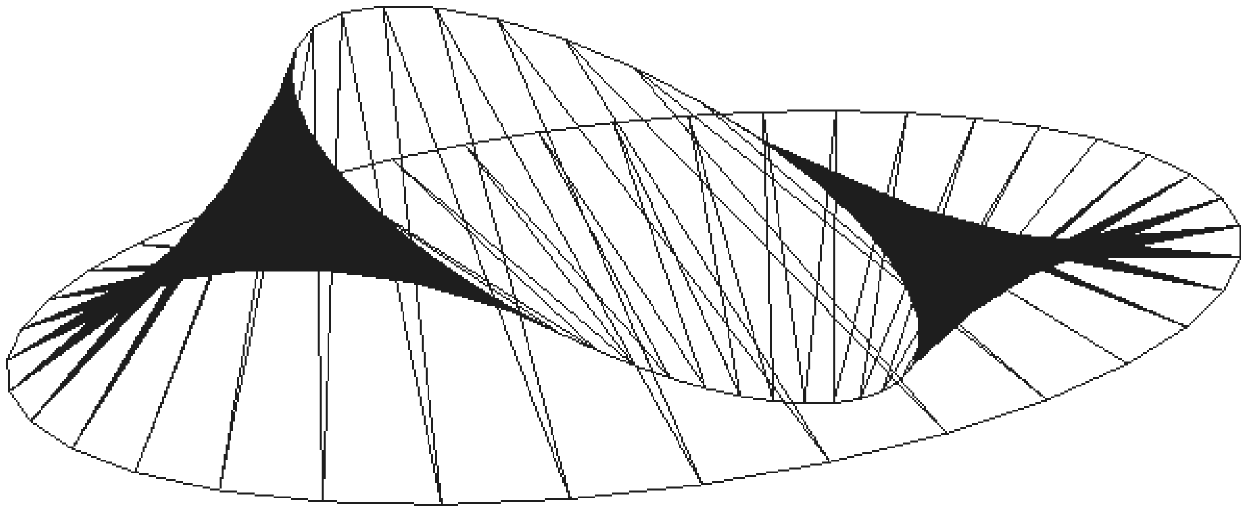
Getting Started with the Java 3D™ API

Chapter 1



Dennis J Bouvier

tutorial v1.5.1 (Java 3D API v1.2)

Table of Contents

Figures

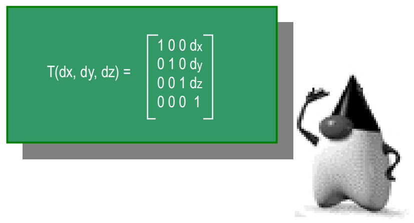
Code Fragments

Reference Blocks

Preface to Chapter 1

This document is the first part of a tutorial on using the Java 3D API. Additional chapters and the full preface to this material is presented in the Module 0 document available at:
`http://java.sun.com/products/javamedia/3d/collateral`

CHAPTER 1 Getting Started



Chapter Objectives

After reading this chapter, you will:

- Be able to explain in general terms what Java 3D is
- Be able to describe the basic structure of Java 3D programs
- Recognize many classes from the Java 3D API
- Be able to write some simple animated Java 3D programs

The Java 3D API is an interface for writing programs to display and interact with three-dimensional graphics. Java 3D is a standard extension to the Java 2 JDK. The API provides a collection of high-level constructs for creating and manipulating 3D geometry and structures for rendering that geometry. Java 3D provides the functions for creation of imagery, visualizations, animations, and interactive 3D graphics application programs.

1.1 What is the Java 3D API?

The Java 3D API is a hierarchy of Java classes which serve as the interface to a sophisticated three-dimensional graphics rendering and sound rendering system. The programmer works with high-level constructs for creating and manipulating 3D geometric objects. These geometric objects reside in a virtual universe, which is then rendered. The API is designed with the flexibility to create precise virtual universes of a wide variety of sizes, from astronomical to subatomic.

Despite all this functionality, the API is still straightforward to use. The details of rendering are handled automatically. By taking advantage of Java threads, the Java 3D renderer is capable of rendering in parallel. The renderer can also automatically optimize for improved rendering performance.

A Java 3D program creates instances of Java 3D objects and places them into a scene graph data structure. The scene graph is an arrangement of 3D objects in a tree structure that completely specifies the content of a virtual universe, and how it is to be rendered.

Java 3D programs can be written to run as stand alone applications, as applets in browsers which have been extended to support Java 3D, or both[1].

1.2 The Java 3D API

Every Java 3D program is at least partially assembled from objects from the Java 3D class hierarchy. This collection of objects describes a *virtual universe*, which is to be rendered. The API defines over 100 classes presented in the `javax.media.j3d` package. These classes are commonly referred to as the Java 3D *core classes*.

There are hundreds of fields and methods in the classes of the Java 3D API. However, a simple virtual universe that includes animation can be built with only a few classes. This chapter describes a minimal set of objects and their interactions to render a simple virtual universe.

This chapter includes the development of one simple but complete Java 3D program called HelloJava3D, which displays a rotating cube. The example program is developed incrementally, and presented in multiple versions, to demonstrate each part of the Java 3D programming process. All of the programs used in this tutorial are available electronically. See the "Getting This Tutorial" section in the Preface for more information.

In addition to the Java 3D core package, other packages are used in writing Java 3D programs. One such package is the `com.sun.j3d.utils` package that is commonly referred to as the Java 3D *utility classes*. The core class package includes only the lowest-level classes necessary in Java 3D programming. The utility classes are convenient and powerful additions to the core.

The utility classes fall into four major categories: content loaders, scene graph construction aids, geometry classes, and convenience utilities. Future functionality, such as nurbs, likely would be added as utility classes, not in the Java 3D core package. Some utility classes may be moved to the core package in future versions of the Java 3D API.

Using utility classes significantly reduces the number of lines of code in a Java 3D program. In addition to the Java 3D core and utility class packages, every Java 3D program uses classes from the `java.awt` package and `javax.vecmath` package. The `java.awt` package defines the Abstract Windowing Toolkit (AWT). AWT classes create a window to display the rendering. The `javax.vecmath` package defines vector math classes for points, vectors, matrices, and other mathematical objects.

In the rest of the text, the term *visual object* is used to refer to an 'object in the scene graph' (e.g., a cube or a sphere). The term *object* is used only to refer to an instance of a class. The term *content* is used to refer to visual objects in a scene graph as a whole.

1.3 Building a Scene Graph

A Java 3D virtual universe is created from a *scene graph*. A scene graph is created using instances of Java 3D classes. The scene graph is assembled from objects to define the geometry, sound, lights, location, orientation, and appearance of visual and audio objects.

A common definition of a graph is a data structure composed of nodes and arcs. A node is a data element, and arc is a relationship between data elements. The nodes in the scene graph are the instances of Java 3D classes. The arcs represent the two kinds of relationships between the Java 3D instances.

[1] Browser support for Java 3D is available through the Java Plugin, which can be downloaded from java.sun.com. All of the example programs in this tutorial are written as applications.

The most common relationship is a *parent-child relationship*. A group node can have any number of children but only one parent. A leaf node can have one parent and no children. The other relationship is a *reference*. A reference associates a *NodeComponent* object with a scene graph Node. NodeComponent objects define the geometry and appearance attributes used to render the visual objects.

A Java 3D scene graphs is constructed of Node objects in parent-child relationships forming a tree structure. In a tree structure, one node is the root. Other nodes are accessible following arcs from the root. The arcs of a tree form no cycles. A scene graph is formed from the trees rooted at the Locale objects. The NodeComponents and reference arcs are not part of the scene graph tree.

Only one path exists from the root of a tree to each of the leaves; therefore, there is only one path from the root of a scene graph to each leaf node. The path from the root of a scene graph to a specific leaf node is the leaf node's *scene graph path*. Since a scene graph path leads to exactly one leaf, there is one scene graph path for each leaf in the scene graph.

Each scene graph path in a Java 3D scene graph completely specifies the state information of its leaf. State information includes the location, orientation, and size of a visual object. Consequently, the visual attributes of each visual object depend only on its scene graph path. The Java 3D renderer takes advantage of this fact and renders the leaves in the order it determines to be most efficient. The Java 3D programmer normally does not have control over the rendering order of objects[2].

Graphic representations of a scene graph can serve as design tool and/or documentation for Java 3D programs. Scene graphs are drawn using standard graphic symbols as shown in Figure 1-1. Java 3D programs may have many more objects than those of the scene graph.

To design a Java 3D virtual universe a scene graph is drawn using the standard set of symbols. After the design is complete, that scene graph drawing is the specification for the program. After the program is complete, the same scene graph is a concise representation of the program (assuming the specification was followed). A scene graph drawn from an existing program documents the scene graph the program creates.

[2] The only control a Java 3D programmer has over the rendering order is with the OrderedGroup class node. This class is not covered in this tutorial. See The Java 3D API Specification for details.

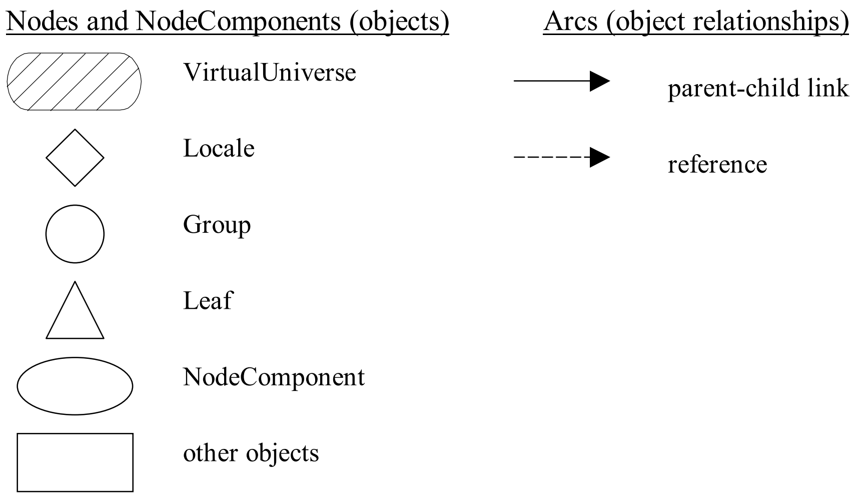


Figure 1-1 Symbols Representing Objects in Scene Graphs

Each of the symbols shown on the left-hand side of Figure 1-1 represents a single object when used in a scene graph. The first two symbols represent objects of specific classes: VirtualUniverse and Locale. The next three symbols on the left represent objects of the Group, Leaf, and NodeComponent classes. These three symbols are often annotated to indicate the subclass of the specific object. The last symbol on the left is used to represent any other class of object.

The solid arrow symbol represents a parent-child relationship between two objects. The dashed arrow is a reference to another object. Referenced objects can be shared among different branches of a scene graph. An example of a simple scene graph is shown in Figure 1-2.

