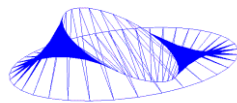


Getting Started with the Java 3D™ API

A Tutorial for Beginners

Chapter 0 Overview and Appendices

Getting Started with
the Java 3D™ API
Chapter 1



Dennis J. Bouvier

www.sun.com/javatm/3dapi/v1.2

Getting Started with
the Java 3D™ API
Chapter 2
Creating Geometry

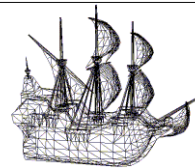


Dennis J. Bouvier

www.sun.com/javatm/3dapi/v1.2

Getting Started with
the Java 3D™ API

Chapter 3
Easier Content Creation

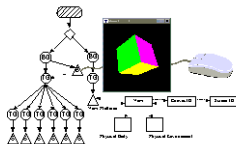


Dennis J. Bouvier

www.sun.com/javatm/3dapi/v1.2

Getting Started with
the Java 3D™ API

Chapter 4
Interaction



Dennis J. Bouvier

www.sun.com/javatm/3dapi/v1.2

Getting Started with
the Java 3D™ API

Chapter 5
Animation



Dennis J. Bouvier

www.sun.com/javatm/3dapi/v1.2

Getting Started with
the Java 3D™ API

Chapter 6
Lights



Dennis J. Bouvier

www.sun.com/javatm/3dapi/v1.2

Getting Started with
the Java 3D™ API

Chapter 7
Textures



Dennis J. Bouvier

www.sun.com/javatm/3dapi/v1.2

Dennis J. Bouvier



© 1999-2001 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A
All Rights Reserved.

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

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

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

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

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

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

Table of Contents

OVERVIEW AND APPENDICES	0-1
0.1 NAVIGATING THE TUTORIAL	0-1
0.1.1 <i>Tutorial Contents</i>	0-1
0.1.2 <i>How Can I Use the Tutorial</i>	0-5
0.1.3 <i>Preface to the Tutorial</i>	0-6
0.1.4 <i>Disclaimers</i>	0-7
0.2 (APPENDIX A) SUMMARY OF EXAMPLE PROGRAMS	0-8
0.2.1 <i>HelloJava3D</i>	0-8
0.2.2 <i>Geometry</i>	0-8
0.2.3 <i>EasyContent</i>	0-10
0.2.4 <i>Loader</i>	0-10
0.2.5 <i>Interaction</i>	0-10
0.2.6 <i>Animation</i>	0-12
0.2.7 <i>Light</i>	0-13
0.2.8 <i>Texture</i>	0-13
0.3 (APPENDIX B) REFERENCE MATERIAL	0-16
0.3.1 <i>Books</i>	0-16
0.3.2 <i>The Java 3D API can be downloaded from the Java 3D Home Page:</i>	0-16
0.3.3 <i>Sun Java Web Pages</i>	0-16
0.3.4 <i>Other Web Pages</i>	0-17
0.4 (APPENDIX C) SOLUTIONS TO SELECTED SELF TEST QUESTIONS	0-18
0.4.1 <i>Answers to Questions in Chapter 1</i>	0-18
0.4.2 <i>Answers to Questions in Chapter 2</i>	0-20
0.4.3 <i>Answers to Questions in Chapter 3</i>	0-21
0.4.4 <i>Answers to Questions in Chapter 4</i>	0-22
0.4.5 <i>Answers to Questions in Chapter 5</i>	0-23
0.4.6 <i>Answers to Questions in Chapter 6</i>	0-24
0.4.7 <i>Answers to Questions in Chapter 7</i>	0-24
0.5 (APPENDIX D) GEOMETRY AND MATH	0-26
0.5.1 <i>Defining a Plane</i>	0-26
0.6 GLOSSARY	0-27

Module 0

Overview and Appendices

Welcome to version 1.6 of The Java 3D API Tutorial. This tutorial contains seven chapters explaining the most commonly used features of the Java 3D API. The tutorial actually contains eight chapter, but chapter 0 supports the other chapters with material such as appendices and glossary.

Since the tutorial has been developed and released incrementally, several versions of the tutorial exist. For this reason the revision history may be important to readers of earlier versions. The following table presents the revision history for each chapter of the tutorial.

Table 0-1 Revision History of the Tutorial.

Chapter	latest tutorial version	release date	reflects API version
0 – Overview and Appendices	1.6.2	June 2001	1.2
1 – Getting Started	1.5.1	October 2000	1.2
2 – Creating Content	1.6	December 2000	1.2
3 – Easier Content Creation	1.6.1	March 2001	1.2
4 – Interaction	1.6	June 2001	1.2
5 – Animation	1.6	June 2001	1.2
6 – Lights	1.5	April 1999	1.1
7 – Textures	1.6	June 2001	1.2

0.1 Navigating the Tutorial

The tutorial is a collection of modules. Each Module is a collection of chapters (except this one, Module 0 has only one chapter, Chapter 0). The chapters in a Module are related. See section 0.1.2 for more information on module and chapter dependencies.

With each chapter of the tutorial being published as separate documents, the following features have been employed:

- **Appendices:** To allow easy updates to the Appendices while keeping them centrally located, including the Glossary, are published with Chapter 0 (this chapter). As a consequence, they appear as numbered sections in this document. However, the letter names used in version 1.0 of the tutorial ('A', 'B', and 'C') are retained for compatibility with the older chapters.
- **Page numbering:** To allow easy reference to pages in specific chapters, each document's page numbering is prepended with the chapter number. For example, this is page 0-1, which is page one of chapter zero.

0.1.1 Tutorial Contents

The tutorial is organized as a collection of four modules; each is outlined in the following sections. Beginning on page 0-3 the section titled "Chapter Contents" presents the contents of each chapter.

Module Overview

Module 0: Navigation and Appendices

The document you are reading is Module 0. In addition to the navigational material, it contains the appendix material and the glossary. This document will be updated with each new chapter.

Module 1: Getting Started with the Java 3D API

The introductory module presents the basics of Java 3D programming. Chapter 1 begins at the level of a Java programmer who has never used Java 3D. By the end of the Chapter 2, the novice Java 3D programmer understands the Java 3D scene graph, how to create a virtual universe by specifying a scene graph, how to create geometry, appearances, and program custom visual objects for use in simple virtual universes

$$T(dx, dy, dz) = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



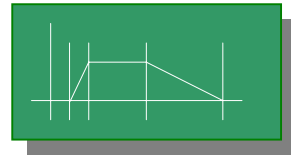
Chapter 1: Getting Started.....33 pages, October 2000

Chapter 2: Creating Content73 pages, December 2000

Chapter 3: Easier Content Creation.....37 pages, March 2001

Module 2: Interaction and Animation

Chapter 4 covers behavior basics, along with topics related to making interactive virtual worlds. Chapter 4 includes material on virtual universe navigation with the mouse and keyboard and picking. Chapter 5 continues with Morph, OrientedShape3D, and specialized behavior classes such as interpolators and level of detail to create animated visual objects.



Chapter 4: Interaction56 pages, June 2001

Chapter 5: Animation48 pages, June 2001

Module 3: Lights and Textures

Visual richness is added to the virtual universe in this module. Using lights, material properties, and textures, a Java 3D programmer can create visually rich objects without creating complex geometry.



Chapter 6: Lights34 pages, April 1999

Chapter 7: Textures43 pages, June 2001

Chapter Contents

Here is a listing of the table of contents for each of the most recently published chapters.

Module 1: Getting Started with the Java 3D API

Chapter 1: Getting Started

1.1	What is Java 3D.....	1-1
1.2	The Java 3D API	1-2
1.3	Building a Scene Graph.....	1-2
1.4	A Basic Recipe for Writing Java 3D Programs.....	1-8
1.5	Some Java 3D Terminology	1-12
1.6	Simple Recipe Example: HelloJava3Da.....	1-13
1.7	Rotating the Cube.....	1-19
1.8	Capabilities and Performance.....	1-22
1.9	Adding Animation Behavior	1-25
1.10	Chapter Summary	1-33
1.11	Self Test.....	1-33

Chapter 2: Creating Content

2.1	Virtual World Coordinate System.....	2-1
2.2	Visual Object Definition Basics.....	2-2
2.3	Geometric Utility Classes.....	2-6
2.4	Mathematical Classes.....	2-15
2.5	Geometry Classes	2-22
2.6	Appearance and Attributes	2-35
2.7	Bounds and Scope	2-48
2.8	Advanced Geometry	2-56
2.9	Clipping.....	2-67
2.10	Chapter Summary	2-73
2.11	Self Test.....	2-73

Chapter 3: Easier Content Creation

3.1	What is in This Chapter.....	3-1
3.2	GeometryInfo	3-2
3.3	Loaders	3-8
3.4	Writing a Loader	3-13
3.5	Text2D.....	3-24
3.6	Text3D.....	3-26
3.7	Background	3-33
3.8	User Data.....	3-37
3.9	Chapter Summary.....	3-37
3.10	Self Test.....	3-37

Module 2: Interaction and Animation

Chapter 4: Interaction

4.1	Behavior: The Base for Interaction and Animation	4-1
4.2	Behavior Basics.....	4-3
4.3	Wakeup Conditions: How Behaviors are Triggered	4-12
4.4	Behavior Utility Classes for Keyboard Navigation	4-25
4.5	Utility Classes for Mouse Interaction	4-29
4.6	Picking.....	4-36
4.7	Chapter Summary.....	4-57
4.8	Self Test	4-58

Chapter 5: Animation

5.1	Animations	5-1
5.2	Interpolators and Alpha Object Provide Time-based Animations	5-2
5.3	Billboard Class	5-24
5.4	OrientedShape3D <new in 1.2>	5-29
5.5	Level of Detail (LOD) Animations	5-31
5.6	Morph	5-36
5.7	GeometryUpdater Interface <new in 1.2>.....	5-41
5.8	Chapter Summary	5-48
5.9	Self Test.....	5-48

Module 3: Lights and Textures

Chapter 6: Lights

6.1	Shading in Java 3D.....	6-1
6.2	Recipe for Lit Visual Objects.....	6-4
6.3	Light Classes	6-9
6.4	Material Object.....	6-20
6.5	Surface Normals	6-24
6.6	Specifying the Influence of Lights	6-25
6.7	Creating Glow-in-the-Dark Objects, Shadows and Other Lighting Issues	6-29
6.9	Chapter Summary	6-34
6.10	Self-Test	6-34

Chapter 7: Textures

7.1	What is Texturing.....	7-1
7.2	Basic Texturing	7-2
7.3	Some Texturing Applications.....	7-15
7.4	Texture Attributes	7-17
7.5	Automatic Texture Coordinate Generation.....	7-22
7.6	Multiple Levels of Texture (Mipmaps).....	7-26
7.7	Texture, Texture2D, and Texture3D API.....	7-31
7.8	Multitexture <new in 1.2>.....	7-35
7.9	TextureLoader and NewTextureLoader API.....	7-41
7.10	Chapter Summary	7-43
7.11	Self Test.....	7-44

What is Not in the Tutorial

This tutorial is on the use of the Java 3D API. The most commonly used features of the Java 3D API are covered. Java 3D API features not covered include collisions, sensors, geometry compression, spatial sound, and multiple views. While many of the Java 3D utilities distributed with the core API are covered in the tutorial, not all are. In addition, non-API issues such as artistic considerations, specific application suggestions, and unusual display devices are also not covered.

