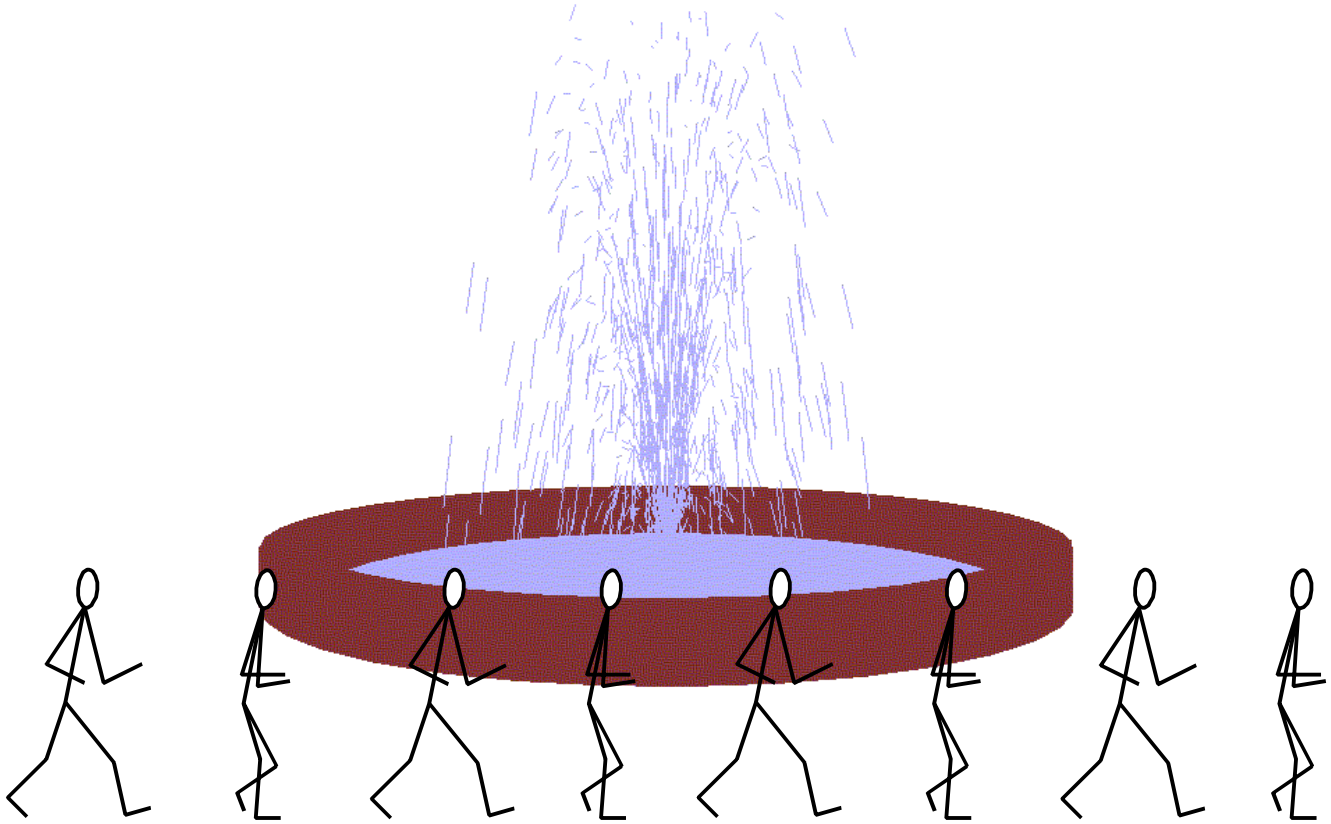
Getting Started with the Java 3D™ API

Chapter 5
Animation



Dennis J Bouvier

Tutorial version 1.6 (Java 3D API v 1.2)

Table of Contents

List of Figures

List of Code Fragments

List of Tables

List of Reference Blocks

Preface to Chapter 5

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`

New for Java 3D API version 1.2

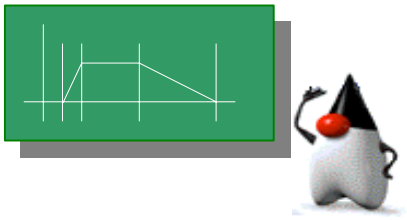
This chapter of the tutorial has been updated to include new features in Java 3D API release version 1.2. You may notice the tag **`<new in 1.2>`** to the right of some section headings and in some reference blocks in this chapter. This tag indicates the that tutorial topic is new in the API release version 1.2. Note that since chapters are updated and released individually not all of the tutorial chapters may reflect the latest version of the Java 3D API.

Cover Image

The water fountain of the cover image is a single frame of an water fountain animated via a particle system implemented using GeometryUpdater interface to BY_REFERENCE geometry. GeometryUpdater and BY_REFERENCE geometry are both new in Java 3D API v1.2. GeometryUpdater is discussed in Section 5.7. BY_REFERENCE geometry is discussed in chapter 2 of the tutorial.

The 'walking stick-figure' of the cover image represents the key frame animation possible using a Morph object and the appropriate behavior. Section 5.6 presents an example program utilizing a Morph object, an Alpha object, and a Behavior object to animate a stick man.

CHAPTER 5
Animation



Chapter Objectives

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

- use Alpha and Interpolator classes to add simple animations
- use LOD and Billboard to provide computation saving animations
- use Morph objects with custom behaviors to provide key frame animations
- create dynamic geometry creating Behaviors to modify BY_REFERENCE data

Certain visual objects change independent of user actions. For example, a clock in the virtual world should keep on ticking without user interaction. The clock is an example of animation. For the purposes of this tutorial, animation is defined as changes in the virtual universe that occur without direct user action[1]. By contrast, changes in the virtual universe as a direct result of user actions are defined as interactions. Chapter 4 presents interaction classes and programs. This chapter is about animations.

5.1 Animations

As with interaction, animations in Java 3D are implemented using Behavior objects[2]. As you might imagine, any custom animation can be created using behavior objects. However, the Java 3D API provides a number of classes useful in creating animations without having to create a new class. It should come as no surprise that these classes are based on the Behavior class.

One set of animation classes are known as interpolators. An Interpolator object, together with an Alpha object, manipulates some parameter of a scene graph object to create a time-based animation. The Alpha object provides the timing. Interpolators and Alpha objects are explained in Section 5.2.

---

[1] The distinction between animation and interaction made in this tutorial is fairly fine (*direct* is the key word here). Chapter 4 provides an example to help clarify this distinction (see "Animation versus Interaction" on page 4-3).

[2] Chapter 4 presents the Behavior class in detail and the application of Behaviors, in general. The material presented in Section 4.2 is a prerequisite for this chapter.

Another set of animation classes animates visual objects in response to view changes. This set of classes includes the *OrientedShape3D* class, *Billboard* and *Level of Detail* (LOD) behaviors which are driven not by the passage of time, but on the position or orientation of the view. The two behavior classes appear in the Java 3D core and are presented in Sections 5.3 and 5.5, respectively. Figure 5-1 shows the high level class hierarchy for Behavior based animation classes.

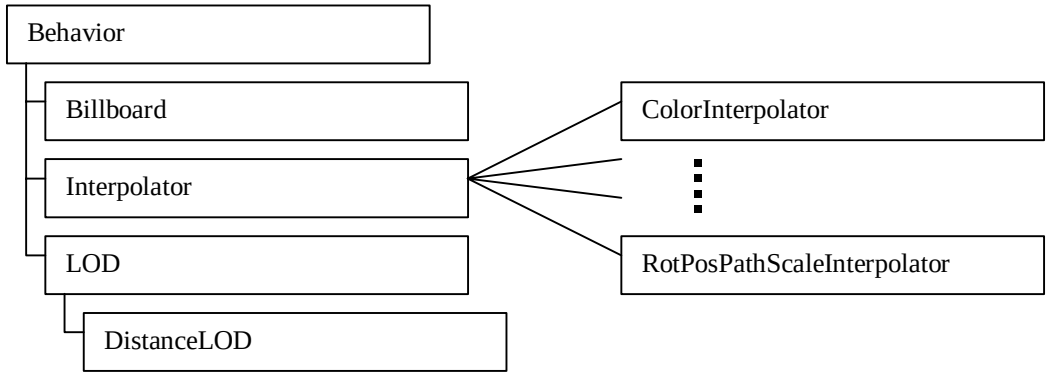


Figure 5-1 Some Classes used in Java 3D Animations

Some animations are not derived from Behavior. These include OrientedShape3D and Morph. Section 5.4 presents the OrientedShape3D class. Added in Java 3D API v1.2, OrientedShape3D provides an alternative to the Billboard behavior of early API versions. Section 5.6 presents the Morph class. The Morph class is used in both animation or interpolator applications.

5.2 Interpolators and Alpha Object Provide Time-based Animations[3]

An Alpha object produces a value between zero and one, inclusive, depending on the time and the parameters of the Alpha object. Interpolators are customized behavior objects that use an Alpha object to provide animations of visual objects. Interpolator actions include changing the location, orientation, size, color, or transparency of a visual object. All interpolator behaviors could be implemented by creating a custom behavior class; however, using an interpolator makes creating these animations much easier. Interpolator classes exist for other actions, including some combinations of these actions. The RotationInterpolator class is used in an example program in Section 5.2.3.

5.2.1 Alpha

An alpha object produces a value, called the alpha value, between 0.0 and 1.0, inclusive. The alpha value changes over time as specified by the parameters of the alpha object. For specific parameter values at any particular time, there is only one alpha value the alpha object will produce. Plotting the alpha value over time shows the waveform that the alpha object produces.

The alpha object waveform has four phases: increasing alpha, alpha at one, decreasing alpha, and alpha at zero. The collection of all four phases is one cycle of the alpha waveform. These four phases correspond with four parameters of the Alpha object. The duration of the four phases is specified by an integer value expressing the duration in milliseconds of time. Figure 5-2 shows the four phases of the alpha waveform.

[3] Section 1.9 introduced the RotationInterpolator and Alpha classes. You may want to read that section first. Also, the Java 3D API Specification covers Alpha in detail.

All alpha timings are relative to the start time for the Alpha object. The start time for all Alpha object is taken from the system start time. Consequently, Alpha objects created at different times will have the same start time. As a result, all interpolator objects, even those based on different Alpha objects, are synchronized.

Alpha objects can have their waveforms begin at different times. The beginning of an alpha object's first waveform cycle may be delayed by either or both of two other parameters: TriggerTime and PhaseDelayDuration. The TriggerTime parameter specifies a time after the StartTime to begin operation of the Alpha object. For a time specified by the PhaseDelayDuration parameter after the TriggerTime, the first cycle of the waveform begins[4]. Figure 5-2 shows the StartTime, TriggerTime and PhaseDelayDuration.

An alpha waveform may cycle once, repeat a specific number of times, or cycle continuously. The number of cycles is specified by the loopCount parameter. When the loopCount is positive, it specifies the number of cycles. A loopCount of –1 specifies continuous looping. When the alpha waveform cycles more than once, only the four cycles repeat. The phase delay is not repeated.

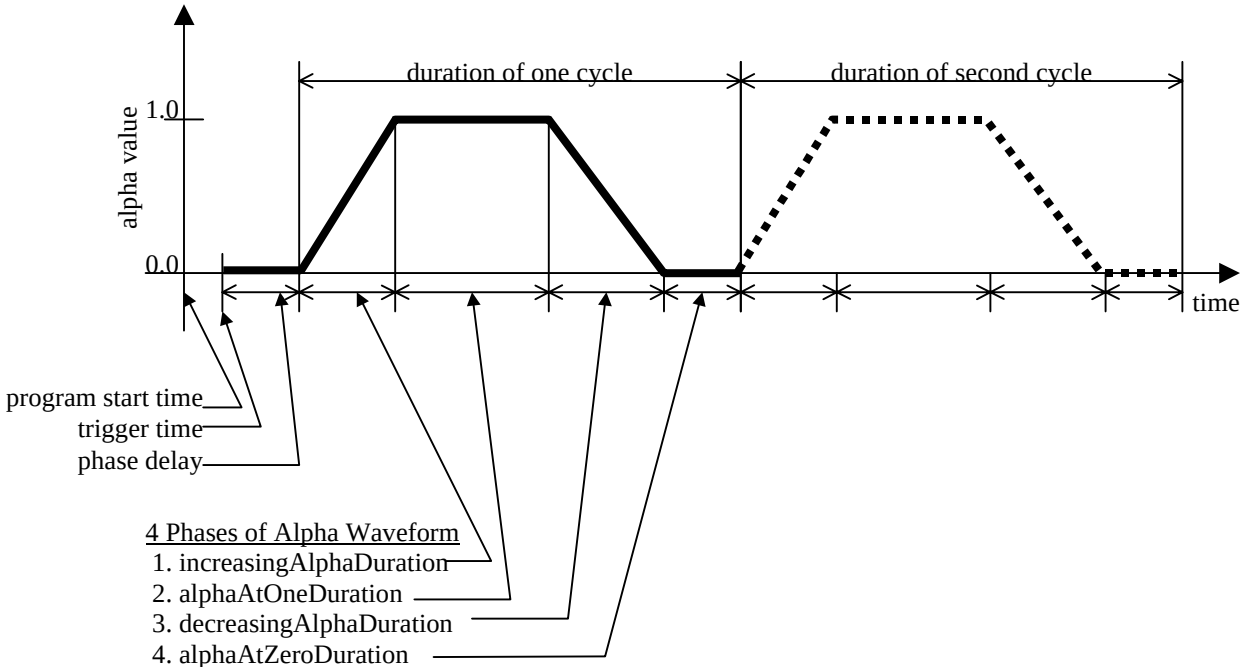


Figure 5-2 Phases of the Alpha Waveform.

An alpha waveform does not always use all four phases. An alpha waveform can be formed from one, two, three, or four phases of the Alpha waveform. Figure 5-3 shows waveforms created using one, two, or three phases of the Alpha waveform. Six of the 15 possible phase combinations are shown in the figure.

[4] Either startTime or phaseDelayDuration can be used for the same purpose. It is a rare application that requires the use of both parameters.

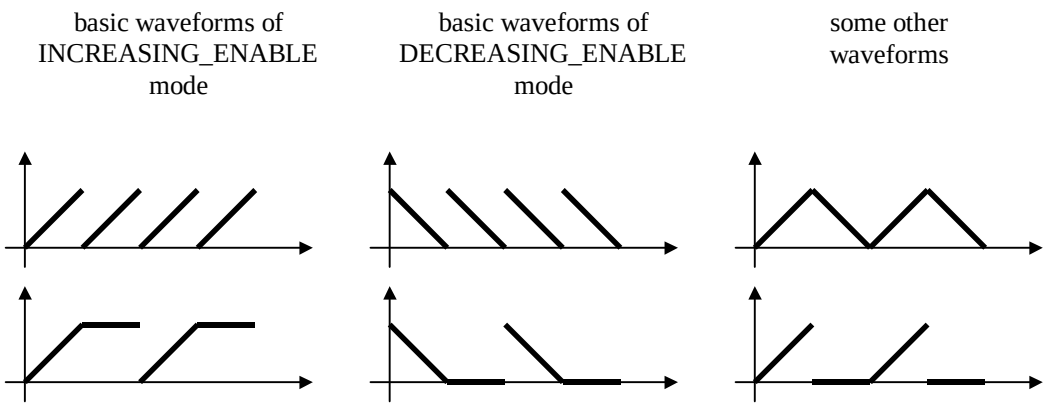


Figure 5-3 Some Basic Waveforms Easily Made with an Alpha Object.

The alpha object has two modes which specify a subset of phases are used. The INCREASING_ENABLE mode indicates the increasing alpha and alpha at one phases are used. The DECREASING_ENABLE mode indicates the decreasing alpha and alpha at zero phases are used. A third mode is the combination of these two modes indicating that all four phases are used.