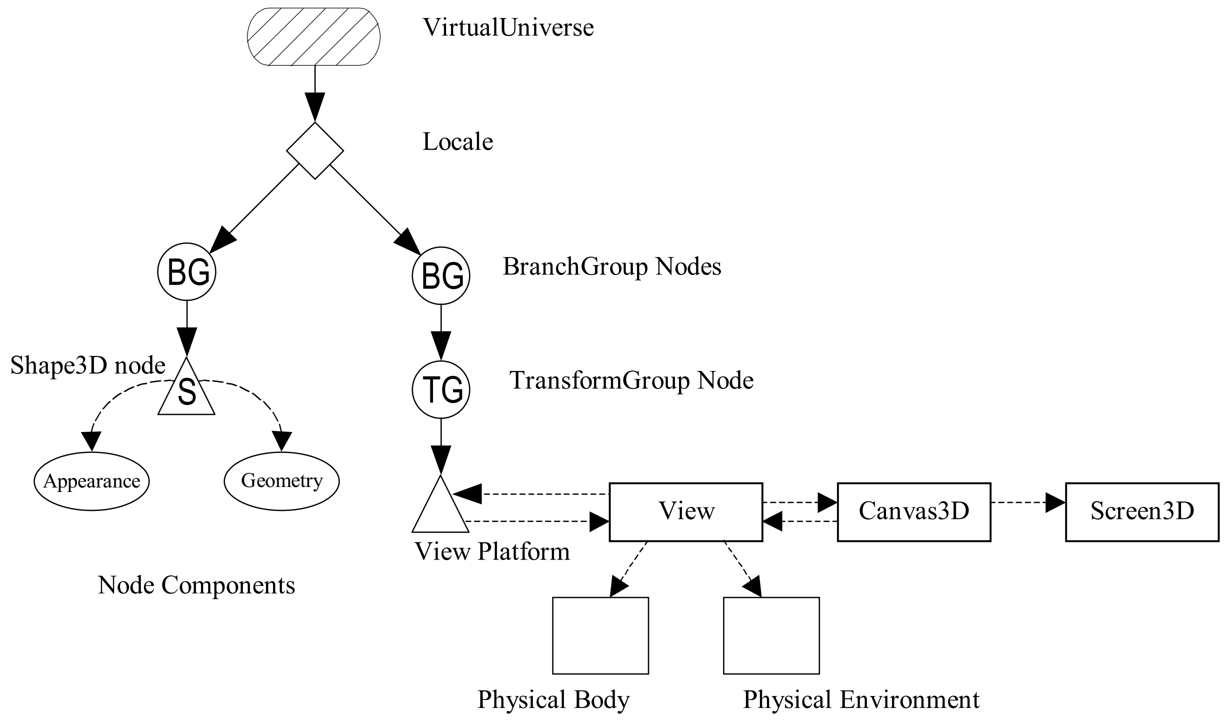


Figure 1-2 First Scene Graph Example

It is possible to create an illegal scene graph. An example illegal scene graph is shown in Figure 1-3. The scene graph depicted in Figure 1-3 is illegal because it violates the properties for a DAG. The problem lies only with the two TransformGroup objects having the same Shape3D leaf object as children. Remember a Leaf object may have only one parent. In other words, there can only be one path from a Locale object to a leaf (or one path from a leaf to a Locale).

You may think the structure shown in Figure 1-3 defines three visual objects in a virtual universe. It appears as though the scene graph defines two visual objects through re-use of the visual (Shape3D) object on the right-hand side of the figure. Conceptually, each of the TransformGroup objects parenting the shared instance of Shape3D could place an image of the visual object in different locations. However, it is an illegal scene graph since the parent-child arcs do not form a tree. In this example, the result is that the Shape3D object has more than one parent.

The discussion of the tree and DAG structures are correct. However, the Java 3D runtime system reports the mistake in terms of child-parent relationships. One result of the tree structure limitation is that each Shape3D object is restricted to one parent. For the example scene graph of Figure 1-3, a 'multiple parent' exception is reported at runtime. Figure 1-4, with one parent for each Shape3D object, shows one possible fix for this scene graph.

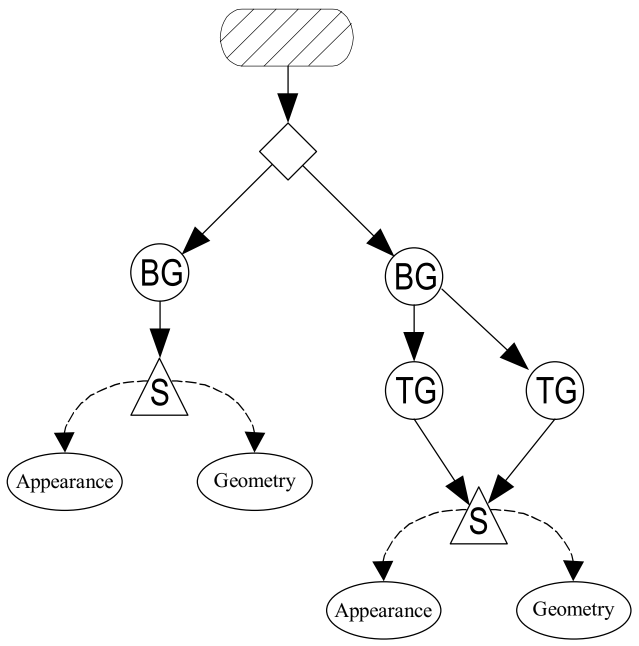


Figure 1-3 Example of an Illegal Scene Graph

A Java 3D program that defines an illegal scene graph may compile, but not render. When a Java 3D program that defines an illegal scene graph is run, the Java 3D system detects the problem. When the problem is detected, the Java 3D system will report an exception. The program may still be running and consequently, needs to be stopped. However, no image will be rendered.

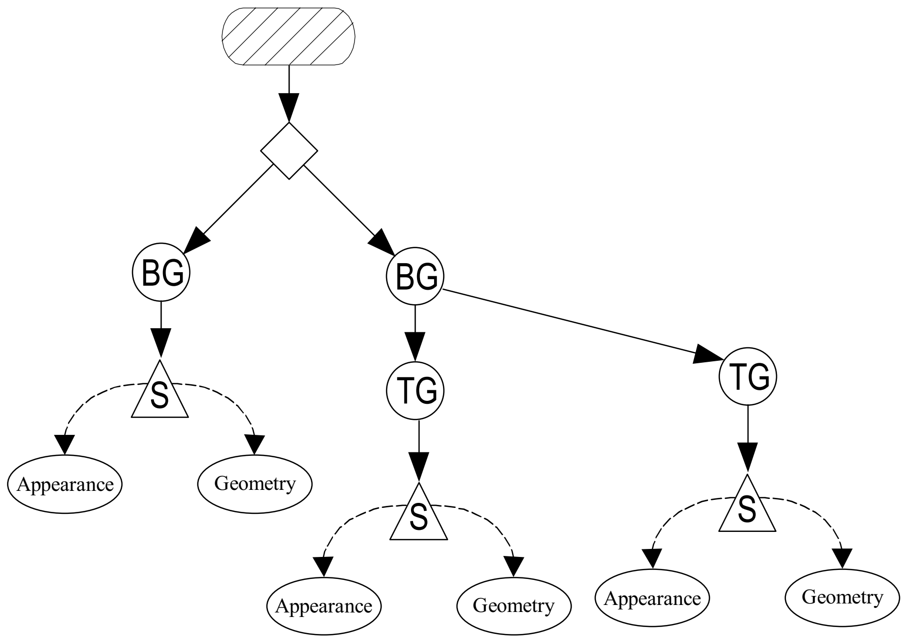


Figure 1-4 One Possible Fix for Illegal Scene Graph of Figure 1-3

Each scene graph has a single VirtualUniverse. The VirtualUniverse object has a list of Locale objects. A Locale object provides a reference point in the virtual universe. Think of a Locale object as being a landmark used to determine the location of visual objects in the virtual universe.

It is technically possible for a Java 3D program to have more than one VirtualUniverse object, thus defining more than one virtual universe. However, there is no inherent way to communicate among virtual universes. Further, a scene graph object can not exist in multiple virtual universes simultaneously. It is highly recommended to use one and only one instance of VirtualUniverse in each Java 3D program.

While a VirtualUniverse object may reference many Locale objects, most Java 3D programs have only one Locale object. Each Locale object may serve as the root of multiple subgraphs of the scene graph. Refer to Figure 1-2 for an example scene graph and note the two subgraph branches from the Locale object in the figure.

A BranchGroup object is the root of a subgraph, or branch graph. There are two different categories of scene subgraph: the *view branch graph* and the *content branch graph*. The content branch graph specifies the *contents* of the virtual universe - geometry, appearance, behavior, location, sound, and lights. The view branch graph specifies the viewing parameters such as the viewing location and direction. Together, the two branches specify much of the work the renderer has to do.

1.3.1 High Level Java 3D API Class Hierarchy

An overview of the first three levels of the Java 3D API hierarchy appears in Figure 1-5. The VirtualUniverse, Locale, Group, and Leaf classes appear in this portion of the hierarchy. Other than the VirtualUniverse and Locale objects, the rest of a scene graph is composed of SceneGraphObject objects. SceneGraphObject is the superclass for nearly every Core and Utility Java 3D class.

SceneGraphObject has two subclasses: Node and NodeComponent. The subclasses of Node provide most of the objects in the scene graph. A Node object is either a Group node or a Leaf node object. Group and Leaf are superclasses to a number of subclasses. Here is a quick look at the Node class, its two subclasses, and the NodeComponent class. After this background material is covered, the construction of Java 3D programs is explained.

Node Class

The Node class is an abstract superclass of Group and Leaf classes. The Node class defines some important common methods for its subclasses. Information on specific methods is presented in later sections after more background material is covered. The subclasses of Node compose scene graphs.

Group Class

The Group class is the superclass used in specifying the location and orientation of visual objects in the virtual universe. Two of the subclasses of Group are BranchGroup and TransformGroup. In the graphical representation of the scene graph, the Group symbols (circles) are often annotated with BG for BranchGroups, TG for TransformGroups, etc. Figure 1-2 shows an example of this.

Leaf Class

The Leaf class is the superclass used in specifying the shape, sound, and behavior of visual objects in the virtual universe. Some of the subclasses of Leaf are Shape3D, Light, Behavior, and Sound. These objects can have no children but may reference NodeComponents.

NodeComponent Class

The NodeComponent class is the superclass used in specifying the geometry, appearance, texture, and material properties of a Shape3D (Leaf) node. NodeComponents are not part of the scene graph, but are referenced by it. A NodeComponent may be referenced by more than one Shape3D object.

