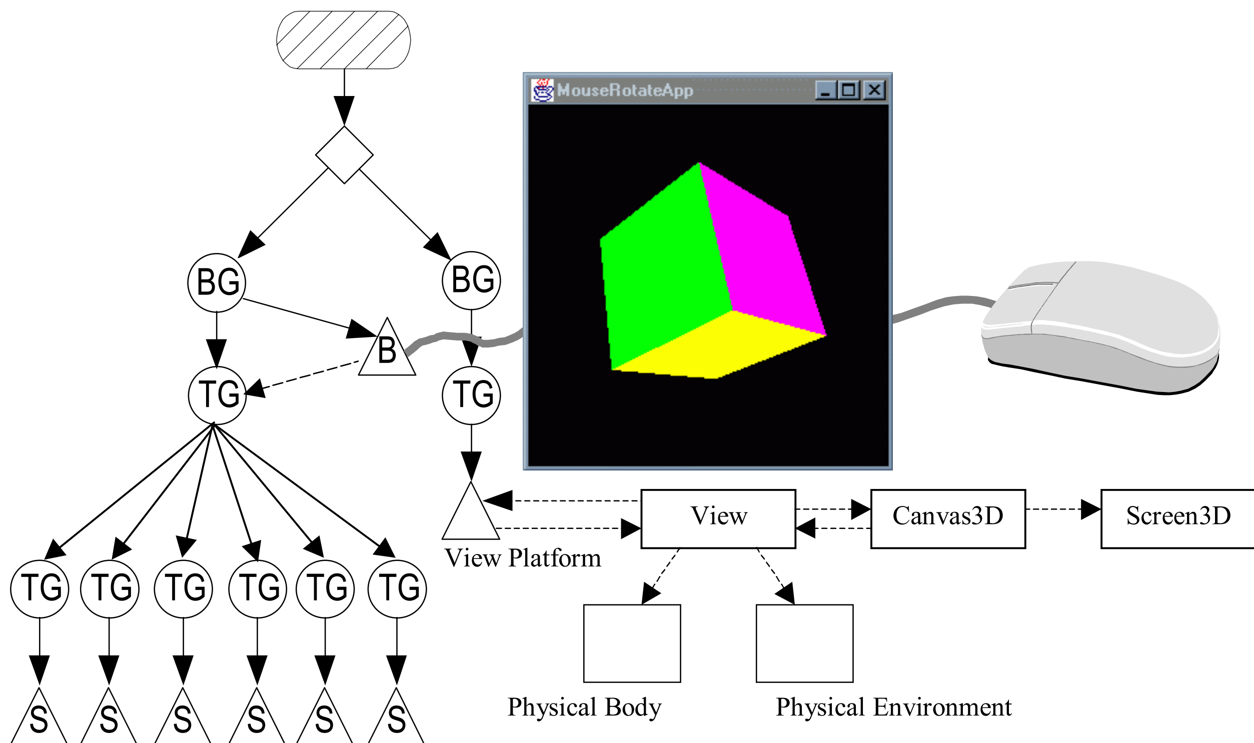
Getting Started with the Java 3D™ API

Chapter 4
Interaction



Dennis J Bouvier

Tutorial version 1.6 (Java 3D API v 1.2)

© 2000-2001 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A
All Rights Reserved.

The information contained in this document is subject to change without notice.

SUN MICROSYSTEMS PROVIDES THIS MATERIAL "AS IS" AND MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SUN MICROSYSTEMS SHALL NOT BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS IN CONNECTION WITH THE FURNISHING, PERFORMANCE OR USE OF THIS MATERIAL, WHETHER BASED ON WARRANTY, CONTRACT, OR OTHER LEGAL THEORY).

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY MADE TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Some states do not allow the exclusion of implied warranties or the limitations or exclusion of liability for incidental or consequential damages, so the above limitations and exclusion may not apply to you. This warranty gives you specific legal rights, and you also may have other rights which vary from state to state.

Permission to use, copy, modify, and distribute this documentation for NON-COMMERCIAL purposes and without fee is hereby granted provided that this copyright notice appears in all copies.

Java, JavaScript, Java 3D, HotJava, Sun, Sun Microsystems, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

Table of Contents

Chapter 4:

INTERACTION 4-1

4.1 BEHAVIOR: THE BASE FOR INTERACTION AND ANIMATION 4-1
4.1.1 Applications of Behavior 4-2
4.1.2 Overview of Behavior Classes 4-3
4.2 BEHAVIOR BASICS 4-3
4.2.1 Writing a Behavior Class 4-4
4.2.2 Using a Behavior Class 4-7
4.2.3 Behavior Class API 4-10
4.3 WAKEUP CONDITIONS: HOW BEHAVIORS ARE TRIGGERED 4-12
4.3.1 WakeupCondition 4-13
4.3.2 WakeupCriterion 4-13
4.3.3 Specific WakeupCriterion Classes 4-14
4.3.4 WakeupCondition Composition 4-24
4.4 BEHAVIOR UTILITY CLASSES FOR KEYBOARD NAVIGATION 4-25
4.4.1 Simple KeyNavigatorBehavior Example Program 4-26
4.4.2 KeyNavigatorBehavior and KeyNavigator Classes 4-28
4.5 UTILITY CLASSES FOR MOUSE INTERACTION 4-29
4.5.1 Using the Mouse Behavior Classes 4-29
4.5.2 Mouse Behavior Foundation 4-31
4.5.3 Specific Mouse Behavior Classes 4-32
4.5.4 Mouse Navigation 4-35
4.6 PICKING 4-36
4.6.1 Using Picking Utility Classes 4-38
4.6.2 Java 3D API Core Picking Classes 4-40
4.6.3 General Picking Package Classes 4-50
4.6.4 Specific Picking Behavior Classes 4-53
4.7 CHAPTER SUMMARY 4-56
4.8 SELF TEST 4-56

List of Figures

Figure 4-1 Hierarchy of Subclasses of Behavior 4-4
Figure 4-2 Recipe for Writing a Custom Behavior Class 4-5
Figure 4-3 Recipe for Using a Behavior Class.. 4-8
Figure 4-4 Scene Graph Diagram of the Content Branch Graph Created in SimpleBehaviorApp.java. .. 4-8
Figure 4-5 An Alternative Scene Graph Placement for the Behavior Object in SimpleBehaviorApp. .. 4-10
Figure 4-6 API Class Hierarchy for Behavior... 4-11
Figure 4-7 The Java 3D API Class Hierarchy for WakeupCondition and Related Classes................... 4-13
Figure 4-8 The Basic View Branch Graph Showing the View Platform Transform............................. 4-26
Figure 4-9Recipe for Using the KeyNavigatorBehavior Utility Class 4-27
Figure 4-10 Recipe for Using Mouse Behavior Classes 4-30
Figure 4-11 Projection of PickRay in the Virtual World... 4-36
Figure 4-12 Scene Graph Diagram for a Cube Composed of Discrete Shape3D Plane Objects. 4-37
Figure 4-13 Recipe for Using Mouse Picking Utility Classes 4-39
Figure 4-14 PickShape Hierarchy 4-41
Figure 4-15 Parameters of PickCone pick shapes.. 4-44
Figure 4-16 Parameters of PickCylinder pick shapes 4-46

List of Tables

Table 4-1 Applications of Behavior Categorized by Stimulus and Object of Change 4-2
Table 4-2 The 14 Specific WakeupCriterion Classes.. 4-14
Table 4-3 KeyNavigatorBehavior Movements 4-28
Table 4-4 Summary of Specific MouseBehavior Classes.. 4-29
Table 4-5 Selection of PickShape 4-41

List of Code Fragments

Code Fragment 4-1 SimpleBehavior Class in SimpleBehaviorApp.java.. 4-6
Code Fragment 4-2 CreateSceneGraph Method in SimpleBehaviorApp.java... 4-8
Code Fragment 4-3 Outline of OpenBehavior Class, an Example of Coordinated Behavior Classes.... 4-17
Code Fragment 4-4 Code using OpenBehavior and CloseBehavior, Coordinated Behavior Classes 4-17
Code Fragment 4-5 Using the KeyNavigatorBehavior Class (part 1) 4-27
Code Fragment 4-6 Using the KeyNavigatorBehavior Class (part 2) 4-28
Code Fragment 4-7 Using the MouseRotate Utility Class.. 4-30
Code Fragment 4-8 Using Mouse Behavior Classes for Interactive Navigation of the Virtual World. . 4-36
Code Fragment 4-9 The createSceneGraph Method of the MousePickApp Example Program. 4-40

List of Reference Blocks

Behavior Method Summary ... 4-12
ViewPlatform Method Summary (partial list) ... 4-12
WakeupCondition Method Summary ... 4-13
WakeupCriterion Method Summary ... 4-14
WakeupOnActivation Constructor Summary ... 4-15
WakeupOnAWTEvent Constructor Summary ... 4-15
WakeupOnAWTEvent Method Summary ... 4-16
WakeupOnBehaviorPost Constructor Summary ... 4-16
WakeupOnBehaviorPost Method Summary ... 4-16
WakeupOnCollisionEntry Constructor Summary ... 4-18
WakeupOnCollisionExit Constructor Summary ... 4-19
WakeupOnCollisionExit Method Summary ... 4-19
WakeupOnCollisionMovement Constructor Summary ... 4-20
WakeupOnCollisionMovement Method Summary ... 4-20
WakeupOnDeactivation Constructor Summary ... 4-21
WakeupOnElapsedFrames Constructor Summary ... 4-21
WakeupOnElapsedFrames Method Summary ... 4-21
WakeupOnElapsedTime Constructor Summary ... 4-22
WakeupOnElapsedTime Method Summary ... 4-22
WakeupOnSensorEntry Constructor Summary ... 4-22
WakeupOnSensorEntry Method Summary ... 4-22
WakeupOnSensorExit Constructor Summary ... 4-23
WakeupOnSensorExit Method Summary ... 4-23
WakeupOnTransformChange Constructor Summary ... 4-23
WakeupOnTransformChange Method Summary ... 4-23
WakeupOnViewPlatformEntry Constructor Summary ... 4-24
WakeupOnViewPlatformEntry Method Summary ... 4-24
WakeupOnViewPlatformExit Constructor Summary ... 4-24
WakeupOnViewPlatformExit Method Summary ... 4-24
WakeupAnd Constructor Summary ... 4-25
WakeupOr Constructor Summary ... 4-25
WakeupAndOfOrs Constructor Summary ... 4-25
WakeupOrOfAnds Constructor Summary ... 4-25
KeyNavigatorBehavior Constructor Summary ... 4-29
Package: com.sun.j3d.utils.behaviors.keyboard Extends: Behavior ... 4-29
KeyNavigatorBehavior Method Summary ... 4-29
MouseBehavior Method Summary ... 4-31
Interface MouseBehaviorCallback Method Summary ... 4-32
MouseRotate Constructor Summary ... 4-32
Package: com.sun.j3d.utils.behaviors.mouse Extends: MouseBehavior ... 4-32
MouseRotate Method Summary ... 4-33
MouseTranslate Constructor Summary ... 4-33
Package: com.sun.j3d.utils.behaviors.mouse Extends: MouseBehavior ... 4-33
MouseTranslate Method Summary ... 4-34
MouseZoom Constructor Summary ... 4-34
Package: com.sun.j3d.utils.behaviors.mouse Extends: MouseBehavior ... 4-34
MouseZoom Method Summary ... 4-35

Node Method (partial list) 4-38
Node Capabilities Summary (partial list) 4-38
PickShape 4-42
PickBounds Constructor Summary 4-42
PickBounds Method Summary 4-42
PickPoint Constructor Summary 4-42
PickPoint Method Summary 4-43
PickRay Constructor Summary 4-43
PickRay Method Summary 4-43
PickSegment Constructor Summary 4-43
PickSegment Method Summary 4-44
PickCone Method Summary 4-44
PickConeRay Constructor Summary 4-45
PickConeRay Method Summary 4-45
PickConeSegment Constructor Summary 4-45
PickConeSegment Method Summary 4-45
PickCylinder Method Summary 4-46
PickCylinderRay Constructor Summary 4-46
PickCylinderRay Method Summary 4-47
PickCylinderSegment Constructor Summary 4-47
PickCylinderRay Method Summary 4-47
SceneGraphPath Overview 4-48
SceneGraphPath Constructor Summary 4-48
SceneGraphPath Method Summary (partial list) 4-49
BranchGroup and Locale picking methods for use with PickShape 4-50
PickMouseBehavior Method Summary 4-50
Package: `com.sun.j3d.utils.behaviors.picking` Extends: `Behavior` 4-50
PickObject Constructor Summary 4-51
PickObject Method Summary (partial list) 4-51
PickObject Method Summary (partial list - continued) 4-51
Interface PickingCallback Method Summary 4-52
Intersect Constructor Summary 4-52
Intersect Method Summary (partial list) 4-53
Intersect Method Summary (partial list - continued) 4-53
PickRotateBehavior Constructor Summary 4-54
PickRotateBehavior Method Summary 4-54
PickTranslateBehavior Constructor Summary 4-55
PickTranslateBehavior Method Summary 4-55
PickZoomBehavior Constructor Summary 4-56
PickZoomBehavior Method Summary 4-56

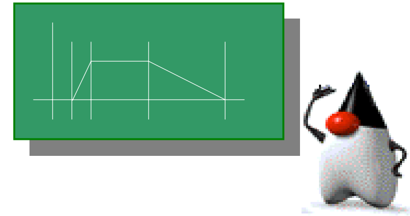
Preface to Chapter 4

This document is one part of a tutorial on using the Java 3D API. You should be familiar with Java 3D API basics to fully appreciate the material presented in this Chapter. Additional chapters and the full preface to this material are presented in the Module 0 document available at: `http://java.sun.com/products/javamedia/3d/collateral`

Cover Image

The cover image represents the interaction possible in Java 3D through the use of the mouse. The mouse appears to be connected to the window with a visual, the cube, but the wire proceeds to the scene graph diagram to the Behavior object. The scene graph diagram represents a cube created with six individual shape objects (each of the six faces of the cube is a plane – of course you don't have to do this). The image of the application is from an early version of `MouseRotateApp.java`, an example program included in the examples jar available with this tutorial. The image of the mouse is from the clip art distributed with Microsoft Office 97.

CHAPTER 4
Interaction



Chapter Objectives

After reading this chapter, you'll be able to:

- Appreciate the Behavior Class as the foundation for interaction and animation
- Create custom behavior classes
- Incorporate behavior objects into virtual worlds to provide interaction
- Use utility classes for keyboard navigation
- Use utility classes for mouse interaction
- Use utility picking classes

In the previous chapters of the tutorial, the Java 3D virtual universes are almost all static. For Java 3D worlds to be more interesting, and more useful, interaction and animation are necessary. Interaction is when the imagery changes in response to user action. Animation is defined as changes in the imagery without direct user action, and usually corresponds with the passage of time.

In Java 3D, both interaction and animations are specified through the use of the Behavior class. This chapter introduces the Behavior class and explains its use in interactive programs. The next chapter, Animation, continues with animation examples and explanations.

4.1 Behavior: the Base for Interaction and Animation

Both interaction and animation are specified with Behavior objects. The Behavior class is an abstract class that provides the mechanism to include code to change the scene graph. The Behavior class, and its descendants, are links to user code providing changes to the graphics and sounds of the virtual universe.

The purpose of a Behavior object in a scene graph is to change the scene graph, or objects in the scene graph, in response to some stimulus. A stimulus can be the press of a key, a mouse movement, the collision of objects, the passage of time, some other event, or a combination of these. Changes produced include adding objects to the scene graph, removing objects from the scene graph, changing attributes of objects in the scene graph, rearranging objects in the scene graph, or a combination of these. The possibilities are only limited by the capabilities of the scene graph objects.

4.1.1 Applications of Behavior

Since a behavior is a link between a stimulus and an action, considering all the combinations of possible stimuli and possible actions is to consider the many applications of Behavior objects. The following table surveys the realm of possibilities with Behavior, listing possible stimuli down the left column and possible changes across the top.

The table does not list all possible applications of Behavior, only the simple ones (one stimulus results in one change). Some combinations of stimulus and change only make sense in a specific setting; these are listed as 'application specific'. Furthermore, combinations of stimuli and combinations of actions are possible.

Table 4-1 Applications of Behavior Categorized by Stimulus and Object of Change

	object of change			
stimulus (reason for change)	TransformGroup (visual objects change orientation or location)	Geometry (visual objects change shape or color)	Scene Graph (adding, removing, or switching objects)	View (change viewing location or direction)
user	interaction	application specific	application specific	navigation
collisions	visual objects change orientation or location	visual objects change appearance in collision	visual objects disappear in collision	View changes with collision
time	animation	animation	animation	animation
View location	billboard	level of detail (LOD)	application specific	application specific

In Table 4-1 some of the possible behaviors are spelled out. For example, collision actions are described. Others, such as billboard or level of detail (LOD) behaviors, may not be familiar to you. Below are some quick explanations.