The mode specification overrides the duration parameter settings. For example, when the mode is INCREASING_ENABLE, the DecreasingAlphaDuration, DecreasingAlphaRampDuration[5], and AlphaAtZeroDuration parameters are ignored. While any waveform may be specified by setting the duration of unwanted phases to zero, the proper specification of the mode increases the efficiency of the Alpha object.

5.2.2 Using Interpolator and Alpha Objects

The recipe for using Interpolator and Alpha objects is very similar to using any behavior object. The major difference from the behavior usage recipe (given in Section 4.2.2) is to include the Alpha object. Figure 5-4 gives the basic interpolator and alpha object usage recipe[6].

1. create the target object with the appropriate capability
2. create the Alpha object
3. create the Interpolator object referencing the Alpha object and target object
4. add scheduling bounds to the Interpolator object
5. add Interpolator object to the scene graph

Figure 5-4 Recipe for Using an Interpolator and Alpha Objects for Animation.

5.2.3 Example Using Alpha and RotationInterpolator

`ClockApp.java` is an example use of the RotationInterpolator class. The scene is of a clock face. The clock is rotated by a RotationInterpolator and Alpha objects once per minute. The complete code for this example is included in the `examples/Animation` subdirectory of the examples jar[7].

[5] The ramp parameters are discussed in the 'Smoothing of the Alpha Waveform' Section on page 5-6

[6] This is the same recipe as given in Section 1.9.4.

[7] The examples jar contains all of the source code for the examples in The Java 3D Tutorial; available for download from The Java 3D website.

In this application, the target object is a TransformGroup object. The ALLOW_TRANSFORM_WRITE capability is required for the TransformGroup target object. Some other interpolators act upon different target objects. For example, the target of a ColorInterpolator object is a Material object. An interpolator object sets the value of its target object based on the alpha value and values that the interpolator object holds.

The interpolator defines the end points of the animation. In the case of the RotationInterpolator, the object specifies the start and end angles of rotation. The alpha controls the animation with respect to the timing and how the interpolator will move from one defined point to the other by the specification of the phases of the alpha waveform.

This application uses the default RotationInterpolator settings of a start angle of zero and an end angle of 2Π (one complete rotation). The default axis of rotation is the y-axis. The alpha object is set to continuously rotate (loopCount = -1) with a period of one minute (60,000 milliseconds). The combination of these two objects will cause the visual object to rotate one full rotation every minute. The cycle continuously and immediately repeats. The result looks like the clock is continuously spinning, not that the clock spins once and starts over.

Code Fragment 5-1 shows the createSceneGraph method from ClockApp.java. This code fragment is annotated with the steps from the recipe of Figure 5-4.

Code Fragment 5-1 Using a RotationInterpolator and Alpha to Spin a Clock (from ClockApp).

```
1. public BranchGroup createSceneGraph() {
2.     // Create the root of the branch graph
3.     BranchGroup objRoot = new BranchGroup();
4.
5.     // create target TransformGroup with Capabilities
6.  ❶ TransformGroup objSpin = new TransformGroup();
7.     objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
8.
9.     // create Alpha that continuously rotates with a period of 1 minute
10. ❷ Alpha alpha = new Alpha (-1, 60000);
11.
12.    // create interpolator object; by default: full rotation about y-axis
13. ❸ RotationInterpolator rotInt = new RotationInterpolator(alpha, objSpin);
14. ❹ rotInt.setSchedulingBounds(new BoundingSphere());
15.
16.    //assemble scene graph
17. ❺ objRoot.addChild(objSpin);
18.    objSpin.addChild(new Clock());
19.    objRoot.addChild(rotInt);
20.
21.    // Let Java 3D perform optimizations on this scene graph.
22.    objRoot.compile();
23.
24.    return objRoot;
25.} // end of CreateSceneGraph method of ClockApp
```

Figure 5-5 is of a scene rendered by ClockApp at 4:30. The clock face is oblique to the viewer since the entire clock face is rotating.

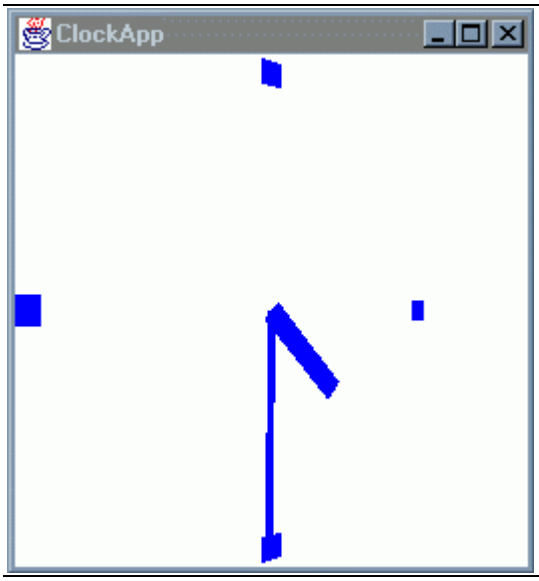


Figure 5-5 Scene Rendered at 4:30 by the ClockApp Example Program.

The ClockApp program shows a simple application of the RotationInterpolator. The Clock object, defined in `Clock.java` available in the `examples/Animation` subdirectory, shows a more advanced application of the RotationInterpolator object. The clock object in the program uses one RotationInterpolator object to animate each hand of the clock[8]. However, only one alpha object is used in the clock. It is not necessary to use one Alpha object to coordinate the hands; as noted above, all Alpha objects are synchronized on the program start time. However, sharing one Alpha object saves system memory.

Some of the potentially interesting features of the Clock Class are:

- the setting of the start and end angles for the hands,
- the setting of the axes of rotation, and
- the setting of the polygonal culling for the various components of the clock.

The source code for the clock is in `Clock.java`, also available in the `examples/Animation` subdirectory. The study of the Clock class is left to the reader.

Smoothing of the Alpha Waveform

In addition to the duration of the four phases, the programmer can specify a ramp duration for the increasing alpha and decreasing alpha phases. During the ramp duration, the alpha value changes gradually. In the case of motion interpolators, it will appear as though the visual object is accelerating and decelerating in a more natural, real world, manner.

The ramp duration value is used for both the beginning and ending portions of the phase and therefore the ramp duration is limited to half of the duration of the phase. Figure 5-6 shows an Alpha waveform with both IncreasingAlphaRampDuration and a DecreasingAlphaRampDuration. Note that the alpha value changes linearly between the two ramp periods.

[8] Since the clock has front and back facing hands, there are four hands and four RotationInterpolator objects.

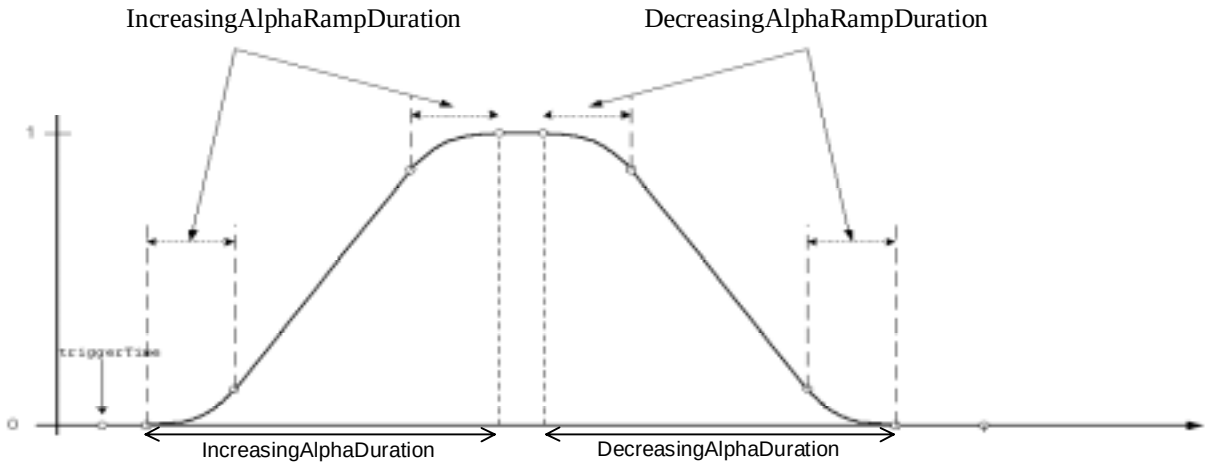


Figure 5-6 Smoothing of the Waveform Produced by Alpha[9]

An example program, `AlphaApp.java`, demonstrates the effect of an IncreasingAlphaRampDuration on an Alpha waveform. In this program there are three car visual objects. The three cars start at the same time from the same x coordinate and travel parallel. The upper car has no ramp (ramp duration = 0), the bottom car has maximum ramp duration (half of the duration of the increasing or decreasing alpha duration), and the middle car has half the maximum ramp duration (one quarter of the duration of the increasing or decreasing alpha duration). Each car takes two seconds to cross the view. In Figure 5-7 shows four scenes rendered from this application.

Figure 5-7 Four Scenes Rendered by AlphaApp Showing the Effect of IncreasingAlphaRampDuration.

At about 0.4 seconds after the cars start, the first (left) image of Figure 5-7 was captured showing the positions of the cars. The top car, which will proceed at a constant rate in the absence of a ramp, has traveled the most distance in the first frame. The other two cars begin more slowly and accelerate. At one second (not shown), all the cars have traveled the same distance. The relative positions reverse during the second half of the phase. At the end of the two second phase, each of the cars have traveled the same distance.

The source for `AlphaApp` is available in the `examples/Animation` subdirectory.

[9] Justin Couch provided the inspiration and most of the artwork for this figure.

5.2.4 Alpha API

The API of the Alpha class is straightforward. Four constructors cover the most common alpha applications. A plethora of methods, listed in the next reference block, make easy work of customizing an Alpha object to fit any application.

Alpha Constructor Summary

extends: `Object`

The alpha class converts a time value into an alpha value (a value in the range 0 to 1, inclusive). The Alpha object is effectively a function of time that generates values in the range [0,1]. A common use of the Alpha provides alpha values for Interpolator behaviors. The characteristics of the Alpha object are determined by user-definable parameters. Refer to Figure 5-2, Figure 5-6, and the text accompanying these figures for more information.

`Alpha()`
Constructs an Alpha object with mode = INCREASING_ENABLE, loopCount = -1, increasingAlphaDuration = 1000, all other parameters = 0, except StartTime. StartTime is set as the start time of the program.

`Alpha(int loopCount, long increasingAlphaDuration)`
This constructor takes only the loopCount and increasingAlphaDuration as parameters, sets the mode to INCREASING_ENABLE and assigns 0 to all of the other parameters (except StartTime).

```
Alpha(int loopCount, long triggerTime, long phaseDelayDuration,
      long increasingAlphaDuration, long increasingAlphaRampDuration,
      long alphaAtOneDuration)
```
Constructs a new Alpha object and sets the mode to INCREASING_ENABLE.

```
Alpha(int loopCount, int mode, long triggerTime, long phaseDelayDuration,
      long increasingAlphaDuration, long increasingAlphaRampDuration,
      long alphaAtOneDuration, long decreasingAlphaDuration,
      long decreasingAlphaRampDuration, long alphaAtZeroDuration)
```
This constructor takes all of the Alpha user-definable parameters.

Alpha Method Summary (partial list)

Refer to Figure 5-2, Figure 5-6, and the text accompanying these figures for more information. Each of the set-methods has a matching parameterless get-method which returns the a value of the type that corresponds to the parameter of the set-method.

`boolean finished()`
Query to test if this alpha object has finished all its activity.

`void setAlphaAtOneDuration(long alphaAtOneDuration)`
Set this alpha's alphaAtOneDuration to the specified value.

`void setAlphaAtZeroDuration(long alphaAtZeroDuration)`
Set this alpha's alphaAtZeroDuration to the specified value.

`void setDecreasingAlphaDuration(long decreasingAlphaDuration)`
Set this alpha's decreasingAlphaDuration to the specified value.

`void setDecreasingAlphaRampDuration(long decreasingAlphaRampDuration)`
Set this alpha's decreasingAlphaRampDuration to the specified value.

`void setIncreasingAlphaDuration(long increasingAlphaDuration)`
Set this alpha's increasingAlphaDuration to the specified value.

`void setIncreasingAlphaRampDuration(long increasingAlphaRampDuration)`
Set this alpha's increasingAlphaRampDuration to the specified value.

`void setLoopCount(int loopCount)`
Set this alpha's loopCount to the specified value.

`void setMode(int mode)`
Set this alpha's mode to that specified in the argument. This can be set to `INCREASING_ENABLE`, `DECREASING_ENABLE`, or the OR-ed value of the two.
`DECREASING_ENABLE` - Specifies that phases 3 and 4 are used
`INCREASING_ENABLE` - Specifies that phases 1 and 2 are used.

`void setPhaseDelayDuration(long phaseDelayDuration)`
Set this alpha's phaseDelayDuration to that specified in the argument.

`void setStartTime(long startTime)`
Sets this alpha's startTime to that specified in the argument; startTime sets the base (or zero) for all relative time computations; the default value for startTime is the system start time.

`void setTriggerTime(long triggerTime)`
Set this alpha's triggerTime to that specified in the argument.

`float value()`
This function returns a value between 0.0 and 1.0 inclusive, based on the current time and the time-to-alpha parameters established for this alpha.

`float value(long atTime)`
This function returns a value between 0.0 and 1.0 inclusive, based on the specified time and the time-to-alpha parameters established for this alpha.

5.2.5 Interpolator Behavior Classes

Figure 5-8 shows the Interpolator classes in the core and utility packages. In this figure, you can see there are over 10 interpolator classes, and that they are all subclasses of the Interpolator class. Also, the Interpolator class is an extension of Behavior. The two shaded boxes represent utility interpolator classes, the other boxes represent core interpolator classes.

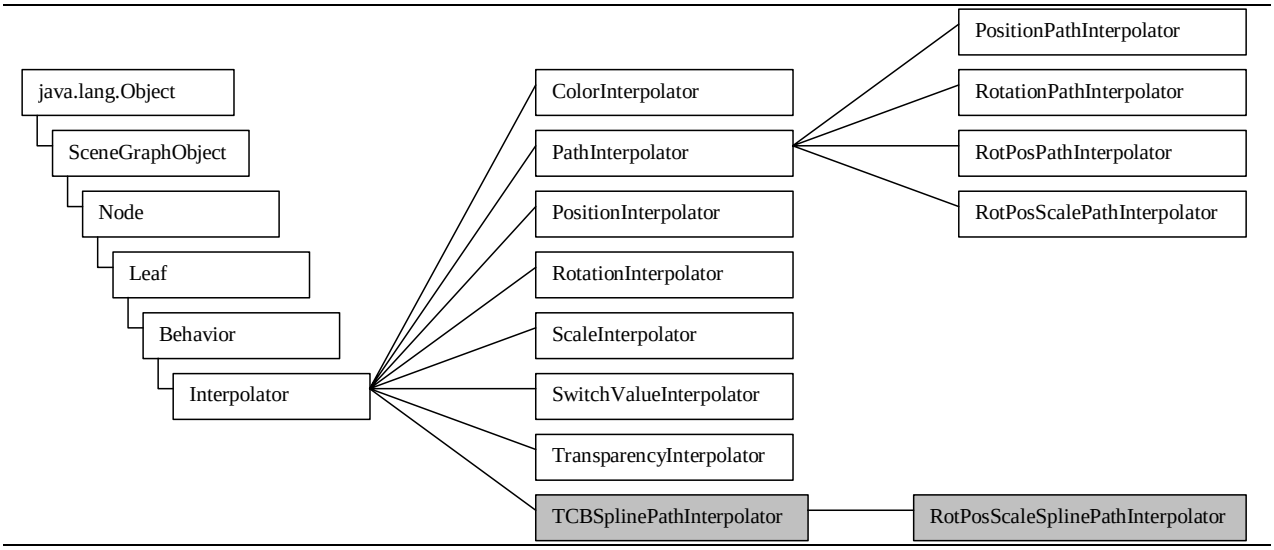


Figure 5-8 Java 3D Core and Utility (shaded boxes) Interpolator Classes Hierarchy.

