Java 3D™ API Specification

Version 1.1 Alpha 01, February 27, 1998

JavaSoft
A Sun Microsystems, Inc. Business
901 San Antonio Road
Palo Alto, CA 94303 USA
415 960-1300 fax 415 969-9131

Contents

Figures

Preface

THIS document describes the Java 3D™ API and presents some details on the implementation of the API. This specification is not intended as a programmer's guide. The programmer's guide will be written after the specification has been finalized.

This specification is written for 3D graphics application programmers. We assume that the reader has at least a rudimentary understanding of computer graphics. This includes familiarity with the essentials of computer graphics algorithms as well as familiarity with basic graphics hardware and associated terminology.

Related Documentation

This specification is intended to be used in conjunction with the Java 3D reference guide, an online, browser-accessible, javadoc-generated API reference.

Style Conventions

The following style conventions are used in this specification:

- `Lucida type` is used to represent computer code and the names of files and directories.
- **`Bold Lucida type`** is used for Java 3D API declarations.
- **Bold** type is used to represent variables.
- *Italic type* is used for emphasis and for equations.

Programming Conventions

Java 3D uses the following programming conventions:

- The default coordinate system is right-handed, with $+Y$ being up, $+X$ horizontal to the right, and $+Z$ directed toward the viewer.
- All angles or rotational representations are in radians.
- All distances are expressed in units or fractions of meters.

Acknowledgments

We gratefully acknowledge Warren Dale for writing the Sound API portion of this specification, Daniel Petersen for writing the scene graph sharing portion of the specification, and Bruce Bartlett for his assistance with the editing, formatting, and indexing of the specification.

We thank the Java 3D partners for their help in defining the Java 3D API. The Java 3D partner companies include Silicon Graphics, Inc., Intel Corporation, Apple Computer, Inc., and Sun Microsystems, Inc.

We also thank the many individuals and companies for their comments and suggestions on the successive drafts of this specification.

Henry Sowizral
Kevin Rushforth
Michael Deering
Sun Microsystems
November 1997

CHAPTER 1

Introduction to Java 3D

THE Java 3D API is an application programming interface used for writing three-dimensional graphics applications and applets. It gives developers high-level constructs for creating and manipulating 3D geometry and for constructing the structures used in rendering that geometry. Application developers can describe very large virtual worlds using these constructs, which provide Java 3D with enough information to render these worlds efficiently.

Java 3D delivers Java's "write once, run anywhere" benefit to developers of 3D graphics applications. Java 3D is part of the JavaMedia suite of APIs, making it available on a wide range of platforms. It also integrates well with the Internet because applications and applets written using the Java 3D API have access to the entire set of Java classes.

The Java 3D API draws its ideas from existing graphics APIs and from new technologies. Java 3D's low-level graphics constructs synthesize the best ideas found in low-level APIs such as Direct3D, OpenGL, QuickDraw3D, and XGL. Similarly, its higher-level constructs synthesize the best ideas found in several scene graph–based systems. Java 3D introduces some concepts not commonly considered part of the graphics environment, such as 3D spatial sound. Java 3D's sound capabilities help to provide a more immersive experience for the user.

1.1 Goals

Java 3D was designed with several goals in mind. Chief among them is high performance. Several design decisions were made so that Java 3D implementations can deliver the highest level of performance to application users. In particular, when trade-offs were made, the alternative that benefited runtime execution was chosen.

Other important Java 3D goals are to

- Provide a rich set of features for creating interesting 3D worlds, tempered by the need to avoid nonessential or obscure features. Features that could be layered on top of Java 3D were not included.
- Provide a high-level object-oriented programming paradigm that enables developers to deploy sophisticated applications and applets rapidly.
- Provide support for runtime loaders. This allows Java 3D to accommodate a wide variety of file formats, such as vendor-specific CAD formats, interchange formats, VRML 1.0, and VRML 2.0.

1.2 Programming Paradigm

Java 3D is an object-oriented API. Applications construct individual graphics elements as separate objects and connect them together into a treelike structure called a *scene graph*. The application manipulates these objects using their predefined accessor, mutator, and node-linking methods.

1.2.1 The Scene Graph Programming Model

Java 3D's scene graph–based programming model provides a simple and flexible mechanism for representing and rendering scenes. The scene graph contains a complete description of the entire scene, or virtual universe. This includes the geometric data, the attribute information, and the viewing information needed to render the scene from a particular point of view. Chapter 2, "Scene Graph Basics," provides more information on the Java 3D scene graph programming model.

The Java 3D API improves on previous graphics APIs by eliminating many of the bookkeeping and programming chores that those APIs impose. Java 3D allows the programmer to think about geometric objects rather than about triangles—about the scene and its composition rather than about how to write the rendering code for efficiently displaying the scene.

1.2.2 Rendering Modes

Java 3D includes three different rendering modes: immediate mode, retained mode, and compiled-retained mode (see Chapter 12, "Execution and Rendering Model"). Each successive rendering mode allows Java 3D more freedom in optimizing an application's execution. Most Java 3D applications will want to take advantage of the convenience and performance benefits that the retained and compiled-retained modes provide.

1.2.2.1 Immediate Mode

Immediate mode leaves little room for global optimization at the scene graph level. Even so, Java 3D has raised the level of abstraction and accelerates immediate mode rendering on a per-object basis. An application must provide a Java 3D draw method with a complete set of points, lines, or triangles, which are then rendered by the high-speed Java 3D renderer. Of course, the application can build these lists of points, lines, or triangles in any manner it chooses.

1.2.2.2 Retained Mode

Retained mode requires an application to construct a scene graph and specify which elements of that scene graph may change during rendering. The scene graph describes the objects in the virtual universe, the arrangement of those objects, and how the application animates those objects.

1.2.2.3 Compiled-Retained Mode

Compiled-retained mode, like retained mode, requires the application to construct a scene graph and specify which elements of the scene graph may change during rendering. Additionally, the application can compile some or all of the subgraphs that make up a complete scene graph. Java 3D compiles these graphs into an internal format. The compiled representation of the scene graph may bear little resemblance to the original tree structure provided by the application, however, it is functionally equivalent. Compiled-retained mode provides the highest performance.

1.2.3 Extensibility

Most Java 3D classes expose only accessor and mutator methods. Those methods operate only on that object's internal state, making it meaningless for an application to override them. Therefore, Java 3D declares most methods as final.

Applications can extend Java 3D's classes and add their own methods. However, they may not override Java 3D's scene graph traversal semantics because the nodes do not contain explicit traversal and draw methods. Java 3D's renderer retains those semantics internally.

Java 3D *does* provide hooks for mixing Java 3D–controlled scene graph rendering and user-controlled rendering using Java 3D's immediate mode constructs (see Section 13.1.2, "Mixed-Mode Rendering"). Alternatively, the application can stop Java 3D's renderer and do all its drawing in immediate mode (see Section 13.1.1, "Pure Immediate-Mode Rendering").

Behaviors require applications to extend the Behavior object and to override its methods with user-written Java code. These extended objects should contain references to those scene graph objects that they will manipulate at run time. Chapter 9, “Behaviors and Interpolators,” describes Java 3D’s behavior model.

1.3 High Performance

Java 3D’s programming model allows the Java 3D API to do the mundane tasks, such as scene graph traversal, managing attribute state changes, and so forth, thereby simplifying the application’s job. Java 3D does this without sacrificing performance. At first glance, it might appear that this approach would create more work for the API, however, it actually has the opposite effect. Java 3D’s higher level of abstraction not only changes the amount but, more important, also the kind of work the API must perform. Java 3D does not need to impose the same type of constraints as do APIs with a lower level of abstraction, thus allowing Java 3D to introduce optimizations not possible with these lower-level APIs.

Additionally, leaving the details of rendering to Java 3D allows it to tune the rendering to the underlying hardware. For example, relaxing the strict rendering order imposed by other APIs allows parallel traversal as well as parallel rendering. Knowing which portions of the scene graph cannot be modified at run time allows Java 3D to flatten the tree, pretransform geometry, or represent the geometry in a native hardware format without the need to keep the original data.

1.3.1 Layered Implementation

Besides optimizations at the scene graph level, one of the more important factors that determines the performance of Java 3D is the time it takes to render the visible geometry. Java 3D implementations are layered to take advantage of the native, low-level API that is available on a given system. In particular, we anticipate that Java 3D implementations that use Direct3D, OpenGL, and QuickDraw3D will become available. This means that Java 3D rendering will be accelerated across the same wide range of systems that are supported by these lower-level APIs.

1.3.2 Target Hardware Platforms

Java 3D is aimed at a wide range of 3D-capable hardware and software platforms, from low-cost PC game cards and software renderers at the low end, through midrange workstations, all the way up to very high-performance specialized 3D image generators.

Java 3D implementations are expected to provide useful rendering rates on most modern PCs, especially those with 3D graphics accelerator cards. On midrange workstations, Java 3D is expected to provide applications with nearly full-speed hardware performance.

Finally, Java 3D is designed to scale as the underlying hardware platforms increase in speed over time. Tomorrow's 3D PC game accelerators will support more complex virtual worlds than high-priced workstations of a few years ago. Java 3D is prepared to meet this increase in hardware performance.

1.4 Support for Building Applications and Applets

Java 3D neither anticipates nor directly supports every possible 3D need. Instead it provides support for adding those features through Java code.

Objects defined using a computer-aided design (CAD) system or an animation system may be included in a Java 3D-based application. Most such modeling packages have an external format (sometimes proprietary). Designers can export geometry designed using an external modeler to a file. Java 3D can use that geometric information, but only if an application provides a means for reading and translating the modeler's file format into Java 3D primitives.

Similarly, VRML loaders will parse and translate VRML files and generate the appropriate Java 3D objects and Java code necessary to support the file's contents. For more information, see Appendix F, "VRML Support."

1.4.1 Browsers

Today's Internet browsers support 3D content by passing such data to plug-in 3D viewers that render into their own window. It is anticipated that, over time, the display of 3D content will become integrated into the main browser display. In fact, some of today's 3D browsers display 2D content as 2D objects within a 3D world.

1.4.2 Games

Developers of 3D game software have typically attempted to wring out every last ounce of performance from the hardware. Historically they have been quite willing to use hardware-specific, nonportable optimizations to get the best performance possible. As such, in the past, game developers have tended to program below the level of easy-to-use software such as Java 3D. However, the trend in

3D games today is to leverage general-purpose 3D hardware accelerators and to use fewer “tricks” in rendering.

So, while Java 3D was not explicitly designed to match the game developer’s every expectation, Java 3D’s sophisticated implementation techniques should provide more than enough performance to support many game applications. One might argue that applications written using a general API like Java 3D may have a slight performance penalty over those employing special, nonportable techniques. However, other factors such as portability, time to market, and development cost must be weighed against absolute peak performance.

1.5 Overview of Java 3D Object Hierarchy

Java 3D defines several basic classes that are used to construct and manipulate a scene graph and to control viewing and rendering. Figure 1-1 shows the overall object hierarchy used by Java 3D. Subsequent chapters provide more detail for specific portions of the hierarchy.

---

javax.media.j3d
- VirtualUniverse
- Locale
- View
- PhysicalBody
- PhysicalEnvironment
- Screen3D
- Canvas3D (extends awt.Canvas)
- SceneGraphObject
    - Node
        - Group
        - Leaf
    - NodeComponent
        - *Various component objects*
- Transform3D

javax.vecmath
- *Matrix classes*
- *Tuple classes*

Figure 1-1 Java 3D Object Hierarchy

1.6 Structuring the Java 3D Program

This section illustrates how a developer might structure a Java 3D application. The simple application in this example creates a scene graph that draws an object in the middle of a window and rotates the object about its center point.

1.6.1 Java 3D Application Scene Graph

The scene graph for the sample application is shown in Figure 1-2.

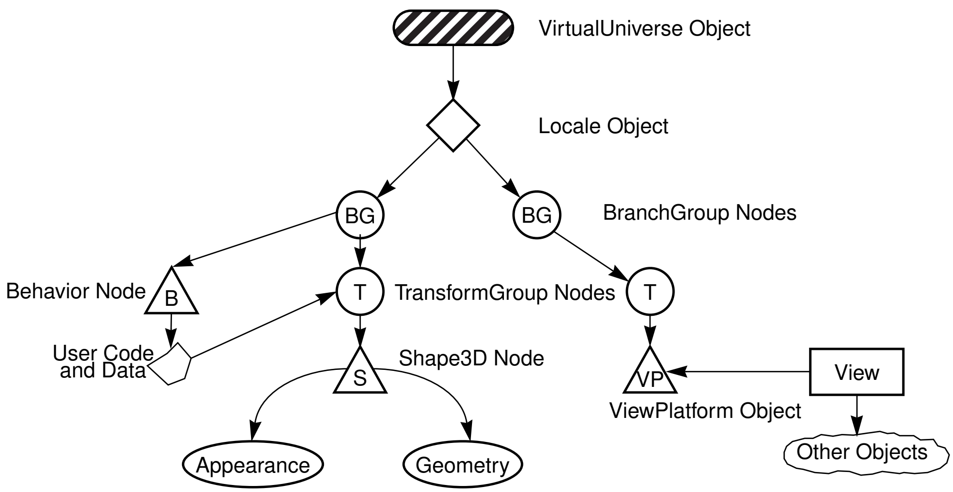


Figure 1-2 Application Scene Graph

The scene graph consists of superstructure components—a VirtualUniverse object and a Locale object—and a set of branch graphs. Each branch graph is a subgraph that is rooted by a BranchGroup node that is attached to the superstructure. For more information, see Chapter 2, “Scene Graph Basics.”

A VirtualUniverse object defines a named universe. Java 3D permits the creation of more than one universe, though the vast majority of applications will use just one. The VirtualUniverse object provides a grounding for scene graphs. All Java 3D scene graphs must connect to a VirtualUniverse object to be displayed. For more information, see Chapter 3, “Scene Graph Superstructure.”

Below the VirtualUniverse object is a Locale object. The Locale object defines the origin, in high-resolution coordinates, of its attached branch graphs. A virtual universe may contain as many Locales as needed. In this example, a single Locale object is defined with its origin at (0.0, 0.0, 0.0).

The scene graph itself starts with the BranchGroup nodes (see Section 4.2, "BranchGroup Node"). A BranchGroup serves as the root of a subgraph, called a *branch graph*, of the scene graph. Only BranchGroup objects can attach to Locale objects.

In this example there are two branch graphs and, thus, two BranchGroup nodes. Attached to the left BranchGroup are two subgraphs. One subgraph consists of a user-extended Behavior leaf node. The Behavior node contains Java code for manipulating the transformation matrix associated with the object's geometry.

The other subgraph in this BranchGroup consists of a TransformGroup node that specifies the position (relative to the Locale), orientation, and scale of the geometric objects in the virtual universe. A single child, a Shape3D leaf node, refers to two component objects: a Geometry object and an Appearance object. The Geometry object describes the geometric shape of a 3D object (a cube in our simple example). The Appearance object describes the appearance of the geometry (color, texture, material reflection characteristics, and so forth).

The right BranchGroup has a single subgraph that consists of a TransformGroup node and a ViewPlatform leaf node. The TransformGroup specifies the position (relative to the Locale), orientation, and scale of the ViewPlatform. This transformed ViewPlatform object defines the end user's view within the virtual universe.

Finally, the ViewPlatform is referenced by a View object that specifies all of the parameters needed to render the scene from the point of view of the ViewPlatform. Also referenced by the View object are other objects that contain information, such as the drawing canvas into which Java 3D renders, the screen that contains the canvas, and information about the physical environment.

1.6.2 Recipe for a Java 3D Program

The following steps are taken by the example program to create the scene graph elements and link them together. Java 3D will then render the scene graph and display the graphics in a window on the screen:

1. Create a Canvas3D object and add it to the Applet panel.
2. Create a BranchGroup as the root of the scene branch graph.
3. Construct a Shape3D node with a TransformGroup node above it.
4. Attach a RotationInterpolator behavior to the TransformGroup.

5. Call the universe builder utility function to do the following:
   a. Establish a virtual universe with a single high-resolution Locale (see Chapter 2, “Scene Graph Basics”).
   b. Create the PhysicalBody, PhysicalEnvironment, View, and ViewPlatform objects.
   c. Create a BranchGroup as the root of the view platform branch graph.
   d. Insert the view platform branch graph into the Locale.
6. Insert the scene branch graph into the universe builder’s Locale.

The Java 3D renderer then starts running in an infinite loop. The renderer conceptually performs the following operations:

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

1.6.3 HelloUniverse: A Sample Java 3D Program

Here are code fragments from a simple program, `HelloUniverse.java`, that creates a cube and a RotationInterpolator behavior object that rotates the cube at a constant rate of $\pi/2$ radians per second.

1.6.3.1 HelloUniverse Class

The HelloUniverse class, on the next page, creates the branch graph that includes the cube and the RotationInterpolator behavior. It then adds this branch graph to the Locale object generated by the UniverseBuilder utility.

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

        // Create the TransformGroup node and initialize it to the
        // identity. Enable the TRANSFORM_WRITE capability so that
        // our behavior code can modify it at run time. Add it to
        // the root of the subgraph.
        TransformGroup objTrans = new TransformGroup();
        objTrans.setCapability(
                        TransformGroup.ALLOW_TRANSFORM_WRITE);
        objRoot.addChild(objTrans);
        // Create a simple Shape3D node; add it to the scene graph.
        objTrans.addChild(new ColorCube().getShape());

        // Create a new Behavior object that will perform the
        // desired operation on the specified transform and add
        // it into the scene graph.
        Transform3D yAxis = new Transform3D();
        Alpha rotationAlpha = new Alpha(
                -1, Alpha.INCREASING_ENABLE,
                0, 0,     4000, 0, 0,         0, 0, 0);
        RotationInterpolator rotator = new RotationInterpolator(
                rotationAlpha, objTrans, yAxis,
                0.0f, (float) Math.PI*2.0f);
        BoundingSphere bounds =
            new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        rotator.setSchedulingBounds(bounds);
        objTrans.addChild(rotator);

        return objRoot;
    }

    public HelloUniverse() {
        setLayout(new BorderLayout());
        Canvas3D c = new Canvas3D(graphicsConfig);
        add("Center", c);
        // Create a simple scene and attach it to the virtual
        // universe
        BranchGroup scene = createSceneGraph();
        UniverseBuilder u = new UniverseBuilder(c);
        u.addBranchGraph(scene);
    }
}
```

1.6.3.2 UniverseBuilder Class

The UniverseBuilder class establishes and initializes Java 3D's virtual universe, Locale, and viewing objects, and constructs the view platform branch graph. The example code shown below is a simplified version of the UniverseBuilder that will be supplied as part of the Java 3D utility package.

```
public class UniverseBuilder extends Object {
    // User-specified canvas
    Canvas3D canvas;

    // Scene graph elements to which the user may want access
    VirtualUniverse     universe;
    Locale              locale;
    TransformGroup      vpTrans;
    View                view;

    public UniverseBuilder(Canvas3D c) {
        this.canvas = c;

        // Establish a virtual universe that has a single
        // hi-res Locale
        universe = new VirtualUniverse();
        locale = new Locale(universe);

        // Create a PhysicalBody and PhysicalEnvironment object
        PhysicalBody body = new PhysicalBody();
        PhysicalEnvironment environment =
                                new PhysicalEnvironment();

        // Create a View and attach the Canvas3D and the physical
        // body and environment to the view.
        view = new View();
        view.addCanvas3D(c);
        view.setPhysicalBody(body);
        view.setPhysicalEnvironment(environment);

        // Create a BranchGroup node for the view platform
        BranchGroup vpRoot = new BranchGroup();

        // Create a ViewPlatform object, and its associated
        // TransformGroup object, and attach it to the root of the
        // subgraph. Attach the view to the view platform.
        Transform3D t = new Transform3D();
        t.set(new Vector3f(0.0f, 0.0f, 2.0f));
        ViewPlatform vp = new ViewPlatform();
        vpTrans = new TransformGroup(t);
```

```
        vpTrans.addChild(vp);
        vpRoot.addChild(vpTrans);

        view.attachViewPlatform(vp);

        // Attach the branch graph to the universe, via the
        // Locale. The scene graph is now live!
        locale.addBranchGraph(vpRoot);
    }

    public void addBranchGraph(BranchGroup bg) {
        locale.addBranchGraph(bg);
    }
}
```

1.6.3.3 ColorCube Class

The ColorCube Class creates a Shape3D node that contains the geometry for an unlit, colored cube.

```
public class ColorCube extends Object {
    private static final float[] verts = {
    // front face
         1.0f, -1.0f,  1.0f,     1.0f,  1.0f,  1.0f,
        -1.0f,  1.0f,  1.0f,    -1.0f, -1.0f,  1.0f,
    // back face
        -1.0f, -1.0f, -1.0f,    -1.0f,  1.0f, -1.0f,
         1.0f,  1.0f, -1.0f,     1.0f, -1.0f, -1.0f,
    // right face
         1.0f, -1.0f, -1.0f,     1.0f,  1.0f, -1.0f,
         1.0f,  1.0f,  1.0f,     1.0f, -1.0f,  1.0f,
    // left face
        -1.0f, -1.0f,  1.0f,    -1.0f,  1.0f,  1.0f,
        -1.0f,  1.0f, -1.0f,    -1.0f, -1.0f, -1.0f,
    // top face
         1.0f,  1.0f,  1.0f,     1.0f,  1.0f, -1.0f,
        -1.0f,  1.0f, -1.0f,    -1.0f,  1.0f,  1.0f,
    // bottom face
        -1.0f, -1.0f,  1.0f,    -1.0f, -1.0f, -1.0f,
         1.0f, -1.0f, -1.0f,     1.0f, -1.0f,  1.0f,
    };
    private static final float[] colors = {
    // front face (red)
        1.0f, 0.0f, 0.0f,       1.0f, 0.0f, 0.0f,
        1.0f, 0.0f, 0.0f,       1.0f, 0.0f, 0.0f,
    // back face (green)
        0.0f, 1.0f, 0.0f,       0.0f, 1.0f, 0.0f,
```

```
        0.0f, 1.0f, 0.0f,       0.0f, 1.0f, 0.0f,
    // right face (blue)
        0.0f, 0.0f, 1.0f,       0.0f, 0.0f, 1.0f,
        0.0f, 0.0f, 1.0f,       0.0f, 0.0f, 1.0f,
    // left face (yellow)
        1.0f, 1.0f, 0.0f,       1.0f, 1.0f, 0.0f,
        1.0f, 1.0f, 0.0f,       1.0f, 1.0f, 0.0f,
    // top face (magenta)
        1.0f, 0.0f, 1.0f,       1.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 1.0f,       1.0f, 0.0f, 1.0f,
    // bottom face (cyan)
        0.0f, 1.0f, 1.0f,       0.0f, 1.0f, 1.0f,
        0.0f, 1.0f, 1.0f,       0.0f, 1.0f, 1.0f,
    };

    private Shape3D shape;

    public ColorCube() {
        QuadArray cube = new QuadArray(24,
                    QuadArray.COORDINATES | QuadArray.COLOR_3);

        cube.setCoordinates(0, verts);
        cube.setColors(0, colors);

        shape = new Shape3D(cube, new Appearance());
    }

    public Shape3D getShape() {
        return shape;
    }
}
```

CHAPTER 2

Scene Graph Basics

A scene graph consists of Java 3D objects, called *nodes*, arranged in a tree structure. The user creates one or more scene subgraphs and attaches them to a virtual universe. The individual connections between Java 3D nodes always represent a directed relationship: parent to child. Java 3D restricts scene graphs in one major way: Scene graphs may not contain cycles. Thus, a Java 3D scene graph is a directed acyclic graph (DAG). See Figure 2-1.

Java 3D refines the Node object class into two subclasses: Group and Leaf node objects. Group node objects group together one or more child nodes. A group node can point to zero or more children but can have only one parent. The SharedGroup node cannot have any parents (although it allows sharing portions of a scene graph, as described in Chapter 6, "Reusing Scene Graphs"). Leaf node objects contain the actual definitions of shapes (geometry), lights, fog, sounds, and so forth. A leaf node has no children and only one parent. The semantics of the various group and leaf nodes are described in subsequent chapters.

2.1 Scene Graph Structure

A scene graph organizes and controls the rendering of its constituent objects. The Java 3D renderer draws a scene graph in a consistent way that allows for concurrence. The Java 3D renderer can draw one object independently of other objects. Java 3D can allow such independence because its scene graphs have a particular form and cannot share state among branches of a tree.

2.1.1 Spatial Separation

The hierarchy of the scene graph encourages a natural spatial grouping on the geometric objects found at the leaves of the graph. Internal nodes act to group their children together. A group node also defines a spatial bound that contains all the geometry defined by its descendants. Spatial grouping allows for efficient implementation of operations such as proximity detection, collision detection, view frustum culling, and occlusion culling.

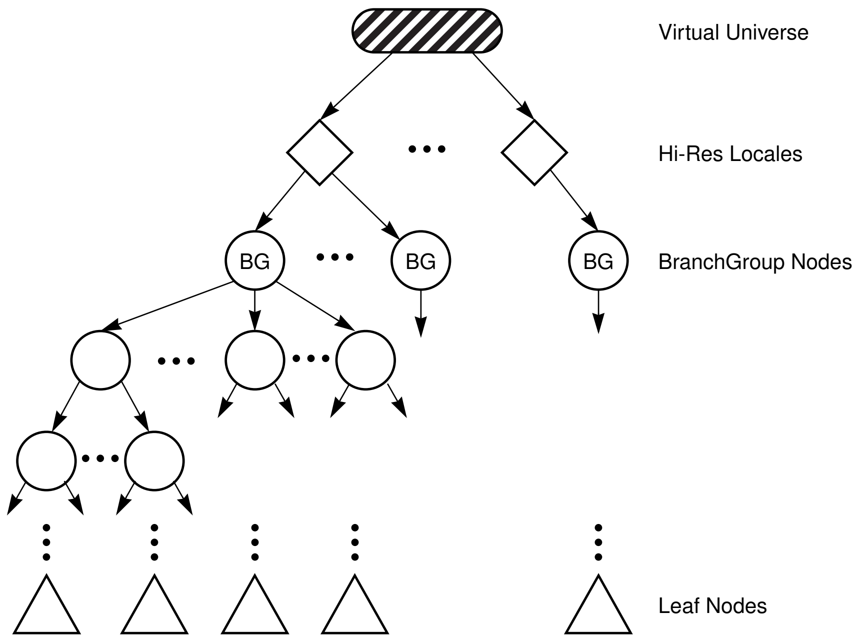


Figure 2-1 A Java 3D Scene Graph Is a DAG (Directed Acyclic Graph)

2.1.2 State Inheritance

A leaf node's state is defined by the nodes in a direct path between the scene graph's root and the leaf. Because a leaf's graphics context only relies on a linear path between the root and that node, the Java 3D renderer can decide to traverse the scene graph in whatever order it wishes. It can traverse the scene graph from left to right and top to bottom, in level order from right to left, or even in parallel. The only exceptions to this rule are spatially bounded attributes such as lights and fog.

This characteristic is in marked contrast to many older scene graph–based APIs (including PHIGS and SGI's Inventor), where if a node above or to the left of a node changes the graphics state, the change affects the graphics state of all nodes below it or to its right.

The most common node object, along the path from the root to the leaf, that changes the graphics state is the TransformGroup object. The TransformGroup object can change the position, orientation, and scale of the objects below it.

Most graphics state attributes are set by a Shape3D leaf node through its constituent Appearance object, thus allowing parallel rendering. The Shape3D node also has a constituent Geometry object that specifies its geometry—this permits different shape objects to share common geometry without sharing material attributes (or vice versa).

2.1.3 Rendering

The Java 3D renderer incorporates all graphics state changes made in a direct path from a scene graph root to a leaf object in the drawing of that leaf object. Java 3D provides this semantic for both retained and compiled-retained modes.

2.2 Scene Graph Objects

A Java 3D scene graph consists of a collection of Java 3D node objects connected in a tree structure. These node objects reference other scene graph objects called *node component objects*. All scene graph node and component objects are subclasses of a common SceneGraphObject class. The SceneGraphObject class is an abstract class that defines methods that are common among nodes and component objects.

Scene graph objects are constructed by creating a new instance of the desired class and are accessed and manipulated using the object's `set` and `get` methods. Once a scene graph object is created and connected to other scene graph objects to form a subgraph, the entire subgraph can be attached to a virtual universe—via a high-resolution Locale object—making the object *live* (see Section 3.6.2, "Locale Object"). Prior to attaching a subgraph to a virtual universe, the entire subgraph can be *compiled* into an optimized, internal format (see Section 4.2, "BranchGroup Node").

An important characteristic of all scene graph objects is that they can only be accessed or modified during the creation of a scene graph, except where explicitly allowed. Access to most `set` and `get` methods of objects that are part of a live or compiled scene graph is restricted. Such restrictions provide the scene graph compiler with usage information it can use in optimally compiling or rendering a scene graph. Each object has a set of capability bits that enable certain functionality when the object is live or compiled. By default, all capability bits are disabled (cleared). Only those `set` and `get` methods corresponding to capability bits that are explicitly enabled (set) prior to the object being compiled or made live are legal. The methods for setting and getting capability bits are described next.

Constructors

The SceneGraphObject specifies one constructor.

`public SceneGraphObject()`

Constructs a new SceneGraphObject.

Methods

The following methods are available on all scene graph objects.

`public final boolean isCompiled()`
`public final boolean isLive()`

The first method returns a flag that indicates whether the node is part of a scene graph that has been compiled. If so, only those capabilities explicitly allowed by the object's capability bits are allowed. The second method returns a flag that indicates whether the node is part of a scene graph that has been attached to a virtual universe via a high-resolution Locale object.

`public final boolean getCapability(int bit)`
`public final void setCapability(int bit)`
`public final void clearCapability(int bit)`

These three methods provide applications with the means for accessing and modifying the capability bits of a scene graph object. The bit positions of the capability bits are defined as public static final constants on a per-object basis. Every instance of every scene graph object has its own set of capability bits. An example of a capability bit is the `ALLOW_BOUNDS_WRITE` bit in node objects. Only those methods corresponding to capabilities that are enabled *before* the object is first compiled or made live are subsequently allowed for that object. A `RestrictedAccessException` is thrown if an application calls `setCapability` or `clearCapability` on live or compiled objects. Note that only a single bit may be set or cleared per method invocation—bits may *not* be ORed together.

`public void setUserData(Object userData)`
`public Object getUserData()`

These methods access or modify the userData field associated with this scene graph object. The userData field is a reference to an arbitrary object and may be used to store any user-specific data associated with this scene graph object—it is not used by the Java 3D API. If this object is cloned, the userData field is copied to the newly cloned object.

2.2.1 Node Objects

Node objects divide into group node objects and leaf node objects. Group nodes serve to group their child node objects together according to the group node's semantics. Leaf nodes specify the actual elements that Java 3D uses in rendering; specifically, geometric objects, lights, and sounds. These node objects are described in Chapter 4, "Group Node Objects" and Chapter 5, "Leaf Node Objects."

Constants

Node object constants allow an application to individually enable runtime capabilities. These capability bits are enforced only when the node is part of a live or compiled scene graph.

`public static final int ALLOW_PICK`

This is a deprecated method. Use `setPickable(boolean)` instead.

`public static final int ALLOW_BOUNDS_READ`
`public static final int ALLOW_BOUNDS_WRITE`

These bits, when set using the `setCapability` method, specify that the node will permit an application to invoke the `getBounds` and `setBounds` methods, respectively. An application can choose to enable a particular `set` method but not the associated `get` method, or vice versa. The application can choose to enable both methods or, by default, leave the method(s) disabled.

`public static final int ALLOW_AUTO_COMPUTE_BOUNDS_READ`
`public static final int ALLOW_AUTO_COMPUTE_BOUNDS_WRITE`

These bits, when set using the `setCapability` method, specify that the node will permit an application to invoke the `getBoundsAutoCompute` and `setBoundsAutoCompute` methods, respectively. An application can choose to enable a particular `set` method but not the associated `get` method, or vice versa. The application can choose to enable both methods or, by default, leave the method(s) disabled.

`public static final int ENABLE_PICK_REPORTING`

This flag specifies that this node will be reported in a SceneGraphPath. By default, this is disabled.

`public static final int ALLOW_PICKABLE_READ`
`public static final int ALLOW_PICKABLE_WRITE`

These flags specify that this Node can have its pickability read or changed.

`public static final int ENABLE_COLLISION_REPORTING`

This flag specifies that this Node will be reported in the collision SceneGraphPath if a collision occurs. This capability is only specifiable for Group nodes; it is ignored for Leaf nodes. The default for Group nodes is false. All interior nodes not needed for uniqueness in a SceneGraphPath that don't have this flag set to true will not be reported in the SceneGraphPath.

`public static final int ALLOW_COLLIDABLE_READ`
`public static final int ALLOW_COLLIDABLE_WRITE`

These flags specify that this Node allows read or write access to its collidability state.

`public static final int ALLOW_LOCAL_TO_VWORLD_READ`

This flag specifies that this node allows read access to its local-coordinates-to-virtual-world-(Vworld)-coordinates transform.

Constructors

The Node object specifies the following constructor.

`public Node()`

This constructor constructs and initializes a Node object. The Node class provides an abstract class for all group and leaf nodes. It provides a common framework for constructing a Java 3D scene graph, specifically, bounding volumes.

Methods

The following methods are available on Node objects, subject to the capabilities that are enabled for live or compiled nodes.

`public final Node getParent()`

Retrieves the parent of this node, or `null` if this node has no parent. This method is only valid during the construction of the scene graph. If this object is part of a live or compiled scene graph, a `RestrictedAccessException` will be thrown.

`public final Bounds getBounds()`
`public final void setBounds(Bounds bounds)`

These methods access or modify this node's geometric bounds.

`public final void getLocalToVworld(Transform3D t)`
`public final void getLocalToVworld(SceneGraphPath path, Transform3D t)`

These methods access the local-coordinates-to-virtual-world-coordinates transform for this node and place the result into the specified Transform3D argument. The first form is used for nodes that are *not* part of a shared subgraph, the second form is used for nodes that *are* part of a shared subgraph. The local-coordinates-to-Vworld-coordinates transform is the composite of all transforms in the scene graph from the root down to this node (via the specified Link nodes, in the second case). It is only valid for nodes that are part of a live scene graph. An exception will be thrown if the node is not part of a live scene graph or if the appropriate capability is not set. Additionally, the first form will throw an exception if the node is part of a shared subgraph.

`public final void setBoundsAutoCompute(boolean autoCompute)`
`public final boolean getBoundsAutoCompute()`

These methods set and get the value that determines whether the node's geometric bounds are computed automatically, in which case the bounds will be read-only, or are set manually, in which case the value specified by `setBounds` will be used. The default is automatic.

`public void setPickable(boolean pickable)`
`public boolean getPickable()`

These methods set and retrieve the flag indicating whether this node can be picked. A setting of `false` means that this node and its children are all unpickable.

`public void setCollidable(boolean collidable)`
`public boolean getCollidable()`

The set method sets the collidable value. The get method returns the collidable value. This value determines whether this node and its children, if a group node, can be considered for collision purposes. If the value is false, neither this node nor any children nodes will be traversed for collision purposes. The default value is true. The collidable setting is the way that an application can perform collision culling.

`public Node cloneNode(boolean forceDuplicate)`

This method creates a new instance of the node. This routine is called by `cloneTree` to duplicate the current node. `cloneNode` should be overridden by any user-subclassed objects. All subclasses must have their `cloneNode` method consist of the following lines:

```
public Node cloneNode(boolean forceDuplicate) {
   UserSubClass usc = new UserSubClass();
   usc.duplicateNode(this, forceDuplicate);
   return usc;
}
```

`public void duplicateNode(Node originalNode,`
`       boolean forceDuplicate)`

This method copies all the node information from the `originalNode` into the current node. This method is called from the `cloneNode` method, which is in turn called by the `cloneTree` method.

For each NodeComponent object contained by the object being duplicated, the NodeComponent's `duplicateOnCloneTree` value is used to determine whether the NodeComponent should be duplicated in the new node or if just a reference to the current node should be placed in the new node. This flag can be overridden by setting the `forceDuplicate` parameter in the `cloneTree` method to `true`.

`public Node cloneTree()`
`public Node cloneTree(boolean forceDuplicate)`
`public Node cloneTree(boolean forceDuplicate,`
`       boolean allowDanglingReference)`

These methods duplicate all the nodes of the specified subgraph. Group nodes are duplicated via a call to `cloneNode`, and then `cloneTree` is called for each child node. For leaf nodes, component data can either be duplicated or be made a reference to the original data. Leaf node `cloneTree` behavior is determined by the `duplicateOnCloneTree` flag found in every leaf node's component data class and by the `forceDuplicate` parameter. The `forceDuplicate` parameter, when set to `true`, causes the `duplicateOnCloneTree` flag to be ignored. The `allowDanglingReferences` flag, when set to `true`, allows the `cloneTree` method to complete even when a dangling reference is discovered. When this parameter is `false`, a `DanglingReferenceException` is generated as soon as `cloneTree` detects this situation.

2.2.2 NodeComponent Objects

Node component objects include the actual geometry and appearance attributes used to render the geometry. These component objects are described in Chapter 7, "Node Component Objects."

Constructors

The NodeComponent object specifies the following constructor.

`public NodeComponent()`

This constructor constructs and initializes a NodeComponent object. The NodeComponent class provides an abstract class for all component objects.

Methods

The following methods are available on NodeComponent objects.

`public void setDuplicateOnCloneTree(boolean duplicate)`
`public boolean getGetDuplicateOnCloneTree()`

These methods access or modify the `duplicateOnCloneTree` value of the NodeComponent object. The `duplicateOnCloneTree` value is used by the `cloneTree` method to determine if NodeComponent objects should be duplicated or just referenced in the cloned leaf object.

`public NodeComponent cloneNodeComponent()`

This method creates a new instance of a NodeComponent object. This method is called by the `cloneNode` method to duplicate the current node. The `cloneNodeComponent` should be overridden by any user-subclassed NodeComponent objects. All subclasses must have their `cloneNodeComponent` method consist of the following lines:

```
public NodeComponent cloneNodeComponent() {
   UserNodeComponent unc = new UserNodeComponent();
   unc.duplicateNodeComponent(this);
   return unc;
}
```

```
public void duplicateNodeComponent(NodeComponent
       originalNodeComponent)
```

This method copies all node information from `originalNodeComponent` into the current node. This method is called from the `cloneNodeComponent` method, which is in turn called by the `cloneNode` method.

2.3 Scene Graph Superstructure Objects

Java 3D defines two scene graph superstructure objects, VirtualUniverse and Locale, which are used to contain collections of subgraphs that comprise the scene graph. These objects are described in more detail in Chapter 3, "Scene Graph Superstructure."

2.3.1 VirtualUniverse Object

A VirtualUniverse object consists of a list of Locale objects that contain a collection of scene graph nodes that exist in the universe. Typically, an application will need only one VirtualUniverse, even for very large virtual databases. Operations on a VirtualUniverse include enumerating the Locale objects contained within the universe. See Section 3.6.1, "VirtualUniverse Object," for more information.

2.3.2 Locale Object

The Locale object acts as a container for a collection of subgraphs of the scene graph that are rooted by a BranchGroup node. A Locale also defines a location within the virtual universe using high-resolution coordinates (HiResCoord) to specify its position. The HiResCoord serves as the origin for all scene graph objects contained within the Locale.

A Locale has no parent in the scene graph, but is implicitly attached to a virtual universe when it is constructed. A Locale may reference an arbitrary number of BranchGroup nodes, but has no explicit children.

The coordinates of all scene graph objects are relative to the HiResCoord of the Locale in which they are contained. Operations on a Locale include setting or getting the HiResCoord of the Locale, adding a subgraph, and removing a subgraph (see Section 3.6.2, "Locale Object," for more information).

2.4 Scene Graph Viewing Objects

Java 3D defines five scene graph viewing objects that are not part of the scene graph per se but serve to define the viewing parameters and to provide hooks into the physical world. These objects are Canvas3D, Screen3D, View, PhysicalBody, and PhysicalEnvironment. They are described in more detail in Chapter 8, "View Model," and Appendix C, "View Model Details."

2.4.1 Canvas3D Object

The Canvas3D object encapsulates all of the parameters associated with the window being rendered into (see Section 8.9, "The Canvas3D Object"). When a Canvas3D object is attached to a View object, the Java 3D traverser renders the specified view onto the canvas. Multiple Canvas3D objects can point to the same View object.

2.4.2 Screen3D Object

The Screen3D object encapsulates all of the parameters associated with the physical screen containing the canvas, such as the width and height of the screen in pixels, the physical dimensions of the screen, and various physical calibration values (see Section 8.8, "The Screen3D Object").

2.4.3 View Object

The View object specifies information needed to render the scene graph. Figure 2-2 shows a View object attached to a simple scene graph for viewing the scene.

The View object is the central Java 3D object for coordinating all aspects of viewing (see Section 8.7, "The View Object"). All viewing parameters in Java 3D are either directly contained within the View object or within objects pointed to by a View object. Java 3D supports multiple simultaneously active View objects, each of which can render to one or more canvases.

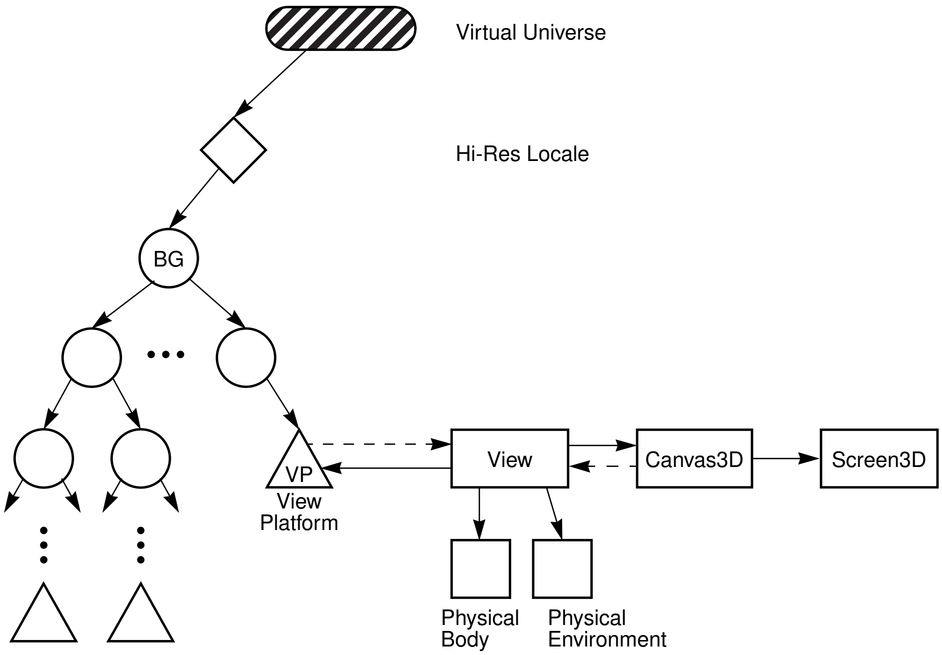


Figure 2-2 Viewing a Scene Graph

2.4.4 PhysicalBody Object

The PhysicalBody object encapsulates all of the parameters associated with the physical body, such as head position, right and left eye position, and so forth. (see Section 8.10, “The PhysicalBody Object”).

2.4.5 PhysicalEnvironment Object

The PhysicalEnvironment object encapsulates all of the parameters associated with the physical environment, such as calibration information for the tracker base for the head or hand tracker (see Section 8.11, “The PhysicalEnvironment Object”).

CHAPTER 3

Scene Graph Superstructure

JAVA 3D's superstructure consists of one or more VirtualUniverse objects, each of which contains a set of one or more high-resolution Locale objects. The Locale objects, in turn, contain collections of subgraphs that comprise the scene graph (see Figure 3-1).

3.1 The Virtual Universe

Java 3D defines the concept of a *virtual universe* as a three-dimensional space with an associated set of objects. Virtual universes serve as the largest unit of aggregate representation, and can also be thought of as databases. Virtual universes can be very large, both in physical space units and in content. Indeed, in most cases a single virtual universe will serve an application's entire needs.

Virtual universes are separate entities in that no node object may exist in more than one virtual universe at any one time. Likewise, the objects in one virtual universe are not visible in, nor do they interact with objects in, any other virtual universe.

To support large virtual universes, Java 3D introduces the concept of Locales that have *high-resolution coordinates* as an origin. Think of high-resolution coordinates as "tie-downs" that precisely anchor the locations of objects specified using less precise floating-point coordinates that are within the range of influence of the high-resolution coordinates.

A Locale, with its associated high-resolution coordinates, serves as the next level of representation down from a virtual universe. All virtual universes contain one or more high-resolution-coordinate Locales, and all other objects are attached to a Locale. High-resolution coordinates act as an upper-level translation-only transform node. For example, the coordinates of all objects attached to a particular Locale are all relative to the location of that Locale's high-resolution coordinates.

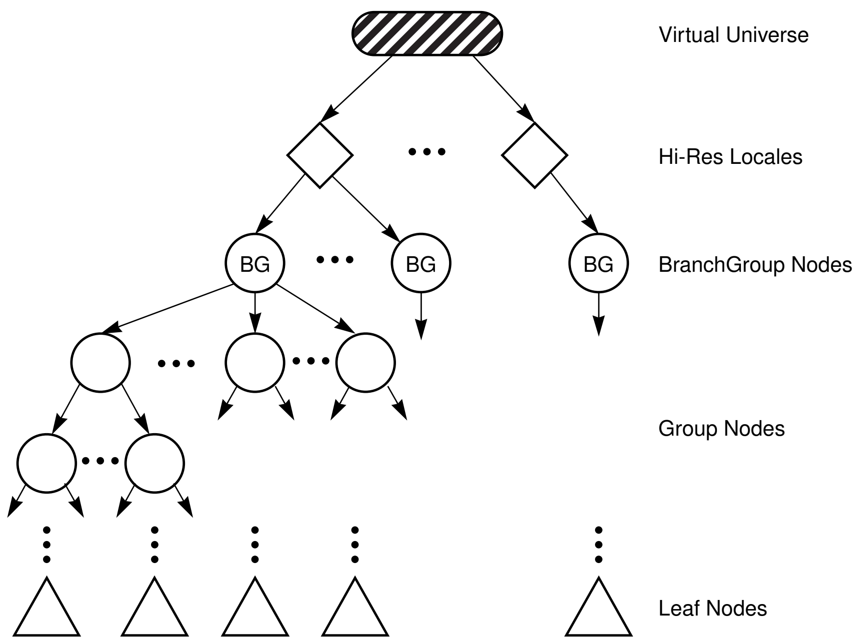


Figure 3-1 The Virtual Universe

While a virtual universe is similar to the traditional computer graphics concept of a scene graph, a given virtual universe can become so large that it is often better to think of a scene graph as the descendent of a high-resolution-coordinate Locale.

3.2 Establishing a Scene

To construct a three-dimensional scene, the programmer must execute a Java 3D program. The Java 3D application must first create a VirtualUniverse object and attach at least one Locale to it. Then the desired scene graph is constructed, starting with a BranchGroup node and including at least one ViewPlatform object, and is attached to the Locale. Finally, a View object is constructed that references the ViewPlatform object (see Section 1.6, "Structuring the Java 3D Program"). As soon as a scene graph containing a ViewPlatform is attached to the VirtualUniverse, Java 3D's rendering loop is engaged, and the scene will appear on the drawing canvas(es) associated with the View object.

3.3 Loading a Virtual Universe

Java 3D is a runtime application programming interface (API), not a file format. As an API, Java 3D provides no direct mechanism for loading or storing a virtual universe. Constructing a scene graph involves the execution of a Java 3D program. However, loaders to convert a number of standard 3D file formats to or from Java 3D virtual universes are expected to be generally available.

3.4 Coordinate Systems

By default, Java 3D coordinate systems are right-handed, with the orientation semantics being that +Y is the local gravitational up, +X is horizontal to the right, and +Z is directly toward the viewer. The default units are meters.

3.5 High-resolution Coordinates

Double-precision floating-point, single-precision floating-point, or even fixed-point representations of three-dimensional coordinates are sufficient to represent and display rich 3D scenes. Unfortunately, scenes are not worlds, let alone universes. If one ventures even a hundred miles away from the (0.0, 0.0, 0.0) origin using only single-precision floating-point coordinates, representable points become quite quantized, to at very best a third of an inch (and much more coarsely than that in practice).

To “shrink” down to a small size (say the size of an IC transistor), even very near (0.0, 0.0, 0.0), the same problem arises.

If a large contiguous virtual universe is to be supported, some form of higher-resolution addressing is required. Thus the choice of 256-bit positional components for “high-resolution” positions.

3.5.1 Java 3D High-resolution Coordinates

Java 3D high-resolution coordinates consist of three 256-bit fixed-point numbers, one each for x, y, and z. The fixed point is at bit 128, and the value 1.0 is defined to be exactly 1 meter. This coordinate system is sufficient to describe a universe in excess of several hundred billion light years across, yet still define objects smaller than a proton (down to below the planck length). Table 3-1 shows how many bits are needed above or below the fixed point to represent the range of interesting physical dimensions.

Table 3-1 Java 3D High-Resolution Coordinates

2^n Meters	Units
87.29	Universe (20 billion light years)
69.68	Galaxy (100,000 light years)
53.07	Light year
43.43	Solar system diameter
23.60	Earth diameter
10.65	Mile
9.97	Kilometer
0.00	Meter
–19.93	Micron
–33.22	Angstrom
–115.57	Planck length

A 256-bit fixed-point number also has the advantage of being able to directly represent nearly any reasonable single-precision floating-point value *exactly.*

High-resolution coordinates in Java 3D are only used to embed more traditional floating point coordinate systems within a much higher-resolution substrate. In this way a visually seamless virtual universe of any conceivable size or scale can be created, without worry about numerical accuracy.

3.5.2 Java 3D Virtual World Coordinates

Within a given virtual world coordinate system, positions are expressed by three floating point numbers. The virtual world coordinate scale is in meters, but this can be affected by scale changes in the object hierarchy.

3.5.3 Details of High-resolution Coordinates

High-resolution coordinates are represented as signed, two's-complement, fixed-point numbers consisting of 256 bits. Although Java 3D keeps the internal representation of high-resolution coordinates opaque, users specify such coordinates using 8-element integer arrays. Java 3D treats the integer found at index 0 as containing the most significant bits and that found at index 7 as containing the least significant bits of the high-resolution coordinate. The binary point is located at bit position 128, or between the integers at index 3 and 4. A high-resolution coordinate of 1.0 is 1 meter.

The semantics of how file loaders deal with high-resolution coordinates is up to the individual file loader, as Java 3D does not directly define any file-loading semantics. However, some general advice can be given (note that this advice is *not* officially part of the Java 3D specification).

For "small" virtual universes (on the order of hundreds of meters across in relative scale), a single Locale with high-resolution coordinates at location (0.0, 0.0, 0.0) as the root node (below the VirtualUniverse object) is sufficient; a loader can automatically construct this node during the loading process, and the point in high-resolution coordinates does not need any direct representation in the external file.

Larger virtual universes are expected to be usually constructed like computer directory hierarchies, that is, as a "root" virtual universe containing mostly external file references to embedded virtual universes. In this case, the file reference object (user-specific data hung off a Java 3D group or hi-res node) defines the location for the data to be read into the current virtual universe.

The data file's contents should be parented to the file object node while being read, thus inheriting the high-resolution coordinates of the file object as the new relative virtual universe origin of the embedded scene graph. If this scene graph itself contains high-resolution coordinates, it will need to be offset (translated) by the amount in the file object's high-resolution coordinates, and then added to the larger virtual universe as new high-resolution coordinates, with their contents hung off below them. Once again, the above procedure is not part of the official Java 3D specification, but some more details on the care and use of high-resolution coordinates in external file formats will probably be available as a Java 3D application note.

Authoring tools that directly support high-resolution coordinates should create additional high-resolution coordinates as a user creates new geometry "sufficiently" far away (or of different scale) from existing high-resolution coordinates.

Semantics of widely moving objects. Most fixed and nearly-fixed objects stay attached to the same high-resolution Locale. Objects that make wide changes in position or scale may need to be periodically reparented to more appropriate high-resolution Locale. If no appropriate high-resolution Locale exists, the application may need to create a new one.

Semantics of viewing. The ViewPlatform object and the associated nodes in its hierarchy are very often widely moving objects. Applications will typically attach the view platform to the most appropriate high-resolution Locale. For display, all objects will first have their positions translated by the difference between the location of their high-resolution Locale, and the view platform's high-resolution Locale. (In the common case of the Locales being the same, no translation is necessary.)

3.6 API for Superstructure Objects

This section describes the API for the VirtualUniverse, Locale, and HiResCoord objects.

3.6.1 VirtualUniverse Object

The VirtualUniverse object consists of a set of Locale objects.

Constructors

The VirtualUniverse object has the following constructors.