The chart does not include all applications of Behavior; combinations of stimuli and/or changes are not shown. *Picking* is also implemented using behaviors but is not listed in the table. Although listed in Table 4-1 and implemented in Java 3D API, collision detection is not addressed in this tutorial.

Natural things, such as trees, take a tremendous amount of geometry to accurately represent all of the branches, leaves and bark structure. One alternative is to use a textured polygon instead of the geometry. This technique is sometime referred to as the billboard approach. This is especially true when a behavior is used to automatically orient the textured polygon orthogonal to the viewer such that only the front textured face is viewed. This orienting behavior is called *billboard behavior*.

The billboard approach is effective when the object to be represented by the texture is distant so that the individual parts of the visual object represented by the texture would not easily be distinguished. For the tree example, if the viewer is so distant that branches are hardly distinguishable, it is hardly worth the memory and computation requirements to represent each leaf of the tree. This technique is recommended for any application requiring visually complex objects in a distance. However, if the viewer were able to

approach the billboard, at some distance the lack of depth of the textured polygon would be detected by the viewer.

The *level of detail* (LOD) behavior has a related application. With LOD, visually complex objects are represented by multiple visual objects of varying levels of detail (hence the name). The visual object representation with the least detail is used when the viewer is far away. The most detailed representation is used when the viewer is close. The LOD behavior automatically switches between the representations based on the objects distance to the viewer.

Both the billboard and level of detail behaviors correspond to classes extended from Behavior which implement these common applications. Other specializations of behavior are possible and several are listed in Figure 4-1. For example, there are several MouseBehavior classes that manipulate a transform in response to mouse movements. Normally the view transform is changed by the mouse behavior to change the view in response to mouse actions.

Also note how the behaviors can chain. For example, mouse movements or key strokes can be used to change the view. In response to the movement of the view, billboard, level of detail, and/or other behaviors may take place. Fortunately, each behavior is specified separately.

Animation Versus Interaction

Since the distinction between animation and interaction used in this tutorial is fairly fine, here is an example to help clarify this distinction. If a user navigates in a program where such a behavior is provided, the view platform will move in response to the keyboard and/or mouse events. The motion of the view platform is an interaction because it is the direct result of the user action. However, other things may change as a result of the view platform motion (e.g., billboard and LOD behaviors). Changes as a result of the view platform motion are indirectly caused by the user and are therefore animations.

4.1.2 Overview of Behavior Classes

The following figure, Figure 4-1, shows specializations of behavior made in the Java 3D API core and utility packages. User defined specializations of Behavior are also possible and are only limited in functionality by the programmer's imagination. This module of the tutorial covers each of the classes in Figure 4-1. This chapter covers the shaded classes in the figure; Chapter 5 covers the remaining classes. This figure does not present the total coverage of the Java 3D API in Chapters 4 and 5; each chapter presents more than the classes in this figure.

4.2 Behavior Basics

As explained in the previous section, Behavior classes are used in many Java 3D applications and in many ways. It is important to understand the workings and programming considerations of the behavior class. This section explains the Behavior class, gives a recipe for programming a custom behavior class, and gives a simple example application using a behavior class.

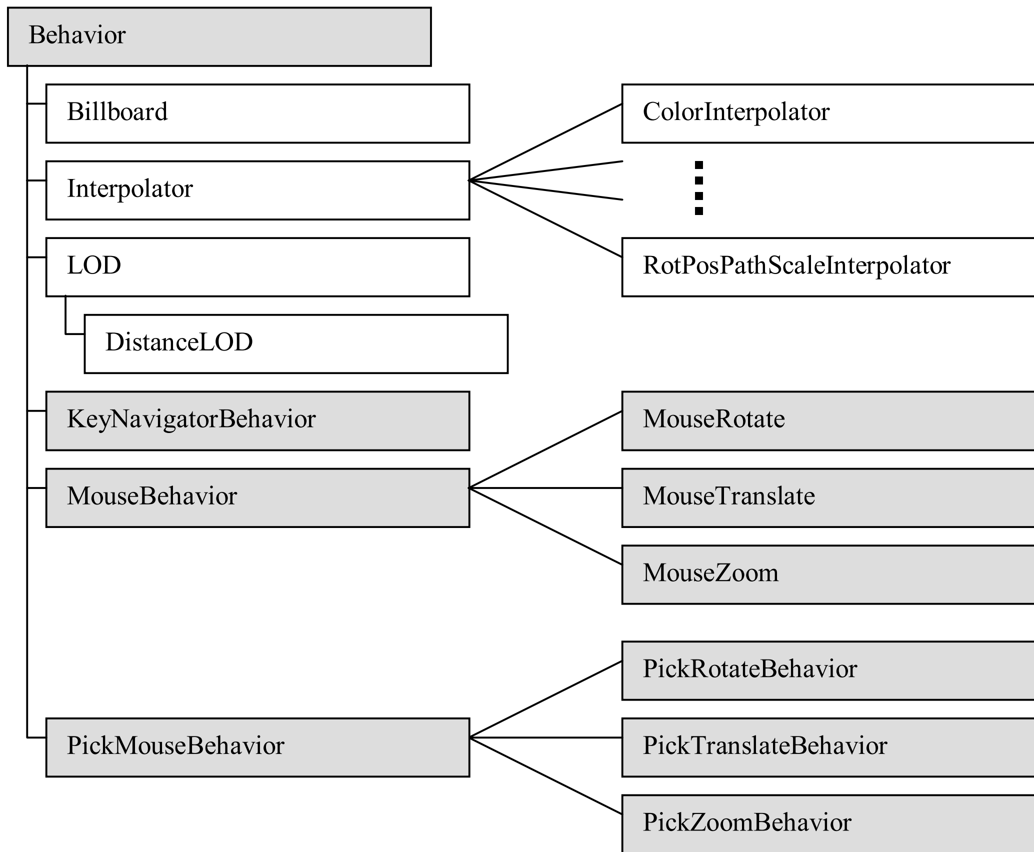


Figure 4-1 Hierarchy of Subclasses of Behavior

4.2.1 Writing a Behavior Class

This section explains how to write a custom behavior class. You know from Section 4.1 that there are behavior classes you can use without writing a class. However, in seeing how to create a Behavior class you learn how behaviors work. So even if you only plan to use a behavior class, you might want to read this section. Also, the class written in this section is used in the next section. (If you don't plan to write a Behavior class you can skip this section for now.)

Mechanics of Behaviors

A custom behavior class implements the initialization and processStimulus methods from the abstract Behavior class. Of course, the custom behavior class also has at least one constructor and may have other methods as well.

Most behaviors will act on a scene graph object to affect the behavior. In Table 4-1, the object a behavior acts upon is refered to as the *object of change*. It is through this object, or objects, that the behavior affects the virtual world. While it is possible to have a behavior that does not have an object of change, most do.

The behavior needs a reference to its object(s) of change to be able to make the behavioral changes. The constructor can be used to set the reference to the object of change. If it does not, another method in the custom behavior class must store this information. In either case, the reference is made at the time the scene graph is being constructed, which is the first computation of the behavior.

The initialization method is invoked when the scene graph containing the behavior class becomes live. The initialization method is responsible for setting the initial trigger event for the behavior and setting the initial condition of the state variables for the behavior. The trigger is specified as a WakeupCondition object, or a combination of WakeupCondition objects.

The processStimulus method is invoked when the trigger event specified for the behavior occurs. The processStimulus method is responsible for responding to the event. As many events may be encoded into a single WakeupCondition object (e.g., a variety of keyboard actions may be encoded in a WakeupOnAWTEvent), this includes decoding the event. The processStimulus method responds to the stimulus, usually by changing its object of change, and, when appropriate, resets the trigger.

The information in this section, Mechanics of Behaviors, is summarized in a recipe for writing custom behavior classes. Figure 4-2 presents this recipe.

---

1. write (at least one) constructor
   store a reference to the object of change
2. override `public void initialization()`
   specify initial wakeup criteria (trigger)
3. override `public void processStimulus()`
   decode the trigger condition
   act according to the trigger condition
   reset trigger as appropriate

Figure 4-2 Recipe for Writing a Custom Behavior Class

The recipe of Figure 4-2 shows the basic steps for creating a custom behavior class. Complex behaviors may require more programming than is described in the recipe. Using a behavior object is another issue and is discussed in Section 4.2.2. But before using a behavior, this recipe is used in creating the following example custom behavior.

Example Custom Behavior Class: SimpleBehavior

As an example of using the custom behavior class recipe of Figure 4-2, this section goes through the process of writing a custom behavior class. For the example custom behavior, the class will implement a simple behavior of making something rotate in response to a keyboard key press.

To create such a behavior class, all that is needed is a reference to a TransformGroup (the object of change for this class), and an angle variable. In response to a key press, the angle variable is changed and the angle of the target TransformGroup is set to the angle. Since the behavior will act on a TransformGroup object, what is being rotated is not an issue.

To create this class nothing more than the three essential programming ingredients listed in the recipe are needed: a constructor, the initialization method, and the processStimulus method. The constructor will store a reference to the TransformGroup object of change. The initialization method sets the initial trigger to WakeOnAWTEvent, and sets the rotation angle to zero. As mentioned above, the stimulus to a behavior is specified as a WakeupCondition object. Section 4.3 introduces WakeupCondition classes.

Since there is only one possible triggering wakeup condition, the processStimulus method does not decode the triggering wakeup condition. It is possible to further decode the key press event to determine which key, or combination of keys, was pressed.

The processStimulus method always increments the angle variable, then uses it to adjust the TransformGroup object of change. The last job of the processStimulus method is to reset the trigger. In this example, the trigger is always reset to a key press. Behaviors can change their trigger event over

time for changing behaviors (another reason for having to decode the trigger event), or not set another trigger for one time behaviors.

Code Fragment 4-1 presents the SimpleBehavior class which is an implementation of the described custom behavior class. The import statements listed in Code Fragment 4-1 are necessary for the behavior class. The java.awt.event import is necessary for the keyboard interaction. The java.util.eumeration import is necessary for decoding the WakeupCondition; and therefore necessary for virtually any custom behavior class. The normal Java 3D API import statements are needed in addition to the listed import statements.

Code Fragment 4-1 is annotated with the step numbers from the recipe.

```
1.   import java.awt.event.*;
2.   import java.util.Enumeration;
3.
4.   //   SimpleBehaviorApp renders a single, rotated cube.
5.
6.   public class SimpleBehaviorApp extends Applet {
7.
8.      public class SimpleBehavior extends Behavior{
9.
10.        private TransformGroup targetTG;
11.        private Transform3D rotation = new Transform3D();
12.        private double angle = 0.0;
13.
14.        // create SimpleBehavior - set TG object of change
15.  ❶     SimpleBehavior(TransformGroup targetTG){
16.            this.targetTG = targetTG;
17.        }
18.
19.        // initialize the Behavior
20.        //     set initial wakeup condition
21.        //     called when behavior becomes live
22.  ❷     public void initialize(){
23.            // set initial wakeup condition
24.            this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
25.        }
26.
27.        // called by Java 3D when appropriate stimulus occurs
28.  ❸     public void processStimulus(Enumeration criteria){
29.            // do what is necessary in response to stimulus
30.            angle += 0.1;
31.            rotation.rotY(angle);
32.            targetTG.setTransform(rotation);
33.            this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
34.        }
35.
36.     } // end of class SimpleBehavior
```

Code Fragment 4-1 SimpleBehavior Class in SimpleBehaviorApp.java

This class is used in the `SimpleBehaviorApp` example found in the `examples/Interaction` directory.

This class only demonstrates the basic programming necessary for this simple behavior. Enhancements to this custom class are possible. For example, the angle of rotation and/or the axis of rotation could be set

by class methods. The behavior class could be further customizable with a method for setting a specific key, or set of keys, that it will respond to.

Another definite improvement in the class would prevent overflow of the angle variable. In the current class, the value of angle could grow without bound even though values of 0.0 to 2Π are all that is necessary. Although unlikely, it is possible for this variable to overflow and cause a run time exception.

Programming Pitfalls of Writing Behavior Classes

In the three steps of the custom behavior class recipe, the two most likely programming mistakes are:

- forgetting to set and reset the behavior trigger, and
- not returning from the behavior class methods.

Obviously, if an initial trigger is not set in the initialization method, the behavior will never be invoked. A little less obvious is that the trigger must be set again in the processStimulus method if a repeat behavior is desired.

Since both the initialization and processStimulus methods are called by the Java 3D system, they must return to allow the rendering to continue. For example, if a spinning behavior were desired, the angle and the TransformGroup would need to be updated periodically. If your behavior implemented this behavior without spawning a thread, nothing further would be rendered. Also, there is a much better way to achieve this type of behavior[1].

4.2.2 Using a Behavior Class

Finding or writing the appropriate behavior class for your application is the beginning of writing an interactive Java 3D program. This section covers the programming issues in adding behavior objects to programs.

The first step in adding a behavior involves making sure the scene graph makes provisions for the behavior. For example, to use the SimpleBehavior class from the previous section there must be a TransformGroup in the scene graph above the object(s) to be rotated. Many behaviors need only a single TransformGroup object; however, scene graph requirements for a behavior is application and behavior dependent and may be more complex.

Having established the support for a behavior, an instance of the class must be added to the scene graph. Without being a part of a live scene graph, there is no way a behavior can be initialized. In fact, a behavior object that is not part of the scene graph will become garbage and be eliminated on the next garbage collection.

The last step for adding a behavior is to provide a scheduling bounds for the behavior. To improve efficiency, Java 3D uses the scheduling bounds to perform *execution culling*. Behavior is only *active* when its scheduling bounds intersects a ViewPlatform's activation volume. Only active behaviors are eligible to receive stimuli. In this way, stimuli can be ignored for some behaviors. The programmer has control over the execution culling through the selection of the scheduling bounds of the behavior.

Figure 4-3 summarizes the steps for using a behavior object in a recipe.

---

[1] A behavior based on time alone is an animation, and as such is discussed in Chapter 5.

1. prepare the scene graph (by adding a TransformGroup or other necessary objects)
2. insert behavior object in the scene graph, referencing the object of change
3. specify a scheduling bounds (or SchedulingBoundingLeaf)
4. set write (and read) capabilities for the target object (as appropriate)

Figure 4-3 Recipe for Using a Behavior Class

The following code fragment, Code Fragment 4-2, is annotated with the step numbers from the recipe. This code fragment is an except from the `SimpleBehaviorApp` example program found in the `examples/Interaction` directory. In this same application the SimpleBehavior class, found in Code Fragment 4-2, is defined. Code Fragment 4-2 continues where Code Fragment 4-1 ended and the line numbers are sequential for the two code fragments.

```
37. public BranchGroup createSceneGraph() {
38.     // Create the root of the branch graph
39.     BranchGroup objRoot = new BranchGroup();
40.
41.     TransformGroup objRotate = new TransformGroup();
42. ❹  objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
43. ❶
44.     objRoot.addChild(objRotate);
45.     objRotate.addChild(new ColorCube(0.4));
46.
47.     SimpleBehavior myRotationBehavior = new SimpleBehavior(objRotate);
48. ❸  myRotationBehavior.setSchedulingBounds(new BoundingSphere());
49. ❷  objRoot.addChild(myRotationBehavior);
50.
51.     // Let Java 3D perform optimizations on this scene graph.
52.     objRoot.compile();
53.
54.     return objRoot;
55. } // end of CreateSceneGraph method of SimpleBehaviorApp
```

Code Fragment 4-2 CreateSceneGraph Method in SimpleBehaviorApp.java

Very little code is needed to complete the program started in Code Fragment 4-1 and 4-2. The complete program, `SimpleBehaviorApp`, is found in the `examples/Interaction` directory. The complete application renders a ColorCube object in a static scene until a keyboard key is pressed. In response to any key press, the ColorCube rotates 0.1 radians (about 6°). Figure 4-4 shows the scene graph diagram for the content branch graph of this application.

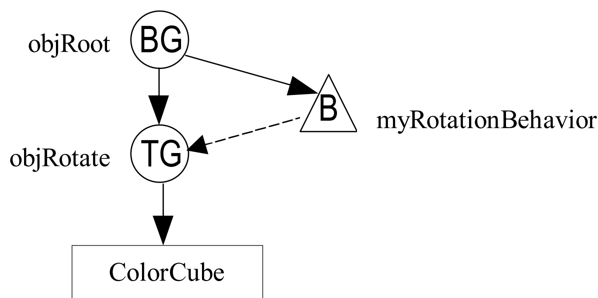


Figure 4-4 Scene Graph Diagram of the Content Branch Graph Created in SimpleBehaviorApp.java.