Each interpolator is a custom behavior with a trigger to wake each frame. In the processStimulus method, an interpolator object checks its associated alpha object for the current alpha value, adjusts the target based on the alpha value, then resets its trigger to wake next frame (unless the alpha is finished). Some of this functionality is provided in the Interpolator class. Most of this behavior is implemented in each individual interpolator class.

Most interpolator objects store two values that are used as the end points for the interpolated action. For example, the RotationInterpolator stores two angles that are the extremes of the rotation provided by this interpolator. For each frame, the interpolator object checks the alpha value of its Alpha object and makes the appropriate rotational adjustment to its target TransformGroup object. If the alpha value is 0, then one of the values is used; if the alpha value is 1, the other value is used. For alpha values between 0 and 1, the interpolator linearly interpolates between the two values based on the alpha value and uses the resulting value for the target object adjustment.

This general interpolator description does not describe the SwitchValueInterpolator nor PathInterpolator classes well. The SwitchValueInterpolator chooses one among the children of the Switch group target node based on the alpha value; therefore, no interpolation is done in this class.

The PathInterpolator class, and its subclasses, are described in Section 5.2.7 on page 5-20.

While the various interpolator classes are similar, they also differ in some details. In summarizing the seven core subclasses of the Interpolator class, Table 5-1 shows some of the differences among interpolator classes.

Table 5-1 Summary of Core Interpolator Classes

Interpolator class	used to	target object type	page
`ColorInterpolator`	change the diffuse color of an object(s)	Material	5-12
`PathInterpolator`[10]	*abstract class*	*TransformGroup*	*5-20*
`PositionInterpolator`	change the position of an object(s)	TransformGroup	5-14
`RotationInterpolator`	change the rotation (orientation) of an object(s)	TransformGroup	5-15
`ScaleInterpolator`	change the size of an object(s)	TransformGroup	5-16
`SwitchValueInterpolator`	choose one of (switch) among a collection of objects	Switch	5-17
`TransparencyInterpolator`	change the transparency of an object(s)	TransparencyAttributes	5-19

An example program, `InterpolatorApp.java`, demonstrates six non-abstract interpolator classes of Table 5-1. In this program, each interpolator object is driven by a single Alpha object. Figure 5-9 shows two scenes rendered by InterpolatorApp. Changes in position, rotation, scale, color, transparency, and visual object (top to bottom) are made by PositionInterpolator, RotationInterpolator, ScaleInterpolator, ColorInterpolator, TransparencyInterpolator, and SwitchValueInterpolator objects, respectively. The complete source code for InterpolatorApp is available in the `examples/Animation` subdirectory of the examples distribution.

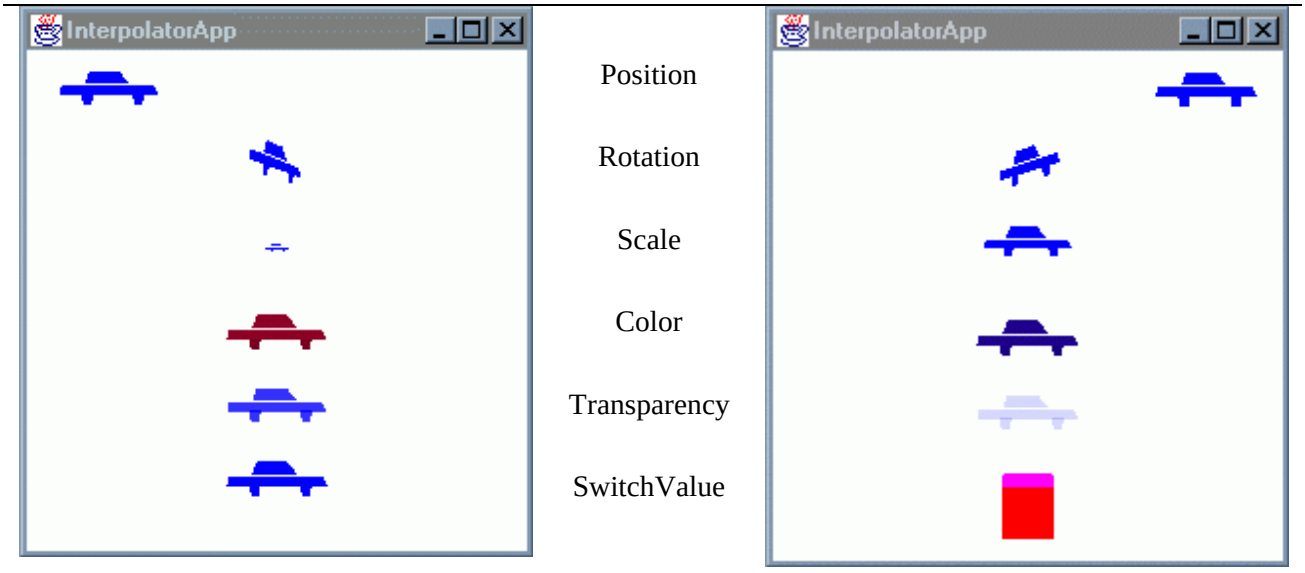


Figure 5-9 Two Scenes from InterpolatorApp Showing Various Interpolators.

[10] The PathInterpolator class is an abstract class and does not have a target object, but each of the known extensions of this interpolator have a TransformGroup target object. See Section 5.2.7 for more information.

Interpolator Programming Pitfalls

Interpolator objects are derived from, and closely related to, behavior objects. Consequently, using interpolator objects give rise to the same programming pitfalls as using behavior objects (see Programming Pitfalls of Using Behavior Objects on page 4-9). In addition to these, there are general Interpolator programming pitfalls, and specific pitfalls for some interpolator classes. Two general pitfalls are listed here while the interpolator class specific ones are listed with the appropriate class' reference blocks in the next section.

One potential interpolator programming pitfall is not realizing that interpolator objects clobber the value of its target objects. You might think that the TransformGroup target of a RotationInterpolator can be used to translate the visual object in addition to the rotation provided by the interpolator. This is not true. The transform set in the target TransformGroup object is re-written on each frame the Alpha object is active. This also means that two interpolators can not have the same target object[11].

Another general interpolator pitfall is not setting the appropriate capability for the target object. Failing to do so will result in a runtime error.

5.2.6 Core Interpolator API

As an abstract class, Interpolator is only used when creating a new subclass. The Interpolator class provides only one method for users of Interpolator subclasses. Methods useful in writing subclasses are not listed here. The majority of the information needed for writing a subclass of interpolator can be gleaned from Chapter 4.

Interpolator Method Summary (partial list)

extends: `Behavior`
known subclasses: `ColorInterpolator, PathInterpolator, PositionInterpolator, RotationInterpolator, ScaleInterpolator, SwitchValueInterpolator, TCBSplinePathInterpolator, TransparencyInterpolator`

The Interpolation behavior is an abstract class that provides the building blocks used by its various interpolation specializations.

`void setAlpha(Alpha alpha)`
Set this interpolator's alpha to the alpha object specified.

ColorInterpolator

A ColorInterpolator object has a Material object as its target. This interpolator changes the diffuse color component of the target material. This makes the ColorInterpolator both powerful and limited. The power comes from the ability of having more than one visual object share the same Material object. So, one ColorInterpolator with one Material target can affect more than one visual object. The limitation is that visual objects with a Material NodeComponent are only visible when lit.

The majority of the potential programming pitfalls are the result of the complexity of shaded (lit) scenes. Lighting is sufficiently complex that it is the subject of an entire chapter, Chapter 6. For example, the color of a shaded visual object is the combination of specular, diffuse, and ambient components. The ColorInterpolator only changes one of three components, the diffuse color, so in certain situations it is

[11] There is nothing preventing this, but only one of the interpolator objects will affect the target with the effect of the others being overwritten.

entirely possible for it to appear that the ColorInterpolator had no affect on the visual object (see Self Test question 2). Rather than get into a detailed analysis of this and other potential lighting problems here, the reader is referred to Chapter 6, specifically Sections 6.1 and 6.4.

Another less exotic potential programming pitfall is failing to add the target Material object to the Shape3D object. Figure 5-10 shows a partial scene graph diagram of a ColorInterpolator and its target Material NodeComponent.

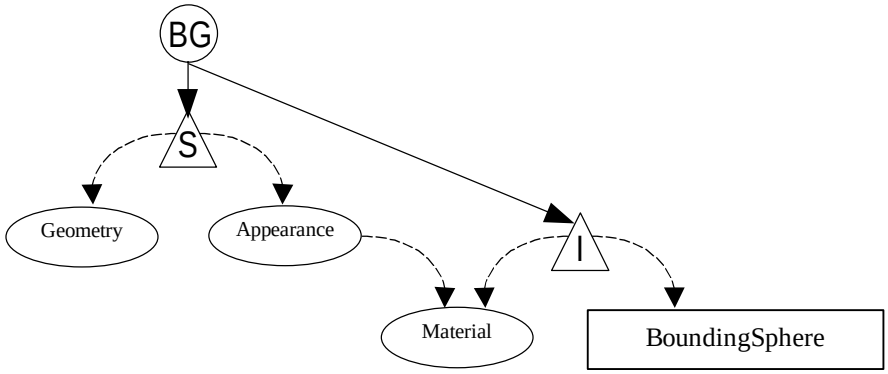


Figure 5-10 Partial Scene Graph Diagram of a ColorInterpolator Object and its Target Material NodeComponent Object.

The ColorInterpolator is different from other interpolators in the format of its get-methods. The get-methods of ColorInterpolator are not paramterless as they are with the other interpolators. Consequently, the get-methods of this class are listed with the set-methods.

ColorInterpolator Constructor Summary

extends: `Interpolator`

This class defines a behavior that modifies the diffuse color of its target material object by linearly interpolating between a pair of specified colors (using the value generated by the specified Alpha object).

`ColorInterpolator(Alpha alpha, Material target)`
Constructs a trivial color interpolator with a specified target, a starting color of black, an ending color of white.

`ColorInterpolator(Alpha alpha, Material target, Color3f startColor, Color3f endColor)`
Constructs a color interpolator with the specified target, starting color, and ending color.

ColorInterpolator Method Summary (partial list)

The get-methods do not follow the convention of other interpolators. They are listed here.

`void setEndColor(Color3f color)`
Sets the endColor for this interpolator.
matching get-method: `void getEndColor(Color3f color)`

`void setStartColor(Color3f color)`
Sets the startColor for this interpolator.
matching get-method: `void getStartColor(Color3f color)`

`void setTarget(Material target)`
Sets the target material component object for this interpolator.
matching get-method: `Material getTarget()`

PositionInterpolator

The PositionInterpolator varies the position of a visual object(s) along an axis. The specification of the end points of interpolation is made with two floating point values and an axis of translation. The default axis of translation is the x-axis.

PositionInterpolator Constructor Summary

extends: `Interpolator`

This class defines a behavior that modifies the translational component of its target TransformGroup by linearly interpolating between a pair of specified positions (using the value generated by the specified Alpha object). The interpolated position is used to generate a translation transform along the local X-axis (or the specified axis of translation) of this interpolator.

`PositionInterpolator(Alpha alpha, TransformGroup target)`
Constructs a trivial position interpolator with a specified target, with the default axis of translation (X), a startPosition of 0.0f, and an endPosition of 1.0f.

`PositionInterpolator(Alpha alpha, TransformGroup target,`
`Transform3D axisOfTranslation, float startPosition, float endPosition)`
Constructs a new position interpolator that varies the target TransformGroup's translational component (startPosition and endPosition) along the specified axis of translation.

PositionInterpolator Method Summary (partial list)

Each of the set-methods has a matching parameterless get-method which returns a value of the type corresponding to the parameter of the set-method.

`void setAxisOfTranslation(Transform3D axisOfTranslation)`
Sets the axis of translation for this interpolator.

`void setEndPosition(float position)`
Sets the endPosition for this interpolator.

`void setStartPosition(float position)`
Sets the startPosition for this interpolator.

`void setTarget(TransformGroup target)`
Sets the target for this interpolator.

RotationInterpolator

The RotationInterpolator varies the rotational orientation of a visual object(s) about an axis. The specification of the end points of interpolation is made with two floating point angle values and an axis of rotation. The default axis of rotation is the positive y-axis.

RotationInterpolator Constructor Summary

extends: `Interpolator`

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

`RotationInterpolator(Alpha alpha, TransformGroup target)`
Constructs a trivial rotation interpolator with a specified target, the default axis of rotation is used (+Y), a minimum angle of 0.0f, and a maximum angle of 2*pi radians.

`RotationInterpolator(Alpha alpha, TransformGroup target,`
`Transform3D axisOfRotation, float minimumAngle, float maximumAngle)`
Constructs a new rotation interpolator that varies the target transform node's rotational component.

RotationInterpolator Method Summary (partial list)

Each of the set-methods has a matching parameterless get-method which returns a value of the type corresponding to the parameter of the set-method.

`void setAxisOfRotation(Transform3D axisOfRotation)`
Sets the axis of rotation for this interpolator, in radians.

`void setMaximumAngle(float angle)`
Sets the maximumAngle for this interpolator, in radians.

`void setMinimumAngle(float angle)`
Sets the minimumAngle for this interpolator, in radians.

`void setTarget(TransformGroup target)`
Sets the target TransformGroup node for this interpolator.

ScaleInterpolator

The ScaleInterpolator varies the size of a visual object(s). The specification of the end points of interpolation is made with two floating point values.

ScaleInterpolator Constructor Summary

extends: `Interpolator`

Scale interpolation behavior. This class defines a behavior that modifies the uniform scale component of its target TransformGroup by linearly interpolating between a pair of specified scale values (using the value generated by the specified Alpha object). The interpolated scale value is used to generate a scale transform in the local coordinate system of this interpolator.

`ScaleInterpolator(Alpha alpha, TransformGroup target)`
Constructs a trivial scale interpolator that varies its target TransformGroup node between the two specified alpha values using the specified alpha, an identity matrix, a minimum scale = 0.1f, and a maximum scale = 1.0f.

`ScaleInterpolator(Alpha alpha, TransformGroup target, Transform3D axisOfScale, float minimumScale, float maximumScale)`
Constructs a new scaleInterpolator object that varies its target TransformGroup node's scale component between two scale values (minimumScale and maximumScale).

ScaleInterpolator Method Summary

Each of the set-methods has a matching parameterless get-method which returns a value of the type corresponding to the parameter of the set-method.

`void setAxisOfScale(Transform3D axisOfScale)`
Sets the AxisOfScale transform for this interpolator.

`void setMaximumScale(float scale)`
Sets the maximumScale for this interpolator.

`void setMinimumScale(float scale)`
Sets the minimumScale for this interpolator.

`void setTarget(TransformGroup target)`
Sets the target TransformGroup for this interpolator.

SwitchValueInterpolator

The SwitchValueInterpolator doesn't interpolate between values as other interpolators do. It selects one of the children of a Switch object for rendering. The threshold values for switching to a different child are determined by evenly dividing the 0.0 to 1.0 range by the number of children the Switch object has. Reference blocks for the Switch class have been included in the next section.

One potential programming pitfall specific to the SwitchValueInterpolator lies in the fact that the interpolator is not updated when the number of children changes in the Switch object. More importantly, the switching threshold values are determined when the SwitchValueInterpolator object is created. So, if the switch has no children before the interpolator is created, or if the number of children changes after the interpolator object is created, then number of children in the interpolator object must be updated. The advantage is that you can specify a subset of indices to be used by an interpolator. The subset is limited to a sequential set of indices.