```
public VirtualUniverse()
```

This constructs a new VirtualUniverse object. This VirtualUniverse can then be used to create Locale objects.

Methods

The VirtualUniverse object has the following methods.

```
public final Enumeration getAllLocales()
public final int numLocales()
```

The first method returns the Enumeration object of all Locales in this virtual universe. The `numLocales` method returns the number of Locales.

3.6.2 Locale Object

The Locale object consists of a point, specified using high-resolution coordinates, and a set of subgraphs, rooted by BranchGroup node objects.

Constructors

The Locale object has the following constructors.

```
public Locale(VirtualUniverse universe)
public Locale(VirtualUniverse universe, int x[], int y[], int z[])
public Locale(VirtualUniverse universe, HiResCoord hiRes)
```

These three constructors create a new high-resolution Locale object in the specified VirtualUniverse. The first form constructs a Locale object located at (0.0, 0.0, 0.0). The other two forms construct a Locale object using the specified high-resolution coordinates. In the second form, the parameters `x`, `y`, and `z` are arrays of eight 32-bit integers that specify the respective high-resolution coordinate.

Methods

The Locale object has the following methods. For the Locale picking methods, see Section 10.3.2, "BranchGroup Node and Locale Node Pick Methods."

```
public VirtualUniverse getVirtualUniverse()
```

This method retrieves the virtual universe within which this Locale object is contained.

```
public void setHiRes(int x[], int y[], int z[])
public void setHiRes(HiResCoord hiRes)
public void getHiRes(HiResCoord hiRes)
```

These methods set or get the high-resolution coordinates of this Locale.

```
public void addBranchGraph(BranchGroup branchGroup)
public void removeBranchGraph(BranchGroup branchGroup)
public void replaceBranchGraph(BranchGroup oldGroup,
       BranchGroup newGroup)
public int numBranchGraphs()
public Enumeration getAllBranchGraphs()
```

The first three methods add, remove, and replace a branch graph in this Locale. Adding a branch graph has the effect of making the branch graph "live." The fourth method retrieves the number of branch graphs in this Locale. The last method retrieves an Enumeration object of all branch graphs.

3.6.3 HiResCoord Object

A HiResCoord object defines a point using a set of three high-resolution coordinates, each of which consists of three two's-complement fixed-point numbers. Each high-resolution number consists of 256 total bits with a binary point at bit 128. Java 3D uses integer arrays of length eight to define or extract a single 256-

bit coordinate value. Java 3D interprets the integer at index 0 as the 32 most significant bits and the integer at index 7 as the 32 least significant bits.

Constructors

The HiResCoord object has the following constructors.

```
public HiResCoord(int x[], int y[], int z[])
public HiResCoord(HiResCoord hc)
public HiResCoord()
```

The first constructor generates the high-resolution coordinate point from three integer arrays of length eight. The integer arrays specify the coordinate values corresponding with their name. The second constructor creates a new high-resolution coordinate point by cloning the high-resolution coordinates `hc`. The third constructor creates new high-resolution coordinates with value (0.0, 0.0, 0.0).

Methods

```
public void setHiResCoord(int x[], int y[], int z[])
public void setHiResCoord(HiResCoord hiRes)
public void setHiResCoordX(int x[])
public void setHiResCoordY(int y[])
public void setHiResCoordZ(int z[])
```

These five methods modify the value of high-resolution coordinates `this`. The first method resets all three coordinate values with the values specified by the three integer arrays. The second method sets the value of `this` to that of high-resolution coordinates `hiRes`. The third, fourth, and fifth methods reset the corresponding coordinate of `this`.

```
public void getHiResCoord(int x[], int y[], int z[])
public void getHiResCoord(HiResCoord hc)
public void getHiResCoordX(int x[])
public void getHiResCoordY(int y[])
public void getHiResCoordZ(int z[])
```

These five methods retrieve the value of the high-resolution coordinates `this`. The first method retrieves the high-resolution coordinates' values and places those values into the three integer arrays specified. All three arrays must have length greater than or equal to eight. The second method updates the value of the high-resolution coordinates `hc` to match the value of `this`. The third, fourth, and fifth methods retrieve the coordinate value that corresponds to their name and update the integer array specified, which must be of length eight or greater.

`public void add(HiResCoord h1, HiResCoord h2)`
`public void sub(HiResCoord h1, HiResCoord h2)`

These two methods perform arithmetic operations on high-resolution coordinates. The first method adds `h1` to `h2` and stores the result in `this`. The second method subtracts `h2` from `h1` and stores the result in `this`.

`public void scale(int scale, HiResCoord h1)`
`public void scale(int scale)`

These methods scale a high-resolution coordinate point. The first method scales `h1` by the scalar value `scale` and places the scaled coordinates into `this`. The second method scales `this` by the scalar value `scale` and places the scaled coordinates back into `this`.

`public void negate(HiResCoord h1)`
`public void negate()`

These two methods negate a high-resolution coordinate point. The first method negates `h1` and stores the result in `this`. The second method negates `this` and stores its negated value back into `this`.

`public void difference(HiResCoord h1, Vector3d v)`

This method subtracts `h1` from `this` and stores the resulting difference vector in the double-precision floating-point vector `v`. Note that although the individual high-resolution coordinate points cannot be represented accurately by double-precision numbers, this difference vector between them *can* be accurately represented by doubles for many practical purposes, such as viewing.

`public boolean equals(HiResCoord h1)`

This method performs an arithmetic comparison between `this` and `h1`. It returns `true` if the two high-resolution coordinate points are equal; otherwise, it returns `false`.

`public double distance(HiResCoord h1)`

This method computes the linear distance between high-resolution coordinate points `this` and `h1`, and returns this value expressed as a double. Note that although the individual high-resolution coordinate points cannot be represented accurately by double precision numbers, this distance between them *can* be accurately represented by a double for many practical purposes.

C H A P T E R 4

Group Node Objects

GROUP nodes are the glue elements used in constructing a scene graph. The following subsections list the seven group nodes (see Figure 4-1) and their definitions. All group nodes can have a variable number of child node objects—including other group nodes as well as leaf nodes. These children have an associated index that allows operations to specify a particular child. However, unless one of the special ordered group nodes is used, the Java 3D renderer can choose to render a group node's children in whatever order it wishes (including rendering the children in parallel).

```
SceneGraphObject
    Node
        Group
            BranchGroup
            OrderedGroup
                DecalGroup
            SharedGroup
            Switch
            TransformGroup
```

Figure 4-1 Group Node Hierarchy

4.1 Group Node

The Group node object is a general-purpose grouping node. Group nodes have exactly one parent and an arbitrary number of children that are rendered in an unspecified order (or in parallel). Operations on Group node objects include adding, removing, and enumerating the children of the Group node. The subclasses of Group node add additional semantics.

Constants

```
public static final int ALLOW_CHILDREN_READ
public static final int ALLOW_CHILDREN_WRITE
public static final int ALLOW_CHILDREN_EXTEND
```

These flags, when enabled using the `setCapability` method, specify that this Group node will allow the following methods, respectively:

- `numChildren`, `getChild`, `getAllChildren`
- `setChild`, `insertChild`, `removeChild`
- `addChild`, `moveTo`

These capability bits are enforced only when the node is part of a live or compiled scene graph.

```
public static final int ALLOW_COLLISION_BOUNDS_READ
public static final int ALLOW_COLLISION_BOUNDS_WRITE
```

These flags, when enabled using the `setCapability` method, specify that this Group node will allow reading and writing of its collision bounds.

Constructors

```
public Group()
```

Constructs and initializes a Group node object.

Methods

The Group node class defines the following methods.

```
public final int numChildren()
public final Node getChild(int index)
```

The first method returns a count of the number of children. The second method returns the child at the specified index.

```
public final void setChild(Node child, int index)
public final void insertChild(Node child, int index)
public final void removeChild(int index)
```

The first method replaces the child at the specified index with a new child. The second method inserts a new child before the child at the specified index. The third method removes the child at the specified index. Note that if this Group node is part of a live or compiled scene graph, only BranchGroup nodes may be added to or removed from it—and only if the appropriate capability bits are set.

public final Enumeration getAllChildren()

This method returns an Enumeration object of all children.

public final void addChild(Node child)

This method adds a new child as the last child in the group. Note that if this Group node is part of a live or compiled scene graph, only BranchGroup nodes may be added to it—and only if the appropriate capability bits are set.

public Node cloneNode(boolean forceDuplicate)

This method creates a new instance of the node. This routine is called by `cloneTree` to duplicate the current node.

public void duplicateNode(Node originalNode, boolean forceDuplicate)

This method copies all the node information from the `originalNode` into the current node. This method is called from the `cloneNode` method, which is in turn called by the `cloneTree` method.

For each NodeComponent object contained by the object being duplicated, the NodeComponent's `duplicateOnCloneTree` flag is used to determine whether the NodeComponent should be duplicated in the new node or if just a reference to the current node should be placed in the new node. This flag can be overridden by setting the `forceDuplicate` parameter in the `cloneTree` method to `true`.

public final void moveTo(BranchGroup branchGroup)

This method moves the specified BranchGroup node from its old location in the scene graph to the end of this group, in an atomic manner. Functionally, this method is equivalent to the following lines:

```
branchGroup.detach();
this.addChild(branchGroup);
```

If either this Group or the specified BranchGroup is part of a live or compiled scene graph, the appropriate capability bits must be set in the affected nodes.

`public final Bounds setCollisionBounds(Bounds bounds)`
`public final Bounds getCollisionBounds()`

These methods set and retrieve the collision bounding object for a node.

`public final void setAlternateCollisionTarget(boolean target)`
`public final boolean getAlternateCollisionTarget()`

The set method causes this Group node to be reported as the collision target when collision is being used and this node or any of its children is in a collision. The default is false. This method tries to set the capability bit `Node.ENABLE_COLLISION_REPORTING`. The get method returns the collision target state.

For collision with USE_GEOMETRY set, the collision traverser will check the geometry of all the Group node's leaf descendants. For collision with USE_BOUNDS set, the collision traverser will check the bounds at this Group node. In both cases, if there is a collision, this Group node will be reported as the colliding object in the SceneGraphPath.

4.2 BranchGroup Node

A BranchGroup is the root of a subgraph of a scene that may be compiled as a unit, attached to a virtual universe, or included as a child of a group node in another subgraph. A subgraph, rooted by a BranchGroup node, can be thought of as a compile unit. The following things may be done with BranchGroup.

- A BranchGroup may be compiled by calling its `compile` method. This causes the entire subgraph to be compiled. If any BranchGroup nodes are contained within the subgraph, they are compiled as well (along with their descendants).
- A BranchGroup may be inserted into a virtual universe by attaching it to a Locale. The entire subgraph is then said to be *live*.
- A BranchGroup that is contained within another subgraph may be reparented or detached at run time if the appropriate capabilities are set. See Figure 4-2.

Note that if a BranchGroup is included in another subgraph, as a child of some other group node, it may not be attached to a Locale.

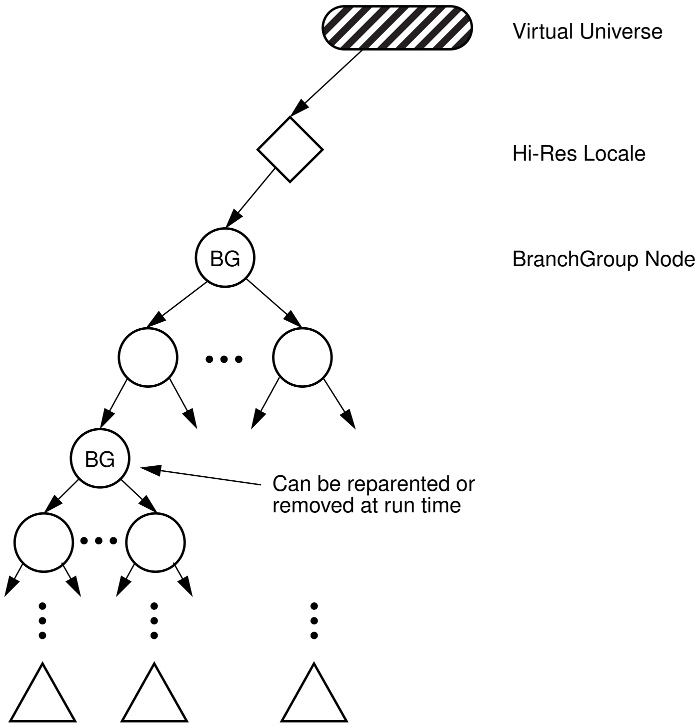


Figure 4-2 Altering the Scene Graph at Run Time

Constants

The BranchGroup class adds the following new constant.

`public static final int ALLOW_DETACH`

This flag, when enabled using the `setCapability` method, allows this BranchGroup node to be detached from its parent group node. This capability flag is enforced only when the node is part of a live or compiled scene graph.

Methods

The BranchGroup class defines the following methods.

`public final void compile()`

This method compiles the scene graph rooted at this BranchGroup and creates and caches a newly compiled scene graph.

```
public final void detach()
```

This method detaches the BranchGroup node from its parent.

```
public Node cloneNode(boolean forceDuplicate)
```

This method creates a new instance of the node. This routine is called by `cloneTree` to duplicate the current node.

```
public void duplicateNode(Node originalNode,
       boolean forceDuplicate)
```

This method copies all the node information from the `originalNode` into the current node. This method is called from the `cloneNode` method, which is in turn called by the `cloneTree` method.

For each NodeComponent object contained by the object being duplicated, the NodeComponent's `duplicateOnCloneTree` value is used to determine whether the NodeComponent should be duplicated in the new node or if just a reference to the current node should be placed in the new node. This flag can be overridden by setting the `forceDuplicate` parameter in the `cloneTree` method to `true`.

4.3 TransformGroup Node

The TransformGroup node specifies a single spatial transformation—via a Transform3D object (see Section 7.1.27, "Transform3D Object")—that can position, orient, and scale all of its children.

The specified transformation must be affine. Further, if the TransformGroup node is used as an ancestor of a ViewPlatform node in the scene graph, then the transformation must be congruent—only rotations, translations, and uniform scales are allowed in a direct path from a Locale to a ViewPlatform node. A `BadTransformException` (see Section D.1, "BadTransformException") is thrown if an attempt is made to specify an illegal transform.

Note: Even though arbitrary affine transformations are allowed, better performance will result if all matrices within a branch graph are congruent—containing only rotations, translation, and *uniform* scale.

The effects of transformations in the scene graph are cumulative. The concatenation of the transformations of each TransformGroup in a direct path from the Locale to a Leaf node defines a composite model transformation (CMT) that takes points in that Leaf node's local coordinates and transforms them into Virtual World (Vworld) coordinates. This composite transformation is used to transform points, normals, and distances into Vworld coordinates. Points are transformed by the CMT. Normals are transformed by the inverse-transpose of the CMT. Distances are transformed by the scale of the CMT. In the case of a transformation containing a nonuniform scale or shear, the maximum scale value in any direction is used. This ensures, for example, that a transformed bounding sphere, which is specified as a point and a radius, continues to enclose all objects that are also transformed using a nonuniform scale.

Constants

The TransformGroup class adds the following new flags.

```
public static final int ALLOW_TRANSFORM_READ
public static final int ALLOW_TRANSFORM_WRITE
```

These flags, when enabled using the `setCapability` method, allow this node's Transform3D to be read or written. They are only used when the node is part of a live or compiled scene graph.

Constructors

```
public TransformGroup()
public TransformGroup(Transform3D t1)
```

These construct and initialize a new TransformGroup. The first form initializes the node's Transform3D to the identity transformation; the second form initializes the node's Transform3D to a copy of the specified transform.

Methods

The TransformGroup class defines the following methods.

```
public final void setTransform(Transform3D t1)
public final void getTransform(Transform3D t1)
```

These methods retrieve or set this node's attached Transform3D object by copying the transform to or from the specified object.

```
public Node cloneNode(boolean forceDuplicate)
public void duplicateNode(Node originalNode,
       boolean forceDuplicate)
```

The first method creates a new instance of the node. This method is called by `cloneTree` to duplicate the current node. The second method copies all the node information from the `originalNode` into the current node. This method is called from the `cloneNode` method, which is in turn called by the `cloneTree` method.

For each NodeComponent object contained by the object being duplicated, the NodeComponent's `duplicateOnCloneTree` flag is used to determine whether the NodeComponent should be duplicated in the new node or a reference to the current node should be placed in the new node. This flag can be overridden by setting the `forceDuplicate` parameter in the `cloneTree` method to `true`.

4.4 OrderedGroup Node

The OrderedGroup node guarantees that Java 3D will render its children in their index order. Only the OrderedGroup node and its subclasses make any use of the order of their children during rendering.

Methods

`public Node cloneNode(boolean forceDuplicate)`

This method creates a new instance of the node. This routine is called by `cloneTree` to duplicate the current node.

```
public void duplicateNode(Node originalNode,
       boolean forceDuplicate)
```

This method copies all the node information from the `originalNode` into the current node. This method is called from the `cloneNode` method, which is in turn called by the `cloneTree` method.

For each NodeComponent object contained by the object being duplicated, the NodeComponent's `duplicateOnCloneTree` value is used to determine whether the NodeComponent should be duplicated in the new node or if just a reference to the current node should be placed in the new node. This flag can be overridden by setting the `forceDuplicate` parameter in the `cloneTree` method to `true`.

4.5 DecalGroup Node

The DecalGroup node is a subclass of the OrderedGroup node. The DecalGroup node is an ordered group node used for defining decal geometry on top of other geometry. The DecalGroup node specifies that its children should be rendered in index order and that they generate coplanar objects. Examples of this include painted decals or text on surfaces and a checkerboard layered on top of a table.

The first child, at index 0, defines the surface on top of which all other children are rendered. The geometry of this child must encompass all other children; otherwise, incorrect rendering may result. The polygons contained within each of the children must be facing the same way. If the polygons defined by the first child are front facing, then all other surfaces should be front facing. In this case, the polygons are rendered in order. The renderer can use knowledge of the coplanar nature of the surfaces to avoid *Z*-buffer collisions (for example, if the underlying implementation supports stenciling or polygon offset, then these techniques may be employed). If the main surface is back facing, then all other surfaces should be back facing and need not be rendered (even if back-face culling is disabled).

Note that using the DecalGroup node does not guarantee that *Z*-buffer collisions are avoided. An implementation of Java 3D may fall back to treating DecalGroup node as an ordinary OrderedGroup node.

Methods

`public Node cloneNode(boolean forceDuplicate)`

This method creates a new instance of the node. This routine is called by `cloneTree` to duplicate the current node.

```
public void duplicateNode(Node originalNode,
       boolean forceDuplicate)
```

This method copies all the node information from the `originalNode` into the current node. This method is called from the `cloneNode` method, which is in turn called by the `cloneTree` method.

For each NodeComponent object contained by the object being duplicated, the NodeComponent's `duplicateOnCloneTree` value is used to determine whether the NodeComponent should be duplicated in the new node or if just a reference to the current node should be placed in the new node. This flag can be overridden by setting the `forceDuplicate` parameter in the `cloneTree` method to `true`.

4.6 Switch Node

The Switch group node allows a Java 3D application to choose dynamically among a number of subgraphs. The Switch node contains an ordered list of children and a switch value. The switch value determines which child or children Java 3D will render. Note that the index order of children is only used for selecting the appropriate child or children—it does not specify rendering order.

Constants

```
public static final int ALLOW_SWITCH_READ
public static final int ALLOW_SWITCH_WRITE
```

These flags, when enabled using the `setCapability` method, allow reading and writing of the values that specify the child-selection criteria. They are only used when the node is part of a live or compiled scene graph.

```
public static final int CHILD_NONE
public static final int CHILD_ALL
public static final int CHILD_MASK
```

These values, when used in place of a non-negative integer index value, indicate which children of the Switch node are selected for rendering. A value of `CHILD_NONE` indicates that no children are rendered. A value of `CHILD_ALL` indicates that all children are rendered, effectively making this Switch node operate as an ordinary Group node. A value of `CHILD_MASK` indicates that the `childMask` BitSet is used to select the children that are rendered.

Constructors

```
public Switch()
public Switch(int whichChild)
public Switch(int whichChild, BitSet childMask)
```

These constructors initialize a new Switch node using the specified parameters. The default values for those parameters not specified are as follows:

whichChild: CHILD_NONE
childMask: empty

Methods

The Switch node class defines the following methods.

```
public final void setWhichChild(int whichChild)
public final int getWhichChild()
```

These methods access or modify the index of the child that the Switch object will draw. The value may be a non-negative integer, indicating a specific child, or it may be one of the following constants: `CHILD_NONE`, `CHILD_ALL`, or `CHILD_MASK`. If the specified value is out of range, then no children are drawn.

```
public final void setChildMask(BitSet childMask)
public final BitSet getChildMask()
```

These methods access or modify the mask used to select the children that the Switch object will draw when the `whichChild` parameter is `CHILD_MASK`. This parameter is ignored during rendering if the `whichChild` parameter is a value other than `CHILD_MASK`.

```
public final Node currentChild()
```

This method returns the currently selected child. If `whichChild` is out of range, or is set to `CHILD_MASK`, `CHILD_ALL`, or `CHILD_NONE`, then `null` is returned.

```
public Node cloneNode(boolean forceDuplicate)
```

This method creates a new instance of the node. This routine is called by `cloneTree` to duplicate the current node.

```
public void duplicateNode(Node originalNode,
       boolean forceDuplicate)
```

This method copies all the node information from the `originalNode` into the current node. This method is called from the `cloneNode` method, which is in turn called by the `cloneTree` method.

For each NodeComponent object contained by the object being duplicated, the NodeComponent's `duplicateOnCloneTree` value is used to determine whether the NodeComponent should be duplicated in the new node or if just a reference to the current node should be placed in the new node. This flag can be overridden by setting the `forceDuplicate` parameter in the `cloneTree` method to `true`.

4.7 SharedGroup Node

A SharedGroup node provides a mechanism for sharing the same subgraph in different parts of the tree via a Link node. See Section 6.1.1, "SharedGroup Node," for a description of this node.

CHAPTER 5

Leaf Node Objects

LEAF nodes define atomic entities such as geometry, lights, and sounds. The leaf nodes and their associated meanings follow.

5.1 Leaf Node

The Leaf node is an abstract class for all scene graph nodes that have no children. Leaf nodes specify lights, geometry, and sounds; provide special linking and instancing capabilities for sharing scene graphs; and provide a view platform for positioning and orienting a view in the virtual world. Figure 5-1 shows the Leaf node object hierarchy.

Constructors

`public Leaf()`

Constructs and initializes a new Leaf object.

Methods

The Leaf node object defines the following methods.

```
public void updateNodeReferences(NodeReferenceTable
       referenceTable)
```

This method is called by the `cloneTree` method (see Section 6.2, "Cloning Subgraphs") after all nodes in the subgraph have been cloned. The user can query the NodeReferenceTable object to determine if any nodes that the Leaf node references have been duplicated by the `cloneTree` call and, if so, what the corresponding Node is in the new subgraph. If a user extends a predefined Java 3D object and adds a reference to another node, this method must be defined in order to ensure proper operation of the `cloneTree` method. The first statement in the user's `updateNodeReferences` method must be `super.updateNodeReferences(referenceTable)`. For predefined Java 3D nodes, this method will be implemented automatically.

The NodeReferenceTable object is passed to the `updateNodeReferences` method and allows references from the old subgraph to be translated into references in the cloned subgraph. See Section 6.2.5, "NodeReferenceTable Object," for more details.

`public Node cloneTree(boolean forceDuplicate)`

This method duplicates all nodes of the specified subgraph. For group nodes, the node is first duplicated via a call to `cloneNode` and then `cloneTree` is called for each child node. For leaf nodes, component data can either be duplicated or be made a reference to the original data. Leaf node `cloneTree` behavior is determined by the `duplicateOnCloneTree` flag found in every leaf node's component data class and by the `forceDuplicate` parameter.

- SceneGraphObject
  - Node
    - Leaf
      - Background
      - Behavior
        - *Predefined behaviors*
      - BoundingLeaf
      - Clip
      - Fog
        - ExponentialFog
        - LinearFog
      - Light
        - AmbientLight
        - DirectionalLight
        - PointLight
          - SpotLight
      - Link
      - Morph
      - Shape3D
      - Sound
        - BackgroundSound
        - PointSound
          - ConeSound
      - Soundscape
      - ViewPlatform

Figure 5-1 Leaf Node Hierarchy

5.2 Shape3D Node

The Shape3D leaf node object specifies all geometric objects. It contains two components: a reference to the shape's geometry and its appearance component. The Geometry object defines the shape's geometric data. The Appearance object specifies that object's appearance attributes, including color, material, texture, and so on. See Chapter 7, "Node Component Objects" for details of the Geometry and Appearance objects.

Constants

The Shape3D node object defines the following flags.

```
public static final int ALLOW_GEOMETRY_READ
public static final int ALLOW_GEOMETRY_WRITE
public static final int ALLOW_APPEARANCE_READ
public static final int ALLOW_APPEARANCE_WRITE
public static final int ALLOW_COLLISION_BOUNDS_WRITE
public static final int ALLOW_COLLISION_BOUNDS_READ
```

These flags, when enabled using the `setCapability` method, allow reading and writing of the Geometry and Appearance component objects and the collision bounds, respectively. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The Shape3D node object defines the following constructors.

```
public Shape3D(Geometry geometry, Appearance appearance)
public Shape3D(Geometry geometry)
public Shape3D()
```

The first form constructs and initializes a new Shape3D object with the specified geometry and appearance components. The second form uses the specified geometry and a `null` appearance component. The third form uses both a `null` geometry component and a `null` appearance component. If the geometry component is `null`, then no geometry is drawn. If the appearance component is `null`, then default values are used for all appearance attributes.

Methods

The Shape3D node object defines the following methods.

public final void setGeometry(Geometry geometry)
public final Geometry getGeometry()

These methods access or modify the Geometry component object associated with this Shape3D node.

public final void setAppearance(Appearance appearance)
public final Appearance getAppearance()

These methods access or modify the Appearance component object associated with this Shape3D node. Setting it to `null` results in default attribute use.

public final void setCollisionBounds(Bounds bounds)
public final Bounds getCollisionBounds()

These methods set and retrieve the collision bounds for this node.

public Node cloneNode(boolean forceDuplicate)

This method creates a new instance of the node. This routine is called by `cloneTree` to duplicate the current node. `cloneNode` should be overridden by any user-subclassed objects. All subclasses must have their `cloneNode` method consist of the following lines:

```
public Node cloneNode(boolean forceDuplicate) {
   UserSubClass usc = new UserSubClass();
   usc.duplicateNode(this, forceDuplicate);
   return usc;
}
```

public void duplicateNode(Node originalNode,
boolean forceDuplicate)

This method copies all the node information from the `originalNode` into the current node. This method is called from the `cloneNode` method, which is in turn called by the `cloneTree` method.

For each NodeComponent object contained by the object being duplicated, the NodeComponent's `duplicateOnCloneTree` flag is used to determine whether the NodeComponent should be duplicated in the new node or if just a reference to the current node should be placed in the new node. This flag can be overridden by setting the `forceDuplicate` parameter in the `cloneTree` method to `true`.

```
public void updateNodeReferences(NodeReferenceTable
        referenceTable)
```

This method is called by the `cloneTree` method (see Section 6.2, "Cloning Subgraphs") after all nodes in the subgraph have been cloned. The user can query the NodeReferenceTable object to determine if any nodes that the leaf node references have been duplicated by the `cloneTree` call and, if so, what the corresponding node is in the new subgraph. If a user extends a predefined Java 3D object and adds a reference to another node, this method must be defined in order to ensure proper operation of the `cloneTree` method. The first statement in the user's `updateNodeReferences` method must be `super.updateNodeReferences(referenceTable)`. For predefined Java 3D nodes, this method will be implemented automatically.

The NodeReferenceTable object is passed to the `updateNodeReferences` method and allows references from the old subgraph to be translated into references in the cloned subgraph. See Section 6.2.5, "NodeReferenceTable Object," for more details.

5.3 BoundingLeaf Node

The BoundingLeaf node defines a bounding region object that can be referenced by other leaf nodes to define a region of influence (Fog and Light nodes), an activation region (Background, Clip, and Soundscape nodes), or a scheduling region (Sound and Behavior nodes). The bounding region is defined in the local coordinate system of the BoundingLeaf node. A reference to a BoundingLeaf node can be used in place of a locally defined bounds object for any of the aforementioned regions.

This allows an application to specify a bounding region in one coordinate system (the local coordinate system of the BoundingLeaf node) other than the local coordinate system of the node that references the bounds. For an example of how this might be used, consider a closed room with a number of track lights. Each light can move independent of the other lights and, as such, needs its own local coordinate system. However, the bounding volume is used by all the lights in the boundary of the room, which doesn't move when the lights move. In this example, the BoundingLeaf node allows the bounding region to be defined in the local coordinate system of the room, rather than in the local coordinate system of a particular light. All lights can then share this single bounding volume.

Constants

The BoundingLeaf node object defines the following flags.

```
public static final int ALLOW_REGION_READ
public static final int ALLOW_REGION_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the bounding region object.

Constructors

The BoundingLeaf node object defines the following constructors.

```
public BoundingLeaf()
Public BoundingLeaf(Bounds region)
```

The first form constructs a BoundingLeaf node with a unit sphere region object. The second form constructs a BoundingLeaf node with the specified bounding region.

Methods

```
public final void setRegion(Bounds region)
public final Bounds getRegion()
```

These methods set and retrieve the BoundingLeaf node's bounding region.

5.4 Background Node

The Background leaf node defines either a solid background color or a background image that is used to fill the window at the beginning of each new frame. It also specifies an application region in which this Background node is active. A Background node is active when its application region intersects the ViewPlatform's activation volume. If multiple Background nodes are active, the Background node that is "closest" to the eye will be used. If no Background nodes are active, then the window is cleared to black.

Constants

The Background node object defines the following flags.

```
public static final int ALLOW_APPLICATION_BOUNDS_READ
public static final int ALLOW_APPLICATION_BOUNDS_WRITE
public static final int ALLOW_IMAGE_READ
public static final int ALLOW_IMAGE_WRITE
public static final int ALLOW_COLOR_READ
```

public static final int ALLOW_COLOR_WRITE
public static final int ALLOW_GEOMETRY_READ
public static final int ALLOW_GEOMETRY_WRITE

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the application region, the image, the color, and the background geometry. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The Background node object defines the following constructors.

public Background()
public Background(Color3f color)
public Background(float r, float g, float b)
public Background(ImageComponent2D image)

The first form constructs a Background leaf node with a default color of black (0.0, 0.0, 0.0). The next two forms construct a Background leaf node with the specified color. The final form constructs a Background leaf node with the specified 2D image.

Methods

The Background node object defines the following methods.

public final void getColor(Color3f color)
public final void setColor(Color3f color)
public final void setColor(float r, float g, float b)

These three methods access or modify the background color.

public final ImageComponent2D getImage()
public final void setImage(ImageComponent2D image)

These two methods access or modify the background image. If the image is not `null` then it is used in place of the color.

public final void setGeometry(BranchGroup branch)
public final BranchGroup getGeometry()

These two methods access or modify the Background geometry. The `setGeometry` method sets the background geometry to the specified BranchGroup node. If non-null, this background geometry is drawn on top of the background color or image using a projection matrix that essentially puts the geometry at infinity. The geometry should be pretessellated onto a unit sphere.

```
public final void setApplicationBounds(Bounds region)
public final Bounds getApplicationBounds()
```

These two methods access or modify the Background node's application bounds. This bounds is used as the application region when the application bounding leaf is set to `null`. The `getApplicationBounds` method returns a copy of the associated bounds.

```
public final void setApplicationBoundingLeaf(BoundingLeaf region)
public final BoundingLeaf getApplicationBoundingLeaf()
```

These two methods access or modify the Background node's application bounding leaf. When set to a value other than `null`, this bounding leaf overrides the application bounds object and is used as the application region.

5.5 Clip Node

The Clip leaf node defines the far clipping plane used to clip objects in the virtual universe. It also specifies an application region in which this Clip node is active. A Clip node is active when its application region intersects the ViewPlatform's activation volume. If multiple Clip nodes are active, the Clip node that is "closest" to the eye will be used. The back distance value specified by this Clip node overrides the value specified in the View object. If no Clip nodes are active, then the back clip distance is used from the View object.

Constants

```
public static final int ALLOW_APPLICATION_BOUNDS_READ
public static final int ALLOW_APPLICATION_BOUNDS_WRITE
public static final int ALLOW_BACK_DISTANCE_READ
public static final int ALLOW_BACK_DISTANCE_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the application region and the back distance. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The Clip node object defines the following constructors.

```
public Clip(double backDistance)
public Clip()
```

The first constructor constructs a Clip leaf node with the rear clip plane at the specified distance, in the local coordinate system, from the eye. The second constructor constructs a Clip leaf node with a default back clipping distance.

Methods

The Clip node object defines the following methods.

```
public final void setBackDistance(double backDistance)
public final double getBackDistance()
```

These methods access or modify the back clipping distances in the Clip node. This distance specifies the back clipping plane in the local coordinate system of the node.

```
public final void setApplicationBounds(Bounds region)
public final Bounds getApplicationBounds()
```

These two methods access or modify the Clip node's application bounds. This bounds is used as the application region when the application bounding leaf is set to `null`. The `getApplicationBounds` method returns a copy of the associated bounds.

```
public final void setApplicationBoundingLeaf(BoundingLeaf region)
public final BoundingLeaf getApplicationBoundingLeaf()
```

These two methods access or modify the Clip node's application bounding leaf. When set to a value other than `null`, this bounding leaf overrides the application bounds object and is used as the application region.

5.6 Fog Node

The Fog leaf node is an abstract class that defines a common set of attributes that control fog, or depth cueing, in the scene. The Fog node includes a parameter that specifies the fog color and a Bounds object that specifies the region of influence for the Fog node.

Objects whose bounding volumes intersect the Fog node's region of influence have fog applied to their color after lighting and texturing have been applied. The Fog node also contains a list of Group nodes that indicates the hierarchical scope of this fog. If the list of scoping nodes is empty, the fog has *universe scope* and will apply to all nodes in the virtual universe that are within the Fog node's region of influence.

If the regions of influence of multiple Fog nodes overlap, the Java 3D system will choose a single set of fog parameters for those objects that lie in the intersection. This is done in an implementation-dependent manner, but in general, the Fog node that is "closest" to the object is chosen.

Constants

The Fog node object defines the following flags.

```
public static final int ALLOW_INFLUENCING_BOUNDS_READ
public static final int ALLOW_INFLUENCING_BOUNDS_WRITE
public static final int ALLOW_COLOR_READ
public static final int ALLOW_COLOR_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read the region of influence, write the region of influence, read color, and write color. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The Fog node object defines the following constructors.

```
public Fog()
public Fog(float r, float g, float b)
public Fog(Color3f color)
```

These constructors each construct a new Fog node. The first constructor uses default values for all parameters. The remaining constructors use the specified parameters and use defaults for those parameters not specified. Default values are as follows:

color: black (0,0,0)
list of scoping nodes: empty
influencingRegion: empty

Methods

The Fog node object defines the following methods.

```
public final void setColor(float r, float g, float b)
public final void setColor(Color3f color)
public final void getColor(Color3f color)
```

These three methods access or modify the Fog node's color. An application will typically set this to the same value as the background color.

```
public final void setInfluencingBounds(Bounds region)
public final Bounds getInfluencingBounds()
```

These methods access or modify the Fog node's influencing bounds. This bounds is used as the region of influence when the influencing bounding leaf is set to `null`. The Fog node operates on all objects that intersect its region of influence. The `getInfluencingBounds` method returns a copy of the associated bounds.

```
public final void setInfluencingBoundingLeaf(BoundingLeaf region)
public final BoundingLeaf getInfluencingBoundingLeaf()
```

These methods access or modify the Fog node's influencing bounding leaf. When set to a value other than `null`, this overrides the influencing bounds object and is used as the region of influence.

```
public final void setScope(Group scope, int index)
public final Group getScope(int index)
public final void addScope(Group scope)
public final void insertScope(Group scope, int index)
public final void removeScope(int index)
public final int numScopes()
public final Enumeration getAllScopes()
```

These methods access or modify the Fog node's hierarchical scope. By default, Fog nodes are scoped only by their regions of influence. These methods allow them to be further scoped by a Group node in the hierarchy. The hierarchical scoping of a Fog node cannot be accessed or modified if the node is part of a live or compiled scene graph.

5.6.1 ExponentialFog Node

The ExponentialFog leaf node extends the Fog leaf node by adding a fog density that is used as the exponent of the fog equation. For more information on the fog equation, see Appendix E, "Equations."

The density is defined in the local coordinate system of the node, but the actual fog equation will ideally take place in eye coordinates.

Constants

The ExponentialFog node object defines the following flags.

```
public static final int ALLOW_DENSITY_READ
public static final int ALLOW_DENSITY_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the density values. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The ExponentialFog node object defines the following constructors.

```
public ExponentialFog()
public ExponentialFog(float r, float g, float b)
public ExponentialFog(Color3f color)
public ExponentialFog(float r, float g, float b, float density)
public ExponentialFog(Color3f color, float density)
```

Each of these constructors creates a new ExponentialFog node. The first constructor uses default values for all parameters. The remaining constructors use the specified parameters and use defaults for those parameters not specified. Default values are as follows:

density: 1.0

Methods

The ExponentialFog node object defines the following methods.

```
public final void setDensity(float density)
public final float getDensity()
```

These two methods access or modify the density in the ExponentialFog object.

5.6.2 LinearFog Node

The LinearFog leaf node extends the Fog leaf node by adding a pair of distance values, in Z, at which fog should start obscuring the scene and should maximally obscure the scene.

The front and back fog distances are defined in the local coordinate system of the node, but the actual fog equation will ideally take place in eye coordinates. For more information on the fog equation, see Appendix E, “Equations.”

Constants

The LinearFog node object defines the following flags.

```
public static final int ALLOW_DISTANCE_READ
public static final int ALLOW_DISTANCE_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the distance values. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The LinearFog node object defines the following constructors.

```
public LinearFog()
public LinearFog(float r, float g, float b)
public LinearFog(Color3f color)
public LinearFog(float r, float g, float b, double frontDistance,
       double backDistance)
public LinearFog(Color3f color, double frontDistance,
       double backDistance)
```

These constructors each construct a new LinearFog node. The first constructor uses default values for all parameters. The remaining constructors use the specified parameters and use defaults for those parameters not specified. Default values are as follows:

front distance: 0.1
back distance: 1.0

Methods

The LinearFog node object defines the following methods.

```
public final void setFrontDistance(float frontDistance)
public final float getFrontDistance()
public final void setBackDistance(float backDistance)
public final float getBackDistance()
```

These four methods access or modify the front and back distances in the LinearFog object. The front distance is the distance at which the fog starts obscuring objects. The back distance is the distance at which the fog fully obscures objects. Objects drawn closer than the front fog distance are not affected by fog. Objects drawn farther than the back fog distance are drawn entirely in the fog color.

5.7 Light Node

The Light leaf node is an abstract class that defines the properties common to all Light nodes. A light has associated with it a color, a state (whether it is on or off), and a Bounds object that specifies the region of influence for the light. Objects whose bounding volumes intersect the Light node's region of influence are lit by this light. The Light node also contains a Group node that indicates the hierarchical scope of this light. If no scoping node is specified, then the light has *universe scope* and applies to all nodes in the virtual universe that are within the light's region of influence.

The Java 3D lighting model is based on a subset of the OpenGL lighting model.

Constants

The Light node object defines the following flags.

```
public static final int ALLOW_INFLUENCING_BOUNDS_READ
public static final int ALLOW_INFLUENCING_BOUNDS_WRITE
public static final int ALLOW_STATE_READ
public static final int ALLOW_STATE_WRITE
public static final int ALLOW_COLOR_READ
public static final int ALLOW_COLOR_WRITE
```

These flags, when enabled using the `setCapability` method, allow reading and writing of the region of influence, the state, and the color, respectively. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The Light node object defines the following constructors.

```
public Light()
public Light(Color3f color)
public Light(boolean lightOn, Color3f color)
```

These three constructors construct and initialize a light.

Methods

The Light node object defines the following methods.

```
public final void setEnable(boolean state)
public final boolean getEnable()
```

These methods access or modify the state of this light (that is, whether the light is enabled).

```
public final void setColor(Color3f color)
public final void getColor(Color3f color)
```

These methods access or modify the current color of this light.

```
public final setInfluencingBounds(Bounds region)
public final Bounds getInfluencingBounds()
```

These methods access or modify the Light node's influencing bounds. This bounds is used as the region of influence when the influencing bounding leaf is set to `null`. The Light node operates on all objects that intersect its region of influence. The `getInfluencingBounds` method returns a copy of the associated bounds.

```
public final setInfluencingBoundingLeaf(BoundingLeaf region)
public final BoundingLeaf getInfluencingBoundingLeaf()
```

These methods access or modify the Light node's influencing bounding leaf. When set to a value other than `null`, this overrides the influencing bounds object and is used as the region of influence.

```
public final void setScope(Group scope, int index)
public final Group getScope(int index)
public final void addScope(Group scope)
public final void insertScope(Group scope, int index)
public final void removeScope(int index)
public final int numScopes()
public final Enumeration getAllScopes()
```

These methods access or modify the Light node's hierarchical scope. By default, Light nodes are scoped only by their regions of influence bounds. These methods allow them to be further scoped by a node in the hierarchy.

5.7.1 AmbientLight Node

An AmbientLight node defines an ambient light source. It has the same attributes as the abstract Light node.

Constructors

The AmbientLight node defines the following constructors.

```
public AmbientLight()
public AmbientLight(Color3f color)
public AmbientLight(boolean lightOn, Color3f color)
```

The first constructor constructs and initializes a new AmbientLight node using default parameters. The next two constructors construct and initialize a new AmbientLight node using the specified parameters. The `color` parameter is the color of the light source. The `lightOn` flag indicates whether this light is on or off.

5.7.2 DirectionalLight Node

A DirectionalLight node defines an oriented light with an origin at infinity. It has the same attributes as a Light node, with the addition of a direction vector to specify the direction in which it shines.

Constants

The DirectionalLight node object defines the following flags.

```
public static final int ALLOW_DIRECTION_READ
public static final int ALLOW_DIRECTION_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read or write the associated direction. These capability flags are enforced only when the node is part of a live or compiled scene graph.

The DirectionalLight's direction vector is defined in the local coordinate system of the node.

Constructors

The DirectionalLight node object defines the following constructors.

```
public DirectionalLight()
```

Constructs and initializes a directional light. The default direction of the light is toward the screen, along the negative *z* axis.

```
public DirectionalLight(Color3f color, Vector3f direction)
public DirectionalLight(boolean LightOn, Color3f color,
       Vector3f direction)
```

These constructors construct and initialize a directional light with the parameters provided.

Methods

The DirectionalLight node object defines the following methods.

```
public final void setDirection(Vector3f direction)
public final void setDirection(float x, float y, float z)
public final void getDirection(Vector3f direction)
```

These methods access or modify the light's current direction.

5.7.3 PointLight Node

A PointLight node defines a point light source located at some point in space and radiating light in all directions (also known as a *positional light*). It has the same attributes as a Light node, with the addition of location and attenuation parameters.

The PointLight's position is defined in the local coordinate system of the node.

Constants

The PointLight node object defines the following flags.

```
public static final int ALLOW_POSITION_READ
public static final int ALLOW_POSITION_WRITE
public static final int ALLOW_ATTENUATION_READ
public static final int ALLOW_ATTENUATION_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read position, write position, read attenuation parameters, and write attenuation parameters. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The PointLight Node defines the following constructors.

```
public PointLight()
```

Constructs and initializes a point light source with the default position at 0.0, 0.0, 0.0.

```
public PointLight(Color3f color, Point3f position,
       Point3f attenuation)
public PointLight(boolean lightOn, Color3f color,
       Point3f position, Point3f attenuation)
```

These constructors construct and initialize a point light with the specified parameters.

Methods

The PointLight node object defines the following methods.

```
public final void setPosition(Point3f position)
public final void setPosition(float x, float y, float z)
public final void getPosition(Point3f position)
```

These methods access or modify the point light's current position.

```
public final void setAttenuation(Point3f attenuation)
public final void setAttenuation(float constant, float linear,
       float quadratic)
public final void getAttenuation(Point3f attenuation)
```

These methods access or modify the point light's current attenuation. The values presented to the methods specify the coefficients of the attenuation polynomial, with `constant` providing the constant term, `linear` providing the linear coefficient, and `quadratic` providing the quadratic coefficient.