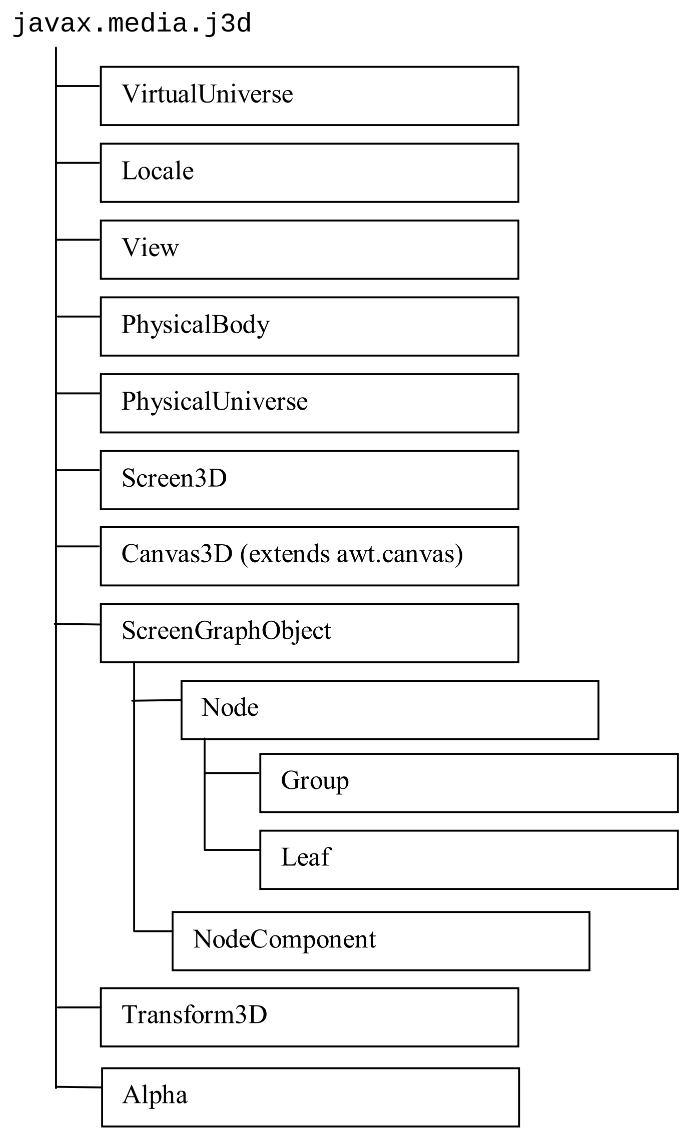


Figure 1-5 An Overview of the Java 3D API Class Hierarchy

1.4 Recipe for Writing Java 3D Programs

The subclasses of SceneGraphObject are the building blocks that are assembled into scene graphs. The basic outline of Java 3D program development consists of seven steps (collectively referred to as a recipe here and in The Java 3D API Specification) presented in Figure 1-6. This recipe can be used to assemble many useful Java 3D programs.

1. Create a Canvas3D object
2. Create a VirtualUniverse object
3. Create a Locale object, attaching it to the VirtualUniverse object
4. Construct a view branch graph
   a. Create a View object
   b. Create a ViewPlatform object
   c. Create a PhysicalBody object
   d. Create a PhysicalEnvironment object
   e. Attach ViewPlatform, PhysicalBody, PhysicalEnvironment, and Canvas3D objects to View object
5. Construct content branch graph(s)
6. Compile branch graph(s)
7. Insert subgraphs into the Locale

Figure 1-6 Recipe for Writing Java 3D Programs

This recipe ignores some detail but illustrates the fundamental concept of all Java 3D programming: creating each branch graph of the scene graph is the majority of the programming. Rather than expand on this recipe, the next section explains an easier way to construct a very similar scene graph with less programming.

1.4.1 A Simple Recipe for Writing Java 3D Programs

Java 3D programs written using the basic recipe have view branch graphs with identical structure. The regularity of view branch graph structure is also found in the SimpleUniverse Utility class. Instances of SimpleUniverse perform steps 2, 3, and 4 from the basic recipe. Using the SimpleUniverse class in Java 3D programming significantly reduces the time and effort needed to create the view branch graph. Consequently, the programmer has more time to concentrate on the content. This is what writing a Java 3D program is really about.

The SimpleUniverse is a good starting point for Java 3D programming because it allows the programmer to ignore the view branch graph. However, using the SimpleUniverse does not allow having multiple views of the virtual universe.

The SimpleUniverse class is used in all the programming examples in this tutorial. Programmers who need more information on View, ViewPlatform, PhysicalBody, and PhysicalEnvironment classes are referred to other references. See Appendix B for a list of references.

SimpleUniverse Class

The SimpleUniverse object constructor creates a scene graph including VirtualUniverse and Locale objects, and a complete view branch graph. The view branch graph created by SimpleUniverse uses instances of ViewingPlatform and Viewer convenience classes in place of the core classes used to create the view branch graph. Note the SimpleUniverse only indirectly uses the View and ViewPlatform objects of the Java 3D core. The SimpleUniverse object supplies the functionality of all of the objects inside the large box in Figure 1-7.

The `com.sun.j3d.utils.universe` package contains the SimpleUniverse, ViewingPlatform, and Viewer convenience utility classes.

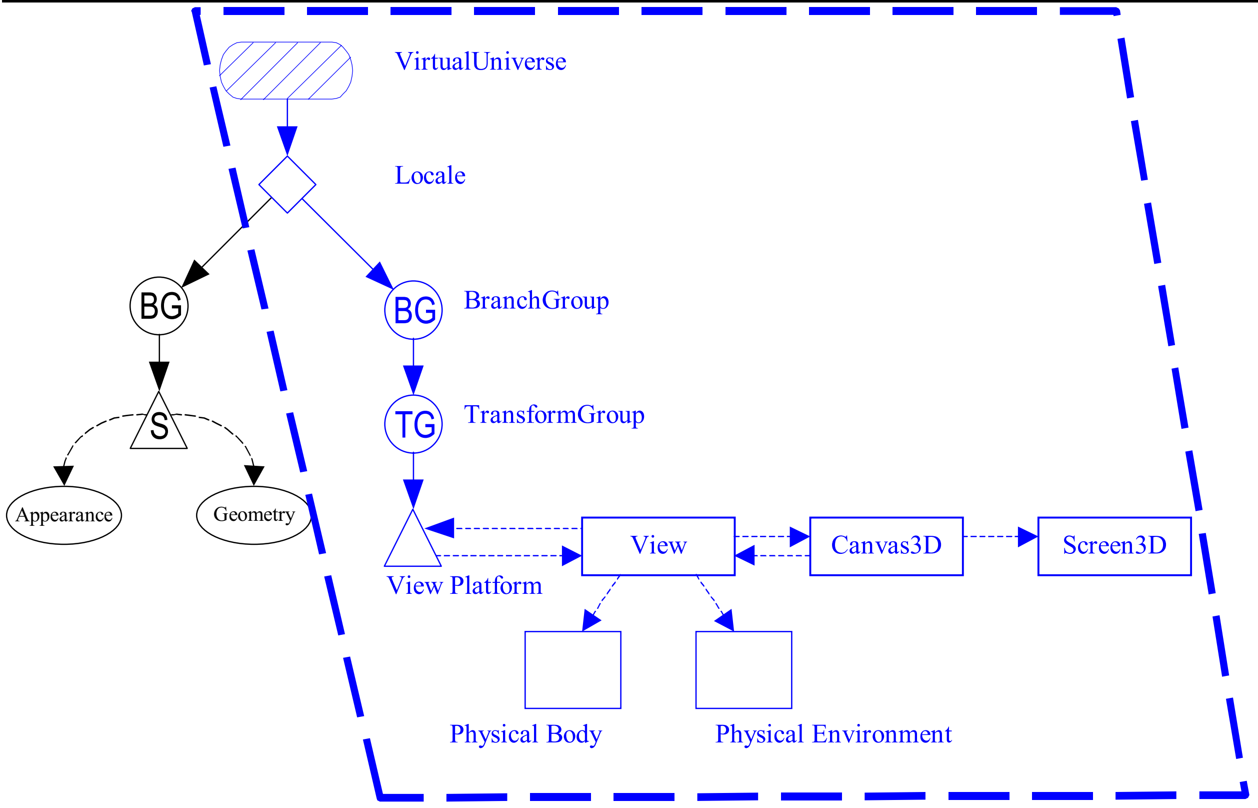


Figure 1-7 A SimpleUniverse Object Provides a Minimal Virtual Universe, Indicated by the Dashed Line.

Using a SimpleUniverse object makes the basic recipe even easier. Figure 1-8 presents the simple recipe, which is the basic recipe modified to use a SimpleUniverse object. Steps 2, 3, and 4 of the basic recipe are replaced by step 2 of the simple recipe.

1. Create a Canvas3D Object
2. Create a SimpleUniverse object which references the earlier Canvas3D object
    a. Customize the SimpleUniverse object
3. Construct content branch
4. Compile content branch graph
5. Insert content branch graph into the Locale of the SimpleUniverse

Figure 1-8 Simple Recipe for Writing Java 3D Programs using SimpleUniverse.

The gray box on the next page is the first instance of a reference block in this tutorial. A reference block lists constructors, methods, or fields of a class. Reference blocks are designed to allow the tutorial reader to learn basic Java 3D API programming without having another reference at hand. The reference blocks in this tutorial do not cover every constructor or method of a class. For that matter, there are many Java 3D API classes without reference block in this tutorial. Therefore, this tutorial document does not replace The Java 3D API Specification. However, for the constructors, methods, or fields listed; the reference blocks in this tutorial typically give more detailed information than The Java 3D API Specification.

SimpleUniverse Constructors

Package: `com.sun.j3d.utils.universe`

This class sets up a minimal user environment to quickly and easily get a Java 3D program up and running. This utility class creates all the necessary objects for a view branch graph. Specifically, this class creates Locale, VirtualUniverse, ViewingPlatform, and Viewer objects (all with their default values). The objects have the appropriate relationships to form the view branch graph.

SimpleUniverse provides all functionality necessary for many basic Java 3D applications. Viewer and ViewingPlatform are convenience utility classes. These classes use the View and ViewPlatform core classes.

`SimpleUniverse()`
Constructs a simple virtual universe.

`SimpleUniverse(Canvas3D canvas3D)`
Construct as simple universe with a reference to the named Canvas3D object.

The SimpleUniverse object creates a complete view branch graph for a virtual universe. The view branch graph includes an *image plate*. An image plate is the conceptual rectangle where the content is projected to form the rendered image. The Canvas3D object, which provides an image in a window on your computer display, can be thought of as the image plate.

Figure 1-9, shows the relationship between the image plate, the eye position, and the virtual universe. The eye position is behind the image plate. The visual objects in front of the image plate are rendered to the image plate. Rendering can be thought of as projecting the visual objects to the image plate. This idea is illustrated with the four *projectors* in the image (dashed lines).

