VirtualUniverse, nor can they interact or be interchanged with objects in another VirtualUniverse).

Listing 1 shows the definition of the `SimpleUniverse` class (the file is located in `package com.sun.j3d.utils`). This class is provided by Sun to enable rapid development and prototyping of 3D environments. Now look at the declaration in Listing 1.

```
public class SimpleUniverse extends VirtualUniverse
```

Listing 1: **SimpleUniverse** class definition

The top node is a VirtualUniverse object. This object must be created before other Java3D objects can be added to the scene graph. The code used to create the VirtualUniverse object in our `Simple3D.java` program is shown in Listing 2.

```
SimpleUniverse simpleUniverse = new SimpleUniverse(canvas3D);
```

Listing 2: Code to create the **VirtualUniverse** object

This single line of code hides much of the complexity associated with the creation of a VirtualUniverse. To get a deeper understanding of Java3D, we will have to look at what is going on behind the scenes.

The Locale

Attached to the VirtualUniverse is a **Locale** object. The Locale object uses high-resoluton coordinates to provide a frame of reference for objects within a particular subgraph of a virtual universe. A VirtualUniverse object may have a Locale for each group of scene graph objects. For example, if you were to model two solar systems that existed in the same universe, you would probably want to define one locale for the planets in one solar system,

and a different locale for the planets in the other solar system.

The **SimpleUniverse** object that we created in our previous program only required one parameter. If you look at the source code for the SimpleUniverse utility class, you will see that the single-parameter constructor actually calls a constructor that accepts four parameters. The four-parameter constructor is defined as shown in Listing 3.

```
public SimpleUniverse(HiResCoord origin, int numTransforms, Canvas3D
canvas, URL userConfig)
```

Listing 3: **SimpleUniverse** Constructor that accepts four parameters

In our program, however, only the **Canvas3D** parameter is used. The SimpleUniverse constructor code creates default objects in place of those we omitted. What we are most concerned with right now is the segment of code that is used to create the Locale object:

```
if(origin == null)
        locale = new Locale(this);
else
        locale = new locale(this, origin);
```

`Locale objects require at least one parameter, a VirtualUniverse object. The locale constructor attaches the locale to the` **`VirtualUniverse`**`. The other parameter is a high-resolution coordinate used to position the locale anywhere in the 3D universe.`

The ViewPlatform

Now we have the VirtualUniverse object and an attached Locale object that can be used to locate other objects in virtual space. The next step is to set up Java3D for rendering.

If you traverse the right side of the scene graph from the Locale object, the first node encountered is a **BranchGroup** node. Sun has provided another utility class called the **ViewingPlatform** that extends the BranchGroup class. This class can be found in the `com.sun.j3d.util.universe` package. An instance of this class is created in the SimpleUniverse constructor: `viewingPlatform = new ViewingPlatform (numTransforms);`

Again, this code hides a lot of details that are of interest to us. For the purposes of this article, we are only interested in the creation of the **TransformGroup** node and the **ViewPlatform** node.

TransformGroup Node

A TransformGroup is used to specify a **transformation** for virtual objects; examples of transformations include position, orientation, and scaling. The TransformGroup object is created in the ViewingPlatform constructor method:

```
TransformGroup tg = mtg.getTransformGroup(0);
addChild(tg);
```

The *mtg* object is an instance of a **MultiTransformGroup** that is used to hold all TransformGroup objects between the **BranchGroup** and the **View** object. In our case, we only have one Transform Group between the two.

ViewPlatform Node

The **ViewPlatform** node is a **Leaf** node that is used to orient, position, and scale the viewer. Navigation through a scene is performed by changing the **TransformGroup** node located above the **ViewPlatform** node in the scene graph. The code used to create the ViewPlatform in the **ViewingPlatform** constructor is as follows:

```
  viewPlatform = new ViewPlatform();
    ... // Set capability flags
  tg.addChild(viewPlatform);
```

This completes the graph from the right side of the **Locale** node to the **ViewPlatform** node.

View

The **ViewPlatform** object contains a **View** object. The View object is used to position, orient, and scale the viewer. Several view objects can exist simultaneously, but in our case we only use one. The View object is created using the Sun-provided **Viewer** convenience class (in the `com.sun.j3d.universe` package).

Viewer

The **Viewer** class is used to contain information about the physical and virtual environment. It contains the View object shown on the scene graph. The first few lines of code in the **Viewer** constructor create the **PhysicalBody** and **PhysicalEnvironment**. The PhysicalBody object contains the geometry for the user's avatar. In our case the default parameters are used since we do not specify a URL from which to load the geometry. The PhysicalEnvironment represents links to the outside world that are used to manipulate the 3D environment, such as virtual reality headsets and audio devices. However, this object is not used in our example. The last few lines of code in the **Viewer** constructor create the **View** object and set the **Canvas3D** object that will be used to render the scene. Segments of the relevant **Viewer** constructor code is show below:

```
  if(physicalBody == null) {
    physicalBody = new PhysicalBody();
        ....
  }
  if(physicalEnvironment == null) {
    physicalEnvironment = new PhysicalEnvironment();
      ...
  }
```

```
  if(userCanvas == null)
    canvas = new Canvas3D(null);
        ... // code to create the frame and add the canvas here
  }
  view = new View();
  view.addCanvas3D(canvas);
  view.setPhysicalBody(physicalBody);
  view.setPhysicalEnvironment(physicalEnvironment);

  ... // set the clipping planes
```

Background

The **Background** node is used to fill the canvas with either a solid color or an image with each rendering frame. The code used to create the background is located in the `createBackground` method in the **Simple3D** code (shown below). We need to create and initialize the **Background** leaf node and then add it to the **BranchGroup** node (located on the left side of the **Locale** node in the scene graph).