5.7.4 SpotLight Node

A SpotLight node defines a point light source located at some point in space and radiating in a specific direction. It has the same attributes as a PointLight node, with the addition of a direction of radiation, a spread angle to specify its limits, and a concentration factor that specifies how quickly the light intensity attenuates as a function of the angle of radiation as measured from the direction of radiation.

Constants

The SpotLight node object defines the following flags.

```
public static final int ALLOW_SPREAD_ANGLE_READ
public static final int ALLOW_SPREAD_ANGLE_WRITE
public static final int ALLOW_CONCENTRATION_READ
public static final int ALLOW_CONCENTRATION_WRITE
public static final int ALLOW_DIRECTION_READ
public static final int ALLOW_DIRECTION_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write spread angle, concentration, and direction. These capability flags are enforced only when the node is part of a live or compiled scene graph.

The SpotLight's direction vector and spread angle are defined in the local coordinate system of the node.

Constructors

The SpotLight node object defines the following constructors.

```
public SpotLight()
```

Constructs and initializes a new spotlight with the default values.

```
public SpotLight(Color3f color, Point3f position,
       Point3f attenuation, Vector3f direction, float spreadAngle,
       float concentration)
public SpotLight(boolean lightOn, Color3f color, Point3f position,
       Point3f attenuation, Vector3f direction, float spreadAngle,
       float concentration)
```

These construct and initialize a new spotlight with the parameters specified.

Methods

The SpotLight node object defines the following methods.

```
public final void setSpreadAngle(float spreadAngle)
public final float getSpreadAngle()
```

These methods access or modify the spread angle, in radians, of this spotlight.

```
public final void setConcentration(float concentration)
public final float getConcentration()
```

These methods access or modify the concentration of this spotlight.

```
public final void setDirection(float x, float y, float z)
public final void setDirection(Vector3f direction)
public final void getDirection(Vector3f direction)
```

These methods access or modify the direction of this spotlight.

5.8 Sound Node

The Sound leaf node is an abstract class that defines the properties common to all Sound nodes. A scene graph can contain multiple sounds. Each Sound node contains a reference to the sound data, an amplitude scale factor, a release flag denoting that the sound associated with this node is to play to the end when the sound is disabled, the number of times the sound is to be repeated, a state (whether the sound is on or off), a scheduling region, a priority, and a flag denoting if the sound is to continue playing “silently” even while it is inactive. Whenever the listener is within the Sound node’s scheduling bounds, the sound is potentially audible.

Constants

The Sound object contains the following flags.

```
public static final int ALLOW_SOUND_DATA_READ
public static final int ALLOW_SOUND_DATA_WRITE
public static final int ALLOW_INITIAL_GAIN_READ
public static final int ALLOW_INITIAL_GAIN_WRITE
public static final int ALLOW_LOOP_READ
public static final int ALLOW_LOOP_WRITE
public static final int ALLOW_RELEASE_READ
public static final int ALLOW_RELEASE_WRITE
public static final int ALLOW_CONT_PLAY_READ
public static final int ALLOW_CONT_PLAY_WRITE
public static final int ALLOW_ENABLE_READ
public static final int ALLOW_ENABLE_WRITE
public static final int ALLOW_SCHEDULING_BOUNDS_READ
public static final int ALLOW_SCHEDULING_BOUNDS_WRITE
public static final int ALLOW_PRIORITY_READ
public static final int ALLOW_PRIORITY_WRITE
```

```
public static final int ALLOW_DURATION_READ
public static final int ALLOW_CHANNELS_USED_READ
public static final int ALLOW_IS_PLAYING_READ
public static final int ALLOW_IS_READY_READ
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the sound data, the initial gain information, the loop information, the release flag, the continuous play flag, the sound on/off switch, the scheduling region, the prioritization value, the duration information, and the sound playing information. These capability flags are enforced only when the node is part of a live or compiled scene graph.

```
public static final float NO_FILTER
```

This constant defines a floating point value that denotes that no filter value is set. Filters are described in Section 5.8.3, "ConeSound Node."

```
public static final int DURATION_UNKNOWN
```

This constant denotes that the sound's duration could not be calculated. A fallback for `getDuration` of a non-cached sound.

Constructors

The Sound node object defines the following constructors.

```
public Sound()
```

Constructs and initializes a new Sound node object that includes the following defaults for its fields:

sound data: null
initial gain: 1.0
loop: 0
release flag: false
continuous flag: false
on switch: false
scheduling region: null (cannot be scheduled)
priority: 1.0

```
public Sound(MediaContainer soundData, float initialGain)
```

Constructs and initializes a new Sound node object using the provided data and gain parameter values, and defaults for all other fields. This constructor implicitly loads the sound data associated with this node if the implementation uses sound caching.

```
public Sound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, boolean continuous,
       boolean enable, Bounds region, float priority)
```

Constructs and initializes a new Sound node object using the provided parameter values.

Methods

The Sound node object defines the following methods.

```
public final void setSoundData(MediaContainer soundData)
public final MediaContainer getSoundData()
```

These methods provide a way to associate different types of audio data with a Sound node. This data can be cached (buffered) or noncached (unbuffered or streaming). If the AudioDevice has been attached to the PhysicalEnvironment, the sound data is made ready to begin playing. Certain functionality cannot be applied to true sreaming sound data: sound duration is unknown, looping is disabled, and the sound cannot be restarted. Furthermore, depending on the implementation of the AudioDevice used, streaming, non-cached data may not be fully spatialized.

```
public final void setInitialGain(float amplitude)
public final float getInitialGain()
```

This gain is a scale factor that is applied to the sound data associated with this sound source to increase or decrease its overall amplitude.

```
public final void setLoop(int loopCount)
public final int getLoop()
```

Data for nonstreaming sound (such as a sound sample) can contain two loop points marking a section of the data that is to be looped a specific number of times. Thus, sound data can be divided into three segments: the *attack* (before the begin loop point), the *sustain* (between the begin and end loop points), and the *release* (after the end loop point). If there are no loop begin and end points defined as part of the sound data (say for Java Media Player types that do not contain sound samples), then the begin loop point is set at the beginning of the sound data, and the end loop point at the end of the sound data. If this is the case, looping the sound means repeating the whole sound. However, these begin and end loop points can be placed anywhere within the sound data, allowing a portion in the middle of the sound to be looped.

A sound can be looped a specified number of times after it is activated and before it is completed. The loop count value explicitly sets the number of times the sound is looped. Any non-negative number is a valid value. A value of 0 denotes that the looped section is not repeated, but is played only once. A value of –1 denotes that the loop is repeated indefinitely.

Changing the loop count of a sound after the sound has been started will not dynamically affect the loop count currently used by the sound playing. The new loop count will be used the next time the sound is enabled.

```
public final void setReleaseEnable(boolean state)
public final boolean getReleaseEnable()
```

When a sound is disabled, its playback would normally stop immediately no matter what part of the sound data was currently being played. By setting the Release flag to `true` for nodes with nonstreaming sound data, the sound is allowed to play from its current position in the sound data to the end of the data (without repeats), thus playing the release portion of the sound before stopping.

```
public final void setContinuousEnable(boolean state)
public final boolean getContinuousEnable()
```

For some applications, it's useful to turn a sound source "off" but to continue "silently" playing the sound so that when it is turned back "on" the sound picks up playing in the same location (over time) as it would have been if the sound had never been disabled (turned off). Setting the continuous flag to `true` causes the sound renderer to keep track of where (over time) the sound would be playing even when the sound is disabled.

```
public final setSchedulingBounds(Bounds region)
public final Bounds getSchedulingBounds()
```

These two methods access or modify the Sound node's scheduling bounds. This bounds is used as the scheduling region when the scheduling bounding leaf is set to `null`. A sound is scheduled for activation when its scheduling region intersects the ViewPlatform's activation volume. The `getSchedulingBounds` method returns a copy of the associated bounds.

```
public final void setSchedulingBoundingLeaf(BoundingLeaf region)
public final BoundingLeaf getSchedulingBoundingLeaf()
```

These two methods access or modify the Sound node's scheduling bounding leaf. When set to a value other than `null`, this bounding leaf overrides the scheduling bounds object and is used as the scheduling region.

```
public final void setPriority(float ranking)
public final float getPriority()
```

These methods access or modify the Sound node's priority, which is used to rank concurrently playing sounds in order of importance during playback. When more sounds are started than the AudioDevice can handle, the Sound node with the lowest priority ranking is deactivated. If a sound is deactivated (due to a sound with a higher priority being started), it is automatically reactivated when resources become available (for example, when a sound with a higher priority finishes playing) or when the ordering of sound nodes is changed due to a change in a Sound node's priority.

Sounds with a lower priority than a sound that cannot be played due to a lack of channels will be played. For example, assume we have eight channels available for playing sounds. After ordering four sounds, we begin playing them in order, checking if the number of channels required to play a given sound are actually available before the sound is played. Furthermore, say the first sound needs three channels to play, the second sound needs four channels, the third sound needs three channels and the fourth sound needs only one channel. The first and seconds sounds can be started because they require seven of the eight available channels. The third sound cannot be audibly started because it requires three channels and only one is still available. Consequently, the third sound starts playing "silently." The fourth sound can and will be started since it only requires one channel. The third sound will be made audible when three channels become available (i.e., when the first or second sound is finished playing).

Sounds given the same priority are ordered randomly. If the application wants a specific ordering it must assign unique priorities to each sound.

Methods to determine what audio output resources are required for playback of a Sound node on a particular AudioDevice and to determine the currently available audio output resources are described in Chapter 11, "Audio Devices."

```
public final void setEnable(boolean state)
public final boolean getEnable()
```

These two methods access or modify the playing state of this sound (that is, whether the sound is enabled). When enabled, the sound source is started and thus can potentially be heard, depending on its activation state, gain control parameters, continuation state, and spatialization parameters. If the continuous state is `true` and the sound is not active, enabling the sound starts the sound silently "playing" so that when the sound is activated, the sound is (potentially) heard from somewhere in the middle of the sound data. The activation state can change from active to inactive any number of times without stopping or starting the sound. To restart a sound at the beginning of its data, re-enable the sound by calling `setEnable` with a value of `true`.

Setting the enable flag to `true` during construction will act as a request to start the sound playing "as soon as it can" be started. This could be close to immediately in limited cases, but several conditions, detailed below, must be meet for a sound to be ready to be played.

`public final boolean isReady()`

This method retrieves the sound's "ready" status denoting that the sound is fully prepared for playing (either audibly or silently) to begin. Sound data associated with a Sound node, either during construction (when the MediaContainer is passed into the constructor as a parameter) or by calling `setSoundData()`, it can be prepared to begin playing only after the following conditions are satisfied:

- The Sound node has non-null sound data associated with it
- The Sound node is live
- There is an active View in the Universe
- There is an initialized AudioDevice associated with the PhysicalEnvironment.

Depending on the type of MediaContainer the sound data is and on the implementation of the AudioDevice used, sound data preparation could consist of opening, attaching, loading, or copying into memory the associated sound data. The query method, `isReady()`) returns `true` when the sound is fully preprocessed so that it is playable (audibly if active, silently if not active).

`public final boolean isPlaying()`

A sound source will not be heard unless it is both enabled (turned on) and activated. While these two conditions are meet, the sound is potentially audible and the method `isPlaying()` will return a status of `true`.

When the sound finishes playing its sound data (including all loops), it is implicitly disabled.

`public final boolean isPlayingSilently()`

This method returns the sound's silent status. If a sound is enabled before it is activated it is begun playing silently. If a sound is enabled then deactivated while playing it continues playing silently. In both of these cases `isPlaying()` returns `false` but the method `isPlayingSilently()` returns `true`.

```
public final long getDuration()
```

This method returns the length of time (in milliseconds) that the sound media associated with the sound source could run (including the number of times its loop section is repeated) if it plays to completion. If the sound media type is streaming, or if the sound is looped indefinitely, then a value of –1 (implying infinite length) is returned.

```
public final int getNumberOfChannelsUsed()
```

When a sound is started it could use more than one channel on the selected AudioDevice it is to be played on. This method returns the number of channels (on the executing audio device) being used by this sound. The method returns 0 if sound is not playing.

5.8.1 BackgroundSound Node

A BackgroundSound node defines an unattenuated, nonspatialized sound source that has no position or direction. It has the same attributes as a Sound node. This type of sound is simply added to the sound mix without modification and is useful for playing a mono or stereo music track, or an ambient sound effect. Unlike a Background (visual) node, more than one BackgroundSound node can be simultaneously enabled and active.

Constructors

The BackgroundSound node specifies the following constructor.

```
public BackgroundSound()
```

Constructs a BackgroundSound node object using the default parameters for Sound nodes.

```
public BackgroundSound(MediaContainer soundData,
       float initialGain)
public BackgroundSound(MediaContainer soundData,
       float initialGain, int loopCount, boolean release,
       boolean continuous, boolean enable, Bounds region,
       float priority)
```

The first constructor constructs a new BackgroundSound node using only the provided parameter values for the sound data and initial gain. The second constructor uses the provided parameter values for the sound data, initial gain, the number of times the loop is looped, a flag denoting whether the sound data is played to the end, a flag denoting whether the sound plays silently when disabled, whether sound is switched on or off, the sound activation region, and a priority value denoting the playback priority ranking.

5.8.2 PointSound Node

The PointSound node defines a spatially located sound whose waves radiate uniformly in all directions from some point in space. It has the same attributes as a Sound object, with the addition of a location and the specification of distance-based gain attenuation for listener positions between an array of distances.

The sound's amplitude is attenuated based on the distance between the listener and the sound source position. A piecewise linear curve (defined in terms of pairs consisting of a distance and a gain scale factor) specifies the gain scale factor slope.

The PointSound's location and attenuation distances are defined in the local coordinate system of the node.

Constants

The PointSound object contains the following flags.

```
public static final int ALLOW_POSITION_READ
public static final int ALLOW_POSITION_WRITE
public static final int ALLOW_DISTANCE_GAIN_READ
public static final int ALLOW_DISTANCE_GAIN_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the position and the distance gain array. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The PointSound node object defines the following constructors.

```
public PointSound()
```

Constructs a PointSound node object that includes the defaults for a Sound object plus the following defaults for its own fields:

Position vector: (0.0, 0.0, 0.0)
Distance gain attenuation: null (no attenuation performed)

```
public PointSound(MediaContainer soundData, float initialGain,
       Point3f position)
public PointSound(MediaContainer soundData, float initialGain,
       float posX, float posY, float posZ)
```

Both of these constructors construct a PointSound node object using only the provided parameter values for sound data, sample gain, and position. The remaining fields are set to the default values specified earlier. The first form uses vectors as input for its position. The second form uses individual float parameters for the elements of the position vector.

```
public PointSound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, boolean continuous,
       boolean enable, Bounds region, float priority,
       Point3f position, Point2f distanceGain[])
public PointSound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, boolean continuous,
       boolean enable, Bounds region, float priority, float posX,
       float posY, float posZ, Point2f distanceGain[])
public PointSound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, boolean continuous,
       boolean enable, Bounds region, float priority,
       Point3f position, float attenuationDistance[],
       float attenuationGain[])
public PointSound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, boolean continuous,
       boolean enable, Bounds region, float priority, float posX,
       float posY, float posZ, float attenuationDistance[],
       float attenuationGain[])
```

These four constructors construct a PointSound node object using the provided parameter values. The first and third forms use points as input for the position. The second and fourth forms use individual float parameters for the elements of the position. The first and second forms accept an array of Point2f for the distance attenuation values where each pair in the array contains a distance and a gain scale factor. The third and fourth forms accept separate arrays for the components of distance attenuation, namely, the distance and gain scale factors. See the description for the `setDistanceGain` method, below, for details on how the separate arrays are interpreted.

Methods

The PointSound node object defines the following methods.

```
public final void setPosition(Point3f position)
public final void setPosition(float x, float y, float z)
public final void getPosition(Point3f position)
```

These methods set and retrieve the position in 3D space from which the sound radiates.

```
public final void setDistanceGain(Point2f attenuation[])
public final void setDistanceGain(float distance[], float gain[])
public final int getDistanceGainLength()
public final void getDistanceGain(Point2f attenuation[])
public final void getDistanceGain(float distance[], float gain[])
```

These methods set and retrieve the sound's distance attenuation. If this is not set, no distance gain attenuation is performed (equivalent to using a gain scale factor of 1.0 for all distances). See Figure 5-2. Gain scale factors are associated with distances from the listener to the sound source via an array of distance and gain scale factor pairs. The gain scale factor applied to the sound source is determined by finding the range of values `distance[i]` and `distance[i+1]` that includes the current distance from the listener to the sound source, then linearly interpolating the corresponding values `gain[i]` and `gain[i+1]` by the same amount.

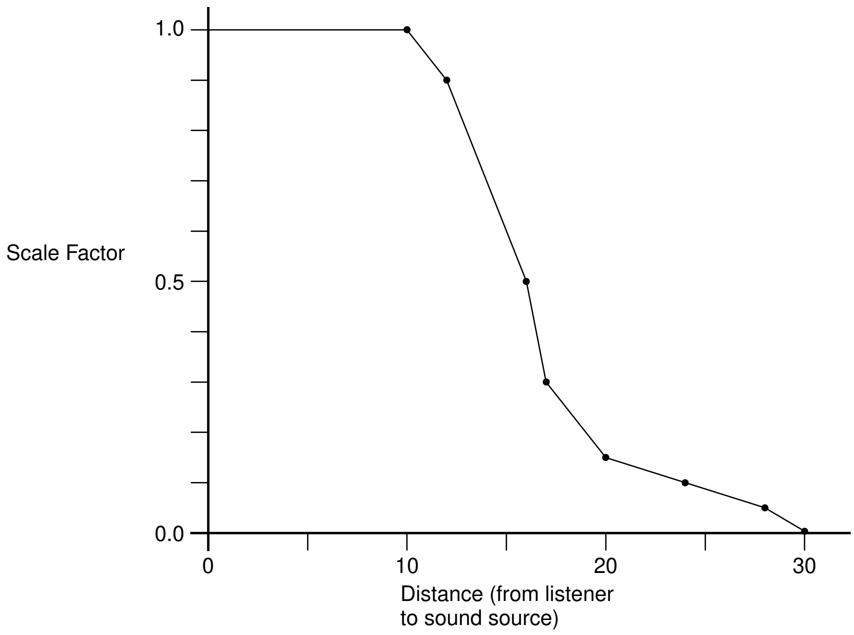


Figure 5-2 PointSound Distance Gain Attenuation

If the distance from the listener to the sound source is less than the first distance in the array, the first gain scale factor is applied to the sound source. This creates a spherical region around the listener within which all sound gain is uniformly scaled by the first gain in the array.

If the distance from the listener to the sound source is greater than the last distance in the array, the last gain scale factor is applied to the sound source.

The first form of `setDistanceGain` takes these pairs of values as an array of Point2f. The second form accepts two separate arrays for these values. The `distance` and `gainScale` arrays should be of the same length. If the `gainScale` array length is greater than the `distance` array length, the `gainScale` array elements beyond the length of the `distance` array are ignored. If the `gainScale` array is shorter than the `distance` array, the last `gainScale` array value is repeated to fill an array of length equal to `distance` array.

There are two methods for `getDistanceGain`, one returning an array of points, the other returning separate arrays for each attenuation component.

Distance elements in this array of Point2f are a monotonically increasing set of floating-point numbers measured from the location of the sound source. Gain scale factor elements in this list of pairs can be any positive floating-point numbers. While for most applications this list of gain scale factors will usually be monotonically decreasing, they do not have to be.

Figure 5-2 shows a graphical representation of a distance gain attenuation list. The values given for distance/gain pairs would be

((10.0, 1.0), (12.0, 0.9), (16.0, 0.5), (17.0, 0.3),
(20.0, 0.16), (24.0, 0.12), (28.0, 0.05), (30.0, 0.0))

Thus if the current distance from the listener to the sound source is 22 units, a scale factor of 0.14 would be applied to the sound amplitude. If the current distance from the listener to the sound source is less than 10 units, the scale factor of 1.0 would be applied to the sound amplitude. If the current distance from the listener to the sound source is greater than 30 units, the scale factor of 0.0 would be applied to the sound amplitude.

The `getDistanceGainLength` method returns the length of the distance gain attenuation arrays. Arrays passed into `getDistanceGain` methods should all be at least this size.

5.8.3 ConeSound Node

The ConeSound node object defines a PointSound node whose sound source is directed along a specific vector in space. A ConeSound source is attenuated by gain scale factors and filters based on the angle between the vector from the source to the listener, and the ConeSound's direction vector. This attenuation is either a single spherical distance gain attenuation (as for a general PointSound source) or dual front and back distance gain attenuations defining elliptical attenuation volumes. The angular filter and the active AuralAttribute component filter define what filtering is applied to the sound source.

This node has the same attributes as a PointSound node, with the addition of a direction vector and an array of points that each contain an angular distance (in radians), a gain scale factor, and a filter (which for now consists of a lowpass filter cutoff frequency). Similar to the definition of the distance gain array for PointSounds, a piecewise linear curve (defined in terms of radians from the axis) specifies the slope of these additional attenuation values.

Figure 5-3 shows an approximation of angular attenuation (disregarding distance attenuation).

Constants

The ConeSound object contains the following flags.

```
public static final int ALLOW_DIRECTION_READ
public static final int ALLOW_DIRECTION_WRITE
public static final int ALLOW_ANGULAR_ATTENUATION_READ
public static final int ALLOW_ANGULAR_ATTENUATION_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the direction and the angular attenuation array. These capability flags are enforced only when the node is part of a live or compiled scene graph.

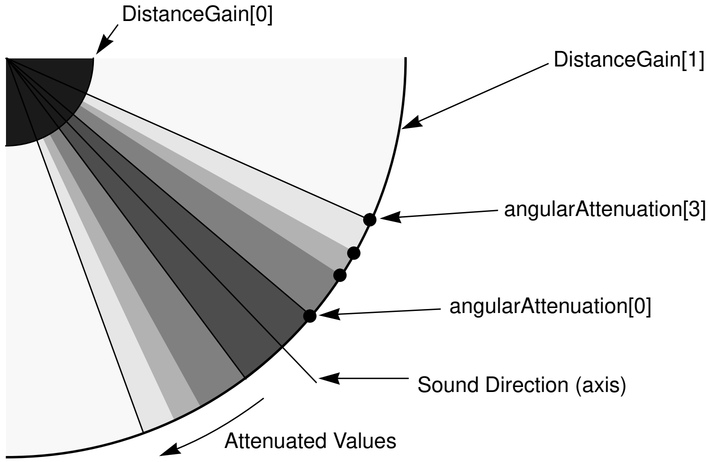


Figure 5-3 ConeSound

Constructors

The ConeSound node object defines the following constructors.

```
public ConeSound()
```

Constructs a ConeSound node object that includes the defaults for a PointSound object plus the following defaults for its own fields:

Direction vector: (0.0, 0.0, 1.0)
Back attenuation: null
Angular attenuation: ((0.0, 1.0), NO_FILTER,(π/2, 0.0, NO_FILTER))

```
public ConeSound(MediaContainer soundData, float initialGain,
       Point3f position, Vector3f direction)
public ConeSound(MediaContainer soundData, float initialGain,
       float posX, float posY, float posZ, float dirX, float dirY,
       float dirZ)
```

Both of these constructors construct a ConeSound node object using only the provided parameter values for sound, overall initial gain, position, and direction. The remaining fields are set to the default values listed earlier. The first form uses points as input for its position and direction. The second form uses individual float parameters for the elements of the position and direction vectors.

```
public ConeSound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, boolean continuous,
       boolean enable, Bounds region, float priority,
       Point3f position, Point2f frontDistanceAttenuation[],
       Point2f backDistanceAttenuation[], Vector3f direction)
public ConeSound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, boolean continuous,
       boolean enable, Bounds region, float priority, float posX,
       float posY, float posZ, float frontDistance[],
       float frontDistanceGain[], float backDistance[],
       float backDistanceGain[], float dirX, float dirY,
       float dirZ)
```

These constructors construct a ConeSound node object using the provided parameter values. The first form uses points or vectors as input for its position, direction, and front/back distance attenuation arrays. The second form uses individual float parameters for the elements of the position, direction, and two distance attenuation arrays.

Unlike the single distance gain attenuation array for PointSounds, which define spherical areas about the sound source between which gains are linearly interpolated, this directed ConeSound can have two distance gain attenuation arrays that define ellipsoidal attenuation areas. See the `setDistanceGain` PointSound method for details on how the separate `distance` and `distanceGain` arrays are interpreted.

The ConeSound's direction vector and angular measurements are defined in the local coordinate system of the node.

```
public ConeSound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, boolean continuous,
       boolean enable, Bounds region, float priority,
       Point3f position, Point2f distanceAttenuation[],
       Vector3f direction, Point3f angularAttenuation[])
public ConeSound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, boolean continuous,
       boolean enable, Bounds region, float priority, float posX,
       float posY, float posZ, float distance[],
       float distanceGain[], float dirX, float dirY, float dirZ,
       float angle[], float angularGain[],
       float frequencyCutoff[])
```

These constructors construct a ConeSound node object using the provided parameter values, which include a single spherical distance attenuation array. The first form uses points and vectors as input for its position, direction, single spherical `distanceAttenuation` array, and `angularAttenuation` array. The second form uses individual float parameters for the elements of the position, direction, `distanceAttenuation` array, and `angularAttenuation` array.

The first form accepts arrays of points for the distance attenuation and angular values. Each Point2f in the `distanceAttenuation` array contains a distance and a gain scale factor. Each Point3f in the `angularAttenuation` array contains an angular distance, a gain scale factor, and a filtering value (which is currently defined as a simple cutoff frequency).

The second form accepts separate arrays for the distance and gain scale factor components of distance attenuation, and separate arrays for the angular distance, angular gain, and filtering components of angular attenuation. See the `setDistanceGain` PointSound method for details on how the separate `distance` and `distanceGain` arrays are interpreted. See the `setAngularAttenuation` ConeSound method for details on how the separate `angularDistance`, `angularGain`, and `filter` arrays are interpreted.

```
public ConeSound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, boolean continuous,
       boolean enable, Bounds region, float priority,
       Point3f position, Point2f frontDistanceAttenuation[],
       Point2f backDistanceAttenuation[], Vector3f direction,
       Point3f angularAttenuation[])
public ConeSound(MediaContainer soundData, float initialGain,
       int loopCount, boolean release, float priority,
       boolean continuous, boolean enable, Bounds region,
       float posX, float posY, float posZ, float frontDistance[],
       float frontDistanceGain[], float backDistance[],
       float backDistanceGain[], float dirX, float dirY,
       float dirZ, float angle[], float angularGain[],
       float frequencyCutoff[])
```

These constructors construct a ConeSound node object using the provided parameter values, which include two distance attenuation arrays defining elliptical distance attenuation regions. The first form uses points and vectors as input for its position, direction, and attenuation arrays. The second form uses individual float parameters for these same elements.

These two constructors differ from the previous two constructors only in the definition of the two distinct front and back distance attenuation arrays. See the `setDistanceGain` ConeSound method for details on how the separate `distance` and `distanceGain` arrays are interpreted. See the `setAngularAttenuation` ConeSound method for details on how the separate `angularDistance`, `angularGain`, and `filter` arrays are interpreted.

Methods

The ConeSound node object defines the following methods.

```
public final void setDistanceGain(Point2f frontAttenuation[],
       Point2f backAttenuation[])
public final void setDistanceGain(float frontDistance[],
       float frontGain[], float backDistance[], float backGain[])

public final void setBackDistanceGain(Point2f attenuation[])
public final void setBackDistanceGain(float distance[],
       float gain[])
public final void getDistanceGain(Point2f frontAttenuation[],
       Point2f backAttenuation[])
public final void getDistanceGain(float frontDistance[],
       float frontGain[], float backDistance[], float backGain[])
```

These methods set and retrieve the ConeSound's two distance attenuation arrays. If these are not set, no distance gain attenuation is performed (equivalent to using a distance gain of 1.0 for all distances). If only one distance attenuation array is set, spherical attenuation is assumed (see Figure 5-4). If both a front and back distance attenuation are set, elliptical attenuation regions are defined (see Figure 5-5). Use the PointSound `setDistanceGain` method to set the front distance attenuation array separately from the back distance attenuation array.

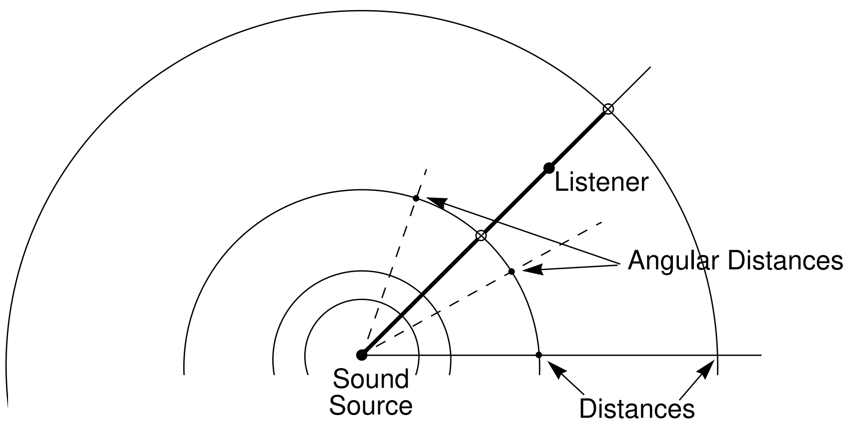


Figure 5-4 ConeSound with a Single Distance Gain Attenuation Array

A front distance attenuation array defines monotonically increasing distances from the sound source origin along the position direction vector. A back distance attenuation array (if given) defines monotonically increasing distances from the sound source origin along the negative direction vector. The two arrays must be of the same length. The `backDistance[i]` gain values must be less than or equal to `frontDistance[i]` gain values.

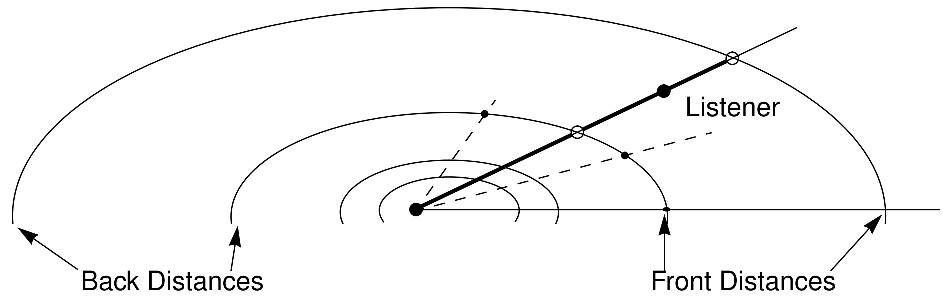


Figure 5-5 ConeSound with Two Distance Gain Attenuation Arrays

Gain scale factors are associated with distances from the listener to the sound source via an array of distance and gain scale factor pairs (see Figure 5-2). The gain scale factor applied to the sound source is the linear interpolated gain value within the distance value range that includes the current distance from the listener to the sound source.

The `getDistanceGainLength` method (defined in PointSound) returns the length of all distance gain attenuation arrays, including the back distance gain arrays. Arrays passed into getBackDistanceGain methods should all be at least this size.

```
public final void setDirection(Vector3f direction)
public final void setDirection(float x, float y, float z)
public final void getDirection(Vector3f direction)
```

This value is the sound source's direction vector. It is the axis from which angular distance is measured.

```
public final void setAngularAttenuation(Point2f attenuation[])
public final void setAngularAttenuation(Point3f attenuation[])
public final void setAngularAttenuation(float angle[],
       float angularGain[], float frequencyCutoff[])
public final int getAngularAttenuationLength()
public final void getAngularAttenuation(Point3f attenuation[])
public final void getAngularAttenuation(float angle[],
       float angularGain[], float frequencyCutoff[])
```

These methods set and retrieve the sound's angular gain and filter attenuation arrays. If these are not set, no angular gain attenuation or filtering is performed (equivalent to using an angular gain scale factor of 1.0 and an angular filter of `NO_FILTER` for all distances). This attenuation is defined as a triple of angular distance, gain scale factor, and filter values. The distance is measured as the angle in radians between the ConeSound's direction vector and the vector from the sound source position to the listener. Both the gain scale factor and filter applied to the sound source are the linear interpolation of values within the distance value range that includes the angular distance from the sound source axis.

If the angular distance from the listener-sound-position vector and the sound's direction vector is less than the first distance in the array, the first gain scale factor and first filter are applied to the sound source. This creates a conical region around the listener within which the sound is uniformly attenuated by the first gain and the first filter in the array.

If the distance from the listener-sound-position vector and the sound's direction vector is greater than the last distance in the array, the last gain scale factor and last filter are applied to the sound source.

Distance elements in this array of points are a monotonically increasing set of floating point numbers measured from 0 to π radians. Gain scale factor elements in this list of points can be any positive floating-point numbers. While for most applications this list of gain scale factors will usually be monotonically decreasing, they do not have to be. The filter (for now) is a single simple frequency cutoff value.

In the first form of `setAngularAttenuation`, only the angular distance and angular gain scale factor pairs are given. The filter values for these tuples are implicitly set to `NO_FILTER`. In the second form of `setAngularAttenuation`, an array of all three values is supplied.

The third form of `setAngularAttenuation` accepts three separate arrays for these angular attenuation values. These arrays should be of the same length. If the `angularGain` or filtering array length is greater than the `angularDistance` array length, the array elements beyond the length of the `angularDistance` array are ignored. If the `angularGain` or filtering array is shorter than the `angularDistance` array, the last value of the short array is repeated to fill an array of length equal to the `angularDistance` array.

The `getAngularAttenuationArrayLength` method returns the length of the angular attenuation arrays. Arrays passed into `getAngularAttenuation` methods should all be at least this size.

There are two methods for `getAngularAttenuation`, one returning an array of points, the other returning separate arrays for each attenuation component.

Figure 5-3 shows an example of an angular attenuation defining four points of the form (radiant distance, gain scale factor, cutoff filter frequency):

((0.12, 0.8, NO_FILTER), (0.26, 0.6, 18000.0), (0.32, 0.4, 15000.0), (0.40, 0.2, 11000.0))

5.9 Soundscape Node

The Soundscape leaf node defines the attributes that characterize the listener's aural environment. This node defines an application region and an associated aural attribute component object that controls reverberation and atmospheric properties that affect sound source rendering. (Aural attributes are described in Section 7.1.15, "AuralAttributes Object.") Multiple Soundscape nodes can be included in a single scene graph.

The Soundscape application region, different from a Sound node's scheduling region, is used to select which Soundscape (and thus which aural attribute object) is to be applied to the sounds being rendered. This selection is based on the position of the ViewPlatform (the "listener"), not the position of the sound.

It will be common for multiple Soundscape regions to be contained within a scene graph. Figure 5-6 shows application regions for two Soundscape nodes: a region with a large open area on the right, and a smaller, more constricted, less reverberant area on the left.

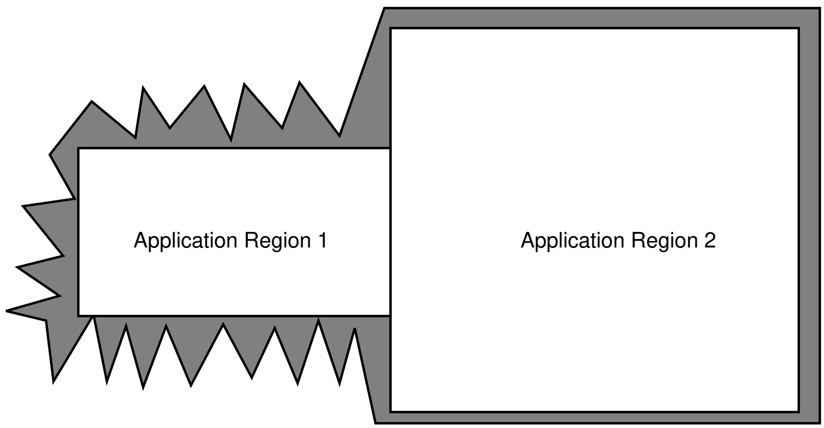


Figure 5-6 Multiple Soundscape Application Regions

The reverberation attributes for these two regions could be set to represent their physical differences so that active sounds are rendered differently depending on which region the listener is in.

Constants

The Soundscape node object defines the following flags.

`public static final int ALLOW_APPLICATION_BOUNDS_READ`
`public static final int ALLOW_APPLICATION_BOUNDS_WRITE`
`public static final int ALLOW_ATTRIBUTES_READ`
`public static final int ALLOW_ATTRIBUTES_WRITE`

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the application region and the aural attributes. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The Soundscape node object defines the following constructors.

`public Soundscape()`

Constructs a Soundscape node object that includes the following defaults for its elements:

> **application region**: null (no active region)
> **aural attributes**: null (uses default aural attributes)

`public Soundscape(Bounds region, AuralAttributes attributes)`

This method constructs a Soundscape node object using the specified application region and aural attributes.

Methods

The Soundscape node object defines the following methods.

`public final void setApplicationBounds(Bounds region)`
`public final Bounds getApplicationBounds()`

These two methods access or modify the Soundscape node's application bounds. This bounds is used as the application region when the application bounding leaf is set to `null`. The aural attributes associated with this Soundscape are used to render the active sounds when this application region intersects the ViewPlatform's activation volume. The `getApplicationBounds` method returns a copy of the associated bounds.

```
public final void setApplicationBoundingLeaf(BoundingLeaf region)
public final BoundingLeaf getApplicationBoundingLeaf()
```

These two methods access or modify the Soundscape node's application bounding leaf. When set to a value other than `null`, this bounding leaf overrides the application bounds object and is used as the application region.

```
public final void setAuralAttributes(AuralAttributes attributes)
public final AuralAttributes getAuralAttributes()
```

These two methods access or modify the aural attributes of this Soundscape. Setting it to `null` results in default attribute use.

5.10 ViewPlatform Node

The ViewPlatform node object defines a viewing platform that is referenced by a View object. The location, orientation, and scale of the composite transforms in the scene graph from the root to the ViewPlatform specify where the viewpoint is located and in which direction it is pointing. A viewer navigates through the virtual universe by changing the transform in the scene graph hierarchy above the ViewPlatform.

Constants

The ViewPlatform node object defines the following flags.

```
public static final int ALLOW_POLICY_READ
public static final int ALLOW_POLICY_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the view attach policy. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Methods

The ViewPlatform node object defines the following methods:

```
public final void setActivationRadius(float activationRadius)
public final float getActivationRadius()
```

The activation radius defines an activation volume surrounding the center of the ViewPlatform. This activation volume intersects with the scheduling regions and application regions of other leaf node objects to determine which of those objects may affect rendering.

Different leaf objects interact with the ViewPlatform's activation volume differently. The Background, Clip, and Soundscape leaf objects each define a set of attributes and an application region in which those attributes are applied. If more than one node of a given type (Background, Clip, or Soundscape) intersects the ViewPlatform's activation volume, the "most appropriate" node is selected.

Sound leaf objects begin playing their associated sounds when their scheduling region intersects a ViewPlatform's activation volume. Multiple sounds may be active at the same time.

Behavior objects act somewhat differently. Those Behavior objects with scheduling regions that intersect a ViewPlatform's activation volume become candidates for scheduling. Effectively, a ViewPlatform's activation volume becomes an additional qualifier on the scheduling of all Behavior objects. See Chapter 9, "Behaviors and Interpolators," for more details.

```
public final void setViewAttachPolicy(int policy)
public final int getViewAttachPolicy()
```

The view attach policy determines how Java 3D places the user's virtual eye point as a function of head position. See Section 8.4.3, "View Attach Policy," for details.

5.11 Behavior Node

The Behavior leaf node allows an application to manipulate a scene graph at run time. Behavior is an abstract class that defines properties common to all Behavior objects in Java 3D. There are several predefined behaviors that are subclasses of Behavior. Additionally, a Behavior leaf node may be subclassed by the user. Behaviors are described in Chapter 9, "Behaviors and Interpolators."

5.12 Morph Node

The Morph leaf node permits an application to morph between multiple GeometryArrays. The Morph node contains a single Appearance node, an array of GeometryArray objects, and an array of corresponding weights. The Morph node combines these GeometryArrays into an aggregate shape based on each GeometryArray's corresponding weight. Typically, Behavior nodes will modify the weights to achieve various morphing effects.

Constants

The Morph node specifies the following flags.

```
public static final int ALLOW_GEOMETRY_ARRAY_READ
public static final int ALLOW_GEOMETRY_ARRAY_WRITE
public static final int ALLOW_APPEARANCE_READ
public static final int ALLOW_APPEARANCE_WRITE
public static final int ALLOW_WEIGHTS_READ
public static final int ALLOW_WEIGHTS_WRITE
public static final int ALLOW_COLLISION_BOUNDS_READ
public static final int ALLOW_COLLISION_BOUNDS_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the GeometryArrays, appearance, weights, and collision Bounds components.

Constructors

The Morph node specifies the following constructors.

```
public Morph(GeometryArray geometryArrays[])
public Morph(GeometryArray geometryArrays[],
       Appearance appearance)
```

The first form constructs and initializes a new Morph leaf node with the specified array of GeometryArray objects and a `null` Appearance object. The second form uses the specified array of GeometryArray objects and the specified Appearance object. The length of the `geometryArrays` parameter determines the number of weighted geometry arrays in this Morph node. If `geometryArrays` is `null`, then a `NullPointerException` is thrown. If the Appearance component is `null`, then default values are used for all appearance attributes.

Methods

The Morph node specifies the following methods.

```
public final void setGeometryArrays(GeometryArray
       geometryArrays[])
```

This method sets the array of GeometryArray objects in the Morph node. Each GeometryArray component specifies colors, normals, and texture coordinates. The length of the `geometryArrays` parameter must be equal to the length of the array with which this Morph node was created; otherwise, an `Illegal-ArgumentException` is thrown.

public final GeometryArray getGeometryArray(int index)

This method retrieves a single geometry array from the Morph node. The `index` parameter specifies which array is returned.

public final void setAppearance(Appearance appearance)
public final Appearance getAppearance()

These methods set and retrieve the Appearance component of this Morph node. The Appearance component specifies material, texture, texture environment, transparency, or other rendering parameters. Setting it to `null` results in default attribute use.

public void setWeights(double weights[])
public double[] getWeights()

These methods set and retrieve the morph weight vector component of this Morph node. The Morph node "weights" the corresponding GeometryArray by the amount specified. The length of the `weights` parameter must be equal to the length of the array with which this Morph node was created; otherwise, an `IllegalArgumentException` is thrown.

public final void setCollisionBounds(Bounds bounds)
public final Bounds getCollisionBounds()

These methods set and retrieve the collision bounding object of this node.

5.13 Link Node

The Link leaf node allows an application to reference a shared subgroup, rooted by a SharedGroup node, from within a branch of the scene graph. Any number of Link nodes can refer to the same SharedGroup node. See Section 6.1.2, "Link Leaf Node," for a description of this node.

CHAPTER 6

Reusing Scene Graphs

JAVA 3D provides application programmers with two different means for reusing scene graphs. First, multiple scene graphs can share a common subgraph. Second, the node hierarchy of a common subgraph can be cloned, while still sharing large component objects such as geometry and texture objects. In the first case, changes in the shared subgraph affect all scene graphs that refer to the shared subgraph. In the second case, each instance is unique—a change in one instance does not affect any other instance.

6.1 Sharing Subgraphs

An application that wishes to share a subgraph from multiple places in a scene graph must do so through the use of the Link leaf node and an associated SharedGroup node. The SharedGroup node serves as the root of the shared subgraph. The Link leaf node refers to the SharedGroup node. It does not incorporate the shared scene graph directly into its scene graph.

6.1.1 SharedGroup Node

A SharedGroup node allows multiple Link leaf nodes to share its subgraph (see Figure 6-1) according to the following semantics:

- A SharedGroup may be referenced by one or more Link leaf nodes. Any runtime changes to a node or component object in this shared subgraph affect all graphs that refer to this subgraph.
- A SharedGroup may be compiled by calling its `compile` method prior to being referenced by any Link leaf nodes.
- Only Link leaf nodes may refer to SharedGroup nodes. A SharedGroup node cannot have parents or be attached to a Locale.

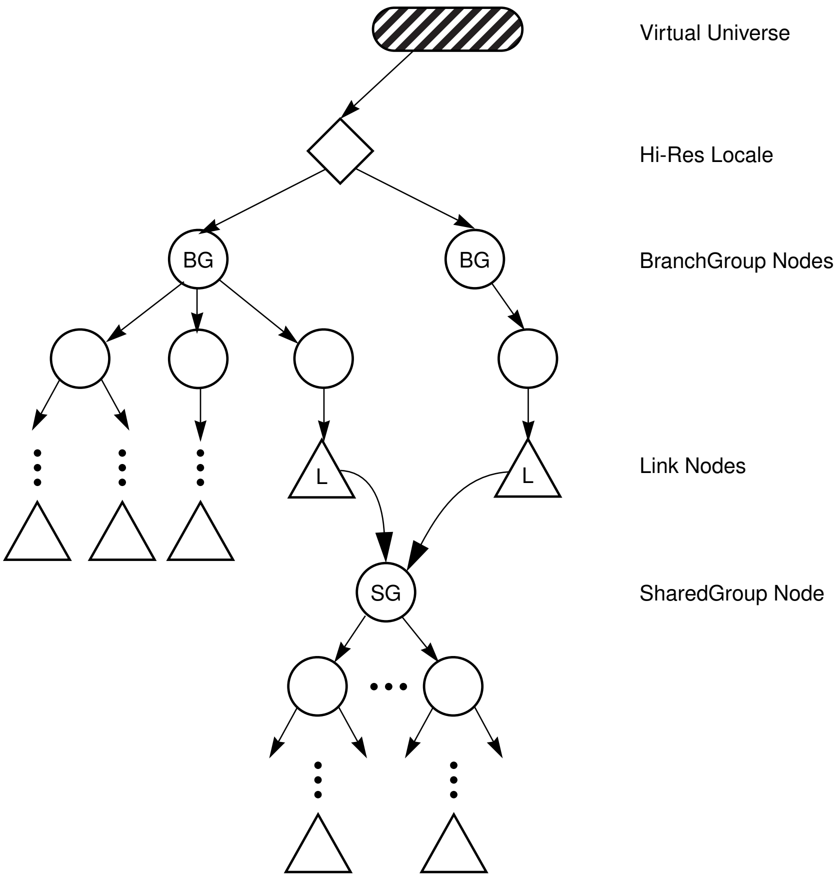


Figure 6-1 Sharing a Subgraph

A shared subgraph may contain any group node, except an embedded SharedGroup node (SharedGroup nodes cannot have parents). However, only the following leaf nodes may appear in a shared subgraph:

- Light
- Link
- Morph
- Shape
- Sound

An `IllegalSharingException` is thrown if any of the following leaf nodes appear in a shared subgraph:

- Background
- BoundingLeaf
- Behavior
- Clip
- Fog
- Soundscape
- ViewPlatform

Methods

The SharedGroup node defines the following methods.

```
public final void compile()
```

This method compiles the source SharedGroup associated with this object and creates and caches a newly compiled scene graph.

```
public Node cloneNode(boolean forceDuplicate)
```

This method creates a new instance of the node. This routine is called by `cloneTree` to duplicate the current node.

```
public void duplicateNode(Node originalNode,
       boolean forceDuplicate)
```

This method copies all the node information from the `originalNode` into the current node. This method is called from the `cloneNode` method, which is in turn called by the `cloneTree` method.

For each NodeComponent object contained by the object being duplicated, the NodeComponent's `duplicateOnCloneTree` value is used to determine whether the NodeComponent should be duplicated in the new node or if just a reference to the current node should be placed in the new node. This flag can be overridden by setting the `forceDuplicate` parameter in the `cloneTree` method to `true`.

6.1.2 Link Leaf Node

The Link leaf node allows an application to reference a shared graph, rooted by a SharedGroup node, from within a branch graph or another shared graph. See Figure 6-1. Any number of Link nodes can refer to the same SharedGroup node.

Constants

The Link node object defines two flags.

```
public static final int ALLOW_SHARED_GROUP_READ
public static final int ALLOW_SHARED_GROUP_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write the SharedGroup node pointed to by this Link node. These capability flags are enforced only when the node is part of a live or compiled scene graph.

Constructors

The Link node object defines two constructors.