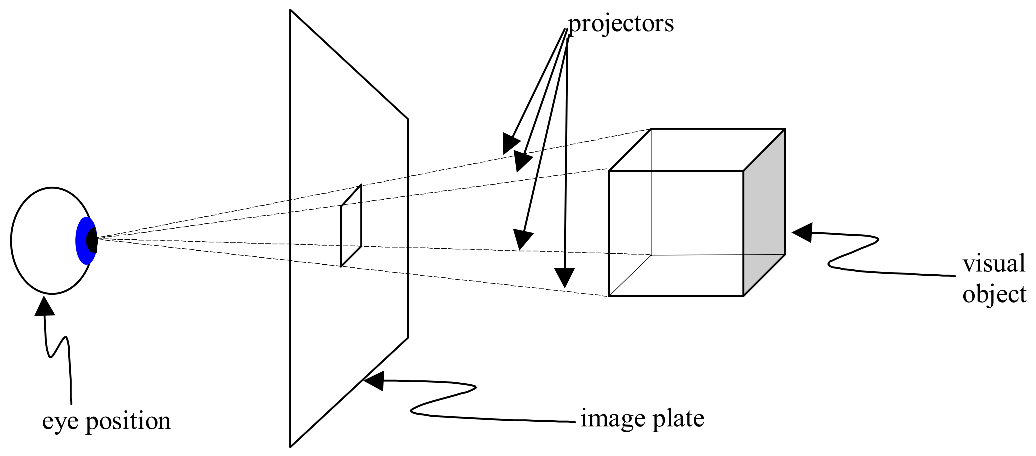


Figure 1-9 Conceptual Drawing of Image Plate and Eye Position in a Virtual Universe.

By default, the image plate is centered at the origin in the SimpleUniverse. The default orientation is to look down the z-axis. From this position, the x-axis is a horizontal line through the image plate, with positive values to the right. The y-axis is a vertical line through the center of the image plate, with positive values up. Consequently, the point (0,0,0) is in the center of the image plate.

The typical Java 3D program moves the view back (positive z) to make objects located at, or near, the origin within the view. The SimpleUniverse class has a member object of ViewingPlatform class. The ViewingPlatform class has the method setNominalViewingTransform which sets the eye position to be centered at (0, 0, 2.41) looking in the negative z direction toward the origin.

ViewingPlatform `setNominalViewingTransform()` Method

Package: `com.sun.j3d.utils.universe`

The ViewingPlatform class is used to set up the view branch graph of a Java 3D scene graph in a SimpleUniverse object. This method is normally used in conjunction with the getViewingPlatform method of the SimpleUniverse class.

`void setNominalViewingTransform()`
Sets the nominal viewing distance to approximately 2.41 meters in the view transform of a SimpleUniverse. At this viewing distance and the default field of view, objects of height and width of 2 meters generally fit on the image plate.

After creating Canvas3D and Simple Universe objects, the next step is the creation of the content branch graph. The regularity of structure found in the view branch graph (that leads to using the SimpleUniverse class) does not exist for the content branch graph. The content branch graph varies from program to program making it impossible to give details for construction in a recipe. It also means that there is no “simple content” class for any universe you may want to assemble.

Creating a content branch graph is the subject of Sections 1.6, 1.7, and 1.9. Compiling the content branch graph is discussed in Section 1.8. If you cannot wait to see some code, see Code Fragment 1-1 for an example of using the SimpleUniverse class.

After creating the content branch graph, it is inserted into the universe using the addBranchGraph method of SimpleUniverse. The addBranchGraph method takes an instance of BranchGroup as the only parameter. This BranchGroup is added as a child of the Locale object created by the SimpleUniverse.

SimpleUniverse Methods (partial list)

Package: `com.sun.j3d.utils.universe`

`void addBranchGraph(BranchGroup bg)`
Used to add Nodes to the Locale object of the scene graph created by the SimpleUniverse. This is used to add a content branch graph to the virtual universe.

`ViewingPlatform getViewingPlatform()`
Used to retrieve the ViewingPlatform object the SimpleUniverse instantiated. This method is used with the setNominalViewingTransform() method of ViewingPlatform to adjust the location of the view position.

1.5 Some Java 3D Terminology

Before moving on to the topic of creating the content branch graph, two Java 3D terms are defined. The terms *live* and *compiled* are defined in this section.

Inserting a branch graph into a Locale makes it *live*, and consequently, each of the objects in that branch graph become live. There are some consequences when an object becomes live. Live objects are subject to being rendered. Also, the parameters of live objects cannot be modified unless the corresponding *capability* has been specifically set before the object became live. Capabilities are explained in Section 1.8.2.

BranchGroup objects can be *compiled*. Compiling a BranchGroup converts the BranchGroup object and all of its ancestors to a more efficient form for the renderer. Compiling BranchGroup objects is recommended as the last step before making it live. It is best to only compile the BranchGroup objects inserted into Locales. Compilation is further discussed in sections 1.8 and 1.8.1.

BranchGroup `compile()` Method

`void compile()`
Compiles the source BranchGroup associated with this object by creating and caching a compiled scene graph.

Concepts of compiled and live are implemented in the SceneGraphObject class. The two methods of the SceneGraphObject class that relate to these concepts are shown in the SceneGraphObject methods reference box below.

SceneGraphObject Methods (partial list)

SceneGraphObject is the superclass of nearly every class used to create a scene graph including Group, Leaf, and NodeComponent. The SceneGraphObject provides a number of common methods and fields for its subclasses; two of which are presented here. The methods of SceneGraphObject associated with "compile" are covered in Section 1.8 Performance Basics.

`boolean isCompiled()`
Returns a flag indicating whether the node is part of a scene graph that has been compiled.

`boolean isLive()`
Returns a flag indicating whether the node is part of a live scene graph.

Note there is no "start the renderer" step in either the basic or simple recipes. The Java 3D renderer starts running in an infinite loop when a branch graph containing an instance of View becomes live in a virtual universe. Once started, the Java 3D renderer conceptually performs the operations shown in Figure 1-10.

```
while(true) {
    Process input
    If (request to exit) break
         Perform Behaviors
         Traverse the scene graph
         and render visual objects
}
Cleanup and exit
```

Figure 1-10 Conceptual Renderer Process

The previous sections explained the construction of a simple virtual universe without a content branch graph. Creating the content branch graph is the subject of the next few sections. Creating content is discussed through the presentation of example programs.

1.6 Simple Recipe Example: HelloJava3Da

The typical Java 3D program begins by defining a new class to extend the `Applet` class. In the `HelloJava3Da.java` example found in the `examples/HelloJava3D` directory, The class

`HelloJava3Da` is extends the Applet class. Some Java 3D programs are written as applications, but using Applet class gives an easy way to produce a windowed application.

The main class of a Java 3D program typically defines a method to construct the content branch graph. In the HelloJava3Da example such a method is defined and it is called createSceneGraph().

All steps of the simple recipe are implemented in the constructor of the HelloJava3Da class. Step 1, create a Canvas3D object, is done on lines 4 through 6. Step 2, create a SimpleUniverse object, is done on line 13. Step 2a, customize the SimpleUniverse object, is accomplished on line 17. Step 3, construct content branch, is accomplished by a call to the createSceneGraph() method. Step 4, compile content branch graph, is done on line 10. Finally, step 5, insert content branch graph into the Locale of the SimpleUniverse, is completed on line 19.

```
1. public class HelloJava3Da extends Applet {
2.     public HelloJava3Da() {
3.         setLayout(new BorderLayout());
4.         GraphicsConfiguration config =
5.            SimpleUniverse.getPreferredConfiguration();
6.         Canvas3D canvas3D = new Canvas3D(config);
7.         add("Center", canvas3D);
8.
9.         BranchGroup scene = createSceneGraph();
10.        scene.compile();
11.
12.        // SimpleUniverse is a Convenience Utility class
13.        SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
14.
15.        // This moves the ViewPlatform back a bit so the
16.        // objects in the scene can be viewed.
17.        simpleU.getViewingPlatform().setNominalViewingTransform();
18.
19.        simpleU.addBranchGraph(scene);
20.    } // end of HelloJava3Da (constructor)
```

Code Fragment 1-1 Class HelloJava3Da

Step 3 of the simple recipe is to create the content branch graph. A content branch graph is created in Code Fragment 1-2. It is probably the simplest content branch graph possible. The content branch created in Code Fragment 1-2 contains one static graphical object, a ColorCube. The ColorCube is located at the origin of the *virtual world* coordinate system. With the given location and orientation of the viewing direction and the cube, the cube appears as a rectangle when rendered. The image is shown after all of the code for the program is presented, in Figure 1-12.

```
    public BranchGroup createSceneGraph() {
        // Create the root of the branch graph
        BranchGroup objRoot = new BranchGroup();

        // Create a simple shape leaf node, add it to the scene graph.
        // ColorCube is a Convenience Utility class
        objRoot.addChild(new ColorCube(0.4));

        return objRoot;
    } // end of createSceneGraph method of HelloJava3Da
} // end of class HelloJava3Da
```

Code Fragment 1-2 Method createSceneGraph for Class HelloJava3Da

The class HelloJava3Da is derived from Applet but the program is runnable as an application with the use of the MainFrame class. The Applet class is used as a base class to make it easy to write a Java 3D program that runs in a window. MainFrame provides an AWT frame (window) for an applet allowing the applet to run as an application. The window size of the resulting application is specified in the construction of the MainFrame class. Code Fragment 1-3 shows the use of MainFrame in `HelloJava3Da.java`.

MainFrame Constructor (partial list)

package: `com.sun.j3d.utils.applet`

MainFrame makes an applet into an application. A class derived from applet may have a main() method which calls the MainFrame constructor. MainFrame extends `java.awt.Frame` and implements `java.lang.Runnable`, `java.applet.AppletStub`, and `java.applet.AppletContext`. The MainFrame class is Copyright © 1996-1998 by Jef Poskanzer email: jef@acme.com http://www.acme.com/java/

`MainFrame(java.applet.Applet applet, int width, int height)`
Creates a MainFrame object that runs an applet as an application.
Parameters:
- applet – the constructor of a class derived from applet. MainFrame provides an AWT frame for this applet.
- width – the width of the window frame in pixels
- height – the height of the window frame in pixels

```
    //  The following allows this to be run as an application
    //  as well as an applet

    public static void main(String[] args) {
        Frame frame = new MainFrame(new HelloJava3Da(), 256, 256);
    } // end of main (method of HelloJava3Da)
```

Code Fragment 1-3 Main() Method of HelloJava3Da Invokes MainFrame

The three preceding code fragments (1-1, 1-2, and 1-3) form a complete Java 3D program when the proper import statements are used. The following import statements are necessary to compile the class `HelloJava3Da`. The most commonly used classes in Java 3D programming are found in the `javax.media.j3d`, or `javax.vecmath` packages. In this example, only the ColorCube Utility class is found in the `com.sun.j3d.utils.geometry` package. Consequently, most Java 3D programs have the import statements shown in Code Fragment 1-4 with the exception of the import for ColorCube.