0.1.2 How Can I Use the Tutorial

Modules are collections of related chapters. However, you may pick and choose the chapters that suit your needs. In general, chapters in the same module are dependent on the earlier chapters in the same module. For example, Chapter 2 depends on knowing the material in Chapter 1. Likewise, the reader of Chapter 5 is expected to be familiar with the topics in Chapter 4.

Module dependencies are represented in the following figure. If you have no experience with Java 3D, start with Module 1 and proceed to either Module 2 or Module 3.

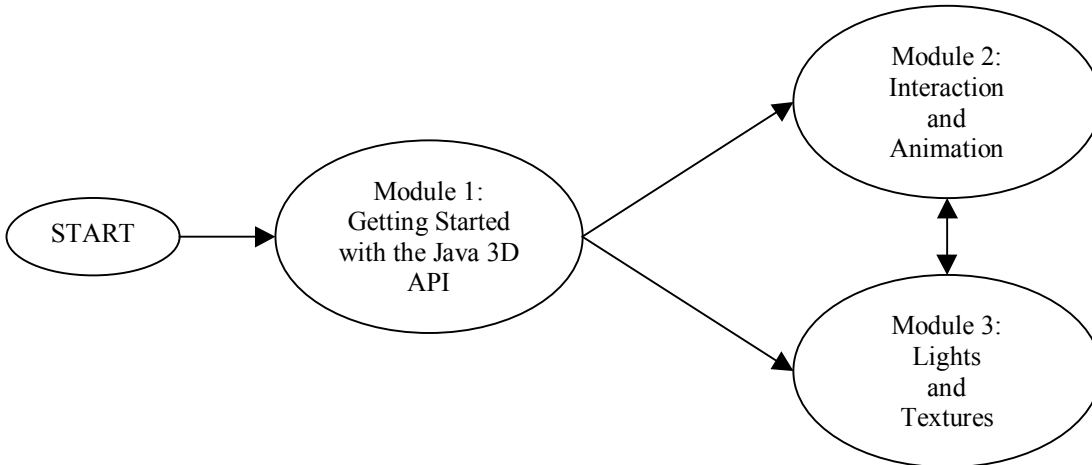


Figure 0-1 Paths Through The Java 3D Tutorial

Throughout the tutorial are *reference blocks* - summaries of the API for certain classes. The reference blocks are provided in the tutorial to make reading easier, not to replace the Java 3D API Specification Guide or any other reference.

The reference blocks were checked for accuracy when this document was published, but the Java 3D API may have changed. If you are having trouble with a program, be sure to check a current edition of the Java 3D API Specification. Also, refer to section 2.2 (page 2-4) for more information on reference blocks.

0.1.3 Preface to the Tutorial

What's Inside

This is a tutorial for the Java 3D API version 1.1.2. It is composed of the text (this document), several other text documents and a number of example applications. The text of the tutorial is available in the Acrobat (PDF) file format. The PDF files include thumbnails, links, and bookmarks making them easier to use online. The files are also readable in hardcopy form. However, several of the images are in color and details are lost when printed monochromatically.

How to download this document

The tutorial documents are available online with the source for the example programs, all of which can be downloaded from <http://java.sun.com/products/java-media/3D/collateral/>

Audience

This tutorial is meant for the Java programmer with some graphics experience, with little or no knowledge of Java 3D. If in addition to being familiar with Java you are familiar with the terms pixel, image plane, RGB, and render, then you have the background to proceed. You don't need to know about z-buffer, 3D transforms, or any other 3D graphics API to understand this tutorial, but it may help. In any case, this tutorial is written to be very accessible.

Feedback

As with all of our products, we strive for excellence in quality. If you have any questions, comments, or have an error to report, please consult the Java 3D Home Page, <http://www.java.sun.com/products/java-media/3D/>, for contact information.

Typographic Conventions

- `Courier` type is used to represent computer code and names of files and directories.
- *Italic* type is used for emphasis.
- **Bold** is used in the text to indicate program elements
- Gray background represents Reference Blocks

Double outline sections are advanced sections

Single outline sections are document meta-information sections

What software is required

Consult the Java 3D Home Page for the most current information.

Cover Image

The cover image is of a twisted strip rendered by Java 3D. The program is discussed in Section 2.6. The code is available with the examples distributed with this tutorial.

0.1.4 Disclaimers

All software associated with this tutorial is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This software included with this tutorial is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Licensee represents and warrants that it will not use or redistribute the Software for such purposes.

0.2 (Appendix A) Summary of Example Programs

This tutorial is distributed with a set of example programs. Each example program included in the distribution is described here. If you do not have the example programs, refer to the preface for download instructions.

0.2.1 HelloJava3D

HelloJava3D is a series of programs used in the first chapter of the tutorial. The complexity of these examples begins at the extreme end of simplicity and builds slightly.

examples/HelloJava3D/HelloJava3Da

This program displays a single color cube object that is static, and is neither transformed nor rotated from the origin. Consequently, it appears as a single rectangle. This program is only intended to demonstrate the basic construction of a Java 3D program. It is also used as the basis for the subsequent examples.

examples/HelloJava3D/HelloJava3Db

This program displays a single color cube object that is static but rotated from the original orientation. Consequently, more than one face of the cube is visible when rendered. This program is only intended to demonstrate the basic construction of a Java 3D program. It is also used as the basis for the subsequent examples.

examples/HelloJava3D/HelloJava3Dc

This program displays a single color cube object that is animated. The cube spins in place at the origin. Consequently, more than one face of the cube is visible as the animation takes place. This program is intended to demonstrate the basic construction of an animated Java 3D program.

examples/HelloJava3D/HelloJava3Dd

This program displays a single color cube object that is transformed and animated. The cube spins in place at the origin. Consequently, more than one face of the cube is visible as the animation takes place. This program is intended to demonstrate the basic construction of an animated Java 3D program.

0.2.2 Geometry

The `examples/Geometry` subdirectory contains the program examples for the second chapter of the tutorial. Each of these programs demonstrates something about specifying geometry for visual objects.

examples/Geometry/Axis.java

`Axis.java` defines a visual object class using an `IndexedLineArray` object used to visualize the axis. This code of this program does not appear in the text of the tutorial, it is intended as a class available for your use. The program `AxisClassDemoApp.java` uses this `Axis` class (see below).

examples/Geometry/AxisApp.java

This program displays the axis to demonstrate using `LineArray` object.

examples/Geometry/AxisClassDemoApp.java

A ColorCube orbits around the origin in this program. The code of this program does not appear in the text of the tutorial. It simply demonstrates a use of the `Axis.java` class (see above).

examples/Geometry/ColorConstants.java

This code is an example of a class that defines a number of color constants. The application spins the yo-yo about the y-axis to show the geometry.

examples/Geometry/ColorYoyoApp.java

This program displays a yo-yo using four `TriangleFanArray` geometry objects with colors. The application spins the yo-yo about the y-axis to show the geometry.

examples/Geometry/ConeYoyoApp.java

This program displays a yo-yo created with two `Cone` objects. A default `Appearance` object is used. The application spins the yo-yo about the y-axis to show the geometry.

examples/Geometry/ModelClipApp.java

This program displays the twisted strip geometry (on the cover of this chapter) clipped by a `ModelClip` object. The `ModelClip` class is new to v1.2 of the API.

examples/Geometry/MultiGeomApp.java

This program displays the three axis arrows created as three `Geometry` objects that are all referenced by a single `Shape3D` object. This capability is new in the `Shape3D` class of API v1.2.

examples/Geometry/TwistByRefApp.java

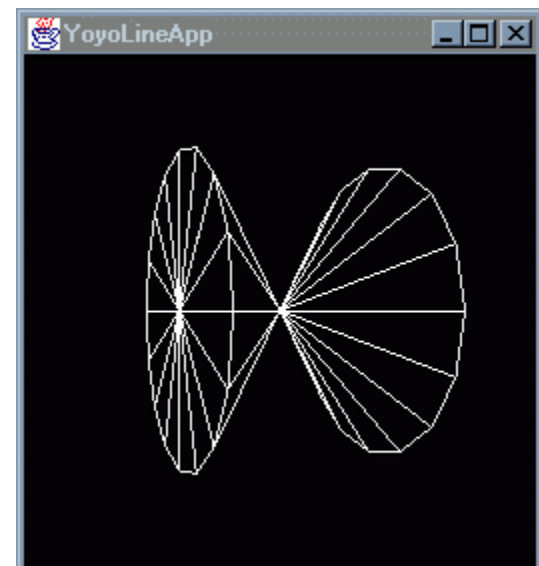
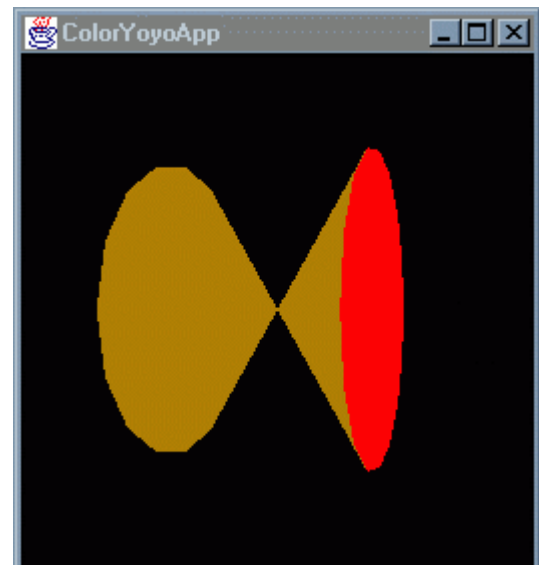
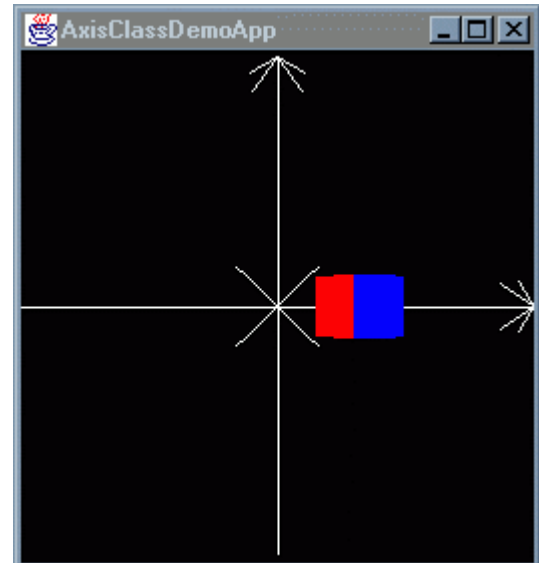
This program displays the twisted strip geometry (on the cover of this chapter) constructed using the `BY_REFERENCE` feature of `Geometry` objects. This feature is new in API v1.2.

examples/Geometry/TwistStripApp.java

This program displays a twisted strip as an example of using the `TriangleStripArray`. The twist strip program also demonstrates culling.

examples/Geometry/YoyoApp.java

This program displays a yo-yo visual object created with four `TriangleFanArray` objects. A default `Appearance` object is used. The application spins the yo-yo about the y-axis to show the geometry.



examples/Geometry/YoyoLineApp.java

This program displays the `TriangleFanArray` object with the `Appearance` set to display lines only. The application spins the yo-yo about the y-axis to show the geometry.

examples/Geometry/YoyoPointApp.java

This program displays the `TriangleFanArray` object with the `Appearance` set to points lines only. The application spins the yo-yo about the y-axis to show the geometry.