```
public Link()
public Link(SharedGroup sharedGroup)
```

The first form constructs a Link node object that does not yet point to a SharedGroup node. The second form constructs a Link node object that points to the specified SharedGroup node.

Methods

The Link node object defines two methods.

```
public final void setSharedGroup(SharedGroup sharedGroup)
public final SharedGroup getSharedGroup()
```

These methods access and modify the SharedGroup node associated with this Link leaf node.

6.2 Cloning Subgraphs

An application developer may wish to reuse a common subgraph without completely sharing that subgraph. For example, the developer may wish to create a parking lot scene consisting of multiple cars, each with a different color. The developer might define three basic types of cars, such as convertible, truck, and sedan. To create the parking lot scene, the application will instantiate each type of car several times. Then the application can change the color of the various instances to create more variety in the scene. Unlike shared subgraphs, each instance is a separate copy of the scene graph definition: Changes to one instance do not affect any other instance.

Java 3D provides the `cloneTree` method for this purpose. The `cloneTree` method allows the programmer to change some attributes (NodeComponent objects) in a scene graph, while at the same time sharing the majority of the scene graph data—the geometry.

Methods

```
public Node cloneTree()
public Node cloneTree(boolean forceDuplicate)
public Node cloneTree(boolean forceDuplicate,
       boolean allowDanglingReferences)
```

These methods start the cloning of the subgraph. The optional `forceDuplicate` parameter, when set to `true`, causes leaf NodeComponent objects to ignore their `duplicateOnCloneTree` value and always be duplicated (see Section 6.2.1, “References to Node Component Objects”). The `allowDanglingReferences` parameter, when set to `true`, will permit the cloning of a subgraph even when a dangling reference is generated (see Section 6.2.3, “Dangling References”). Setting `forceDuplicate` and `allowDanglingReferences` to `false` is the equivalent of calling `cloneTree` without any parameters. This will result in NodeComponent objects being either duplicated or referenced in the cloned node, based on their `duplicateOnCloneTree` value. A `DanglingReferenceException` will be thrown if a dangling reference is encountered.

When the `cloneTree` method is called on a node, that node is duplicated along with its entire internal state. If the node is a Group node, `cloneTree` is then called on each of the node’s children.

The `cloneTree` method cannot be called on a live or compiled scene graph.

6.2.1 References to Node Component Objects

When `cloneTree` reaches a leaf node, there are two possible actions for handling the leaf node’s NodeComponent objects (such as Material, Texture, and so forth). First, the cloned leaf node can reference the original leaf node’s NodeComponent object—the NodeComponent object itself is not duplicated. Since the cloned leaf node shares the NodeComponent object with the original leaf node, changing the data in the NodeComponent object will effect a change in both nodes. This mode would also be used for objects that are read-only at run time.

Alternatively, the NodeComponent object can be duplicated, in which case the new leaf node would reference the duplicated object. This mode allows data referenced by the newly created leaf node to be modified without that modification affecting the original leaf node.

Figure 6-2 shows two instances of NodeComponent objects that are shared and one NodeComponent element that is duplicated for the cloned subgraph.

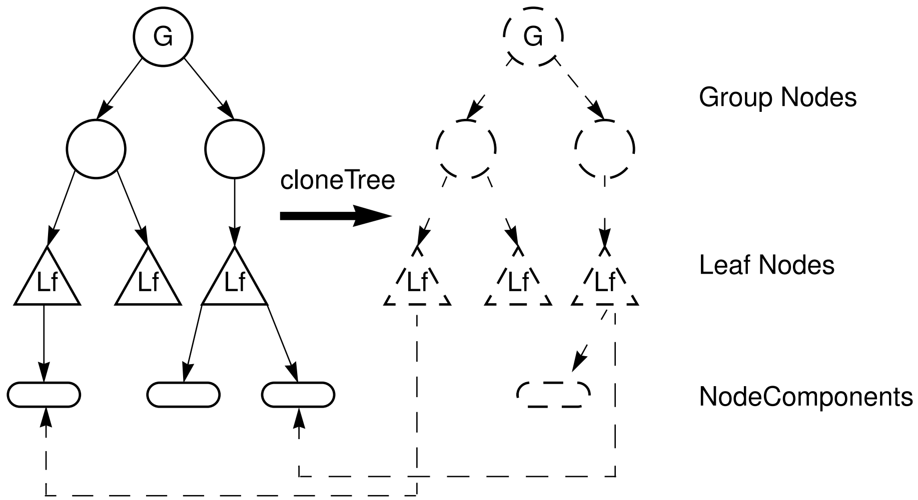


Figure 6-2 Referenced and Duplicated NodeComponent Objects

Methods

```
public final void setDuplicateOnCloneTree(boolean)
public final void getDuplicateOnCloneTree()
```

These methods set a flag that controls whether a NodeComponent object is duplicated or referenced on a call to `cloneTree`. By default this flag is `false`, meaning that the NodeComponent object will not be duplicated on a call to `cloneTree`—newly created leaf nodes will refer to the original NodeComponent object instead.

If the `cloneTree` method is called with the `forceDuplicate` parameter set to `true`, the `duplicateOnCloneTree` flag is ignored and the entire scene graph is duplicated.

6.2.2 References to Other Scene Graph Nodes

Leaf nodes that contain references to other nodes (for example, Light nodes reference a Group node) can create a problem for the `cloneTree` method. After the `cloneTree` operation is performed, the reference in the cloned leaf node will still refer to the node in the original subgraph—a situation that is most likely incorrect (see Figure 6-3).

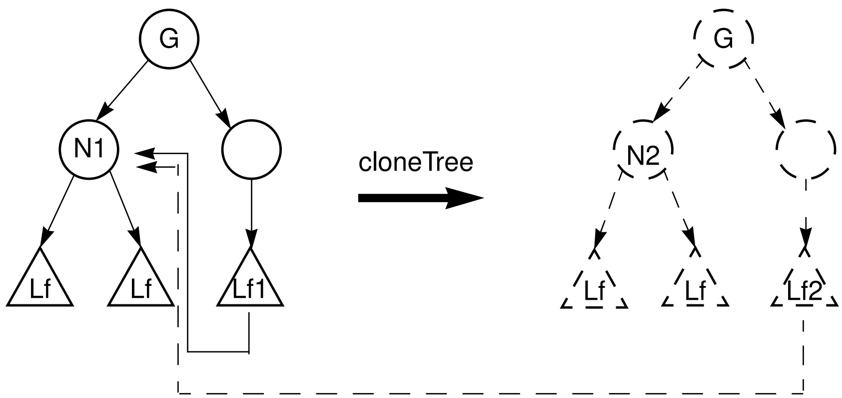


Figure 6-3 References to Other Scene Graph Nodes

To handle these ambiguities, a callback mechanism is provided.

A leaf node that needs to update referenced nodes upon being duplicated by a call to `cloneTree` must implement the `updateNodeReferences` method. By using this method, the cloned leaf node can determine if any nodes referenced by it have been duplicated and, if so, update the appropriate references to their cloned counterparts.

Suppose, for instance, that the leaf node Lf1 in Figure 6-3 implemented the `updateNodeReferences` method. Once all nodes had been duplicated, the `cloneTree` method would then call each cloned leaf's node `updateNodeReferences` method. When cloned leaf node Lf2's method was called, Lf2 could ask if the node N1 had been duplicated during the `cloneTree` operation. If the node had been duplicated, leaf Lf2 could then update its internal state with the cloned node, N2 (see Figure 6-4).

All predefined Java 3D nodes will automatically have their `updateNodeReferences` method defined. Only subclassed nodes that reference other nodes need to have this method overridden by the user.

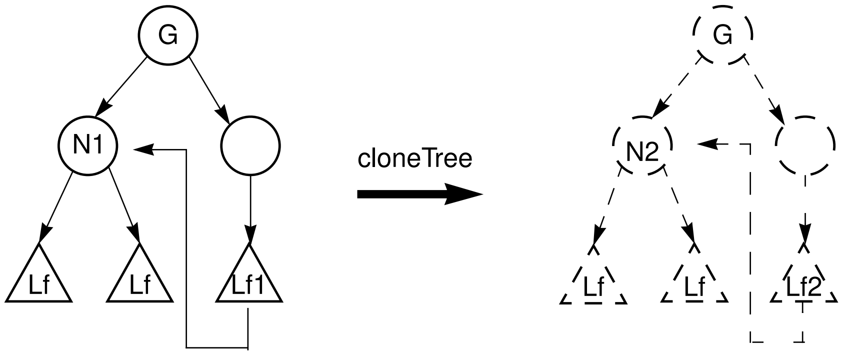


Figure 6-4 Updated Subgraph after `updateNodeReferences` Call

Methods

`public void updateNodeReferences(NodeReferenceTable referenceTable)`

This Leaf node method is called by the `cloneTree` method after all nodes in the subgraph have been cloned. The user can query the NodeReferenceTable object (see Section 6.2.5, "NodeReferenceTable Object") to determine if any nodes that the leaf node references have been duplicated by the `cloneTree` call and, if so, what the corresponding node is in the new subgraph. If a user extends a predefined Java 3D object and adds a reference to another node, this method must be defined in order to ensure proper operation of the `cloneTree` method. The first statement in the user's `updateNodeReferences` method must be `super.updateNodeReferences(referenceTable)`. For predefined Java 3D nodes, this method will be implemented automatically.

The NodeReferenceTable object is passed to the `updateNodeReferences` method and allows references from the old subgraph to be translated into refer-ences in the cloned subgraph. The translation is performed by the `getNew-NodeReference` method.

`public final Node getNewNodeReference(Node oldReference)`

Deprecated method. See the `getNewObjectReference` method.

```
public final SceneGraphObject
       getNewObjectReference(SceneGraphObject oldReference)
```

This method takes a reference to the node in the original subgraph as an input parameter and returns a reference to the equivalent node in the just-cloned subgraph. If the equivalent node in the cloned subgraph does not exist, either an exception is thrown or a reference to the original node is returned (see Section 6.2.3, "Dangling References").

6.2.3 Dangling References

Because `cloneTree` is able to start the cloning operation from any node, there is a potential for creating *dangling references*. A dangling reference can occur only when a leaf node that contains a reference to another scene graph node is cloned. If the referenced node is not cloned, a dangling reference situation exists: There are now two leaf nodes that access the same node (Figure 6-5). A dangling reference is discovered when a leaf node's `updateNodeReferences` method calls the `getNewNodeReference` method and the cloned subgraph does not contain a counterpart to the node being looked up.

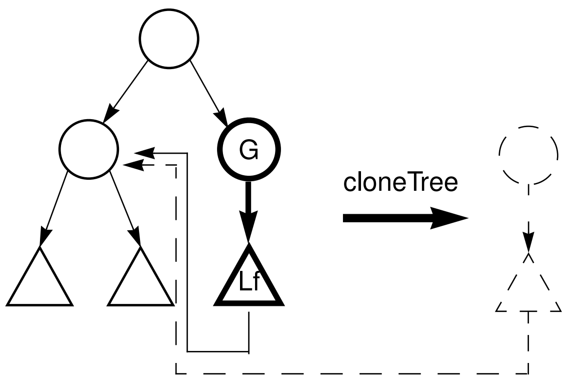


Figure 6-5 Dangling Reference: Bold Nodes Are Being Cloned

When a dangling reference is discovered, `cloneTree` can handle it in one of two ways. If `cloneTree` is called without the `allowDanglingReferences` parameter set to `true`, a dangling reference will result in a `DanglingReferenceException` being thrown. The user can catch this exception if desired. If `cloneTree` is called with the `allowDanglingReferences` parameter set to `true`, the `update-NodeReferences` method will return a reference to the same object passed into the `getNewNodeReference` method. This will result in the `cloneTree` operation completing with dangling references, as in Figure 6-5.

6.2.4 Subclassing Nodes

All Java 3D predefined nodes (for example, Interpolators and LOD nodes) automatically handle all node reference and duplication operations. When a user subclasses a Leaf object or a NodeComponent object, certain methods must be provided in order to ensure the proper operation of `cloneTree`.

Leaf node subclasses (for example, Behaviors) that contain any user node-specific data that needs to be duplicated during a `cloneTree` operation must define the following two methods:

`Node cloneNode(boolean forceDuplicate);`
`void duplicateNode(Node n, boolean forceDuplicate)`

The `cloneNode` method consists of three lines:

```
UserLeafNode un = new UserLeafNode();
un.duplicateNode(this, forceDuplicate);
return un;
```

The `duplicateNode` method must first call `super.duplicateNode` before duplicating any necessary user-specific data or setting any user-specific state.

NodeComponent subclasses that contain any user node-specific data must define the following two methods:

`NodeComponent cloneNodeComponent();`
`void duplicateNodeComponent(NodeComponent nc);`

The `cloneNodeComponent` method consists of three lines:

```
UserNodeComponent un = new UserNodeComponent();
un.duplicateNodeComponent(this);
return un;
```

The `duplicateNodeComponent` must first call `super.duplicateNodeComponent` and then can duplicate any user-specific data or set any user-specific state as necessary.

6.2.5 NodeReferenceTable Object

The NodeReferenceTable object is used by a leaf node's `updateNodeReferences` method called by the `cloneTree` operation. The NodeReferenceTable maps nodes from the original subgraph to the new nodes in the cloned subgraph. This information can than be used to update any cloned leaf node references to reference nodes in the cloned subgraph. This object can only be created by Java 3D.

Methods

`public final Node getNewNodeReference(Node oldReference)`

Deprecated method. See the `getNewObjectReference` method.

`public final SceneGraphObject`
`getNewObjectReference(SceneGraphObject oldReference)`

This method takes a reference to the node in the original subgraph as an input parameter and returns a reference to the equivalent node in the just-cloned subgraph. If the equivalent node in the cloned subgraph does not exist, either an exception is thrown or a reference to the original node is returned (see Section 6.2.3, "Dangling References").

6.2.6 Example User Behavior Node

The following is an example of a user-defined Behavior object to show how to properly define a node to be compatible with the `cloneTree` operation.

```
class RotationBehavior extends Behavior {
   TransformGroup objectTransform;
   WakeupOnElapsedFrames w;

   Matrix4d rotMat = new Matrix4d();
   Matrix4d objectMat = new Matrix4d();
   Transform3D t = new Transform();

   // Override Behavior's initialize method to set up wakeup
   // criteria
   public void initialize() {
      // Establish initial wakeup criteria
      wakeupOn(w);
   }

   // Override Behavior's stimulus method to handle the event
   public void processStimulus(Enumeration criteria) {
      // Rotate by another PI/120.0 radians
```

```
        objectMat.mul(objectMat, rotMat);
        t.set(objectMat);
        objectTransform.setTransform(t);

        // Set wakeup criteria for next time
        wakeupOn(w);
    }

    // Constructor for rotation behavior.
    public RotationBehavior(TransformGroup tg, int numFrames) {
        w = new WakeupOnElapsedFrames(numFrames);
        objectTransform = tg;
        objectMat.setIdentity();

        // Create a rotation matrix that rotates PI/120.0
        // radians per frame
        rotMat.rotX(Math.PI/120.0);

        // Note: When this object is duplicated via cloneTree,
        // the cloned RotationBehavior node needs to point to
        // the TransformGroup in the just-cloned tree.
    }

    // Sets a new TransformGroup.
    public void setTransformGroup(TransformGroup tg) {
        objectTransform = tg;
    }

    // The next two methods are needed for cloneTree to operate
    // correctly.
    // cloneNode is needed to provide a new instance of the user
    // derived subclass.
    public Node cloneNode(boolean forceDuplicate) {
        // Get all data from current node needed for
        // the constructor
        int numFrames = w.getElapsedFrameCount();

        RotationBehavior r =
            new RotationBehavior(objectTransform, w);
        r.duplicateNode(this, forceDuplicate);
        return r;
    }
    // duplicateNode is needed to duplicate all super class
    // data as well as all user data.
    public void duplicateNode(Node n, boolean forceDuplicate) {
        super.duplicateNode(n, forceDuplicate);
        // Nothing to do here - all unique data was handled
        // in the constructor in the cloneNode routine.
    }
```

```
    // Callback for when this leaf is cloned. For this object
    // we want to find the cloned TransformGroup node that this
    // clone Leaf node should reference.
    public void updateNodeReferences(NodeReferenceTable t) {
        super.updateNodeReferences(t);

        // Update node's TransformGroup to proper reference
        TransformGroup newTg =
         (TransformGroup)t.getNewNodeReference(objectTransform);
        setTransformGroup(newTg);
    }
}
```

CHAPTER 7

Node Component Objects

NODE component objects include the actual geometry and appearance attributes used to render the geometry.

7.1 Node Component Objects: Attributes

Node objects by themselves do not fully specify their exact semantics. They contain information that further refines their exact meaning. Some of that information is specified as an attribute and an associated floating-point or integer value. In many cases, however, the information consists of references to more complex entities called *node component objects*. Node component objects encapsulate related state information in a single entity. See Figure 7-1.

7.1.1 Appearance Object

The Appearance object is a component object of a Shape3D node that defines all rendering state attributes for that shape node. If the Appearance object in a Shape3D node is `null`, default values will be used for all rendering state attributes.

Constants

The Appearance component object defines the following flags.

```
public static final int ALLOW_MATERIAL_READ
public static final int ALLOW_MATERIAL_WRITE
public static final int ALLOW_TEXTURE_READ
public static final int ALLOW_TEXTURE_WRITE
public static final int ALLOW_TEXGEN_READ
public static final int ALLOW_TEXGEN_WRITE
```

```
SceneGraphObject
    NodeComponent
        Appearance
        AuralAttributes
        ColoringAttributes
        LineAttributes
        PointAttributes
        PolygonAttributes
        RenderingAttributes
        TextureAttributes
        TransparencyAttributes
        Material
        MediaContainer
        TexCoordGeneration
        Texture
            Texture2D
            Texture3D
        ImageComponent
            ImageComponent2D
            ImageComponent3D
        DepthComponent
            DepthComponentFloat
            DepthComponentInt
            DepthComponentNative
Bounds
    BoundingBox
    BoundingPolytope
    BoundingSphere
Transform3D
```

Figure 7-1 Attribute Component Object Hierarchy

```
public static final int ALLOW_TEXTURE_ATTRIBUTES_READ
public static final int ALLOW_TEXTURE_ATTRIBUTES_WRITE
public static final int ALLOW_COLORING_ATTRIBUTES_READ
public static final int ALLOW_COLORING_ATTRIBUTES_WRITE
public static final int ALLOW_TRANSPARENCY_ATTRIBUTES_READ
public static final int ALLOW_TRANSPARENCY_ATTRIBUTES_WRITE
public static final int ALLOW_RENDERING_ATTRIBUTES_READ
public static final int ALLOW_RENDERING_ATTRIBUTES_WRITE
public static final int ALLOW_POLYGON_ATTRIBUTES_READ
public static final int ALLOW_POLYGON_ATTRIBUTES_WRITE
public static final int ALLOW_LINE_ATTRIBUTES_READ
public static final int ALLOW_LINE_ATTRIBUTES_WRITE
```

public static final int ALLOW_POINT_ATTRIBUTES_READ
public static final int ALLOW_POINT_ATTRIBUTES_WRITE

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read and write the specified component object reference (material, texture, texture coordinate generation, and so forth). These capability flags are enforced only when the object is part of a live or compiled scene graph.

Constructors

The Appearance object has the following constructor.

public Appearance()

Constructs and initializes an Appearance object. All component object references are initialized to `null`.

The default values, for those objects with `null` references, are as follows:

color: white (1,1,1)
texture environment mode: TEXENV_REPLACE
texture environment color: white (1,1,1,1)
depth test enable: true
shade model: SHADE_SMOOTH
polygon mode: POLYGON_FILL
transparency enable: false
transparency mode: FASTEST
cull face: CULL_BACK
point size: 1.0
line width: 1.0
line pattern: PATTERN_SOLID
point antialiasing enable: false
line antialiasing enable: false

Methods

The Appearance object has the following methods.

public final void setMaterial(Material material)
public final Material getMaterial()

The Material object specifies the desired material properties used for lighting. Setting it to `null` disables lighting.

public final void setTexture(Texture texture)
public final Texture getTexture()

The Texture object specifies the desired texture map and texture parameters. Setting it to `null` disables texture mapping.

public final void setTextureAttributes(TextureAttributes textureAttributes)
public final TextureAttributes getTextureAttributes()

These methods set and retrieve the TextureAttributes object. Setting it to `null` results in default attribute use.

public final void setColoringAttributes(ColoringAttributes coloringAttributes)
public final ColoringAttributes getColoringAttributes()

These methods set and retrieve the ColoringAttributes object. Setting it to `null` results in default attribute use.

public final void setTransparencyAttributes(TransparencyAttributes transparencyAttributes)
public final TransparencyAttributes getTransparencyAttributes()

These methods set and retrieve the TransparencyAttributes object. Setting it to `null` results in default attribute use.

public final void setRenderingAttributes(RenderingAttributes renderingAttributes)
public final RenderingAttributes getRenderingAttributes()

These methods set and retrieve the RenderingAttributes object. Setting it to `null` results in default attribute use.

public final void setPolygonAttributes(PolygonAttributes polygonAttributes)
public final PolygonAttributes getPolygonAttributes()

These methods set and retrieve the PolygonAttributes object. Setting it to `null` results in default attribute use.

public final void setLineAttributes(LineAttributes lineAttributes)
public final LineAttributes getLineAttributes()

These methods set and retrieve the LineAttributes object. Setting it to `null` results in default attribute use.

```
public final void setPointAttributes(PointAttributes
       pointAttributes)
public final PointAttributes getPointAttributes()
```

These methods set and retrieve the PointAttributes object. Setting it to `null` results in default attribute use.

```
public final void setTexCoordGeneration(TexCoordGeneration
       texCoordGeneration)
public final TexCoordGeneration getTexCoordGeneration()
```

These methods set and retrieve the TexCoordGeneration object. Setting it to `null` disables texture coordinate generation.

```
public NodeComponent cloneNodeComponent()
```

This method creates a new Appearance object. The method is called from a leaf node's `duplicateNode` method.

```
public void duplicateNodeComponent(NodeComponent originalNode)
```

This method copies the information found in `originalNode` to the current node. This routine is called as part of the `cloneTree` operation.

7.1.2 ColoringAttributes Object

The ColoringAttributes object defines attributes that apply to color mapping.

Constants

```
public static final int ALLOW_COLOR_READ
public static final int ALLOW_COLOR_WRITE
public static final int ALLOW_SHADE_MODEL_READ
public static final int ALLOW_SHADE_MODEL_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write its color component and shade model component information.

Constructors

```
public ColoringAttributes()
public ColoringAttributes(Color3f color, int shadeModel)
public ColoringAttributes(float red, float green, float blue,
       int shadeModel)
```

These constructors create a ColoringAttributes object with the specified values.

Methods

```
public final void setColor(Color3f color)
public final void setColor(float r, float g, float b)
public final void getColor(Color3f color)
```

These methods set and retrieve the intrinsic color of this ColoringAttributes component object. This color is used when lighting is disabled or when the Material is `null`.

```
public final void setShadeModel(int shadeModel)
public final int getShadeModel()
```

These methods set and retrieve the shade model for this ColoringAttributes component object. The shade model is one of the following:

- FASTEST: Uses the fastest available method for shading.
- NICEST: Uses the nicest (highest quality) available method for shading.
- SHADE_FLAT: Does not interpolate color across the primitive.
- SHADE_GOURAUD: Smoothly interpolates the color at each vertex across the primitive.

```
public NodeComponent cloneNodeComponent()
```

This method creates a new ColoringAttributes object. This method is called from a leaf node's `duplicateNode` method.

```
public void duplicateNodeComponent(NodeComponent originalNode)
```

This method copies the information found in `originalNode` to the current node. This method is called as part of the `cloneTree` operation.

7.1.3 LineAttributes Object

The LineAttributes object defines attributes that apply to line primitives.

Constants

The LineAttributes object specifies the following variables.

```
public static final int ALLOW_WIDTH_READ
public static final int ALLOW_WIDTH_WRITE
public static final int ALLOW_PATTERN_READ
public static final int ALLOW_PATTERN_WRITE
public static final int ALLOW_ANTIALIASING_READ
public static final int ALLOW_ANTIALIASING_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read and write its individual component field information.

```
public static final int PATTERN_SOLID
```

Draws a solid line with no pattern.

```
public static final int PATTERN_DASH
```

Draws a dashed line. Ideally, this will be drawn with a repeating pattern of eight pixels on and eight pixels off.

```
public static final int PATTERN_DOT
```

Draws a dotted line. Ideally, this will be drawn with a repeating pattern of one pixel on and seven pixels off.

```
public static final int PATTERN_DASH_DOT
```

Draws a dashed-dotted line. Ideally, this will be drawn with a repeating pattern of seven pixels on, four pixels off, one pixel on, and four pixels off.

Constructors

```
public LineAttributes()
public LineAttributes(float lineWidth, int linePattern,
       boolean lineAntialiasing)
```

The first constructor creates a LineAttributes object with default values. The second constructor creates a LineAttributes object with specified values of line width, pattern, and whether antialiasing is enabled or disabled.

Methods

```
public final void setLineWidth(float lineWidth)
public final float getLineWidth()
```

These methods respectively set and retrieve the line width, in pixels, for this LineAttributes component object.

`public final void setLinePattern(int linePattern)`
`public final int getLinePattern()`

These methods respectively set and retrieve the line pattern for this LineAttributes component object. The `linePattern` value describes the line pattern to be used, which is one of the following: `PATTERN_SOLID`, PATTERN_DASH, `PATTERN_DOT`, or `PATTERN_DASH_DOT`.

`public final void setLineAntialiasingEnable(boolean state)`
`public final boolean getLineAntialiasingEnable()`

The `set` method enables or disables line antialiasing for this LineAttributes component object. The `get` method retrieves the state of the line antialiasing flag. The flag is `true` if line antialiasing is enabled, `false` if line antialiasing is disabled.

`public NodeComponent cloneNodeComponent()`
`public void duplicateNodeComponent(NodeComponent originalNode)`

The first method creates a new LineAttributes object; this method is called from a leaf node's `duplicateNode` method. The second method copies the information found in `originalNode` to the current node; this method is called as part of the `cloneTree` operation.

7.1.4 PointAttributes Object

The PointAttributes object defines attributes that apply to point primitives.

Constants

The PointAttributes object specifies the following variables.

`public final static int ALLOW_SIZE_READ`
`public final static int ALLOW_SIZE_WRITE`
`public final static int ALLOW_ANTIALIASING_READ`
`public final static int ALLOW_ANTIALIASING_WRITE`

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read and write its individual component field information.

Constructors

```
public PointAttributes()
public PointAttributes(float pointSize,
       boolean pointAntialiasing)
```

These constructors create a new PointAttributes object.

Methods

```
public final void setPointSize(float pointSize)
public final float getPointSize()
```

These methods set and retrieve the point size, in pixels, for this Appearance component object.

```
public final void setPointAntialiasingEnable(boolean state)
public final boolean getPointAntialiasingEnable()
```

The `set` method enables or disables point antialiasing for this PointAttributes component object. The `get` method retrieves the state of the point antialiasing flag. The flag is `true` if point antialiasing is enabled, `false` if point antialiasing is disabled.

```
public NodeComponent cloneNodeComponent()
public void duplicateNodeComponent(NodeComponent originalNode)
```

The first method creates a new PointAttributes object; this method is called from a leaf node's `duplicateNode` method. The second method copies the information found in `originalNode` to the current node; this method is called as part of the `cloneTree` operation.

7.1.5 PolygonAttributes Object

The PolygonAttributes object defines attributes that apply to polygon primitives.

Constants

The PolygonAttributes object specifies the following variables.

```
public final static int ALLOW_CULL_FACE_READ
public final static int ALLOW_CULL_FACE_WRITE
public final static int ALLOW_MODE_READ
public final static int ALLOW_MODE_WRITE
public final static int ALLOW_OFFSET_READ
public final static int ALLOW_OFFSET_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read and write its individual component field information.

Constructors

```
public PolygonAttributes()
public PolygonAttributes(int polygonMode, int cullFace,
       float polygonOffset)
```

These constructors create a new PolygonAttributes object.

Methods

```
public final void setCullFace(int cullFace)
public final int getCullFace()
```

These methods set and retrieve the face culling flag for this PolygonAttributes component object. The face culling flag is one of the following:

- CULL_NONE: Performs no face culling.
- CULL_FRONT: Culls all front-facing polygons.
- CULL_BACK: Culls all back-facing polygons.

```
public final void setPolygonMode(int polygonMode)
public final int getPolygonMode()
```

These methods set and retrieve the polygon rasterization mode for this Appearance component object. The polygon rasterization mode is one of the following:

- POLYGON_POINT: Renders polygonal primitives as points drawn at the vertices of the polygon.
- POLYGON_LINE: Renders polygonal primitives as lines drawn between consecutive vertices of the polygon.
- POLYGON_FILL: Renders polygonal primitives by filling the interior of the polygon.

```
public final void setPolygonOffset(float polygonOffset)
public final float getPolygonOffset()
```

These methods set and retrieve the polygon offset. This screen-space offset is added to the final, device-coordinate Z value of polygon primitives.

```
public NodeComponent cloneNodeComponent()
public void duplicateNodeComponent(NodeComponent originalNode)
```

The first method creates a new PolygonAttributes object; this method is called from a leaf node's `duplicateNode` method. The second method copies the information found in `originalNode` to the current node; this method is called as part of the `cloneTree` operation.

7.1.6 RenderingAttributes Object

The RenderingAttributes object defines per-pixel rendering state attributes common to all primitive types.

Constants

```
public static final int ALLOW_ALPHA_TEST_VALUE_READ
public static final int ALLOW_ALPHA_TEST_VALUE_WRITE
public static final int ALLOW_ALPHA_TEST_FUNCTION_READ
public static final int ALLOW_ALPHA_TEST_FUNCTION_WRITE
public static final int ALLOW_DEPTH_ENABLE_READ
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write its individual test value and function information.

Constructors

```
public RenderingAttributes()
public RenderingAttributes(boolean depthBufferEnable,
       boolean depthBufferWriteEnable, float alphaTestValue,
       int alphaTestFunction)
```

These constructors create a new RenderingAttributes object.

Methods

`public final void setDepthBufferEnable(boolean state)`
`public final boolean getDepthBufferEnable()`

These methods set and retrieve the depth buffer enable flag for this RenderingAttributes component object. The flag is `true` if the depth buffer mode is enabled, `false` if disabled.

`public final void setDepthBufferWriteEnable(boolean state)`
`public final boolean getDepthBufferWriteEnable()`

These methods set and retrieve the depth buffer write enable flag for this RenderAttributes component object. The flag is `true` if the depth buffer mode is writable, `false` if the depth buffer is read-only.

`public final void setAlphaTestValue(float value)`
`public final float getAlphaTestValue()`

These methods set and retrieve the alpha test value used by the alpha test function. This value is compared to the alpha value of each rendered pixel.

`public final void setAlphaTestFunction(int function)`
`public final int getAlphaTestFunction()`

These methods set and retrieve the alpha test function. The alpha test function is one of the following:

- ALWAYS: Indicates pixels are always drawn irrespective of the alpha value. This effectively disables alpha testing.
- NEVER: Indicates pixels are never drawn irrespective of the alpha value.
- EQUAL: Indicates pixels are drawn if the pixel alpha value is equal to the alpha test value.
- NOT_EQUAL: Indicates pixels are drawn if the pixel alpha value is not equal to the alpha test value.
- LESS: Indicates pixels are drawn if the pixel alpha value is less than the alpha test value.
- LESS_OR_EQUAL: Indicates pixels are drawn if the pixel alpha value is less than or equal to the alpha test value.
- GREATER: Indicates pixels are drawn if the pixel alpha value is greater than the alpha test value.
- GREATER_OR_EQUAL: Indicates pixels are drawn if the pixel alpha value is greater than or equal to the alpha test value.

```
public NodeComponent cloneNodeComponent()
public void duplicateNodeComponent(NodeComponent originalNode)
```

The first method creates a new RenderingAttributes object; this method is called from a leaf node's `duplicateNode` method. The second method copies the information found in `originalNode` to the current node; this method is called as part of the `cloneTree` operation.

7.1.7 TextureAttributes Object

The TextureAttributes object defines attributes that apply to texture mapping.

Constants

```
public static final int ALLOW_MODE_READ
public static final int ALLOW_MODE_WRITE
public static final int ALLOW_BLEND_COLOR_READ
public static final int ALLOW_BLEND_COLOR_WRITE
public static final int ALLOW_TRANSFORM_READ
public static final int ALLOW_TRANSFORM_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write its individual component field information.

Constructors

```
public TextureAttributes()
public TextureAttributes(int textureMode, Transform3D transform,
       Color4f textureBlendColor, int perspCorrectionMode)
```

These constructors create a new TextureAttributes object.

Methods

```
public final void setTextureMode(int textureMode)
public final int getTextureMode()
```

These methods set and retrieve the texture mode parameter for this TextureAttributes component object. The texture mode is one of the following:

- MODULATE: Modulates the object color with the texture color.
- DECAL: Applies the texture color to the object as a decal.
- BLEND: Blends the texture blend color with the object color.
- REPLACE: Replaces the object color with the texture color.

```
public final void setTextureBlendColor(Color4f textureBlendColor)
public final void setTextureBlendColor(float r, float g, float b,
       float a)
public final void getTextureBlendColor(Color4f textureBlendColor)
```

These methods set and retrieve the texture blend color for this TextureAttributes component object. The texture blend color is used when the texture mode parameter is `BLEND`.

```
public final void setTextureTransform(Transform3D transform)
public final void getTextureTransform(Transform3D transform)
```

These methods set and retrieve the texture transform object used to transform texture coordinates. A copy of the specified Transform3D object is stored in this TextureAttributes object.

```
public final void setPerspectiveCorrectionMode(int mode)
public final int getPerspectiveCorrectionMode()
```

These methods set and retrieve the perspective correction mode to be used for color and texture coordinate interpolation. The perspective correction mode is one of the following:

- NICEST: Uses the nicest (highest quality) available method for texture mapping perspective correction.
- FASTEST: Uses the fastest available method for texture mapping perspective correction.

```
public NodeComponent cloneNodeComponent()
public void duplicateNodeComponent(NodeComponent originalNode)
```

The first method creates a new TextureAttributes object; this method is called from a leaf node's `duplicateNode` method. The second method copies the information found in `originalNode` to the current node; this method is called as part of the `cloneTree` operation.

7.1.8 TransparencyAttributes Object

The TransparencyAttributes object defines all attributes affecting the transparency of the object.

Constants

```
public static final int ALLOW_MODE_READ
public static final int ALLOW_MODE_WRITE
public static final int ALLOW_VALUE_READ
public static final int ALLOW_VALUE_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write its individual component field information.

Constructors

```
public TransparencyAttributes()
public TransparencyAttributes(int tMode, float tVal)
```

These constructors create a new TransparencyAttributes object.

Methods

```
public final void setTransparencyMode(int transparencyMode)
public final int getTransparencyMode()
```

These methods set and retrieve the transparency mode for this Appearance component object. The transparency mode is one of the following:

- FASTEST: Uses the fastest available method for transparency.
- NICEST: Uses the nicest available method for transparency.
- SCREEN_DOOR: Uses screen-door transparency. This is done using an on/off stipple pattern in which the percentage of transparent pixels is approximately equal to the value specified by the transparency parameter.
- BLENDED: Uses alpha blended transparency. A blend equation of (alpha*src + (1 – alpha)*dst) is used, where alpha is (1 – transparency).
- NONE: No transparency; opaque object.

```
public final void setTransparency(float transparency)
public final float getTransparency()
```

These methods set and retrieve this Appearance object's transparency value. The transparency value is in the range [0.0, 1.0], with 0.0 being fully opaque and 1.0 being fully transparent.

```
public NodeComponent cloneNodeComponent()
public void duplicateNodeComponent(NodeComponent originalNode)
```

The first method creates a new TransparencyAttributes object; this method is called from a leaf node's `duplicateNode` method. The second method copies the information found in `originalNode` to the current node; this method is called as part of the `cloneTree` operation.

7.1.9 Material Object

The Material object is a component object of an Appearance object that defines the material properties used when lighting is enabled. If the Material object in an Appearance object is `null`, lighting is disabled for all nodes that use that Appearance object.

Constants

The Material object defines two flags.

```
public static final int ALLOW_COMPONENT_READ
public static final int ALLOW_COMPONENT_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that respectively read and write its individual component field information.

Constructors

The Material object has the following constructors.

```
public Material()
```

Constructs and initializes a Material object using default values for all attributes. The default values are as follows:

ambient color: 0.2, 0.2, 0.2
emissive color: black (0.0, 0.0, 0.0)
diffuse color: white (1.0, 1.0, 1.0)
specular color: white (1.0, 1.0, 1.0)
shininess: 64.0

```
public Material(Color3f ambientColor, Color3f emmissiveColor,
       Color3f diffuseColor, Color3f specularColor,
       float shininess)
```

Constructs and initializes a new Material object using the specified parameters. The ambient color, emissive color, diffuse color, specular color, and shininess parameters are specified.

Methods

The Material object has the following methods.

```
public final void setAmbientColor(Color3f color)
public final void setAmbientColor(float r, float g, float b)
public final void getAmbientColor(Color3f color)
```

This parameter specifies this material's ambient color, that is, how much ambient light is reflected by the material's surface.

```
public final void setEmissiveColor(Color3f color)
public final void setEmissiveColor(float r, float g, float b)
public final void getEmissiveColor(Color3f color)
```

This parameter specifies the color of light, if any, that the material emits. This color is added to the color produced by applying the lighting equation.

```
public final void setDiffuseColor(Color3f color)
public final void setDiffuseColor(float r, float g, float b)
public final void setDiffuseColor(float r, float g, float b,
       float a)
public final void getDiffuseColor(Color3f color)
```

This parameter specifies the color of the material when illuminated by a light source. In addition to the diffuse color (red, green, and blue), the alpha value is used to specify transparency such that transparency = (1 – alpha).

```
public final void setSpecularColor(Color3f color)
public final void setSpecularColor(float r, float g, float b)
public final void getSpecularColor(Color3f color)
```

This parameter specifies the specular highlight color of the material.

`public final void setShininess(float shininess)`
`public final float getShininess()`

This parameter specifies a material specular scattering exponent, or shininess. It takes a floating-point number in the range [1.0, 128.0], with 1.0 being not shiny and 128.0 being very shiny.

`public final void setLightingEnable(boolean state)`
`public final boolean getLightingEnable()`

These methods set and retrieve the current state of the lighting enable flag (`true` or `false`) for this Appearance component object.

`public NodeComponent cloneNodeComponent()`
`public void duplicateNodeComponent(NodeComponent originalNode)`

The first method creates a new Material object; this method is called from a leaf node's `duplicateNode` method. The second method copies the information found in `originalNode` to the current node; this method is called as part of the `cloneTree` operation.

`public String toString()`

This method returns a string representation of this Material's values. If the scene graph is live, only those values with their capability bit set will be displayed.

7.1.10 Texture Object

The Texture object is a component object of an Appearance object that defines the texture properties used when texture mapping is enabled. If the Texture object in an Appearance object is `null`, then texture mapping is disabled for all nodes that use that Appearance object. The Texture object is an abstract class. As such, all texture objects must be created as either a Texture2D object or a Texture3D object.

Constants

The Texture object defines the following flags:

```
public static final int ALLOW_ENABLE_READ
public static final int ALLOW_ENABLE_WRITE
public static final int ALLOW_BOUNDARY_MODE_READ
public static final int ALLOW_FILTER_READ
public static final int ALLOW_IMAGE_READ
public static final int ALLOW_MIPMAP_MODE_READ
public static final int ALLOW_BOUNDARY_COLOR_READ
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read, and in some cases write, its individual component field information.

Constructors

The Texture object has the following constructor.

```
public Texture()
```

This constructor is not very useful as the default width and height are 0. The other default values are as follows:

boundaryModeS: WRAP
boundaryModeT: WRAP
minification filter: BASE_LEVEL_POINT
magnification filter: BASE_LEVEL_POINT
boundary color: black (0,0,0,0)
texture image: null

```
public Texture(int mipmapMode, int format, int width, int height)
```

Constructs an empty Texture object with specified `mipmapMode` format, width, and height. Image at level 0 must be set by the application using the `setImage` method. The `mipmapMode` can be one of the following:

- BASE_LEVEL: Indicates that this Texture object only has a base-level image. If multiple levels are needed, they will be implicitly computed.
- MULTI_LEVEL_MIPMAP: Indicates that this Texture object has multiple images—one for each mipmap level (that is, $\log_2(\max(W,H)) + 1$ separate images). If `mipmapMode` is set to `MULTI_LEVEL_MIPMAP`, images for *all* levels must be set.

The `format` is the data of textures saved in this object. The `format` can be one of the following:

- INTENSITY: Specifies Texture contains only intensity values.
- LUMINANCE: Specifies Texture contains only luminance values.

- ALPHA: Specifies Texture contains only alpha values.
- LUMINANCE_ALPHA: Specifies Texture contains luminance and alpha values.
- RGB: Specifies Texture contains red, green, and blue color values.
- RGBA: Specifies Texture contains red, green, and blue color values, and an alpha value.

Methods

The Texture object has the following methods.

```
public final void setBoundaryModeS(int boundaryModeS)
public final int getBoundaryModeS()
public final void setBoundaryModeT(int boundaryModeT)
public final int getBoundaryModeT()
```

These parameters specify the boundary mode for the S and T coordinates in this Texture object. The boundary mode is as follows:

- CLAMP: Clamps texture coordinates to be in the range [0, 1]. A constant boundary color is used for U,V values that fall outside this range.
- WRAP: Repeats the texture by wrapping texture coordinates that are outside the range [0, 1]. Only the fractional portion of the texture coordinates is used; the integer portion is discarded.

```
public final void setMinFilter(int minFilter)
public final int getMinFilter()
```

This parameter specifies the minification filter function. This function is used when the pixel being rendered maps to an area greater than one texel. The minification filter is one of the following:

- FASTEST: Uses the fastest available method for processing geometry.
- NICEST: Uses the nicest available method for processing geometry.
- BASE_LEVEL_POINT: Selects the nearest texel in the level 0 texture map.
- BASE_LEVEL_LINEAR: Performs a bilinear interpolation on the four nearest texels in the level 0 texture map.
- MULTI_LEVEL_POINT: Selects the nearest texel in the nearest mipmap.
- MULTI_LEVEL_LINEAR: Performs trilinear interpolation of texels between four texels each from the two nearest mipmap levels.

```
public final void setMagFilter(int magFilter)
public final int getMagFilter()
```

This parameter specifies the magnification filter function. This function is used when the pixel being rendered maps to an area less than or equal to one texel. The value is one of the following:

- FASTEST: Uses the fastest available method for processing geometry.
- NICEST: Uses the nicest available method for processing geometry.
- BASE_LEVEL_POINT: Selects the nearest texel in the level 0 texture map.
- BASE_LEVEL_LINEAR: Performs a bilinear interpolation on the four nearest texels in the level 0 texture map.

```
public final void setImage(int level, ImageComponent image)
public final ImageComponent getImage(int level)
```

These methods set and retrieve a specified mipmap level. Level 0 is the base level.

```
public final void setBoundaryColor(Color4f boundaryColor)
public final void setBoundaryColor(float r, float g, float b,
       float a)
public final void getBoundaryColor(Color4f boundaryColor)
```

This parameter specifies the texture boundary color for this Texture object. The texture boundary color is used when `boundaryModeS` or `boundaryModeT` is set to `CLAMP`.

```
public final void setEnable(boolean state)
public final boolean getEnable()
```

These methods set and retrieve the state of texture mapping for this Texture object. A value of `true` means that texture mapping is enabled, `false` means that texture mapping is disabled.

```
public final void setMipMapMode(int mipmapMode)
public final int getMipMapMode()
```

These methods set and retrieve the mipmap mode for texture mapping for this Texture object. The mipmap mode is either `BASE_LEVEL` or `MULTI_LEVEL_MIP_MAP`.

7.1.11 Texture2D Object

The Texture2D object is a subclass of the Texture class. It extends the Texture class by adding a constructor for setting a 2D texture image.

Constructors

The Texture2D object has the following constructors.

```
public Texture2D()
```

This constructor is not very useful as the default width and height are 0.

```
public Texture2D(int mipmapMode, int format, int width, int height)
```

Constructs and initializes a Texture2D object with the specified attributes. The `mipmapMode` parameter is either `BASE_LEVEL` or `MULTI_LEVEL_MIPMAP`. The `format` parameter is one of the following: `INTENSITY`, `LUMINANCE`, `ALPHA`, `LUMINANCE_ALPHA`, `RGB`, or `RGBA`.

Methods

```
public NodeComponent cloneNodeComponent()
public void duplicateNodeComponent(NodeComponent originalNode)
```

The first method creates a new Texture2D object; this method is called from a leaf node's `duplicateNode` method. The second method copies the information found in `originalNode` to the current node; this method is called as part of the `cloneTree` operation.

7.1.12 Texture3D Object

The Texture3D object is a subclass of the Texture class. It extends the Texture class by adding a third texture coordinate and by adding a constructor for setting a 3D texture image.

Constructors

The Texture3D object has the following constructors.

```
public Texture3D()
```

This constructor is not very useful as the default width, height, and depth are 0.

```
public Texture3D(int mipmapMode, int format, int width, int height,
       int depth)
```

Constructs and initializes a Texture3D object using the specified attributes. The `mipmapMode` parameter is either `BASE_LEVEL` or `MULTI_LEVEL_MIPMAP`. The `format` parameter is one of `INTENSITY`, `LUMINANCE`, `ALPHA`, `LUMINANCE_ALPHA`, `RGB`, or `RGBA`. The default value for a Texture3D object is as follows:

- **boundaryModeR**: WRAP

Methods

The Texture3D object has the following methods.

```
public final void setBoundaryModeR(int boundaryModeR)
public final int getBoundaryModeR()
```

This parameter specifies the boundary mode for the R coordinate in this Texture object. The boundary mode is as follows:

- CLAMP: Clamps texture coordinates to be in the range [0, 1]. A constant boundary color is used for R values that fall outside this range.
- WRAP: Repeats the texture by wrapping texture coordinates that are outside the range [0, 1]. Only the fractional portion of the texture coordinates is used; the integer portion is discarded.

```
public NodeComponent cloneNodeComponent()
public void duplicateNodeComponent(NodeComponent originalNode)
```

The first method creates a new Texture3D object; this method is called from a leaf node's `duplicateNode` method. The second method copies the information found in `originalNode` to the current node; this method is called as part of the `cloneTree` operation.

7.1.13 TexCoordGeneration Object

The TexCoordGeneration object is a component object of an Appearance object that defines the parameters used when texture coordinate generation is enabled. If the TexCoordGeneration object in an Appearance object is `null`, texture coordinate generation is disabled for all nodes that use that Appearance object.

Constants

The TexCoordGeneration object specifies the following variables.

```
public final static int ALLOW_ENABLE_READ
public final static int ALLOW_ENABLE_WRITE
public final static int ALLOW_FORMAT_READ
public final static int ALLOW_MODE_READ
public final static int ALLOW_PLANE_READ
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read, and in some cases write, its individual component field information.

```
public final static int OBJECT_LINEAR
```

Generates texture coordinates as a linear function in object coordinates.

```
public final static int EYE_LINEAR
```

Generates texture coordinates as a linear function in eye coordinates.

```
public final static int SPHERE_MAP
```

Generates texture coordinates using a spherical reflection mapping in eye coordinates.

```
public final static int TEXTURE_COORDINATE_2
```

Generates 2D texture coordinates (S and T).

```
public final static int TEXTURE_COORDINATE_3
```

Generates 3D texture coordinates (S, T, and R).

Constructors

The TexGen object has the following constructors.

```
public TexCoordGeneration()
public TexCoordGeneration(int genMode, int format)
public TexCoordGeneration(int genMode, int format,
       Vector4f planeS)
public TexCoordGeneration(int genMode, int format,
       Vector4f planeS, Vector4f planeT)
public TexCoordGeneration(int genMode, int format,
       Vector4f planeS, Vector4f planeT, Vector4f planeR)
```

The first form constructs a TexGen object using default values for all state variables. The other forms construct a TexGen object by initializing the specified fields. Default values are used for those state variables not specified in the constructor. The parameters are as follows:

- genMode: Texture generation mode. One of OBJECT_LINEAR, EYE_LINEAR, or SPHERE_MAP.
- format: Texture format (2D or 3D). Either TEXTURE_COORDINATE_2 or TEXTURE_COORDINATE_3.
- planeS: Plane equation for the S coordinate.
- planeT: Plane equation for the T coordinate.
- planeR: Plane equation for the R coordinate.