The above scene graph diagram clearly shows the relationship between the behavior object and the object of change, the TransformGroup object. The example rotates a ColorCube, but the behavior class is not limited to this. It can rotate any visual object, or portion of a scene graph that is a child of a TransformGroup object.

This simple example is not intended to demonstrate all of the possibilities of behaviors; it is only a starting point in the exploration of behaviors. Section 4.2.3 presents the Behavior class API. Other behavior class programming considerations are discussed before that.

Programming Pitfalls of Using Behavior Objects

In the three steps of the using a behavior class recipe, the two most likely programming mistakes are:

- not specifying a scheduling bounds (correctly), and
- not adding a behavior to the scene graph.

The intersection of the scheduling bounds of a behavior with the activation volume of a view determines whether or not Java 3D even considers the trigger stimulus for the behavior. Java 3D will not warn you of a missing scheduling bounds - the behavior will never be triggered. Also, keep the scheduling bounds of each behavior object as small as possible for the best overall performance.

As mentioned above, a behavior object that is not part of the scene graph will be considered garbage and eliminated on the next garbage collection cycle. This, too, will happen without error or warning.

Where in the Scene Graph Should a Behavior Object Go?

Behaviors can be placed anywhere in the scene graph. The issues in picking a scene graph location for a behavior object are 1) the effect on the scheduling bounds, and 2) code maintenance.

The bounds object referenced by a behavior object is subject to the local coordinate system of the behavior object's position in the scene graph. In the scene graph created in SimpleBehaviorApp, the SimpleBehavior object and the ColorCube are not subject to the same local coordinate system. In the example application this does not create a problem. The TransformGroup object of the example only rotates the ColorCube so that the scheduling bounds for the myRotationBehavior object always encloses the ColorCube object allowing interaction with the ColorCube when it is visible[2].

However, if the TransformGroup object were used to translate the ColorCube object, it would be possible to move the ColorCube out of the view. Since the bounds object stays with the behavior object in this scene, the user would be able to continue to translate the object. As long as the activation volume of a view still intersects the scheduling bounds for the behavior, the behavior is still active.

Being able to interact with a visual object that is not in the view is not bad (if that is what you want). The problem lies in that if the view were to change such that the activation volume no longer intersects the scheduling bounds of the behavior, even to include the visual object, the behavior is inactive. So the visual object you want to interact with may be in your view but not active. Most users will consider this a problem (even if it is intentional).

There two solutions to this problem. One is to change the scene graph to keep the scheduling bounds of the behavior with the visual object. This is easily accomplished as demonstrated in Figure 4-5. The

---

[2] The typical graphical application allows a user to interact with visible objects (visual objects that are in the view). If you want a different semantic, that is fine.

alternative solution uses a BoundingLeaf object for the scheduling bounds. Consult Section 3.7 or the Java 3D API Specification for information on the BoundingLeaf class.

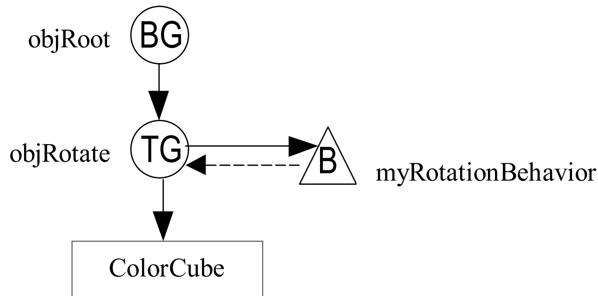


Figure 4-5 An Alternative Scene Graph Placement for the Behavior Object in SimpleBehaviorApp.

Behavior Class Design Recommendations

The mechanics of writing a custom behavior are simple. However, you should be aware that a poorly written behavior can degrade rendering performance[3]. While there are other considerations in writing a behavior, two things to avoid are: *memory burn* and unnecessary trigger conditions.

'Memory burn' is the term for unnecessarily creating objects in Java. Excessive memory burn will cause garbage collections. Occasional pauses in rendering is typical of memory burn since during the garbage collection, the rendering will stop[4][5].

Behavior class methods are often responsible for creating memory burn problems. For example, in Code Fragment 4-1 the processStimulus uses a `'new'` in the invocation of `wakeupOn` (line 24). This causes a new object to be created each time the method is invoked. That object becomes garbage each time the behavior is triggered.

Potential memory burn problems are normally easy to identify and avoid. Look for any use of 'new' in the code to find the source of memory burn problems. Whenever possible, replace the use of the new with code that reuses an object.

Later, in Section 4.3,the classes and methods used in setting the trigger conditions for a behavior object are discussed. In that section, you will see it is possible to set a trigger condition that will wake a behavior every frame of the rendering. If there is nothing for the behavior to do, this is an unnecessary waste of processor power invoking the behavior's processStimulus method. Not to say that there isn't a good reason to trigger a behavior on every frame, just make sure you have the reason.

4.2.3 Behavior Class API

This section presents the detail of the Behavior class API. Figure 4-6 shows the Java 3D API class hierarchy including the Behavior class. As an abstract class, Behavior must be extended before a behavior object can be instantiated. Of course, you can write your own custom behavior classes. In

---

[3] The amount of performance degradation depends heavily on the execution environment. If you plan to distribute your applications, consider users with software rendering environments.

[4] How often and how regular the pause depends on the execution environment.

[5] You can diagnose a memory burn problem by invoking the Java virtual machine with the `-verbose:gc` command line option. If memory burn is the cause for rendering pauses, then the garbage collection report produced to the console will coincide with the pauses.

addition, there are many existing behaviors in the Java 3D API utility packages. As an extension of Leaf, instances of classes that extend Behavior can be children of a group in a scene graph.

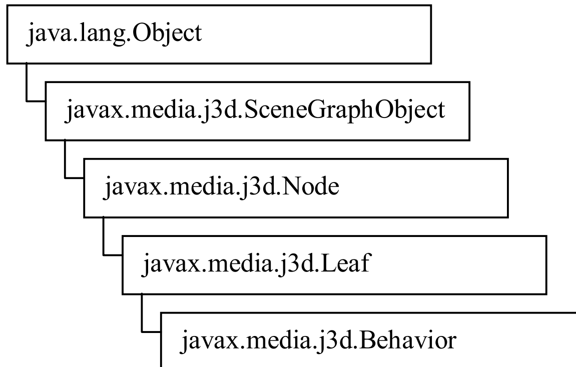


Figure 4-6 API Class Hierarchy for Behavior

As documented in Section 4.2.1, the processStimulus and initialize methods provide the interface Java 3D uses to incorporate behaviors in the virtual universe. The other Behavior class methods are discussed below. All Behavior class methods are listed in the Behavior Method Summary reference block on the next page.

The wakeupOn method is used in the initialize and processStimulus methods to set the trigger for the behavior. The parameter to this method is a WakeupCondition object. WakeupCondition, and related classes, are presented in Section 4.3.

The postId method allows a behavior to communicate with another method. One of the wakeup conditions is WakeupOnBehaviorPost. Behavior objects can be coordinated to create complex collaborations using the postId method in conjunction with appropriate WakeupOnBehaviorPost conditions. See page 4-16 for information on the WakeupOnBehaviorPost class

The setEnable method provides a way to disable a behavior even when the bounds makes it active. The default is true (i.e., the behavior object is enabled).

A behavior object is active only when its scheduling bounds intersects the activation volume of a View. Since it is possible to have multiple views in a virtual universe, a behavior can be made active by more than one view.

The getView method is useful with behaviors that rely on per-View information (e.g., Billboard, LOD) and with behaviors in general in regards to scheduling. This method returns a reference to the primary View object currently associated with the behavior. There is no corresponding setView method. The "primary" view is defined to be the first View attached to a live ViewPlatform, if there is more than one active View. So, for instance, Billboard behaviors would be oriented toward this primary view, in the case of multiple active views into the same scene graph.

Behavior Method Summary

Behavior is an abstract class that contains the framework for all behavioral components in Java 3D.

`View getView()`
Returns the primary view associated with this behavior.

`void initialize()`
Initialize this behavior.

`void postId(int postId)`
Post the specified Id.

`void processStimulus(java.util.Enumeration criteria)`
Process a stimulus meant for this behavior.

`void setEnable(boolean state)`
Enables or disables this Behavior.

`void setSchedulingBoundingLeaf(BoundingLeaf region)`
Set the Behavior's scheduling region to the specified bounding leaf.

`void setSchedulingBounds(Bounds region)`
Set the Behavior's scheduling region to the specified bounds.

`void wakeupOn(WakeupCondition criteria)`
Defines this behavior's wakeup criteria.

ViewPlatform API

Behaviors are active (able to be triggered) only when their scheduling bounds (or BoundingLeaf) intersects the activation volume of a ViewPlatform.

ViewPlatform Method Summary (partial list)

These methods of the ViewPlatform class get and set the activation volume (sphere) radius. Default activation radius = 62.

`float getActivationRadius()`
Get the ViewPlatform's activation radius.

`void setActivationRadius(float activationRadius)`
Set the ViewPlatform's activation radius which defines an activation volume around the view platform.

4.3 Wakeup Conditions: How Behaviors are Triggered

Active behaviors are triggered by the occurrence of a specified one or more wakeup stimuli. The wakeup stimuli for a behavior are specified using descendants of the WakeupCondition class.

The abstract class, WakeupCondition, is the base of the all the wakeup classes in the Java 3D API hierarchy. Five classes extend WakeupCondition, one is the abstract class WakeupCriterion, the other four allow the composition of multiple wakeup conditions in a single wakeup condition. Figure 4-7 shows the class hierarchy for these classes.

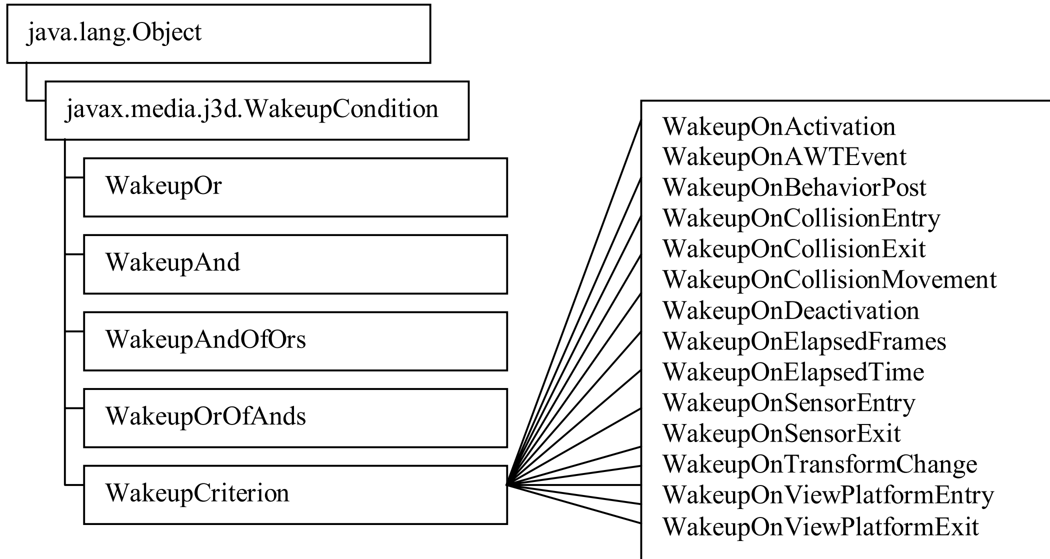


Figure 4-7 The Java 3D API Class Hierarchy for WakeupCondition and Related Classes.

A behavior object's wakeup condition can be specified as one of the specific wakeup criterion or as a combination of criteria using the wakeup composition classes. The following sections describe WakeupCondition and its descendant classes.

4.3.1 WakeupCondition

The WakeupCondition class provides two methods. The first method, allElements, returns the enumeration list of all wakeup criterion for the WakeupCondition object. The other method, triggeredElements, enumerates which of the wakeup criterion has caused the behavior to be triggered. This method may be useful in the processStimulus method of a Behavior object.

WakeupCondition Method Summary

The WakeupCondition abstract class is the base for all wakeup classes. It provides the following two methods.

`Enumeration allElements()`
Returns an enumeration of all WakeupCriterion objects in this Condition.

`Enumeration triggeredElements()`
Returns an enumeration of all triggered WakeupCriterion objects in this Condition.

4.3.2 WakeupCriterion

WakeupCriterion is an abstract method for the 14 specific wakeup criterion classes. WakeupCondition provides only one method: hasTriggered. You probably don't need to use this method as the triggeredElements method of WakeupCondition performs this operation for you.

WakeupCriterion Method Summary

`boolean hasTriggered()`
Returns true if this criterion triggered the wakeup.

4.3.3 Specific WakeupCriterion Classes

Table 4-2 presents the 14 specific WakeupCriterion classes. These classes are used to specify the wakeup conditions for behavior objects. Instances of these classes are used individually or in combinations when using the wakeup condition composition classes presented in Section 4.3.4.

Table 4-2 The 14 Specific WakeupCriterion Classes

Wakeup Criterion	Trigger	page
WakeupOnActivation	on first detection of a ViewPlatform's activation volume intersecting with this object's scheduling region.	4-15
WakeupOnAWTEvent	when a specific AWT event occurs	4-15
WakeupOnBehaviorPost	when a specific behavior object posts a specific event	4-16
WakeupOnCollisionEntry	on the first detection of the specified object colliding with any other object in the scene graph	4-17
WakeupOnCollisionExit	when the specified object no longer collides with any other object in the scene graph	4-19
WakeupOnCollisionMovement	when the specified object moves while in collision with any other object in the scene graph	4-20
WakeupOnDeactivation	when a ViewPlatform's activation volume no longer intersects with this object's scheduling region	4-21
WakeupOnElapsedFrames	when a specific number of frames have elapsed	4-21
WakeupOnElapsedTime	when a specific number of milliseconds have elapsed	4-21
WakeupOnSensorEntry	on first detection of any sensor intersecting the specified boundary	4-22
WakeupOnSensorExit	when a sensor previously intersecting the specified boundary no longer intersects the specified boundary	4-23
WakeupOnTransformChange	when the transform within a specified TransformGroup changes	4-23
WakeupOnViewPlatformEntry	on first detection of a ViewPlatform activation volume intersecting with the specified boundary	4-23
WakeupOnViewPlatformExit	when a View activation volume no longer intersects the specified boundary	4-24

Reference blocks for the individual WakeupCriterion classes appear on the next ten pages. Some WakeupCriterion classes have very simple APIs. For example, the WakeupOnActivation class has just one constructor. The scheduling region, a parameter of this wakeup condition, is specified in the behavior object that uses this criterion.

General WakeupCriterion Comments

A number of WakeupCriterion classes trigger on the "first detection" of an event. What this means is the criterion will trigger only once for the event. For example, a WakeupOnActivation object will trigger the intersection of the activation volume of a ViewPlatform and the scheduling region of the associated behavior object is detected. As long as this intersection persists, the WakeupCondition does not trigger again. The same is true for each of the sequentially following frames. Not until Java 3D detects that the volumes no longer intersect can this WakeupCondition trigger again.

Also, there are a number of WakeupCriterion classes in matched pairs (Entry/Exit or Activation/Deactivation). These criteria only trigger in strict alternation beginning with the Entry or Activation criterion.

WakeupOnActivation

It is possible for a scheduling region to intersect a ViewPlatform's activation volume so briefly that it is not detected. Consequently, neither the Activation nor Deactivation conditions are triggered. Under these circumstances, the behavior does not become active either.

WakeupOnActivation Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup on first detection of a ViewPlatform's activation volume intersection with this object's scheduling region. WakeupOnActivation is paired with WakeupOnDeactivation which appears on page 4-21.

`WakeupOnActivation()`
Constructs a new WakeupOnActivation criterion.

WakeupOnAWTEvent

Several of the WakeupCriterion classes have trigger dependent constructors and methods. For example, WakeupOnAWTEvent has two constructors and a method. The constructors allow the specification of AWT events using AWT class constants. The method returns the array of consecutive AWT events that caused the trigger.

WakeupOnAWTEvent Constructor Summary

extends: `WakeupCriterion`

Class that specifies a Behavior wakeup when a specific AWT event occurs. Consult an AWT reference for more information.