0.2.3 EasyContent**examples/easyContent/BackgroundApp.java**

This application demonstrates defining geometry for the background of a virtual world. The scene is of a grid of lines to represent the ground, a `PointArray` for stars, and a `LineArray` for a constellation. The stars and constellation are in the background. The viewer can move around in the scene and experience the relative motion between the ground and the stars in the background.

The interaction (motion) is provided through the `KeyNavigator` class (documented in Chapter 4) and a `BoundingLeaf` application bounds, which provides interaction in the virtual world without bounds. The `BoundingLeaf` is added to the view branch graph in this application.

examples/easyContent/GeomInfoApp.java

This application demonstrates the use of the `GeometryInfo` class to create Java 3D geometry specified by arbitrary polygons. This application creates the surface of a car using polygons. The `Triangulator`, `Stripifier`, and `NormalGenerator` classes are used to convert the polygons into triangle strips with normals so the specified geometry can be shaded. A wire frame view of the geometry can be viewed by providing any command line argument when invoking the program. For example: `java GeomInfoApp -lines` will show the wire frame instead of the shaded surfaces.

examples/easyContent/Text2DApp.java

A simple example of using the `Text2D` object to add text to a Java 3D virtual world. The `Text2D` object rotates in the virtual world.

examples/easyContent/Text3DApp.java

A simple example of using the `Text3D` object to add text to a Java 3D virtual world. The `Text3D` object rotates in the virtual world.

0.2.4 Loader**examples/Loader/SimpleQuadLoader/SimpleQuadLoad.java**

An example application that utilizes the `SimpleQuadFileLoader` implemented in the same directory. The loader loads QUAD files of the OOGL file family. See Chapter 2 for more information on the loader and QUAD file format. After compiling the contents of the directory, an example can be run:

```
java SimpleQuadLoad -s ../QuadFile/dodec.quad
```

0.2.5 Interaction

The programs collected in the `examples/Interaction` subdirectory correspond to the topics presented in Chapter 4. Creating and using behaviors to provide user interaction is the subject of the chapter.

examples/Interaction/DoorApp.java

This program demonstrates using the `postId()` method and `WakeupOnBehaviorPost WakeupCriterion` objects to coordinate behavior objects. In this program two behavior classes are defined: `OpenBehavior` and `CloseBehavior`. Then one instance of each behavior class are used to open and close a door. Actually, a `ColorCube` is used as a stand-in for the door.

examples/Interaction/KeyNavigatorApp.java

This program demonstrates using a `KeyNavigatorBehavior` object to provide keyboard based viewer navigation in the virtual world. The user is able to press keys to move forward, back, left, right, up and down as well as rotate left, right, up and down.

examples/Interaction/MouseBehaviorApp.java

This program shows how all three `MouseBehavior` classes (`MouseRotate`, `MouseTranslate`, and `MouseZoom`) can be combined to provide a variety of interactions using the mouse. `MouseRotateApp` is a simpler version of this program as it only uses the `MouseRotate` class.

examples/Interaction/MouseNavigatorApp.java

This program demonstrates how the `MouseBehavior` classes (`MouseRotate`, `MouseTranslate`, and `MouseZoom`) can be used to provide mouse-based viewer navigation in the virtual world. The user is able to move and rotate in response to combinations of mouse button presses and movements.

examples/Interaction/MousePickApp.java

This program demonstrates the picking interaction possible using the `PickRotateBehavior` class. The user is able to pick a visual object and rotate it with mouse movements. This is in contrast to `MouseRotate2App` where the program demonstrates that without picking, the user is constrained as to which objects are available for interaction.

examples/Interaction/MouseRotateApp.java

A demonstration of using the `MouseRotate` class to provide interaction with specific visual objects. The user is able to rotate the programmer-specified visual object with the mouse. `MouseBehaviorApp` is a more complex version of this program providing translation and zoom interactive capabilities in addition to rotation. `MouseRotate2App` demonstrates a limitation of this class.

examples/Interaction/MouseRotate2App.java

A demonstration of using the `MouseRotate` class to provide interaction with specific visual objects. Two cubes rotate in response to user actions. There is no way to interact with just one of the cubes in this program. This program intentionally demonstrates the limitation of interaction with this class. `MouseRotateApp` is a one cube version of this program.

examples/Interaction/PickCallbackApp.java

This is `MousePickApp` modified to include a callback from the picking behavior. The user is able to pick a visual object and rotate it with mouse movements. The simple callback behavior displays a message to the console. This is the answer to question 6 in the self test section.

examples/Interaction/SimpleBehaviorApp.java

A simple behavior class is created and then used in this program. The simple behavior changes a `TransformGroup` object in response to key presses. The effect is to rotate a visual object using key strokes. The program demonstrates behavior usage basics and how to write custom behavior classes.

0.2.6 Animation

examples/Animation/AlphaApp.java

This program illustrates the smoothing possible for the waveform produced by an Alpha object. Three visual objects are translated using three PositionInterpolators and three Alpha objects. Only the IncreasingAlphaRampDuration parameter of the Alpha objects differ among the three car-interpolator-alpha sets. Refer to Section 5.2 and Figure 5-7 for more information.

examples/Animation/BillboardApp.java

This program illustrates the billboard behavior provided by Billboard Class objects. A Billboard object orients a visual object such that it always faces the viewer. The user of this program is free to navigate the virtual world using the arrow keys. Refer to Section 5.3 for more information on applications and API of the Billboard Class.

examples/Animation/ClockApp.java

This program uses one Alpha object and one RotationInterpolator to rotate an analog clock face once per minute. The clock face, defined in Clock.java, is constructed from one Alpha object and two RotationInterplotors. The main program, in ClockApp.java, is a simple example of using a RotationInterpolator. The construction of the clock is somewhat more complex.

examples/Animation/InterpolatorApp.java

This program illustrates six different interpolator classes in one scene to illustrate the variety of interpolator classes available.

examples/Animation/LODApp.java

This program uses a DistanceLOD object to represent a visual object as one of several different geometric representations of varying levels of detail. The DistanceLOD object picks one of the geometric representations based on the distance between the visual object and the viewer.

examples/Animation/MorphApp.java

In this program, a custom behavior classes animates a stick figure walking based on four GeometryArray object key frames. Of course, to truly appreciate the animations, you have to run the program.

examples/Animation/Morph3App.java

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.

examples/Animation/OrientedShape3DApp.java

A forest of 2D trees demonstrates the use of OrientedShape3D objects. Each tree faces the viewer even as the viewer moves and changes view direction. In the application, there is one tree that does not reorient (intentionally not a child of OrientedShape3D) – see if you can find it.

examples/Animation/ParticleApp.java

Demonstrates the use of a GeometryUpdater in creating a particle system to animate a water fountain. The core of the code is explained in some detail within the text of Chapter 6. An image of the fountain appears on the cover of the chapter.

0.2.7 Light

examples/light/LightsNPlanes.java

This program renders a scene where three planes are lit by three different lights. One light is directional, one is a point light, and one is a spot light. See Figure 6-16.

examples/light/LitPlane.java

This program is a basic example of using lights. It renders a scene with a plane and a sphere. See Figure 6-2.

examples/light/LitSphere.java

This program is a basic example of using lights. It renders a scene with a single sphere. See Figure 6-15, among others.

examples/light/LitTwist.java

This program demonstrates the lighting of a two sided object (`setBackFaceNormalsFlip()`). See Figure 6-21.

examples/light/LightScope.java

This program demonstrates the use of scoping to limit the influence of light sources. See Figure 6-25.

examples/light/LocalEyeApp.java

This program illustrates the difference between local eye lighting and infinite eye lighting. See Figure 6-29.

examples/light/ShadowApp.java

This program demonstrates SimpleShadow class. SimpleShadow creates shadow polygons for simple visual objects in certain scenes. See Figure 6-28.

examples/light/ShininessApp.java

This program renders a static scene of nine spheres with different material properties. The only difference among the material properties of the spheres is the shininess value. See Figure 6-20.

examples/light/SpotLightApp.java

This program illustrates the difference various values for the spot light parameters make in rendering. See Figure 6-18.