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.*;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.ColorCube;
import javax.media.j3d.*;
import javax.vecmath.*;
```

Code Fragment 1-4 Import Statements for HelloJava3Da.java

In the `HelloJava3Da.java` example program, a single graphic object was placed in a single locale. The resulting scene graph is shown in Figure 1-11.

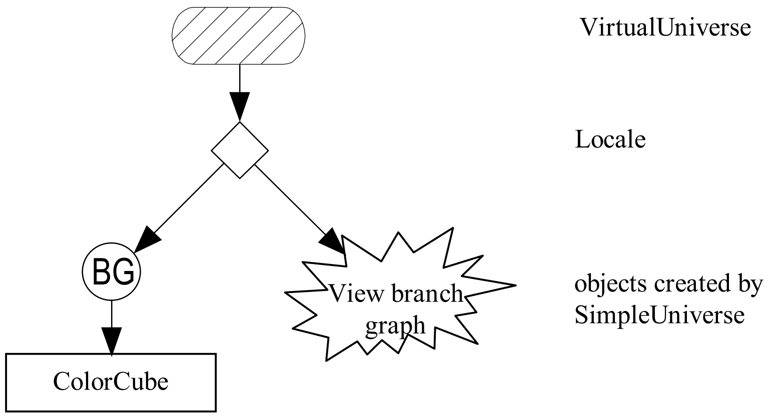


Figure 1-11 Scene Graph for HelloJava3Da Example

The four preceding code fragments (1-1, 1-2, 1-3, and 1-4) form the complete `HelloJava3Da.java` example program. The complete program is found in the `examples/HelloJava3D` directory of the distribution. Compile the code by issuing the command: `javac HelloJava3Da.java`. Run the program with the command: `java HelloJava3Da`. The image produced by HelloJava3Da is shown in Figure 1-12.

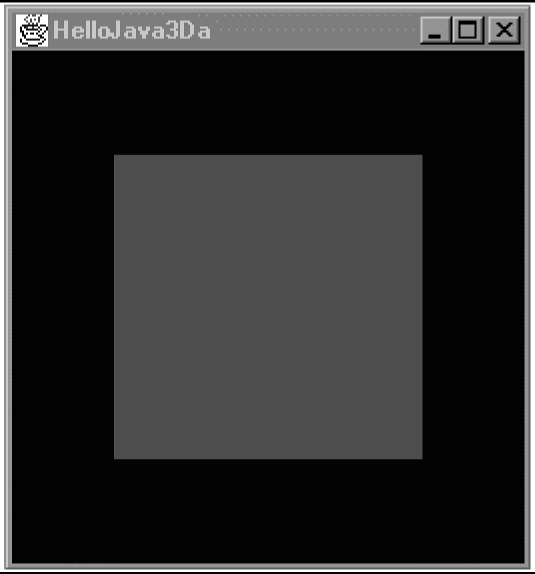


Figure 1-12 Image Produced by HelloJava3Da

While not every line of code of the HelloJava3Da example is explained, the basic ideas of assembling a Java 3D program should be clear having read the example. The following section fills in some of the gaps by presenting the classes used in the program.

1.6.1 Java 3D Classes Used in HelloJava3Da

To add to the understanding of the Java 3D API and the `HelloJava3Da` example a synopsis of each of the Java 3D API classes used in the `HelloJava3Da` example program are presented here.

BranchGroup Class

Objects of this type are used to form scene graphs. Instances of BranchGroup are the root of subgraphs. BranchGroup objects are the only objects allowed to be children of Locale objects. BranchGroup objects can have multiple children. The children of a BranchGroup object can be other Group or Leaf objects.

BranchGroup Default Constructor

`BranchGroup()`
Instances of BranchGroup serve as roots of scene graph branches; BranchGroup objects are the only objects that can be inserted into a Locale's set of objects.

Canvas3D Class

The Canvas3D class is derived from the Canvas class of the Abstract Windowing Toolkit (AWT). At least one Canvas3D object must be referenced in the viewing branch graph of the scene graph[3]. For more information on the Canvas class, consult a reference on the AWT. A list of references appears in Appendix B.

Canvas3D Constructor

`Canvas3D(GraphicsConfiguration graphicsconfiguration)`
Constructs and initializes a new Canvas3D object that Java 3D can render given a valid GraphicsConfiguration object. It is an extension of the AWT Canvas class. For more information on the GraphicsConfiguration object see the Java 2D specification, which is part of the AWT in JDK 1.2.

Transform3D Class

Transform3D objects represent transformations of 3D geometry such as translation and rotation. These objects are typically only used in the creation of a TransformGroup object. First, the Transform3D object is constructed, possibly from a combination of Transform3D objects. Then the TransformGroup object is constructed using the Transform3D object.

Transform3D Default Constructor

A generalized transform object is represented internally as a 4x4 double precision floating-point matrix. The mathematical representation is row major. A Transform3D object is not used in a scene graph. It is used to specify the transformation of a TransformGroup object.

`Transform3D()`
Constructs a Transform3D object that represents the identity matrix (no transformation).

A Transform3D object can represent translation, rotation, scaling, or a combination of these. When specifying a rotation, the angle is expressed in radians. One full rotation is 2 PI radians. One way to specify angles is to use the constant `Math.PI`. Another way is to specify values directly. Some approximations are: 45 degrees is 0.785, 90 degrees is 1.57, and 180 degrees is 3.14.

[3] It is possible to have more than one. To keep things simple, using the SimpleUniverse, there will be only one instance of Canvas3D in the programs presented.

Transform3D Methods (partial list)

Transform3D objects represent geometric transformations such as rotation, translation, and scaling. Transform3D is one of the few classes not directly used in any scene graph. The transformations represented by a Transform3D object are used to create TransformGroup objects that are used in scene graphs.

`void rotX(double angle)`
Sets the value of this transform to a counter clockwise rotation about the x-axis. The angle is specified in radians.

`void rotY(double angle)`
Sets the value of this transform to a counter clockwise rotation about the y-axis. The angle is specified in radians.

`void rotZ(double angle)`
Sets the value of this transform to a counter clockwise rotation about the z-axis. The angle is specified in radians.

`void set(Vector3f translate)`
Sets the translational value of this matrix to the Vector3f parameter values, and sets the other components of the matrix as if this transform were an identity matrix.

TransformGroup Class

As a subclass of the Group class, instances of TransformGroup are used in the creation of scene graphs and have a collection of child node objects. TransformGroup objects hold geometric transformations such as translation and rotation. The transformation is typically created in a Transform3D object, which is not a scene graph object.

TransformGroup Constructors

TransformGroup objects are holders of transformations in the scene graph.

`TransformGroup()`
Constructs and initializes a TransformGroup using an identity transform.

`TransformGroup(Transform3D t1)`
Constructs and initializes a TransformGroup from the Transform3D object passed.
Parameters:
t1 - the transform3D object

The transform held in a Transform3D object is copied to a TransformGroup object either when the TransformGroup object is created, or by using the setTransform() method.

TransformGroup `setTransform()` Method

`void setTransform(Transform3D t1)`
Sets the transform component of this TransformGroup to the value of the passed transform.
Parameters:
t1 - the transform to be copied.

Vector3f Class

Vector3f is a math class found in the `javax.vecmath` package for specifying a vector using three floating-point values. Vector objects are often used to specify translations of geometry. Vector3f objects are not used directly in the construction of a scene graph. They are used to specify translation, surface normals, or other things.

Vector3f Constructors

A 3-element vector that is represented by single precision floating point x, y, and z coordinates.

`Vector3f()`
Constructs and initializes a Vector3f to (0,0,0).

`Vector3f(float x, float y, float z)`
Constructs and initializes a Vector3f from the specified x, y, z coordinates.

ColorCube

ColorCube is a utility class found in the `com.sun.j3d.utils.geometry` package that defines the geometry and colors of a cube centered at the origin with different colors on each face. The default ColorCube object is in a cube that is 2 meters high, wide and deep. If a non-rotated cube is placed at the origin (as in HelloJava3Da), the red face will be visual from the nominal viewing location. Other colors are blue, magenta, yellow, green, and cyan.

ColorCube Constructors

Package: `com.sun.j3d.utils.geometry`

A ColorCube is a simple color-per-vertex cube visual object with a different color for each face. ColorCube extends the Shape3D class; therefore, it is a Leaf node. ColorCube is easy to use when putting together a virtual universe.

`ColorCube()`
Constructs a color cube of default size. By default, a corner is located 1 meter along each of the axis from the origin, resulting in a cube that is centered at the origin and is 2 meters high, wide and deep.

`ColorCube(double scale)`
Constructs a color cube scaled by the value specified. The default size is 2 meters on an edge. The resulting ColorCube has corners at (scale, scale, scale) and (-scale, -scale, -scale).

1.7 Rotating the Cube

A simple rotation of the cube can be made to show more than one face of the cube. The first step is to create the desired transformation using a Transform3D object.

The Code Fragment 1-5 incorporates a TransformGroup object in the scene graph to rotate the cube about the x-axis. The rotation transformation is first created using the Transform3D object **rotate**. The Transform3D object is created on line 6. The rotation is specified using the rotX() method on line 8. The TransformGroup object is then created holding the rotation transform on line 10.

Two parameters specify the rotation: the axis of revolution, and the angle of rotation. The axis is chosen by selecting the proper method. The angle of rotation is the value that is the argument to the method. Since the angle of rotation is specified in radians, the value Π/4 is 1/8 of a full rotation, or 45 degrees. After creating the Transform3D object, **rotate**, it is used in the creation of a TransformGroup object objRotate (line 10). The Transform3D object is used in the scene graph. The **objRotate** object then makes the ColorCube object its child (line 11). In turn, the **objRoot** object makes **objRotate** its child (line 12).

The Transform3D methods rotX(), rotY, and rotZ() are listed in a reference block in the previous section.