SwitchValueInterpolator Constructor Summary

extends: `Interpolator`

This class defines a behavior that modifies the selected child of the target switch node by linearly interpolating between a pair of specified child index values (using the value generated by the specified Alpha object).

`SwitchValueInterpolator(Alpha alpha, Switch target)`
Constructs a SwitchValueInterpolator behavior that varies its target Switch node's child index between 0 and n-1, where n is the number of children in the target Switch node.

`SwitchValueInterpolator(Alpha alpha, Switch target, int firstChildIndex, int lastChildIndex)`
Constructs a SwitchValueInterpolator behavior that varies its target Switch node's child index between the two values provided.

SwitchValueInterpolator Method Summary (partial list)

Each of the set-methods has a matching parameterless get-method which returns a value of the type corresponding to the parameter of the set-method.

`void setFirstChildIndex(int firstIndex)`
Sets the firstChildIndex for this interpolator.

`void setLastChildIndex(int lastIndex)`
Sets the lastChildIndex for this interpolator.

`void setTarget(Switch target)`
Sets the target for this interpolator.

Switch

The switch class is listed here because it is used in the SwitchValueInterpolator (and later in the DistanceLOD). Switch is derived from Group and is the parent zero or more scene graph sub branches. A Switch object can select zero, one, or more, including all, of its children to be rendered. Of course a Switch object can be used without an interpolator or LOD object. The most commonly used method is the `addChild()` method derived from the Group Class.

Switch Constructor Summary

extends: `Group`

The Switch node controls which of its children will be rendered. It defines a child selection value (a switch value) that can either select a single child, or it can select 0 or more children using a mask to indicate which children are selected for rendering.

`Switch()`
Constructs a Switch node with default parameters.

`Switch(int whichChild)`
Constructs and initializes a Switch node using the specified child selection index.
`CHILD_ALL` all children are rendered
`CHILD_MASK` the childMask BitSet is used to select which children are rendered
`CHILD_NONE` no children are rendered

`Switch(int whichChild, java.util.BitSet childMask)`
Constructs and initializes a Switch node using the specified child selection index and mask.

Switch Method Summary (partial list)

Each of the set-methods has a matching parameterless get-method which returns a value of the type corresponding to the parameter of the set-method.

`void setChildMask(java.util.BitSet childMask)`
Sets the child selection mask.

`void setWhichChild(int child)`
Sets the child selection index that specifies which child is rendered.

Switch Capability Summary

`ALLOW_SWITCH_READ | WRITE`
Specifies that this node allows reading its child selection and mask values and its current child.

TransparencyInterpolator

A TransparencyInterpolator object has a TransparencyAttributes NodeComponent as its target. This interpolator changes the transparency value of the target object. More than one visual object may share one TransparencyAttributes object. So, one TransparencyInterpolator can affect more than one visual object. Also, be aware that the various transparency modes may affect the rendering performance and appearance of the visual object. Refer to the Java 3D API Specification for more information on the TransparencyAttributes Class.

A potential programming pitfall specific to the TransparencyInterpolator is failing to add the target TransparencyAttributes object to the appearance bundle of the visual object(s). This is similar to a potential ColorInterpolator problem. See Figure 5-10 for an illustration of a visual object with an appearance bundle.

TransparencyInterpolator Constructor Summary

extends: `Interpolator`

This class defines a behavior that modifies the transparency of its target TransparencyAttributes object by linearly interpolating between a pair of specified transparency values (using the value generated by the specified Alpha object).

`TransparencyInterpolator(Alpha alpha, TransparencyAttributes target)`
Constructs a trivial transparency interpolator with a specified target, a minimum transparency of 0.0f and a maximum transparency of 1.0f.

`TransparencyInterpolator(Alpha alpha, TransparencyAttributes target, float minimumTransparency, float maximumTransparency)`
Constructs a new transparency interpolator that varies the target material's transparency between the two transparency values.

TransparencyInterpolator Method Summary

Each of the set-methods has a matching parameterless get-method which returns a value of the type corresponding to the parameter of the set-method.

`void setMaximumTransparency(float transparency)`
Sets the maximumTransparency for this interpolator.

`void setMinimumTransparency(float transparency)`
Sets the minimumTransparency for this interpolator.

`void setTarget(TransparencyAttributes target)`
Sets the target TransparencyAttributes object for this interpolator.

5.2.7 Path Interpolator Classes

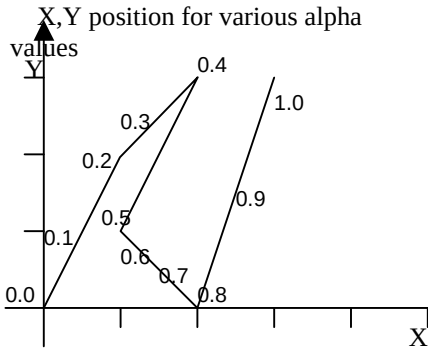
Path interpolator classes differ from the other interpolators in that they may store two or more values for interpolation. The Java 3D core provides path interpolator classes for position interpolation, rotation interpolation, position and rotation interpolation, and position, rotation, and scale interpolation. The target of a path interpolator object is a TransformGroup object which changes the position, orientation, and scale, as appropriate, for its child objects.

Path interpolator objects store a set of values, or knots, that are used two at a time for interpolation. The alpha value determines which two knot values are used. The knot values are in the range of 0.0 to 1.0 inclusive, which corresponds to the range of values of the alpha object. The first knot must have a value of 0.0 and the last knot must have a value of 1.0. The remaining knots must be stored in increasing order in the path interpolator object.

The knot values correspond with values for the variable parameter(s) (e.g., position or rotation) used in interpolation. There is one parameter value specified for each knot value. The knot with the largest value equal or less than the alpha value, and the next knot, are used. The knots are specified in order, so as the alpha value changes, the knots are used in adjacent pairs.

The left panel of Figure 5-11 shows the knot values for a position path interpolator. For illustrative purposes, only 2D positions are used. The center panel of the figure maps the position of the visual object over the alpha values, 0.0 to 1.0. The right panel of the figure shows the knot values used for the various alpha values of this example. The combination of knot values and alpha parameters determines the animation.

knot	knot value	position(x, y, z)
0	0.0	(0.0, 0.0, 0.0)
1	0.2	(1.0, 2.0, 0.0)
2	0.4	(2.0, 3.0, 0.0)
3	0.5	(1.0, 1.0, 0.0)
4	0.8	(2.0, 0.0, 0.0)
5	1.0	(3.0, 3.0, 0.0)



alpha value (a)	knots used
0.0	0
0.0 < a < 0.2	0, 1
0.2	1
0.2 < a < 0.4	1, 2
0.4	2
0.4 < a < 0.5	2, 3
0.5	3
0.5 < a < 0.8	3, 4
0.8	4
0.8 < a < 1.0	4, 5
1.0	5

Figure 5-11 The Relationship Between Knots and Alpha Value for a 2D Position Example.

PathInterpolator Example Application

Using a path interpolator object follows the same recipe as other interpolator objects. The only difference is in the number of values used to initialize the path interpolator object. Figure 5-12 presents the path interpolator recipe.

1. create the target object with the appropriate capability
2. create the Alpha object
3. create arrays of knot and other values
4. create the path interpolator object referencing the Alpha object, target object, and arrays of settings
5. add scheduling bounds to the Interpolator object
6. add path interpolator object to the scene graph

Figure 5-12 Recipe for Using a Path Interpolator Object

The `RotPosPathApp.java` example program uses an RotPosPathInterpolator object to animate a ColorCube object through a number of position and rotation values. The RotPosPathInterpolator stores sets of rotations (as an array of Quat4f), positions (as an array of Point3f), and knot values (as an array of float). The complete source for `RotPosPathApp.java` is available in the `examples/Animation` subdirectory. Code Fragment 5-2 shows an excerpt of the example annotated with the recipe step numbers.

Code Fragment 5-2 An Excerpt from the CreateSceneGraph Method of RotPosPathApp.java.

```
1. public BranchGroup createSceneGraph() {
2.     BranchGroup objRoot = new BranchGroup();
3.
4.     TransformGroup target = new TransformGroup();                    ❶
5.     Alpha alpha = new Alpha(-1, 10000);                              ❷
6.     Transform3D axisOfRotPos = new Transform3D();
7.     float[] knots = {0.0f, 0.3f, 0.6f ,1.0f};
8.     Quat4f[] quats = new Quat4f[4];                                  ❸
9.     Point3f[] positions = new Point3f[4];
10.
11.    target.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);      ❶
12.
13.    AxisAngle4f axis = new AxisAngle4f(1.0f,0.0f,0.0f,0.0f);
14.    axisOfRotPos.set(axis);
15.
16.    quats[0] = new Quat4f(0.0f, 1.0f, 1.0f, 0.0f);
17.    quats[1] = new Quat4f(1.0f, 0.0f, 0.0f, 0.0f);
18.    quats[2] = new Quat4f(0.0f, 1.0f, 0.0f, 0.0f);                   ❸
19.
20.    positions[0]= new Point3f(  0.0f,  0.0f, -1.0f);
21.    positions[1]= new Point3f(  1.0f, -1.0f, -2.0f);
22.    positions[2]= new Point3f( -1.0f,  1.0f, -3.0f);
23.
24.    RotPosPathInterpolator rotPosPath = new RotPosPathInterpolator(  ❹
25.                alpha, target, axisOfRotPos, knots, quats, positions);
26.    rotPosPath.setSchedulingBounds(new BoundingSphere());            ❺
27.
28.    objRoot.addChild(target);
29.    objRoot.addChild(rotPosPath);                                    ❻
30.    target.addChild(new ColorCube(0.4));
31.
32.    return objRoot;
33.} // end of createSceneGraph method of RotPosPathApp
```

Code Fragment 5-2 is based on the createSceneGraph method in `RotPosPathApp.java`. The difference is in the number of knots shown in the code fragment and used in the example program. `RotPosPathApp.java` defines nine knots while Code Fragment 5-3 only shows three. Figure 5-13 shows an image from RotPosPathApp. In the application, a red point is displayed for each of the nine knot positions. One position is reused, thus the eight red dots in the figure.

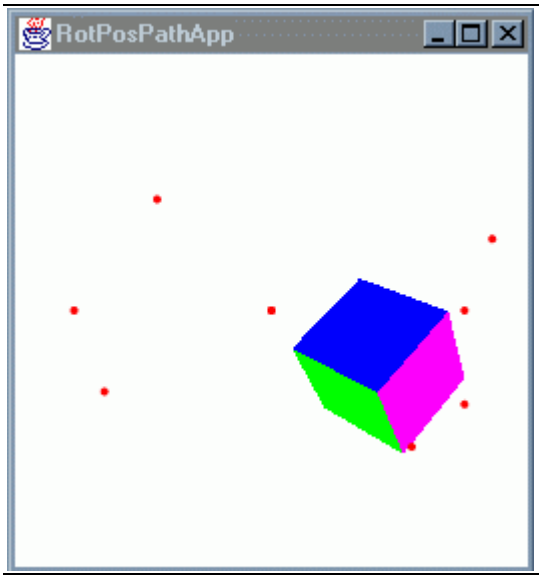


Figure 5-13 A Scene Rendered by RotPosPathApp Showing the Interpolation of the Rotation and Position of the Color Cube. The Red Dots Show the Knots Positions of the Example Application.

When the RotPosPathApp example program is run, ColorCube moves from knot position to knot position while rotating to achieve the various knot rotations. As with all interpolators, the resulting animation depends on the combination of interpolator values and the Alpha parameters used.

As mentioned before, there are a variety of subclasses of the PathInterpolator Class. In addition to these subclasses in the Java 3D core, there are a couple of related classes in the utility package. The TCBPathSplineInterpolator Class is a class similar to PathInterpolator. It has one subclass in the utility package. Refer back to Figure 5-8 to see the relationships among the interpolator classes.

In the RotPosPathApp example, the animation does not appear natural mainly due to the combination of knot positions chosen. The ColorCube moves to each knot position specified and as soon as that position is reached, the motion suddenly changes to achieve the next position. This does not appear natural as this type of action does not happen in the real world where all objects have some inertia.

TCBPathSplineInterpolator is a utility class that provides behavior and functionality similar to that of the PathInterpolator Class, but smooths the path of the visual object into that of a spline based on the knot position. The spline path mimics the real world motion of objects. On the spline motion path, the visual object may not pass through all (or any) of the specified knot positions. An example program using this class, `SplineAnim.java`, is distributed with the Java 3D API examples and can be found in the `jdk1.2/demo/java3d/SplineAnim` subdirectory.

PathInterpolator

PathInterpolator is an abstract class providing the basic interface and functionality of its subclasses. PathInterpolator objects store the knot values and calculates the index of knot values to be used based on the current alpha value.

PathInterpolator

extends `Interpolator`
Direct Known Subclasses: `PositionPathInterpolator, RotationPathInterpolator, RotPosPathInterpolator, RotPosScalePathInterpolator`

This abstract class defines the base class for all Path Interpolators. Subclasses have access to the method to compute the currentInterpolationValue given the current time and alpha. The method also computes the currentKnotIndex, which is based on the currentInterpolationValue. The currentInterpolationValue is calculated by linearly interpolating among a series of predefined knots (using the value generated by the specified Alpha object).

The first knot must have a value of 0.0 and the last knot must have a value of 1.0. An intermediate knot with index k must have a value strictly greater than any knot with index less than k.

PathInterpolator Method Summary (partial list)

`int getArrayLengths()`
This method retrieves the length of the knots array.

`void setKnot(int index, float knot)`
This method sets the knot at the specified index for this interpolator.

`void setKnots(float[] knots)` **`<new in 1.2>`**
This method sets (replaces) all the knots for this interpolator.

RotPosPathInterpolator

A RotPosPathInterpolator object varies the rotation and position of a visual object based on a set of knot values. The constructor is the most important of the API features of this class. In the constructor all of the values and related objects are specified. Be aware that each of the arrays must be the same length in this and all PathInterpolator objects.

RotPosPathInterpolator Constructor Summary

extends `PathInterpolator`

RotPosPathInterpolator behavior. This class defines a behavior that modifies the rotational and translational components of its target TransformGroup by linearly interpolating among a series of predefined knot/position and knot/orientation pairs (using the value generated by the specified Alpha object). The interpolated position and orientation are used to generate a transform in the local coordinate system of this interpolator.

```
RotPosPathInterpolator(Alpha alpha, TransformGroup target,
            Transform3D axisOfRotPos, float[] knots, Quat4f[] quats,
            Point3f[] positions)
```

Constructs a new interpolator that varies the rotation and translation of the target TransformGroup's transform.

RotPosPathInterpolator Method Summary (partial list)

`void getPositions(Point3f[] positions)` **`<new in 1.2>`**
Copy all the position values from this interpolator to the array. The array must be large enough to hold the data.

`void getQuats(Quat4f[] quats)` **`<new in 1.2>`**
Copy all the quaternion values from this interpolator to the array. The array must be large enough to hold the data.

`void setAxisOfRotPos(Transform3D axisOfRotPos)`
Sets the axis of RotPos value for this interpolator.

`void setPosition(int index, Point3f position)`
Sets the position at the specified index for this interpolator.

`void setQuat(int index, Quat4f quat)`
Sets the quaternion at the specified index for this interpolator.

`void setTarget(TransformGroup target)`
Sets the target TransformGroup for this interpolator.

5.3 Billboard Class

The term "billboard" used in computer graphics context refers to the technique of automatically rotating a planar visual object such that it is always facing the viewer. The original motivation for the billboard behavior was to enable using a textured plane as a low cost replacement for complex geometry[12]. Billboard behavior is still commonly used for this application, but is also used for other purposes, such as keeping text visible from any angle in the virtual environment. In Java 3D, the billboard technique is implemented in a subclass of the Behavior Class, thus the phrase "billboard behavior" used in Java 3D literature.