0.2.8 Texture

examples/texture/BoundaryColorApp

This program loads a single texture image into four Texture2D objects for use with four visual objects. Each of the four textures are configured with a Boundary Color and different Boundary Mode settings. The resulting image illustrates the interaction between the Boundary Mode setting in the presence of a Boundary Color.

examples/texture/BoundaryModeApp

This program loads a single texture image into four Texture2D objects for use with four visual objects. Each of the four textures are configured with a different set of Boundary Mode settings (CLAMP or

WRAP). The resulting image illustrates the possible combinations of Boundary Mode setting for a 2D texture.

examples/texture/MIPmapApp

This program loads a single texture image into a Texture2D object with the MIPmap Mode set to MULTI_LEVEL. The images for each level (other than the base level) are created at runtime from the loaded base image by the TextureLoader utility. Compare this program to MIPmapApp2.

examples/texture/MIPmapApp2

This program loads multiple texture images into a Texture2D object with the MIPmap Mode set to MULTI_LEVEL. Each image is loaded by the TextureLoader utility. Compare this program to MIPmapApp.

examples/texture/MIPmapDemo

This program loads multiple texture images into a Texture2D object with the MIPmap Mode set to MULTI_LEVEL. Each image is loaded by the TextureLoader utility. The texture images used distinguish this application from the typical MIPmap application. The textures are solid color and alternate between red and green on each level. The resulting image shows how textures from a variety of levels can be used for a single visual object.

examples/texture/SimpleTextureApp

This is a very simple example using a texture for a single plane. This application is the result of the straightforward application of the simple texture recipe presented in Section 7.2. See also the TexturedPlaneApp.

examples/texture/SimpleTextureSpinApp

This application takes the SimpleTextureApp one step further and animates the textured plane. It illustrates the single sided nature of textured objects.

examples/texture/Text2DTextureApp

A Text2D object (see chapter 3) creates its image using a Texture2D object. This program applies the texture created by texture Text2D object to another object.

examples/texture/TextureCoordApp

Each of the four planes in this application is textured with the same texture, but each plane is different. This program demonstrates some of the alternate orientations a texture may have when applied to a plane. More texture orientations are demonstrated in TextureRequestApp.

examples/texture/TextureCoordGenApp

In this program a TexCoordGeneration object is used to create the texture coordinates at runtime. This allows the programmer to ignore this detail. It is especially useful for adding textures to visual objects loaded from files. Also, the TexCoordGeneration object is capable of creating varying texture coordinates (in EYE_LINEAR mode) which would hardly be possible otherwise.

examples/texture/TexturedLineApp

This program uses a 1D texture to texture the lines (not the filled polygons) of some geometry. It is an example of a less common application of textures.

examples/texture/TexturedPlaneApp

This application is a demonstration of the TexturedPlane class, which is separately compiled. The difficulty in having a separately compiled class that loads textures lies in using a TextureLoader object outside of an applet. This simple example shows one way to solve the problem.

examples/texture/TexturedPrimitiveApp

This program demonstrates the use of the texture coordinates created by a primitive geometric object (see Chapter 2 for a discussion of geometric primitives).

examples/texture/TexturedSceneApp

This application generates the image on the cover of this chapter.

examples/texture/TextureRequestApp

This program shows some of the possible renderings for a plane using the same texture. The scene is of four planes that differ only in the assignment of texture coordinate values. One plane is solid blue when rendered, the others are striped, but none look like the others nor the texture image. The texture assignments made in this program are examples of possible mistakes while all are legitimate applications. This is the illustration of the phrase "In texturing, you get what you ask for." Other texture orientations are illustrated in TextureCoordApp.

0.3 (Appendix B) Reference Material

0.3.1 Books

Henry Sowizral, Kevin Rushforth, and Michael Deering, *The Java 3D API Specification*, Addison-Wesley, Reading, Mass., December 1997. ISBN 0-201-32576-4

This book describes version 1.0 of the Java 3D API. There are some differences between this specification and the current release of the product. It is comprehensive in coverage, but not intended as a programmer's guide.

It is also available online at <http://java.sun.com/products/java-media/3D>

It is also available in Japanese: translated by Yukio Andoh, Rika Takeuchi; ISBN 4-7561-3017-8

Ken Arnold and James Gosling, *The Java Programming Language*, Addison-Wesley, Reading, Mass.
The Java reference.

David M. Geary, *graphic JAVA Mastering the AWT*, Sunsoft Press, 1997
Complete coverage of the AWT.

Foley, vanDam, Feiner, and Hughes, *Computer Graphics*, Addison-Wesley

This book is widely considered the “bible of computer graphics”. Comprehensive coverage of general computer graphics concepts including representation of points, lines, surfaces, and transformations. Other topics include projection, texturing, z-buffer, and many, many others.

OpenGL ARB, *OpenGL Programming Guide*, Addison-Wesley

While not directly related, this book provides a good foundation in graphics programming via the OpenGL API. Java 3D resembles OpenGL in many ways and some implementations of Java 3D are built on an OpenGL implementation.

0.3.2 The Java 3D API can be downloaded from the Java 3D Home Page:

<http://java.sun.com/products/java-media/3D/>

Follow the "Java 3D Implementation" link to the `download.html` page. Also from this page, you can download documentation for Java 3D API classes.

0.3.3 Sun Java Web Pages

For additional information, refer to these Sun Microsystems pages on the World Wide Web:

<http://java.sun.com/products/java-media/3D>

The Java 3D marketing homepage, this links to many related pages.

<http://java.sun.com/>

The Java Software web site, with the latest information on Java technology, product information, news, and features.

<http://java.sun.com/products/jdk/1.2/>
JDK 1.2 Product and Download Page

<http://java.sun.com/docs>
Java Platform Documentation provides access to white papers, the Java Tutorial and other documents.

<http://developer.java.sun.com/>
The Java Developer Connection web site. (Free registration required.) Additional technical information, news, and features; user forums; support information, and much more.

<http://java.sun.com/products/>
Java Technology Products & API

<http://www.sun.com/solaris/java/>
Java Development Kit for Solaris - Production Release

0.3.4 Other Web Pages

For additional information, refer to the Java 3D web page for links to related resources.

0.4 (Appendix C) Solutions To Selected Self Test Questions

Each chapter concludes with a Self Test section containing questions designed to test and increase the reader's understanding of the material in that chapter. This section presents answers to some of those questions.

Note for questions relating to programming. As in any programming task, there are many answers possible to the programming questions. This section only provides one possible programming solution.

0.4.1 Answers to Questions in Chapter 1

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

Answer:

In general, the final orientation of an object depends on the order of rotations applied. There are cases when the order of rotations will not change the final orientation.

To effect the change in the order of application of the rotations make the following two edits to HelloJava3Db.java. Then compile and run the changed program. Note, if you change the name of the file, you also need to change the name of the high level class in the file. For example, if you change the file to HelloJava3Dbalt.java, then class HelloJava3Db must be changed to HelloJava3Dbalt, along with the name of the constructor and the call to the constructor. Of course, the comments should change to reflect programming changes. Change:

```
rotate.mul(tempRotate);
```

to:

```
tempRotate.mul(rotate);
```

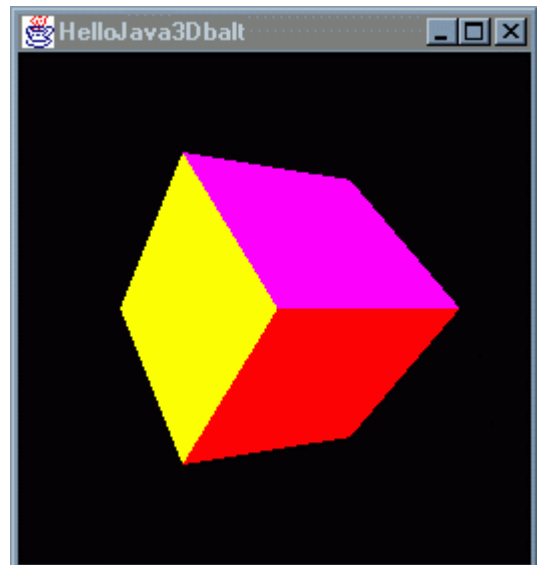
Change:

```
TransformGroup objRotate = new TransformGroup(rotate);
```

to:

```
TransformGroup objRotate = new TransformGroup(tempRotate);
```

After making these changes, compiling, and running the program, the above image at the is produced. If you compare this image to Figure 1-12 in Chapter 1, you can see the difference changing the order of rotations made in this program.



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

Answer:

As in the previous program, the change in order does make a difference. Also as in the first question, only two lines of code need be edited to test the results of the change the question asks about.

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

Answer:

It can be done; however, it is not worth the effort. Unless a change in the scene graph results in fewer Shape3D objects, it does not make sense to make the program harder to write, read, and maintain.

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

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

Answer:

The order of transformations does make a difference.

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

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

Answer:

The BoundingBoxSphere only needs a radius of 0.8 ($0.4 * 2.0$). The center should be (0, 0, 0). The location of the BoundingBoxSphere will be transformed by any TransformGroup objects above it in its scene graph path.

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

Answer:

There are many ways to add a new visible object to a virtual universe. Some possibilities include:

- Add a ColorCube object as a child of the existing BranchGroup.
- Add a ColorCube object as a child of a new BranchGroup and adding this BranchGroup to the Locale of the SimpleUniverse.
- Add a ColorCube object as a child of the existing TransformGroup. Note that since two ColorCube objects will coincide, only one will be visible.
- Add a ColorCube object as a child of a new TransformGroup object and making the new TransformGroup a child of the Locale of the SimpleUniverse.
- Combinations of the above possibilities.

A ColorCube object can not be added as the child of the Locale object.

0.4.2 Answers to Questions in Chapter 2

1. Try your hand at creating a new yo-yo using two cylinders instead of two cones. Using ConeYoyoApp.java as a starting point, what changes are needed?

Answer:

All that is needed is to replace the Cone objects with Cylinder objects.

2. A two-cylinder yo-yo can be created with two quad-strip objects and four triangle-fan objects. Another way is to reuse one quad-strip and one triangle fan. What objects would form this yo-yo visual object? The same approach can be used to create the cone yo-yo. What object would form this yo-yo visual object?

Answer:

In each of the solutions, the visual object can be defined by one Group with two TransformGroup objects, each with a Shape3D child. Each of the Shape3D objects refers to the same Geometry NodeComponent.

3. The default culling mode is used in `YoyoLineApp.java` and `YoyoPointApp.java`. Change either, or both, of these programs to cull nothing, then compile and run the modified program. What difference do you see?

Answer:

With the default, the lines (or points) are culled from back faces. Turning culling off allows all lines (points) to be rendered in all orientations. If you haven't already, try culling front faces.

0.4.3 Answers to Questions in Chapter 3

1. Using the `GeomInfoApp.java` example program as a starting point, try the various settings for ear clipping (see the `Triangulator` reference block in Chapter 3). Using the wire frame view in the program, you can see the effect of the triangulation. For more experience, change the specification of the polygons to use three polygons (one for each side, and one for the roof, hood, trunk lid and other surfaces. How does the `Triangulator` do with this surface?

Answer:

The code for the single polygon description of the hood, roof, trunk, and front and rear glass is embedded in the application code. See the source code for more details. However, there are three other lines of code that must be changed to use this alternative polygon.

Unfortunately, the `Triangulator` does not triangulate this polygon correctly. You might try breaking the one large polygon into smaller ones to see what works. The lesson here is you still have to give some thought to how surfaces are described, even when using `GeometryInfo`.