Shape3D

Finally, we are ready to add an object to our scene graph. In our case, we use the convenience class **ColorCube** provided by Sun (in package `com.sun.j3d.utils.geometry`). Since we want the cube to rotate, we have to add a **TransformGroup** object to the **BranchGroup** node, and then attach the cube to the **TransformGroup** node. The code to do this can be found in the `createTransformation` method of the `Simple3D` class.

Conclusion

In this article, we examined the scene graph associated with a Java3D program. This information is important in order to fully understand the Java3D programming model. For further study, try constructing a scene graph from one of the demos provided with the Java3D SDK.

```
/*
 * Simple3D.java
 *
 * Author: George Crawford III
 *
 * Permission to use and modify this source code is hereby granted provided
 * the author's name is included in all modified versions of this software.
 */
import java.applet.Applet;
import java.awt.Frame;
import java.awt.BorderLayout;
import com.sun.j3d.utils.applet.MainFrame;
```

```
import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.SimpleUniverse;
import com.sun.j3d.utils.image.TextureLoader;
import javax.media.j3d.BranchGroup;
import javax.media.j3d.BoundingSphere;
import javax.media.j3d.Background;
import javax.media.j3d.Canvas3D;
import javax.media.j3d.Transform3D;
import javax.media.j3d.TransformGroup;
import javax.vecmath.Point3d;

public class Simple3D extends Applet {
 public void createSimpleUniverse() {
    SimpleUniverse simpleUniverse = new SimpleUniverse(canvas3D);
    simpleUniverse.getViewingPlatform().setNominalViewingTransform();
    simpleUniverse.addBranchGraph(sceneGraph);
  }
  public void createBackground() {
    BoundingSphere boundingSphere =
          new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 100.0);
        TextureLoader backgroundTexture =
      new TextureLoader(backgroundImage, this);
        Background background =
          new Background(backgroundTexture.getImage());
        background.setApplicationBounds(boundingSphere);
        sceneGraph.addChild(background);
  }
  public void createSceneGraph() {
        // Create the root of the scene graph
        sceneGraph = new BranchGroup();

        // Create the background for the scene and add it to the scene graph
        createBackground();

        // Create a 3D transformation and add it to the scene graph.
        createTransformation();

        // Allow Java3D to optimize the scene graph.
    sceneGraph.compile();
  }
  public void createTransformation() {
    // Create a 3D transformation that will
        // rotate the cube a specified number of degrees on the x axis
        Transform3D transform3D = new Transform3D();
    transform3D.rotX(xRotationAngle);

        // Create the transform group node and add the transformation
```

```
    // to the node. Add the transformation group to the scene graph.
        TransformGroup transformGroup = new TransformGroup(transform3D);
        sceneGraph.addChild(transformGroup);

        // Add a colored cube to the transform group node.
        transformGroup.addChild(new ColorCube(cubeScale));
  }
  public void layoutComponents() {
    setLayout(new BorderLayout());
    canvas3D = new Canvas3D(null);
    add("Center", canvas3D);
  void layoutComponents() {
    setLayout(new BorderLayout());
    canvas3D = new Canvas3D(null);
    add("Center", canvas3D);
  }
  public Simple3D(String background, double scale, double xAngle) {
        backgroundImage = background;
        cubeScale = scale;
        xRotationAngle = (Math.PI/180)*xAngle; // Convert to radians
        layoutComponents();
        createSceneGraph();
        createSimpleUniverse();
  }
  public static void main(String[] args) {
        String backgroundImage = "m18.jpg";
        double scale = 0.5;
        double xAngle = 90.0;
        if(args.length > 0)
          try {
            for(int i = 0; i  <-xAngle [degrees]>");
        System.err.println("Where: ");
        System.err.print("\t[image] is the image to display in the");
        System.err.println("background");
        System.err.println("\t[scale] is the size of the bounding sphere");
        System.err.print("\t[degrees] is the angle in degrees ");
        System.err.println("for axis rotation");
        System.exit(0);
  }
  private String backgroundImage;       // Background image filename
  private double cubeScale;             // Used to scale the 3D cube
  private double xRotationAngle; // The rotation angle in degrees
  private Canvas3D canvas3D;    // Used to render the 3D scene
  private BranchGroup sceneGraph; // Contains 3D scene information
}
```

Resources

1

Sun Microsystems, Inc. The Java 3D API. Sun Microsystems, Inc. October 4, 1997.

2

Sun Microsystems, Inc. Java 3D API Documentation. July, 1998.

3

Sun Microsystems, Inc. Java 3D API Specification Version 1.1 Beta 01. Sun Microsystems, Inc.

George Crawford III is a software engineer at MPI Software Technology, Inc. He is completing his M.S. degree in computer science at Mississippi State University. His technical areas of interest include software design, distributed object technology, Java programming, game programming, and DirectX/Direct3D programming.