The classic example application of the billboard behavior is to represent trees as textured 2D geometry such as a quad. Of course, if the quads are statically oriented, as the viewer moves, the trees' 2D nature is revealed. However, if the trees reorient themselves such that they are always viewed parallel to their surface normal, they appear as 3D objects. This is especially true if the trees are in the background of a scene or viewed at a distance.

The billboard behavior works for trees because trees look basically the same when viewed from the front, from the back, or from any angle. Since the billboard behavior makes a visual object appear exactly the same when viewed from any angle, it is appropriate to use billboards and 2D images to represent 3D objects that are geometrically symmetric about the y-axis such as cylindrical buildings, grain silos, water towers, or any cylindrical object. Billboard behavior can also be used for non-symmetric objects when viewed from sufficient distance as to hide the details of the 2D model.

Note: In Java 3D API v1.2 the functionality of Billboard behavior is largely superceded by OrientedShape3D. You may want to read this section for more descriptive information about billboarding applications, but program using OrientedShape3D.

Figure 5-14 shows a diagram of a portion of a scene graph constructed to use a Billboard node. Two TransformGroup nodes are typically used. The upper TransformGroup is used to position the visual object and may be static. The lower TransformGroup is used to orient the visual object and is

[12] "Cost" here refers to rendering cost, or the computational cost of rendering.

manipulated at runtime by the Billboard object. More programming details are given in the following sections.

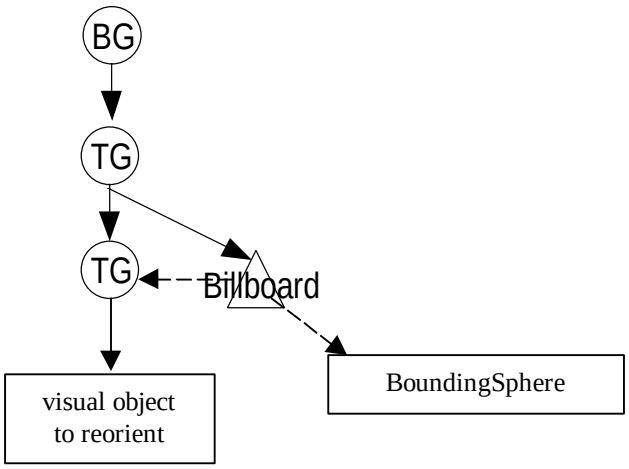


Figure 5-14 Diagram of Scene Graph Using a Billboard Object.

5.3.1 Using a Billboard Object

Using a billboard object is similar to using an interpolator except there is no Alpha object to drive the animation. The animation of the Billboard object is driven by its relative position to the viewer in the virtual world. Figure 5-15 shows the steps in the Billboard usage recipe.

1. create a target TransformGroup with ALLOW_TRANSFORM_WRITE capability
2. create a Billboard object referencing the target TransformGroup
3. supply a scheduling bounds (or bounding leaf) for the Billboard object
4. assemble the scene graph

Figure 5-15 Recipe for Using a Billboard Object to Provide Animation.

Billboard Programming Pitfalls

Even though the usage of a Billboard object is straightforward, there are a couple of potential programming mistakes. The first thing to realize it that the target TransformGroup is clobbered in each time it is updated. Consequently, this TransformGroup can not be used to position the visual object. If you attempt to use the target for placement, the billboard will work, but on the first update of rotation, the position information in the target will be lost and the visual object will appear at the origin.

Without the ALLOW_TRANSFORM_WRITE capability set for the target, a runtime error will be the result. Also, if the bounds is not set, or not set properly, the Billboard object will not animate the visual object. The scheduling bounds is typically specified by BoundingSphere with a radius great enough to enclose the visual object. Just like other behavior objects, leaving the Billboard object out of the scene graph will eliminate it from the virtual world without error or warning.

There is one limitation of the Billboard Class to be noted. In applications with more than one view, each Billboard object will animate properly for only one of the views. While this may be desired for some applications, others will certainly view it as a limitation. Java3D API v1.2 addressed this issue with the introduction of the OrientedShape3D class (Section 5.4, page 5-29). An OrientedShape3D object performs the same function as a Billboard object but performs correctly for multiple views.

A Billboard behavior either rotates about an axis or a point. In either case the Billboard behavior object manipulates a TransformGroup object such that the local +z-axis of the TransformGroup and its children face the viewer. Since the Billboard orients the local +z-axis of the geometry toward the viewer, it makes no sense to attempt to rotate about the geometry about the z-axis. For this reason, if an axis of rotation is specified as (0,0,z) then the Billboard will simply not rotate. The transformation of the target TransformGroup will be set to the identity matrix.

5.3.2 Example Billboard Program

The BillboardApp example program creates a virtual world with billboard behavior trees. Even though the trees are crudely created (from a triangle fan) they do not appear as 2D objects in the background[13].

There are two TransformGroup objects for each tree in this example. One TransformGroup, TGT, simply translates the tree into the position for the application. The TGT transform is not changed at runtime. The second TransformGroup, TGR, provides the rotation for the tree. The TGR is the target of Billboard. Code Fragment 5-3 is annotated with the steps of the recipe from Figure 5-15.

Code Fragment 5-3 Except From the createSceneGraph Method of BillboardApp.java.

```
1.     public BranchGroup createSceneGraph(SimpleUniverse su) {
2.         BranchGroup objRoot = new BranchGroup();
3.
4.         Vector3f translate = new Vector3f();
5.         Transform3D T3D = new Transform3D();
6.         TransformGroup TGT = new TransformGroup();
7.         TransformGroup TGR = new TransformGroup(); ❶
8.         Billboard billboard = null;
9.         BoundingSphere bSphere = new BoundingSphere();
10.
11.        translate.set(new Point3f(1.0f, 1.0f, 0.0f));
12.        T3D.setTranslation(translate);
13.        TGT.set(T3D);
14.
15.        // set up for billboard behavior
16.        TGR.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE); ❶
17.        billboard = new Billboard(TGR); ❷
18.        billboard.setSchedulingBounds(bSphere); ❸
19.
20.        // assemble scene graph
21.        objRoot.addChild(TGT);
22.        objRoot.addChild(billboard); ❹
23.        TGT.addChild(TGR);
24.        TGR.addChild(createTree());
25.
26.        // add KeyNavigatorBehavior(vpTrans) code removed;
27.
28.        return objRoot;
29.    } // end of CreateSceneGraph method of BillboardApp
```

Figure 5-16 shows the scene graph diagram of the objects assembled in Code Fragment 5-3.

[13] Better trees could be created from transparent textures. Textures are covered in Chapter 7.

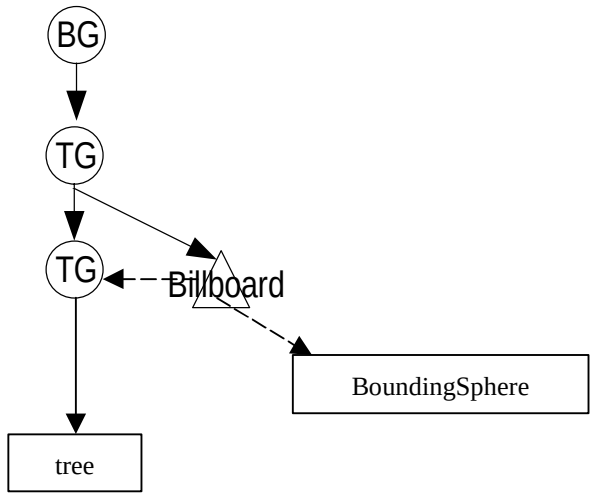


Figure 5-16 Diagram of Scene Graph Using a Billboard Object as Created in Code Fragment 5-3.

Figure 5-17 shows an image rendered from the BillboardApp example program. Code Fragment 5-3 shows the code for placing one Billboard animated tree in a virtual world. The BillboardApp program places several trees on the virtual landscape which is why four trees are visible in Figure 5-17.

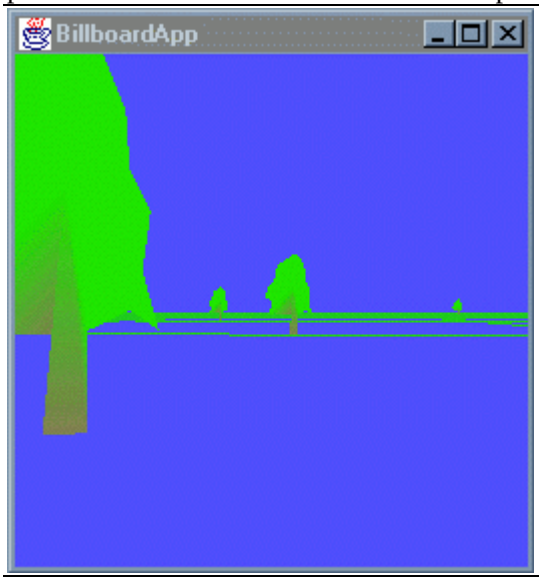


Figure 5-17 Image of BillboardApp with all 2D 'Trees' Facing the Viewer.

The BillboardApp example program provides a KeyNavigatorBehavior so that the user can move around and observe the trees from various positions and orientations. See Section 4.4.2, or all of Chapter 4, for more information on the KeyNavigatorBehavior class.

5.3.3 Billboard API

The example shows using the default mode of the Billboard object, which is to rotate about an axis. In this default mode, the visual object will be rotated about the y-axis only. So, if the trees in the BillboardApp program are viewed from above or below, their 2D geometry would be revealed.

The alternative mode is to rotate about a point. In this mode, the rotation would be about a point such that the visual object is always viewed orthogonally from any viewing position. In other words, it will never

be obvious that the visual object is two dimensional. One possible application is to represent the moon, or other distant spherical objects as a circle. Spherical objects appear as a circle when viewed from any angle.

More information on the two modes of the Billboard accompanies the summaries of constructors and methods in the next two reference blocks.

Billboard Constructor Summary

extends: `Behavior`

The Billboard behavior node operates on the TransformGroup node to cause the local +z axis of the TransformGroup to point at the viewer's eye position. This is done regardless of the transforms above the specified TransformGroup node in the scene graph. Billboard nodes provide the most benefit for complex, roughly-symmetric objects. A typical use might consist of a quadrilateral textured with the image of a tree.

`Billboard()`
Constructs a Billboard node with default parameters: mode = `ROTATE_ABOUT_AXIS`, axis `=(0,1,0)`.

`Billboard(TransformGroup tg)`
Constructs a Billboard node with default parameters that operates on the specified TransformGroup node.

`Billboard(TransformGroup tg, int mode, Vector3f axis)`
Constructs a Billboard node with the specified axis and mode that operates on the specified TransformGroup node. See `setMode()` method for an explanation of the mode parameter.

`Billboard(TransformGroup tg, int mode, Point3f point)`
Constructs a Billboard node with the specified rotation point and mode that operates on the specified TransformGroup node. See `setMode()` method for an explanation of the mode parameter.

Billboard Method Summary (partial list)

`void setAlignmentAxis(Vector3f axis)`
Sets the alignment axis.

`void setAlignmentAxis(float x, float y, float z)`
Sets the alignment axis.

`void setAlignmentMode(int mode)`
Sets the alignment mode, where mode is one of:

`ROTATE_ABOUT_AXIS` - Specifies that rotation should be about the specified axis.
`ROTATE_ABOUT_POINT` - Specifies that rotation should be about the specified point and that the children's Y-axis should match the view object's Y-axis.

`void setRotationPoint(Point3f point)`
Sets the rotation point.

`void setRotationPoint(float x, float y, float z)`
Sets the rotate point.

`void setTarget(TransformGroup tg)`
Sets the target TransformGroup object for this Billboard object.

5.4 OrientedShape3D **<new in 1.2>**

OritentedShape3D objects are used to perform the same function as the Billboard Behavior of the previous section. The major differences between these two choices are that OrientedShape3D object work for applications with more than one view where Billboards don't, and OrientedShape3D objects can be a shared object (see the SharedGroup class in the Java 3D Specification) while Billboards can't.

OrientedShape3D objects also require less code. An OrientedShape3D does not modify a TransformGroup object, thus eliminating the code for creating such. Also, since an OrientedShape3D is not a behavior, consequently it does not have an scheduling bounds to consider.

The above comparison makes the OrientedShape3D the obvious choice for all billboard applications. Simply put, it is. The only reason Billboard class is not deprecated in the API is for backward compatibility with existing applications.

A OrientedShape3D either rotates about an axis or a point. In either case the OrientedShape3D object orients itself such that the local +z-axis of its children face the viewer. Since the OrientedShape3D orients the local +z-axis of the geometry toward the viewer, it makes no sense to attempt to rotate about the geometry about the z-axis. For this reason, if an axis of rotation must not be parallel to the Z axis. That is, the rotation axis must not be specified as (0,0,z) for any value of z. It is not possible for the +Z axis to point at the viewer's eye position by rotating about itself. If an axis parallel to the Z axis is specified, the OrientedShaped3D will simply not rotate – it will be as a TransformGroup set to the identity matrix.

There is no restriction on the value that may be used as a rotation point. If the alignment mode is ROTATE_ABOUT_POINT, then the rotation will be about the specified point, with an additional rotation to align the +y axis of the TransformGroup with the +y axis in the View.

5.4.1 OrientedShape3D API

The API of the OrientedShape3D class is similar to that of the Billboard class, but different. The differences in API is due primarily to the difference in class hierarchy. OrientedShape3D extends Shape3D, not Behavior as Billboard does.

The list of constructors for OrientedShape3D is not as rich as that for Shape. For example, there is no constructor with just a Geometry parameter. Unfortunately (in this case) constructors are not inherited from the base class. An application may use the parameterless constructor and some methods to create the needed object.

OrientedShape3D Constructor Summary

extends: `Shape3D`

`OrientedShape3D()` **<new in 1.2>**
Constructs an OrientedShape3D node with default parameters.

`OrientedShape3D(Geometry geometry, Appearance appearance, int mode, Point3f point)` **<new in 1.2>**
Constructs an OrientedShape3D node with the specified geometry component, appearance component, mode, and rotation point.

`OrientedShape3D(Geometry geometry, Appearance appearance, int mode, Vector3f axis)` **<new in 1.2>**
Constructs an OrientedShape3D node with the specified geometry component, appearance component, mode, and axis.

As an extension of the Shape3D class a number of useful methods are available in OrientedShape3D such as setGeometry() and setAppearance(). The following reference block only shows the methods defined in OrientedShape3D. Refer to Chapter 1 and 2 of the tutorial or the Java 3D API Specification for the methods defined in Shape3D class.

OrientedShape3D Method Summary

`void setAlignmentAxis(float x, float y, float z)` **<new in 1.2>**
Sets the new alignment axis.

`void setAlignmentAxis(Vector3f axis)` **<new in 1.2>**
Sets the new alignment axis. There is a corresponding get method.

`void setAlignmentMode(int mode)` **<new in 1.2>**
Sets the alignment mode. The modes are **`ROTATE_ABOUT_AXIS`** and **`ROTATE_ABOUT_POINT`**. There is a corresponding get method.

`void setRotationPoint(float x, float y, float z)` **<new in 1.2>**
Sets the new rotation point.

`void setRotationPoint(Point3f point)` **<new in 1.2>**
Sets the new rotation point. There is a corresponding get method.

The following reference block lists the capabilities for OrientedShape3D.

OrientedShape3D Capability Summary

`ALLOW_AXIS_READ | WRITE` **<new in 1.2>**
allow read (write) access to its alignment axis information

`ALLOW_MODE_READ | WRITE` **<new in 1.2>**
allows read (write) access to its alignment mode information

`ALLOW_POINT_READ | WRITE` **<new in 1.2>**
allows read (write) access to its rotation point information

5.4.2 OrientedShape3D Example Application

Using an OrientedShape3D is easy. In place of a Shape3D object simply use an OrientedShape3D object.