2. The code to make the `Text2D` object visible from both sides is included in `Text2DApp.java`. You can uncomment the code, recompile and run it. Other experiments with this program include using the texture from the `Text2D` object on other visual objects. For example, try adding a geometric primitive and apply the texture to that object. Of course, you may want to wait until you read Chapter 7 for this exercise.

Answer:

The code to make the `Text2D` object visible from both sides is embedded in the comments of the `Text2DApp.java` example program. See the source code file for more information.

3. Using `Text3DApp.java` as a starting point, experiment with the various alignment and path settings. Other experiments include changing the appearance of the `Text3D` object.

Answer:

Just try some difference settings and observe the results.

4. Playing with the `BackgroundApp.java` example program, if you move far enough away from the origin of the virtual world the background disappears. Why does this happen? If you add another `Background` object to the `BackgroundApp`, what will the effect be?

Answer:

The application of a background is controlled by the `ApplicationBounds` (or `ApplicationBoundingLeaf`) for the background. In the example program, the background has an `ApplicationBounds` sphere centered at the origin with a radius of 10,000.

If another background were added, the effect would depend on the `ApplicationBounds` specified for the new background. In any case, only one background is rendered at a time for any scene. The selection of the background depends on the `ApplicationBounds` and the position of the viewer in the virtual world.

0.4.4 Answers to Questions in Chapter 4

1. Write a custom behavior application that moves visual objects to the left (and right) when a the left (and right) arrow keys are pressed. Then use the class in an application similar to SimpleBehaviorApp.java. Of course, you can use SimpleBehaviorApp.java as a starting point for both the custom behavior class and the application. What happens as the ColorCube object moves out of the view? How do you fix the problem?

Answer:

When the cube moves a sufficient distance, the behavior scheduling bounds no longer coincides with the visual object. Since the bounds object is associated with the behavior, and the behavior is not moving with the visual object, they will eventually be separated to the point where the behavior is only active when the cube is not visible. Conversely, the behavior will be inactive when the cube is visible. So, if you add navigational capabilities to the program, seeing the cube will not necessarily mean you can interact with it.

There is more than one way to change the program so that the position of the cube and the scheduling bounds of the behavior coincide. One way is to make the behavior a child of the transform group object that is moving the visual object. Another way involves using a BoundingLeaf object. See Chapter 3 for more information on the BoundingLeaf class.

2. In SimpleBehaviorApp, the rotation is computed using a angle variable of type double. The angle variable is used to set the rotation of a Transform3D object which sets the transform of the TransformGroup. An alternative would eliminate the angle variable using only a Transform3D object to control the angle increment. There are two variations on this approach, one would read the current transform of the TransformGroup and then multiply, another would store the transform in a local Transform3D object. In either case, the new rotation is found by multiplying the previous Transform3D with the Transform3D that holds the rotation increment. What problem may occur with this alternative? What improvement can be made to this approach?

Answer:

Successive rotations (or transformations of any type) can be implemented using successive multiplication of transforms. The problem lies in the loss of precision through the repeated multiplications. It takes many iterations, but eventually the error will accumulate and result in strange effects in the renderings.

3. Change the trigger condition in the SimpleBehavior class to `new ElapsedFrame(0)`. Compile and run the modified program. Notice the result. Change the code to remove the memory burn problem from the class. Then recompile and run the fixed program.

Answer:

The program will trigger on each frame (and therefore is now an animation application). As a result, a new object is created on each frame. Since objects are being created at a fairly quick rate and not being reused, this is a case of memory burn. The rendering will pause when the garbage collector activates to clean up memory.

4. Change the scheduling bounds for the KeyNavigatorBehavior object to something smaller (e.g., a bounding sphere with a radius of 10), then run the application again. What happens when you move beyond the new bounds? Convert the scheduling bounds for KeyNavigatorApp to a universal application so that you can't get stuck at the edge of the world. See Chapter 3 for more information on BoudingLeaf nodes.

Answer:

When the viewer moves beyond the scheduling bounds, the navigation behavior becomes inactive and

therefore can no longer respond to keystrokes (or any other trigger). The viewer becomes stuck and must exit the program.

5. Use the `KeyNavigatorBehavior` with a `TransformGroup` above a visual object in the content branch graph. What is the effect?

Answer:

The effect is to move the visual object, not the viewer. The action is backward from the normal application of this behavior.

6. Extend the picking behavior in the `MousePickApp` by providing a callback. You can start by simply producing a text string ("picking") to the console. You can also get more ambitious and read the user data from the target transform group or report the translation and/or rotations of the target transform group. With the proper capabilities, you can also access the children of the `TransformGroup` object.

Answer:

See `examples/Interaction/PickingCallbackApp.java`.

0.4.5 Answers to Questions in Chapter 5

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?

Answer:

If each interpolator used a different Alpha object the interpolators would all be synchronized anyway. Each Alpha object is assigned the same start time when it is created. If you wanted the Alpha objects to be out of phase, either change the start time or assign a phase delay duration.

2. If the light in `InterpolatorApp` is changed to `Vector3f(-0.7f, -0.7f, 0.0f)` what happens? Why?

Answer:

The body of the car disappears; however, the wheels are still visible and are changing colors as before. Why do the different parts of the car render differently? If you change the background to a different color, you will see the body of the car is still there but it is solid white. This is the result of complete specular reflection from the body of the car. The combination of the direction of the light and the normals of the car body reflect the light source as a mirror does. This is truly a problem for chapter six, but is a potential head-scratcher for readers of chapter five.

3. Why are there fewer distances than visual objects specified for a `DistanceLOD` object?

Answer:

By design the threshold to begin to use the first child of the target switch object(s) is zero. This threshold is not specified by the user. The threshold distances specified by the user are the minimum distances at which the remaining children are to be used. Therefore, there are one fewer distances specified than there are children.

4. 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?

Answer:

Since `Morph` animates between frames based on vertex ordering, with just two frames the animation would just move back and forth between the two frames. Since the 'feet' of the animation never appear in the same place in one key frame, four frames are necessary. The difference between the frames that

appear the same is the vertex ordering. It would be possible to animate walking with two frames if in one of the frames one foot is behind the other (from the viewer's point of view). Of course, this would only work with 2D models.

5. In using a morph object, the same number of vertices are used. How can you accommodate geometric models of differing numbers of vertices?

Answer:

In the geometry with the smaller number of vertices the vertex count must be increased to match that of the larger geometry. The new vertices can be redundant or internal to a surface.

0.4.6 Answers to Questions in Chapter 6

1. Add a green `DirectionalLight` pointing up to the `LitSphereApp` to illustrate additive color mixing with all three primary colors. Don't forget to add the light to the scene graph and set the influencing bounds for the new light source object. Which two primary colors make yellow?

Answer:

In the positive color system, green and red combine (add) to yield yellow. This is not the result of mixing green and red paint, but green and red light. Mixing red and green paint will likely result in some shade of brown (depending on the characteristics of the paint).

2. To learn more about the interaction between material colors and light colors, create a scene with red, green, and blue visual objects. Light the objects with one single color light. What did you see? You may use `LitSphereApp.java` or `MaterialApp.java` as a starting point.

Answer:

In monochromatic light (pure red, green, or blue) only visual objects with material color with some of the color of the light are visible. By contrast, in monochromatic light, visual objects with material color absent of the color of the light are invisible.

The results you get from your program will depend on your program. If you only set the diffuse color of the visual objects, then the specular highlight will appear in the color of the light (which by default is white).

3. Using `LightScopeApp.java` as a starting point (see Section 6.6.2), change the program to create the shadow of the lit box through the use of scoping only.

Answer:

An additional group node is necessary. It is placed between the `litBoxTG` and the `litBox` object. They the scope references that new group node. You also need to change the material diffuse color to white.

The image rendered from the scene is different since the shadow is lit by the other lights in the scene.

0.4.7 Answers to Questions in Chapter 7

1. What happens if a texture coordinate assignment is not made for a vertex in some geometry? Is there an exception? A warning? Does it render? If it renders, what is the result?

Answer:

In the event a texture coordinate assignment is not made for one or more vertices, the texture coordinate is the default value for texture coordinates (0,0), or (0,0,0) for 3D. There is no exception or warning. The resulting render is the same as if the default value for a texture coordinate had been made for the vertex (vertices).

2. How can a single image of a texture (not repeated) be mapped onto a visual object and have the image surrounded by a single solid color? Assume the texture image is the typical non-solid-color image. How could the surrounding color be assigned or modified at runtime?

Answer:

The CLAMP Boundary Mode is used to apply a single image of a texture to a visual object. One way to have the object appear with a solid color border is to create the texture image with a single texel border with the desired border color. In REPLACE texture mode the border color will be used everywhere beyond the texture application. When a linear filter is used and the Boundary Mode is CLAMP, the Boundary Color (default: black) will be used. In this case you will probably need to set the boundary color to white. Keep in mind that the entire texture image is still subject to the size constraints.

Another way to apply a single image of a texture with a solid color border is to create the texture image with a single texel border that is transparent. In DECAL texture mode the visual object will provide the color beyond the texture image.

Neither of these approaches allows for a changing border color – at least not easily changed. To have a dynamic border color, or just one assigned at runtime, use the same technique as the first solution above: Boundary Mode CLAMP, and linear texture filters. Make a one texel white border color around the texture and apply a boundary color. If the boundary color is to change after the visual object is live, make sure the appropriate capability is set.

3. How can multiple textures be applied to a single visual object? How can you apply different textures to the opposite sides of the same visual object? How can you apply different textures to the lines and the surfaces of a polygon?

Answer:

A visual object can have only one texture. So if you want to use more than one texture on what appears as a single visual object, then it must be created of multiple visual object with the various textures.

4. How would you animate a shadow that moves across a stationary visual object as the shadow moves?

Answer:

A texture can be animated in certain limited ways (stretching, rotating, moving) by changing the texture transform for the visual object. With the appropriate capability set such that the texture transform can be changed at runtime, a texture that represents the shadow can be moved, made larger or smaller, or rotated on a visual object.

5. How would you animate a shadow that moves across a visual object as the object passes through a stationary shadow?

Answer:

The solution to question four can be used, however this would require coordination of the object's movement with the movement of the texture. There is (at least) one other way. A TexCoordGeneration object with EYE_LINEAR generation mode assigns texture coordinates that are stationary in the virtual space. Using a TexCoordGeneration object configured in this way would make the texture stationary even as the object moved through the virtual space.