`WakeupOnAWTEvent(int AWTId)`
Constructs a new WakeupOnAWTEvent object, where `AWTId` is one of `KeyEvent.KEY_TYPED, KeyEvent.KEY_PRESSED, KeyEvent.KEY_RELEASED, MouseEvent.MOUSE_CLICKED, MouseEvent.MOUSE_PRESSED, MouseEvent.MOUSE_RELEASED, MouseEvent.MOUSE_MOVED, MouseEvent.MOUSE_DRAGGED,` or one of many other event values.

`WakeupOnAWTEvent(long eventMask)`
Constructs a new WakeupOnAWTEvent object using ORed EVENT_MASK values. AWT EVENT_MASK values are: `KEY_EVENT_MASK, MOUSE_EVENT_MASK, MOUSE_MOTION_EVENT_MASK,` or other values.

WakeupOnAWTEvent Method Summary

`AWTEvent[] getAWTEvent()`
Retrieves the array of consecutive AWT events that triggered this wakeup.

WakeupOnBehaviorPost

The WakeupOnBehaviorPost condition together with the postID method of the Behavior class provides a mechanism through which behaviors can coordinate. A Behavior object can post a particular integer ID value. Another behavior can specify its wakeup condition, using a WakeupOnBehaviorPost, as the posting of a particular ID from a specific behavior object. This allows for the creation of parenthetical behavior objects such as having one behavior open a door and different one closing it. For that matter, even more complex behaviors can be formulated using behaviors and post coordination.

WakeupOnBehaviorPost Constructor Summary

extends: `WakeupCriterion`

Class that specifies a Behavior wakeup when a specific behavior object posts a specific event.

`WakeupOnBehaviorPost(Behavior behavior, int postId)`
Constructs a new WakeupOnBehaviorPost criterion.

Since a WakeupCondition can be composed of a number of WakeupCriterion objects, including more than one WakeupOnBehaviorPost, the methods to determine the specifics of the triggering post are necessary to interpret a trigger event.

WakeupOnBehaviorPost Method Summary

`Behavior getBehavior()`
Returns the behavior specified in this object's constructor.

`int getPostId()`
Retrieve the WakeupCriterion's specified postId

`Behavior getTriggeringBehavior()`
Returns the behavior that triggered this wakeup.

`int getTriggeringPostId()`
Returns the postId that caused the behavior to wakeup.

Code Fragment 4-3 and Code Fragment 4-4 show a partial code for an example program of using behavior posting for coordinated behaviors. The example is that of opening and closing a door. The code fragments includes one class: OpenBehavior, and code that creates the two behavior objects. The second object is an instance of CloseBehavior, which is almost and exact duplicate of OpenBehavior. In CloseBehavior, the condition is swapped in the initialization method (and the opposite behavior performed).

```
1. public class OpenBehavior extends Behavior{
2.
3.     private TransformGroup  targetTG;
4.     private WakeupCriterion pairPostCondition;
5.     private WakeupCriterion AWTEventCondition;
6.
7.     OpenBehavior(TransformGroup targetTG){
8.         this.targetTG = targetTG;
9.         AWTEventCondition = new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED);
10.    }
11.
12.    public void setBehaviorObjectPartner(Behavior behaviorObject){
13.        pairPostCondition = new WakeupOnBehaviorPost(behaviorObject, 1);
14.    }
15.
16.     public void initialize(){
17.        this.wakeupOn(AWTEventCondition);
18.    }
19.
20.    public void processStimulus(Enumeration criteria){
21.        if (AWTEventCondition.hasTriggered()){
22.            // make door open - code excluded
23.            this.wakeupOn(pairPostCondition);
24.            postId(1);
25.        } else {
26.            this.wakeupOn(AWTEventCondition);
27.        }
28.    }
29.
30. } // end of class OpenBehavior
```

Code Fragment 4-3 Outline of OpenBehavior Class, an Example of Coordinated Behavior Classes

```
1. // inside a method to assemble the scene graph ...
2.
3.     // create the relevant objects
4.     TransformGroup doorTG      = new TransformGroup();
5.     OpenBehavior   openObject  = new OpenBehavior(doorTG);
6.     CloseBehavior  closeObject = new CloseBehavior(doorTG);
7.
8.     //prepare the behavior objects
9.     openObject.setBehaviorObjectPartner(closeObject);
10.    closeObject.setBehaviorObjectPartner(openObject);
11.
12.    // set scheduling bounds for behavior objects - code excluded
13.
14.    // assemble scene graph - code excluded
15.
```

Code Fragment 4-4 Code using OpenBehavior and CloseBehavior, Coordinated Behavior Classes

Objects of these two classes would respond in strict alternation to key press events. The open behavior object would trigger in response to the first key press. In its response, it signals the close behavior object and sets its trigger condition to be a signal from the close object. The open behavior object opens the door (or whatever) in response to the key press, as well. The close behavior object sets its trigger to be a key press in response to the signal from the open behavior object. An example program included in the `examples/Interaction` subdirectory, `DoorApp.java`, utilizes an open and close behavior pair.

The next key press triggers the close object. The close object now performs the same functions that the open object just performed: send a signal and reset its own trigger condition. The close object closes the door (or whatever) in response to the key press. Back to the initial conditions, the next key press would begin the process again.

WakeupOnCollisionEntry

Java 3D can detect the collision of objects in the virtual world. There are three WakeupCriterion classes useful in processing the collision of objects: WakeupOnCollisionEntry, WakeupOnCollisionMovement, and WakeupOnCollisionExit.

A WakeupOnCollisionEntry criterion will trigger when an object first collides. Then, WakeupOnCollisionMovement criterion will trigger (potentially multiple triggers) while the two objects are in collision if there is relative movement between the objects. Finally, a single WakeupOnCollisionExit will trigger when the collision is over.

Java 3D can handle only one collision for an object at a time. Once a collision is detected for an object, collisions with other objects are not detected until that first collision is over. Also, it is possible for a collision to occur so briefly that it is not detected. Consequently, neither the CollisionEntry nor CollisionExit conditions are triggered.

Collision detection is more complex than this discussion of the collision wakeup conditions. However, this tutorial does not address collision detection in detail. Refer to the Java 3D API Specification for more information on collision detection.

WakeupOnCollisionEntry Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup on the first detection of a specified object colliding with any other object in the scene graph. See also: WakeupOnCollisionMovement, and WakeupOnCollisionExit.

`WakeupOnCollisionEntry(Bounds armingBounds)`
Constructs a new WakeupOnCollisionEntry criterion.

`WakeupOnCollisionEntry(Node armingNode)`
Constructs a new WakeupOnCollisionEntry criterion.

`WakeupOnCollisionEntry(Node armingNode, int speedHint)`
Constructs a new WakeupOnCollisionEntry criterion, where speedHint is either:
`USE_BOUNDS` - Use geometric bounds as an approximation in computing collisions.
`USE_GEOMETRY` - Use geometry in computing collisions.

`WakeupOnCollisionEntry(SceneGraphPath armingPath)`
Constructs a new WakeupOnCollisionEntry criterion with USE_BOUNDS for a speed hint.

`WakeupOnCollisionEntry(SceneGraphPath armingPath, int speedHint)`
Constructs a new WakeupOnCollisionEntry criterion, where speedHint is either `USE_BOUNDS` or `USE_GEOMETRY`.

WakeupOnCollisionExit

WakeupOnCollisionExit Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup when the specified object no longer collides with any other object in the scene graph. See also: WakeupOnCollisionMovement, and WakeupOnCollisionEntry.

`WakeupOnCollisionExit(Bounds armingBounds)`
Constructs a new WakeupOnCollisionExit criterion.

`WakeupOnCollisionExit(Node armingNode)`
Constructs a new WakeupOnCollisionExit criterion.

`WakeupOnCollisionExit(Node armingNode, int speedHint)`
Constructs a new WakeupOnCollisionExit criterion, where speedHint is either:
`USE_BOUNDS` - Use geometric bounds as an approximation in computing collisions.
`USE_GEOMETRY` - Use geometry in computing collisions.

`WakeupOnCollisionExit(SceneGraphPath armingPath)`
Constructs a new WakeupOnCollisionExit criterion.

`WakeupOnCollisionExit(SceneGraphPath armingPath, int speedHint)`
Constructs a new WakeupOnCollisionExit criterion, where speedHint is either `USE_BOUNDS`, or `USE_GEOMETRY`.

WakeupOnCollisionExit Method Summary

`Bounds getArmingBounds()`
Returns the bounds object used in specifying the collision condition.

`SceneGraphPath getArmingPath()`
Returns the path used in specifying the collision condition.

`Bounds getTriggeringBounds()`
Returns the Bounds object that caused the collision

`SceneGraphPath getTriggeringPath()`
Returns the path describing the object causing the collision.

WakeupOnCollisionMovement

WakeupOnCollisionMovement Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup when the specified object moves while in collision with any other object in the scene graph. See also: WakeupOnCollisionEntry, and WakeupOnCollisionExit.

`WakeupOnCollisionMovement(Bounds armingBounds)`
Constructs a new WakeupOnCollisionMovement criterion.

`WakeupOnCollisionMovement(Node armingNode)`
Constructs a new WakeupOnCollisionMovement criterion.

`WakeupOnCollisionMovement(Node armingNode, int speedHint)`
Constructs a new WakeupOnCollisionMovement criterion, where speedHint is either:
`USE_BOUNDS` - Use geometric bounds as an approximation in computing collisions.
`USE_GEOMETRY` - Use geometry in computing collisions.

`WakeupOnCollisionMovement(SceneGraphPath armingPath)`
Constructs a new WakeupOnCollisionMovement criterion.

`WakeupOnCollisionMovement(SceneGraphPath armingPath, int speedHint)`
Constructs a new WakeupOnCollisionMovement criterion, where speedHint is either `USE_BOUNDS`, or `USE_GEOMETRY`.

WakeupOnCollisionMovement Method Summary

`Bounds getArmingBounds()`
Returns the bounds object used in specifying the collision condition.

`SceneGraphPath getArmingPath()`
Returns the path used in specifying the collision condition.

`Bounds getTriggeringBounds()`
Returns the Bounds object that caused the collision

`SceneGraphPath getTriggeringPath()`
Returns the path describing the object causing the collision.

WakeupOnDeactivation

WakeupOnDeactivation Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup on first detection of a ViewPlatform's activation volume no longer intersecting with this object's scheduling region. See also: WakeupOnActivation (page 4-15).

`WakeupOnDeactivation()`
Constructs a new WakeupOnDeactivation criterion.

WakeupOnElapsedFrames

WakeupOnElapsedFrames object is used to trigger an active object after the specified number of frames have elapsed. A frameCount of 0 specifies a wakeup on the next frame.

WakeupOnElapsedFrames Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup when a specific number of frames have elapsed.

`WakeupOnElapsedFrames(int frameCount)`
Constructs a new WakeupOnElapsedFrames criterion.

frameCount - the number of frames that Java 3D should draw before awakening this behavior object; a value of N means wakeup at the end of frame N, where the current frame is zero, a value of zero means wakeup at the end of the current frame.

`WakeupOnElapsedFrames(int frameCount, boolean passive)` **<new in 1.2>**
Constructs a WakeupOnElapsedFrames criterion with frameCount (see above) and passive parameters.

passive - flag indicating whether this behavior is passive; a non-passive behavior will cause the rendering system to run continuously, while a passive behavior will only run when some other event causes a frame to be run.

WakeupOnElapsedFrames Method Summary

`int getElapsedFrameCount()`
Retrieve the WakeupCriterion's elapsed frame count that was used when constructing this object.

`boolean isPassive()` **<new in 1.2>**
Retrieves the state of the passive flag that was used when constructing this object. Returns: true if this wakeup criterion is passive, false otherwise.

WakeupOnElapsedTime

Java 3D can not guarantee the exact timing of the wakeup trigger for a WakeupOnElapsedTime criterion. A wakeup will occur at the specified time, or shortly thereafter.

WakeupOnElapsedTime Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup after a specific number of milliseconds have elapsed.

`WakeupOnElapsedTime(long milliseconds)`
Constructs a new WakeupOnElapsedTime criterion.

WakeupOnElapsedTime Method Summary

`long getElapsedFrameTime()`
Retrieve the WakeupCriterion's elapsed time value that was used when constructing this object.

WakeupOnSensorEntry

In Java 3D, any input devices other than the keyboard or mouse is a *sensor*. A sensor is an abstract concept of an input device. Each sensor has a hotspot defined in the sensor's coordinate system. The intersection of a sensor's hotspot with a region can be detected with the WakeupOnSensorEntry and WakeupOnSensorExit classes.

It is possible for a sensor to enter and exit an armed region so quickly that neither the SensorEntry nor SensorExit conditions are triggered.

WakeupOnSensorEntry Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup on first detection of the intersection of any sensor with the specified boundary.

`WakeupOnSensorEntry(Bounds region)`
Constructs a new WakeupOnEntry criterion.

WakeupOnSensorEntry Method Summary

`Bounds getBounds()`
Returns this object's bounds specification

`Sensor getTriggeringSensor()` **`<new in 1.2>`**
Retrieves the Sensor object that caused the wakeup.

WakeupOnSensorExit

WakeupOnSensorExit Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup on first detection of a sensor previously intersecting the specified boundary no longer intersecting the specified boundary. See also: WakeupOnSensorEntry.

`WakeupOnSensorExit(Bounds region)`
Constructs a new WakeupOnExit criterion.

WakeupOnSensorExit Method Summary

`Bounds getBounds()`
Returns this object's bounds specification

`Sensor getTriggeringSensor()` **`<new in 1.2>`**
Retrieves the Sensor object that caused the wakeup.

WakeupOnTransformChange

The WakeupOnTransformChange criterion is useful for detecting changes in position or orientation of visual objects in the scene graph. This criterion offers an alternative to using the postId method for creating coordinated behaviors. This is especially useful when the behavior with which you want to coordinate is already written, such as the behavior utilities presented in sections 4.4 and 4.5.

WakeupOnTransformChange Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup when the transform within a specified TransformGroup changes

`WakeupOnTransformChange(TransformGroup node)`
Constructs a new WakeupOnTransformChange criterion.

WakeupOnTransformChange Method Summary

`TransformGroup getTransformGroup()`
Returns the TransformGroup node used in creating this WakeupCriterion

WakeupOnViewPlatformEntry

The detection of the intersection of the ViewPlatform with a specified region is made possible with the WakeupOnViewPlatfomEntry and WakeupOnViewPlatformExit criterion classes.

It is possible for the specified boundary to intersect a ViewPlatform's activation volume so briefly that it is not detected. In this case neither the WakeupOnViewPlatformEntry nor WakeupOnViewPlatformExit conditions are triggered.

WakeupOnViewPlatformEntry Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup on first ViewPlatform intersection with the specified boundary.

`WakeupOnViewPlatformEntry(Bounds region)`
Constructs a new WakeupOnEntry criterion.

WakeupOnViewPlatformEntry Method Summary

`Bounds getBounds()`
Returns this object's bounds specification

WakeupOnViewPlatformExit

WakeupOnViewPlatformExit Constructor Summary

extends: `WakeupCriterion`

Class specifying a wakeup on first detection of a Viewplatform no longer intersecting the specified boundary. See also WakeupOnViewPlatformEntry.

`WakeupOnViewPlatformExit(Bounds region)`
Constructs a new WakeupOnExit criterion.

WakeupOnViewPlatformExit Method Summary

`Bounds getBounds()`
Returns this object's bounds specification

4.3.4 WakeupCondition Composition

Multiple WakeupCriteron objects can be composed into a single WakeupCondition using the four classes presented in this section. The first two classes allow the composition of a WakeupCondition from a collection WakeupCriterion objects that are logically ANDed or ORed together, respectively. The third and forth allow compositions of instances of the first two classes into more complex WakeupCondition objects.

WakeupAnd

WakeupAnd Constructor Summary

extends: `WakeupCondition`

Class specifying any number of wakeup criterion logically ANDed together.

`WakeupAnd(WakeupCriterion[] conditions)`
Constructs a new WakeupAnd condition.

WakeupOr

WakeupOr Constructor Summary

extends: `WakeupCondition`

Class specifying any number of wakeup criterion logically ORed together.

`WakeupAnd(WakeupCriterion[] conditions)`
Constructs a new WakeupOr condition.

WakeupAndOfOrs

WakeupAndOfOrs Constructor Summary

extends: `WakeupCondition`

Class specifying any number of WakeupOr condition logically ANDed together.

`WakeupAndOfOrs(WakeupOr[] conditions)`
Constructs a new WakeupAndOfOrs condition.

WakeupOrOfAnds

WakeupOrOfAnds Constructor Summary

extends: `WakeupCondition`

Class specifying any number of WakeupAnd condition logically ORed together.

`WakeupOrsOfAnds(WakeupAnd[] conditions)`
Constructs a new WakeupOrOfAnds condition.