```
1. public BranchGroup createSceneGraph() {
2.         // Create the root of the branch graph
3.         BranchGroup objRoot = new BranchGroup();
4.
5.         // rotate object has composite transformation matrix
6.         Transform3D rotate = new Transform3D();
7.
8.         rotate.rotX(Math.PI/4.0d);
9.
10.        TransformGroup objRotate = new TransformGroup(rotate);
11.        objRotate.addChild(new ColorCube(0.4));
12.        objRoot.addChild(objRotate);
13.        return objRoot;
14.    } // end of createSceneGraph method
```

Code Fragment 1-5 One Rotation in the Content Branch Graph

The content branch graph now includes a TransformGroup object in the scene graph path to the ColorCube object. Each of the objects in the scene graph path is necessary. The BranchGroup object is the only object that can be a child of a Locale. The TransformGroup object is the only object that can change the location, orientation, or size of a visual object. In this case, the TransformGroup object changes the orientation. Of course, the ColorCube object is necessary to supply the visual object.

The content branch graph produced by Code Fragment 1-5 is shown in Figure 1-13.

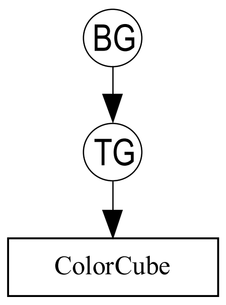


Figure 1-13 Scene Graph for Content Branch Graph Created in Code Fragment 1-5

Code Fragment 1-5 is not used in a program in the example subdirectory. It is only presented as a stepping stone to bigger and more interesting programs. The bigger problem, presented next, is combining two transformations in one TransformGroup object.

1.7.1 Combination of Transformations Example: HelloJava3Db

Quite often a visual object is translated and rotated, or rotated about two axes. In either case, two different transformations are specified for a single visual object. The two transformations can be combined into one transformation matrix and held by a single TransformGroup object. An example of this is shown in Code Fragment 1-6.

Two rotations are combined in the example program `HelloJava3Db`. Making two simultaneous rotations requires combining two rotation Transform3D objects. The example rotates the cube around both the x and y-axes. Two Transform3D objects, one for each rotation, are created (lines 6 and 7). The individual rotations are specified for the two TransformGroup objects (lines 9 and 10). Then the rotations are combined by multiplying the Transform3D objects (line 11). The combination of the two transforms is then loaded into the TransformGroup object (line 12).