Code Fragment 5-4 is an excerpt from OrientedShape3DApp.java[14]. This application creates a landscape populated with 2D billboard trees. Each of the trees is a child of an OrientedShape3D object giving them billboard behavior. Compare this code to that for the BillboardApp for a contrast in the ease of use (see Code Fragment 5-3).

Code Fragment 5-4 Excerpt from OrientedShape3DApp example.

```
1.    public BranchGroup createSceneGraph(SimpleUniverse su) {
2.         // Create the root of the branch graph
3.         BranchGroup objRoot = new BranchGroup();
4.
5.         Vector3f translate = new Vector3f();
6.         Transform3D T3D = new Transform3D();
7.         TransformGroup positionTG = null;
8.         OrientedShape3D orientedShape3D = null;
9.
10.        Geometry treeGeom = createTree();
11.
12.        //specify the position of the trees
13.        float[][] position = {{  0.0f, 0.0f,  -2.0f},
14.                              {-13.0f, 0.0f,  23.0f},
15.                              {  1.0f, 0.0f,  -3.5f}};
16.
17.        // for the positions in the array create a OS3D
18.        for (int i = 0; i < position.length; i++){
19.                translate.set(position[i]);
20.                T3D.setTranslation(translate);
21.                positionTG = new TransformGroup(T3D);
22.
23.                orientedShape3D = new OrientedShape3D();
24.                orientedShape3D.addGeometry(treeGeom);
25.
26.                objRoot.addChild(positionTG);
27.                positionTG.addChild(orientedShape3D);
28.        }
```

The code reproduced in Code Fragment 5-4 varies slightly from the program in the examples jar only for brevity. One feature of the application not illustrated here is one tree which is not a child of a OrientedShape3D object. See if you can find it in the application.

5.5 Level of Detail (LOD) Animations

Level of Detail (LOD) is a general term for a technique that varies the amount of detail in a visual object based on some value from the virtual world. The typical application is to vary the level of detail based on the distance to the viewer. As the distance to a visual object increases, the fewer details will appear in the rendering. So, reducing the complexity of the visual object may not affect the visual result. However, decreasing the amount of detail in the visual object when it is far from the viewer reduces the amount of rendering computation. If it is done well, a significant computational savings can be made without visual loss of content.

[14] a

.

The DistanceLOD Class provides LOD behavior based on distance to the viewer. Other possible LOD applications include varying the level of detail based on the rendering speed (frames per second) in hopes of maintaining a minimum frame rate, the speed of the visual object, or the level of detail could be controlled by user settings.

Each LOD object has one or more Switch objects as a target. As mentioned before, a Switch object is a special group that includes zero, one, or more, of its children in the scene graph for rendering (see "Switch" on page 5-18 for more information). With a DistanceLOD object, the selection of the child of the target Switch object is controlled by the distance of the DistanceLOD object to the view based on a set of threshold distances.

The threshold distances are specified in an array beginning with the maximum distance the first child of the switch target(s) will be used. The first child is typically the most detailed visual object. When the distance from the DistanceLOD object to the view is greater than this first threshold, the second child of the switch is used. Each subsequent distance threshold must be greater than the previous and specifies the distance at which the next child of the target switch is used. Thus, there are one fewer threshold distances than there are children of the switch target(s).

If more than one Switch is added as a target of the LOD object, then each Switch target is used in parallel. That is, the child of the same index is selected simultaneously for each of the Switch targets. In this way, a complex visual object can be represented by multiple geometric objects which are children of different switch nodes.

5.5.1 Using a DistanceLOD Object

Using a DistanceLOD object is similar to using an interpolator except there is no Alpha object to drive the animation. The animation of the LOD object is driven by its relative distance to the view in the virtual world; in this way using a DistanceLOD object is very similar to using a Billboard object. Using a DistanceLOD object also requires setting the threshold distances. Figure 5-18 shows the steps in the LOD usage recipe.

1. create a target Switch object(s) with ALLOW_SWITCH_WRITE capability
2. create list of distance thresholds array for the DistanceLOD object
3. create DistanceLOD object using the distance thresholds array
4. set the target switch object for the DistanceLOD object
5. supply a scheduling bounds (or bounding leaf) for the DistanceLOD object
6. assemble the scene graph, including adding children to target Switch object(s)

Figure 5-18 Recipe for Using a DistanceLOD Object to Provide Animation.

LOD Programming Pitfalls

Even thought the usage of a LOD object is straightforward, there are a couple of potential programming mistakes. The most common mistake is to fail to include the target switch object(s) in the scene graph. Setting the switch object(s) as the target(s) of the DistanceLOD object does not automatically include them in the scene graph.

Without the ALLOW_SWITCH_WRITE capability set for the target switch object(s), a runtime error will result. Also, if the bounds is not set, or not set properly, the LOD object will not animate the visual object. The scheduling bounds is typically specified by a BoundingSphere with a radius great enough to enclose the visual object. Just like other behavior objects, leaving the Billboard object out of the scene graph will eliminate it from the virtual world without error or warning.

There is one problem with the LOD classes that can not be overcome. Just like with Billboard applications, in applications that have more than one view, the LOD object will only animate properly for one of the views.

5.5.2 Example Usage of DistanceLOD

Code Fragment 5-5 shows an excerpt from the createSceneGraph method in the DistanceLODApp. The program can be found in the `examples/Animation` subdirectory in the examples jar distribution as `DistanceLODApp.java`. The code of Code Fragment 5-5 is annotated with the steps from the recipe of Figure 5-18.

Code Fragment 5-5 Excerpt from createSceneGraph Method in DistanceLODApp.

```
1.     public BranchGroup createSceneGraph() {
2.         BranchGroup objRoot = new BranchGroup();
3.         BoundingSphere bounds = new BoundingSphere();
4.
5.         // create target TransformGroup with Capabilities
6.         TransformGroup objMove = new TransformGroup();
7.
8.         // create DistanceLOD target object ❶
9.         Switch targetSwitch = new Switch();
10.        targetSwitch.setCapability(Switch.ALLOW_SWITCH_WRITE);
11.
12.        // add visual objects to the target switch ❻
13.        targetSwitch.addChild(new Sphere(.40f, 0, 25));
14.        targetSwitch.addChild(new Sphere(.40f, 0, 15));
15.        targetSwitch.addChild(new Sphere(.40f, 0, 10));
16.        targetSwitch.addChild(new Sphere(.40f, 0,  4));
17.
18.        // create DistanceLOD object
19.        float[] distances = { 5.0f, 10.0f, 20.0f};❷
20.        DistanceLOD dLOD = new DistanceLOD(distances, new Point3f());❸
21.        dLOD.addSwitch(targetSwitch);     ❹
22.        dLOD.setSchedulingBounds(bounds); ❺
23.
24.        // assemble scene graph ❻
25.        objRoot.addChild(objMove);
26.        objMove.addChild(dLOD);         // make the bounds move with object
27.        objMove.addChild(targetSwitch); // must add switch to scene graph
28.
29.        return objRoot;
30.    } // end of CreateSceneGraph method of DistanceLODApp
```

Figure 5-19 shows the scene graph diagram for the scene graph created in Code Fragment 5-5. Note that the target Switch object is both a child of a TransformGroup object and referenced by the DistanceLOD object. Both relationships are required.

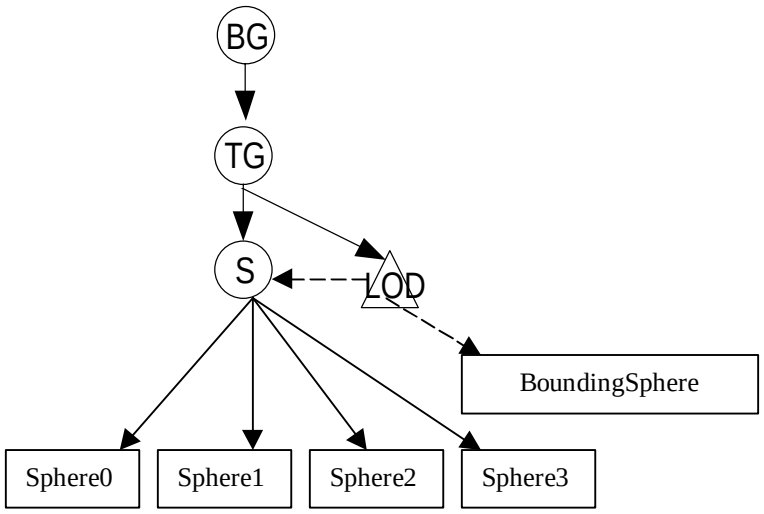


Figure 5-19 Partial Scene Graph Diagram for DistanceLODApp Example Program.

Figure 5-20 shows two scenes rendered by DistanceLODApp. Each scene has two static spheres and one sphere that moves. (In the right scene, the leftmost sphere is occluded.) The moving sphere is represented by a DistanceLOD object with four spheres of varying geometric complexity. The small green sphere is the most detailed sphere used by the DistanceLOD object at the maximum distance. The large red sphere is the least detailed sphere of the DistanceLOD object at the minimum distance. The two static spheres are included for comparison purposes.

In this application the DistanceLOD object is represented by different color spheres to illustrate the switching. Normally each visual object used by a LOD object would look as much alike as appropriate.

A PositionInterpolator is used to move the DistanceLOD object forward and back in the scene. As the DistanceLOD object moves further from the view, it switches visual objects. Without the color change in this application, it would not be easy to tell when the switching occurs.

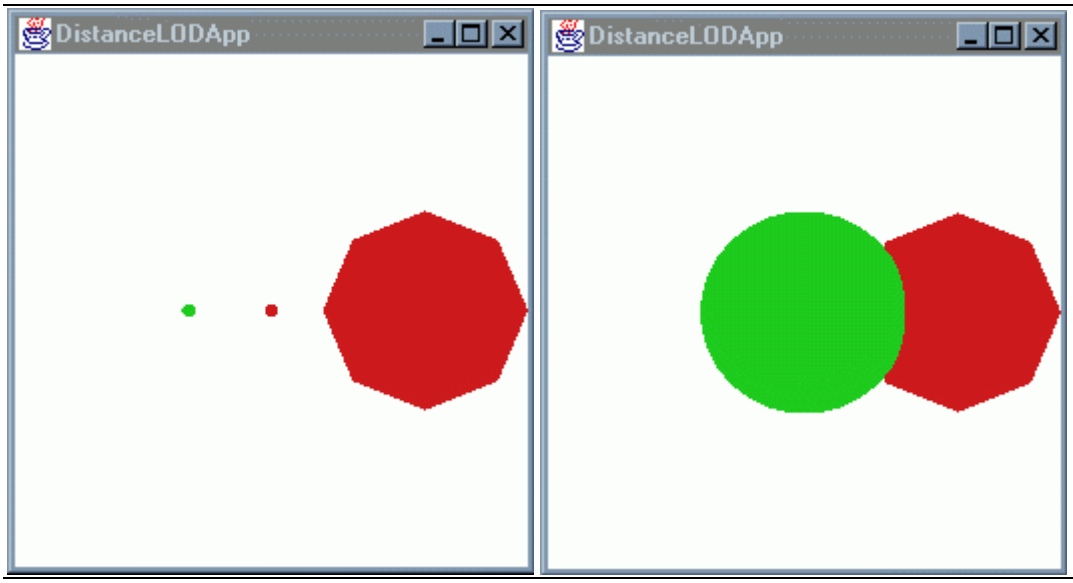


Figure 5-20 Two Scenes Rendered from DistanceLODApp.

In practice, you typically need to experiment with the threshold distances and various representations to achieve the desired visual and computational results.

5.5.3 DistanceLOD API

In Java 3D, the LOD Class provides the basic functionality for all LOD applications. The DistanceLOD Class extends the LOD Class to add the 'switch on distance to viewer' computations. Several methods of the LOD Class are necessary in the use of a DistanceLOD object. The API for the LOD Class is presented following the DistanceLOD reference blocks.

DistanceLOD Constructor Summary

This class defines a distance-based LOD behavior node that operates on a Switch group node to select one of the children of that Switch node based on the distance of this LOD node from the viewer. An array of n monotonically increasing distance values is specified, such that distances[0] is associated with the highest level of detail and distances[n-1] is associated with the lowest level of detail. Based on the actual distance from the viewer to this DistanceLOD node, these n distance values [0, n-1] select from among n+1 levels of detail [0, n]. If d is the distance from the viewer to the LOD node, then the equation for determining which level of detail (child of the Switch node) is selected is:

0, if d <= distances[0]
i, if distances[i-1] < d <= distances[i]
n, if d > distances[n-1]

Note that both the position and the array of distances are specified in the local coordinate system of this node.

`DistanceLOD()`
Constructs and initializes a DistanceLOD node with default values.

`DistanceLOD(float[] distances)`
Constructs and initializes a DistanceLOD node with the specified array of distances and a default position of (0,0,0).

`DistanceLOD(float[] distances, Point3f position)`
Constructs and initializes a DistanceLOD node with the specified array of distances and the specified position.

DistanceLOD Method Summary

`int numDistances()`
Returns a count of the number of LOD distance cut-off parameters.

`void setDistance(int whichDistance, double distance)`
Sets a particular LOD cut-off distance.

`void setPosition(Point3f position)`
Sets the position of this LOD node.

5.5.4 LOD (Level of Detail) API

As an abstract class, the LOD Class is not directly used in Java 3D programs. Methods of the LOD Class are used to manage the target Switch object(s) of a DistanceLOD object. Also, other LOD applications could be created by extending this class as appropriate.

LOD Constructor Summary

An LOD leaf node is an abstract behavior class that operates on a list of Switch group nodes to select one of the children of the Switch nodes. The LOD class is extended to implement various selection criteria.

`LOD()`
Constructs and initializes an LOD node.

LOD Method Summary

`void addSwitch(Switch switchNode)`
Appends the specified switch node to this LOD's list of switches.

`java.util.Enumeration getAllSwitches()`
Returns the enumeration object of all switches.

`void insertSwitch(Switch switchNode, int index)`
Inserts the specified switch node at specified index.

`int numSwitches()`
Returns a count of this LOD's switches.

`void removeSwitch(int index)`
Removes the switch node at specified index.

`void setSwitch(Switch switchNode, int index)`
Replaces the specified switch node with the switch node provided.

5.6 Morph

Interpolator classes change various visual attributes in the virtual world. However, there is no interpolator to change the geometry of a visual object. This is exactly what the Morph Class does. A Morph object creates the geometry for a visual object through interpolating from a set of GeometryArray objects[15]. In this way the Morph Class is like the interpolator classes. But, Morph is not an interpolator; it isn't even an extension of the Behavior class. The Morph Class extends Node.

Chapter 4 explains that all changes to a live scene graph or the objects in a live scene graph are normally made through the processStimulus method of Behavior objects. Since there is no specific behavior class for use with a Morph object, a custom behavior class must be written for a Morph application. Whether the Morph class is considered an animation or interaction class depends on the stimulus for the behavior driving the Morph object. Before getting into the details of using the Morph class, a little discussion of Morph applications is in order.

Morph objects can be used to turn pyramids into cubes, cats into dogs, or change any geometry into any other geometry. The only limitation is that the geometry objects used for interpolation are the same class, each a subclass of GeometryArray, with the same number of vertices. The restriction on the number of vertices is not as limiting as it first seems. An example program that changes a pyramid into a cube, `Pyramid2Cube.java`, is distributed with the Java 3D API.

[15] The GeometryArray Class and related classes are covered in Chapter 2.

Morph objects can also be used to animate a visual object (e.g., to make a person walk, or to make a hand grasp). An example program that animates a hand, `Morphing.java`, is also distributed with the Java 3D API examples. A third Morph example which makes a stick figure walk is the subject of the next section.