4.4 Behavior Utility Classes for Keyboard Navigation

To this point in the tutorial, the viewer has been in a fixed location with a fixed orientation. Being able to move the viewer is an important capability in many 3D graphics applications. Java 3D is capable of moving the viewer. In fact there are Java 3D utility classes which implement this functionality.

Figure 4-8 shows the basic view branch graph for a Java 3D virtual universe. In this figure, the view platform transform is seen. If the transform is changed, the effect is to move, or re-orient, or both, the

viewer. From this, you can see that the basic design of keyboard navigation is simple: have a behavior object change the view platform transform in response to key strokes.

This simple design is exactly how the Java 3D keyboard utility classes work. Of course you could build your own keyboard navigation behavior. The rest of this section explains how to use the Java 3D keyboard navigation classes.

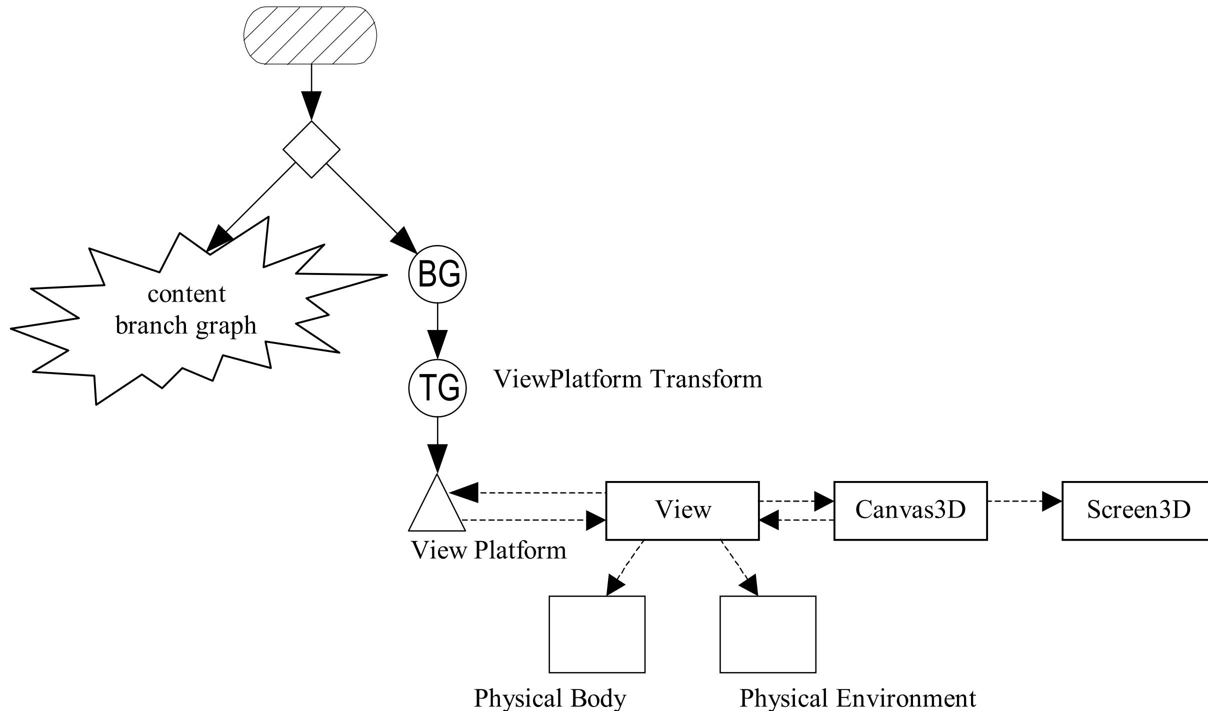


Figure 4-8 The Basic View Branch Graph Showing the View Platform Transform

How to Navigate in a SimpleUniverse

You might be thinking that needing access to the view platform transform group object means abandoning the SimpleUniverse utility. However, the SimpleUniverse, and related classes, provide a combination of method to retrieve the ViewPlatformTransform object. Therefore, you can have your SimpleUniverse and navigate in it too!

Specifically, the following line of code retrieves the ViewPlatformTransform from a SimpleUniverse object, su.

```
TransformGroup vpt =
          su.getViewingPlatform().getViewPlatformTransform();
```

4.4.1 Simple KeyNavigatorBehavior Example Program

It is easy to use the KeyNavigatorBehavior utility class in a Java 3D program. This section demonstrates using the class in the `KeyNavigatorApp` example program found in the `examples/Interaction` subdirectory. In this program you can see that the steps involved in using the KeyNavigatorBehavior class are essentially identical to those of using any behavior class (as discussed in section 4.2.2 on page 4-7). The steps for using KeyNavigatorBehavior are summarized in Figure 4-9.

1. create a KeyNavigatorBehavior object, setting the transform group
2. add the KeyNavigatorBehavior object to the scene graph
3. provide a bounds (or BoundingLeaf) for the KeyNavigatorBehavior object

Figure 4-9Recipe for Using the KeyNavigatorBehavior Utility Class

Like any programming problem, there are a variety of ways to implement the steps of this recipe. One approach is to incorporate these steps in the createSceneGraph method[6]. Code Fragment 4-5 shows the steps of the recipe as implemented for the `KeyNavigatorApp` example program found in the `examples/Interaction` subdirectory. The code fragment is annotated with the step numbers from the recipe. Like many of the other recipes, the exact ordering of all steps is not critical.

```
1.      public BranchGroup createSceneGraph(SimpleUniverse su) {
2.          // Create the root of the branch graph
3.          TransformGroup vpTrans = null;
4.
5.          BranchGroup objRoot = new BranchGroup();
6.
7.          objRoot.addChild(createLand());
8.
9.             // create other scene graph content
10.
11.
12.         vpTrans = su.getViewingPlatform().getViewPlatformTransform();
13.         translate.set( 0.0f, 0.3f, 0.0f);   // 3 meter elevation
14.         T3D.setTranslation(translate);      // set as translation
15.         vpTrans.setTransform(T3D);          // used for initial position
16.  ❶      KeyNavigatorBehavior keyNavBeh = new KeyNavigatorBehavior(vpTrans);
17.  ❸      keyNavBeh.setSchedulingBounds(new BoundingSphere(
18.                                             new Point3d(),1000.0));
19.  ❷      objRoot.addChild(keyNavBeh);
20.
21.         // Let Java 3D perform optimizations on this scene graph.
22.         objRoot.compile();
23.
24.         return objRoot;
25.     } // end of CreateSceneGraph method of KeyNavigatorApp
```

Code Fragment 4-5 Using the KeyNavigatorBehavior Class (part 1)

Performing step 1 of the recipe in the createSceneGraph method requires access to the ViewPlatform transform group. This implementation passes the SimpleUniverse object (line 34 of Code Fragment 4-6) to the createSceneGraph method making it available to access the ViewPlatform transform (line 12 of Code Fragment 4-5).

Passing the SimpleUniverse object to the createSceneGraph method makes it possible to gain access to other view branch graph features of the SimpleUniverse, such as PlatformGeometry, ViewerAvatar, or adding a BoundingLeaf to the view branch graph.

Lines 13 through 15 of Code Fragment 4-5 provide an initial position for the viewer. In this case, the viewer is translated to a position 0.3 meters above the origin of the virtual world. This is only an initial position, and in no way limits the viewer's future position or orientation.

[6] In this tutorial, `createSceneGraph()` is a standard method of the main class of a Java 3D program.

```
26.     public KeyNavigatorApp() {
27.         setLayout(new BorderLayout());
28.         Canvas3D canvas3D = new Canvas3D(null);
29.         add("Center", canvas3D);
30.
31.         // SimpleUniverse is a Convenience Utility class
32.         SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
33.
34.         BranchGroup scene = createSceneGraph(simpleU);
35.
36.         simpleU.addBranchGraph(scene);
37.     } // end of KeyNavigatorApp (constructor)
```

Code Fragment 4-6 Using the KeyNavigatorBehavior Class (part 2)

How to make Universal Application of a Behavior

As with any behavior object, the KeyNavigtorBehavior object is only active when its scheduling bounds intersects the activation volume of a ViewPlatform. This can be particularly limiting for a navigation behavior, where the behavior should always be on. Chapter 3 discusses a solution to this problem using a BoundingLeaf. Refer to the BoundingLeaf section of Chapter 3 for more information.

4.4.2 KeyNavigatorBehavior and KeyNavigator Classes

The keyboard navigation utility is implemented as two classes. At run time there are two objects. The first object is the KeyNavigatorBehavior object, the second is a KeyNavigator object. The second class is not documented here since neither the programmer nor the user need to know that the second class or object exists.

The KeyNavigatorBehavior object performs all the typical functions of a behavior class, except that it calls on the KeyNavigator object to perform the processStimulus function. The KeyNavigator class takes the AWTEvent and processes it down to the individual key stroke level. Table 4-3 shows the effect of the individual key strokes. KeyNavigator implements motion with acceleration.

Table 4-3 KeyNavigatorBehavior Movements

Key	MOVEMENT	**Alt-key movement**
←	rotate left	lateral translate left
→	rotate right	lateral translate right
↑	move forward	
↓	move backward	
PgUp	rotate up	translation up
PgDn	rotate down	translation down
+	restore back clip distance (and return to the origin)	
-	reduce back clip distance	
=	return to center of universe	

The following reference blocks show the API for the KeyNavigatorBehavior utility class.

KeyNavigatorBehavior Constructor Summary

`Package:` com.sun.j3d.utils.behaviors.keyboard
`Extends:` Behavior

This class is a simple behavior that invokes the KeyNavigator to modify the view platform transform.

`KeyNavigatorBehavior(TransformGroup targetTG)`
Constructs a new key navigator behavior node that operates on the specified transform group.

KeyNavigatorBehavior Method Summary

`void initialize()`
Override Behavior's initialize method to setup wakeup criteria.

`void processStimulus(java.util.Enumeration criteria)`
Override Behavior's stimulus method to handle the event.

4.5 Utility Classes for Mouse Interaction

the mouse utility behavior package (`com.sun.j3d.utils.behaviors.mouse`) contains behavior classes in which the mouse is used as input for interaction with visual objects. Included are classes for translating (moving in a plane parallel to the image plate), zooming (moving forward and back), and rotating visual objects in response to mouse movements.

Table 4-4 summarizes the three specific mouse behavior classes included in the package. In addition to these three classes, there is the abstract MouseBehavior class, and MouseCallback Interface in the mouse behavior package. This abstract class and the interface are used in the creation of the specific mouse behavior classes and are useful for creating custom mouse behaviors.

Table 4-4 Summary of Specific MouseBehavior Classes

MouseBehavior class	Action in Response to Mouse Action	Mouse Action
MouseRotate	rotate visual object in place	left-button held with mouse movement
MouseTranslate	translate the visual object in a plane parallel to the image plate	right-button held with mouse movement
MouseZoom	translate the visual object in a plane orthogonal to the image plate	middle-button held with mouse movement

4.5.1 Using the Mouse Behavior Classes

The specific mouse behavior classes are easy to use. Usage of these class is essentially the same as using any other behavior class. Figure 4-10 presents the recipe for using Mouse Behavior classes.

1. provide read and write capabilities for the target transform group
2. create a MouseBehavior object
3. set the target transform group
4. provide a bounds (or BoundingLeaf) for the MouseBehavior object
5. add the MouseBehavior object to the scene graph

Figure 4-10 Recipe for Using Mouse Behavior Classes

As with some other recipes, the steps don't have to be performed strictly in the given order. Step two must be performed before three, four, and five; the other steps can be performed in any order. Also, steps two and three can be combined using a different constructor.

Code Fragment 4-7 presents the createSceneGraph method from the `MouseRotateApp` example program included in the `examples/Interaction` subdirectory. The scene graph includes ColorCube object. The user can rotate the ColorCube using the mouse due to the inclusion of a MouseRotate object in the scene graph. Code Fragment 4-7 is annotated with the step numbers from the recipe of Figure 4-10.

```
1. public class MouseRotateApp extends Applet {
2.
3.     public BranchGroup createSceneGraph() {
4.         // Create the root of the branch graph
5.         BranchGroup objRoot = new BranchGroup();
6.
7.         TransformGroup objRotate = new TransformGroup();
8.     ❶   objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
9.     ❶   objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
10.
11.        objRoot.addChild(objRotate);
12.        objRotate.addChild(new ColorCube(0.4));
13.
14.    ❷   MouseRotate myMouseRotate = new MouseRotate();
15.    ❸   myMouseRotate.setTransformGroup(objRotate);
16.    ❹   myMouseRotate.setSchedulingBounds(new BoundingSphere());
17.    ❺   objRoot.addChild(myMouseRotate);
18.
19.        // Let Java 3D perform optimizations on this scene graph.
20.        objRoot.compile();
21.
22.        return objRoot;
23.    } // end of CreateSceneGraph method of MouseRotateApp
```

Code Fragment 4-7 Using the MouseRotate Utility Class

The same recipe will work for the other two mouse behavior classes. In fact all three behaviors can be used in the same application operating on the same visual object. Since each of the mouse behaviors reads the target transform before writing to it, only one TransformGroup object is needed even with three mouse behaviors. The MouseBehaviorApp example program does just that. You can find this example program in the `examples/Interaction` subdirectory.

The next example shows how two mouse behaviors work in a single virtual world. The example program `MouseRotate2App` creates a scene graph with two ColorCube objects near each other in the virtual world. Each of the ColorCubes has a MouseRotate object associated with it. Since both mouse behavior objects are active, when the user clicks and moves the mouse, both ColorCubes rotate.

If you didn't want both objects to rotate, there are two solutions: 1) change the viewer's position, or change the behavior scheduling bounds, so that only one behavior is active, or 2) use a picking mechanism to isolate the behavior. Picking is discussed in section 4.6. In that section you will find classes that combine the mouse behaviors described in this section with picking, allowing the user to interact with one visual object at a time.

4.5.2 Mouse Behavior Foundation

The specific mouse behavior classes (MouseRotate, MouseTranslate, and MouseZoom) are extensions of the MouseBehavior abstract class and implement the MouseCallback Interface.

Mouse Behavior Abstract Class

This abstract is presented here in the event you want to extend it to write a custom mouse behavior class. The setTransformGroup() method is probably the only method users of an instance of MouseBehavior will use. The other methods are intended for the authors of custom mouse behavior classes.

MouseBehavior Method Summary

Base class for all mouse manipulators (see MouseRotate and MouseZoom for examples of how to extend this base class).

`void initialize()`
Initializes the behavior.

`void processMouseEvent(java.awt.event.MouseEvent evt)`
Handles mouse events.

`void processStimulus(java.util.Enumeration criteria)`
All mouse manipulators must implement this method of Behavior (respond to stimuli).

`void setTransformGroup(TransformGroup transformGroup)`
Set the target TransformGroup for the behavior.

`void wakeup()`
Manually wake up the behavior.

MouseCallback Interface

A class implementing this interface provides the transformChanged method which will be called when the target transform changes in the specified way. Each of the three specific mouse behavior classes implement this class. A programmer can simply override the transformChanged method of one of those classes to specify a method to be called when the transform is changed.

Interface MouseBehaviorCallback Method Summary

Package: `com.sun.j3d.utils.behaviors.mouse`

`void transformChanged(int type, Transform3D transform)`
Classes implementing this interface that are registered with one of the MouseBehaviors will be called every time the behavior updates the Transform. The type is one of `MouseCallback.ROTATE`, `MouseCallback.TRANSLATE`, or `MouseCallback.ZOOM`.

4.5.3 Specific Mouse Behavior Classes

Mouse Rotate

A scene graph that includes a MouseRotate object allows the user to rotate visual objects in the virtual world. The use of this class is explained in section 4.5.1. The example programs `MouseRotateApp`, `MouseRotate2App`, and `MouseBehaviorApp` demonstrate the use of this class.

MouseRotate Constructor Summary

Package: `com.sun.j3d.utils.behaviors.mouse`
Extends: `MouseBehavior`

MouseRotate is a Java3D behavior object that lets users control the rotation of an object via a mouse left-button drag. To use this utility, first create a transform group that this rotate behavior will operate on. The user can rotate any child object of the target TransformGroup.

`MouseRotate()`
Creates a default mouse rotate behavior.

`MouseRotate(TransformGroup transformGroup)`
Creates a rotate behavior given the transform group.

`MouseRotate(int flags)`
Creates a rotate behavior with flags set, where the flags are:
`MouseBehavior.INVERT_INPUT` Set this flag if you want to invert the inputs.
`MouseBehavior.MANUAL_WAKEUP` Set this flag if you want to manually wakeup the behavior.