Default values for parameters that are not specified are as follows:

genMode: OBJECT_LINEAR
format: TEXTURE_COORDINATE_2
planeS: (1, 0, 0, 0)
planeT: (0, 1, 0, 0)
planeR: (0, 0, 0, 0)

Methods

The TexGen object has the following methods.

```
public final void setEnable(boolean state)
public final boolean getEnable()
```

This parameter enables or disables texture coordinate generation for this Appearance component object. The value is `true` if texture coordinate generation is enabled, `false` if texture coordinate generation is disabled.

```
public final void setFormat(int format)
public final int getFormat()
```

This parameter specifies the format, or dimension, of the generated texture coordinates. The format value is either TEXTURE_COORDINATE_2 or TEXTURE_COORDINATE_3.

```
public final void setGenMode(int genMode)
public final int getGenMode()
```

This parameter specifies the texture coordinate generation mode. The value is one of OBJECT_LINEAR, EYE_LINEAR, or SPHERE_MAP.

```
public final void setPlaneS(Vector4f planeS)
public final void getPlaneS(Vector4f planeS)
```

This parameter specifies the S coordinate plane equation. This plane equation is used to generate the S coordinate in `OBJECT_LINEAR` and `EYE_LINEAR` texture generation modes.

```
public final void setPlaneT(Vector4f planeT)
public final void getPlaneT(Vector4f planeT)
```

This parameter specifies the T coordinate plane equation. This plane equation is used to generate the T coordinate in `OBJECT_LINEAR` and `EYE_LINEAR` texture generation modes.

```
public final void setPlaneR(Vector4f planeR)
public final void getPlaneR(Vector4f planeR)
```

This parameter specifies the R coordinate plane equation. This plane equation is used to generate the R coordinate in `OBJECT_LINEAR` and `EYE_LINEAR` texture generation modes.

```
public NodeComponent cloneNodeComponent()
public void duplicateNodeComponent(NodeComponent originalNode)
```

The first method creates a new TexCoordGeneration object; this method is called from a leaf node's `duplicateNode` method. The second method copies the information found in `originalNode` to the current node; this method is called as part of the `cloneTree` operation.

7.1.14 MediaContainer Object

The MediaContainer object is a component object of a Sound node that defines the sound data associated with a Sound node. This component object's fields reference a Java Media Framework Player (which contains audio data), a Java Media Sound data container, or explicit sound sample data. Its fields include a cache flag and a URL path to sound data recognized by JavaSound (a proposed Java Media API) as a valid container that includes audio data. Eventually, when the JavaSound API is completed, the application can use JavaSound query methods to determine the format, precision, encoding and compression type, data length, and number of channels required for playback for a particular MediaContainer at the given URL path.

Constants

The MediaContainer object has the following flags.

```
public static final int ALLOW_CACHE_READ
public static final int ALLOW_CACHE_WRITE
public static final int ALLOW_URL_READ
public static final int ALLOW_URL_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read or write its cached flag and its URL string.

Constructors

The MediaContainer object has the following constructors.

```
public MediaContainer()
```

Constructs and initializes a new MediaContainer object using the following default values.

cache data: false
URL: null

```
public MediaContainer(String path)
public MediaContainer(URL url)
```

Constructs and initializes a new MediaContainer object using the specified path and forcing the cache data flag to `true`.

Methods

The Sound object has the following methods.

```
public final void setCacheEnable(boolean flag)
public final boolean getCacheEnable()
```

This parameter specifies whether this component contains a noncached reference to the sound data or explicit cached sound data.

```
public final void setURL(String path)
public final void setURL(URL url)
public final String getURL()
```

This parameter specifies the string path (URL) of the sound data associated with this component.

7.1.15 AuralAttributes Object

The AuralAttributes object is a component object of a Soundscape node that defines environmental audio parameters that affect sound rendering. These attributes include gain scale factor, atmospheric rolloff, and parameters controlling reverberation, distance frequency filtering, and velocity-activated Doppler effect.

7.1.15.1 Reverberation

Within Java 3D's simple model for auralization, there are three components to sound reverberation for a particular listening space:

- **Delay time**: Approximates the time from the start of a sound until it reaches the listener after reflecting once off the surfaces in the region.
- **Reflection coefficient**: Attenuates the reverberated sound uniformly (for all frequencies) as it bounces off surfaces.
- **Feedback loop**: Controls the maximum number of times a sound is reflected off the surfaces.

None of these parameters are affected by sound position. Figure 7-2 shows the interaction of these parameters.

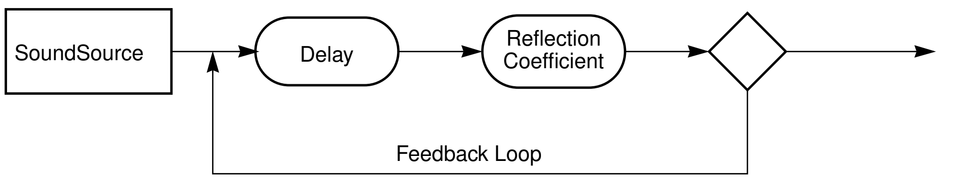


Figure 7-2 Sound Reverberation Parameters

The reflection coefficient for reverberation is a single scale factor used to approximate the overall reflective or absorptive characteristics of the surfaces in a reverberation region in which the listener is located. This scale factor is applied to the sound's amplitude regardless of the sound's position. A value of 1.0 represents complete (unattenuated) sound reflection, while a value of 0.0 represents full absorption (reverberation is disabled).

The reverberation delay time is set either explicitly (in milliseconds), or implicitly by supplying an additional bounds volume (so the delay time can be calculated). The bounds of the reverberation space do not have to be the same as the application region of the Soundscape node using this object.

The reverberation order defines the number of reverberation (feedback) loop iterations to be executed while a sound is played. As long as the reflection coefficient is small enough, the reverberated sound decreases (as it would naturally) each successive iteration. A value of 0 disables reverberation, a value of 1 creates a single echo (given that the reverb delay is long enough), and a value of −1 signifies that reverberation is to loop until it reaches an amplitude of *effective zero* (>60 dB or 1/1000 of sound amplitude). All other positive values are used as the number of loop iterations.

7.1.15.2 Doppler Effect

Doppler effect can be used to create a greater sense of movement of sound sources, and can help unambiguate front-back localization errors. The frequency of sound waves emanating from the source are lowered based on the speed of the source in relation to the listener, and the sound's wavelength.

The Doppler scale factor can be used to increase or reduce the change of frequency associated with normal Doppler calculation. To create this Doppler effect, the relative velocity (change in distance in the local coordinate system between the sound source and the listener over time, in meters per second) must be specified. This is nonzero even if the listener is moving but the sound is not.

Constants

The AuralAttributes object has the following flags.

```
public static final int ALLOW_ATTRIBUTE_GAIN_READ
public static final int ALLOW_ATTRIBUTE_GAIN_WRITE
public static final int ALLOW_ROLLOFF_READ
public static final int ALLOW_ROLLOFF_WRITE
public static final int ALLOW_REFLECTION_COEFFICIENT_READ
public static final int ALLOW_REFLECTION_COEFFICIENT_WRITE
public static final int ALLOW_REVERB_DELAY_READ
public static final int ALLOW_REVERB_DELAY_WRITE
public static final int ALLOW_REVERB_ORDER_READ
public static final int ALLOW_REVERB_ORDER_WRITE
public static final int ALLOW_DISTANCE_FILTER_READ
public static final int ALLOW_DISTANCE_FILTER_WRITE
public static final int ALLOW_DOPPLER_SCALE_FACTOR_READ
public static final int ALLOW_DOPPLER_SCALE_FACTOR_WRITE
public static final int ALLOW_DOPPLER_VELOCITY_READ
public static final int ALLOW_DOPPLER_VELOCITY_WRITE
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read or write the associated parameters.

Constructors

The AuralAttributes object has the following constructors.

```
public AuralAttributes()
```

Constructs and initializes a new AuralAttributes object using the following default values:

attribute gain: 1.0
rolloff: 1.0
reflection coefficient: 0.0
reverb delay: 0.0
reverb order: 0
distance filtering: null (no filtering performed)
Doppler scale factor: 1.0
Doppler velocity: 0.0

```
public AuralAttributes(float gain, float rolloff,
       float reflectionCoefficient, float reverbDelay,
       int reverbOrder, Point2f distanceFilter[],
       float dopplerScaleFactor, float dopplerVelocity)
public AuralAttributes(float gain, float rolloff,
       float reflectionCoefficient, float reverbDelay,
       int reverbOrder, float distance[], float frequencyCutoff,
       float dopplerScaleFactor, float dopplerVelocity)
```

Construct and initialize a new AuralAttributes object using the specified parameters.

Methods

The AuralAttributes object has the following methods.

```
public final void setAttributeGain(float gain)
public final float getAttributeGain()
```

This parameter specifies an amplitude scale factor applied to the sound. Valid values are ≥ 0.0.

```
public final void setRolloff(float rolloff)
public final float getRolloff()
```

This scale factor is used to model simple atmospheric conditions that affect the speed of sound. This affects the time a sound takes to reach the listener after it has begun playing. The normal speed of sound is scaled by this single rolloff scale factor, thus increasing or decreasing the usual attenuation. Valid values are

≥ 0.0. Values > 1.0 increase the speed of sound, while values < 1.0 decrease its speed.

```
public final void setReflectionCoefficient(float reflectionCoeff)
public final float getReflectionCoefficient()
```

This parameter specifies an average amplitude scale factor for all sound waves (independent of their frequencies) as they reflect off all surfaces within the activation region in which the listener is located. There is currently no method to assign different reflective audio properties to individual surfaces. The range of values is 0.0 to 1.0. A value of 0.0 represents a fully absorptive surface (no sound waves reflect off), while a value of 1.0 represents a fully reflective surface (amplitudes of sound waves reflecting off surfaces are not decreased).

```
public final void setReverbDelay(float reverbDelay)
public final void setReverbDelay(Bounds reverbVolume)
public final float getReverbDelay()
```

This parameter specifies the delay time between each order of reflection while reverberation is being rendered. In the first form of `setReverbDelay`, an explicit delay time is given in milliseconds. In the second form, a reverberation bounds volume is specified, and then the delay time is calculated, becoming the new reverb time delay. A value of 0.0 for delay time disables reverberation.

```
public final void setReverbOrder(int reverbOrder)
public final int getReverbOrder()
```

This parameter specifies the maximum number of times reflections will be added to the reverberation being calculated. When the amplitude of the *n*-th reflection reaches effective zero, no further reverberations need be added to the sound image. A value of 0 disables reverberation. A value of −1 specifies that the reverberation calculations will loop indefinitely, until the *n*-th reflection term reaches effective zero.

```
public final void setDistanceFilter(Point2f attenuation[])
public final void setDistanceFilter(float distance[],
       float frequencyCutoff[])
public final int getDistanceFilterLength()
public final void getDistanceFilter(Point2f attenuation[])
public final void getDistanceFilter(float distance[],
       float frequencyCutoff[])
```

This parameter specifies a (distance, filter) attenuation pairs array. If this is not set, no distance filtering is performed (equivalent to using a distance filter of `Sound.NO_FILTER` for all distances). Currently, this filter is a low-pass cutoff frequency. This array of pairs defines a piecewise linear slope for a range of values. This attenuation array is similar to the PointSound node's `distanceAttenuation` pair array, except that frequency values are paired with distances in this list. Using these pairs, distance-based low-pass frequency filtering can be applied during sound rendering. Distances, specified in the local coordinate system in meters, must be > 0. Frequencies (in Hz) must be > 0.

If the distance from the listener to the sound source is less than the first distance in the array, the first filter is applied to the sound source. This creates a spherical region around the listener within which a sound is uniformly attenuated by the first filter in the array. If the distance from the listener to the sound source is greater than the last distance in the array, the last filter is applied to the sound source.

The first form of `setDistanceFilter` takes these pairs of values as an array of Point2f. The second form accepts two separate arrays for these values. The `distance` and `frequencyCutoff` arrays should be of the same length. If the `frequencyCutoff` array length is greater than the `distance` array length, the `frequencyCutoff` array elements beyond the length of the `distance` array are ignored. If the `frequencyCutoff` array is shorter than the `distance` array, the last `frequencyCutoff` array value is repeated to fill an array of length equal to the `distance` array.

The `getDistanceFilterLength` method returns the length of the distance filter arrays. Arrays passed into `getDistanceFilter` methods should all be at least this size.

There are two methods for `getDistanceFilter`, one returning an array of points, the other returning separate arrays for each attenuation component.

Distance elements in this array of pairs are a monotonically increasing set of floating-point numbers measured from the location of the sound source. Frequency cutoff elements in this list of pairs can be any positive float. While for most applications this list of values will usually be monotonically decreasing, they do not have to be.

```
public final void setDopplerScaleFactor(float
       frequencyScaleFactor)
public final float getDopplerScaleFactor()
```

This parameter specifies a scale factor is used to increase or decrease the change of frequency resulting from the Doppler effect calculated during sound rendering. This allows the application to exaggerate or reduce the change in frequency normally resulting from applying the standard Doppler equation to the sound. Valid values are ≥ 0.0. A value of 0.0 disables any Doppler calculation.

```
public final void setDopplerVelocity(float velocityScaleFactor)
public final float getDopplerVelocity()
```

This parameter specifies a scale factor applied to the *relative velocity* (change in distance in the local coordinate system between the sound source and the listener over time) automatically calculated by the Doppler equation during sound rendering. This allows the application to exaggerate or reduce the relative velocity calculated by the standard Doppler equation. Valid values are ≥ 0.0. A value of 0.0 disables any Doppler calculation.

7.1.16 ImageComponent Object

The ImageComponent classes are used for texture and background images. The ImageComponent object is an abstract class that is used to define 2D or 3D ImageComponent classes.

Constants

The ImageComponent object has the following flags:

```
public static final int ALLOW_SIZE_READ
public static final int ALLOW_FORMAT_READ
public static final int ALLOW_IMAGE_READ
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read the associated parameters.

The ImageComponent object specifies the following variables, used to define 2D or 3D ImageComponent classes. These variables specify the format of the pixel data.

```
public final static int FORMAT_RGB
```

Specifies that each pixel contains three eight-bit channels, one each for red, green, and blue. This is the same as `FORMAT_RGB8`.

```
public final static int FORMAT_RGBA
```

Specifies that each pixel contains four eight-bit channels, one each for red, green, blue, and alpha. This is the same as `FORMAT_RGBA8`.

public final static int FORMAT_RGB8

Specifies that each pixel contains three eight-bit channels, one each for red, green, and blue. This is the same as FORMAT_RGB.

public final static int FORMAT_RGBA8

Specifies that each pixel contains four eight-bit channels, one each for red, green, blue, and alpha. This is the same as FORMAT_RGBA.

public final static int FORMAT_RGB5

Specifies that each pixel contains three five-bit channels, one each for red, green, and blue.

public final static int FORMAT_RGB5_A1

Specifies that each pixel contains three five-bit channels, one each for red, green, and blue, and a one-bit channel for alpha.

public final static int FORMAT_RGB4

Specifies that each pixel contains three four-bit channels, one each for red, green, and blue.

public final static int FORMAT_RGBA4

Specifies that each pixel contains four four-bit channels, one each for red, green, blue, and alpha.

public final static int FORMAT_LUM4_ALPHA4

Specifies that each pixel contains two four-bit channels, one each for luminance and alpha.

public final static int FORMAT_LUM8_ALPHA8

Specifies that each pixel contains two eight-bit channels, one each for luminance and alpha.

public static final int FORMAT_R3_G3_B2

Specifies that each pixel contains two three-bit channels, one each for red and green, and a two-bit channel for blue.

```
public static final int FORMAT_CHANNEL8
```

Specifies that each pixel contains one eight-bit channel. The channel can be used for only luminance, alpha, or intensity.

Constructors

The ImageComponent object defines the following constructor.

```
public ImageComponent(int format, int width, int height)
```

This constructor constructs and initializes a new ImageComponent object.

Methods

The ImageComponent object defines the following methods.

```
public final int getWidth()
public final int getHeight()
public final int getFormat()
```

These methods retrieve the width, height, and format of this image component object.

7.1.17 ImageComponent2D Object

The ImageComponent2D class defines a 2D array of pixels, used for texture and background images.

Constructors

The ImageComponent2D object defines the following constructors.

```
public ImageComponent2D(int format, int width, int height)
public ImageComponent2D(int format, BufferedImage image)
```

The first constructor constructs and initializes a 2D image component object using the specified format, width, and height. The second constructor constructs and initializes a 2D image component object using the specified format and buffered image. A copy of the image is made.

Methods

The ImageComponent2D object defines the following methods.

```
public void set(BufferedImage image)
```

This method copies the specified buffered image to this 2D image component object.

Note: The image must be completely loaded before calling this function.

```
public final BufferedImage getImage()
```

This method retrieves a copy of the image in this ImageComponent2D object.

7.1.18 ImageComponent3D Object

The ImageComponent3D class defines a 3D array of pixels, used for texture images.

Constructors

The ImageComponent3D object defines the following constructors.

```
public ImageComponent3D(int format, int width, int height,
       int depth)
public ImageComponent3D(int format, BufferedImage image[])
```

The first constructor constructs and initializes a 3D image component object using the specified format, width, height, and depth. The second constructor constructs and initializes a 3D image component object using the specified format and the buffered image array.

Methods

The ImageComponent3D object defines the following methods.

```
public final int getDepth()
```

This method retrieves the depth of this 3D image component object.

```
public final BufferedImage[] getImage()
public final BufferedImage getImage(int index)
```

These methods retrieve a copy of the images in this ImageComponent3D object.

```
public final void set(BufferedImage images[])
public final void set(int index, BufferedImage image)
```

The first method copies the specified array of BufferedImage objects to this 3D image component object. The second method copies the specified BufferedImage object to this 3D image component object at the specified index.

7.1.19 DepthComponent Object

The DepthComponent object is an abstract base class that defines a 2D array of depth (Z) values.

Constants

The DepthComponent object has the following flags:

```
public static final int ALLOW_SIZE_READ
public static final int ALLOW_DATA_READ
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read the associated parameters.

Methods

```
public int getWidth()
public int getHeight()
```

These methods get the width and height of this object.

7.1.20 DepthComponentFloat Object

The DepthComponentFloat object extends the DepthComponent object and defines a 2D array of depth (Z) values in floating-point format in the range [0, 1]. A value of 0.0 indicates the closest Z value to the user, while a value of 1.0 indicates the farthest Z value.

Constructors

The DepthComponentFloat object defines the following constructors.

```
public DepthComponentFloat(int width, int height)
```

Constructs a new floating-point depth (Z-buffer) component object with the specified width and height.

Methods

```
public void setDepthData(float depthData[])
public void getDepthData(float depthData[])
```

These methods set and retrieve the specified depth data for this object.

7.1.21 DepthComponentInt Object

The DepthComponentInt object extends the DepthComponent object and defines a 2D array of depth (*Z*) values in integer format. Values are in the range [0, $(2^n) - 1$], where *n* is the *Z*-buffer pixel depth.

Constructors

The DepthComponentInt object defines the following constructor.

```
public DepthComponentInt(int width, int height)
```

Constructs a new integer depth (*Z*-buffer) component object with the specified width and height.

Methods

```
public void setDepthData(int depthData[])
public void getDepthData(int depthData[])
```

These methods set and retrieve the specified depth data for this object.

7.1.22 DepthComponentNative Object

The DepthComponentNative object extends the DepthComponent object and defines a 2D array of depth (*Z*) values stored in the most efficient format for a particular device. Values are not accessible by the user and may only be used to read the *Z* values and subsequently write them back.

Constructors

The DepthComponentNative object defines the following constructor.

```
public DepthComponentNative(int width, int height)
```

Constructs a new native depth (*Z*-buffer) component object with the specified width and height.

7.1.23 Bounds Object

Bounds objects define three varieties of containing volumes. Java 3D uses these containing volumes to support various culling operations. The types of bounds include an axis-aligned-box volume, a spherical volume, and a bounding polytope.

Methods

The Bounds object defines the following methods.

```
public abstract Object clone()
```

Clone this object.

```
public abstract void set(Bounds boundsObject)
```

This method sets the value of this Bounds object to enclose the specified bounding object.

```
public abstract boolean intersect(Point3d origin,
       Point3d direction)
public abstract boolean intersect(Point3d point)
public abstract boolean intersect(Bounds boundsObject)
public abstract boolean intersect(Bounds boundsObjects[])
```

These methods test for the intersection of this Bounds object with a ray, a point, another Bounds object, or an array of Bounds objects, respectively.

```
public abstract Bounds closestIntersection(Bounds boundsObjects[])
```

This method finds the closest bounding object that intersects this bounding object.

```
public abstract void combine(Bounds boundsObject)
public abstract void combine(Bounds boundsObjects[])
public abstract void combine(Point3d point)
public abstract void combine(Point3d points[])
```

These methods combine this Bounds object with a bounding object, an array of bounding objects, a point, or an array of points, respectively.

```
public abstract void transform(Bounds bounds, Transform3D trans)
public abstract void transform(Transform3D trans)
```

The first method tranforms a Bounds object so that it bounds a volume that is the result of transforming the given bounding object by the given transform. The second method transforms the Bounds object by the given transform.

```
public abstract boolean isEmpty()
```

This method tests whether the bounds is empty. A bounds is empty if it is `null` (either by construction or as the result of a null intersection) or if its volume is negative. A bounds with a volume of zero is *not* empty.

7.1.24 BoundingBox Object

BoundingBox objects are axis-aligned bounding box volumes.

Constructors

The BoundingBox object defines the following constructors.

```
public BoundingBox()
public BoundingBox(Point3d lower, Point3d upper)
public BoundingBox(Bounds boundsObject)
public BoundingBox(Bounds bounds[])
```

The first constructor constructs and initializes a 2X unity BoundingBox about the origin. The second constructor constructs and initializes a BoundingBox from the given minimum and maximum in x, y, and z. The third constructor constructs and initializes a BoundingBox from a bounding object. The fourth constructor constructs and initializes a BoundingBox from an array of bounding objects.

Methods

The BoundingBox object defines the following methods.

```
public void getLower(Point3d p1)
public void setLower(Point3d p1)
public void setLower(double xmin, double ymin, double zmin)
```

This parameter specifies the lower corner of this bounding box.

```
public void getUpper(Point3d p1)
public void setUpper(Point3d p1)
public void setUpper(double xmax, double ymax, double zmax)
```

This parameter specifies the upper corner of this bounding box.

```
public void set(Bounds boundsObject)
```

Sets the value of this bounding region to enclose the specified bounding object.

```
public Object clone()
```

Creates a copy of this bounding box.

```
public void combine(Bounds boundsObject)
public void combine(Bounds boundsObjects[])
public void combine(Point3d point)
public void combine(Point3d points[])
```

These methods combine this bounding box with a bounding object, an array of bounding objects, a point, or an array of points, respectively.

```
public void transform(Bounds boundsObject, Transform3D matrix)
public void transform(Transform3D matrix)
```

The first method transforms a bounding box so that it bounds a volume that is the result of transforming the given bounding object by the given transform. The second method transforms the bounding box by the given transform.

```
public boolean intersect(Point3d origin, Vector3d direction)
public boolean intersect(Point3d point)
public boolean intersect(Bounds boundsObject)
public boolean intersect(Bounds boundsObjects[])
```

These methods test for the intersection of this bounding box with a ray, a point, another Bounds object, and an array of Bounds objects, respectively.

```
public boolean intersect(Bounds boundsObject,
       BoundingBox newBoundBox)
public boolean intersect(Bounds boundsObjects[],
       BoundingBox newBoundBox)
```

These methods compute a new BoundingBox that bounds the volume created by the intersection of this BoundingBox with another Bounds object or array of Bounds objects.

```
public Bounds closestIntersection(Bounds boundsObjects[])
```

This method finds the closest bounding object that intersects this bounding box.

```
public boolean isEmpty()
```

This method tests whether the bounding box is empty. A bounding box is empty if it is `null` (either by construction or as the result of a null intersection) or if its volume is negative. A bounding box with a volume of zero is *not* empty.

7.1.25 BoundingSphere Object

The BoundingSphere object defines a spherical bounding volume. It has two associated values: the center point and the radius of the sphere.

Constructors

The BoundingSphere object defines the following constructors.

```
public BoundingSphere()
public BoundingSphere(Point3D center, double radius)
public BoundingSphere(Bounds boundsObject)
public BoundingSphere(Bounds boundsObjects[])
```

The first constructor constructs and initializes a BoundingSphere to unity (radius = 1.0 and center at 0.0, 0.0, 0.0). The second constructor constructs and initializes a BoundingSphere from a center and radius. The third constructor constructs and initializes a BoundingSphere from a bounding object. The fourth constructor constructs and initializes a BoundingSphere from an array of bounding objects.

Methods

The BoundingSphere object defines the following methods.

```
public double getRadius()
public void setRadius(double r)
```

This parameter specifies the bounding sphere radius.

```
public void getCenter(Point3d center)
public void setCenter(Point3d center)
```

This parameter defines the position of the bounding sphere.

```
public void set(Bounds boundsObject)
```

Sets the value of this bounding sphere to enclose the volume specified by the Bounds object.

```
public Object clone()
```

Creates a copy of the bounding sphere.

```
public void combine(Bounds boundsObject)
public void combine(Bounds boundsObjects[])
public void combine(Point3d point)
public void combine(Point3d points[])
```

These methods combine this bounding sphere with a bounding object, an array of bounding objects, a point, or an array of points, respectively.

```
public boolean intersect(Point3d origin, Point3d direction)
public boolean intersect(Point3d point)
public boolean intersect(Bounds boundsObject)
public boolean intersect(Bounds boundsObjects[])
```

These methods test for the intersection of this bounding sphere with the given ray, point, another Bounds object, or an array of Bounds objects.

```
public boolean intersect(Bounds boundsObject,
       BoundingSphere newBoundSphere)
public boolean intersect(Bounds boundsObjects[],
       BoundingSphere newBoundSphere)
```

These methods compute a new BoundingSphere that bounds the volume created by the intersection of this BoundingSphere with another Bounds object or array of Bounds objects.

```
public Bounds closestIntersection(Bounds boundsObjects[])
```

This method finds the closest bounding object that intersects this bounding sphere.

```
public void transform(Bounds boundsObject, Transform3D matrix)
public void transform(Transform3D matrix)
```

The first method transforms a bounding sphere so that it bounds a volume that is the result of transforming the given bounding object by the given transform. The second method transforms the bounding sphere by the given transform. Note that when transforming a bounding sphere by a transformation matrix containing a nonuniform scale or a shear, the result is a bounding sphere with a radius equal to the maximal scale in any direction—the bounding sphere does not transform into an ellipsoid.

```
public String toString()
```

This method returns a string representation of this class.

```
public boolean isEmpty()
```

This method tests whether the bounding sphere is empty. A bounding sphere is empty if it is `null` (either by construction or as the result of a null intersection) or if its volume is negative. A bounding sphere with a volume of zero is *not* empty.

7.1.26 BoundingPolytope Object

A BoundingPolytope object defines a set of planes that prescribe a convex, closed polygonal bounding region.

Constructors

The BoundingPolytope object defines the following constructors.

```
public BoundingPolytope()
public BoundingPolytope(Vector4d planes[])
public BoundingPolytope(Bounds boundsObject)
public BoundingPolytope(Bounds boundsObjects[])
```

The first constructor constructs a new BoundingPolytope object and initializes it to a cube where $-1 = x,y,z \leq 1$. The second constructor constructs and initializes a BoundingPolytope from an array of bounding planes. The third constructor constructs and initializes a BoundingPolytope from a Bounds object. The final constructor constructs and initializes a BoundingPolytope from an array of Bounds objects.

Methods

The BoundingPolytope object defines the following methods.

```
public void setPlanes(Vector4d planes[])
public void getPlanes(Vector4d planes[])
```

These methods set and retrieve the bounding planes for this BoundingPolytope object.

```
public int getNumPlanes()
```

This method returns the number of bounding planes for this bounding polytope.

```
public void set(Bounds boundsObject)
```

This method sets the planes for this BoundingPolytope by keeping its current number and direction of the planes and computing new plane positions to enclose the given Bounds object.

```
public Object clone()
```

This method creates a copy of the BoundingPolytope object.

```
public void combine(Bounds boundsObject)
public void combine(Bounds boundsObjects[])
public void combine(Point3d point)
public void combine(Point3d points[])
```

These methods combine this BoundingPolytope with a bounding object, an array of bounding objects, a point, or an array of points, respectively.

```
public void transform(Bounds bounds, Transform3D matrix)
public void transform(Transform3D matrix)
```

The first method tranforms a bounding polytope so that it bounds a volume that is the result of transforming the given bounding object by the given transform. The second method transforms the bounding polytope by the given transform.

```
public boolean intersect(Point3d origin, Vector3d direction)
public boolean intersect(Point3d point)
public boolean intersect(Bounds boundsObject)
public boolean intersect(Bounds boundsObjects[])
```

These methods test for the intersection of this BoundingPolytope with the given ray, point, another Bounds object, or array of Bounds objects, respectively.

```
public boolean intersect(Bounds boundsObject,
       BoundingPolytope newBoundPolytope)
public boolean intersect(Bounds boundsObjects[],
       BoundingPolytope newBoundPolytope)
```

These methods compute a new BoundingPolytope that bounds the volume created by the intersection of this BoundingPolytope with another Bounds object or array of Bounds objects.

```
public Bounds closestIntersection(Bounds boundsObjects[])
```

This method finds the closest bounding object that intersects this bounding polytope.

```
public boolean isEmpty()
```

This method tests whether the bounding polytope is empty. A bounding polytope is empty if it is `null` (either by construction or as the result of a null intersection) or if its volume is negative. A bounding polytope with a volume of zero is *not* empty.

7.1.27 Transform3D Object

Transformations are represented by matrix multiplication and include such operations as rotation, scaling, and translation. The Transform3D object is represented internally as a 4×4 double-precision floating point matrix. The mathematical representation is row major, as in traditional matrix mathematics.

Constants

```
public static final int ZERO
public static final int IDENTITY
public static final int SCALE
public static final int TRANSLATION
public static final int ORTHOGONAL
public static final int RIGID
public static final int CONGRUENT
public static final int AFFINE
public static final int NEGATIVE_DETERMINANT
```

A Transform3D has an associated type that is internally computed when the transform object is constructed and updated any time it is modified. A matrix will typically have multiple types. For example, the type associated with an identity matrix is the result of ORing all of the types, except for `ZERO` and `NEGATIVE_DETERMINANT`, together. There are public methods available to get the ORed type of the transformation, the sign of the determinant, and the least general matrix type. The matrix type flags are defined as follows:

- ZERO: Zero matrix.
- IDENTITY: Identity matrix.
- SCALE: This matrix is a uniform scale matrix—there are no rotational or translation components.

- TRANSLATION: This matrix has translation components only. The scale is unity and there are no rotational components.
- ORTHOGONAL: The four row vectors that make up an orthogonal matrix form a basis, meaning that they are mutually orthogonal. The scale is unity and there are no translation components.
- RIGID: The upper 3×3 of the matrix is orthogonal, and there is a translation component—the scale is unity.
- CONGRUENT: This is an angle- and length-preserving matrix, meaning that it can translate, rotate, and reflect about an axis, and scale by an amount that is uniform in all directions. These operations preserve the distance between any two points, and the angle between any two intersecting lines.
- AFFINE: An affine matrix can translate, rotate, reflect, scale anisotropically, and shear. Lines remain straight, and parallel lines remain parallel, but the angle between intersecting lines can change.

A matrix is also classified by the sign of its determinant:

- NEGATIVE_DETERMINANT: This matrix has a negative determinant. An orthogonal matrix with a positive determinant is a rotation matrix. An orthogonal matrix with a negative determinant is a reflection and rotation matrix.

The Java 3D model for 4×4 transformations is

$$\begin{bmatrix} m00 & m01 & m02 & m03 \\ m10 & m11 & m12 & m13 \\ m20 & m21 & m22 & m23 \\ m30 & m31 & m32 & m33 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

$$\begin{aligned} x' &= m00 \cdot x + m01 \cdot y + m02 \cdot z + m03 \cdot w \\ y' &= m10 \cdot x + m11 \cdot y + m12 \cdot z + m13 \cdot w \\ z' &= m20 \cdot x + m21 \cdot y + m22 \cdot z + m23 \cdot w \\ w' &= m30 \cdot x + m31 \cdot y + m32 \cdot z + m33 \cdot w \end{aligned}$$

Note: When transforming a Point3f or a Point3d, the input *w* is set to 1. When transforming a Vector3f or Vector3d, the input *w* is set to 0.

Constructors

The Transform3D object defines the following constructors.

```
public Transform3D()
```

This constructs and initializes a new Transform3D object to the identity transformation.

```
public Transform3D(Transform3D t1)
```

This constructs and initializes a new Transform3D object from the specified transform.

```
public Transform3D(Matrix3f m1, Vector3d t1, double s)
public Transform3D(Matrix3d m1, Vector3d t1, double s)
public Transform3D(Matrix3f m1, Vector3f t1, float s)
```

These construct and initialize a new Transform3D object from the rotation matrix, translation, and scale values. The scale is applied only to the rotational component of the matrix (upper 3×3) and not to the translational components of the matrix.

```
public Transform3D(Matrix4f m1)
public Transform3D(Matrix4d m1)
```

These construct and initialize a new Transform3D object from the 4×4 matrix. The type of the constructed transform is classified automatically.

```
public Transform3D(float matrix[])
public Transform3D(double matrix[])
```

These construct and initialize a new Transform3D object from the array of length 16. The top row of the matrix is initialized to the first four elements of the array, and so on. The type of the constructed transform is classified automatically.

```
public Transform3D(Quat4d q1, Vector3d t1, double s)
public Transform3D(Quat4f q1, Vector3d t1, double s)
public Transform3D(Quat4f q1, Vector3f t1, float s)
```

These construct and initialize a new Transform3D object from the quaternion `q1`, the translation `t1`, and the scale `s`. The scale is applied only to the rotational components of the matrix (the upper 3×3) and not to the translational components of the matrix.

`public Transform3D(GMatrix m1)`

This constructs and initializes a new Transform3D object and initializes it to the upper 4×4 of the specified GMatrix. If the specified matrix is smaller than 4×4, the remaining elements in the transformation matrix are assigned to zero.

Methods

The Transform3D object defines the following methods.

`public final int getType()`

This method retrieves the type of this matrix. The type is an ORed bitmask of all of the type classifications to which it belongs.

`public final int getBestType()`

This method retrieves the least general type of this matrix. The order of generality from least to most is as follows: `ZERO`, `IDENTITY`, `SCALE`, `TRANSLATION`, `ORTHOGONAL`, `RIGID`, `CONGRUENT`, and `AFFINE`. If the matrix is `ORTHOGONAL`, calling the method `getDeterminantSign` will yield more information.

`public final void setAutoNormalize(boolean autoNormalize)`
`public final boolean getAutoNormalize()`

These methods set and retrieve the state of autonormalization. Autonormalization performs an automatic singular value decomposition (SVD) normalization of the rotational components (upper 3×3) of this matrix after every subsequent matrix operation on this object, unless the boolean is subsequently set to `false`. The default value for this parameter is `false`.

`public final boolean getDeterminantSign()`

This method returns the sign of the determinant of this matrix. A return value of `true` indicates a positive determinant. A return value of `false` indicates a negative determinant. In general, an orthogonal matrix with a positive determinant is a pure rotation matrix; an orthogonal matrix with a negative determinant is both a rotation and a reflection matrix.

`public final void setIdentity()`

This method sets this transform to the identity matrix.

`public final void setZero()`

This method sets this transform to all zeros.

`public final void setEuler(Vector3d euler)`

This method sets the rotational component (upper 3×3) of this transform to the rotation matrix converted from the Euler angles provided. The `euler` parameter is a Vector3d consisting of roll, pitch, and yaw.

`public final void setRotation(Matrix3d m1)`
`public final void setRotation(Matrix3f m1)`

These methods set the rotational component (upper 3×3) of this transform to the values in the specified matrix; the other elements of this transform are unchanged. A singular value decomposition is performed on this object's upper 3×3 matrix to factor out the scale, then this object's upper 3×3 matrix components are replaced by the input rotational components, and finally the scale is reapplied to the rotational components.

`public final void setRotation(Quat4f q1)`
`public final void setRotation(Quat4d q1)`

These methods set the rotational component (upper 3×3) of this transform to the appropriate values derived from the specified quaternion; the other elements of this transform are unchanged. A singular value decomposition is performed on this object's upper 3×3 matrix to factor out the scale, then this object's upper 3×3 matrix components are replaced by the matrix equivalent of the quaternion, and finally the scale is reapplied to the rotational components.

`public final void setRotation(AxisAngle4d a1)`
`public final void setRotation(AxisAngle4f a1)`

These methods set the rotational component (upper 3×3) of this transform to the appropriate values derived from the specified axis-angle; the other elements of this transform are unchanged. A singular value decomposition is performed on this object's upper 3×3 matrix to factor out the scale, then this object's upper 3×3 matrix components are replaced by the matrix equivalent of the axis-angle, and finally the scale is reapplied to the rotational components.

`public final void setScale(double scale)`
`public final double getScale()`

The `set` method sets the scale component of this transform by factoring out the current scale from the rotational component and multiplying by the new scale. The `get` method performs an SVD normalization of this transform to calculate and return the scale factor; this transform is not modified.

public final void setScale(Vector3d scale)
public final void getScale(Vector3d scale)

The `set` method sets the possibly non-uniform scale component to the current transform. Any existing scale is first factored out of the existing transform before the new scale is applied. The `get` method returns the possibly non-uniform scale components of the current transform and places them into the scale vector.

public final void setNonUniformScale(double xScale, double yScale, double zScale)

This is a deprecated method. Use the `setScale(Vector3d)` method instead.

public final void scaleAdd(double s, Transform3D t1, Transform3D t2)
public final void scaleAdd(double s, Transform3D t1)

The first method scales transform `t1` by a uniform scale matrix with scale factor `s`, then adds transform `t2` (this = S * t1 + t2). The second method scales this transform by a uniform scale matrix with scale factor `s`, then adds transform t1 (this = S * this + t1).

public final void setRotationScale(Matrix3f m1)
public final void setRotationScale(Matrix3d m1)
public final void getRotationScale(Matrix3f m1)
public final void getRotationScale(Matrix3d m1)

The `set` methods replace the upper 3×3 matrix values of this transform with the values in the matrix `m1`. The `get` methods retrieve the upper 3×3 matrix values of this transform and place them in the matrix `m1`.

public String toString()

This method returns the matrix elements of this transform as a string.

public final void add(Transform3D t1)
public final void add(Transform3D t1, Transform3D t2)
public final void sub(Transform3D t1)
public final void sub(Transform3D t1, Transform3D t2)

The first `add` method adds this transform to the transform `t1` and places the result back into `this`. The second `add` method adds the transforms `t1` and `t2` and places the result into `this`. The first `sub` method subtracts transform `t1` from this transform and places the result back into `this`. The second `sub` method subtracts transform `t2` from `t1` and places the result into `this`.

```
public final void add(double scalar)
public final void add(double scalar, Transform3D t1)
```

The first method adds a scalar to each component of this transform. The second method adds a scalar to each component of the transform `t1` and places the result into `this`. Transform `t1` is not modified.

```
public final void transpose()
public final void transpose(Transform3D t1)
```

The first method transposes this matrix in place. The second method transposes transform `t1` and places the value into this transform. The transform t1 is not modified.

```
public void rotX(double angle)
public void rotY(double angle)
public void rotZ(double angle)
```

These three methods set the value of this matrix to a rotation matrix about the specified axis. The angle to rotate is specified in radians.

```
public final void setTranslation(Vector3f trans)
public final void setTranslation(Vector3d trans)
```

This method modifies the translational components of this transform to the values of the argument. The other values of this transform are not modified.

```
public final void set(Quat4f q1)
public final void set(Quat4d q1)
```

These methods set the value of this transform to the matrix conversion of the quaternion argument.

```
public final void set(Quat4d q1, Vector3d t1, double s)
public final void set(Quat4f q1, Vector3d t1, double s)
public final void set(Quat4f q1, Vector3f t1, float s)
```

These methods set the value of this matrix from the rotation expressed by the quaternion `q1`, the translation `t1`, and the scale `s`.

```
public final void set(Vector3d trans)
public final void set(Vector3f trans)
```

These methods set the translational value of this matrix to the specified vector parameter values and set the other components of the matrix as if this transform were an identity matrix.

```
public final void set(Vector3d v1, double scale)
public final void set(Vector3f v1, float scale)
```

These methods set the value of this transform to a scale and translation matrix; the translation is scaled by the scale factor and all of the matrix values are modified.

```
public final void set(Transform3D t1)
```

This method sets the matrix, type, and state of this transform to the matrix, type, and state of the transform `t1`.

```
public final void set(double matrix[])
public final void set(float matrix[])
```

These methods set the matrix values of this transform to the specified matrix values.

```
public final void set(double scale)
public final void set(double scale, Vector3d v1)
public final void set(float scale, Vector3f v1)
```

The first method sets the value of this transform to a uniform scale; all of the matrix values are modified. The next two methods set the value of this transform to a scale and translation matrix; the scale is not applied to the translation and all of the matrix values are modified.

```
public final void set(Matrix4d m1)
public final void set(Matrix4f m1)
```

These methods set the matrix values of this transform to the matrix values in the specified matrix.

```
public final void set(Matrix3f m1)
public final void set(Matrix3d m1)
```

These methods set the rotational and scale components (upper 3×3) of this transform to the matrix values in the specified matrix. The remaining matrix values are set to the identity matrix. All values of the matrix are modified.

```
public final void set(Matrix3f m1, Vector3f t1, float s)
public final void set(Matrix3f m1, Vector3d t1, double s)
public final void set(Matrix3d m1, Vector3d t1, double s)
```

These methods set the value of this matrix from the rotation expressed by the rotation matrix `m1`, the translation `t1`, and the scale `s`. The scale is only applied to the rotational component of the matrix (upper 3×3) and not to the translational component of the matrix.

`public final void set(GMatrix matrix)`

These methods set the matrix values of this transform to the matrix values in the specified matrix. The GMatrix object must specify a 4×4, 3×4, or 3×3 matrix.

`public final void set(AxisAngle4f a1)`
`public final void set(AxisAngle4d a1)`

These methods set the rotational component (upper 3×3) of this transform to the matrix conversion of the specified axis-angle argument. The remaining matrix values are set to the identity matrix. All values of the matrix are modified.

`public final void get(double matrix[])`
`public final void get(float matrix[])`

These methods place the values of this transform into the specified matrix of length 16. The first four elements of the array will contain the top row of the transform matrix, and so on.

`public final void get(Matrix4d matrix)`
`public final void get(Matrix4f matrix)`

These methods place the values of this transform into the `matrix` argument.

`public final void get(Matrix3d m1)`
`public final void get(Matrix3f m1)`

These methods place the normalized rotational component of this transform into the 3×3 matrix argument.

`public final double get(Matrix3d m1, Vector3d t1)`
`public final float get(Matrix3f m1, Vector3f t1)`
`public final double get(Matrix3f m1, Vector3d t1)`

These methods place the normalized rotational component of this transform into the `m1` parameter and the translational component into the `t1` parameter.

`public final void get(Quat4d q1)`
`public final void get(Quat4f q1)`

These methods perform an SVD normalization of this matrix to acquire the normalized rotational component. The values are placed into the quaternion `q1` parameter.

public final double get(Quat4d q1, Vector3d t1)
public final float get(Quat4f q1, Vector3f t1)
public final double get(Quat4f q1, Vector3d t1)

These methods perform an SVD normalization of this transform to calculate the rotation as a quaternion, the translation, and the scale. None of the matrix values are modified.

public final void get(Vector3d trans)
public final void get(Vector3f trans)

These methods retrieve the translational components of this transform.

public final void invert()
public final void invert(Transform3D t1)

The first method inverts this transform in place. The second method sets the value of this transform to the inverse of the transform `t1`. Both of these methods use the transform type to determine the optimal algorithm for inverting the transform.

public final double determinant()

This method calculates and returns the determinant of `this` transform.

public final void mul(Transform3D t1)
public final void mul(Transform3D t1, Transform3D t2)

The first method sets the value of this transform to the result of multiplying itself with transform `t1` (this = this * t1). The second method sets the value of this transform to the result of multiplying transform `t1` by transform `t2` (this = t1 * t2).

public final void mul(double scalar)
public final void mul(double scalar, Transform3D t1)

The first method multiplies this transform by the scalar constant. The second method multiplies transform t1 by the scalar constant and places the value into `this` transform.

public final void mulInverse(Transform3D t1)
public final void mulInverse(Transform3D t1, Transform3D t2)

The first method multiplies this transform by the inverse of transform `t1` and places the result into `this` transform (this = this * $t1^{-1}$). The second method multiplies transform `t1` by the inverse of transform `t2` and places the result into `this` transform (this = t1 * $t2^{-1}$).

public final void mulTransposeRight(Transform3D t1,Transform3D t2)
public final void mulTransposeLeft(Transform3D t1, Transform3D t2)
public final void mulTransposeBoth(Transform3D t1, Transform3D t2)

The first method multiplies the transform `t1` by the transpose of transform `t2` and places the result into this transform (this = t1 * transpose(t2)). The second method multiplies the transpose of transform `t1` by transform `t2` and places the result into `this` transform (this = transpose(t1) * t2). The third method multiplies the transpose of transform `t1` by the transpose of `t2` and places the result into `this` transform (this = transpose(t1) * transpose(t2)).

public final void normalize()
public final void normalize(Transform3D t1)

Both of these methods use an SVD normalization. The first `normalize` method normalizes the rotational components (upper 3×3) of matrix `this` and places the results back into `this`. The second `normalize` method normalizes the rotational components (upper 3×3) of transform `t1` and places the result in `this`.

public final void normalizeCP()
public final void normalizeCP(Transform3D t1)

Both of these methods use a cross-product (CP) normalization. The first `normalizeCP` method normalizes the rotational components (upper 3×3) of this transform and places the result into this transform. The second `normalizeCP` method normalizes the rotational components (upper 3×3 of transform `t1` and places the result into `this` transform.

public boolean equals(Transform3D t1)

This method returns `true` if all of the data members of transform `t1` are equal to the corresponding data members in `this` transform.

public boolean epsilonEquals(Transform3D t1, double epsilon)

This method returns `true` if the L_∞ distance between this transform and transform `m1` is less than or equal to the epsilon parameter; otherwise, it returns `false`. The L_∞ distance is equal to:

MAX[i=0,1,2,3 ; j=0,1,2,3 ; abs[(this.m(i,j) – m1.m(i,j)]

public int hashCode()

This method returns a hash number based on the data values in this object. Two different Transform3D objects with identical data values (that is, `true` is returned for `trans.equals(Transform3D)`) will return the same hash number. Two

Transform3D objects with different data members may return the same hash value, although this is not likely.

```
public final void transform(Vector4d vec, vector4d vecOut)
public final void transform(Vector4f vec, Vector4f vecOut)
public final void transform(Vector4d vec)
public final void transform(Vector4f vec)
```

The first two methods transform the vector `vec` by this transform and place the result into `vecOut`. The last two methods transform the vector `vec` by this transform and place the result back into `vec`.

```
public final void transform(Point3d point, Point3d pointOut)
public final void transform(Point3f point, point3f pointOut)
public final void transform(Point3d point)
public final void transform(Point3f point)
```

The first two methods transform the `point` parameter by this transform and place the result into `pointOut`. The last two methods transform the `point` parameter by this transform and place the result back into `point`. In both cases, the fourth element of the `point` input parameter is assumed to be 1.

```
public final void transform(Vector3d normal, Vector3d normalOut)
public final void transform(Vector3f normal, Vector3f normalOut)
public final void transform(Vector3d normal)
public final void transform(Vector3f normal)
```

The first two methods transforms the `normal` parameter by this transform and place the value into `normalOut`. The third and fourth methods transform the `normal` parameter by this transform and place the value back into `normal`.

7.1.27.1 View Model Compatibility Mode Methods: Viewing Matrix

```
public void lookAt(Point3d eye, Point3d center, Vector3d up)
```

This is a utility method that specifies the position and orientation of a viewing transformation. It works very much like the similar function in OpenGL. The inverse of this transform can be used to control the ViewPlatform object within the scene graph. Alternatively, this transform can be passed directly to the View's `VpcToEc` transform via the compatibility mode viewing functions defined in Section C.11.2, "Using the Camera-based View Model."