5.6.1 Using a Morph Object

To understand the usage of the Morph object requires knowing how the Morph object functions. Fortunately, a Morph object is not very complex. A Morph object stores an array of GeometryArray objects. You may recall from Chapter 2 that GeometryArray is the superclass of TriangleArray, QuadStripArray, IndexedLineStripArray, and TriangleFanArray (just to name a few).

The individual GeometryArray objects each completely defines one complete geometric specification for the visual object including color, normals, and texture coordinates. The GeometryArray objects can be thought of as key frames in an animation, or more properly, as constants in an equation to create a new GeometryArray object.

In addition to the array of GeometryArray objects, a Morph object has an array of weight values – these are the variables in the equation. Using the GeometryArray objects and the weights, a Morph object constructs a new geometry array object using the weighted average of the coordinate, color, normals, and texture coordinate information from the GeometryArray objects. Changing the weights changes the resulting geometry.

All that is required to use a Morph object is to create the array of GeometryArray objects and set the weighting values. Figure 5-18 summarizes the steps to use a Morph object.

1. create an array of GeometryArray objects
2. create a Morph object with ALLOW_WEIGHTS_WRITE
3. assemble the scene graph, including adding children to target Switch object(s)

Figure 5-21 Recipe for Using a Morph Object.

As you can see, using a morph object is not hard; however, these steps provide neither animation nor interaction. Animation or interaction is provided through a behavior object. Consequently, using a Morph object usually means writing a behavior class. Writing a custom Behavior Class is covered in Section 4.2.1., so the general details of this task are not covered here. Of course, a Morph object can be used without a behavior, but then it would not be animated. Section 5.6.2 presents a simple morph behavior class useful in creating *key frame* animations.

A Morph object can refer to an appearance bundle. The appearance bundle is used with the GeometryArray object created by the Morph object. Be aware that the Morph object always makes a GeometryArray object with per-vertex-colors. As a consequence, a ColoringAttributes color and Material diffuse color specifications are ignored. See Chapter 6 for more information on coloring and shading of visual objects.

Morph Programming Pitfalls

Even as simple as Morph usage is, there is an associated potential programming pitfall (not yet mentioned). Weights that do not sum to 1.0 results in a runtime error. I have already mentioned the appearance limitation.

5.6.2 Example Morph Application: Walking

This Morph application uses a custom behavior object to provide animation. The first step in this development is to write the custom behavior.

In a behavior used to animate a Morph object, the processStimulus method changes the weights of the Morph object. This process is only as complex as necessary to achieve the desired animation or interaction effect. In this program, The processStimulus method sets the weights values based on the alpha value from an alpha object. This happens on each frame of rendering where the trigger condition has been satisfied.

Code Fragment 5-6 shows the code for the custom behavior of the MorphApp program. In this code, only the processStimulus method is interesting.

Code Fragment 5-6 MorphBehavior Class from MorphApp.

```
1.     public class MorphBehavior extends Behavior{
2.
3.         private Morph targetMorph;
4.         private Alpha alpha;
5.         // the following two members are here for efficiency
6.         private double[] weights = {0, 0, 0, 0};
7.         private WakeupCondition trigger = new WakeupOnElapsedFrames(0);
8.
9.         // create MorphBehavior
10.        MorphBehavior(Morph targetMorph, Alpha alpha){
11.            this.targetMorph = targetMorph;
12.            this.alpha = alpha;
13.        }
14.
15.        public void initialize(){
16.            // set initial wakeup condition
17.            this.wakeupOn(trigger);
18.        }
19.
20.        public void processStimulus(Enumeration criteria){
21.            // don't need to decode event since there is only one trigger
22.            weights[0] = 0; weights[1] = 0; weights[2] = 0; weights[3] = 0;
23.
24.            float alphaValue = 4f * alpha.value();       // get alpha
25.            int alphaIndex = (int) alphaValue;           // which Geom obj
26.            weights[alphaIndex] = (double) alphaValue - (double)alphaIndex;
27.            if(alphaIndex < 3)                           // which other obj
28.                weights[alphaIndex + 1] = 1.0 - weights[alphaIndex];
29.            else
30.                weights[0] = 1.0 - weights[alphaIndex];
31.
32.            targetMorph.setWeights(weights);
33.
34.            this.wakeupOn(trigger);            // set next wakeup condition
35.        }
36.    } // end of class MorphBehavior
```

The MorphBehavior class does a key frame animation using the GeometryArray objects two at a time in a cyclical pattern. This class is suitable for any morph animation of four key frames and can be easily changed to accommodate other numbers of key frames.

With the custom behavior written, all that remains is to develop the key frames for the animation. Figure 5-22 shows the hand drawings used as the key frames for this example application. Better key frames could have been made using some 3D package.

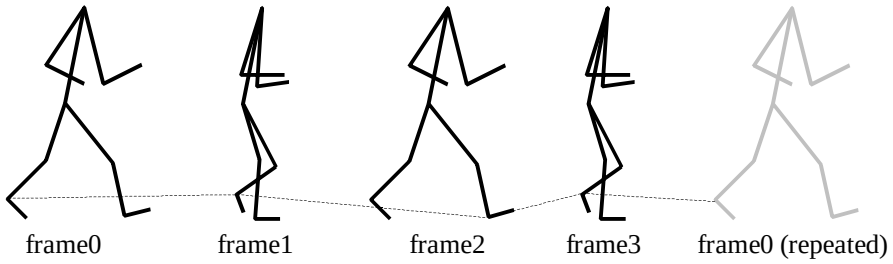


Figure 5-22 Key Frame Images from MorphApp with the Trace of One Vertex.

The black figures may look like two key frames, each repeated once, but in actuality, they are four unique key frames. The difference is in the order the vertices are specified.

Code Fragment 5-7shows and excerpt from the createSceneGraph method of `MorphApp.java` annotated with the steps of the recipe in Figure 5-21. In this method, a MorphBehavior object, Alpha object, and a Morph object are created and assembled into the scene graph. The key frame GeometryArray objects are created using some other methods (not shown here). The complete code is distributed in the examples jar.

Code Fragment 5-7 An Excerpt from the createSceneGraph Method of MorphApp.

```
1. public BranchGroup createSceneGraph() {
2.     // Create the root of the branch graph
3.     BranchGroup objRoot = new BranchGroup();
4.
5.          Transform3D t3d = new Transform3D();
6.          t3d.set(new Vector3f(0f, -0.5f, 0f));
7.          TransformGroup translate = new TransformGroup(t3d);
8.
9.          // create GeometryArray[] (array of GeometryArray objects) ❶
10.         GeometryArray[] geomArray = new GeometryArray[4];
11.         geomArray[0] = createGeomArray0();
12.         geomArray[1] = createGeomArray1();
13.         geomArray[2] = createGeomArray2();
14.         geomArray[3] = createGeomArray3();
15.
16.         // create morph object ❷
17.         Morph morphObj = new Morph(geomArray);
18.         morphObj.setCapability(Morph.ALLOW_WEIGHTS_WRITE);
19.
20.         // create alpha object
21.         Alpha alpha = new Alpha(-1, 2000);   // continuous 2 sec. period
22.         alpha.setIncreasingAlphaRampDuration(100);
23.
24.         // create morph driving behavior
25.         MorphBehavior morphBehav = new MorphBehavior(morphObj, alpha);
26.         morphBehav.setSchedulingBounds(new BoundingSphere());
27.
28.         //assemble scene graph ❸
29.         objRoot.addChild(translate);
30.         translate.addChild(morphObj);
31.         objRoot.addChild(morphBehav);
32.
```

```
33:     return objRoot;
34:     } // end of CreateSceneGraph method of MorphApp
```

It is interesting to note that a variety of animations are possible using the key frames created for this example application with different behavior classes. Figure 5-23 shows a scene rendered by Morph3App. In this program, three other behavior classes create animations based on some, or all, of the GeometryArray objects of MorphApp. They are called (left to right in the figure) "In Place", "Tango", and "Broken". Not all of the animations are good. Of course, to truly appreciate the animations, you have to run the program. The source is included in the examples jar.

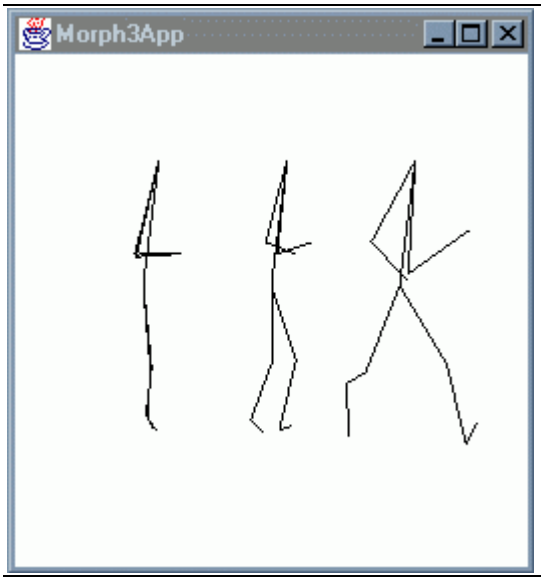


Figure 5-23 A Scene Rendered from Morph3App Showing the Animations of Three Alternative Behavior Classes (not all are good).

5.6.3 Morph API

With the simplicity of the usage recipe (Figure 5-21), you would expect a relatively simple API – and it is. The API is summarized in the next three reference blocks.

Morph Constructor Summary

extends: `Node`

Morph objects create a new GeometryArray object using the weighted average of the GeometryArray objects. If an appearance object is provided, it is used with the resulting geometry. The weights are specified with the `setWeights` method. A Morph object is usually used with a custom behavior object to adjust the weights at runtime to provide animation (or interaction).

`Morph(GeometryArray[] geometryArrays)`
Constructs and initializes a Morph object with the specified array of GeometryArray objects and a null Appearance object.

`Morph(GeometryArray[] geometryArrays, Appearance appearance)`
Constructs and initializes a Morph object with the specified array of GeometryArray objects and the specified appearance object.

Morph Method Summary (partial list)

`void setAppearance(Appearance appearance)`
Sets the appearance component of this Morph node.

`void setGeometryArrays(GeometryArray[] geometryArrays)`
Sets the geometryArrays component of the Morph node.

`void setAppearanceOverrideEnable(boolean flag)` **`<new in 1.2>`**
Set the flag to have an AlternateAppearance leaf node be the appearance for this morph node.

`void setWeights(double[] weights)`
Sets this Morph node's morph weight vector.

Morph Capabilities Summary

`ALLOW_APPEARANCE_READ | WRITE`
Specifies that the node allows read/write access to its appearance information.

`ALLOW_GEOMETRY_ARRAY_READ | WRITE`
Specifies that the node allows read/write access to its geometry information.

`ALLOW_WEIGHTS_READ | WRITE`
Specifies that the node allows read/write access to its morph weight vector.

5.7 GeometryUpdater Interface **`<new in 1.2>`**

In the previous sections the animation is accomplished primarily by moving geometry, not changing or creating geometry. The one noted exception is the Morph node which creates geometry interpolated from other geometry. Java 3D API version 1.2 introduced the GeometryUpdater interface which, along with BY_REFERENCE geometry (see Chapter 2), comes the ability to change the geometry data at runtime.

With the GeometryUpdater interface, an application programmer can produce any type of animation that depends on changing geometry information. This includes animations similar to billboard, level of detail, and morphing[16]. The GeometryUpdater interface gives the application programmer the flexibility to do much more than these techniques.

Some possible applications for using the GeometryUpdater include standard animation techniques such as inverse kinematics and particle systems. Other possibilities includes automatic shadow generation or special effects such as lightening. Since the GeometryUpdate interface provides access to all per vertex geometry data the animation possibilities are unlimited.

Keep in mind that while an application could modify BY_REFERENCE geometry data without using a GeometryUpdater object this, should be avoided as the results of doing so are neither predictable nor portable.

[16] Of course, for billboard, level of detail, and morph, it would be advisable to use the classes designed to provide these animation techniques rather than write your own.

5.7.1 Using GeometryUpdater

Using a GeometryUpdater object for dynamic geometry requires creating BY_REFERENCE geometry with the appropriate capabilities, creating a GeometryUpdater class and instantiating an object of it, and creating a custom Behavior class and instantiating an object of it. All this is not as difficult as it may seem.

Creating the BY_REFERENCE geometry is no more work than creating other geometry. The GeometryUpdater object modifies the geometry when invoked. The behavior object schedules the invocation of the GeometryUpdater object in a GeometryUpdater application.

The following reference blocks show the API features relevant to creating an application utilizing a GeometryUpdater animation. The first reference block shows the `updateData()` method that must be implemented in the class defined in the application. The second reference block shows the `updateData()` method of GeometryArray which will invoke the method implemented in the GeometryUpdater class. With the two methods having the same name, it is easy to confuse them. Hopefully, the example presented in the next section helps.

interface GeometryUpdater Method Summary

The GeometryUpdater interface is used in updating geometry data that is accessed by reference from a live or compiled GeometryArray object. Applications that wish to modify such data must define a class that implements this interface. An instance of that class is then passed to the updateData method of the GeometryArray object to be modified. This method is not directly called by the application.

`void updateData(Geometry geometry)` **`<new in 1.2>`**
Updates geometry data that is accessed by reference.

GeometryUpdater method of GeometryArray

`void updateData(GeometryUpdater updater)` **`<new in 1.2>`**
This method calls the updateData method of the specified GeometryUpdater object to synchronize updates to geometry data that is referenced by this GeometryArray object.

updater – GeometryArray object whose updateData callback method will be called to update the data referenced by this GeometryArray.

5.7.2 Fountain Particle System Example of a GeometryUpdater Application

Particle systems are often used to model water, smoke, fireworks, or other fluid-like phenomena. In a particle system there are two basic design parameters: what will the particles look like, and how will their position and orientation be updated. Often particles either appear as points or lines; however, other geometry are possible. The motion updating can either mimic the natural behavior of objects (i.e., model the laws of physics) or any other desired motion (e.g., something artistic). Usually some portion of the code has a random component to avoid all particles behaving exactly the same way.

In the example application water drops are represented as lines and the motion is that of physics (i.e., the acceleration due to gravity). A line is specified by two points.

Figure 5-24 shows a sequence of images captured from the ParticleApp example. The left image is of the fountain before any water particles have been initiated. The middle image shows the initial burst of 'water' from the fountain. In this image there are approximately 300 particles active. The right image

shows the fountain after it has been running for a while. In this image there an estimated 500 particles active.

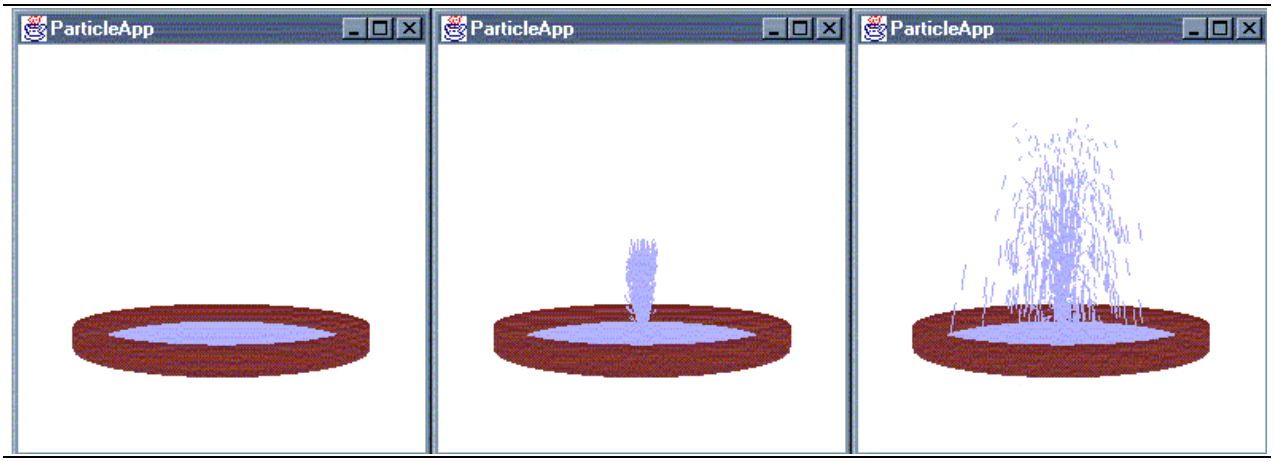


Figure 5-24 A sequence of images captured from the ParticleApp example.