MouseRotate Method Summary

void setFactor(double factor)
Set the x-axis and y-axis movement multiplier with factor.

void setFactor(double xFactor, double yFactor)
Set the x-axis and y-axis movement multiplier with xFactor and yFactor respectively.

void setupCallback(MouseBehaviorCallback callback)
The transformChanged method in the callback class will be called every time the transform is updated

void transformChanged(Transform3D transform)
Users can overload this method which is called every time the Behavior updates the transform. The default implementation does nothing.

MouseTranslate

A scene graph that includes a MouseTranslate object allows the user to move visual objects in a plane parallel to the image plate in the virtual world. The use of this class is explained in section 4.5.1. The example program `MouseBehaviorApp` demonstrates the use of this class.

MouseTranslate Constructor Summary

Package: `com.sun.j3d.utils.behaviors.mouse`
Extends: `MouseBehavior`

MouseTranslate is a Java3D behavior object that lets users control the translation (X, Y) of an object via a mouse drag motion with the right mouse button.

`MouseTranslate()`
Creates a default translate behavior.

`MouseTranslate(TransformGroup transformGroup)`
Creates a mouse translate behavior given the transform group.

`MouseTranslate(int flags)`
Creates a translate behavior with flags set, where the flags are:
`MouseBehavior.INVERT_INPUT` Set this flag if you want to invert the inputs.
`MouseBehavior.MANUAL_WAKEUP` Set this flag if you want to manually wakeup the behavior.

MouseTranslate Method Summary

`void setFactor(double factor)`
Set the x-axis and y-axis movement multiplier with factor.

`void setFactor(double xFactor, double yFactor)`
Set the x-axis and y-axis movement multiplier with xFactor and yFactor respectively.

`void setupCallback(MouseBehaviorCallback callback)`
The transformChanged method in the callback class will be called every time the transform is updated

`void transformChanged(Transform3D transform)`
Users can overload this method which is called every time the Behavior updates the transform. The default implementation does nothing.

MouseZoom

A scene graph that includes a MouseZoom object allows the user to move visual objects in a plane orthogonal to the image plate in the virtual world. The use of this class is explained in section 4.5.1. The example program `MouseBehaviorApp` demonstrates the use of this class.

MouseZoom Constructor Summary

Package: `com.sun.j3d.utils.behaviors.mouse`
Extends: `MouseBehavior`

MouseZoom is a Java3D behavior object that lets users control the Z axis translation of an object via a mouse drag motion with the second middle button (alt-click on PC with two-button mouse). See MouseRotate for similar usage info.

`MouseZoom()`
Creates a default mouse zoom behavior.

`MouseZoom(TransformGroup transformGroup)`
Creates a zoom behavior given the transform group.

`MouseZoom(int flags)`
Creates a zoom behavior with flags set, where the flags are:
`MouseBehavior.INVERT_INPUT` Set this flag if you want to invert the inputs.
`MouseBehavior.MANUAL_WAKEUP` Set this flag if you want to manually wakeup the behavior.

MouseZoom Method Summary

`void setFactor(double factor)`
Set the z-axis movement multiplier with factor.

`void setupCallback(MouseBehaviorCallback callback)`
The transformChanged method in the callback class will be called every time the transform is updated

`void transformChanged(Transform3D transform)`
Users can overload this method which is called every time the Behavior updates the transform. The default implementation does nothing.

4.5.4 Mouse Navigation

The three specific mouse behavior classes can be used to create a virtual universe in which the mouse is used for navigation. Each of the specific mouse behavior classes has a constructor that takes a single `int flags` parameter. When the `MouseBehavior.INVERT_INPUTS` is used as the argument to this constructor, the mouse behavior responds in the opposite direction. This reverse behavior is appropriate for changing the ViewPlatform transform. In other words, the mouse behavior classes can be used for navigational control.

The example program `MouseNavigatorApp` uses instances of the three specific mouse behavior classes for navigational interaction. The complete source for this example program is in the `examples/Interaction` subdirectory. Code Fragment 4-8 shows the createSceneGraph method from this example program.

The target TransformGroup for each of the mouse behavior objects is the ViewPlatform transform. The SimpleUniverse object is an argument to the createSceneGraph method so that the ViewPlatform transform object can be accessed.

```
1. public BranchGroup createSceneGraph(SimpleUniverse su) {
2.    // Create the root of the branch graph
3.    BranchGroup objRoot = new BranchGroup();
4.         TransformGroup vpTrans = null;
5.         BoundingSphere mouseBounds = null;
6.
7.         vpTrans = su.getViewingPlatform().getViewPlatformTransform();
8.
9.         objRoot.addChild(new ColorCube(0.4));
10.        objRoot.addChild(new Axis());
11.
12.        mouseBounds = new BoundingSphere(new Point3d(), 1000.0);
13.
14.        MouseRotate myMouseRotate = new
                                   MouseRotate(MouseBehavior.INVERT_INPUT);
15.        myMouseRotate.setTransformGroup(vpTrans);
16.        myMouseRotate.setSchedulingBounds(mouseBounds);
17.        objRoot.addChild(myMouseRotate);
18.
19.        MouseTranslate myMouseTranslate = new
                                MouseTranslate(MouseBehavior.INVERT_INPUT);
20.        myMouseTranslate.setTransformGroup(vpTrans);
21.        myMouseTranslate.setSchedulingBounds(mouseBounds);
22.        objRoot.addChild(myMouseTranslate);
23.
24.        MouseZoom myMouseZoom = new MouseZoom(MouseBehavior.INVERT_INPUT);
```

```
25.         myMouseZoom.setTransformGroup(vpTrans);
26.         myMouseZoom.setSchedulingBounds(mouseBounds);
27.         objRoot.addChild(myMouseZoom);
28.
29.         // Let Java 3D perform optimizations on this scene graph.
30.         objRoot.compile();
31.
32.         return objRoot;
33.     } // end of createSceneGraph method of MouseNavigatorApp
```

Code Fragment 4-8 Using Mouse Behavior Classes for Interactive Navigation of the Virtual World.

The bounds object for the mouse behavior objects is specified as a BoundingSphere with a radius of 1000 meters. If the viewer moves beyond this sphere, the behavior objects will no longer be active.

4.6 Picking

In the MouseRotate2App example program, both ColorCube objects rotated in response to the actions of the user. In that application, there is no way to manipulate the cubes separately. Picking gives the user a way to interact with individual visual objects in the scene.

Picking is implemented by a behavior typically triggered by mouse button events. In picking a visual object, the user places the mouse pointer over the visual object of choice and presses the mouse button. The behavior object is triggered by this button press and begins the picking operation. A ray is projected into the virtual world from the position of the mouse pointer parallel with the projection. Intersection of this ray with the objects of the virtual world is computed. The visual object intersecting closest to the image plate is selected for interaction[7]. Figure 4-1 shows a pick ray projected in a virtual world.

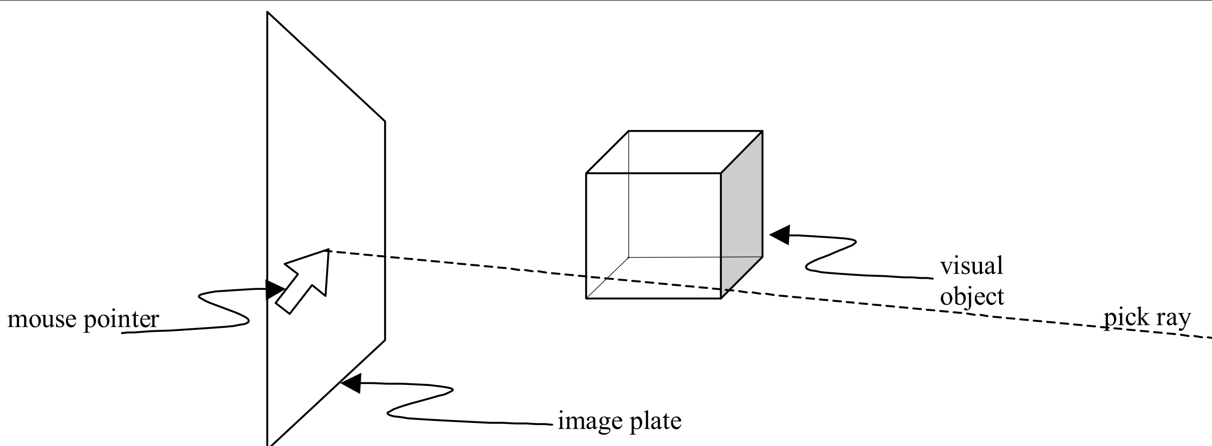


Figure 4-11 Projection of PickRay in the Virtual World

In some cases the interaction is not directly with the selected object, but with an object along the scene graph path to the object. For example, in picking a ColorCube object for rotation, the ColorCube object is not manipulated; the TransformGroup object above the ColorCube in the scene graph path to the ColorCube is. On the other hand, if the pick operation selects a visual object for which a color change is intended, then the visual object selected is indeed required.

[7] While interacting with the closest visual object is the most common picking application, picking operations are not limited to selecting the closest visual object. Some picking operations produce a list of all intersected objects.

The determination of the object for further processing is not always easy. If a cubic visual object that is to be rotated is composed of six individual Shape3D objects arranged by six TransformGroup objects, as in the scene graph diagram of Figure 4-12, it is not the TransformGroup object above the intersected Shape3D object that needs to be modified. The 'cube' is rotated by manipulation of the TransformGroup object that is the child of the BranchGroup object in the scene graph. For this reason, the result of some picking operations is to return the scene graph path for further processing.

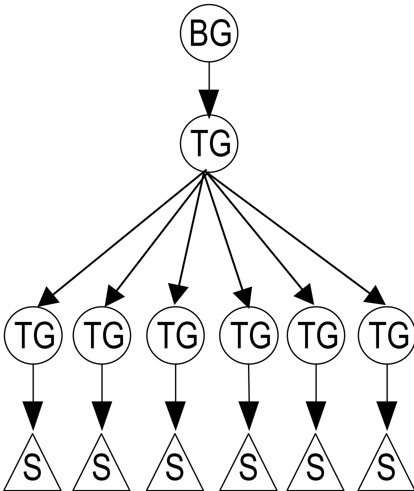


Figure 4-12 Scene Graph Diagram for a Cube Composed of Discrete Shape3D Plane Objects.

Intersection testing is computationally expensive. Therefore, picking is computationally expensive and is more expensive with increasing complexity of the scene. The Java 3D API provides a number of ways that a programmer can limit the amount of computation done in picking. One important way is through the capabilities and attributes of scene graph nodes. Whether or not a scene graph node is pickable is set with the `setPickable()` method of the class. A node with `setPickable()` set to `false` is not pickable and neither are any of its children. Consequently, these nodes are not considered when calculating intersections.

Another pick related feature of the Node class is the `ENABLE_PICK_REPORTING` capability. This capability applies only to Group nodes. When set for a Group, that group object will always be included in the scene graph path returned by a pick operation. Group nodes not needed for uniqueness in a scene graph path will be excluded when the capability is not set. Not having the right settings for scene graph nodes is a common source of frustration in developing applications utilizing picking operations.

The following two reference blocks list Node methods and capabilities, respectively.

Node Method (partial list)

extends: `SceneGraphObject`
subclasses: `Group, Leaf`

The Node class provides an abstract class for all Group and Leaf Nodes. It provides a common framework for constructing a Java 3D scene graph, specifically bounding volumes, picking and collision capabilities.

`void setBounds(Bounds bounds)`
Sets the geometric bounds of a node.

`void setBoundsAutoCompute(boolean autoCompute)`
Turns the automatic calcuation of geometric bounds of a node on/off.

`setPickable(boolean pickable)`
When set to true this Node can be Picked. Setting to false indicates that this node and it's children are ALL unpickable.

Node Capabilities Summary (partial list)

`ENABLE_PICK_REPORTING`
Specifies that this Node will be reported in the pick SceneGraphPath if a pick occurs. This capability is only specifiable for Group nodes; it is ignored for leaf nodes. The default for Group nodes is false. Interior nodes not needed for uniqueness in a SceneGraphPath that don't have ENABLE_PICK_REPORTING set will be excluded from the SceneGraphPath.

`ALLOW_BOUNDS_READ | WRITE`
Specifies that this Node allows read (write) access to its bounds information.

`ALLOW_PICKABLE_READ | WRITE`
Specifies that this Node allows reading (writing) its pickability state.

Another way a programmer can reduce the computation of picking is to use bounds intersection testing instead of geometric intersection testing. Several of the pick related classes have constructors and/or methods have a parameter, which is set to one of: `USE_BOUNDS` or `USE_GEOMETRY`. When the `USE_BOUNDS` option is selected, the pick is determined using the bounds of the visual objects, not the actual geometry. The determination of a pick using the bounds is significantly easier (computationally) for all but the simplest geometric shapes and therefore, results in better performance. Of course, the drawback is the picking is not as precise when using bounds pick determination.

A third programming technique for reducing the computational cost of picking is to limit the scope of the pick testing to the relevant portion of the scene graph. In each picking utility class a node of the scene graph is set as the root of the graph for pick testing. This node is not necessarily the root of the content branch graph. On the contrary, the node passed should be the root of the content subgraph that only contains pickable objects, if possible. This consideration may be a major factor in determining the construction of the scene graph for some applications.

4.6.1 Using Picking Utility Classes

There are two basic approaches to using the picking features of Java 3D: use objects of picking classes, or create custom picking classes and use instances of these custom classes. The picking package includes

classes for pick/rotate, pick/translate, and pick/zoom. That is, a user can pick and rotate an object by pressing the mouse button when the mouse pointer is over the desired object and then dragging the mouse (while holding the button down). Each of the picking classes uses a different mouse button making it possible to use objects of all three picking class in the same application simultaneously.

Section 4.6.4 presents the `PickRotateBehavior`, `PickTranslateBehavior`, and `PickZoomBehavior` utility classes. Section 4.6.2 presents classes useful for creating custom picking classes. This section presents a complete programming example using the PickRotate class.

Since a picking behavior object will operate on any scene graph object (with the appropriate capabilities), only one picking behavior object is necessary to provide picking. The following two lines of code is just about all that is needed to include picking via the picking utility classes in a Java 3D program:

```
PickRotateBehavior behavior = new PickRotateBehavior(root, canvas, bounds);
root.addChild(behavior);
```

The above `behavior` object will monitor for any picking events on the scene graph (below `root` node) and handle mouse drags on pick hits. The `root` provides the portion of the scene graph to be checked for picking, the `canvas` is the place where the mouse is, and the `bounds` is the scheduling bounds of the picking `behavior` object.

Figure 4-13 shows the simple recipe for using the mouse picking utility classes.

1. Create your scene graph.
2. Create a picking behavior object with root, canvas, and bounds specification.
3. Add the behavior object to the scene graph.
4. Enable the appropriate capabilities for scene graph objects

Figure 4-13 Recipe for Using Mouse Picking Utility Classes

Programming Pitfalls when Using Picking Objects

Using the picking behavior classes leads to the same programming pitfalls as using other behavior classes. Common problems include: forgetting to include the behavior object in the scene graph, and not setting an appropriate bounds for the behavior object. See "Programming Pitfalls of Using Behavior Objects" on page 4-9 for more details.

There are pitfalls specific to picking in addition to the pitfalls common to using behavior objects. The most common problem is not setting the proper capabilities for scene graph objects. Two other possible problems are less likely, however you should check for these if your picking application is not working. One is not setting the root of the scene graph properly. Another potential problem is not setting the canvas properly. None of these programming mistakes will generate an error or warning message.

MousePickApp Example Program

Code Fragment 4-9 shows the `createSceneGraph` method of `MousePickApp`. The complete source code for this example program is included in the `examples/Interaction` subdirectory of the example programs jar. This program uses a PickRotate object to provide interaction. This code is annotated with the step numbers from the recipe of Figure 4-13.