0.5 (Appendix D) Geometry and Math

0.5.1 Defining a Plane

Planes are infinite. As such, it is not possible to specify the vertices for a plane (as is done with other geometry); instead, planes are typically defined by specifying four values: A, B, C, and D. The surface of the plane is defined by the equation:

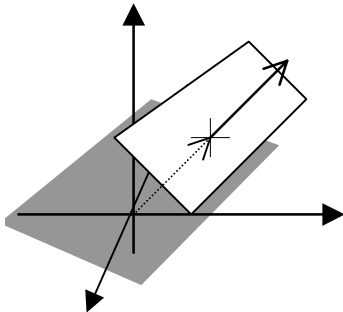
$$Ax + By + Cz + D = 0$$

The plane is the set of all points (x, y, z) that satisfy the above equation. This is an infinite set of points.

An easier way to think of the plane is to think of the first three values (A, B, C) as the specification of the surface normal for the plane. A vector only specifies the orientation of the plane, the fourth value, D, fixes the plane in 3-space. If A, B, and C, specify a unit (normalized) vector, then one point of the plane is D distance from the origin in the opposite direction of the surface normal vector.

The plane specified as 1, 0, 0, 0 is a plane with surface normal vector in the x direction and the point (0, 0, 0) is in the plane. $Ax + By + Cz + D = 1x + 0y + 0z + 0 = 0 \rightarrow x = 0$. In words, any point where the x component is 0 is a point in the plane.

The plane specified as 1, 0, 0, -10 is a plane with surface normal vector in the x direction and the point (10, 0, 0) is in the plane. $Ax + By + Cz + D = 1x + 0y + 0z - 10 = 0 \rightarrow x = 10$. In words, any point where the x component is 10 is a point in the plane.



The plane of the above figure is defined by the 4-tuple (.707, .707, 0, -1). The first three values is the vector halfway between the X and Y axes. The plane is orthogonal to that vector and a distance of 1 away from the origin along that vector. The shadow is meant to give some perspective to the illustration.

Specification of half-space

Half spaces are infinite; they are all the points that lie on one side of a plane. Half spaces are typically defined by specifying four values: A, B, C, and D. The half space is defined by the equation:

$$Ax + By + Cz + D < 0$$

Notice that this equation is the same as the equation for a plane except for the relational operator. The half space is the set of all points (x, y, z) that satisfy the above equation. This is an infinite set of points. In the above figure, the half space is the back side of the plane (the shadow side).

Half spaces are used in defining BoundingPolytope and ModelClip objects. Both of these classes are described in chapter 2 of this tutorial.

0.6 Glossary

A

activation volume	A volume associated with a ViewPlatform Node. When the activation volume intersects application regions or scheduling regions, the objects associated with the intersecting regions affect rendering. The affected objects include backgrounds, behaviors, clip, fog, and light. See also <i>ViewPlatform</i> , <i>application bounds</i> , and <i>scheduling bounds</i> .
active	Said of a behavior object that is able to receive a wakeup stimulus and therefore execute. A behavior object is active when it's scheduling bounds intersects a ViewPlatform's activation volume. When a behavior is not active, Java 3D will ignore the wakeup criteria for that behavior. See also <i>activation volume</i> and <i>behavior</i> .
aliasing	The appearance of jaggies in lines and curves. The result of sampling the a continuous function. See also <i>jaggies</i> and <i>antialiasing</i> .
alpha	In computer graphics, the term 'alpha' normally refers to transparency. In Java 3D alpha also refers to transparency, but may also refer to the Alpha class, or an alpha value. This may be a source of confusion for experienced graphics programmers. See also <i>Alpha class</i> , <i>alpha value</i> , <i>RGBA</i> , and <i>transparency</i> .
Alpha class	An Alpha class object creates a time-varying function of alpha values. Normally, and Alpha object is used with Interpolator objects to create time-based animations. See also <i>alpha value</i> , <i>animation</i> and <i>interaction</i> .
alpha value	A value in the range 0.0 to 1.0, inclusive. Alpha values are used in various capacities in Java 3D (e.g., transparency value or Interpolator behavior objects). Time varying alpha values are produced by Alpha objects. In the specification of color, an alpha value expresses the opacity of the color. If alpha is 0 then it is not opaque (it is fully transparent); if alpha is 1, the color is fully opaque. See also <i>Alpha Class</i> , <i>behavior</i> , and <i>interpolator</i> .
ambient (material) color	Part of the material properties of a visual object. The ambient color of a material and the ambient light in scene produce an ambient reflection.
ambient light	A light source present in all places shining in all directions used to model the complex inter-object reflections (repeated reflection of light from one object to another) present in the real world. See also <i>ambient color</i> .
ambient reflection	The light produced by an ambient light source reflected from a visual object with material appearance attributes. The result depends on the ambient color of the object and the color of the ambient light source. See <i>ambient color</i> , <i>ambient light</i> , and <i>lighting model</i> .
ancestors (of an object)	All of the objects in a scene graph that are children of a specific object and all of the childrens' ancestors. See also <i>object</i> , <i>scene graph</i> , and <i>parent-child</i> relationship.
animation	The automatic change of visual content (e.g., changes based on time alone). In this tutorial, animations are driven by the passage of time,

	changes of the view platform location, and possibly other non-user action influences. Animation is the subject of Chapter 5. See also <i>interaction</i> .
antialiasing	A process of smoothing the drawing of points or lines that would otherwise appear jagged. See also <i>jaggies</i> , <i>line antialiasing</i> and <i>point antialiasing</i> .
API	see <i>Application Programming Interface</i>
Appearance class	Instances of Appearance define the appearance of a Shape3D object. Appearance objects include AppearanceAttribute objects. See also <i>AppearanceAttributes</i> and <i>Shape3D</i> .
application bounds	The region for which something is applied. Backgrounds, Behaviors, and Clip Nodes have application bounds. For example, when the view's activation volume intersects the application bounds of a particular background, that background is used in rendering. See also <i>bounds</i> .
Application Programming Interface	(API) General term referring to a collection of classes (or procedures) that form the programming interface to some computer system. The Java 3D API classes form the programmers interface to the Java 3D renderer.
Attributes classes	Instances of Attributes define specific appearance attributes of a Shape3D object. There are several Attributes classes. Appearance objects include Attribute objects. See also <i>Appearance class</i> and <i>Shape3D</i> .
B	
base level	Level 0 of a texture. Base level is the only level of a texture unless the Mipmap Mode is set to MULTI_LEVEL. Base level also refers to a single level texture as opposed to a multiple level texture. In a multiple level texture base level, level 0, is the largest texture image. See also <i>texture mapping</i> and <i>multiple levels of texture</i> .
billboard	Automatic orientation of a polygon in an orthogonal orientation to the viewer such that only the front face of the polygon is viewed. Typically the polygon is textured with an image.
Behavior class	A javax.media.j3d class used to specify how some characteristic, such as location, orientation, or size, changes at runtime. Behavior classes are used in creating animations as well as interactive programs. See also <i>behavior</i> , <i>Interpolator</i> , and <i>Alpha class</i> .
behavior	The changing of some characteristic, such as location, orientation, size, color, or a combination of these, of a visual object at run time. A behavior is either an animation or an interaction. See also <i>Behavior class</i> .
bounding volume	A volume that defines the limits for which something is defined. Bounds are used with behaviors, lights, sounds, and other features of Java 3D. Bounds are defined in a program with the Bounds class. When the See also <i>Bounds class</i> and <i>scheduling region</i> .
Bounds class	An abstract class to specify a bounding volume for which something is defined. Bounds are used with behaviors, backgrounds, sounds, and other features of Java 3D. Bounds can be specified with a BoundingBox,

	BoundingSphere or a BoundingPolytope. See also <i>bounding volume</i> , <i>polytope</i> , and <i>scheduling region</i> .
bounds	See <i>bounding volume</i> and <i>Bounds class</i> .
BoundingBox class	Instances of BoundingBox define bounds as a axis-aligned box. See also <i>bounding volume</i> .
BoundingSphere class	Instances of BoundingSphere define bounds as a sphere. See also <i>bounding volume</i> .
branch graph	That portion of the scene graph rooted in a BranchGroup object. See also: <i>BranchGroup</i> , <i>scene graph</i> , <i>content branch graph</i> , and <i>view branch graph</i> .
BranchGroup class	Instances of BranchGroup are the root of individual scene graph branches, called branch graphs. A BranchGroup may have many children that are Group and/or Leaf objects. A BranchGroup is the only object that may be inserted into a Locale object. See also <i>scene graph</i> , <i>branch graph</i> , and <i>Locale</i> .
C	
capabilities	In the Java 3D sense, access to the parameters of a live object are controlled with capabilities. For example, a transform of a live TransformGroup can not be changed unless the capability to do so was set for that instance of TransformGroup before it was made live. See also <i>live</i> and <i>TransformGroup</i> .
CLAMP	One of the Boundary Modes available in texture mapping. Clamps texture coordinates to be in the range [0, 1]. See also <i>texture mapping</i> .
class	The specification of a set of values and a set of operations. A class is analogous to a type in a procedural language such as C. See also <i>object</i> .
Color* classes	A set of classes used to represent a single color value. The classes are Color3b, Color3f, Color4b, and Color4f.
Color3* classes	A set of classes used to represent a single RGB color value. The classes are Color3b and Color3f. See also <i>Color*</i> and <i>RGB</i> .
Color4* classes	A set of classes used to represent a single RGBA color value. The classes are Color4b and Color4f. See also <i>Color*</i> and <i>RGBA</i> .
compile	In the Java 3D sense, <code>compile()</code> is a method of BranchGroup to allow Java 3D to make run-time performance improvements in the branch graph rooted on the BranchGroup. See also <i>BranchGroup</i> and <i>branch graph</i> .
concentration	A parameter of spot light objects that determines the spread of light from the light source. See also <i>spot light</i> .
content	The visual (including shape and lights) and audio objects of a virtual universe. See also <i>visual object</i> .
content branch graph	The portion of the scene graph that contains the visual objects. See also <i>content</i> and <i>visual object</i> .
convenience class	A class that extends a core class for the purpose of making a core class (or classes) easier to use. Convenience classes are found in the <code>com.sun.j3d.*</code> packages.

crease angle	The threshold angle between the surfaces of adjacent triangles in a surface for which it is taken to be a discontinuous crease. When a crease is detected by the NormalGenerator it produces multiple normals for vertices.
cull	To remove something not valuable or not needed. See also <i>cull face</i> .
cull face	The face that is not to be rendered. A cull face is specified to avoid rendering a face that is not needed, for example, the interior face of an enclosed surface. Cull faces are specified as either front face or back face. A cull face is specified to improve rendering performance. See also <i>front face</i> .
D	
DAG	See <i>Directed Acyclic Graph</i>
deactivation	When a behavior's scheduling region and the ViewPlatform's activation volume no longer intersect, the behavior is deactivated. Since a behavior can only be triggered when active, a deactivated behavior object will not do anything. See also <i>active</i> and <i>behavior</i> .
diffuse reflection	The most common reflection of light from visual objects present in the real world. The diffuse color is the color of an object in typical lighting conditions. See also <i>lighting model</i> .
directed acyclic graph	A data structure composed of elements and directed arcs in which no cycles are formed. In the Java 3D world, the elements of the DAG are Group and Leaf objects, and the arcs are the parent/child relationships between Groups and Leaf objects. In forming the structure without cycles means no element can be its own parent.
E	
emissive (material) color	A color specified to make an object self illuminating. The object is not a light source and does not lit other object, but is visible in the absence of light sources. See also <i>material properties</i> .
examples.jar	The archive of example programs published with this tutorial. See also <i>jar</i> .
exception	An expected runtime problem that halts execution when detected. The Java 3D API defines several exceptions.
execution culling	Since computational resources are shared for rendering and behavior execution; some (or all) computational resources are not available for rendering while behaviors are executed. This could degrade rendering performance when executing behavior objects. Therefore, Java 3D ignores behaviors when they are not active to conserve execution resources. This is termed execution culling since execution of deactivated behavior objects is culled. See also <i>active</i> and <i>behavior</i> .
eye vector	The vector between a vertex (to be shaded) and the viewing position. This vector is used in calculating the specular reflection of an object. See also <i>specular reflection</i> , <i>local eye</i> , and <i>infinite eye</i> .