```
1. public BranchGroup createSceneGraph() {
2.          // Create the root of the branch graph
3.          BranchGroup objRoot = new BranchGroup();
4.
5.          // rotate object has composite transformation matrix
6.          Transform3D rotate = new Transform3D();
7.          Transform3D tempRotate = new Transform3D();
8.
9.          rotate.rotX(Math.PI/4.0d);
10.         tempRotate.rotY(Math.PI/5.0d);
11.         rotate.mul(tempRotate);
12.         TransformGroup objRotate = new TransformGroup(rotate);
13.
14.         objRotate.addChild(new ColorCube(0.4));
15.         objRoot.addChild(objRotate);
16.         return objRoot;
```

Code Fragment 1-6 Two Rotation Transformations in HelloJava3Db

Either Code Fragment 1-5 or Code Fragment 1-6 could replace Code Fragment 1-2 in HelloJava3Da to create a new program. Code Fragment 1-6 is used in `HelloJava3Db.java`. The complete example program, with the combined rotations, appears in `examples/HelloJava3D/` in file `HelloJava3Db.java`. This program is run as an application as HelloJava3Da was.

The scene graph created in `HelloJava3Db.java` is shown in Figure 1-14. The view branch graph is the same one produced in HelloJava3Da, which is constructed by SimpleUniverse and represented by the large star. The content branch graph now includes a TransformGroup in the scene graph path to the ColorCube object.

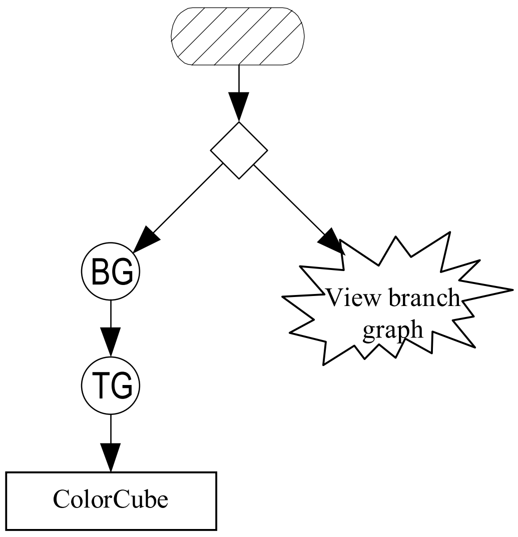


Figure 1-14 Scene Graph for HelloJava3Db Example

The image in Figure 1-15 shows the rotated ColorCube from HelloJava3Db.

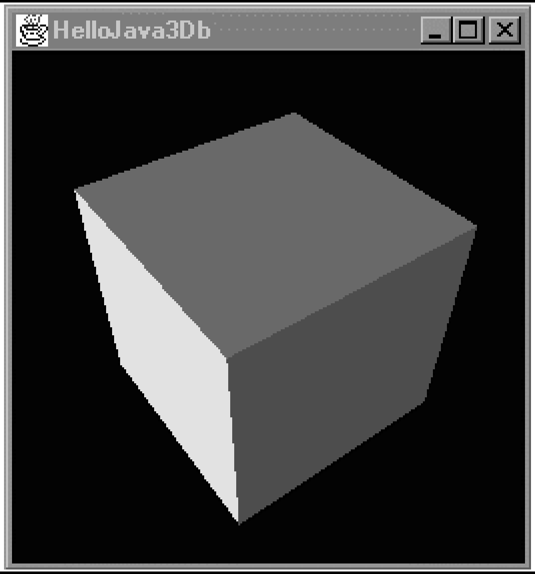


Figure 1-15 Image of the Rotated ColorCube Rendered by HelloJava3Db

1.8 Capabilities and Performance

The scene graph constructed by a Java 3D program could be used directly for rendering. However, the representation is not very efficient. The flexibility built in to each scene graph object (which has not been discussed in this tutorial) makes it a sub-optimal representation of the virtual universe. A more efficient representation of the virtual universe is used to improve rendering performance.

Java 3D has an internal representation for a virtual universe and methods for making the conversion. There are two ways to have the Java 3D system make the conversion to the internal representation. One way is to compile each branch graph. The other way is to insert the branch graph into a virtual universe to make it live. Compiling a branch graph is the subject of the next section. The effects of the conversion to the internal representation are discussed in Section 1.8.2.

1.8.1 Compiling Contents

The BranchGroup object has a compile method. Invoking this method converts the entire branch graph below the branch group to the Java 3D internal representation of the branch graph. In addition to the conversion, the internal representation may be optimized in one or more ways.

The possible optimizations are not specified by the Java 3D API. However, efficiencies can be gained in a number of ways. One of the possible optimizations is to combine TransformGroups along scene graph paths. For example, if a scene graph has two TransformGroup objects in a parent-child relationship they can be represented by one TransformGroup object. Another possibility is to combine Shape3D objects which have a static physical relationship. These types of optimizations are made possible when the capabilities are not set. Other optimizations are possible as well.

Figure 1-16 presents a conceptual representation of the conversion to a more efficient representation. The scene graph on the left-hand side of the Figure is compiled and transformed into the internal representation shown on the right-hand side of the Figure. The Figure only represents the concept of the internal representation, not how Java 3D actually performs.

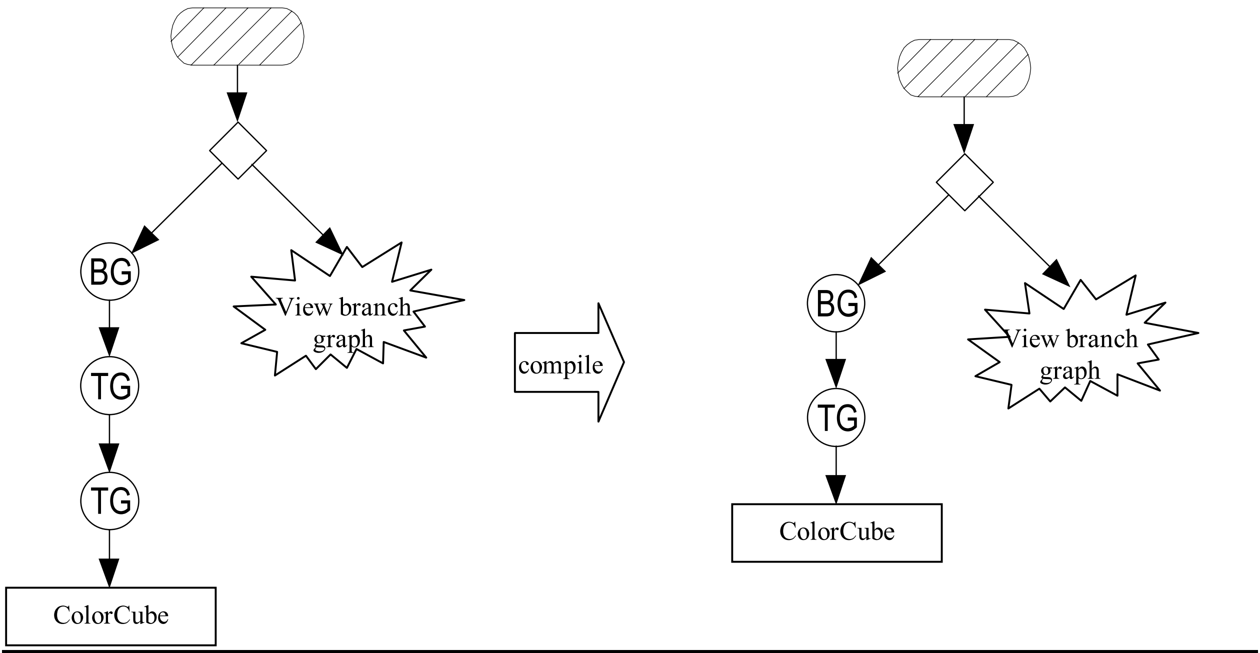


Figure 1-16 Conceptual Example of the Result of Compiling a Scene Graph

1.8.2 Capabilities

Once a branch graph is made live or compiled the Java 3D rendering system converts the branch graph to a more efficient internal representation. The most important effect of converting the scene graph to the internal representation is to improve rendering performance.

Making the transformation to the internal representation has other effects as well. One effect is to fix the value of transformations and other objects in the scene graph. Unless specifically provided for in the program, the program no longer has the capability to change the values of the scene graph objects after they are live.

There are cases when a program still needs the capability to change values in a scene graph object after it becomes live. For example, changing the value of a TransformGroup object creates animations. For this to happen, the transform must be able to change after it is live. The list of parameters than can be accessed, and in which way, is called the *capabilities* of the object.

Each SceneGraphObject has a set of capability bits. The values of these bits determine what capabilities exist for the object after it is compiled or becomes live. The set of capabilities varies by class.

SceneGraphObject Methods (partial list)

SceneGraphObject is the superclass of nearly every class used to create a scene graph including Group, Leaf, and NodeComponent. Section 1.5 presents some other SceneGraphObject methods.

`void clearCapability(int bit)`
Clear the specified capability bit.

`boolean getCapability(int bit)`
Retrieves the specified capability bit.

`void setCapability(int bit)`
Sets the specified capability bit.

As an example, to be able to read the value of the transform represented by a TransformGroup object, that capability must be set before it is either compiled or becomes live. Similarly, to be able to change the value of the transform in a TransformGroup object, its transform write capability must be set before it becomes live, or is compiled. The following reference block shows the capabilities of the non-inherited TransformGroup class. Attempting to make a change in a live or compiled object for which the proper capability is not set results in an exception.

In the next section, animations are created using a time varying rotation transformation. For this to be possible, the TransformGroup object must have its ALLOW_TRANSFORM_WRITE capability set before it either is compiled or becomes live.

TransformGroup Capabilities (partial list)

The two capabilities listed here are the only ones defined by TransformGroup. TransformGroup inherits a number of capability bits from its ancestor classes: Group and Node. Capability settings are set, reset, or retrieved using methods defined in SceneGraphObject.

`ALLOW_TRANSFORM_READ`
Specifies the TransformGroup node allows access to the transform information of its object.

`ALLOW_TRANSFORM_WRITE`
Specifies the TransformGroup node allows writing the transform information of its object.

Capabilities control access to other aspects of a TransformGroup object as well. TransformGroup object inherits capability settings from its ancestor classes: Group and Node. Some of the capabilities of Group are shown in the following reference block.

Group Capabilities (partial list)

TransformGroup inherits a number of capability bits from its ancestor classes

`ALLOW_CHILDREN_EXTEND`
Setting this capability allows children to be added to the Group node after it is compiled, or made live.

`ALLOW_CHILDREN_READ`
Setting this capability allows the references to the children of the Group node to be read after it is compiled, or made live.

`ALLOW_CHILDREN_WRITE`
Setting this capability allows the references to the children of the Group node to be written (changed) after it is compiled, or made live.

1.9 Adding Animation Behavior

In Java 3D, Behavior is a class for specifying animations of or interaction with visual objects. The behavior can change virtually any attribute of a visual object. A programmer can use a number of pre-defined behaviors or specify a custom behavior. Once a behavior is specified for a visual object, the Java 3D system updates the position, orientation, color, or other attributes, of the visual object automatically.

The distinction between animation and interaction is whether the behavior is activated in response to the passing of time or in response to user activities, respectively.

Each visual object in the virtual universe can have its own predefined behavior. In fact, a visual object may have multiple behaviors. To specify a behavior for a visual object, the programmer creates the objects that specify the behavior, adds the visual object to the scene graph, and makes the appropriate references among scene graph objects and the behavior objects.

In a virtual universe with many behaviors, a significant amount of computing power could be required just for computing the behaviors. Since both the renderer and behaviors use the same processor(s), it is possible the computational power requirement for behaviors could degrade rendering performance[4].

Java 3D allows the programmer to manage this problem by specifying a spatial boundary for a behavior to take place. This boundary is called a *scheduling region*. A behavior is not active unless the ViewPlatform's *activation volume* intersects a Behavior object's scheduling region. In other words, if there is no one in the forest to see the tree falling, it does not fall. The scheduling region feature makes Java 3D more efficient in handling a virtual universe with many behaviors.

An Interpolator is one of a number of predefined behavior classes in the core Java 3D package created which are subclasses of Behavior. Based on a time function, the Interpolator object manipulates the parameters of a scene graph object. For example, for the RotationInterpolator, manipulates the rotation specified by a TransformGroup to affect the rotation of the visual objects which are ancestors of the TransformGroup.

[4] Special graphics processors may be involved in the rendering process depending on the hardware environment and the implementation of Java 3D. Even so, it is still possible to have too many behaviors in a virtual universe to render quickly.

Figure 1-17 enumerates the steps involved in specifying an animation with an interpolator object. The five steps of Figure 1-17 form a recipe for creating an interpolation animation behavior.

1. Create a target TransformGroup
   Set the ALLOW_TRANSFORM_WRITE capability
2. Create an Alpha[5] object
   Specify the time parameters for the alpha
3. Create the interpolator object
   Have it reference the Alpha and TransformGroup objects
   Customize the behavior parameters
4. Specify a scheduling region
   Set the scheduling region for the behavior
5. Make the behavior a child of the TransformGroup

Figure 1-17 Recipe for Adding Behaviors to a Java 3D Visual Objects

1.9.1 Specifying Animation Behavior

A behavior action can be a change in location (PositionInterpolator), orientation (RotationInterpolator), size (ScaleInterpolator), color (ColorInterpolator), or transparency (TransparencyInterpolator) of a visual object. As mentioned before, Interpolators are predefined behavior classes. All of the mentioned behaviors are possible without using an interpolator; however, interpolators make creating a behavior much easier. Interpolators classes exist to provide other actions, including combinations of these actions. The details of these classes are presented in Section 5.2 and can be found in the API specification. The RotationInterpolator class is used in an example below.

RotationInterpolator Class

This class is used to specify a rotation behavior of a visual object or a group of visual objects. A RotationIterpolator object changes a TransformGroup object to a specific rotation in response to the value of an Alpha object. Since the value of an Alpha object changes over time, the rotation changes as well. A RotationInterpolator object is flexible in specification of what axis of rotation, starting angle, and ending angle.

For simple constant rotations, the RotationInterpolator object has the following constructor that can be used:

RotationInterpolator Constructor (partial list)

This class defines a behavior that modifies the rotational component of its target TransformGroup by linearly interpolating between a pair of specified angles (using the value generated by the specified Alpha object). The interpolated angle is used to generate a rotation transform.

`RotationInterpolator(Alpha alpha, TransformGroup target)`
This constructor uses default values for some parameters of the interpolator to construct a full rotation about the y-axis using the specified interpolator target TransformGroup.
parameters:
 alpha – the time varying function to reference.
 target – the TransformGroup object to modify.

[5] Alpha is a class in Java 3D for creating time varying functions. See section 1.9.2 and/or the glossary for more information.

The target TransformGroup object of an interpolator must have write capability. Information on capabilities is presented in Section 1.8.

1.9.2 Time Varying Functions: Mapping a Behavior to Time

Mapping an action to time is done using an Alpha object. The specification of the alpha object can be complex. Some basic information on the Alpha class is presented here.

Alpha Class

Alpha class objects are used to create a time varying function. The Alpha class produces a value between zero and one, inclusive. The value it produces is dependent on the time and the parameters of the Alpha object. Alpha objects are commonly used with an Interpolator behavior object to provide animations of visual objects.

There are ten parameters to Alpha, giving the programmer tremendous flexibility. Without getting into the details of each parameter, know that an instance of Alpha can easily be combined with a behavior to provide simple rotations, pendulum swings, and one-time events such as door openings, or rocket launches.

Alpha Constructor

The alpha class provides objects for converting time into an alpha value (a value in the range 0 to 1). The Alpha object is effectively a function of time that generates alpha values in the range zero to one, inclusive. A use of the Alpha object is to provide values for Interpolator behaviors. The function f(t) and the characteristics of the Alpha object are determined by user-definable parameters:

`Alpha()`
Continuous looping with a period of one second.

`Alpha(int loopCount, long increasingAlphaDuration)`
This constructor takes only the loopCount and increasingAlphaDuration as parameters and assigns the default values to all of the other parameters. The resulting Alpha object produces values starting at zero increasing to one. This repeats the number of times specified by loopCount. If loopCount is –1, the alpha object repeats indefinitely. The time it takes to get from zero to one is specified in the second parameter using a scale of milliseconds.
Parameters:
loopCount - number of times to run this alpha object; a value of -1 specifies that the alpha loops indefinitely.
increasingAlphaDuration - time in milliseconds during which alpha goes from zero to one

1.9.3 Scheduling Region

As mentioned in section 1.9, each behavior has scheduling bounds. The scheduling bounds for a behavior are set using the setSchedulingBounds method of the Behavior class.

There are a number of ways to specify a scheduling region, the simplest of which is to create a BoundingSphere object. Other options include bounding box and a bounding polytope. The BoundingSphere class is discussed below. For information on BoundingBox and BoundingPolytope, the reader is referred to the API specification.

Behavior setSchedulingBounds method

`void setSchedulingBounds(Bounds region)`
Set the Behavior's scheduling region to the specified bounds.
Parameters:
region - the bounds that contains the Behavior's scheduling region.

BoundingSphere Class

Specifying a bounding sphere is accomplished by specifying a center point and a radius for the sphere. The normal use of the bounding sphere is to use the center at (0, 0, 0). The radius is then selected large enough such that the sphere contains the visual object, including all possible locations for the object.

Bounding Sphere Constructors (partial list)

This class defines a spherical bounding region that is defined by a center point and a radius.

`BoundingSphere()`
This constructor creates a bounding sphere centered at the origin (0, 0, 0) with a radius of 1.

`BoundingSphere(Point3d center, double radius)`
Constructs and initializes a BoundingSphere using the specified center point and radius.

1.9.4 Behavior Example: HelloJava3Dc

Code Fragment 1-7 shows a complete example of using one of the interpolator classes to create an animation. The animation created in this code is a continuous rotation with a total rotation time of four seconds. Code Fragment 1-7 correlates with the interpolation animation recipe given in Figure 1-17.

Step 1 of the recipe is to create a TransformGroup object to modify at runtime. The target TransformGroup object of an interpolator must have write capability set. The TransformGroup object named objSpin is created on line 7. The capability of objSpin is set on line 8 of Code Fragment 1-7.

Step 2 is to create an alpha object to drive the interpolator. In the simple example shown in Code Fragment 1-7 the Alpha object, rotationAlpha, is used to specify a continuous rotation. The two parameters specified on line 16 of Code Fragment 1-7 are the number of loop iterations and the time for one cycle. The value “-1” for the loop count specifies continuous looping. The time is specified in milliseconds. The value 4000 used in the program is 4000 milliseconds which is 4 seconds. Therefore, the behavior is to rotate once every four seconds.

Step 3 of the recipe is to create the interpolator object. The RotationInterpolator object rotate is created on lines 21 and 22. The interpolator must have references to the target transform and alpha objects. This is accomplished in the constructor. In this example, the RotationInterpolator’s default behavior is used. The default behavior of the RotationInterpolator is to make a full rotation about the y-axis.

Step 4 is to specify a scheduling region. In Code Fragment 1-7, a BoundingSphere object is used with its default values. The BoundingSphere object is created on line 25. The sphere is set as the bounds for the behavior on line 26.

The final step, step 5, makes the behavior a child of the TransformGroup. This is accomplished on line 27.