Note that since the construction of the picking object requires the Canvas3D object, the createSceneGraph method differs from earlier versions by the inclusion of the canvas parameter. Of course, the invocation of createSceneGraph changes correspondingly.

```
1.     public BranchGroup createSceneGraph(Canvas3D canvas) {
2.         // Create the root of the branch graph
3.         BranchGroup objRoot = new BranchGroup();
4.
5.         TransformGroup objRotate = null;
6.         PickRotateBehavior pickRotate = null;
7.         Transform3D transform = new Transform3D();
8.         BoundingSphere behaveBounds = new BoundingSphere();
9.
10.        // create ColorCube and PickRotateBehavior objects
11.        transform.setTranslation(new Vector3f(-0.6f, 0.0f, -0.6f));
12.        objRotate = new TransformGroup(transform);
13.      { objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
14. ❹   { objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
15.      { objRotate.setCapability(TransformGroup.ENABLE_PICK_REPORTING);
16.
17.        objRoot.addChild(objRotate);
18.        objRotate.addChild(new ColorCube(0.4));
19.
20. ❷      pickRotate = new PickRotateBehavior(objRoot,canvas, behaveBounds);
21. ❸      objRoot.addChild(pickRotate);
22.
23.        // add a second ColorCube object to the scene graph
24.        transform.setTranslation(new Vector3f( 0.6f, 0.0f, -0.6f));
25.        objRotate = new TransformGroup(transform);
26.      { objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
27. ❹   { objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
28.      { objRotate.setCapability(TransformGroup.ENABLE_PICK_REPORTING);
29.
30.        objRoot.addChild(objRotate);
31.        objRotate.addChild(new ColorCube(0.4));
32.
33.        // Let Java 3D perform optimizations on this scene graph.
34.        objRoot.compile();
35.
36.        return objRoot;
37.    } // end of createSceneGraph method of MousePickApp
```

Code Fragment 4-9 The createSceneGraph Method of the MousePickApp Example Program.

This code is similar to that of MouseRotate2App, but is different in some very distinct ways. Primarily, there is only one behavior object used in this program, where the MouseRotate2App used two behavior objects – one per visual object. While the code is similar, the behavior is very different. This program allows the user to pick an object to interact with. MouseRotate2App rotates both objects or neither object.

4.6.2 Java 3D API Core Picking Classes

There are three 'levels' of picking classes provided in Java 3D. The Java 3D API core provides the lowest level functionality. The picking utility package provides general picking behavior classes, suitable for customization. The picking utility package also provides specific picking behavior classes which can be used directly in Java 3D programs.

The core classes include PickShape and SceneGraphPath classes, and methods of BranchGroup and Locale. These classes provide the mechanisms for specifying a shape used in testing for intersection with visual objects. This section presents the API of the PickShape and SceneGraphPath classes, and related classes and methods.

The general picking utility package classes combine basic picking operations in behavior objects. The specific picking utility classes use the general classes to implement specific picking behaviors.

PickShape classes

This abstract class provides neither constructors nor methods. It provides abstraction for five subclasses: `PickBounds`, `PickRay`, `PickSegment`, `PickPoint`, and `PickCone`. The variety of picking classes are for various picking applications. Normally, the geometry which may be the target of picking will determine which of the pick shapes to use.

For polygonal geometry any of the pick shapes will probably serve the purpose. Keep in mind that there is likely to be performance differences among the various subclasses of PickShape. However, if the geometry appears small,

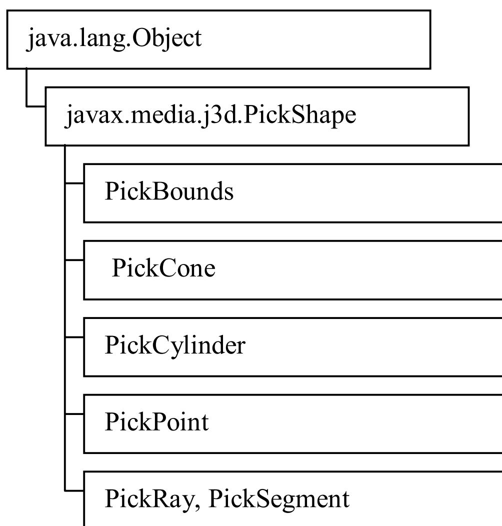


Figure 4-14 PickShape Hierarchy

Table 4-5 Selection of PickShape

application	geometry polygons appear large	polygons appear small	points and/or lines
general	PickRay	PickRay, PickBounds, PickCone, PickCylinder	PickBounds, PickCone, PickCylinder
accuracy	PickRay, PickSegment, PickPoint	PickRay, PickSegment, PickPoint,	PickRay, PickSegment, PickPoint
speed	PickRay	PickRay	PickRay

PickShape

Known Subclasses: `PickBounds, PickRay, PickSegment, PickPoint, and PickCone`

A general class for describing a pick shape which can be used with the BranchGroup and Locale pick methods.

PickBounds

PickBounds objects represent a bounds for pick testing. As a subclass of PickShape, PickBounds objects are used with BranchGroup and Locale pick testing as well as picking package classes.

PickBounds Constructor Summary

extends: `PickShape`

A bounds to supply to the BranchGroup and Locale pick methods

`PickBounds()`
Create a PickBounds.

`PickBounds(Bounds boundsObject)`
Create a PickBounds with the specified bounds.

PickBounds Method Summary

`Bounds get()`
Get the `boundsObject` from this PickBounds.

`void set(Bounds boundsObject)`
Set the `boundsObject` into this PickBounds.

PickPoint

PickPoint objects represent a point for picking. As a subclass of PickShape, PickPoint objects are used with BranchGroup and Locale pick testing as well as picking package classes.

PickPoint Constructor Summary

extends: `PickShape`

A point to supply to the BranchGroup and Locale pick methods

`PickPoint()`
Create a PickPoint at (0, 0, 0).

`PickPoint(Point3d location)`
Create a PickPoint at `location.`

PickPoint Method Summary

`void set(Point3d location)`
Set the position of this PickPoint. There is a matching get method.

PickRay

PickRay objects represent a ray (a point and a direction) for picking. As a subclass of PickShape, PickRay objects are used with BranchGroup and Locale pick testing as well as picking package classes.

PickRay Constructor Summary

extends: `PickShape`

PickRay is an encapsulation of a ray for passing to the pick methods in BranchGroup and Locale

`PickRay()`
Create a PickRay with origin and direction of (0, 0, 0).

`PickRay(Point3d origin, Vector3d direction)`
Create a ray cast from origin in direction `direction`.

PickRay Method Summary

`void set(Point3d origin, Vector3d direction)`
Set the ray to point from origin in direction `direction`. There is a matching get method.

PickSegment

PickSegment objects represent a line segment (defined by two points) for picking. As a subclass of PickShape, PickSegment objects are used with BranchGroup and Locale pick testing as well as picking package classes.

PickSegment Constructor Summary

extends: `PickShape`

PickRay is an encapsulation of ray for passing to the pick methods in BranchGroup and Locale

`PickSegment()`
Create a PickSegment.

`PickSegment(Point3d start, Point3d end)`
Create a pick segment from start point to end point.

PickSegment Method Summary
void set(Point3d start, Point3d end) Set the pick segment from start point to end point. There is a matching get method.

PickCone <new in 1.2>

PickCone is a new class in API version 1.2. PickCone is the abstract base class of all cone pick shapes. A cone pick shape is useful when picking among fine geometry. PickCone is similar to PickCylinder (next subsection, page 4-45). The application will determine which of these is more appropriate.

There are two PickCone subclasses: PickConeRay and PickConeSegment. The difference between them is length; PickConeRay is a cone of infinite length while PickConeSegment has finite length. PickConeRay is defined by an origin and a direction. PickConeSegment is defined by two points which determine both the direction and the length of the cone.

Figure 4-16 illustrates the parameters of the PickCone shapes. Note that a PickCone has an end point only if it is a PickConeSegment.

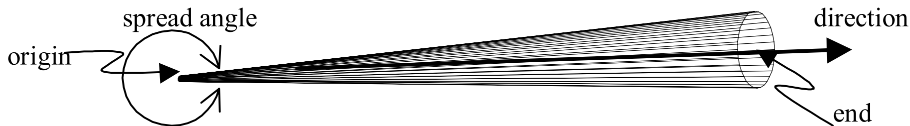


Figure 4-15 Parameters of PickCone pick shapes

The methods of PickCone are useful with objects of both subclass for retrieving the parameters of a PickCone object. The set methods, being class specific, are defined within the subclasses.

PickCone Method Summary
void getDirection(Vector3d direction) <new in 1.2> Gets the direction of this PickCone. There is no setDirection() method, the direction is set by parameters in the constructor.
void getOrigin(Point3d origin) <new in 1.2> Gets the origin of this PickCone. There is no setOrigin() method, the origin is set by setting the starting and ending points in the constructor.
double getSpreadAngle() <new in 1.2> Gets the spread angle of this PickCone.

PickConeRay <new in 1.2>

A specialization of the PickCone class of infinite length as defined by the origin point and direction.

PickConeRay Constructor Summary

extends `PickCone`

PickConeRay is an infinite cone ray pick shape. It can be used as an argument to the picking methods in BranchGroup and Locale.

`PickConeRay()` **<new in 1.2>**
Constructs an empty PickConeRay.

`PickConeRay(Point3d origin, Vector3d direction, double spreadAngle)` **<new in 1.2>**
Constructs an infinite cone pick shape from the specified parameters.

PickConeRay Method Summary

`void set(Point3d origin, Vector3d direction, double spreadAngle)` **<new in 1.2>**
Sets the parameters of this PickCone to the specified values. Get methods are supplied by the super class.

PickConeSegment <new in 1.2>

A specialization of the PickCone class having a finite length as defined by the origin and end points.

PickConeSegment Constructor Summary

extends: `PickCone`

PickConeSegment is a finite cone segment pick shape. It can be used as an argument to the picking methods in BranchGroup and Locale.

`PickConeSegment()` **<new in 1.2>**
Constructs an empty PickConeSegment.

`PickConeSegment(Point3d origin, Point3d end, double spreadAngle)` **<new in 1.2>**
Constructs a finite cone pick shape from the specified parameters.

PickConeSegment Method Summary

`void getEnd(Point3d end)` **<new in 1.2>**
Gets the end point of this PickConeSegment. Other get methods are supplied by the super class.

`void set(Point3d origin, Point3d end, double spreadAngle)` **<new in 1.2>**
Sets the parameters of this PickCone to the specified values.

PickCylinder <new in 1.2>

PickCylinder and its subclasses is completely analogous to PickCone and its subclasses (previously described, page 4-44). A cylinder or cone pick shape maybe useful when picking fine geometry; the application will determine which of these is more appropriate.

PickCone is the abstract base class of two concrete classes: PickCylinderRay and PickCylinderSegment. PickCylinder is similar to PickCone The difference is the length of the cylinder. PickCylinderRay is a cylinder of infinite length and is defined by an origin and a direction. PickCylinderSegment is a cylinder of finite length defined by two points. The two points define both the direction and the length of the cylinder.

Figure 4-16 illustrates the parameters of the PickCylinder shapes. Note that a PickCylinder has an defined end point only if it is a PickCylinderSegment.

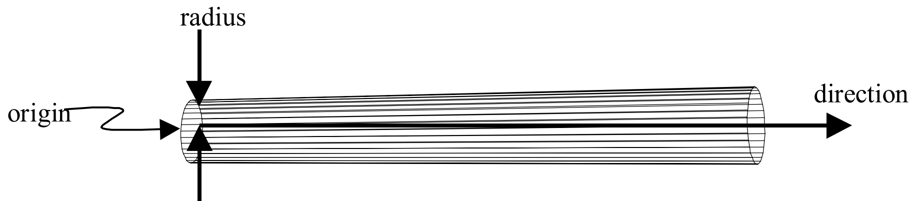


Figure 4-16 Parameters of PickCylinder pick shapes

The methods of PickCylinder are useful with objects of both subclass for retrieving the parameters of a PickCylinder object. The set methods, being class specific, are defined within the subclasses.

PickCylinder Method Summary

`void getDirection(Vector3d direction)` **<new in 1.2>**
Gets the direction of this cylinder.

`void getOrigin(Point3d origin)` **<new in 1.2>**
Gets the origin point of this cylinder object.

`double getRadius()` **<new in 1.2>**
Gets the radius of this cylinder object

PickCylinderRay <new in 1.2>

A specialization of the PickCylinder class of infinite length as defined by the origin point and direction.

PickCylinderRay Constructor Summary

extends: `PickCylinder`

PickCylinderRay is an infinite cylindrical ray pick shape. It can be used as an argument to the picking methods in BranchGroup and Locale.

`PickCylinderRay()` **<new in 1.2>**
Constructs an empty PickCylinderRay. Use the `set(Point3d, Vector3d double)` method to specify pick shape parameters.

`PickCylinderRay(Point3d origin, Vector3d direction, double radius)` **<new in 1.2>**
Constructs an infinite cylindrical ray pick shape from the specified parameters.

PickCylinderRay Method Summary

`void set(Point3d origin, Vector3d direction, double radius)` **<new in 1.2>**
Sets the parameters of this PickCylinderRay to the specified values.

PickCylinderSegment <new in 1.2>

A specialization of the PickCone class of infinite length as defined by the origin point and direction.

PickCylinderSegment Constructor Summary

extends: `PickCylinder`

PickCylinderSegment is a finite cylindrical segment pick shape. It can be used as an argument to the picking methods in BranchGroup and Locale.

`PickCylinderSegment()` **<new in 1.2>**
Constructs an empty PickCylinderSegment.

`PickCylinderSegment(Point3d origin, Point3d end, double radius)` **<new in 1.2>**
Constructs a finite cylindrical segment pick shape from the specified parameters.

PickCylinderRay Method Summary

`void getEnd(Point3d end)` **<new in 1.2>**
Gets the end point of this PickCylinderSegment.

`void set(Point3d origin, Point3d end, double radius)` **<new in 1.2>**
Sets the parameters of this PickCylinderSegment to the specified values.

SceneGraphPath

The class SceneGraphPath is used in most picking applications. This is because picking usually involves finding a scene graph path that the picked object lies in to allow manipulation of the object or a TransformGroup object in the path.

A SceneGraphPath object represents the scene graph path to the picked object allowing manipulation of the object or a TransformGroup object in the path to the object. SceneGraphPath is used in the picking package as well as Java 3D core

SceneGraphPath Overview

A SceneGraphPath object represents the path from a Locale to a terminal node in the scene graph. This path consists of a Locale, a terminal node, and an array of internal nodes that are in the path from the Locale to the terminal node. The terminal node may be either a Leaf node or a Group node. A valid SceneGraphPath must uniquely identify a specific instance of the terminal node. For nodes that are not under a SharedGroup, the minimal SceneGraphPath consists of the Locale and the terminal node itself. For nodes that are under a SharedGroup, the minimal SceneGraphPath consists of the Locale, the terminal node, and a list of all Link nodes in the path from the Locale to the terminal node. A SceneGraphPath may optionally contain other interior nodes that are in the path. A SceneGraphPath is verified for correctness and uniqueness when it is sent as an argument to other methods of Java 3D.

In the array of internal nodes, the node at index 0 is the node closest to the Locale. The indices increase along the path to the terminal node, with the node at index length-1 being the node closest to the terminal node. The array of nodes does not contain either the Locale (which is not a node) or the terminal node.

SceneGraphPath Constructor Summary

When a SceneGraphPath is returned from the picking or collision methods of Java 3D, it will also contain the value of the LocalToVworld transform of the terminal node that was in effect at the time the pick or collision occurred. Note that ENABLE_PICK_REPORTING and ENABLE_COLLISION_REPORTING are disabled by default. This means that the picking and collision methods will return the minimal SceneGraphPath by default.

`SceneGraphPath()`
Constructs a SceneGraphPath object with default parameters.

`SceneGraphPath(Locale root, Node object)`
Constructs a new SceneGraphPath object.

`SceneGraphPath(Locale root, Node[] nodes, Node object)`
Constructs a new SceneGraphPath object.

SceneGraphPath Method Summary (partial list)

`boolean equals(java.lang.Object o1)`
Returns true if the Object o1 is of type SceneGraphPath and all of the data members of o1 are equal to the corresponding data members in this SceneGraphPath and if the values of the transforms is equal.

`Transform3D getTransform()`
Returns a copy of the transform associated with this SceneGraphPath; returns null if there is no transform.

`int hashCode()`
Returns a hash number based on the data values in this object.

`boolean isSamePath(SceneGraphPath testPath)`
Determines whether two SceneGraphPath objects represent the same path in the scene graph; either object might include a different subset of internal nodes; only the internal link nodes, Locale, and the Node itself are compared.

`int nodeCount()`
Retrieves the number of nodes in this path.

`void set(SceneGraphPath newPath)`
Sets this path's values to that of the specified path.

`void setLocale(Locale newLocale)`
Sets this path's Locale to the specified Locale.

`void setNode(int index, Node newNode)`
Replaces the node at the specified index with newNode.

`void setNodes(Node[] nodes)`
Sets this path's node objects to the specified node objects.

`void setObject(Node object)`
Sets this path's terminal node to the specified node object.

`void setTransform(Transform3D trans)`
Sets the transform component of this SceneGraphPath to the value of the passed transform.

`java.lang.String toString()`
Returns a string representation of this object; the string contains the class names of all Nodes in the SceneGraphPath, the toString() method of any associated user, and also prints out the transform if it is not null.