F

flat shading	Shading an object with the colors of each vector without interpolation. See also <i>shading model</i> .
font	The collection of glyphs for the alphabet. A typeface at a specific point size and attributes (i.e., italics or bold). See also <i>glyph</i> and <i>typeface</i> .
front face	The face of a polygon for which the vertices are in counter-clockwise order. An easy way to remember is to apply the right-hand rule. See also <i>right-hand rule</i> and <i>cull face</i> .
frustum	see <i>view frustum</i>

G

get* or get-method	A method of a class that retrieves a field or value from an object. See also <i>set*</i>
Geometry classes	The abstract class that is the super class for all geometric primitive classes such as <i>GeometryArray</i> , and <i>Text3D</i> .
glyph	The image of a character in a font. See also <i>font</i> and <i>typeface</i> .
Gouraud shading	Smooth shading of an object through trilinear interpolation of color values at the object's vertices. See also <i>flat shading</i> , <i>trilinear interpolation</i> .
Group class	Group is an abstract class. Instances of subclasses Group are used in creating scene graphs. A Group is a scene graph object whose primary function is to be the parent of other scene graph objects (Leaf objects and other Group objects. See also <i>BranchGroup</i> and <i>TransformGroup</i> .

H

half-space	The space on one side of a plane. A plane divides all of space in two halves, one half on each side of the plane. See also <i>plane</i> .
------------	---

I

image plate	The imaginary rectangle in the virtual universe to which the scene is projected. See Figure 1-9 for an illustration. See also <i>view frustum</i> .
image observer	An object that implements the image observer interface which allows it to monitor the loading of images from a file or URL.
infinite eye (lighting)	Rendering an scene as though it were viewed from a position at infinity. The actual effect is to have a constant eye vector (0, 0, 1). This reduces rendering time, but may look 'funny' due to the placement of specular reflections. Infinite eye lighting is the default. See also <i>lighting model</i> , <i>specular reflection</i> , <i>eye vector</i> , and <i>local eye (lighting)</i> .
influence (of a light)	The region (volume) for which the bounding volume of a visual object must intersect for the visual object to be lit by the light source. See also <i>bounds</i> .
influencing bounds (of a light)	See <i>influence (of a light)</i> .
instance (of a class)	An instance of a class is a specific, individual object constructed of the named class.

intensity	A single value that represents the perceived brightness of a light source. Also used as a setting for a texture image that has a single value for each texel which is taken as the value for R, G, B, and A.
interaction	Changing the state of a virtual world in response to user action. This is in contrast to animation which is defined as a change in the state of the virtual world not directly caused by user action. Interaction is the subject of Chapter 4. See also <i>animation</i> .
interface	Like an abstract class, an interface defines methods to be implemented by other classes. No constructor is defined in an interface and all of the methods defined in an interface are abstract.
interpolator	Refers to one of several classes subclassed from the Interpolator class, or an object of one of these classes. Interpolator objects provide animation in the Java 3D virtual world by varying some parameter(s) of a target scene graph object. The changes made by the interpolator are determined by the parameters of the interpolator and the Alpha object driving the interpolator. See also <i>Alpha Class</i> , <i>behavior</i> , and <i>target object</i> .
interpolation	The computation of a value, or set of values, based on the value of a single integer. Sometimes the derived value is interpolated between two values, or two sets of values; other times, the derived value is the result of a formula. Specifically, there are a set of <i>interpolator</i> classes useful for <i>animation</i> . See Section 4.1 and Chapter 5 for more details.

J

Java 2D	An API for 2D graphics.
jaggies	The technical term for the roughness that may appear on points, lines, or polygons when rendered. The roughness appears when the individual pixels used stand out. See also <i>antialiasing</i> .
jar	<ol style="list-style-type: none">1. An archive file format useful for distributing a collection of files.2. The utility that creates and reads such archive files (Java ARchive).

K

K Computing	The training and consulting company that developed this tutorial document. See also http://www.kcomputing.com
key frame	A term used in traditional and computer animation for a frame of the animation on which others are based. The process of creating frames in between the key frames is called "in-betweening". Animations made from key frames and in-betweening are called key frame animations. A Morph object can be used to do key frame animations in Java 3D.

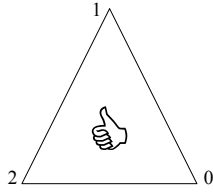
L

level of detail (LOD)	Level of detail (LOD) refers to an application specific behavior that changes the representation of a visual object based on its distance (normally) to the viewer. When the LOD object is close to the viewer, more detailed representations are used. Less detailed representations are used with more distance. The goal is to reduce the rendering computation
-----------------------	--

	without degrading rendered imagery. See Section 4.1 and Chapter 5 for more details.
light vector	The vector between a light source and the vertex being shaded. See also <i>lighting model</i> .
lighting model	The calculation of the color (shade) of a specific vertex of a visual object as the result of influencing light sources and the material properties of the visual object. The shade is the result of ambient, diffuse, and specular reflections as well as the emissive color of the material. See also <i>ambient reflection</i> , <i>diffuse reflection</i> , <i>specular reflection</i> , and <i>material properties</i> .
line antialiasing	An appearance attribute that, when enabled, causes the renderer to apply antialiasing to lines as they are rendered. See also <i>antialiasing</i> and <i>render</i> .
live	The term ‘live’ refers to the state of a SceneGraphObject being associated with a rendering engine. In other words, a live object is a member of a branch graph associated with a Locale which is a member of a VirtualUniverse that has a View attached (through a Locale object) and, therefore, has the potential to be rendered. Live objects have restricted capabilities. See also <i>render</i> , <i>SceneGraphObject</i> , <i>branch graph</i> , <i>Locale class</i> , <i>VirtualUniverse class</i> , <i>scene graph</i> , and <i>capabilities</i> .
loader	A Java 3D utility class that creates scene graph elements from a file. Loaders exist for many common 3D file formats.
local eye (lighting)	Rendering an scene as though it were viewed from the local eye position - as opposed to the default infinite eye. This increases rendering time. Infinite eye lighting is the default. See also <i>lighting model</i> , <i>specular reflection</i> , <i>eye vector</i> , and <i>infinite eye (lighting)</i> .
Locale class	Instances of Locale provide landmarks in the virtual universe. The position of all visual objects is relative to some Locale object. Locale objects are the only object VirtualUniverse objects reference. BranchGroup objects are the only children of a Locale object. See also <i>BranchGroup</i> , <i>VirtualUniverse class</i> , <i>visual object</i> , and <i>branch graph</i> .
luminance	A single value that represents the perceived brightness for a surface. Also used as a setting for a texture image that has a two values for each texel where one value is taken as the value for R, G and B and the other value is used for alpha.
M	
magnification filter	A filter used in texture mapping when the pixel size is larger than the texel size. See also <i>texture mapping</i> and <i>texel</i> .
material properties	The specification of the ambient, diffuse, specular, and emissive colors, and concentration of a material used in calculating the shade of an object. The first three colors are used in calculating the appropriate reflection. See also <i>ambient reflection</i> , <i>diffuse reflection</i> , <i>specular reflection</i> , <i>lighting model</i> , and <i>shade</i> .
memory burn	The rate of memory allocation and garbage collection as an application runs. This is typically due to unnecessarily creating and destroying

	objects. Memory burn can adversely affect rendering performance in some runtime environments. Avoid memory burn in behaviors.
minification filter	A filter used in texture mapping when the pixel size is smaller than the texel size. See also <i>texture mapping</i> and <i>texel</i> .
MIP Map	MIP (<i>multum in parvo</i> – Latin for many things in a small place) Map 1 . Refers to a specific storage technique for storing a series of images for multi level texturing, where each successive image is one quarter the size of the next ($\frac{1}{2}$ the size in each dimension) until the size of the smallest image is 1 texel by 1 texel [See Williams, <i>SIGGRAPH</i> 1983]. 2 . The common use of the term means "multi level texturing". See Chapter 7. See also <i>texture mapping</i> , <i>multiple levels of texture</i> , and <i>texel</i> .
multi level texturing	See <i>multiple levels of texture</i> .
multiple levels of texture	Having a series of texture images at various resolutions available so that the rendering system can select the texture map of a size that is the closest match to the visual object being rendered. See Chapter 7. See also <i>texture mapping</i> and <i>MIPmap</i> .
multitexture	Applying multiple texture images to one visual object. Refer to Chapter 7. See also <i>texture mapping</i> and <i>texture unit</i> .
N	
normal	A vector that defines the surface orientation. In Java 3D normals are associated with coordinate points in geometry. See also <i>Geometry</i> .
normal vector	See <i>normal</i> .
O	
object	An instantiation of a class. See also <i>class</i> and <i>visual object</i> .
object of change	The scene graph object that is changed by a behavior and through which the behavior affects the virtual world. For example, a TransformGroup object is often the object of change for interactive behaviors. See Section 4.2 for more information.
P	
pick	To select an object with the mouse. See also <i>picking</i> .
picking	To select a visual object for interaction with the mouse. Picking is implemented in Java 3D through behaviors. See Chapter 4 for more information on Behaviors, Picking, and example programs utilizing picking classes. See also <i>behavior</i> .
pick ray	A pick ray is a ray whose end point is the mouse location and direction is parallel to the projection (parallel with projectors). In many picking applications, the pick ray is used in picking an object for interaction. Also, PickRay is a subclass of PickShape. See the appropriate reference block or API reference for the class. See also <i>picking</i> .
pixel	An individual picture element of the display. Each pixel is addressable by an [x, y] coordinate and assigned a single color. In Java 3D, programs do