7.1.27.2 View Model Compatibility Mode Methods: Projection Matrix

```
public void frustum(double left, double right, double bottom,
       double top, double near, double far)
public void perspective(double fovx, double aspect, double zNear,
       double zFar)
public void ortho(double left, double right, double bottom,
       double top, double near, double far)
```

These three utility methods allow an application to create a perspective or parallel (orthographic) projection matrix. These three methods work very much like the similar functions in OpenGL. The resulting Transform3D can be used to directly set the View's left and right projection transforms when in compatibility mode. See Section C.11.2, "Using the Camera-based View Model," for details. The `fovx` parameter specifies the field of view in the *x* direction in radians.

7.2 Node Component Objects: Geometry

A Geometry object is an abstract class that specifies the geometry component information required by a Shape3D node. Geometry objects describe both the geometry and topology of the Shape3D nodes that reference them. Geometry objects consist of four generic geometric types: CompressedGeometry, GeometryArray, Raster, and Text3D (see Figure 7-3). Each of these geometric types defines a visible object or set of objects. A Geometry object is used as a component object of a Shape3D leaf node.

7.2.1 GeometryArray Object

A GeometryArray object is an abstract class from which several classes are derived to specify a set of geometric primitives. A GeometryArray contains separate arrays of the following vertex components: coordinates, colors, normals, and texture coordinates, and a bitmask indicating which of these components are present.

A single GeometryArray contains a predefined collection of per-vertex information; all of the vertices in a GeometryArray object have the same format and primitive type. Different GeometryArrays can contain different per-vertex information. One GeometryArray might contain only three-space coordinates; another might contain per-vertex coordinates, normals, colors, and texture coordinates; yet another might contain any subset of the previous example.

Constants

The GeometryArray object defines the following flags.

```
SceneGraphObject
    NodeComponent
        Geometry
            CompressedGeometry
            Raster
            Text3D
            GeometryArray
                GeometryStripArray
                    LineStripArray
                    TriangleStripArray
                    TriangleFanArray
                LineArray
                PointArray
                QuadArray
                TriangleArray
                IndexedGeometryArray
                    IndexedGeometryStripArray
                        IndexedLineStripArray
                        IndexedTriangleStripArray
                        IndexedTriangleFanArray
                    IndexedLineArray
                    IndexedPointArray
                    IndexedQuadArray
                    IndexedTriangleArray
```

Figure 7-3 Geometry Component Object Hierarchy

`public static final int ALLOW_COORDINATE_READ`
`public static final int ALLOW_COORDINATE_WRITE`

These flags specify that the GeometryArray object allows reading or writing of the array of coordinates.

`public static final int ALLOW_COLOR_READ`
`public static final int ALLOW_COLOR_WRITE`

These flags specify that the GeometryArray object allows reading or writing of the array of colors.

`public static final int ALLOW_NORMAL_READ`
`public static final int ALLOW_NORMAL_WRITE`

These flags specify that the GeometryArray object allows reading or writing of the array of normals.

```
public static final int ALLOW_TEXCOORD_READ
public final static int ALLOW_TEXCOORD_WRITE
```

These flags specify that the GeometryArray object allows reading or writing of the array of texture coordinates.

```
public final static int ALLOW_COUNT_READ
```

This flag specifies that the GeometryArray object allows reading any count data (such as the vertex count) associated with the GeometryArray.

Constructors

The GeometryArray object has the following constructor.

```
public GeometryArray(int vertexCount, int vertexFormat)
```

Constructs an empty GeometryArray object with the specified vertex format and number of vertices. The `vertexCount` parameter specifies the number of vertex elements in this array. The `vertexFormat` parameter is a mask indicating which vertex components are present in each vertex. The vertex format is specified as a set of flags that are bitwise ORed together to describe the per-vertex data. The following vertex formats are supported.

- COORDINATES: Specifies that this vertex array contains coordinates. This bit must be set.
- NORMALS: Specifies that this vertex array contains normals.
- COLOR_3: Specifies that this vertex array contains colors without alpha. Colors are specified as floating-point values in the range [0.0, 1.0].
- COLOR_4: Specifies that this vertex array contains colors with alpha. Colors are specified as floating-point values in the range [0.0, 1.0]. This takes precedence over `COLOR_3`.
- TEXTURE_COORDINATE_2: Specifies that this vertex array contains 2D texture coordinates (S and T).
- TEXTURE_COORDINATE_3: Specifies that this vertex array contains 3D texture coordinates (S, T, and R). This takes precedence over `TEXTURE_COORDINATE_2`.

Methods

GeometryArray methods provide access (`get` and `set` methods) to individual vertex component arrays in two different modes: as individual elements or as arrays of multiple elements.

public final int getVertexCount()

Retrieves the number of vertices in the GeometryArray.

public final int getVertexFormat()

Retrieves the vertex format of the GeometryArray.

```
public final void setCoordinate(int index, float coordinate[])
public final void getCoordinate(int index, float coordinate[])
public final void setCoordinate(int index, double coordinate[])
public final void getCoordinate(int index, double coordinate[])
```

Sets or retrieves the coordinate associated with the vertex at the specified index. The `coordinate` parameter is an array of three values containing the new coordinate.

```
public final void setCoordinate(int index, Point3f coordinate)
public final void getCoordinate(int index, Point3f coordinate)
public final void setCoordinate(int index, Point3d coordinate)
public final void getCoordinate(int index, Point3d coordinate)
```

Sets or retrieves the coordinate associated with the vertex at the specified index. The `coordinate` parameter is a point containing the new coordinate.

```
public final void setCoordinates(int index, float coordinates[])
public final void getCoordinates(int index, float coordinates[])
public final void setCoordinates(int index, double coordinates[])
public final void getCoordinates(int index, double coordinates[])
```

Sets or retrieves the coordinates associated with the vertices starting at the specified index. The `coordinates` parameter is an array of 3**n* values containing *n* new coordinates.

```
public final void setCoordinates(int index, Point3f coordinates[])
public final void getCoordinates(int index, Point3f coordinates[])
public final void setCoordinates(int index, Point3d coordinates[])
public final void getCoordinates(int index, Point3d coordinates[])
```

Sets or retrieves the coordinates associated with the vertices starting at the specified index. The `coordinates` parameter is an array of points containing new coordinates.

```
public final void setCoordinates(int index, Point3d coordinates[],
       int start, int length)
public final void setCoordinates(int index, Point3f coordinates[],
       int start, int length)
```

```
public final void setCoordinates(int index, float coordinates[],
       int start, int length)
public final void setCoordinates(int index, double coordinates[],
       int start, int length)
```

These methods set the coordinates associated with the vertices starting at the specified index for this object, using coordinate data starting from vertex index `start` for `length` vertices.

```
public final void setColor(int index, float color[])
public final void getColor(int index, float color[])
public final void setColor(int index, byte color[])
public final void getColor(int index, byte color[])
```

Sets or retrieves the color associated with the vertex at the specified index. The `color` parameter is an array of three or four values containing the new color.

```
public final void setColor(int index, Color3f color)
public final void getColor(int index, Color3f color)
public final void setColor(int index, Color4f color)
public final void getColor(int index, Color4f color)
public final void setColor(int index, Color3b color)
public final void getColor(int index, Color3b color)
public final void setColor(int index, Color4b color)
public final void getColor(int index, Color4b color)
```

Sets or retrieves the color associated with the vertex at the specified index. The `color` parameter is an array containing the new color.

```
public final void setColors(int index, float colors[])
public final void getColors(int index, float colors[])
public final void setColors(int index, byte colors[])
public final void getColors(int index, byte colors[])
```

Sets or retrieves the colors associated with the vertices starting at the specified index. The `colors` parameter is an array of 3**n* or 4**n* values containing *n* new colors.

```
public final void setColors(int index, Color3f colors[])
public final void getColors(int index, Color3f colors[])
public final void setColors(int index, Color4f colors[])
public final void getColors(int index, Color4f colors[])
public final void setColors(int index, Color3b colors[])
public final void getColors(int index, Color3b colors[])
```

public final void setColors(int index, Color4b colors[])
public final void getColors(int index, Color4b colors[])

Sets or retrieves the colors associated with the vertices starting at the specified index. The `colors` parameter is an array containing the new colors.

public final void setColors(int index, float colors[], int start, int length)
public final void setColors(int index, byte colors[], int start, int length)
public final void setColors(int index, Color3f colors[], int start, int length)
public final void setColors(int index, Color4f colors[], int start, int length)
public final void setColors(int index, Color3b colors[], int start, int length)
public final void setColors(int index, Color4b colors[], int start, int length)

These methods set the colors associated with the vertices starting at the specified index for this object, using data in `colors` starting at index `start` for `length` colors.

public final void setNormal(int index, float normal[])
public final void getNormal(int index, float normal[])

Sets or retrieves the normal associated with the vertex at the specified index. The `normal` parameter is an array of three values containing the new normal.

public final void setNormal(int index, Vector3f normal)
public final void getNormal(int index, Vector3f normal)

Sets or retrieves the normal associated with the vertex at the specified index. The `normal` parameter is a vector containing the new normal.

public final void setNormals(int index, float normals[])
public final void getNormals(int index, float normals[])

Sets or retrieves the normals associated with the vertices starting at the specified index. The `normals` parameter is an array of $3*n$ values containing n new normals.

public final void setNormals(int index, Vector3f normals[])
public final void getNormals(int index, Vector3f normals[])

Sets or retrieves the normals associated with the vertices starting at the specified index. The `normals` parameter is an array of vectors containing new normals.

```
public final void setNormals(int index, float normals[], int start,
        int length)
public final void setNormals(int index, Vector3f normals[],
        int start, int length)
```

These methods set the normals associated with the vertices starting at the specified index for this object, using data in `normals` starting at index `start` and ending at index `start+length`.

```
public final void setTextureCoordinate(int index,
        float texCoord[])
public final void getTextureCoordinate(int index,
        float texCoord[])
```

Sets or retrieves the texture coordinate associated with the vertex at the specified index. The `texCoord` parameter is an array of two or three values containing the new texture coordinate.

```
public final void setTextureCoordinate(int index,
        Point2f texCoord)
public final void getTextureCoordinate(int index,
        Point2f texCoord)
public final void setTextureCoordinate(int index,
        Point3f texCoord)
public final void getTextureCoordinate(int index,
        Point3f texCoord)
```

Sets or retrieves the texture coordinate associated with the vertex at the specified index. The `texCoord` parameter is a point containing the new texture coordinate.

```
public final void setTextureCoordinates(int index,
        float texCoords[])
public final void getTextureCoordinates(int index,
        float texCoords[])
```

Sets or retrieves the texture coordinates associated with the vertices starting at the specified index. The `texCoords` parameter is an array of 2*n or 3*n values containing n new texture coordinates.

```
public final void setTextureCoordinates(int index,
        Point2f texCoords[])
public final void getTextureCoordinates(int index,
        Point2f texCoords[])
public final void setTextureCoordinates(int index,
        Point3f texCoords[])
```

```
public final void getTextureCoordinates(int index,
        Point3f texCoords[])
```

Sets or retrieves the texture coordinates associated with the vertices starting at the specified index. The `texCoords` parameter is an array of points containing the new texture coordinate.

```
public final void setTextureCoordinates(int index,
        float texCoords[], int start, int length)
public final void setTextureCoordinates(int index,
        Point2f texCoords[], int start, int length)
public final void setTextureCoordinates(int index,
        Point3f texCoords[], int start, int length)
```

These methods set the texture coordinates associated with the vertices starting at the specified index for this object, using data in `texCoords` starting at index `start` and ending at index `start+length`.

7.2.2 PointArray Object

The PointArray object extends GeometryArray and provides no additional methods. Objects of this class draw the array of vertices as individual points.

Constructors

```
public PointArray(int vertexCount, int vertexFormat)
```

Constructs an empty PointArray object with the specified vertex format and number of vertices.

7.2.3 LineArray Object

The LineArray object extends GeometryArray and provides no additional methods. Objects of this class draw the array of vertices as individual line segments. Each pair of vertices defines a line segment to be drawn.

Constructors

```
public LineArray(int vertexCount, int vertexFormat)
```

Constructs an empty LineArray object with the specified vertex format and number of vertices.

7.2.4 TriangleArray Object

The TriangleArray object extends GeometryArray and provides no additional methods. Objects of this class draw the array of vertices as individual triangles. Each group of three vertices defines a triangle to be drawn.

Constructors

```
public TriangleArray(int vertexCount, int vertexFormat)
```

Constructs an empty TriangleArray object with the specified vertex format and number of vertices.

7.2.5 QuadArray Object

The QuadArray object extends GeometryArray and provides no additional methods. Objects of this class draw the array of vertices as individual quadrilaterals. Each group of four vertices defines a quadrilateral to be drawn. A quadrilateral must be planar and convex or results are undefined. A quadrilateral may be rendered as a pair of triangles with either diagonal line arbitrarily chosen to split the quad.

Constructors

```
public QuadArray(int vertexCount, int vertexFormat)
```

Constructs an empty QuadArray object with the specified vertex format and number of vertices.

7.2.6 GeometryStripArray Object

GeometryStripArray is an abstract class from which all strip primitives (line strip, triangle strip, and triangle fan) are derived. In addition to specifying the array of vertex elements, which is inherited from GeometryArray, the GeometryStripArray class specifies an array of per-strip vertex counts that specifies where the separate strips appear in the vertex array.

Constructors

The GeometryStripArray object has the following constructor.

```
public GeometryStripArray(int vertexCount, int vertexFormat,
       int stripVertexCounts[])
```

Constructs an empty GeometryStripArray object with the specified number of vertices, vertex format, and an array of vertex counts per strip. The `vertexCount` parameter specifies the number of vertex elements in this array.

The `stripVertexCounts` parameter is an array that specifies the count of the number of vertices for each separate strip. The length of this array specifies the number of separate strips. The sum of the vertex counts for all strips, as specified by the `stripVertexCounts` array, must equal the total count of all vertices as specified by the `vertexCount` parameter.

Methods

The GeometryStripArray object has the following methods.

```
public final int getNumStrips()
```

This method returns the number of strips in the GeometryStripArray.

```
public final void getStripVertexCounts(int stripVertexCounts[])
```

This method gets an array containing a list of vertex counts for each strip.

7.2.7 LineStripArray Object

The LineStripArray extends GeometryStripArray and provides no additional methods. Objects of this class draw an array of vertices as a set of connected line strips. An array of per-strip vertex counts specifies where the separate strips appear in the vertex array. For every strip in the set, each vertex, beginning with the second vertex in the array, defines a line segment to be drawn from the previous vertex to the current vertex.

Constructors

```
public LineStripArray(int vertexCount, int vertexFormat,
       int stripVertexCounts[])
```

Constructs an empty LineStripArray object with the specified number of vertices, vertex format, and array of vertex counts per strip.

7.2.8 TriangleStripArray Object

The TriangleStripArray extends GeometryStripArray and provides no additional methods. Objects of this class draw an array of vertices as a set of connected triangle strips. An array of per-strip vertex counts specifies where the separate strips appear in the vertex array. For every strip in the set, each vertex, beginning with the third vertex in the array, defines a triangle to be drawn using the current vertex and the two previous vertices.

Constructors

```
public TriangleStripArray(int vertexCount, int vertexFormat,
       int stripVertexCounts[])
```

Constructs an empty TriangleStripArray object with the specified number of vertices, vertex format, and array of vertex counts per strip.

7.2.9 TriangleFanArray Object

The TriangleFanArray extends GeometryStripArray and provides no additional methods. Objects of this class draw an array of vertices as a set of connected triangle fans. An array of per-strip vertex counts specifies where the separate strips (fans) appear in the vertex array. For every strip in the set, each vertex, beginning with the third vertex in the array, defines a triangle to be drawn using the current vertex, the previous vertex, and the first vertex. This can be thought of as a collection of convex polygons.

Constructors

```
public TriangleFanArray(int vertexCount, int vertexFormat,
       int stripVertexCounts[])
```

Constructs an empty TriangleFanArray object with the specified number of vertices, vertex format, and array of vertex counts per strip.

7.2.10 IndexedGeometryArray Object

An IndexedGeometryArray object is an abstract class that extends GeometryArray to allow vertex data to be accessed via a level of indirection. In addition to the separate arrays of coordinates, colors, normals, and texture coordinates—inherited from GeometryArray—an IndexedGeometryArray object adds corresponding arrays of coordinate indices, color indices, normal indices, and texture coordinate indices.

Constants

The IndexedGeometryArray object defines the following flags.

```
public final static int ALLOW_COORDINATE_INDEX_READ
public final static int ALLOW_COORDINATE_INDEX_WRITE
```

These flags specify that the IndexedGeometryArray object allows reading or writing of the array of coordinate indices.

```
public static final int ALLOW_COLOR_INDEX_READ
public static final int ALLOW_COLOR_INDEX_WRITE
```

These flags specify that the IndexedGeometryArray object allows reading or writing of the array of color indices.

```
public static final int ALLOW_NORMAL_INDEX_READ
public static final int ALLOW_NORMAL_INDEX_WRITE
```

These flags specify that the IndexedGeometryArray object allows reading or writing of the array of normal indices.

```
public static final int ALLOW_TEXCOORD_INDEX_READ
public static final int ALLOW_TEXCOORD_INDEX_WRITE
```

These flags specify that the IndexedGeometryArray object allows reading or writing of the array of texture coordinate indices.

Constructors

The IndexedGeometryArray object has one constructor that accepts the same parameters as GeometryArray.

```
public IndexedGeometryArray(int vertexCount, int vertexFormat,
       int indexCount)
```

Constructs an empty IndexedGeometryArray object with the specified number of vertices, vertex format, and indices.

Methods

IndexedGeometryArray methods provide access (`get` and `set` methods) to the individual vertex component index arrays that are used when rendering the geometry. This access is allowed in two different modes: as individual index elements or as arrays of multiple index elements.

```
public final void setCoordinateIndex(int index,
       int coordinateIndex)
public final int getCoordinateIndex(int index)
```

Sets or retrieves the coordinate index associated with the vertex at the specified index.

```
public final void setCoordinateIndices(int index,
       int coordinateIndices[])
public final void getCoordinateIndices(int index,
       int coordinateIndices[])
```

Sets or retrieves the coordinate indices associated with the vertices starting at the specified index.

```
public final void setColorIndex(int index, int colorIndex)
public final int getColorIndex(int index)
```

Sets or retrieves the color index associated with the vertex at the specified index.

```
public final void setColorIndices(int index, int colorIndices[])
public final void getColorIndices(int index, int colorIndices[])
```

Sets or retrieves the color indices associated with the vertices starting at the specified index.

```
public final void setNormalIndex(int index, int normalIndex)
public final int getNormalIndex(int index)
```

Sets or retrieves the normal index associated with the vertex at the specified index.

```
public final void setnormalIndices(int index, int normalIndices[])
public final void getNormalIndices(int index, int normalIndices[])
```

Sets or retrieves the normal indices associated with the vertices starting at the specified index.

```
public final void setTextureCoordinateIndex(int index,
       int texCoordIndex)
public final int getTextureCoordinateIndex(int index)
```

Sets or retrieves the texture coordinate index associated with the vertex at the specified index.

```
public final void setTextureCoordinateIndices(int index,
       int texCoordIndices[])
public final void getTextureCoordinateIndices(int index,
       int texCoordIndices[])
```

Sets or retrieves the texture coordinate indices associated with the vertices starting at the specified index.

```
public final int getIndexCount()
```

Retrieves the number of indices for this IndexedGeometryArray.

7.2.11 IndexedPointArray Object

The IndexedPointArray object extends IndexedGeometryArray and provides no additional methods. Objects of this class draw the array of vertices as individual points.

Constructors

The IndexedPointArray object has the following constructor.

```
public IndexedPointArray(int vertexCount, int vertexFormat,
       int indexCount)
```

Constructs an empty IndexedPointArray object with the specified number of vertices, vertex format (see Section 7.2.1, "GeometryArray Object," for a description of the supported vertex formats), and the number of indices in this array.

7.2.12 IndexedLineArray Object

The IndexedLineArray object extends IndexedGeometryArray and provides no additional methods. Objects of this class draw the array of vertices as individual line segments. Each pair of vertices defines a line segment to be drawn.

Constructors

The IndexedLineArray object has the following constructor.

```
public IndexedLineArray(int vertexCount, int vertexFormat,
       int indexCount)
```

Constructs an empty IndexedLineArray object with the specified number of vertices, vertex format, and the number of indices in this array. The `vertexFormat` is a mask indicating which components are present in each vertex (see

Section 7.2.1, "GeometryArray Object," for a description of the supported vertex formats).

7.2.13 IndexedTriangleArray Object

The IndexedTriangleArray object extends IndexedGeometryArray and provides no additional methods. Objects of this class draw the array of vertices as individual triangles. Each group of three vertices defines a triangle to be drawn.

Constructors

The IndexedTriangleArray object has the following constructor.

```
public IndexedTriangleArray(int vertexCount, int vertexFormat,
       int indexCount)
```

Constructs an empty IndexedTriangleArray object with the specified number of vertices, vertex format, and the number of indices in this array. The `vertexFormat` is a mask indicating which components are present in each vertex (see Section 7.2.1, "GeometryArray Object" for a description of the supported vertex formats).

7.2.14 IndexedQuadArray Object

The IndexedQuadArray object extends IndexedGeometryArray and provides no additional methods. Objects of this class draw the array of vertices as individual quadrilaterals. Each group of four vertices defines a quadrilateral to be drawn. A quadrilateral must be planar and convex or results are undefined. A quadrilateral may be rendered as a pair of triangles with either diagonal line arbitrarily chosen to split the quad.

Constructors

The IndexedQuadArray object has the following constructor.

```
public IndexedQuadArray(int vertexCount, int vertexFormat,
       int indexCount)
```

Constructs an empty IndexedQuadArray object with the specified number of vertices, vertex format (see Section 7.2.1, "GeometryArray Object," for a description of the supported vertex formats), and the number of indices in this array.

7.2.15 IndexedGeometryStripArray Object

IndexedGeometryStripArray is an abstract class from which all strip primitives (line strip, triangle strip, and triangle fan) are derived. In addition to specifying the array of vertex elements, which is inherited from IndexedGeometryArray, the IndexedGeometryArrayStrip class specifies an array of per-strip index counts that specifies where the separate strips appear in the indexed vertex array.

Constructors

The IndexedGeometryStripArray object has the following constructor.

```
public IndexedGeometryStripArray(int vertexCount,
        int vertexFormat, int indexCount, int stripIndexCounts[])
```

Constructs an empty IndexedGeometryStripArray object with the specified number of vertices, vertex format, number of indices in the array, and an array of index counts per strip. The `vertexCount` parameter specifies the number of vertex elements in this array. The `vertexFormat` parameter is a mask indicating which vertex components are present in each vertex. The `indexCount` parameter specifies the number of indices in this array. The `stripIndexCounts` parameter is an array that specifies the count of the number of indices for each separate strip. The length of this array specifies the number of separate strips. The sum of the index counts for all strips, as specified by the `stripIndexCounts` array, must equal the total count of all indices as specified by the `indexCount` parameter.

Methods

The IndexedGeometryArrayStrip object has the following methods.

```
public final int getNumStrips()
```

Gets the number of strips in the IndexedGeometryStripArray.

```
public final void getStripIndexCounts(int stripIndexCounts[])
```

Gets a list of the `indexCounts` for each strip.

7.2.16 IndexedLineStripArray Object

The IndexedLineStripArray extends IndexedGeometryStripArray and provides no additional methods. Objects of this class draw an array of vertices as a set of connected line strips. An array of per-strip index counts specifies where the separate strips appear in the indexed vertex array. For every strip in the set, each vertex, beginning with the second vertex in the array, defines a line segment to be drawn from the previous vertex to the current vertex.

Constructors

The IndexedLineStripArray object has the following constructor.

```
public IndexedLineStripArray(int vertexCount, int vertexFormat,
       int indexCount, int stripIndexCounts[])
```

Constructs an empty IndexedLineStrip object with the specified number of vertices, vertex format, number of indices in this array, and an array that specifies number of indices for each strip. The `vertexFormat` parameter is a mask indicating which components are present in each vertex. This is specified as one or more individual flags that are bitwise ORed together to describe the per-vertex data (see Section 7.2.1, "GeometryArray Object," for a description of the supported vertex formats).

7.2.17 IndexedTriangleStripArray Object

The IndexedTriangleStripArray extends IndexedGeometryStripArray and provides no additional methods. Objects of this class draw an array of vertices as a set of connected triangle strips. An array of per-strip index counts specifies where the separate strips appear in the indexed vertex array. For every strip in the set, each vertex, beginning with the third vertex in the array, defines a triangle to be drawn using the current vertex and the two previous vertices.

Constructors

The IndexedTriangleStripArray object has the following constructor.

```
public IndexedTriangleStripArray(int vertexCount,
       int vertexFormat, int indexCount, int stripIndexCounts[])
```

Constructs an empty IndexedTriangleStripArray object with the specified number of vertices, vertex format, number of indices in this array, and an array of index counts per strip. The `vertexFormat` parameter is a mask indicating which components are present in each vertex. This is specified as one or more individual flags that are bitwise ORed together to describe the per-vertex data (see Section 7.2.1, "GeometryArray Object," for a description of the supported vertex formats).

7.2.18 IndexedTriangleFanArray Object

The IndexedTriangleFanArray extends IndexedGeometryStripArray and provides no additional methods. Objects of this class draw an array of vertices as a set of connected triangle fans. An array of per-strip index counts specifies where the separate strips (fans) appear in the indexed vertex array. For every strip in the set, each vertex, beginning with the third vertex in the array, defines a triangle to be drawn using the current vertex, the previous vertex, and the first vertex. This can be thought of as a collection of convex polygons.

Constructors

The IndexedTriangleFanArray object has the following constructor.

```
public IndexedTriangleFanArray(int vertexCount, int vertexFormat,
       int indexCount, int stripIndexCounts[])
```

Constructs an empty IndexedTriangleFanArray object with the specified number of vertices, vertex format, number of indices in this array, and an array of index counts per strip. The `vertexFormat` parameter is a mask indicating which components are present in each vertex. This is specified as one or more individual flags that are bitwise ORed together to describe the per-vertex data (see Section 7.2.1, “GeometryArray Object,” for a description of the supported vertex formats).

7.2.19 CompressedGeometry Object

The CompressedGeometry object is used to store geometry in a compressed format. CompressedGeometry objects use a special format for representing geometric information in one order of magnitude less space. The representation, though lossy, preserves significant object quality during compression. There will be parameters to allow the user to specify the degree of lossy-ness (for example, a space versus quality knob).

For more information, see Appendix B, “3D Geometry Compression.”

Constants

The CompressedGeometry object specifies the following variables.

```
public final static int ALLOW_COUNT_READ
public final static int ALLOW_HEADER_READ
public final static int ALLOW_GEOMETRY_READ
```

These flags, when enabled using the `setCapability` method, allow an application to invoke methods that read its individual component field information.

Constructors

```
public CompressedGeometry(CompressedGeometryHeader hdr,
       byte geometry[])
```

Constructs a CompressedGeometry node component. The `hdr` field is copied into the CompressedGeometry object. The `geometry` parameter must conform to the compressed geometry format as described in Appendix B, "3D Geometry Compression."

Methods

```
public final int getByteCount()
```

Retrieves the size, in bytes, of the compressed geometry buffer.

```
public final void getCompressedGeometryHeader
       (CompressedGeometryHeader hdr)
```

Retrieves the header for this CompressedGeometry object. The header is copied into the CompressedGeometryHeader object provided.

```
public final void getCompressedGeometry(byte compGeom[])
```

Retrieves the compressed geometry associated with the CompressedGeometry object. Copies the compressed geometry from the CompressedGeometry object into the given array.

```
public final Shape3D[] decompress()
```

Decompresses the compressed geometry. Returns an array of Shape nodes containing the decompressed geometry objects.

7.2.20 CompressedGeometryHeader Object

The CompressedGeometryHeader object is used in conjunction with the CompressedGeometry object. The CompressedGeometryHeader object contains information specific to the compressed geometry data stored in the CompressedGeometry NodeComponent object. This header is used to aid in the processing of the compressed geometry by decompression routines. All members in the CompressedGeometryHeader node are public, so no `get` or `set` routines are provided. The CompressedGeometryHeader object should be created, and all values set, by the geometry compression utility.

Constants

`public static final int POINT_BUFFER`
`public static final int LINE_BUFFER`
`public static final int TRIANGLE_BUFFER`

These flags indicate whether the compressed geometry is made up of individual points, line segments, or triangles.

`public static final int COLOR_IN_BUFFER`
`public static final int COLOR_ALPHA_IN_BUFFER`

These flags indicate whether RGB or alpha color information is initialized in the compressed geometry buffer.

`public int majorVersionNumber`
`public int minorVersionNumber`
`public int minorMinorVersionNumber`

These indicate the major, minor, and minor-minor version numbers for the compressed geometry format that was used to compress the geometry.

`public int bufferType`

This flag describes the type of data in the compressed geometry buffer. Only one type may be present in any given compressed geometry buffer.

`public int bufferDataPresent`

This flag indicates whether a particular data component (for example, color) is present in the compressed geometry buffer, preceding any geometric data. If a particular data type is not present then this information will be inherited from the Appearance object.

```
public double scale
public int size
public double xOffset
public double yOffset
public double zOffset
```

These flags indicate the scale, size, and *x*, *y*, and *z* offsets that need to be applied to every point in the compressed geometry buffer to restore the geometry to its original (uncompressed) position.

Constructors

```
public CompressedGeometryHeader()
```

Creates a new CompressedGeometryHeader object to be used for the creation of a CompressedGeometry NodeComponent object.

7.2.21 Raster Object

The Raster object extends Geometry to allow drawing a raster image that is attached to a 3D location in the virtual world. The Raster object contains a point that is defined in the local object coordinate system of the Shape3D node that references the Raster. The Raster object also contains a type specifier, a reference to an ImageComponent2D object or a DepthComponent object, and an integer *x*,*y* offset and a size (width, height) to allow reading or writing of a portion of the referenced image. In addition to being used as a type of geometry for drawing, a Raster object may be used to read back pixel data (color and Z-buffer) from the frame buffer in immediate mode.

Constants

The Raster object defines the following flags.

```
public static final int ALLOW_POSITION_READ
public static final int ALLOW_POSITION_WRITE
public static final int ALLOW_OFFSET_READ
public static final int ALLOW_OFFSET_WRITE
public static final int ALLOW_IMAGE_READ
public static final int ALLOW_IMAGE_WRITE
public static final int ALLOW_DEPTH_COMPONENT_READ
public static final int ALLOW_DEPTH_COMPONENT_WRITE
public static final int ALLOW_SIZE_READ
public static final int ALLOW_SIZE_WRITE
public static final int ALLOW_TYPE_READ
```

These flags specify that the Raster object allows reading or writing of the position, offset, image, depth component, size, or type.

```
public static final int RASTER_COLOR
```

Specifies a Raster object with color data. In this mode, the ImageComponent reference must point to a valid ImageComponent object.

```
public static final int RASTER_DEPTH
```

Specifies a Raster object with depth (Z-buffer) data. In this mode, the depth component reference must point to a valid DepthComponent object.

```
public static final int RASTER_COLOR_DEPTH
```

Specifies a Raster object with both color and depth (Z-buffer) data. In this mode, the image component reference must point to a valid ImageComponent object, and the depth component reference must point to a valid DepthComponent object.

Constructors

```
public Raster()
public Raster(Point3f pos, int type, int xOffset, int yOffset,
       int width, int height, ImageComponent2D image,
       DepthComponent depthComponent)
public Raster(Point3f pos, int type, Point offset, Dimension size,
       ImageComponent2D image, DepthComponent depthComponent)
```

Constructs and initializes a new Raster object. The first form uses default values. The next two forms construct a new raster image with the specified values.

Methods

```
public void setPosition(Point3f pos)
public void getPosition(Point3f pos)
```

These methods set and retrieve the position, in object coordinates, of this raster. This position is transformed into device coordinates and is used as the upper-left corner of the raster.

```
public void setType(int type)
public int getType()
```

These methods set and retrieve the type of this Raster object. The `type` is one of the following: `RASTER_COLOR`, `RASTER_DEPTH`, or `RASTER_COLOR_DEPTH`.

```
public void setOffset(int xOffset, int yOffset)
public void setOffset(Point offset)
public void getOffset(Point offset)
```

These methods set and retrieve the offset within the array of pixels at which to start copying.

```
public void setSize(int width, int height)
public void setSize(Dimension size)
public void getSize(Dimension size)
```

These methods set and retrieve the number of pixels to be copied from the pixel array.

```
public void setImage(ImageComponent2D image)
public ImageComponent2D getImage()
```

These methods set and retrieve the pixel array used to copy pixels to or from a Canvas3D. This is used when the type is RASTER_COLOR or RASTER_COLOR_DEPTH.

```
public void setDepthComponent(DepthComponent depthComponent)
public DepthComponent getDepthComponent()
```

These methods set and retrieve the DepthComponent used to copy pixels to or from a Canvas3D. This is used when the `type` is `RASTER_DEPTH` or `RASTER_COLOR_DEPTH`.

7.2.22 Font3D Object

The Font3D object is used to contain 3D glyphs used in rendering 3D text. These 3D glyphs are constructed from a Java 2D font object and a FontExtrusion object (see Section 7.2.23, "FontExtrusion Object"). To ensure correct rendering, the 2D font object should be created with the default transform. The point size of the 2D font will be used as a rough measure of how fine a tessellation to use when creating the Font3D object: the larger the point size, in general, the finer the tessellation.

Constructors

`public Font3D(Font font, FontExtrusion extrusionPath)`

Creates a Font3D object from the specified Font object. The FontExtrusion object (see Section 7.2.23, "FontExtrusion Object") contains the extrusion path to use on the 2D font glyphs. To ensure correct rendering, the font must be created with the default AffineTransform. The point size of a Font object is used as a rough measure of how finely to tessellate the glyphs. A larger point size will, in general, have finer detail than the same font with a smaller point size. Passing `null` for the `FontExtrusion` parameter results in no extrusion being done.

Custom 3D fonts as well as methods to save 3D fonts to disk will be addressed after the 1.0 release of this specification.

Methods

`public GeometryStripArray[] getAsTriangles(int glyphCode)`

This method returns an array of GeometryStripArrays representing the 3D glyph. The amount of tessellation is roughly determined by the point size used to create the 2D Font object. A larger point size will, in general, have finer detail than the same font with a smaller point size.

A 3D glyph is always defined in a normalized space in which the base of the glyph is 0.0 on the *y*-axis and the left side of the glyph is at 0.0 on the *x*-axis. Because of descenders, the glyph's coordinates can be negative. The maximum value of this space is the maximum glyph width and height (obtained from `FontDesignMetrics.getBounds()`).

`public Bounds getBounds(int glyphCode)`

This method returns the 3D bounding box of the specified glyph code.

`public Font getFont()`

This method returns the Java 2D font used to create this Font3D object.

`public void getFontExtrusion(FontExtrusion extrudePath)`

This method retrieves the FontExtrusion object used to create this Font3D object, and copies it into the specified parameter. For information about the FontExtrusion object, see Section 7.2.23, "FontExtrusion Object."

7.2.23 FontExtrusion Object

The FontExtrusion object is used to describe the extrusion path for a Font3D object (see Section 7.2.22, "Font3D Object"). The extrusion path is used in conjunction with a Font2D object. The extrusion path defines the edge contour of 3D text. This contour is perpendicular to the face of the text. The contour has its origin at the edge of the glyph, with 1.0 being the height of the tallest glyph.

Constructors

`public FontExtrusion()`
`public FontExtrusion(Shape extrusionShape)`

Both of these constructors create a FontExtrusion object. The first constructor creates the object with the default extrusion shape. The default shape is a straight line from 0.0 to 0.2 (straight bevel). The second constructor creates a FontExtrusion object with the specified extrusion shape. The `extrusionShape` parameter is used to construct the edge contour of a Font3D object. Each shape begins with an implicit point at 0.0.

Methods

`public final void setExtrusionShape(Shape extrusionShape)`
`public final void getExtrusionShape(Shape extrusionShape)`

These methods set and retrieve the 2D Shape object associated with this FontExtrusion object. The Shape object describes the extrusion path used to create a 3D glyph from a 2D glyph. The `get` method copies the shape from this object to the given parameter. The `set` method copies the given shape into this FontExtrusion object.

7.2.24 Text3D Geometry Object

A Text3D object is a text string that has been converted to 3D geometry. The Font3D object (see Section 7.2.22, "Font3D Object") determines the appearance of the Text3D NodeComponent object. Each Text3D object has a text position—a point in 3D space where the text should be placed. The 3D text can be placed around this position using different alignments and paths.

Constants

The Text3D object defines the following flags.

```
public static final int ALLOW_FONT3D_READ
public static final int ALLOW_FONT3D_WRITE
public static final int ALLOW_STRING_READ
public static final int ALLOW_STRING_WRITE
public static final int ALLOW_POSITION_READ
public static final int ALLOW_POSITION_WRITE
public static final int ALLOW_ALIGNMENT_READ
public static final int ALLOW_ALIGNMENT_WRITE
public static final int ALLOW_PATH_READ
public static final int ALLOW_PATH_WRITE
public static final int ALLOW_CHARACTER_SPACING_READ
public static final int ALLOW_CHARACTER_SPACING_WRITE
public static final int ALLOW_BOUNDING_BOX_READ
```

These flags control reading and writing of the Font3D component information for Font3D, the String object, the text position value, the text alignment value, the text path value, the character spacing, and the bounding box.

Constructors

```
public Text3D()
public Text3D(Font3D font3D)
public Text3D(Font3D font3D, String string)
public Text3D(Font3D font3D, String string, Point3f position)
public Text3D(Font3D font3D, String string, Point3f position,
       int alignment, int path)
```

Create a new Text3D object. The first constructor creates the Text3D object with no Font3D object associated with it, a `null` string, and all default values: a position of (0.0, 0.0, 0.0), an alignment of `ALIGN_FIRST`, and a path of `PATH_RIGHT`. The other constructors set the appropriate values to the passed-in parameters.

Methods

```
public final Font3D getFont3D()
public final void setFont3D(Font3D font3d)
```

These methods get and set the Font3D object associated with this Text3D object.

```
public final String getString()
public final void setString(String string)
```

These methods get and set the character string associated with this Text3D object.

```
public final void getPosition(Point3f position)
public final void setPosition(Point3f position)
```

These methods get and set the text position. The `position` parameter is used to determine the initial placement of the string. The text position is used in conjunction with the alignment and path to determine how the glyphs are to be placed in the scene. The default value is (0.0, 0.0, 0.0).

```
public final void setAlignment(int alignment)
public final int getAlignment()
```

These methods set and get the text alignment policy for this Text3D NodeComponent object (see Figure 7-4). The `alignment` parameter is used to specify how glyphs in the string are placed in relation to the `position` field. Valid values for the alignment field are:

- ALIGN_CENTER: places the center of the string on the position point.
- ALIGN_FIRST: places the first character of the string on the position point.
- ALIGN_LAST: places the last character of the string on the position point.

The default value of this field is `ALIGN_FIRST`.

```
public final void setPath(int path)
public final int getPath()
```

These methods set and get the node's path field. This field is used to specify how succeeding glyphs in the string are placed in relation to the previous glyph (see Figure 7-4). The path is relative to the local coordinate system of the Text3D node. The default coordinate system (see Section 3.4, "Coordinate Systems") is right-handed with *+Y* being up, *+X* horizontal to the right, and *+Z* directed toward the viewer. Valid values for this field are as follows:

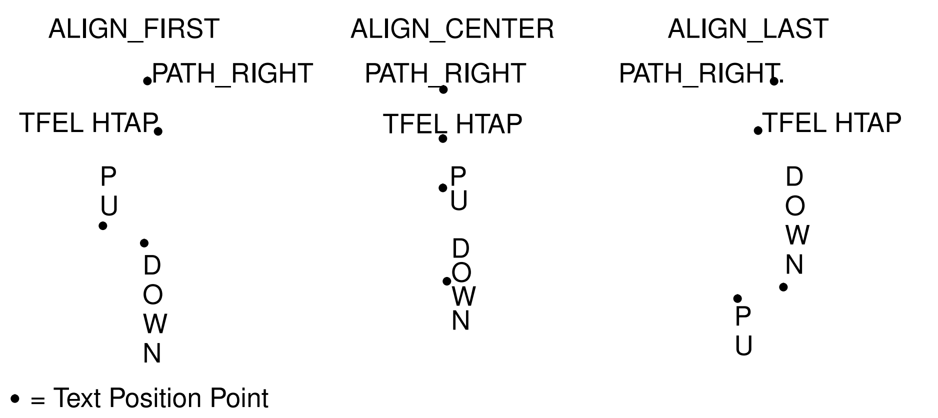


Figure 7-4 Various Text Alignments and Paths

- PATH_LEFT: places succeeding glyphs to the left (the –*X* direction) of the current glyph.
- PATH_RIGHT: places succeeding glyphs to the right (the +*X* direction) of the current glyph.
- PATH_UP: places succeeding glyphs above (the +*Y* direction) the current glyph.
- PATH_DOWN: places succeeding glyphs below (the –*Y* direction) the current glyph.

The default value of this field is `PATH_RIGHT`.

```
public final void getBoundingBox(BoundingBox bounds)
```

This method retrieves the 3D bounding box that encloses this Text3D object.

```
public final void setCharacterSpacing(float characterSpacing)
public final float getCharacterSpacing()
```

These methods set and get the character spacing used to construct the Text3D string. This spacing is in addition to the regular spacing between glyphs as defined in the Font object. A value of 1.0 in this space is measured as the width of the largest glyph in the 2D font. The default value is 0.0.

7.3 Math Component Objects

Java 3D defines a number of additional objects that are used in the construction and manipulation of other Java 3D objects. These objects provide low-level storage and manipulation control for users. They provide methods for representing vertex components (for example, color and position), volumes, vectors, and matrices.

The tuple and matrix math classes are not part of Java 3D per se, but they are needed by Java 3D and are defined here for convenience. Java 3D uses these classes internally and also makes them available for use by applications. These classes will be delivered in a separate `javax.vecmath` package. The tuple and matrix math classes are described in detail in Appendix A, "Math Objects."

7.3.1 Tuple Objects

The tuple objects, listed in Table 7-1, store tuples of length two, three, and four. Java 3D tuples are used to store various kinds of information such as colors, normals, texture coordinates, vertices, and so forth.

The tuple classes are further subdivided by storage type, such as point, vector, color, and so forth, and by class—whether the vector consists of single- or double-precision floating-point numbers or bytes. Only the floating-point tuple classes support math operations.

Table 7-1 Tuple Objects

Class	Description
Tuple2f	Used to represent two-component coordinates in single-precision floating-point format. This class is further divided into the following: Point2f: Represents *x*,*y* point coordinates. TexCoord2f: Represents *x*,*y* texture coordinates. Vector2f: Represents *x*,*y* vector coordinates.
Tuple3b	Used to represent three-component color information stored as three bytes. This class is further divided into the following: Color3b: Represents RGB color values.
Tuple3d	Used to represent point and vector coordinates in double-precision floating-point format. This class is further divided into the following: Point3d: Represents *x*,*y*,*z* point coordinates. Vector3d: Represents *x*,*y*,*z* vector coordinates.

Table 7-1 Tuple Objects (Continued)

Class	Description
Tuple3f	Used to represent three-component colors, point coordinates, texture coordinates, and vectors in single-precision floating-point format. This class is further divided into the following: Color3f: Represents RGB color values. Point3f: Represents *x*,*y*,*z* point coordinates. TexCoord3f: Represents *x*,*y*,*z* texture coordinates. Vector3f: Represents *x*,*y*,*z* vector coordinates.
Tuple4b	Used to represent four-component color information stored as four bytes. This class is further divided into the following: Color4b: Represents RGBα color values.
Tuple4d	Used to represent four-component color information, quaternions, and vectors stored in double-precision floating-point format. This class is further divided into the following: Point4d: Represents *x*,*y*,*z*,*w* point coordinates. Quat4d: Represents *x*,*y*,*z*,*w* quaternion coordinates. Vector4d: Represents *x*,*y*,*z*,*w* vector coordinates.
Tuple4f	Used to represent four-component color information, point coordinates, quaternions, and vectors in single-precision floating-point format. This class is further divided into the following: Color4f: Represents RGBα color values. Point4f: Represents *x*,*y*,*z*,*w* point coordinates. Quat4f: Represents *x*,*y*,*z*,*w* quaternion coordinates. Vector4f: Represents *x*,*y*,*z*,*w* vector coordinates.
AxisAngle4d	Used to represent four-component axis-angle rotations consisting of double-precision floating-point *x*, *y*, and *z* coordinates and a rotation angle in radians.
AxisAngle4f	Used to represent four-component axis-angle rotations consisting of single-precision floating point *x*, *y*, and *z* coordinates and a rotation angle in radians.
GVector	Used to represent a general, dynamically resizeable, one-dimensional vector class.

These are described in more detail in Appendix A, “Math Objects.”

7.3.2 Matrix Objects

The matrix objects, listed in Table 7-2, define a complete 3×3 or 4×4 floating-point transformation matrix. All the vector subclasses operate using this one matrix type.

Table 7-2 Matrix Objects

Class	Description
Matrix3d	Used to represent a double-precision floating-point 3×3 matrix.
Matrix3f	Used to represent a single-precision floating-point 3×3 matrix.

Table 7-2 Matrix Objects (Continued)

Class	**Description**
Matrix4d	Used to represent a double-precision floating-point 4×4 matrix.
Matrix4f	Used to represent a single-precision floating-point 4×4 matrix.
GMatrix	A double-precision, general, dynamically resizeable $N \times M$ matrix class.

These are described in more detail in Appendix A, "Math Objects."

CHAPTER 8

View Model

JAVA 3D introduces a new view model that takes Java's vision of "write once, run anywhere" and generalizes it to include display devices and six-degrees-of-freedom input peripherals such as head trackers. This "write once, view everywhere" nature of the new view model means that an application or applet written using the Java 3D view model can render images to a broad range of display devices, including standard computer displays, multiple-projection display rooms, and head-mounted displays, without modification of the scene graph. It also means that the same application, once again without modification, can render stereoscopic views and can take advantage of the input from a head tracker to control the rendered view.

Java 3D's view model achieves this versatility by cleanly separating the virtual and the physical world. This model distinguishes between how an application positions, orients, and scales a ViewPlatform object (a viewpoint) within the virtual world and how the Java 3D renderer constructs the final view from that viewpoint's position and orientation. The application controls the ViewPlatform's position and orientation; the renderer computes what view to render using this position and orientation, a description of the end-user's physical environment, and the user's position and orientation within the physical environment.

This chapter first explains why Java 3D chose a different view model and some of the philosophy behind that choice. It next describes how that model operates in the simple case of a standard computer screen without head tracking—the most common case. Finally, it presents the relevant parts of the API from a developer's perspective. Appendix C, "View Model Details," describes the Java 3D view model from an advanced developer and Java 3D implementor's perspective.

8.1 Why a New Model?

Camera-based view models as found in low-level APIs give developers control over all rendering parameters. This makes sense when dealing with custom applications, less sense when dealing with systems that wish to have broader applicability: systems such as viewers or browsers that load and display whole worlds as a single unit or systems where the end users view, navigate, display, and even interact with the virtual world.

Camera-based view models emulate a camera in the virtual world, not a human in a virtual world. Developers must continuously reposition a camera to emulate "a human in the virtual world."

The Java 3D view model incorporates head tracking directly, if present, with no additional effort from the developer, thus providing end users with the illusion that they actually exist inside a virtual world.

The Java 3D view model, when operating in a non-head-tracked environment and rendering to a single, standard display, acts very much like a traditional camera-based view model, with the added functionality of being able to transparently generate stereo views.

8.1.1 The Physical Environment Influences the View

Letting the application control all viewing parameters is not reasonable in systems in which the physical environment dictates some of the view parameters.

One example of this is a head-mounted display (HMD), where the optics of the head-mounted display directly determine the field of view that the application should use. Different HMDs have different optics, making it unreasonable for application developers to hard-wire such parameters or allow end users to vary that parameter at will.

Another example is a system that automatically computes view parameters as a function of the user's current head position. The specification of a world and a predefined flight path through that world may not exactly specify an end-user's view. HMD users would expect to look and thus see to their left or right even when following a fixed path through the environment—imagine an amusement park ride with vehicles that follow fixed paths to present content to their visitors, but visitors can continue to move their heads while on those rides.

Depending on the physical details of the end-user's environment, the values of the viewing parameters, particularly the viewing and projection matrices, will vary widely. The factors that influence the viewing and projection matrices include the size of the physical display, how the display is mounted (on the user's head or on a table), whether the computer knows the user's head location in three space, the head mount's actual field of view, the display's pixels per inch, and other such parameters. For more information, see Appendix C, "View Model Details."