```
1. public BranchGroup createSceneGraph() {
2.         // Create the root of the branch graph
3.         BranchGroup objRoot = new BranchGroup();
```

```java

        // Create the transform group node and initialize it to the
        // identity. Add it to the root of the subgraph.
        TransformGroup objSpin = new TransformGroup();
        objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        objRoot.addChild(objSpin);

        // Create a simple shape leaf node, add it to the scene graph.
        // ColorCube is a Convenience Utility class
        objSpin.addChild(new ColorCube(0.4));

        // create time varying function to drive the animation
        Alpha rotationAlpha = new Alpha(-1, 4000);

        // Create a new Behavior object that performs the desired
        // operation on the specified transform object and add it into
        // the scene graph.
        RotationInterpolator rotator =
                new RotationInterpolator(rotationAlpha, objSpin);

        // a bounding sphere specifies a region a behavior is active
        BoundingSphere bounds = new BoundingSphere();
        rotator.setSchedulingBounds(bounds);
        objSpin.addChild(rotator);

        return objRoot;
    } // end of createSceneGraph method
```

Code Fragment 1-7 createSceneGraph method with RotationInterpolator Behavior

Code Fragment 1-7 is used with some previous code fragments to form the example program `HelloJava3Dc`. `HelloJava3Dc.java` can be found in the `examples/HelloJava3D/` directory and can be run as an application. The running application renders the ColorCube with the behavior of rotating once every four seconds.

The HelloJava3Dc creates the scene graph shown in Figure 1-18. The rotation object is both a child of the TransformGroup **objSpin** and has a reference to it. While this relationship appears to violate the no-cycles restriction of the scene graph, it does not. Recall that reference arcs (dashed arrows) are not part of the scene graph. The dashed line from the Behavior to the TransformGroup is that reference.

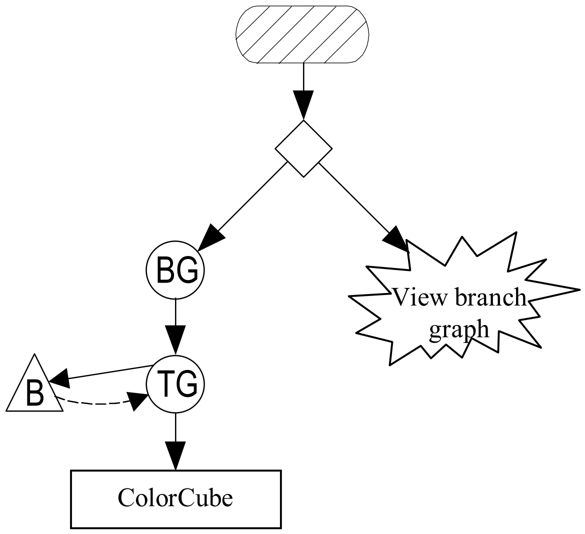


Figure 1-18 Scene Graph for HelloJava3Dc Example

The image in Figure 1-19 shows one frame of the spinning ColorCube from HelloJava3Dc.

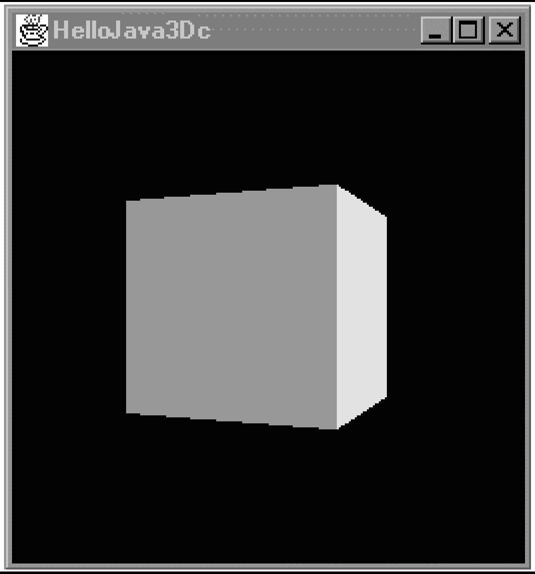


Figure 1-19 An Image of the ColorCube in Rotation as Rendered by HelloJava3Dc

1.9.5 Transformation and Behavior Combination Example: HelloJava3Dd

Of course, you can combine behaviors with the rotation transforms of the previous examples. In `HelloJava3Dd.java` this is done. In the content branch graph, there are objects named **objRotate** and **objSpin**, which distinguish between the static rotation and the continuous spin (rotation behavior) of the cube object respectively. The code is shown in Code Fragment 1-8. The resulting scene graph is in Figure 1-20.

```
1. public BranchGroup createSceneGraph() {
2.    // Create the root of the branch graph
```

```
   BranchGroup objRoot = new BranchGroup();

   // rotate object has composite transformation matrix
   Transform3D rotate = new Transform3D();
   Transform3D tempRotate = new Transform3D();

   rotate.rotX(Math.PI/4.0d);
   tempRotate.rotY(Math.PI/5.0d);
   rotate.mul(tempRotate);

   TransformGroup objRotate = new TransformGroup(rotate);

   // Create the transform group node and initialize it to the
   // identity.  Enable the TRANSFORM_WRITE capability so that
   // our behavior code can modify it at runtime.  Add it to the
   // root of the subgraph.
   TransformGroup objSpin = new TransformGroup();
   objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

   objRoot.addChild(objRotate);
   objRotate.addChild(objSpin);

   // Create a simple shape leaf node, add it to the scene graph.
   // ColorCube is a Convenience Utility class
   objSpin.addChild(new ColorCube(0.4));

   // Create a new Behavior object that performs the desired
   // operation on the specified transform object and add it into
   // the scene graph.
   Transform3D yAxis = new Transform3D();
   Alpha rotationAlpha = new Alpha(-1, 4000);

   RotationInterpolator rotator =
       new RotationInterpolator(rotationAlpha, objSpin, yAxis,
                                  0.0f, (float) Math.PI*2.0f);

   // a bounding sphere specifies a region a behavior is active
   // create a sphere centered at the origin with radius of 1
   BoundingSphere bounds = new BoundingSphere();
   rotator.setSchedulingBounds(bounds);
   objSpin.addChild(rotator);

       return objRoot;
} // end of createSceneGraph method of HelloJava3Dd
```

Code Fragment 1-8 Content Branch for Rotated Spinning ColorCube of HelloJava3Dd

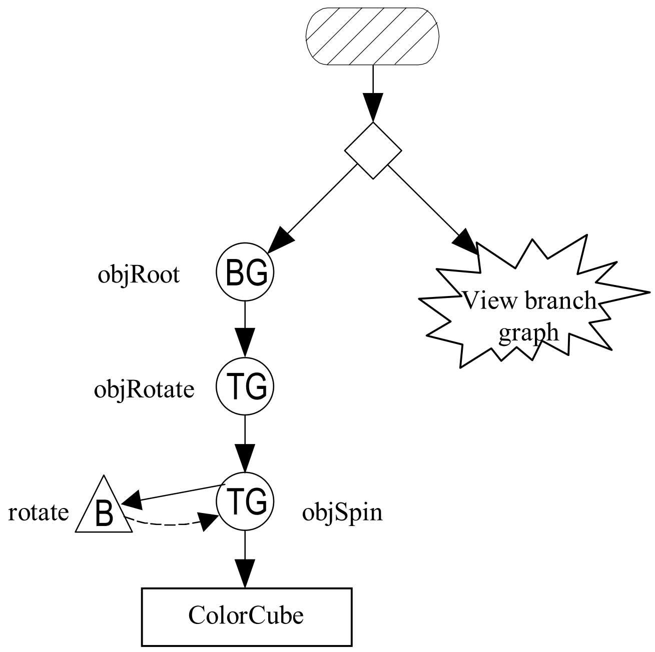


Figure 1-20 Scene Graph for HelloJava3Dd Example

The image in Figure 1-21 shows one frame of the spinning and rotated ColorCube from HelloJava3Dd.

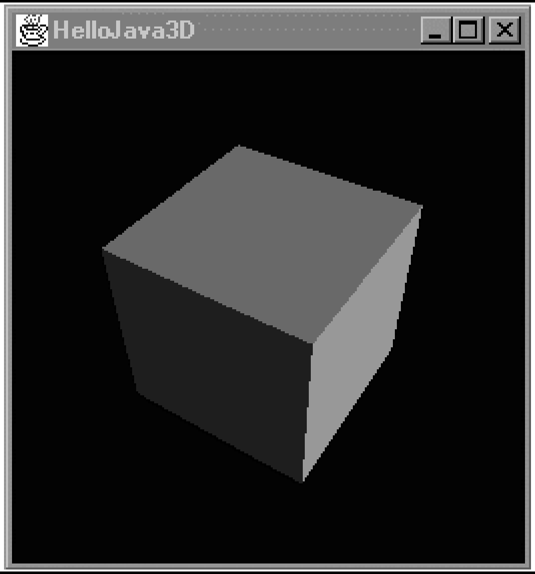


Figure 1-21 An Image of the ColorCube in Rotation as Rendered by HelloJava3Dd

1.10 Chapter Summary

This chapter begins assuming the reader knows nothing about Java 3D. Through the course of the chapter the reader is introduced to some of the most important classes in the Java 3D API. Explanation is given for how these classes, and classes from other packages, are used to assemble a scene graph. The scene graph, which describes a virtual universe, and how the view is to be rendered, is discussed in some detail. The SimpleUniverse utility class is used to create a series of example programs that demonstrate the simplest Java 3D program, a simple transformation, a combination of transformations, behavior, and combining transformation and behavior. Later in the chapter come explanations of capabilities of objects and the compiling of branch graphs.

1.11 Self Test

On this page are a few exercises intended to test and enhance your understanding of the material presented in this chapter. The solutions to some of these exercises are given in Appendix C.

1. In the HelloJava3Db program, which combines two rotations in one TransformGroup, what would be the difference if you reverse the order of the multiplication in the specification of the rotation? Alter the program to see if your answer is correct. There are only two lines of code to change to accomplish this.

2. In the HelloJava3Dd program, what would be the difference if you reverse the order of the Transform Nodes above the ColorCube in the content branch graph? Alter the program to see if your answer is correct.

3. In search of performance improvements, a programmer might want to make the scene graph smaller[6]. Can you combine the rotation and the spin target transform of HelloJava3Dd into one TransformGroup object?

4. Translate the ColorCube 1 unit in the Y dimension and rotate the cube. You can use HelloJava3Db as a starting point. The code that follows the question shows the specification of a translation transformation. Try the transformation in the opposite order. Do you expect to see a difference in the results? If so, why? If not, why not? Try it and compare your expectations to the actual results.

```
   Transform3D translate = new Transform3D();
   Vector3f vector = new Vector3f(0.0f, 1.0f, 0.0f);
   translate.setTransform(vector);
```

5. In HelloJava3Dc, the bounding sphere has a radius of 1 meter. Is this value larger or smaller than it needs to be? What is the smallest value that would guarantees the cube will be rotating if it is in view? Experiment with the program to verify your answers. The following line of code can be used to specify a bounding sphere. In this line, the Point3D object specifies the center, followed by the radius.

```
   BoundingSphere bounds =
                new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
```

[6] Performance is directly related to scene graph size. The most effective change is to reduce the number of Shape3D objects in a scene graph. Refer to the Java 3D performance whitepaper available at java.sun.com/docs.

6. The example programs give sufficient information for assembling a virtual universe with multiple color cubes. How do you construct such a scene graph? In what part of the code would this be accomplished?