	not typically address individual pixels, instead 3D elements are rasterized. See also <i>rasterize</i> .
plane	A flat surface extending infinitely in all directions. The orientation of a plane is normally expressed as a surface normal. A plane can be uniquely defined by a single point and a normal vector.
plane equation	A plane is uniquely specified by a 4-tuple. The first three values represent the surface normal vector for the plane. The fourth value specifies the distance from the origin to the plane along a vector parallel to the plane's surface normal vector.
Point* classes	Point* refers to one, or all, of a number of classes used to represent points in Java 3D. Consult a reference for Point2f, Point3f, Point3d, ... classes.
point antialiasing	An appearance attribute that, when enabled, causes the renderer to apply antialiasing to points as they are rendered, causing the points to look less jagged. See also <i>antialiasing</i> , <i>render</i> , and <i>jaggies</i> .
polytope	A bounding volume defined by a closed intersection of half-spaces.
processStimulus	A method of Behavior. The processStimulus method is called by Java 3D when the appropriate trigger condition has occurred. It is through this method that the behavior object responds to the trigger. See Chapter 4 for more information. See also <i>behavior</i> and <i>wakeup condition</i> .
project	To express the world coordinate geometry of 3D objects in 2D image plate space.
projector	The lines that correlate the vertices of a 3D object in world coordinate space with the pixels in image plate space. A straight line drawn between a 3D vertex and the viewpoint (viewer's eye) is a projector and determines the pixel(s) the vertex will rasterize to.
Q	
quad	Short for quadrilateral. A four sided polygon.
quaternion	A quaternion is defined using four floating point values $ x\ y\ z\ w $. A quaternion specifies rotation in four dimensions.
R	
raster	The per-pixel memory of the display hardware.
rasterize	To convert visual objects and their components to their projected images. The term comes from the use of a raster display device on virtually all common computers.
ray tracing	Applications which render scenes by modeling individual rays of light. These applications can model inter-object effects such as shadows but are not fast enough for real time rendering.
render	To produce the image represented by the scene graph.
renderer	Software to produce the image from a scene graph.
RGB	Red, Green, and Blue, the three components used to represent color.

RGBA	Red, Green, Blue, and Alpha, the three components used to represent color with a transparency value. If alpha is 0 then it is not opaque (it is fully transparent); if alpha is 1, the color is fully opaque.
right-hand rule	<p>"Right-hand rule" refers to the correlation between the direction the fingers curl and the direction the thumb points on your right hand. The right-hand rule applies in determining the front face of a polygon, when computing a cross product of vectors, and when figuring out which way to turn right-handed nuts, bolts, and screws. The figure at the right shows the fingers of the right hand curling in the order in which the vertex coordinates were defined for a triangle. The thumb, pointing up (out of the page), indicates we are seeing the front face of the triangle. See also <i>front face</i> and <i>culling</i>.</p> 
S	
scale	To change the shape of a visual object by transforming each of the vertices of the object. The shape of the visual object can be preserved or distorted depending on the scale transform. See also <i>transform</i> .
scanline	A single row of pixels of the output device.
scanline order	The ordering of pixels of a window taken left to right and top to bottom – like the order of characters are read (in English). This order is normally used in rendering pixels.
scene graph	The Java 3D data structure that specifies the visual objects and viewing parameters in a virtual universe.
scene graph path	The path from a locale object, or an interior node, to a leaf of the scene graph. <code>SceneGraphPath</code> is a class used in picking. See Chapter 4 for more information on the class.
scheduling bounds	A region defined for a behavior. The activation volume of a view must intersect the scheduling bounds of a behavior for the behavior to be active. A behavior must be active to be able to execute in response to a stimulus. See also <i>active</i> , <i>activation volume</i> , and <i>behavior</i> .
scheduling bounding leaf	An alternative to a scheduling bounds. See also <i>BoundingLeaf</i> and <i>scheduling bounds</i> .
scheduling region	See <i>scheduling bounds</i> .
scope	The portion of a scene graph for which an object's (e.g., light node, fog node, model clip node) influence is considered. By default, the scope of a node is the universe. Sub-graphs of the scene graph can be specified as the scope of a light. This does not replace the specification of a light's region of influence, but is complementary to the influence of a light. See also <i>influence (of a light)</i> and <i>universe scope</i> .
sensor	The abstract concept of an input device such as joy-stick, tracker, or data glove.

set* or set-method	A method of a class that sets a field or value in an object. See also <i>get*</i>
shade	n. The color of a vertex or pixel as a result of the application of the lighting model (for a vertex) or the shade model (for pixel). v. To calculate the color of a vertex or pixel by the application of the lighting model (for a vertex) or the shade model (for pixel).
shade model	The calculation of each pixel's shade value from the shade of each neighboring vertex shade. See also <i>shade</i> , <i>Gouraud shading</i> and <i>flat shading</i> .
shadow polygon	A polygon used to create a shadow in a scene. See section 6.7.3
shininess	The specification of how shiny a material surface is. This value (in the range 1.0 to 128.0) is used in calculating the specular reflection of a light from a visual object. The higher the value the more concentrated the specular reflection is. See also <i>specular reflection</i> and <i>material properties</i> .
specular reflection	The highlight reflection of light from a shiny visual object. In the real world presence of a specular reflection depends heavily on how smooth the surface is. In Java 3D this is modeled as a specular material color and a concentration value. See also <i>material properties</i> , <i>specular color</i> , and <i>concentration</i> .
SpotLight class	A light source class that has a position, direction, spread angle and concentration values. See Chapter 6 for more information. See also <i>concentration</i> .
stitching	When the same geometry is rendered as a wire frame and as filled polygons (of different color), by default the depth of the pixels rendered for each will not correspond and so the wire frame will appear to move in and out of the surface and appear as a thread though a cloth surface. See Section 2.6.3. PolygonAttributes in Module 1 for additional information.
stripification	Organization of triangles into triangle strips for rendering efficiency.
surface normal	See <i>normal</i> .
T	
texel	A TEXTure ELEMENT. A pixel of a texture image. See <i>texture mapping</i> .
texture	1. n. The image used in texture mapping a visual object. 2. v. To apply an image to a visual object through texture mapping. See also <i>texture mapping</i> .
texture mapping	The application of a texture image to a visual object based on the assignment of texture coordinate values to geometric vertices and texture mapping filters. See also <i>texture</i> , <i>minification filter</i> , and <i>magnification filter</i> .
texture unit	That portion of the low-level hardware/software rendering system responsible for texture mapping a single texture on a visual object. Refer to Chapter 7. See also <i>texture mapping</i> and <i>multitexture</i> .
three space	Three dimensional space.

transform	The mathematical operation performed on a vertex, or collection of vertices, to translate, rotate, or scale the vertices. Transformations are represented as 4 x 4 matrices and stored in TransformGroup objects.
translate	Move a vertex or vertices.
TransformGroup class	A subclass of Group that also applies a transformation to its child nodes. See also <i>transformation</i> .
target object	The scene graph object changed by a behavior or interpolator.
triangulation	The subdivision of a polygon into triangles.
trilinear interpolation	The use of three linear interpolations to arrive at a value. This technique is used to calculate a shade value for a pixel from the shade values of vertices in Gouraud shading. See also <i>Gouraud shading</i> and <i>flat shading</i> .
Tuple* classes	A set of classes defined in the <code>javax.vecmath</code> package used to represent tuples. The seven individual Tuple classes are Tuple2f, Tuple3b, Tuple3d, Tuple3f, Tuple4b, Tuple4f, and Tuple4d. These classes are the superclasses of Color*, Point* and Vector* classes (among others).
typeface	The style of printing text. For example Times Roman, Helvetica, and Courier are all typefaces. By contrast, a font is a typeface with other specific attributes. For example, "10 point, italic, Times Roman" is a font. See also <i>font</i> .
U	
universe scope	The default scope for those scene graph object that can have a scope specification, which is that the scope is all scene graph objects in the virtual universe.
utility class	A class in the <code>com.sun.j3d.utils</code> package, that builds upon the core classes to enhance programming capabilities.
V	
vecmath	An extension package that defines classes for computing vector math. Among these are the Tuple* classes.
View class	The View object is the central object for coordinating all aspects of a view including which canvas(es). Java 3D supports multiple simultaneous views.
view branch graph	The portion of the scene graph containing a View object. The parameters of the viewing environment (e.g., stereo) and physical parameters (e.g., physical position of the viewer) are specified in the view branch graph.
view frustum	A truncated pyramid-shaped viewing volume that defines how much of the virtual universe the viewer sees. Visual objects not within the view frustum are not visible. Objects that intersect the boundaries of the viewing frustum are clipped. See Figure 1-9 for an illustration. See also <i>clip</i> , <i>viewer</i> , and <i>visual object</i> .
viewer	The (primary) person viewing the display device Java 3D is rendering to. It is for this person the PhysicalBody calibration parameters are set.

virtual universe	The conceptual space in which the visual objects 'exist'. A virtual universe may contain multiple virtual worlds as defined by Locale objects.
VirtualUniverse class	The core class for Java 3D. An instance of VirtualUniverse must be the root of the scene graph.
virtual world	The conceptual space in which the visual objects 'exist' as defined by a single Locale object. A virtual world is contained by a virtual universe.
visual object	The term “visual object” is used in places where ‘object’ would make sense in English but not in the Object Oriented sense. A visual object may or may not be visible in a view depending on many factors. “Visual object” most often refers to a Shape3D object. (see section 1.2) See also <i>content</i> .

W

wakeup condition	The combination of wakeup criterion that specifies the trigger condition for a behavior object. Java 3D calls the processStimulus method of the behavior object in response to the occurrence of the wakeup condition for an active behavior object. WakeupCondition is a class presented in Chapter 4. See also <i>behavior</i> , <i>processStimulus</i> , and <i>wakeup criterion</i> .
wakeup criterion	Combinations of wakeup criterion objects form a wakeup condition for a behavior object. Some possible wakeup criterion include AWTEvents, collisions, behavior activation, passage of time, and a specified number of frames have been drawn. WakeupCriterion is a class presented in Chapter 4. See also <i>wakeup condition</i> .
WRAP	One of the Boundary Modes available in texture mapping. Repeats the texture by wrapping texture coordinates that are outside the range [0,1]. See <i>texture mapping</i> .

X**Y**

yo-yo	A toy.
-------	--------

Z

z-buffer	A data structure internal to the renderer to determine the relative depth (distance from image plate) of visual objects on a per pixel basis. Only the visual object closest to the image plate is visible.
----------	---