8.2 Separation of Physical and Virtual

The Java 3D view model separates the virtual environment, where the application programmer has placed objects in relation to one another, from the physical environment, where the user exists, sees computer displays, and manipulates input devices.

Java 3D also defines a fundamental correspondence between the user's physical world and the virtual world of the graphic application. This physical-to-virtual-world correspondence defines a single common space, a space where an action taken by an end user affects objects within the virtual world and where any activity by objects in the virtual world affects the end-user's view.

8.2.1 The Virtual World

The virtual world is a common space in which virtual objects exist. The virtual world coordinate system exists relative to a high-resolution Locale—each Locale object defines the origin of virtual world coordinates for all of the objects attached to that Locale. The Locale that contains the currently active ViewPlatform object defines the virtual world coordinates that are used for rendering. Java3D eventually transforms all coordinates associated with scene graph elements into this common virtual world space.

8.2.2 The Physical World

The physical world is just that—the real, physical world. This is the space in which the physical user exists, and within which he or she moves his or her head and hands. This is the space in which any physical trackers define their local coordinates, and in which several calibration coordinate systems are described.

The physical world is a space, not a common coordinate system between different execution instances of Java 3D. So while two different computers at two different physical locations on the globe may be running at the same time, there is no mechanism directly within Java 3D to relate their local physical world coordinate systems with each other. Because of calibration issues, the local tracker (if any) defines the local physical world coordinate system known to a particular instance of Java 3D.

8.3 The Objects That Define the View

Java 3D distributes its view model parameters across several objects, specifically, the View object and its associated component objects, the PhysicalBody object, the PhysicalEnvironment object, the Canvas3D object, and the Screen3D object. Figure 8-1 shows graphically the central role of the View object and the subsidiary role of its component objects.

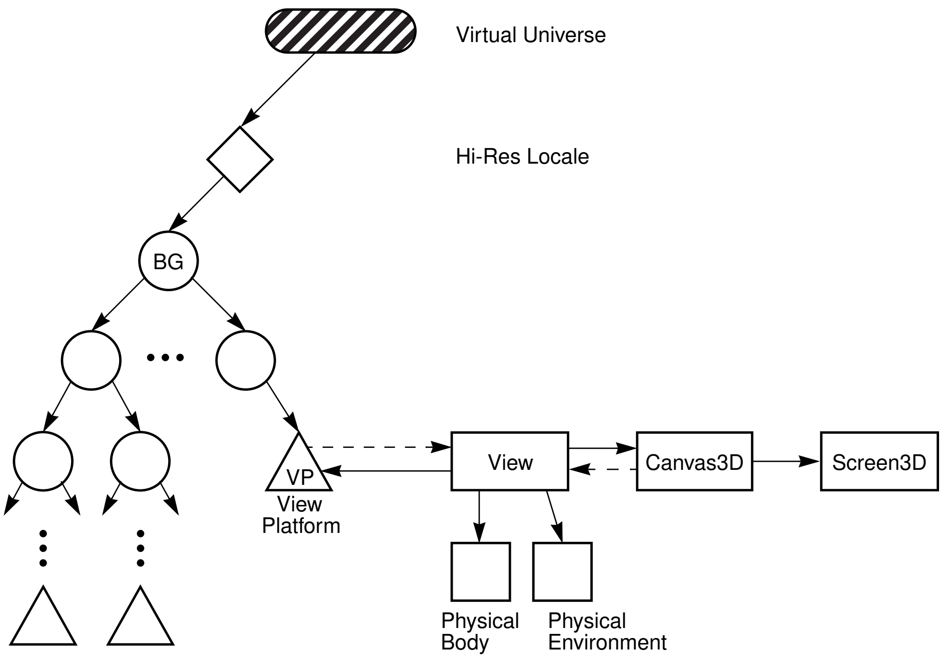


Figure 8-1 View Object, Its Component Objects, and Their Interconnection

The view-related objects shown in Figure 8-1 and their roles are as follows. For each of these objects, the portion of the API that relates to modifying the virtual world and the portion of the API that is relevant to non-head-tracked standard display configurations are derived in this chapter. The remainder of the details are described in Appendix C, “View Model Details.”

- *ViewPlatform*: A leaf node that locates a view within a scene graph. The ViewPlatform’s parents specify its location, orientation, and scale within the virtual universe. See Section 5.10, "ViewPlatform Node," and Section 8.4, "ViewPlatform: A Place in the Virtual World," for more information.

- *View*: The main view object. It contains many pieces of view state. See Section 8.7, "The View Object," for more information.
- *Canvas3D*: The 3D version of the Abstract Windowing Toolkit (AWT) Canvas object. It represents a window in which Java 3D will draw images. It contains a reference to a Screen3D object and information describing the Canvas3D's size, its shape, and its location within the Screen3D object. See Section 8.9, "The Canvas3D Object," for more information.
- *Screen3D*: An object that contains information describing the display screen's physical properties. Java 3D places display-screen information in a separate object to prevent the duplication of screen information within every Canvas3D object that shares a common screen. See Section 8.8, "The Screen3D Object," for more information.
- *PhysicalBody*: An object that contains calibration information describing the user's physical body. See Section 8.10, "The PhysicalBody Object," for more information.
- *PhysicalEnvironment*: An object that contains calibration information describing the physical world, mainly information that describes the environment's six-degrees-of freedom tracking hardware, if present. See Section 8.11, "The PhysicalEnvironment Object," for more information.

Together, these objects describe the geometry of viewing rather than explicitly providing a viewing or projection matrix. The Java 3D renderer uses this information to construct the appropriate viewing and projection matrices. The geometric focus of these view objects provides more flexibility in generating views—a flexibility needed to support alternative display configurations.

8.4 ViewPlatform: A Place in the Virtual World

A ViewPlatform leaf node defines a coordinate system, and thus a reference frame with its associated origin or reference point, within the virtual world. The ViewPlatform serves as a point of attachment for View objects and as a base for determining a renderer's view.

Figure 8-2 shows a portion of a scene graph containing a ViewPlatform node. The nodes directly above a ViewPlatform determine where that ViewPlatform is located and how it is oriented within the virtual world. By modifying the Transform3D object associated with a TransformGroup node anywhere directly above a ViewPlatform, an application or behavior can move that ViewPlatform anywhere within the virtual world. A simple application might define one TransformGroup node directly above a ViewPlatform, as shown in Figure 8-2.

A VirtualUniverse may have many different ViewPlatforms, but a particular View object can only attach itself to a single ViewPlatform. Thus, each rendering onto a Canvas3D is done from the point of view of a single ViewPlatform.

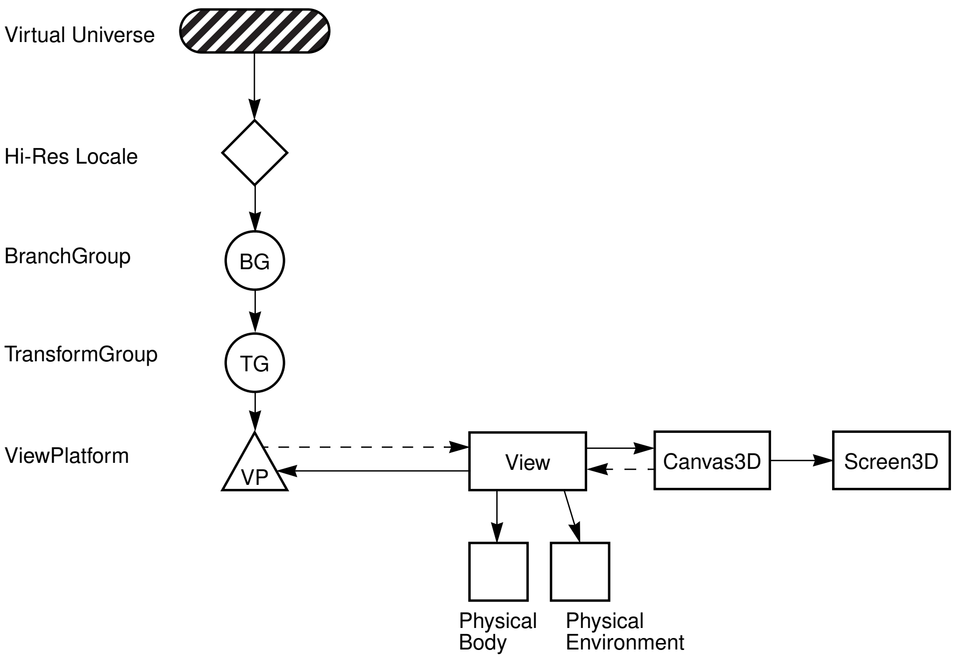


Figure 8-2 A Portion of a Scene Graph Containing a ViewPlatform Object

8.4.1 Moving Through the Virtual World

An application navigates within the virtual world by modifying a ViewPlatform's parent TransformGroup. Examples of applications that modify a ViewPlatform's location and orientation include browsers, object viewers that provide navigational controls, applications that do architectural walkthroughs, and even search-and-destroy games.

Controlling the ViewPlatform object can produce very interesting and useful results. Our first simple scene graph (see Figure 1-2) defines a scene graph for a simple application that draws an object in the center of a window and rotates that object about its center point. In that figure, the Behavior object modifies the TransformGroup directly above the Shape3D node.

An alternative application scene graph, shown in Figure 8-3, leaves the central object alone and moves the ViewPlatform around the world. If the shape node contains a model of the earth, this application could generate a view similar to that seen by astronauts as they orbit the earth.

Had we populated this world with more objects, this scene graph would allow navigation through the world via the Behavior node.

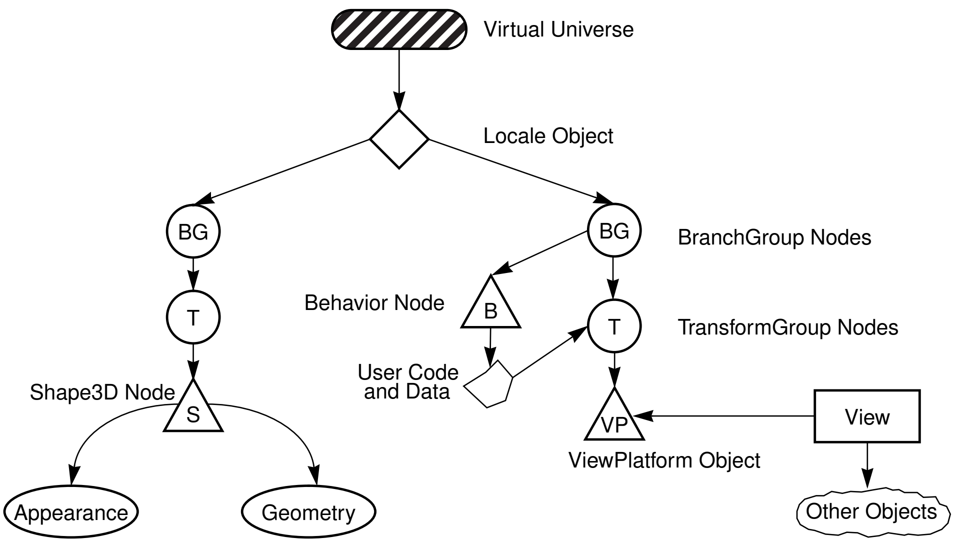


Figure 8-3 A Simple Scene Graph with View Control

Applications and behaviors manipulate a TransformGroup through its access methods. These methods (defined in Section 4.3, "TransformGroup Node") allow an application to retrieve and set the Group node's Transform3D object. Transform3D Node methods include `getTransform` and `setTransform`.

8.4.2 Dropping In on a Favorite Place

A scene graph may contain multiple ViewPlatform objects. If a user detaches a View object from a ViewPlatform and then reattaches that View to a different ViewPlatform, the image on the display will now be rendered from the point of view of the new ViewPlatform. For more information, see Section 8.7, "The View Object."

8.4.3 View Attach Policy

The actual view that Java 3D's renderer draws depends on the view attach policy specified within the currently attached ViewPlatform. The ViewPlatform defines the following methods for setting and retrieving the view attach policy.

Methods

```
public final void setViewAttachPolicy(int policy)
public final int getViewAttachPolicy()
```

These methods set and retrieve the coexistence center in virtual world policy. A ViewPlatform's *view attach policy* determines how Java 3D places the virtual eyepoint within the ViewPlatform. The policy can have one of the following values:

- NOMINAL_HEAD: Ensures that the end-user's nominal eye position in the physical world corresponds to the virtual eye's nominal eye position in the virtual world (the ViewPlatform's origin). In essence, this policy tells Java 3D to position the virtual eyepoint relative to the ViewPlatform origin in the same way as the physical eyepoint is positioned relative to its nominal physical-world origin. Deviations in the physical eye's position and orientation from nominal in the physical world generate corresponding deviations of the virtual eye's position and orientation in the virtual world**.**
- NOMINAL_FEET: Ensures that the end-user's virtual feet always touch the virtual ground. This policy tells Java 3D to compute the physical-to-virtual-world correspondence in a way that enforces this constraint. Java 3D does so by appropriately offsetting the physical eye's position by the end-user's physical height. Java 3D uses the `nominalEyeHeightFromGround` parameter found in the PhysicalBody object (see Section 8.10, "The PhysicalBody Object") to perform this computation.
- NOMINAL_SCREEN: Allows an application to always have the virtual eyepoint appear at some "viewable" distance from a point of interest. This policy tells Java 3D to compute the physical-to-virtual-world correspondence in a way that ensures that the renderer moves the nominal virtual eyepoint away from the point of interest by the amount specified by the `nominalEyeOffsetFromNominalScreen` parameter found in the PhysicalBody object (see Section 8.10, "The PhysicalBody Object").
- NOMINAL_SCREEN_SCALED: This value is deprecated. All view attach policies are now affected by the screen scale so this policy is identical to `NOMINAL_SCREEN`, which should be used instead.

8.4.4 Associating Geometry with a ViewPlatform

Java 3D does not have any built-in semantics for displaying a visible manifestation of a ViewPlatform within the virtual world (an *avatar*). However, a developer can construct and manipulate an avatar using standard Java 3D constructs.

A developer can construct a small scene graph consisting of a TransformGroup node, a behavior leaf node, and a shape node and insert it directly under the BranchGroup node associated with the ViewPlatform object. The shape node would contain a geometric model of the avatar's head. The behavior node would change the TransformGroup's transform periodically to the value stored in a View object's `UserHeadToVworld` parameter, (see Appendix C, "View Model Details"). The avatar's virtual head, represented by the shape node, will now move around in lock-step with the ViewPlatform's TransformGroup *and* any relative position and orientation changes of the user's actual physical head (if a system has a head tracker).

8.5 Generating a View

Java 3D generates viewing matrices in one of a few different ways, depending on whether the end user has a head-mounted or a room-mounted display environment and whether or not head tracking is enabled. This section describes the computation for a non-head-tracked, room-mounted display—a standard computer display. Other environments are described in Appendix C, "View Model Details."

In the absence of head tracking, the ViewPlatform's origin specifies the virtual eye's location and orientation within the virtual world. However, the eye location provides only part of the information needed to render an image. The renderer also needs a projection matrix. In the default mode, Java 3D uses the projection policy, the specified field-of-view information, and the front and back clipping distances to construct a viewing frustum.

8.5.1 Composing Model and Viewing Transformations

Figure 8-4 shows a simple scene graph. To draw the object labeled "S," Java 3D internally constructs the appropriate model, view platform, eye, and projection matrices. Conceptually, the model transformation for a particular object is computed by concatenating all the matrices in a direct path between the object and the VirtualUniverse. The view matrix is then computed—again, conceptually—by concatenating all the matrices between the VirtualUniverse object and the

ViewPlatform attached to the current View object. The eye and projection matrices are constructed from the View object and its associated component objects.

In our scene graph, what we would normally consider the model transformation would consist of the following three transformations: $\mathbf{LT}_1\mathbf{T}_2$. By multiplying $\mathbf{LT}_1\mathbf{T}_2$ by a vertex in the shape object, we would transform that vertex into the virtual universe's coordinate system. What we would normally consider the view platform transformation would be $(\mathbf{LT}_{v1})^{-1}$ or $\mathbf{T}_{v1}{}^{-1}\mathbf{L}^{-1}$. This presents a problem since coordinates in the virtual universe are 256-bit fixed-point values, which cannot be used to efficiently represent transformed points.

Fortunately, however, there is a solution to this problem. Composing the model and view platform transformations gives us

$$\mathbf{T}_{v1}{}^{-1}\mathbf{L}^{-1}\mathbf{LT}_1\mathbf{T}_2 = \mathbf{T}_{v1}{}^{-1}\mathbf{IT}_1\mathbf{T}_2 = \mathbf{T}_{v1}{}^{-1}\mathbf{T}_1\mathbf{T}_2,$$

the matrix that takes vertices in an object's local coordinate system and places them in the ViewPlatform's coordinate system. Note that the high-resolution Locale transformations cancel each other out, which removes the need to actually transform points into high-resolution VirtualUniverse coordinates. The general formula of the matrix that transforms object coordinates to ViewPlatform coordinates is $\mathbf{T}_{vn}{}^{-1}\ldots\mathbf{T}_{v2}{}^{-1}\mathbf{T}_{v1}{}^{-1}\mathbf{T}_1\mathbf{T}_2\ldots\mathbf{T}_m$.

As was mentioned above, the View object contains the remainder of the view information, specifically, the eye matrix, **E**, that takes points in the ViewPlatform's local coordinate system and translates them into the user's eye coordinate system, and the projection matrix, **P**, that projects objects in the eye's coordinate system into clipping coordinates. The final concatenation of matrices for rendering our shape object "S" on the specified Canvas3D is $\mathbf{PET}_{v1}{}^{-1}\mathbf{T}_1\mathbf{T}_2$. In general this is $\mathbf{PET}_{vn}{}^{-1}\ldots\mathbf{T}_{v2}{}^{-1}\mathbf{T}_{v1}{}^{-1}\mathbf{T}_1\mathbf{T}_2\ldots\mathbf{T}_m$.

The details of how Java 3D constructs the matrices **E** and **P** in different end-user configurations are described in Appendix C, "View Model Details."

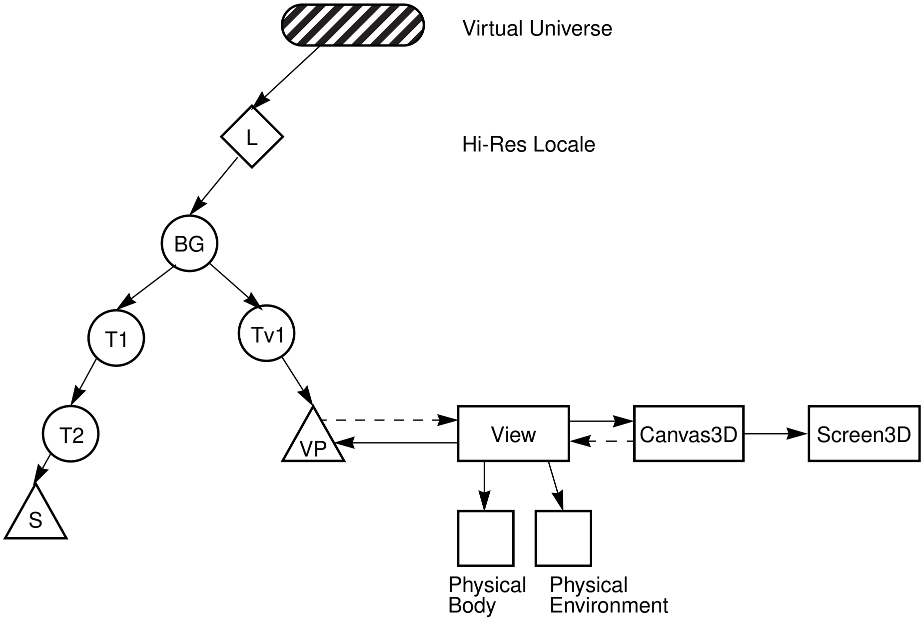


Figure 8-4 Object and ViewPlatform Transformations

8.5.2 Multiple Locales

Java 3D supports multiple high-resolution Locales. In some cases, these Locales are close enough to each other that they can "see" each other, meaning that objects can be rendered even though they are not in the same Locale as the ViewPlatform object that is attached to the View. Java 3D automatically handles this case without the application having to do anything. As in the previous example, where the ViewPlatform and the object being rendered are attached to the same Locale, Java 3D internally constructs the appropriate matrices for cases in which the ViewPlatform and the object being rendered are *not* attached to the same Locale.

Let's take two Locales, L_1 and L_2, with the View attached to a ViewPlatform in L_1. According to our general formula, the modeling transformation—the transformation that takes points in object coordinates and transforms them into VirtualUniverse coordinates—is $\mathbf{L}\mathbf{T}_1\mathbf{T}_2\ldots\mathbf{T}_m$. In our specific example, a point in Locale L_2 would be transformed into VirtualUniverse coordinates by $\mathbf{L}_2\mathbf{T}_1\mathbf{T}_2\ldots\mathbf{T}_m$. The view platform transformation would be $(\mathbf{L}_1\mathbf{T}_{v1}\mathbf{T}_{v1}\ldots\mathbf{T}_{vn})^{-1}$ or $\mathbf{T}_{vn}^{-1}\ldots\mathbf{T}_{v2}^{-1}\mathbf{T}_{v1}^{-1}\mathbf{L}_1^{-1}$. Composing these two matrices gives us

$$\mathbf{T}_{vn}^{-1}\ldots\mathbf{T}_{v2}^{-1}\mathbf{T}_{v1}^{-1}\mathbf{L}_1^{-1}\mathbf{L}_2\mathbf{T}_1\mathbf{T}_2\ldots\mathbf{T}_m.$$

Thus, to render objects in another Locale, it is sufficient to compute $\mathbf{L}_1^{-1}\mathbf{L}_2$ and use that as the starting matrix when composing the model transformations. Given that a Locale is represented by a single high-resolution coordinate position, the transformation $\mathbf{L}_1^{-1}\mathbf{L}_2$ is a simple translation by $\mathbf{L}_2 - \mathbf{L}_1$. Again, it is not necessary to actually transform points into high-resolution VirtualUniverse coordinates.

In general, Locales that are close enough that the difference in their high-resolution coordinates can be represented in double precision by a noninfinite value are close enough to be rendered. In practice, more sophisticated culling techniques can be used to only render those Locales that really are "close enough."

8.6 A Minimal Environment

An application must create a minimal set of Java 3D objects before Java 3D can render to a display device. In addition to a Canvas3D object, the application must create a View object, with its associated PhysicalBody and PhysicalEnvironment objects, and the following scene graph elements:

- A VirtualUniverse object
- A high-resolution Locale object
- A BranchGroup node object
- A TransformGroup node object with associated transform
- A ViewPlatform leaf node object that defines the position and orientation within the virtual universe for generating views

8.7 The View Object

The View object coordinates all aspects of the rendering process. It contains all the parameters or references to objects containing the parameters that determine how to render images to the windows represented by its Canvas3D objects. It also contains the set of canvases that represent various "windows" onto a view.

Java 3D allows applications to specify multiple simultaneously active View objects, each controlling its own set of canvases. For more details on a View object's internals, see Section C.5, "The View Object."

Constructors

The View object specifies the following constructor.

```
public View()
```

Constructs and initializes a new View object.

Methods

The View object specifies the following methods.

```
public final void setPhysicalBody(PhysicalBody physicalBody)
public final PhysicalBody getPhysicalBody()
```

These methods set and retrieve the View's PhysicalBody object. See Section 8.10, "The PhysicalBody Object," for more information on the PhysicalBody object.

```
public final void setPhysicalEnvironment(PhysicalEnvironment
       physicalEnvironment)
public final PhysicalEnvironment getPhysicalEnvironment()
```

These methods set and retrieve the View's PhysicalEnvironment object. See Section 8.11, "The PhysicalEnvironment Object," for more information on the PhysicalEnvironment object.

```
public final void attachViewPlatform(ViewPlatform vp)
```

This method attaches a ViewPlatform leaf node to this View, replacing the existing ViewPlatform. If the ViewPlatform is part of a live scene graph, or is subsequently made live, the scene graph is rendered into all canvases in this View object's list of Canvas3D objects. To remove a ViewPlatform without attaching a new one—causing the View to no longer be rendered—a `null` reference may be passed to this method. In this case, the behavior is as if rendering were simultaneously stopped on all canvases attached to the View—the last frame that was rendered in each remains visible until the View is again attached to a live ViewPlatform object. See Section 5.10, "ViewPlatform Node," for more information on ViewPlatform objects.

```
public final ViewPlatform getViewPlatform()
```

This method retrieves the currently attached ViewPlatform object.

```
public final Canvas3D getCanvas3D(int index)
public final void setCanvas3D(Canvas3D canvas3D, int index)
public final void addCanvas3D(Canvas3D canvas3D)
public final void insertCanvas3D(Canvas3D canvas3D, int index)
```

```
public final void removeCanvas3D(int index)
public final void removeCanvas3D(Canvas3D canvas3D)
```

These methods set, retrieve, add to, insert after, and remove a Canvas3D object from this View. The index specifies the reference to the Canvas3D object within the View object. See Section 8.9, “The Canvas3D Object” for more information on Canvas3D objects.

```
public final Enumeration getAllCanvas3Ds()
```

This method gets the Enumeration object of all the Canvas3Ds.

```
public final void addInputDevice(InputDevice device)
public final Enumeration allInputDevices()
```

These methods allow the introduction of new input devices into a Java 3D environment and the retrieval of all the input devices available within a Java 3D environment. See Section 10.1, “InputDevice Interface” for more information on input devices.

```
public final void addAudioDevice(AudioDevice device)
public final Enumeration allAudioDevices()
```

These methods allow the introduction of new audio devices into a Java 3D environment and the retrieval of all the audio devices available within a Java 3D environment. See Section 11.1, “AudioDevice Interface,” for more information on audio devices.

```
public final void setAudioDevice(AudioDevice device)
```

This method adds an AudioDevice to the list of audio devices.

8.7.1 Projection Policy

The projection policy informs Java 3D whether it should generate a parallel projection or a perspective projection. This policy is attached to the Java 3D View object.

Methods

```
public final void setProjectionPolicy(int policy)
public final int getProjectionPolicy()
```

These two methods set and retrieve the current projection policy for this view. The projection policies are as follows:

- PARALLEL_PROJECTION: Specifies that Java 3D should compute a parallel projection.
- PERSPECTIVE_PROJECTION: Specifies that Java 3D should compute a perspective projection. This is the default setting.

```
public final void setLocalEyeLightingEnable(boolean flag)
public final boolean getLocalEyeLightingEnable()
```

These methods set and retrieve the local eye lighting flag, which indicates whether the local eyepoint is used in lighting calculations for perspective projections. If this flag is set to `true`, the view vector is calculated per vertex based on the direction from the actual eyepoint to the vertex. If this flag is set to `false`, a single view vector is computed from the eyepoint to the center of the view frustum. This is called *infinite eye lighting*. Local eye lighting is disabled by default, and is ignored for parallel projections.

8.7.1.1 Window Sizing and Movement

When users resize or move windows, Java 3D can choose to think of the window as attached either to the physical world or to the virtual world. The *window resize policy* allows an application to specify how the view model will handle resizing requests. The window resize policies are specified by two constants.

Constants

```
public static final int PHYSICAL_WORLD
```

This variable specifies the policy for resizing and moving windows. This policy is used in specifying `windowResizePolicy` and `windowMovementPolicy`. This variable specifies that the specified action takes place only in the physical world.

```
public static final int VIRTUAL_WORLD
```

This variable specifies that Java 3D applies the associated policy in the virtual world.

Methods

```
public final void setWindowResizePolicy(int policy)
public final int getWindowResizePolicy()
```

This variable specifies how Java 3D modifies the view when a user resizes a window. A value of `PHYSICAL_WORLD` states that Java 3D will treat window resizing operations as *only* happening in the physical world. This implies that rendered objects continue to fill the same percentage of the newly sized window, using more or less pixels to draw those objects, depending on whether the window grew or shrank in size. A value of VIRTUAL_WORLD states that Java 3D will treat window resizing operations as also happening in the virtual world whenever a resizing occurs in the physical world. This implies that rendered objects remain the same size (use the same number of pixels), but since the window becomes larger or smaller, the user sees more or less of the virtual world. The default value is PHYSICAL_WORLD.

```
public final void setWindowMovementPolicy(int policy)
public final int getWindowMovementPolicy()
```

This variable specifies what part of the virtual world Java 3D will draw as a function of the window location on the display screen. A value of PHYSICAL_WORLD states that the window acts as if it moves *only* on the physical screen. As the user moves the window on the screen, the window's position on the screen changes but Java 3D continues to draw exactly the same image within that window. A value of VIRTUAL_WORLD states that the window acts as if it also moves within the virtual world. As the user moves the window on the physical screen, the window's position on the screen changes and the image that Java 3D draws changes as well to match what would be visible in the virtual world from a window in that new position. The default value is PHYSICAL_WORLD.

8.7.2 Clip Policies

The clip policies determine how Java 3D interprets clipping distances to both the near and far clip planes. The policies can contain one of four values specifying whether a distance measurement should be interpreted in the physical or the virtual world and whether that distance measurement should be interpreted relative to the physical eyepoint or the physical screen.

Methods

```
public final void setFrontClipPolicy(int policy)
public final int getFrontClipPolicy()
public final void setBackClipPolicy(int policy)
public final int getBackClipPolicy()
```

The *front clip policy* determines where Java 3D places the front clipping plane. The value is one of the following: PHYSICAL_EYE, PHYSICAL_SCREEN, VIRTUAL_EYE, or VIRTUAL_SCREEN. The default value is PHYSICAL_EYE.

The *back clip policy* determines where Java 3D places the back clipping plane. The value is one of the following: `PHYSICAL_EYE`, `PHYSICAL_SCREEN`, `VIRTUAL_EYE`, or `VIRTUAL_SCREEN`. The default value is `PHYSICAL_EYE`.

These policies are defined as follows.

- PHYSICAL_EYE: Specifies that the plane is located relative to the eye's position as measured in the physical space (in meters).
- PHYSICAL_SCREEN: Specifies that the plane is located relative to the screen (that is, the image plate) as measured in physical space (in meters).
- VIRTUAL_EYE: Specifies that the plane is located relative to the virtual eyepoint as measured in virtual world coordinates.
- VIRTUAL_SCREEN: Specifies that the plane is located relative to the screen (that is, the image plate) as measured in virtual world coordinates.

8.7.3 Projection and Clip Parameters

The projection and clip parameters determine the view model's field of view and the front and back clipping distances.

```
public final void setFieldOfView(double fieldOfView)
public final double getFieldOfView()
```

In the default non-head-tracked mode, this value specifies the view model's horizontal field of view in radians. This value is ignored when the view model is operating in head-tracked mode, or when the Canvas3D's window eyepoint policy is set to a value other than the default setting of `RELATIVE_TO_FIELD_OF_VIEW` (see Section C.5.3, "Window Eyepoint Policy").

```
public void setFrontClipDistance(double distance)
public double getFrontClipDistance()
```

This value specifies the distance away from the clip origin, specified by the front clip policy variable, in the direction of gaze where objects stop disappearing. Objects closer than the clip origin (eye or screen) plus the front clip distance are not drawn. Measurements are done in the space (physical or virtual) that is specified by the associated front clip policy parameter.

```
public void setBackClipDistance(double distance)
public double getBackClipDistance()
```

This value specifies the distance away from the clip origin (specified by the back clip policy variable) in the direction of gaze where objects begin disappearing.

Objects farther away from the clip origin (eye or screen) plus the back clip distance are not drawn. Measurements are done in the space (physical or virtual) that is specified by the associated back clip policy parameter. The View object's back clip distance is ignored if the scene graph contains an active Clip leaf node (see Section 5.5, "Clip Node").

8.7.4 Frame Start Time, Duration, and Number

The following methods are used to get information about system execution and performance.

```
public long getCurrentFrameStartTime()
```

This method returns the time at which the most recent rendering frame started. It is defined as the number of milliseconds since January 1, 1970 00:00:00 GMT. Since multiple canvases might be attached to this View, the start of a frame is defined as the point just prior to clearing any canvas attached to this View.

```
public long getLastFrameDuration()
```

This method returns the duration, in milliseconds, of the most recently completed rendering frame. The time taken to render all canvases attached to this View is measured. This duration is computed as the difference between the start of the most recently completed frame and the end of that frame. Since multiple canvases might be attached to this View, the start of a frame is defined as the point just prior to clearing any canvas attached to this View, while the end of a frame is defined as the point just after swapping the buffer for all canvases.

```
public long getFrameNumber()
```

This method returns the frame number for this view. The frame number starts at 0 and is incremented prior to clearing all the canvases attached to this view.

```
public static int getMaxFrameStartTimes()
```

This method retrieves the implementation-dependent maximum number of frames whose start times will be recorded by the system. This value is guaranteed to be at least 10 for all implementations of the Java 3D API.

```
public long getFrameStartTimes(long times[])
```

This method copies the last *k* frame start time values into the user-specified array. The most recent frame start time is copied to location 0 of the array, the next most-recent frame start time is copied into location 1 of the array, and so on. If `times.length` is smaller that `maxFrameStartTimes`, only the last `times.length` values are copied. If `times.length` is greater than `maxFrameStartTimes`, all array elements after index `maxFrameStartTimes` – 1 are set to 0.

8.7.5 View Traversal and Behavior Scheduling

The following methods control the traversal, the rendering, and the execution of the behavior scheduler for this view.

```
public final long[] stopBehaviorScheduler()
public final void startBehaviorScheduler()
public final boolean isBehaviorSchedulerRunning()
```

The first method stops the behavior scheduler after all currently-scheduled behaviors are executed. Any frame-based behaviors scheduled to wake up on the next frame will be executed at least once before the behavior scheduler is stopped. The method returns a pair if integers that specify the beginning and ending time (in milliseconds since January 1, 1970 00:00:00 GMT) of the behavior scheduler's last pass. The second method starts the behavior scheduler running after it has been stopped. The third method retrieves a flag that indicates whether the behavior scheduler is currently running.

```
public final void stopView()
public final void startView()
public final boolean isViewRunning()
```

The first method stops traversing this view after the current state of the scene graph is reflected on all canvases attached to this view. The renderers associated with these canvases are also stopped. The second method starts traversing this view and starts the renderers associated with all canvases attached to this view. The third method returns a flag indicating whether the traverser is currently running on this view.

Note: The above six methods are heavy-weight methods intended for verification and image capture (recording). They are not intended to be used for flow control.

8.7.6 Scene Antialiasing

```
public final void setSceneAntialiasingEnable(boolean flag)
public final boolean getSceneAntialiasingEnable()
```

These methods set and retrieve the scene antialiasing flag. Scene antialiasing is either enabled or disabled for this view. If enabled, the entire scene will be antialiased on each canvas in which scene antialiasing is available. Scene antialiasing is disabled by default.

8.7.7 Depth Buffer

```
public final void setDepthBufferFreezeTransparent(boolean flag)
public final boolean getDepthBufferFreezeTransparent()
```

The `set` method enables or disables automatic freezing of the depth buffer for objects rendered during the transparent rendering pass (that is, objects rendered using alpha blending) for this view. If enabled, depth buffer writes are disabled during the transparent rendering pass regardless of the value of the depth-buffer-write-enable flag in the RenderingAttributes object for a particular node. This flag is enabled by default. The `get` method retrieves this flag.

8.8 The Screen3D Object

The Screen3D object provides a 3D version of the AWT screen object. It contains the screen's physical properties. Java 3D will support multiple active Screen3D objects as soon as AWT support is available. Of course, multiple screens are only available if the machine configuration has multiple output screens. Java 3D primarily needs to know the physical size (in meters) of the Screen3D's visible, addressable raster (the *image plate*) and, in head-tracking mode, the position and orientation of this raster relative to a well-defined physical world coordinate system, specifically, the tracker base coordinate system. Java 3D also needs to know how many pixels the raster can display in both the x and y dimensions. This information allows Java 3D to calculate a pixel's physical dimension.

Calibration utilities can change a Screen3D's physical characteristics or calibration transforms. See Section C.6, "The Screen3D Object."

The Screen3D object has no public constructors. Instead, the Screen3D object associated with a particular Canvas3D object can be obtained from the canvas by calling the `getScreen3D` method. See Section 8.9.2, "Other Canvas3D Parameters."

Methods

These methods provide applications with information concerning the underlying display hardware, such as the screen's width and height in pixels or in meters.

```
public Dimension getSize()
```

This method retrieves the screen's (image plate's) width and height in pixels.

```
public final double getPhysicalScreenWidth()
public final double getPhysicalScreenHeight()
```

These methods retrieve the screen's (image plate's) physical width and height in meters.

8.9 The Canvas3D Object

The Canvas3D object extends the AWT Canvas object to include 3D-related information such as the size of the canvas in pixels, the Canvas3D's location, also in pixels, within a Screen3D object, and whether or not the canvas has stereo enabled. Because all Canvas3D objects contain a reference to a Screen3D object and because Screen3D objects define the size of a pixel in physical units, Java 3D can convert a Canvas3D size in pixels to a physical world size in meters. It can also determine the Canvas3D's position and orientation in the physical world.

Constructors

The Canvas3D object specifies one constructor.

```
public Canvas3D(GraphicsConfiguration graphicsConfiguration)
```

This constructs and initializes a new Canvas3D object given a valid Graphics-Configuration object. Java 3D can render into this Canvas3D object.

For more information on the GraphicsConfiguration object see the Java 2D specification, which will be part of the AWT in JDK 1.2.

8.9.1 Window System–Provided Parameters

Java 3D specifies the size of a Canvas3D in pixels. It extracts this information directly from the AWT's window system. Java 3D only allows applications to access these values, not change them.

```
public Dimension getLocationOnScreen()
public Dimension getSize()
```

These methods, inherited from the parent Canvas class, retrieve the Canvas3D's screen position and size in pixels.

8.9.2 Other Canvas3D Parameters

`public final boolean getStereoAvailable()`

This method specifies whether the underlying hardware supports field-sequential stereo on this canvas.

`public final boolean getStereoEnable()`
`public final void setStereoEnable(boolean flag)`

These methods set or retrieve the flag indicating whether this Canvas3D has stereo enabled. If enabled, Java 3D generates left and right eye images. If the Canvas3D's `StereoAvailable` flag is `false`, Java 3D displays only the *left* eye's view even if an application sets `StereoEnable` to `true`. This parameter allows applications to enable or disable stereo on a canvas-by-canvas basis.

`public final void getDoubleBufferAvailable()`

This method specifies whether the underlying hardware supports double buffering on this canvas.

`public final boolean getDoubleBufferEnable()`
`public final void setDoubleBufferEnable(boolean flag)`

These methods set or retrieve the flag indicating whether this Canvas3D has double buffering enabled. If disabled, all drawing is to the front buffer and no buffer swap will be done between frames. It should be stressed that running Java 3D with double buffering disabled is not recommended.

`public final boolean getSceneAntialiasingAvailable()`

This method specifies whether the underlying hardware supports scene-level antialiasing.

`public final View getView()`

Retrieves the View object that points to this Canvas3D.

`public final Screen3D getScreen3D()`

Retrieves the Screen3D object to which this Canvas3D is attached.

8.10 The PhysicalBody Object

Java 3D defines a PhysicalBody object that contains information concerning the end user's physical characteristics. The head parameters allow end users to specify their own head's characteristics, such as the location of the eyes and the interpupilary distance. See Section C.8, "The PhysicalBody Object," for details. The default values are sufficient for applications that are running in a non-head-tracked environment and that do not manually set the eyepoint.

Constructors

```
public PhysicalBody()
```

This constructor constructs and initializes a default PhysicalBody object.

8.11 The PhysicalEnvironment Object

The PhysicalEnvironment object defines several methods that are described in Section C.9, "The PhysicalEnvironment Object." The default values are sufficient for applications that do not use continuous input devices that are run in a non-head-tracked display environment.

Constructors

```
public PhysicalEnvironment()
```

Constructs and initializes a default PhysicalEnvironment object.

CHAPTER 9

Behaviors and Interpolators

BEHAVIOR nodes provide the means for animating objects, processing keyboard and mouse inputs, reacting to movement, and enabling and processing pick events. Behavior nodes contain Java code and state variables. A Behavior node's Java code can interact with Java objects, change node values within a Java 3D scene graph, change the behavior's internal state—in general, perform any computation it wishes.

Simple behaviors can add surprisingly interesting effects to a scene graph. For example, one can animate a rigid object by using a Behavior node to repetitively modify the TransformGroup node that points to the object one wishes to animate. Alternatively, a Behavior node can track the current position of a mouse and modify portions of the scene graph in response.

9.1 Behavior Object

A Behavior leaf node object contains a scheduling region and two methods: an `initialize` method called once when the behavior becomes "live" and a `processStimulus` method called whenever appropriate by the Java 3D behavior scheduler. The Behavior object also contains the state information needed by its `initialize` and `processStimulus` methods.

The **scheduling region** defines a spatial volume that serves to enable the scheduling of Behavior nodes. A Behavior node is *active* (can receive stimuli) whenever a ViewPlatform's activation volume intersects a Behavior object's scheduling region. Only active behaviors can receive stimuli.

The `initialize` method allows a Behavior object to initialize its internal state and specify its initial wakeup condition(s). Java 3D invokes a behavior's initialize code when the behavior's containing BranchGroup node is added to the virtual universe. Java 3D does not invoke the `initialize` method in a new thread.

Thus, for Java 3D to regain control, the `initialize` method must not execute an infinite loop: It must return. Furthermore, a wakeup condition must be set or else the behavior's `processStimulus` method is never executed.

The `processStimulus` method receives and processes a behavior's ongoing messages. The Java 3D behavior scheduler invokes a Behavior node's `processStimulus` method when a ViewPlatform's activation volume intersects a Behavior object's scheduling region and all of that behavior's wakeup criteria are satisfied. The `processStimulus` method performs its computations and actions (possibly including the registration of state change information that could cause Java 3D to wake other Behavior objects), establishes its next wakeup condition, and finally exits.

9.1.1 Code Structure

When the Java 3D behavior scheduler invokes a Behavior object's `processStimulus` method, that method may perform any computation it wishes. Usually, it will change its internal state and specify its new wakeup conditions. Most probably, it will manipulate scene graph elements. However, the behavior code can only change those aspects of a scene graph element permitted by the capabilities associated with that scene graph element. A scene graph's capabilities restrict behavioral manipulation to those manipulations explicitly allowed.

The application must provide the Behavior object with references to those scene graph elements that the Behavior object will manipulate. The application provides those references as arguments to the behavior's constructor when it creates the Behavior object. Alternatively, the Behavior object itself can obtain access to the relevant scene graph elements either when Java 3D invokes its `initialize` method or each time Java 3D invokes its `processStimulus` method.

Behavior methods have a very rigid structure. Java 3D assumes that they always run to completion (if needed, they can spawn threads). Each method's basic structure consists of the following:

- Code to decode and extract references from the WakeupCondition enumeration that caused the object's awakening
- Code to perform the manipulations associated with the WakeupCondition
- Code to establish this behavior's new WakeupCondition
- A path to Exit (so that execution returns to the Java 3D behavior scheduler)

9.1.2 WakeupCondition Object

A WakeupCondition object is an abstract class specialized to fourteen different WakeupCriterion objects and to four combining objects containing multiple WakeupCriterion objects.

A Behavior node provides the Java 3D behavior scheduler with a WakeupCondition object. When that object's WakeupCondition has been satisfied, the behavior scheduler hands that same WakeupCondition back to the Behavior via an enumeration.

9.1.3 WakeupCriterion Object

Java 3D provides a rich set of wakeup criteria that Behavior objects can use in specifying a complex WakeupCondition. These wakeup criteria can cause Java 3D's behavior scheduler to invoke a behavior's `processStimulus` method whenever

- The center of a ViewPlatform enters a specified region
- The center of a ViewPlatform exits a specified region
- A behavior is activated
- A behavior is deactivated
- A specified TransformGroup node's transform changes
- Collision is detected between a specified Shape3D node's Geometry object and any other object
- Movement occurs between a specified Shape3D node's Geometry object and any other object with which it collides
- A specified Shape3D node's Geometry object no longer collides with any other object
- A specified Behavior object posts a specific event
- A specified AWT event occurs
- A specified time interval elapses
- A specified number of frames have been drawn
- The center of a specified Sensor enters a specified region
- The center of a specified Sensor exits a specified region

A Behavior object constructs a WakeupCriterion by constructing the appropriate criterion object. The Behavior object must provide the appropriate arguments (usually a reference to some scene graph object and possibly a region of interest). Thus, to specify a WakeupOnViewPlatformEntry, a behavior would specify the region that will cause the behavior to execute if a ViewPlatform enters it.

9.1.4 Composing WakeupCriterion Objects

A Behavior object can combine multiple WakeupCriterion objects into a more powerful, composite WakeupCondition. Java 3D behaviors construct a composite WakeupCondition in one of the following ways:

- WakeupAnd: An array of WakeupCriterion objects ANDed together.

  ```
  WakeupCriterion && WakeupCriterion && ...
  ```

- WakeupOr: An array of WakeupCriterion objects ORed together.

  ```
  WakeupCriterion || WakeupCriterion || ...
  ```

- WakeupAndOfOrs: An array of WakeupOr WakeupCondition objects that are then ANDed together.

  ```
  WakeupOr && WakeupOr && ...
  ```

- WakeupOrOfAnds: An array of WakeupAnd WakeupCondition objects that are then ORed together.

  ```
  WakeupAnd || WakeupAnd || ...
  ```

9.2 Composing Behaviors

Behavior objects can condition themselves to awaken only when signaled by another Behavior node. The WakeupOnBehaviorPost WakeupCriterion takes as arguments a reference to a Behavior node and an integer. These two arguments allow a behavior to limit its wakeup criterion to a specific post by a specific behavior.

The WakeupOnBehaviorPost WakeupCriterion permits behaviors to chain their computations, allowing parenthetical computations—one behavior opens a door and the second closes the same door, or one behavior highlights an object and the second unhighlights the same object.

9.3 Scheduling

As a virtual universe grows large, Java 3D must carefully husband its resources to ensure adequate performance. In a 10,000-object virtual universe with 400 or so Behavior nodes, a naive implementation of Java 3D could easily end up consuming the majority of its compute cycles in executing the behaviors associated with the 400 Behavior objects before it draws a frame. In such a situation, the frame rate could easily drop to unacceptable levels.

Behavior objects are usually associated with geometric objects in the virtual universe. In our example of 400 Behavior objects scattered throughout a 10,000-object virtual universe, only a few of these associated geometric objects would be visible at a given time. A sizable fraction of the Behavior nodes—those associated with nonvisible objects—need not be executed. Only those relatively few Behavior objects that are associated with visible objects must be executed.

Java 3D mitigates the problem of a large number of Behavior nodes in a high-population virtual universe through execution culling—choosing only to invoke those behaviors that have high relevance.

Java 3D requires each behavior to have a *scheduling region* and to post a wakeup condition. Together a behavior's scheduling region and wakeup condition provide Java 3D's behavior scheduler with sufficient domain knowledge to selectively prune behavior invocations and only invoke those behaviors that absolutely need to be executed.

9.4 How Java 3D Performs Execution Culling

Java 3D finds all scheduling regions associated with Behavior nodes and constructs a scheduling/volume tree. It also creates an AND/OR tree containing all the Behavior node wakeup criteria. These two data structures provide the domain knowledge Java 3D needs to prune unneeded behavior execution (to perform "execution triage").

Java 3D must track a behavior's wakeup conditions only if a ViewPlatform object's activation volume intersects with that Behavior object's scheduling region. If the ViewPlatform object's activation volume does not intersect with a behavior's scheduling region, Java 3D can safely ignore that behavior's wakeup criteria.

In essence, the Java 3D scheduler performs the following checks:

- Find all Behavior objects with scheduling regions that intersect the ViewPlatform object's activation volume.
- For each Behavior object within the ViewPlatform's activation volume, if that behavior's WakeupCondition is `true`, schedule that Behavior object for execution.

Java 3D's behavior scheduler executes those Behavior objects that have been scheduled by calling the behavior's `processStimulus` method.

9.5 The Behavior API

The Java 3D behavior API spreads its functionality across three objects: the Behavior leaf node, the WakeupCondition object and its subclasses, and the WakeupCriterion objects.