BranchGroup and Local Picking Methods

Presented in the following reference block are methods of the BranchGroup and Local classes for intersection testing with PickShape objects. This is the lowest level pick computation provided by the Java 3D API.

BranchGroup and Locale picking methods for use with PickShape

`SceneGraphPath[] pickAll(PickShape pickShape)`
Returns an array referencing all the items that are pickable below this BranchGroup that intersect with PickShape. The resultant array is unordered.

`SceneGraphPath[] pickAllSorted(PickShape pickShape)`
Returns a sorted array of references to all the Pickable items that intersect with the pickShape. Element [0] references the item closest to origin of PickShape, with successive array elements further from the origin. Note: If pickShape is of type PickBounds, the resultant array is unordered.

`SceneGraphPath pickClosest(PickShape pickShape)`
Returns a SceneGraphPath which references the pickable item which is closest to the origin of pickShape. Note: If pickShape is of type PickBounds, the return is any pickable node below this BranchGroup.

`SceneGraphPath pickAny(PickShape pickShape)`
Returns a reference to any item that is Pickable below this BranchGroup which intersects with pickShape.

4.6.3 General Picking Package Classes

Included in the `com.sun.j3d.utils.behaviors.picking` package are general and specific pick behavior classes. The general picking classes are useful in creating new picking behaviors. The general picking classes include `PickMouseBehavior`, `PickObject`, and `PickCallback`. The specific mouse behavior classes, presented in the next section, are subclasses of the `PickMouseBehavior` class.

PickMouseBehavior Class

This is the base class for the specific picking behaviors provided in the package. It is also useful for extending to custom picking behavior classes.

PickMouseBehavior Method Summary

Package: `com.sun.j3d.utils.behaviors.picking`
Extends: `Behavior`

Base class that allows programmers to add picking and mouse manipulation to a scene graph (see PickDragBehavior for an example of how to extend this base class).

`void initialize()`
This method should be overridden to provide initial state and the initial trigger condition.

`void processStimulus(java.util.Enumeration criteria)`
This method should be overridden to provide the behavior in response to a wakeup condition.

`void updateScene(int xpos, int ypos)`
Subclasses shall implement this update function

PickObject Class

The PickObject class provides methods for determining which object was selected by a user pick operation. A wide variety of methods provide results in various formats for various possible picking applications. It is useful in creating custom picking classes.

PickObject Constructor Summary

package: `com.sun.j3d.utils.behaviors.picking`
extends: `java.lang.Object`

Contains methods to aid in picking. A PickObject is created for a given Canvas3D and a BranchGroup. SceneGraphObjects under the specified BranchGroup can then be checked to determine if they have been picked.

`PickObject(Canvas3D c, BranchGroup root)`
Creates a PickObject.

PickObject Method Summary (partial list)

PickObject has numerous method for computing the intersection of a pickRay with scene graph objects. Some of the methods differ by only one parameter. For example, the second pickAll method (not listed) exists with the method signature of: `SceneGraphPath[] pickAll(int xpos, int ypos, int flag)`, where `flag` is one of: `PickObject.USE_BOUNDS`, or `PickObject.USE_GEOMETRY`.

This list has been shortened by excluding the methods with the flag parameter. These methods are identical to methods included in this list with the difference of the flag parameter. These methods are: `pickAll` , `pickSorted`, `pickAny`, and `pickClosest`.

`PickShape generatePickRay(int xpos, int ypos)`
Creates a PickRay that starts at the viewer position and points into the scene in the direction of (xpos, ypos) specified in window space.

PickObject Method Summary (partial list - continued)

`SceneGraphPath[] pickAll(int xpos, int ypos)`
Returns an array referencing all the items that are pickable below the BranchGroup (specified in the PickObject constructor) that intersect with a ray that starts at the viewer position and points into the scene in the direction of (xpos, ypos) specified in window space.

`SceneGraphPath[] pickAllSorted(int xpos, int ypos)`
Returns a sorted array of references to all the Pickable items below the BranchGroup (specified in the PickObject constructor) that intersect with the ray that starts at the viewer position and points into the scene in the direction of (xpos, ypos) in the window space.

`SceneGraphPath pickAny(int xpos, int ypos)`
Returns a reference to any item that is Pickable below the specified BranchGroup (specified in the PickObject constructor) which intersects with the ray that starts at the viewer position and points into the scene in the direction of (xpos, ypos) in window space.

SceneGraphPath pickClosest(int xpos, int ypos)
Returns a reference to the item that is closest to the viewer and is Pickable below the BranchGroup (specified in the PickObject constructor) which intersects with the ray that starts at the viewer position and points into the scene in the direction of (xpos, ypos) in the window space.

Node pickNode(SceneGraphPath sgPath, int node_types)
Returns a reference to a Pickable Node that is of the specified type that is contained in the specified SceneGraphPath. Where node_types is the logical OR of one or more of: `PickObject.BRANCH_GROUP`, `PickObject.GROUP`, `PickObject.LINK`, `PickObject.MORPH`, `PickObject.PRIMITIVE`, PickObject.SHAPE3D, `PickObject.SWITCH`, `PickObject.TRANSFORM_GROUP`.

Node pickNode(SceneGraphPath sgPath, int node_types, int occurrence)
Returns a reference to a Pickable Node that is of the specified type that is contained in the specified SceneGraphPath. Where node_types is as defined for the above method. The `occurrence` parameter indicates which object to return.

PickingCallback Interface

The PickingCallback Interface provides a framework for extending an existing picking class. In particular, each of the specific pick classes (in Section 4.6.4) implements this interface allowing the programmer to provide a method to be called when a pick operation takes place.

Interface PickingCallback Method Summary

package: `com.sun.j3d.utils.behaviors.picking`

void transformChanged(int type, TransformGroup tg)
Called by the Pick Behavior with which this callback is registered each time the pick is attempted. Valid values for `type` are: ROTATE, TRANSLATE, ZOOM or NO_PICK (the user made a selection but nothing was actually picked).

Intersect Class

The Intersect Class provides a number of methods for testing for the intersection of a `PickShape` (core class) object and geometry primitives. This class is useful in creating custom picking classes.

Intersect Constructor Summary

package: `com.sun.j3d.utils.behaviors.picking`
extends: `java.lang.Object`

Contains static methods to aid in the intersection test between various PickShape classes and geometry primitives (such as quad, triangle, line and point).

Intersect()
Create an intersect object.

Intersect Method Summary (partial list)

The Intersect class has numerous intersection methods, some of which only differ by one parameter type. For example, the method: `boolean pointAndPoint(PickPoint point, Point3f pnt)` differs from the second listed method here by only the type of the pnt parameter. Most of the methods listed here with a parameter of type Point3d have a corresponding method with a parameter of type Point3f.

`boolean pointAndLine(PickPoint point, Point3d[] coordinates, int index)`
Return true if the PickPoint and Line objects intersect. coordinates[index] and coordinates[index+1] define the line

`boolean pointAndPoint(PickPoint point, Point3d pnt)`
Return true if the PickPoint and Point3d objects intersect.

`boolean rayAndLine(PickRay ray, Point3d[] coordinates, int index, double[] dist)`
Return true if the PickRay and Line objects intersect. coordinates[index] and coordinates[index+1] define the line.

`boolean rayAndPoint(PickRay ray, Point3d pnt, double[] dist)`
Return true if the PickRay and Point3d objects intersect.

`boolean rayAndQuad(PickRay ray, Point3d[] coordinates, int index, double[] dist)`
Return true if the PickRay and quadrilateral objects intersect.

`boolean rayAndTriangle(PickRay ray, Point3d[] coordinates, int index, double[] dist)`
Return true if triangle intersects with ray and the distance, from the origin of ray to the intersection point, is stored in dist[0]. coordinates[index], coordinates[index+1], and coordinates[index+2] define the triangle.

`boolean segmentAndLine(PickSegment segment, Point3d[] coordinates, int index, double[] dist)`
Return true if line intersects with segment; the distance from the start of segment to the intersection point is stored in dist[0]. coordinates[index] and coordinates[index+1] define the line.

Intersect Method Summary (partial list - continued)

`boolean segmentAndPoint(PickSegment segment, Point3d pnt, double[] dist)`
Return true if the PickSegment and Point3d objects intersect.

`boolean segmentAndQuad(PickSegment segment, Point3d[] coordinates, int index, double[] dist)`
Return true if quad intersects with segment; the distance from the start of segment to the intersection point is stored in dist[0].

`boolean segmentAndTriangle(PickSegment segment, Point3d[] coordinates, int index, double[] dist)`
Return true if triangle intersects with segment; the distance from the start of segment to the intersection point is stored in dist[0].

4.6.4 Specific Picking Behavior Classes

Included in the `com.sun.j3d.utils.behaviors.picking` package are specific pick behavior classes: `PickRotateBehavior`, `PickTranslateBehavior`, and `PickZoomBehavior`. These

classes allow the user to interact with a picked object with the mouse. The individual behaviors respond to different mouse buttons (left=rotate, right=translate, middle=zoom). All three specific mouse behavior classes are subclasses of the PickMouseBehavior class.

Objects of these classes can be incorporated in Java 3D virtual worlds to provide interaction by following the recipe provided in Figure 4-13. Since each of these classes implements the PickingCallback Interface, the operation of the picking can be augmented with a call to a user defined method. Refer to the PickingCallback Interface documentation in 4.6.2 for more information.

PickRotateBehavior

The PickRotateBehavior allows the user to interactively pick and rotate visual objects. The user uses the left mouse button for pick selection and rotation. An instance of PickRotateBehavior can be used in conjunction with other specific picking classes.

PickRotateBehavior Constructor Summary

package: `com.sun.j3d.utils.behaviors.picking`
extends: `PickMouseBehavior`
implements: `PickingCallback`

A mouse behavior that allows user to pick and rotate scene graph objects; expandable through a callback.

`PickRotateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds)`
Creates a pick/rotate behavior that waits for user mouse events for the scene graph.

`PickRotateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds, int pickMode)`
Creates a pick/rotate behavior that waits for user mouse events for the scene graph. The pickMode parameter is specified as one of `PickObject.USE_BOUNDS` or `PickObject.USE_GEOMETRY`. Note: If `pickMode` is set to `PickObject.USE_GEOMETRY`, all geometry objects in the scene graph intended to be available for picking must have their `ALLOW_INTERSECT` bit set.

PickRotateBehavior Method Summary

`void setPickMode(int pickMode)`
Sets the pickMode component of this PickRotateBehavior to one of `PickObject.USE_BOUNDS` or `PickObject.USE_GEOMETRY`. Note: If `pickMode` is set to `PickObject.USE_GEOMETRY`, all geometry objects in the scene graph intended to be available for picking must have their `ALLOW_INTERSECT` bit set.

`void setupCallback(PickingCallback callback)`
Register the class callback to be called each time the picked object moves.

`void transformChanged(int type, Transform3D transform)`
Callback method from MouseRotate. This is used when the Picking callback is enabled.

`void updateScene(int xpos, int ypos)`
Update the scene to manipulate any nodes.

PickTranslateBehavior

The PickTranslateBehavior allows the user to interactively pick and translate visual objects. The user uses the right mouse button for pick selection and translation. An instance of PickTranslateBehavior can be used in conjunction with other specific picking classes.

PickTranslateBehavior Constructor Summary

package: `com.sun.j3d.utils.behaviors.picking`
extends: `PickMouseBehavior`
implements: `PickingCallback`

A mouse behavior that allows user to pick and translate scene graph objects. The behavior is expandable through a callback.

`PickTranslateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds)`
Creates a pick/translate behavior that waits for user mouse events for the scene graph.

`PickTranslateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds, int pickMode)`
Creates a pick/translate behavior that waits for user mouse events for the scene graph. . The pickMode parameter is specified as one of `PickObject.USE_BOUNDS` or `PickObject.USE_GEOMETRY`. Note: If `pickMode` is set to `PickObject.USE_GEOMETRY`, all geometry objects in the scene graph intended to be available for picking must have their `ALLOW_INTERSECT` bit set.

PickTranslateBehavior Method Summary

`void setPickMode(int pickMode)`
Sets the pickMode component of this PickTranslateBehavior to the value of the passed pickMode.

`void setupCallback(PickingCallback callback)`
Register the class callback to be called each time the picked object moves.

`void transformChanged(int type, Transform3D transform)`
Callback method from MouseTranslate. This is used when the Picking callback is enabled.

`void updateScene(int xpos, int ypos)`
Update the scene to manipulate any nodes.

PickZoomBehavior

The PickZoomBehavior allows the user to interactively pick and zoom visual objects. The user uses the middle mouse button for pick selection and zooming. An instance of PickZoomBehavior can be used in conjunction with other specific picking classes.

PickZoomBehavior Constructor Summary

package: `com.sun.j3d.utils.behaviors.picking`
extends: `PickMouseBehavior`
implements: `PickingCallback`

A mouse behavior that allows user to pick and zoom scene graph objects. The behavior is expandable through a callback.

`PickZoomBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds)`
Creates a pick/zoom behavior that waits for user mouse events for the scene graph.

`PickZoomBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds, int pickMode)`
Creates a pick/zoom behavior that waits for user mouse events for the scene graph. The `pickMode` parameter is specified as one of `PickObject.USE_BOUNDS` or `PickObject.USE_GEOMETRY`. Note: If `pickMode` is set to `PickObject.USE_GEOMETRY`, all geometry objects in the scene graph intended to be available for picking must have their `ALLOW_INTERSECT` bit set.

PickZoomBehavior Method Summary

`void setPickMode(int pickMode)`
Sets the pickMode component of this PickZoomBehavior to the value of the passed pickMode.

`void setupCallback(PickingCallback callback)`
Register the class callback to be called each time the picked object moves.

`void transformChanged(int type, Transform3D transform)`
Callback method from MouseZoom. This is used when the Picking callback is enabled.

`void updateScene(int xpos, int ypos)`
Update the scene to manipulate any nodes.

4.7 Chapter Summary

This chapter begins by explaining the significance of the Behavior class in providing interaction and animation in the Java 3D virtual universe. This chapter provides a comprehensive view of Java 3D core and utility classes used in providing interaction for viewer navigation of the virtual world, picking and interacting with individual visual objects, and how to create new interactive behavior classes.

Section 4.2 shows how custom behavior classes are written and then shows how to incorporate behavior objects to provide interaction in a Java 3D virtual world. Section 4.3 discusses the various classes used in the specification of behavior trigger conditions. Section 4.4 discusses the KeyNavigatorBehavior class which is used for view navigation through key strokes. Section 4.5 presents mouse interaction classes. Section 4.6 presents the topic of picking in general and discusses utility classes used to provide picking interaction.

4.8 Self Test

1. Write a custom behavior application that moves visual objects to the left and right when a the left and right arrow keys are pressed, respectively. Then use the class in an application similar to

SimpleBehaviorApp.java. Of course, you can use SimpleBehaviorApp.java as a starting point for both the custom behavior class and the application. What happens as the ColorCube object moves out of the view? How do you fix the problem?

2. In SimpleBehaviorApp, the rotation is computed using an angle variable of type double. The angle variable is used to set the rotation of a Transform3D object which sets the transform of the TransformGroup. An alternative would eliminate the angle variable using only a Transform3D object to control the angle increment. There are two variations on this approach: one would read the current transform of the TransformGroup and then multiply, another would store the transform in a local Transform3D object. In either case, the new rotation is found by multiplying the previous Transform3D with the Transform3D that holds the rotation increment. What problem may occur with this alternative? What improvement can be made to this approach?
3. Change the trigger condition in the SimpleBehavior class to **`new ElapsedFrame(0)`**. Compile and run the modified program. Note the result. Change the code to remove the memory burn problem from the class. Then recompile and run the fixed program.
4. Change the scheduling bounds for the KeyNavigatorBehavior object to something smaller (e.g., a bounding sphere with a radius of 10), then run the application again. What happens when you move beyond the new bounds? Convert the scheduling bounds for KeyNavigatorApp to a universal application so that you can't get stuck at the edge of the world. See Chapter 3 for more information on BoundingLeaf nodes.
5. Use the KeyNavigatorBehavior with a TransformGroup above a visual object in the content branch graph. What is the effect?
6. Can an invisible object be picked?
7. Experiment with the various picking shapes. In particular you might want to try picking some wireframe geometry with a PickRay pick shape.
8. Extend the picking behavior in the MousePickApp by providing a callback. You can start by simply writing a text string (e.g., "picking") to the console. You can also get more ambitious and read the user data from the target transform group, toggle the use of an AlternateAppearance, or report the translation and/or rotations of the target transform group. With the proper capabilities, you can also access the children of the TransformGroup object.