In the ParticleApp.java example, the entire fountain rotates to show the 3D nature of the water animation.

In the example, the custom Behavior class and GeometryUpdater class are inner classes to Fountain class. There are other possible ways to design such an application. However, having these class inner to Fountain makes Fountain a complete animated graphic artifact. Also, since both the Behavior and GeometryUpdater classes are specific to this applications, there is no reason to make these classes more available.

Code Fragment 5-8, Code Fragment 5-9, and Code Fragment 5-10 show the definitions of the geometry, behavior, and geometryUpdater for the fountain ParticleApp, respectively. These three code fragments form the core of the animation in the ParticleApp.

Code Fragment 5-8 begins the definition of the Fountain class. In particular, this code fragment shows the definition of a few fields and the creation of the geometry used to represent the water.

Three fields are defined near the tag ❼. The fields `waterLines` and `baseElevation` are used in a number of methods, it are declared as a field of the Fountain class to make them available to the various methods. The field `waterLines` is a reference to the geometry `LineArray` that is the water particle geometry. The field `baseElevation` is the value used for the y-coordinate of the base of the fountain geometry. The usage of these field is explained in each of code fragments where it is used.

The third field holds a reference to a WaterUpdater object created here. The WaterUpdater class is defined in Code Fragment 5-10. The field is used to keep a reference to the object so as not to create a memory burn situation.

The first method of the Fountain class is `createWaterGeometry()`. In this method the definition of an int `N` is given ❶. `N` is the number of water particles (lines of the LineArray) used to represent the water. There is nothing special about the value 1400, the initial value assigned to N in the code, except that with it the animation looks reasonable. Nearly any value can be assigned to `N`. Obviously, the more water particles animated, the longer each frame will take to render. `N` is the first of the 'magic values' used in the code[17].

[17] The phrase "magic value" refers to a constant found in the code that doesn't have an obvious meaning.

In the createWaterGeometry() method the LineArray object is created on the line labeled ❷. The number of vertices is `N*2` because each particle, or line, requires a beginning vertex and an ending vertex. Note the vertex format includes BY_REFERENCE.

Water particles are animated by changing the vertex coordinate data for the lines. This is only possible when the appropriate capability is set. The first line labeled ❸ sets the capability to write the vertex data. This capability is necessary for any GeometryUpdater application. In most GeometryUpdater applications read access is also necessary. This is true in this application, thus the read vertex data capability is set in the second line labeled ❸.

The Depending on your application and how the GeometryUpdater object is designed, certain geometry information beyond the vertex data may be needed from the Geometry object. For example, if the GeometryUpdater does not 'know' how many vertices are used, the value would be read from the Geometry object passed to it. Of course, this information is only available if the appropriate capability is set. On the line following the two lines labeled ❸ the capability is set to allow reading the number of vertices. This application could be rewritten to void reading this value.

Code Fragment 5-8 Creating water particle geometry in fountain ParticleApp.

```
1.   public class Fountain extends BranchGroup{
2.
3.     protected LineArray waterLines = null;                                    ❶
4.     protected float baseElevation = -0.45f;
5.     protected GeometryUpdater geometryUpdater = new WaterUpdater();
6.
7.     Geometry createWaterGeometry(){
8.
9.       int N = 1400;                  // number of 'drops'                    ❶
10.
11.      waterLines = new LineArray(N*2,
12.                      LineArray.COORDINATES | LineArray.BY_REFERENCE);       ❷
13.
14.      waterLines.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);          ❸
15.      waterLines.setCapability(GeometryArray.ALLOW_REF_DATA_READ);           ❸
16.      waterLines.setCapability(GeometryArray.ALLOW_COUNT_READ);
17.
18.      float[] coordinates = new float[N*3*2];
19.
20.      int p;
21.      for(p = 0; p < N; p+=2){        // for each particle                   ❹
22.        coordinates[p*3+0] = 0.0f;
23.        coordinates[p*3+1] = baseElevation;
24.        coordinates[p*3+2] = 0.0f;
25.        coordinates[p*3+3] = 0.0f;
26.        coordinates[p*3+4] = baseElevation;
27.        coordinates[p*3+5] = 0.0f;
28.      }
29.      waterLines.setCoordRefFloat(coordinates);
30.      // the following statements would be redundant
31.      // waterLines.setInitialCoordIndex(0);
32.      // waterLines.setValidVertexCount(N*2);
33.
34.      return waterLines;
35.
36.    } // end of createWaterGeometry
37.
```

The remaining lines of Code Fragment 5-8 simply initializes the coordinate data for the `N` vertices. This is accomplished in the for loop beginning on the line labeled ❹. In this loop is the first use of the field `baseElevation`. Each of the vertices is initialized with coordinates `(0, baseElevation, 0)`. Thus none of the particles are initially visible.

Code Fragment 5-9 shows the definition of UpdateWaterBehavior, an extension of the Behavior class used in the example application. This is the simplest portion of the code for a GeometryUpdater application. The Behavior drives the animation by calling the updateGeometry method of the geometry to be animated when its processStimulus() method is called.

The UpdateWaterBehavior definition includes a field `w`, which is a reference to the WakeupOnElasedFrames object to be used as the behavior trigger. The WakeupOnElapsedFrames object is created in the constructor, which begins on the line labeled ❺. The initialize() method of the UpdateWaterBehavior class, beginning on the line labeled ❻, sets the initial wakeup condition for the behavior.

The `processStimulus()` method, beginning on the line labeled ❼, defines the behavior's action in response to the wakeup criteria. In this case, the `updateData()` method is invoked for `waterLines`, passing the parameter `geometryUpdater`. Here is another use of the `Fountain` fields `waterLines` and `geometryUpdater`.

If you need additional information on Behaviors read Chapter 4 of this tutorial or refer to the Java 3D API Specification.

Code Fragment 5-9 Creating update particle water behavior in fountain ParticleApp.

```
38.    class UpdateWaterBehavior extends Behavior{
39.
40.      WakeupOnElapsedFrames w = null;
41.
42.      public UpdateWaterBehavior(){                                    ❺
43.        w = new WakeupOnElapsedFrames(0);
44.      }
45.
46.      public void initialize(){                                        ❻
47.        wakeupOn(w);
48.      }
49.
50.      public void processStimulus(Enumeration critiria){               ❼
51.        waterLines.updateData(geometryUpdater);
52.        wakeupOn(w);
53.      } // end processStimulus
54.
55.    } // end class UpdateWaterBehavior
56.
```

Code Fragment 5-10 completes the fountain code fragment with the definition of the GeometryUpdater class as found in ParticleApp.java. The GeometryUpdater moves the particles by changing the vertex coordinate data.

The GeometryUpdater class defined in this application has one constructor and one method. In the constructor, tagged ⓪, a Random object is created for use in the one method of the class, the `udpateData()` method. As mentioned above, most particle systems have a randomness to them to allow more natural looking animations. The randomness of `updateData()` is provided via the Random object created here.

The `updateData()` method, tagged ①, animates the water particles. Typically not all water particles are active (i.e., moving) at the same time. Those that are inactive have y-coordinates equal to the y-coordinate of the fountain base. This particular application set the y-coordinate of the fountain base to the value of `baseElevation`. The value of `baseElevation` was picked to lower the floor of the Fountain[18]. So, in this application, when a particle has a y-coordinate equal to `baseElevation` the particle is considered inactive and consequently does not move. Initially all of the water particles are inactive – they were created that way in Code Fragment 5-8.

For the moment, consider the particle system sometime after it has started so there are some active and some inactive particles. Each time the `updateData()` method is invoked, the animation process begins by getting the relevant information about the geometry to be updated. On the line tagged ② the Geometry parameter is typecast to a GeometryArray object. On the line tagged ③ a reference to the vertex coordinate data is retrieved. On the line tagged ④ the number of vertices is recorded.

Note that this application could be made more efficient by computing this information once and storing it in fields of the object. However, the efficiency gained is minor and makes the code less reusable. This particular GeometryUpdater class can be used for fountains of other sizes (provided the fountain uses `baseElevation`).

Having made the appropriate preparations, the `updateData()` method considers each particle one at a time in the loop tagged ⑤. For those particles which are active, as determined by comparing the y-coordinate of the particle on line ⑥, the next position of the natural parabolic motion of the particle is calculated. The coordinates of the first vertex of a particle are assigned the appropriate values to model the motion, then the old coordinates of the first vertex are assigned to the second vertex.

The code is not particularly readable and exhibits additional 'magic numbers'. Part of the obfuscation is due to the indexing of the `coord[]` array. Other possible confusion could come from the calculations. The calculations (lines 73 to 78) simply perform the operations described so eloquently in the previous paragraph.

The if-statement tagged ⑦ checks if the particle proceeded below `baseElevation`. If this condition is true, then the particle is terminated by updating the coordinate values for both vertices to the initial, inactive, values.

For the inactive particles, the else part of the condition of line ⑥, some are randomly initiated as determined by the condition on line ⑧. The magic number here determines the rate of initiation of new particles. In this example an average of 20% of the inactive particles will initiate. The criticality of this magic number may surprise programmers new to particle systems. Be aware that if the initiation rate is too high, then the pool of inactive particles may be depleted. If this occurs, no particles may initiate until particles terminate, causing a pulsing in the flow of water. This behavior mimics a fountain with fixed water supply which is insufficient to provide constant maximum flow.

Since only the initial values for a particle are randomly generated, the particle's initial values determine its the entire path. The magic values scale the random values assigned to the particles as they are initiated to fit within the space of the fountain. All of this magic begins on the line tagged ⑨. Note that only the first vertex coordinates need be initialized, the coordinate values of the second vertex of an inactive particle are appropriate for a newly initiated particle.

Having processed all particles, the `updateData()` method is complete.

---

[18] Using `baseElevation` to lower the fountain is not necessarily the most portable design, but works for this application.

Code Fragment 5-10 Creating a GeometryUpdater for the fountain in ParticleApp.

```
57.   public class WaterUpdater implements GeometryUpdater{
58.
59.     Random random;
60.
61.     public WaterUpdater(){                                              ⓪
62.       random = new Random();
63.     }
64.
65.     public void updateData(Geometry geometry){                          ①
66.
67.       GeometryArray geometryArray = (GeometryArray)geometry;            ②
68.       float[] coords = geometryArray.getCoordRefFloat();                ③
69.       int N = geometryArray.getValidVertexCount();                      ④
70.
71.       int i;
72.
73.       for(i = 0; i < N; i+=2){        // for each particle              ⑤
74.         if(coords[i*3+1] > baseElevation){  // update active particles  ⑥
75.           coords[i*3+0] += coords[i*3+0] - coords[i*3+3];       //x1
76.           coords[i*3+1] += coords[i*3+1] - coords[i*3+4]-0.01f; //y1
77.           coords[i*3+2] += coords[i*3+2] - coords[i*3+5];       //z1
78.           coords[i*3+3] = (coords[i*3+0]+coords[i*3+3])/2;      //x2
79.           coords[i*3+4] = (coords[i*3+1]+coords[i*3+4]+0.01f)/2;//y2
80.           coords[i*3+5] = (coords[i*3+2]+coords[i*3+5])/2;      //z2
81.
82.           if(coords[i*3+1] < baseElevation){  // if particle below base ⑦
83.             coords[i*3+0] = 0.0f; //x1
84.             coords[i*3+1] = baseElevation; //y1
85.             coords[i*3+2] = 0.0f; //z1
86.             coords[i*3+3] = 0.0f; //x2
87.             coords[i*3+4] = baseElevation; //y2
88.             coords[i*3+5] = 0.0f; //z2
89.           }
90.         } else {                          // an inactive particle
91.           if(random.nextFloat() > 0.8){ // randomly start a drop        ⑧
92.             coords[i*3+0] = 0.03f*(random.nextFloat()-0.5f);        //x1⑨
93.             coords[i*3+1] = 0.14f*random.nextFloat()+baseElevation; //y1
94.             coords[i*3+2] = 0.03f*(random.nextFloat()-0.5f);        //z1
95.           } // end if
96.         } // end if-else
97.       } // end for loop
98.     } // end updateData(Geometry)
99.} // end of class WaterUpdater
100.
```

`ParticleApp.java` consists of approximately 400 lines of code. The 300 lines of code not reproduced here includes the creating the geometry for the base of the fountain (~130 lines), creating the main portion of the scene graph including the background and rotation interpolator (~45 lines), a little documentation, and some legal stuff. Some of these could be created in less code. In particular, the base of the fountain would probably benefit from an artist's hand with a modeler. However, even if the fountain were a model loaded, the water particle geometry, behavior, and GeometryUpdater code would all be essentially the same as it appears here.

The animation could be improved by providing a better distribution for the initial y-coordinate values. In this particle system, the initial y-coordinate also provides the initial vertical velocity and consequently

determines the maximum height of the particles. In this application initial y-values vary from 0.14 to 0.0. With a little more math, the range of values could be made to be between 0.10 and 0.14 which would provide a little more uniform water behavior.

The water would look better if the lines were drawn unaliased. This is easily accomplished in the code by using `setLineAntialiasedEnalble(true)` of a `LineAttributes` object added to the Appearance node component for the water. However, the performance penalty may not be worth the visual improvement.

Hopefully the example code and the discussion here provides you with sufficient knowledge and confidence to experiment with the UpdateGeometry feature of the Java 3D API. Be sure to provide feedback on this chapter, and the entire tutorial to SUN via the Java 3D API homepage: `java.sun.com/products/j3d`

5.8 Chapter Summary

After Section 5.1 introduces animations in Java 3D, Section 5.2 explains the application of interpolator classes with an Alpha object to add time based animations to a Java 3D virtual world. This section explains many of the various interpolator classes in detail. Sections 5.3. 5.4 and 5.5 explain the Billboard, OrientedShape3D and DistanceLOD classes, respectively, which are useful in creating animations for the purpose of reducing the rendering cost of visual objects.

In Section 5.6 the Morph class is introduced. That section also presents a complete example using a Morph object to create a key frame animation. Section 5.7 introduces the GeometryUpdater interface and explains its use through a complete particle system animation.

5.9 Self Test

1. The InterpolatorApp example program uses six different interpolator objects. Each of the interpolator objects refers to the same Alpha object. The result is to coordinate all the interpolators. What would be the result if each interpolator object had its own Alpha object? How could you change the timing?
2. If the light in InterpolatorApp is changed to `Vector3f(-0.7f,-0.7f,0.0f)` what happens? Why?
3. Why are there fewer distances than visual objects specified for a DistanceLOD object?
4. How could an OrientedShape3D object be used to have 2D geometry appear as a horizontal cylinder, such as a pipeline? How could an OrientedShape3D object be used to have 2D geometry appear as a sphere?
5. In MorphApp there are four frames of which two look like duplicates of the other two. Why are four frames necessary? Asked another way, what would the animation look like with just two frames?
6. In using a morph object, the same number of vertices are used. How can you accommodate geometric models of differing numbers of vertices?
7. There are a variety of variations that could be made to the FountainApp example program. Try changing N, the number of water particles, or turn on line antialiasing to see the performance and animation differences.