9.5.1 The Behavior Node

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

Methods

The Behavior leaf node class defines the following methods.

```
public abstract void initialize()
```

This method, invoked by Java 3D's behavior scheduler, is used to initialize the behavior's state variables and to establishes its WakeupConditions. Classes that extend Behavior must provide their own `initialize` method.

```
public abstract void processStimulus(Enumeration criteria)
```

This method processes stimuli destined for this behavior. The behavior scheduler invokes this method if its WakeupCondition is satisfied. Classes that extend Behavior must provide their own `processStimulus` method.

```
public final void setSchedulingBounds(Bounds region)
public final Bounds getSchedulingBounds()
```

These two methods access or modify the Behavior node's scheduling bounds. This bounds is used as the scheduling region when the scheduling bounding leaf is set to `null`. A behavior is scheduled for activation when its scheduling region intersects the ViewPlatform's activation volume (if its wakeup criteria have been satisfied). The `getSchedulingBounds` method returns a copy of the associated bounds.

public final void setSchedulingBoundingLeaf(BoundingLeaf region)
public final BoundingLeaf getSchedulingBoundingLeaf()

These two methods access or modify the Behavior node's scheduling bounding leaf. When set to a value other than `null`, this bounding leaf overrides the scheduling bounds object and is used as the scheduling region.

public void wakeupOn(WakeupCondition criteria)

This method defines this behavior's wakeup criteria. This method may only be called from a Behavior object's `initialize` or `processStimulus` methods to (re)arm the next wakeup. It should be the last thing done by those methods.

public void postId(int postId)

This method, when invoked by a behavior, informs the Java 3D scheduler of the identified event. The scheduler will schedule other Behavior objects that have registered interest in this posting.

public void duplicateNode(Node originalNode,
boolean forceDuplicate)

This method copies all the node information from `originalNode` into the current node. This method is called from the `cloneTree` method.

public void updateNodeReferences(NodeReferenceTable
referenceTable)

This is a callback method used to allow a node to check if any nodes referenced by that node have been duplicated via a call to `cloneTree`. This method is called by the `cloneTree` method after all nodes in the subgraph have been duplicated. The cloned leaf node's method will be called and the leaf node can then look up any node references by using the `getNewNodeReference` method found in the NodeReferenceTable object. If a match is found, a reference to the corresponding node in the newly cloned subgraph is returned. If no corresponding reference is found, either a `DanglingReferenceException` is thrown or a reference to the original node is returned, depending on the value of the `allowDanglingReferences` parameter passed in the `cloneTree` call.

protected View getView()

This method returns the primary view associated with this behavior. This method is useful with certain types of behaviors, such as Billboard and LOD, that rely on per-View information and with behaviors in general in regards to scheduling (the distance from the view platform determines the active behaviors). The "primary" view is defined to be the first View attached to a live ViewPlatform, if there is more than one active View. So, for instance, Billboard behaviors would be oriented toward this primary view, in the case of multiple active views into the same scene graph.

9.5.2 WakeupCondition Object

WakeupCondition is an abstract class that is extended by the WakeupCriterion, WakeupOr, WakeupAnd, WakeupOrOfAnds, and WakeupAndOfOr classes. A Behavior node hands a WakeupCondition object to the behavior scheduler and the behavior scheduler hands back an enumeration of that WakeupCondition.

Methods

The Java 3D API provides two methods for constructing WakeupCondition enumerations.

```
public Enumeration allElements()
public Enumeration triggeredElements()
```

These two methods create enumerators that sequentially access this WakeupCondition's wakeup criteria. The first method creates an enumerator that sequentially presents all wakeup criteria that were used to construct this WakeupCondition. The second method creates an enumerator that sequentially presents only those wakeup criteria that have been satisfied.

9.5.3 The WakeupCriterion Objects

WakeupCriterion is an abstract class that consists of several subclasses. Each subclass specifies one particular wakeup criterion, that criterion's associated arguments (if any), and either a flag that indicates whether this criterion caused a Behavior object to awaken or a return field containing the information that caused the Behavior object to awaken.

Methods

```
public boolean hasTriggered()
```

This predicate method returns `true` if this WakeupCriterion contributed to waking a Behavior object.

9.5.3.1 WakeupOnAWTEvent

This WakeupCriterion object specifies that Java 3D should awaken a behavior when the specified AWT event occurs.

Constructors

`public WakeupOnAWTEvent(int AWTId)`
`public WakeupOnAWTEvent(long eventMask)`

The first constructor creates a WakeupOnAWTEvent object that informs the Java 3D scheduler to wake up the specified Behavior object whenever the AWT event specified by `AWTId` occurs. The second constructor creates a WakeupOnAWTEvent object that informs the Java 3D scheduler to wake up the specified Behavior object whenever any of the specified AWT `EVENT_MASK` events occur. The `eventMask` consists of an ORed collection of `EVENT_MASK` values.

Methods

`public AWTEvent[] getAWTEvent()`

This method returns the array of consecutive AWT events that triggered this WakeupCriterion to awaken the Behavior object. The Behavior object can retrieve the `AWTEvent` array and process it in any way it wishes.

9.5.3.2 WakeupOnActivation

The WakeupOnActivation object specifies a wakeup the first time the ViewPlatform's activation region intersects with this object's scheduling region. This gives the behavior an explicit means of executing code when it is activated.

Constructors

`public WakeupOnActivation()`

This constructor creates a WakeupOnActivation criterion.

9.5.3.3 WakeupOnBehaviorPost

This WakeupCriterion object specifies that Java 3D should awaken this behavior when the specified behavior posts the specified ID.

Constructors

`public WakeupOnBehaviorPost(Behavior behavior, int postId)`

This constructor creates a WakeupOnBehaviorPost object that informs the Java 3D scheduler to wake up this Behavior object whenever the specified behavior posts the specified `postId`. A `postId` of 0 specifies that this behavior should awaken on any post from the specified behavior. Specifying a `null` behavior implies that this behavior should awaken whenever any behavior posts the specified `postId`.

Methods

`public int getPostId()`

This method returns the `postId` used in creating this WakeupCriterion.

`public Behavior getBehavior()`

This method returns the behavior specified in this object's constructor.

`public int getTriggeringPostId()`

This method returns the postid that caused the behavior to wake up. If the postid used to construct this wakeup criterion was not zero, the triggering postid will always be equal to the postid used in the constructor.

`public Behavior getTriggeringBehavior()`

This method returns the behavior that triggered this wakeup. If the arming behavior used to construct this object was not null, the triggering behavior will be the same as the arming behavior.

9.5.3.4 WakeupOnDeactivation

The WakeupOnDeactivation object specifies a wakeup on the first detection of a ViewPlatform's activation region no longer intersecting with this object's scheduling region. This gives the behavior an explicit means of executing code when it is deactivated.

Constructors

`public WakeupOnDeactivation()`

This constructor creates a new WakeupOnDeactivation criterion.

`public WakeupOnDeactivation(Bounds region)`

Deprecated constructor. Use the empty constructor instead.

9.5.3.5 WakeupOnElapsedFrames

This WakeupCriterion object specifies that Java 3D should awaken this behavior after it has rendered the specified number of frames. A value of 0 implies that Java 3D will awaken this behavior at the next frame.

Constructors

`public WakeupOnElapsedFrames(int frameCount)`

This constructor creates a WakeupOnElapsedFrames object that informs the Java 3D scheduler to wake up the specified Behavior object after it has drawn `frameCount` frames. A `frameCount` value of 0 means wake up at the next frame.

Methods

`public int getElapsedFrameCount()`

This method returns the frame count used in creating this WakeupCriterion.

9.5.3.6 WakeupOnElapsedTime

This WakeupCriterion object specifies that Java 3D should awaken this behavior after an elapsed number of milliseconds.

Constructors

`public WakeupOnElapsedTime(long milliseconds)`

This constructor creates a WakeupOnElapsedTime object that informs the Java 3D scheduler to wake up the specified Behavior object after the specified number of milliseconds.

Note: The Java 3D scheduler will schedule the object after the specified number of milliseconds have elapsed, not before. However, the elapsed time may actually be slightly greater than the time specified.

Methods

`public long getElapsedFrameTime()`

This method returns the WakeupCriterion's elapsed time value in milliseconds.

9.5.3.7 WakeupOnSensorEntry

This WakeupCriterion object specifies that Java 3D should awaken this behavior when any sensor enters the specified region.

Note: There can be situations in which a sensor may enter and then exit an armed region so rapidly that neither the Entry nor Exit condition is engaged.

Constructors

`public WakeupOnSensorEntry(Bounds region)`

This constructor creates a WakeupOnSensorEntry object that informs the Java 3D scheduler to wake up the specified Behavior object whenever it detects a sensor within the specified `region` for the first time.

Methods

`public Bounds getBounds()`

This method returns the Bounds object used in creating this WakeupCriterion.

9.5.3.8 WakeupOnSensorExit

This WakeupCriterion object specifies that Java 3D should awaken this behavior when any sensor, already marked as within the region, is no longer in that region.

Note: This semantic guarantees that an Exit condition is engaged if its corresponding Entry condition was engaged.

Constructors

`public WakeupOnSensorExit(Bounds region)`

This constructor creates a WakeupOnSensorExit object that informs the Java 3D scheduler to wake up the specified Behavior object the first time it detects that a sensor has left the specified `region`.

Methods

```
public Bounds getBounds()
```

This method returns the Bounds object used in creating this WakeupCriterion.

9.5.3.9 WakeupOnCollisionEntry

This WakeupCriterion object specifies that Java 3D should awaken the WakeupOnCollisionEntry behavior when the specified object collides with any other object in the scene graph.

Constants

```
public static final int USE_GEOMETRY
public static final int USE_BOUNDS
```

These constants specify whether collision against a Group, Shape, or Morph node is done using the actual geometry or whether the geometric bounds are used as an approximation.

Constructors

```
public WakeupOnCollisionEntry(SceneGraphPath armingPath)
public WakeupOnCollisionEntry(SceneGraphPath armingPath,
       int speedHint)
public WakeupOnCollisionEntry(Node armingNode)
public WakeupOnCollisionEntry(Node armingNode, int speedHint)
public WakeupOnCollisionEntry(Bounds armingBounds)
```

These constructors create a WakeupOnCollisionEntry object that informs the Java 3D scheduler to wake up the specified Behavior object if the specified "armed" node's geometry or the specified "armed" bounds collides with any other object in the scene graph. The `speedHint` flag is either `USE_GEOMETRY` or `USE_BOUNDS`.

Methods

```
public SceneGraphPath getArmingPath()
public Bounds getArmingBounds()
```

These methods return the "collideable" path or bounds object used in specifying the collision detection.

```
public SceneGraphPath getTriggeringPath()
public Bounds getTriggeringBounds()
```

These methods return the path or bounds object that caused the collision.

9.5.3.10 WakeupOnCollisionExit

This WakeupCriterion object specifies that Java 3D should awaken the WakeupOnCollisionExit behavior when the specified object no longer collides with any other object in the scene graph.

Constants

```
public static final int USE_GEOMETRY
public static final int USE_BOUNDS
```

These constants specify whether collision against a Group, Shape, or Morph node is done using the actual geometry or whether the geometric bounds are used as an approximation.

Constructors

```
public WakeupOnCollisionExit(SceneGraphPath armingPath)
public WakeupOnCollisionExit(SceneGraphPath armingPath,
       int speedHint)
public WakeupOnCollisionExit(Node armingNode)
public WakeupOnCollisionExit(Node armingNode, int speedHint)
public WakeupOnCollisionExit(Bounds armingBounds)
```

These constructors create a WakeupOnCollisionExit object that informs the Java 3D scheduler to wake up the specified Behavior object if the specified "armed" node's geometry or the specified "armed" bounds no longer collides with any other object in the scene graph. The `speedHint` flag is either `USE_GEOMETRY` or `USE_BOUNDS`.

Methods

```
public SceneGraphPath getArmingPath()
public Bounds getArmingBounds()
```

These methods return the "collideable" path or bounds object used in specifying the collision detection.

```
public SceneGraphPath getTriggeringPath()
public Bounds getTriggeringBounds()
```

These methods return the path or bounds object that caused the collision.

9.5.3.11 WakeupOnCollisionMovement

This WakeupCriterion object specifies that Java 3D should awaken the WakeupOnCollisionMovement behavior when the specified object moves while in a state of collision with any other object in the scene graph.

Constants

```
public static final int USE_GEOMETRY
public static final int USE_BOUNDS
```

These constants specify whether collision against a Group, Shape, or Morph node is done using the actual geometry or whether the geometric bounds are used as an approximation.

Constructors

```
public WakeupOnCollisionMovement(SceneGraphPath armingPath)
public WakeupOnCollisionMovement(SceneGraphPath armingPath,
       int speedHint)
public WakeupOnCollisionMovement(Node armingNode)
public WakeupOnCollisionMovement(Node armingNode, int speedHint)
public WakeupOnCollisionMovement(Bounds armingBounds)
```

These constructors create a WakeupOnCollisionMovement object that informs the Java 3D scheduler to wake up the specified Behavior object if the specified node's geometry or the specified bounds collides with any other object in the scene graph. The `speedHint` flag is either `USE_GEOMETRY` or `USE_BOUNDS`.

Methods

```
public SceneGraphPath getArmingPath()
public Bounds getArmingBounds()
```

These methods return the "collideable" path or bounds object used in specifying the collision detection.

```
public SceneGraphPath getTriggeringPath()
public Bounds getTriggeringBounds()
```

These methods return the path or bounds object that caused the collision.

9.5.3.12 WakeupOnViewPlatformEntry

This WakeupCriterion object specifies that Java 3D should awaken the WakeupOnViewPlatformEntry behavior when any ViewPlatform enters the specified region.

Note: There can be situations in which a ViewPlatform may enter and then exit an armed region so rapidly that neither the Entry nor Exit condition is engaged.

Constructors

`public WakeupOnViewPlatformEntry(Bounds region)`

This constructor creates a WakeupOnViewPlatformEntry object that informs the Java 3D scheduler to wake up the specified Behavior object whenever it detects a ViewPlatform center within the specified `region` for the first time.

Methods

`public Bounds getBounds()`

This method returns the Bounds object used in creating this WakeupCriterion.

9.5.3.13 WakeupOnViewPlatformExit

This WakeupCriterion object specifies that Java 3D should awaken the WakeupOnViewPlatformExit behavior when any ViewPlatform, already marked as within the region, is no longer in that region.

Note: This semantic guarantees that an Exit condition gets engaged if its corresponding Entry condition was engaged.

Constructors

`public WakeupOnViewPlatformExit(Bounds region)`

This constructor creates a WakeupOnViewPlatformExit object that informs the Java 3D scheduler to wake up the specified Behavior object the first time it detects that a ViewPlatform has left the specified `region`.

Methods

```
public Bounds getBounds()
```

This method returns the Bounds object used in creating this WakeupCriterion.

9.5.3.14 WakeupOnTransformChange

The WakeupOnTransformChange object specifies a wakeup when the transform within a specified TransformGroup changes.

Constructors

```
public WakeupOnTransformChange(TransformGroup node)
```

This constructor creates a new WakeupOnTransformChange criterion.

Methods

```
public TransformGroup getTransformGroup()
```

This method returns the TransformGroup node used in creating this WakeupCriterion.

9.5.3.15 WakeupAnd

The WakeupAnd class specifies any number of wakeup conditions ANDed together. This WakeupCondition object specifies that Java 3D should awaken this Behavior when all of the WakeupCondition's constituent wakeup criteria become valid.

Constructors

```
public WakeupAnd(WakeupCriterion conditions[])
```

This constructor creates a WakeupAnd object that informs the Java 3D scheduler to wake up this Behavior object when all the conditions specified in the array of WakeupCriterion objects have become valid.

9.5.3.16 WakeupOr

The WakeupOr class specifies any number of wakeup conditions ORed together. This WakeupCondition object specifies that Java 3D should awaken this Behavior when any of the WakeupCondition's constituent wakeup criteria becomes valid.

Constructors

```
public WakeupOr(WakeupCriterion conditions[])
```

This constructor creates a WakeupOr object that informs the Java 3D scheduler to wake up this Behavior object when any condition specified in the array of WakeupCriterion objects becomes valid.

9.5.3.17 WakeupAndOfOrs

The WakeupAndOfOrs class specifies any number of OR wakeup conditions ANDed together. This WakeupCondition object specifies that Java 3D should awaken this Behavior when all of the WakeupCondition's constituent WakeupOr conditions become valid.

Constructors

```
public WakeupAndOfOrs(WakeupOr conditions[])
```

This constructor creates a WakeupAndOfOrs object that informs the Java 3D scheduler to wake up this Behavior object when all of the WakeupOr conditions specified in the array of WakeupOr objects become valid.

9.5.3.18 WakeupOrOfAnds

The WakeupOrOfAnds class specifies any number of AND wakeup conditions ORed together. This WakeupCondition object specifies that Java 3D should awaken this Behavior when any of the WakeupCondition's constituent WakeupAnd conditions becomes valid.

Constructors

```
public WakeupOrOfAnds(WakeupAnd conditions[])
```

This constructor creates a WakeupOrOfAnds object that informs the Java 3D scheduler to wake up this Behavior object when any of the WakeupAnd conditions specified in the array of WakeupAnd objects becomes valid.

9.6 Interpolator Behaviors

This section describes Java 3D's predefined Interpolator behaviors. They are called *interpolators* because they smoothly interpolate among the two extreme values that an interpolator can produce. Interpolators perform simple behavioral acts, yet they provide broad functionality.

The Java 3D API provides interpolators for a number of functions: manipulating transforms within a TransformGroup, modifying the values of a Switch node, and modifying Material attributes such as color and transparency.

These predefined Interpolator behaviors share the same mechanism for specifying and later for converting a temporal value into an alpha value. Interpolators consist of two portions: a generic portion that all interpolators share and a domain-specific portion.

The generic portion maps time in milliseconds onto a value in the range [0.0, 1.0] inclusive. The domain-specific portion maps an alpha value in the range [0.0, 1.0] onto a value appropriate to the predefined behavior's range of outputs. An alpha value of 0.0 generates an interpolator's minimum value, an alpha value of 1.0 generates an interpolator's maximum value, and an alpha value somewhere in between generates a value proportionally in between the minimum and maximum values.

9.6.1 Mapping Time to Alpha

Several parameters control the mapping of time onto an alpha value. That mapping is deterministic as long as its parameters do not change. Thus, two different interpolators with the same parameters will generate the same alpha value given the same time value. This means that two interpolators that do not communicate can still precisely coordinate their activities, even if they reside in different threads or even different processors—as long as those processors have consistent clocks.

Figure 9-1 shows the components of an interpolator's time-to-alpha mapping. Time is represented on the horizontal axis. Alpha is represented on the vertical axis. As we move from left to right, we see the alpha value start at 0.0, rise to 1.0, and then decline back to 0.0 on the right-hand side.

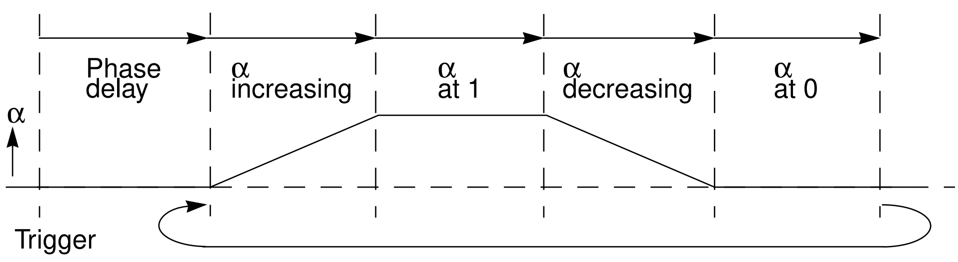


Figure 9-1 An Interpolator's Generic Time-to-Alpha Mapping Sequence

On the left-hand side, the trigger time defines when this interpolator's waveform begins in milliseconds. The region directly to the right of the trigger time, labeled Phase Delay, defines a time period where the waveform does not change. During phase delays α is either 0 or 1, depending on which region it precedes. Phase delays provide an important means for offsetting multiple interpolators from one another, especially where the interpolators have all the same parameters. The next four regions, labeled α increasing, α at 1, α decreasing, and α at 0, all specify durations for the corresponding values of alpha.

Interpolators have a loop count that determines how many times to repeat the sequence of α increasing, α at 1, α decreasing, and α at 0; they also have associated mode flags that enable either the increasing or decreasing portions, or both, of the waveform.

Developers can use the loop count in conjunction with the mode flags to generate various kinds of actions. Specifying a loop count of 1 and enabling the mode flag for only the α-increasing and α-at-1 portion of the waveform, we would get the waveform shown in Figure 9-2.

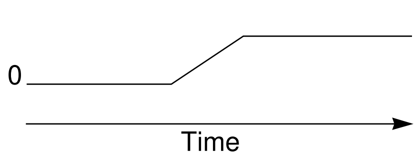


Figure 9-2 An Interpolator Set to a Loop Count of 1 with Mode Flags Set to Enable Only the α-Increasing and α-at-1 Portion of the Waveform

In Figure 9-2, the alpha value is 0 before the combination of trigger time plus the phase delay duration. The alpha value changes from 0 to 1 over a specified interval of time, and thereafter the alpha value remains 1 (subject to the reprogramming of the interpolator's parameters). A possible use of a single α-increasing value might be to combine it with a rotation interpolator to program a door opening.

Similarly, by specifying a loop count of 1 and a mode flag that enables only the α-decreasing and α-at-0 portion of the waveform, we would get the waveform shown in Figure 9-3.

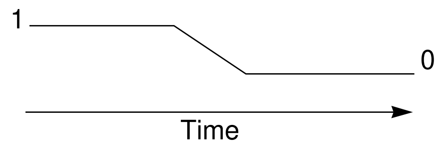


Figure 9-3 An Interpolator Set to a Loop Count of 1 with Mode Flags Set to Enable Only the α-Decreasing and α-at-0 Portion of the Waveform

In Figure 9-3, the alpha value is 1 before the combination of trigger time plus the phase delay duration. The alpha value changes from 1 to 0 over a specified interval, and thereafter the alpha value remains 0 (subject to the reprogramming of the interpolator's parameters). A possible use of a single α-decreasing value might be to combine it with a rotation interpolator to program a door closing.

We can combine both of the above waveforms by specifying a loop count of 1 and setting the mode flag to enable both the α-increasing and α-at-1 portion of the waveform as well as the α-decreasing and α-at-0 portion of the waveform. This combination would result in the waveform shown in Figure 9-4.

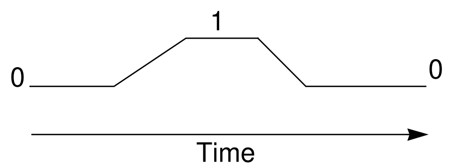


Figure 9-4 An Interpolator Set to a Loop Count of 1 with Mode Flags Set to Enable All Portions of the Waveform

In Figure 9-4, the alpha value is 0 before the combination of trigger time plus the phase delay duration. The alpha value changes from 0 to 1 over a specified period of time, remains at 1 for another specified period of time, then changes from 1 to 0 over a third specified period of time, and thereafter the alpha value remains 0 (subject to the reprogramming of the interpolator's parameters). A possible use of an α-increasing followed by an α-decreasing value might be to combine it with a rotation interpolator to program a door swinging open and then closing.

By increasing the loop count, we can get repetitive behavior, such as a door swinging open and closed some number of times. At the extreme, we can specify a loop count of −1 (representing infinity).

We can construct looped versions of the waveforms shown in Figure 9-2, Figure 9-3, and Figure 9-4. Figure 9-5 shows a looping interpolator with mode flags set to enable only the α-increasing and α-at-1 portion of the waveform.

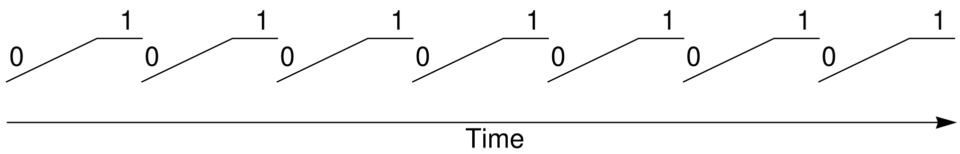


Figure 9-5 An Interpolator Set to Loop Infinitely and Mode Flags Set to Enable Only the α-Increasing and α-at-1 Portion of the Waveform

In Figure 9-5, alpha goes from 0 to 1 over a fixed duration of time, stays at 1 for another fixed duration of time, and then repeats.

Similarly, Figure 9-6 shows a looping interpolator with mode flags set to enable only the α-decreasing and α-at-0 portion of the waveform.

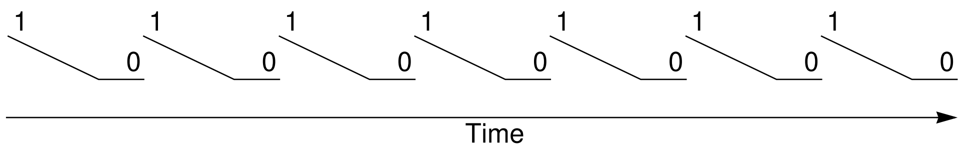


Figure 9-6 An Interpolator Set to Loop Infinitely and Mode Flags Set to Enable Only the α-Decreasing and α-at-0 Portion of the Waveform

Finally, Figure 9-7 shows a looping interpolator with both the increasing and decreasing portions of the waveform enabled.

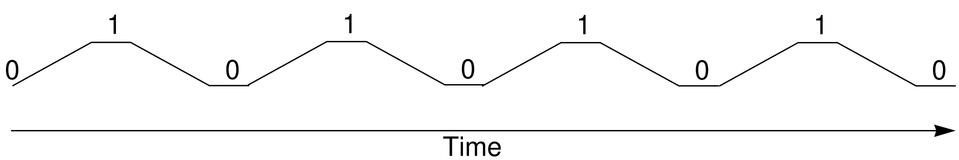


Figure 9-7 An Interpolator Set to Loop Infinitely and Mode Flags Set to Enable All Portions of the Waveform

In all three cases shown by Figure 9-5, Figure 9-6, and Figure 9-7, we can compute the exact value of alpha at any point in time.

Java 3D's preprogrammed behaviors permit other behaviors to change their parameters. When such a change occurs, the alpha value changes to match the state of the newly parameterized interpolator.

9.6.2 Acceleration of Alpha

Commonly, developers want alpha to change slowly at first and to speed up until the change in alpha reaches some appropriate rate. This is analogous to accelerating your car up to the speed limit—it does not start off immediately at the speed limit. Developers specify this "ease-in, ease-out" behavior through two additional parameters, the `increasingAlphaRampDuration` and the `decreasingAlphaRampDuration`.

Each of these parameters specifies a period within the increasing or decreasing alpha duration region during which the "change in alpha" is accelerated (until it reaches its maximum per-unit-of-time step size) and then symmetrically decelerated. Figure 9-8 shows three general examples of how the `increasingAlphaRampDuration` method can be used to modify the alpha waveform. A value of 0 for the increasing ramp duration implies that α is not accelerated; it changes at a constant rate. A value of 0.5 or greater (clamped to 0.5) for this increasing ramp duration implies that the change in α is accelerated during the first half of the period and then decelerated during the second half of the period. For a value of n that is less than 0.5, alpha is accelerated for duration n, held constant for duration $(1.0 - 2n)$, then decelerated for duration n of the period.

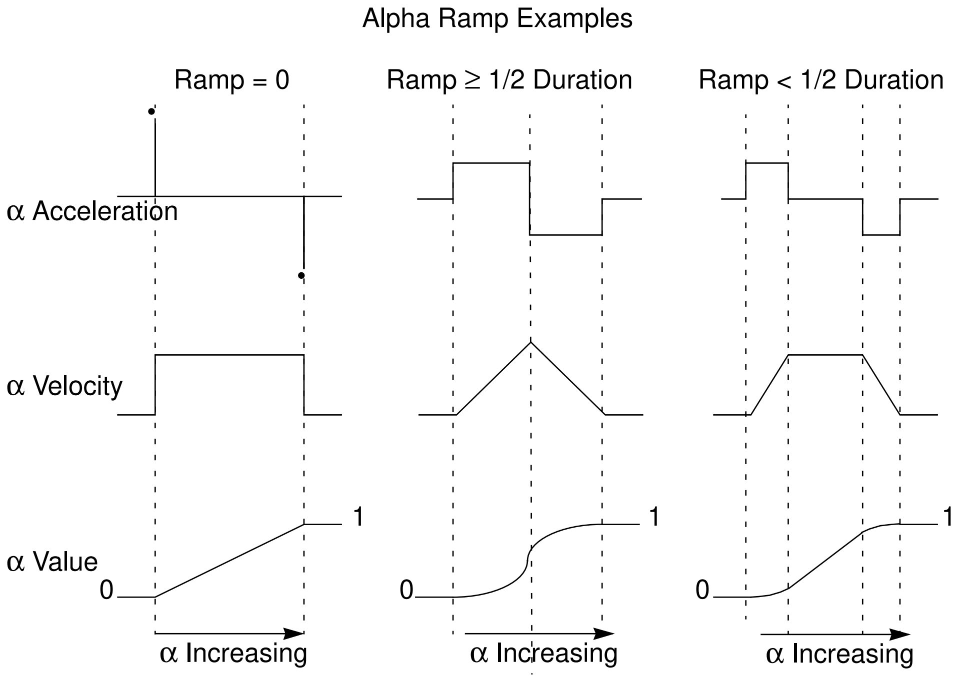


Figure 9-8 How an α-Increasing Waveform Changes with Various Values of `increasingAlphaRampDuration`

9.6.3 The Alpha Class

The Alpha class provides common methods for converting a time value into an alpha value (a value in the range 0.0 to 1.0). The Alpha object is effectively a function of time that generates alpha values in the range [0,1] when sampled: $f_t = [0,1]$. The function f_t and the characteristics of the Alpha object are determined by the following user-definable parameters:

- **`loopCount`**: Specifies the number of times to run this Alpha. A value of –1 specifies that the Alpha loops indefinitely.
- **`triggerTime`**: Specifies the time in milliseconds since the system start time that this object first triggers. If `systemStartTime – currentTime` is less than zero, the Alpha object is started as soon as possible by the system.
- **`phaseDelayDuration`**: Specifies the number of milliseconds to wait after `triggerTime` before actually starting this Alpha.
- **`mode`**: The mode can be set to INCREASING_ENABLE or DECREASING_ENABLE, or the ORed value of the two. INCREASING_ENABLE activates the increasing Alpha parameters described below.

DECREASING_ENABLE activates the decreasing Alpha parameters listed below.

The increasing Alpha parameters are:

- **`increasingAlphaDuration`**: Specifies the time period during which Alpha goes from zero to one.
- **`increasingAlphaRampDuration`**: Specifies the time period during which the Alpha step size increases at the beginning of the `increasingAlphaDuration` and, correspondingly, decreases at the end of the `increasingAlphaDuration`. This parameter is clamped to half of `increasingAlphaDuration`. When this parameter is non-zero, one gets constant acceleration while it is in effect; constant positive acceleration at the beginning of the ramp and constant negative acceleration at the end of the ramp. If this parameter is zero, the effective velocity of the Alpha value is constant and the acceleration is zero (i.e., linearly increasing alpha ramp).
- **`alphaAtOneDuration`**: Specifies the time period that Alpha stays at one.

The decreasing Alpha parameters are:

- **`decreasingAlphaDuration`**: Specifies the time period during which Alpha goes from one to zero.
- **`decreasingAlphaRampDuration`**: Specifies the time period during which the Alpha step size increases at the beginning of the `decreasingAlphaDuration` and, correspondingly, decreases at the end of the `decreasingAlphaDuration`. This parameter is clamped to half of `decreasingAlphaDuration`. When this parameter is non-zero, one gets constant acceleration while it is in effect; constant positive acceleration at the beginning of the ramp and constant negative acceleration at the end of the ramp. If this parameter is zero, the effective velocity of the Alpha value is constant and the acceleration is zero (i.e., a linearly-decreasing alpha ramp).
- **`alphaAtZeroDuration`**: Specifies the time period that Alpha stays at zero.

Constants

```
public static final int INCREASING_ENABLE
public static final int DECREASING_ENABLE
```

These flags specify that this alpha's mode is to use the increasing or decreasing component of the alpha, respectively.

Constructors

```
public Alpha()
public Alpha(int loopCount, long increasingAlphaDuration)
public Alpha(int loopCount, long triggerTime,
       long phaseDelayDuration, long increasingAlphaDuration,
       long increasingAlphaRampDuration, long alphaAtOneDuration)
public Alpha(int loopCount, int mode, long triggerTime,
       long phaseDelayDuration, long increasingAlphaDuration,
       long increasingAlphaRampDuration,
       long alphaAtOneDuration, long decreasingAlphaDuration,
       long decreasingAlphaRampDuration,
       long alphaAtZeroDuration)
```

The first form constructs a new Alpha object using default values. The remaining forms construct a new Alpha object using the specified parameters to define the alpha phases for the object. The default values for the parameters not specified by the constructors are as follows:

loopCount: –1
mode: INCREASING_ENABLE
triggerTime: 0
phaseDelayDuration: 0
increasingAlphaDuration: 1000
increasingAlphaRampDuration: 0
alphaAtOneDuration: 0
decreasingAlphaDuration: 0
decreasingAlphaRampDuration: 0
alphaAtZeroDuration: 0

Methods

```
public float value()
public float value(long atTime)
```

These methods return the alpha value (between 0.0 and 1.0 inclusive) based on the time-to-alpha parameters established for this interpolator. The first method returns the alpha for the current time. The second method returns the alpha for an arbitrary given time. If the alpha mapping has not started, the starting alpha value is returned. If the alpha mapping has completed, the ending alpha value is returned.

```
public void setStartTime(long startTime)
public long getStartTime()
```

These methods set and retrieve this alpha's start time, the base for all relative time specifications.

```
public void setLoopCount(int loopCount)
public int getLoopCount()
```

These methods set and retrieve this alpha's loop count.

```
public void setMode(int mode)
public int getMode()
```

These methods set and retrieve this alpha's mode, which defines which of the alpha regions are active. The mode is one of the following values: `INCREASING_ENABLE`, `DECREASING_ENABLE`, or both (when both of these modes are ORed together).

If the mode is `INCREASING_ENABLE`, the `increasingAlphaDuration`, `increasingAlphaRampDuration`, and `alphaAtOneDuration` are active. If the mode is `DECREASING_ENABLE`, the `decreasingAlphaDuration`, `decreasingAlphaRampDuration`, and `alphaAtZeroDuration` are active. If the mode is both constants ORed, all regions are active. Active regions are all preceded by the phase delay region.

```
public void setTriggerTime(long triggerTime)
public long getTriggerTime()
```

These methods set and retrieve this alpha's trigger time.

```
public void setPhaseDelayDuration(long phaseDelayDuration)
public long getPhaseDelayDuration()
```

These methods set and retrieve this alpha's phase delay duration.

```
public void setIncreasingAlphaDuration(long
       increasingAlphaDuration)
public long getIncreasingAlphaDuration()
```

These methods set and retrieve this alpha's `increasingAlphaDuration`.

```
public void setIncreasingAlphaRampDuration(long
       increasingAlphaRampDuration)
public long getIncreasingAlphaRampDuration()
```

These methods set and retrieve this alpha's `increasingAlphaRampDuration`.

```
public void setAlphAtOneDuration(long alphaAtOneDuration)
public long getAlphaAtOneDuration()
```

These methods set and retrieve this alpha's `alphaAtOneDuration`.

```
public void setDecreasingAlphaDuration(long
       decreasingAlphaDuration)
public long getDecreasingAlphaDuration()
```

These methods set and retrieve this alpha's `decreasingAlphaDuration`.

```
public void setDecreasingAlphaRampDuration(long
       decreasingAlphaRampDuration)
public long getDecreasingAlphaRampDuration()
```

These methods set and retrieve this alpha's `decreasingAlphaRampDuration`.

```
public void setAlphAtZeroDuration(long alphaAtZeroDuration)
public long getAlphaAtZeroDuration()
```

These methods set and retrieve this alpha's `alphaAtZeroDuration`.

```
public boolean finished()
```

This method returns `true` if this Alpha object is past its activity window, that is, if it has finished all its looping activity. This method returns `false` if this Alpha object is still active.

9.6.4 The Interpolator Base Class

Interpolator is an abstract behavior class from which several subclasses are derived. The base Interpolator class contains an Alpha object that provides the means for converting a time value (in milliseconds) into an alpha value in the range [0.0, 1.0] inclusive. Its subclasses map this alpha value into domain-specific values in their range.

Constants

```
protected WakeupCriterion defaultWakeupCriterion
```

This is the default WakeupCondition for all interpolators. The `wakeupOn` method of Behavior, which takes a WakeupCondition as the method parameter, will need to be called at the end of the `processStimulus` method of any class that subclasses Interpolator. This is done with the following method call:

```
wakeupOn(defaultWakeupCriterion);
```

Constructors

The Interpolator behavior class has the following constructors.

`public Interpolator()`
`public Interpolator(Alpha alpha)`

The first form constructs and initializes a new Interpolator with default values. The second form provides the common initialization code for all specializations of Interpolator.

Methods

`public void setAlpha(Alpha alpha)`
`public Alpha getAlpha()`

These methods set and retrieve this interpolator's Alpha object. Setting it to `null` causes the Interpolator to stop running.

`public void setEnable(boolean state)`
`public boolean getEnable()`

These methods set and retrieve this Interpolator's enabled state—the default is enabled.

`public void initialize()`

This is the generic predefined interpolator `initialize` method. It sets the interpolator start time to the current time and schedules the behavior to awaken at the next frame.

9.6.5 PositionInterpolator Object

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

Constructors

The PositionInterpolator object specifies the following constructors.

`public PositionInterpolator(Alpha alpha, TransformGroup target)`

Constructs a trivial position interpolator with a specified target, an `axisOfTranslation` set to the identity transformation, a `startPosition` of 0.0, and an `endPosition` of 1.0 along the *X*-axis.

```
public PositionInterpolator(Alpha alpha, TransformGroup target,
       Transform3D axisOfTranslation, float startPosition,
       float endPosition)
```

Constructs and initializes a new PositionInterpolator that varies the target TransformGroup node's translational component (`startPosition` and `endPosition`). The `axisOfTranslation` parameter specifies the transform that defines the local coordinate system in which this interpolator operates. The translation is done along the *X*-axis of this local coordinate system.

Methods

The PositionInterpolator object specifies the following methods.

```
public void setStartPosition(float position)
public float getStartPosition()
```

These two methods set and get the Interpolator's start position.

```
public void setEndPosition(float position)
public float getEndPosition()
```

These two methods set and get the Interpolator's end position.

```
public void setTarget(TransformGroup target)
public TransformGroup getTarget()
```

These two methods set and get the Interpolator's target TransformGroup node.

```
public void setAxisOfTranslation(Transform3D axis)
public Transform3D getAxisOfTranslation()
```

These two methods set and get the Interpolator's axis of translation.

```
public void processStimulus(Enumeration criteria)
```

This method is invoked by the behavior scheduler every frame. It maps the alpha value that corresponds to the current time into a translation value, computes a transform based on this value, and updates the specified TransformGroup node with this new transform.

9.6.6 RotationInterpolator Object

The RotationInterpolator class extends Interpolator. It modifies the rotational component of its target TransformGroup by linearly interpolating between a pair of specified angles (using the value generated by the specified Alpha object). The interpolated angle is used to generate a rotation transform about the local *Y*-axis of this interpolator.

Constructors

`public RotationInterpolator(Alpha alpha, TransformGroup target)`

Constructs a trivial rotation interpolator with a specified `target`, an `axisOfRotation` set to identity, a minimum angle of 0 radians, and a maximum angle of 2π radians.

```
public RotationInterpolator(Alpha alpha, TransformGroup target,
       Transform3D axisOfRotation, float minimumAngle,
       float maximumAngle)
```

Constructs a new rotation interpolator that varies the target TransformGroup node's rotational component. The `minimumAngle` parameter is the starting angle, in radians; `maximumAngle` is the ending angle, in radians. The `axisOfRotation` parameter specifies the transform that defines the local coordinate system in which this interpolator operates. The rotation is done about the *Y*-axis of this local coordinate system.

Methods

```
public void setMinimumAngle(float angle)
public float getMinimumAngle()
```

These two methods set and get the interpolator's minimum rotation angle, in radians.

```
public void setMaximumAngle(float angle)
public float getMaximumAngle()
```

These two methods set and get the interpolator's maximum rotation angle, in radians.

```
public void setAxisOfRotation(Transform3D axis)
public Transform3D getAxisOfRotation()
```

These two methods set and get the interpolator's axis of rotation.

```
public void setTarget(TransformGroup target)
public TransformGroup getTarget()
```

These two methods set and get the interpolator's target TransformGroup node.

```
public void processStimulus(Enumeration criteria)
```

This method is invoked by the behavior scheduler every frame. It maps the alpha value that corresponds to the current time into a rotation angle, computes a transform based on this angle, and updates the specified TransformGroup node with this new transform.

9.6.7 ColorInterpolator Object

The ColorInterpolator class extends Interpolator. It modifies the color of its target material object by linearly interpolating between a pair of specified colors (using the value generated by the specified Alpha object).

Constructors

```
public ColorInterpolator(Alpha alpha, Material target)
```

Constructs a trivial color interpolator with a specified target, a start color of black, and an end color of white.

```
public ColorInterpolator(Alpha alpha, Material target,
       Color3f startColor, color3f endColor)
```

Constructs a new ColorInterpolator object that varies the target material between two color values (`startColor` and `endColor`).

Methods

```
public void setStartColor(Color3f color)
public void getStartColor(Color3f color)
```

These two methods set and get the interpolator's start color.

```
public void setEndColor(Color3f color)
public void getEndColor(Color3f color)
```

These two methods set and get the interpolator's end color.

```
public void setTarget(Material target)
public Material getTarget()
```

These two methods set and get the interpolator's target Material component object.

`public void processStimulus(Enumeration criteria)`

This method is invoked by the behavior scheduler every frame. It maps the alpha value that corresponds to the current time into a color value and updates the specified Material object with this new color value.

9.6.8 ScaleInterpolator Object

The ScaleInterpolator class extends Interpolator. It modifies the uniform scale component of its target TransformGroup by linearly interpolating between a pair of specified scale values (using the value generated by the specified Alpha object). The interpolated scale value is used to generate a scale transform in the local coordinate system of this interpolator.

Constructors

`public ScaleInterpolator(Alpha alpha, TransformGroup target)`

Constructs a trivial scale interpolator that varies its target TransformGroup node between the two scale values, using the specified alpha, an identity matrix, a minimum scale of 0.1, and a maximum scale of 1.0.

```
public ScaleInterpolator(Alpha alpha, TransformGroup target,
       Transform3D axisOfScale, float minimumScale,
       float maximumScale)
```

Constructs a new ScaleInterpolator object that varies the target TransformGroup node's scale component between two scale values (`minimumScale` and `maximumScale`). The `axisOfScale` parameter specifies the transform that defines the local coordinate system in which this interpolator operates. The scale is done about the origin of this local coordinate system.

Methods

```
public void setMinimumScale(float scale)
public float getMinimumScale()
```

These two methods set and get the interpolator's minimum scale.

```
public void setMaximumScale(float scale)
public float getMaximumScale()
```

These two methods set and get the interpolator's maximum scale.

```
public void setAxisOfScale(Transform3D axis)
public Transform3D getAxisOfScale()
```

These two methods set and get the interpolator's axis of scale.

```
public void setTarget(TransformGroup target)
public TransformGroup getTarget()
```

These two methods set and get the interpolator's target TransformGroup node.

```
public void processStimulus(Enumeration criteria)
```

This method is invoked by the behavior scheduler every frame. It maps the alpha value that corresponds to the current time into a scale value, computes a transform based on this value, and updates the specified TransformGroup node with this new transform.

```
public void updateNodeReferences(NodeReferenceTable
       referenceTable)
```

This is a callback method used to allow a node to check if any nodes referenced by that node have been duplicated via a call to `cloneTree`. This method is called by the `cloneTree` method after all nodes in the subgraph have been duplicated. The cloned leaf node's method will be called and the leaf node can then look up any node references by using the `getNewNodeReference` method found in the NodeReferenceTable object. If a match is found, a reference to the corresponding node in the newly cloned subgraph is returned. If no corresponding reference is found, either a `DanglingReferenceException` is thrown or a reference to the original node is returned, depending on the value of the `allowDanglingReferences` parameter passed in the `cloneTree` call.

9.6.9 SwitchValueInterpolator Object

The SwitchValueInterpolator class extends Interpolator. It modifies the selected child of the target Switch node by linearly interpolating between a pair of specified child index values (using the value generated by the specified Alpha object).

Constructors

```
public SwitchValueInterpolator(Alpha alpha, Switch target)
public SwitchValueInterpolator(Alpha alpha, Switch target,
       int firstChildIndex, int lastChildIndex)
```

Constructs a new SwitchValueInterpolator object that varies the target Switch node's child index between the two values provided (`firstChildIndex`, the index of the first children in the Switch node to select, and `lastChildIndex`, the index of the last children in the Switch node to select).

Methods

```
public void setFirstChildIndex(int firstIndex)
public int getFirstChildIndex()
```

These two methods set and get the interpolator's first child index.

```
public void setLastChildIndex(int lastIndex)
public int getLastChildIndex()
```

These two methods set and get the interpolator's last child index.

```
public void setTarget(Switch target)
public Switch getTarget()
```

These two methods set and get the interpolator's target Switch node.

```
public void processStimulus(Enumeration criteria)
```

This method is invoked by the behavior scheduler every frame. It maps the alpha value that corresponds to the current time into a child index value and updates the specified Switch node with this new child index value.

9.6.10 TransparencyInterpolator Object

The TransparencyInterpolator class extends Interpolator. It modifies the transparency of its target TransparencyAttributes object by linearly interpolating between a pair of specified transparency values (using the value generated by the specified Alpha object).

Constructors

```
public TransparencyInterpolator(Alpha alpha,
       TransparencyAttributes target)
```

Constructs a trivial transparency interpolator with a specified target, a minimum transparency of 0.0, and a maximum transparency of 1.0.

```
public TransparencyInterpolator(Alpha alpha,
       TransparencyAttributes target, float minimumTransparency,
       float maximumTransparency)
```

Constructs a new TransparencyInterpolator object that varies the target material's transparency between the two transparency values (`minimumTransparency`, the starting transparency, and `maximumTransparency`, the ending transparency).

Methods

```
public void setMinimumTransparency(float transparency)
public float getMinimumTransparency()
```

These two methods set and get the interpolator's minimum transparency.

```
public void setMaximumTransparency(float transparency)
public float getMaximumTransparency()
```

These two methods set and get the interpolator's maximum transparency.

```
public void setTarget(TransparencyAttributes target)
public TransparencyAttributes getTarget()
```

These two methods set and get the interpolator's target TransparencyAttributes component object.

```
public void processStimulus(Enumeration criteria)
```

This method is invoked by the behavior scheduler every frame. It maps the alpha value that corresponds to the current time into a transparency value and updates the specified TransparencyAttributes object with this new transparency value.

9.6.11 PositionPathInterpolator Object

The PositionPathInterpolator class extends Interpolator. It modifies the translational component of its target TransformGroup by linearly interpolating among a series of predefined knot/position pairs (using the value generated by the specified Alpha object). The interpolated position is used to generate a translation transform in the local coordinate system of this interpolator.

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

Constructors

```
public PositionPathInterpolator(Alpha alpha,
       TransformGroup target, Transform3D axisOfTranslation, float
       knots[], Point3f positions[])
```

Constructs a new PositionPathInterpolator that varies the translation of the target TransformGroup's transform. The `axisOfTranslation` parameter specifies the transform that defines the local coordinate system in which this interpolator operates. The `knots` parameter specifies an array of knot values that specifies a spline. The `positions` parameter specifies an array of position values at the knots.

Methods

```
public int getArrayLengths()
```

This method retrieves the lengths of the interpolator's knots and positions arrays.

```
public void setPosition(int index, Point3f position)
public void getPosition(int index, Point3f position)
```

These two methods set and get the interpolator's indexed position.

```
public void setKnot(int index, float knot)
public float getKnot(int index)
```

These two methods set and get the interpolator's indexed knot value.

```
public void setAxisOfTranslation(Transform3D axis)
public Transform3D getAxisOfTranslation()
```

These two methods set and get the interpolator's axis of translation.

```
public void setTarget(TransformGroup target)
public TransformGroup getTarget()
```

These two methods set and get the interpolator's target TransformGroup object.

```
public void processStimulus(Enumeration criteria)
```

This method is invoked by the behavior scheduler every frame. It maps the alpha value that corresponds to the current time into a translation value, computes a transform based on this value, and updates the specified TransformGroup node with this new transform.

9.6.12 RotPosPathInterpolator Object

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

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

Constructors

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

This constructor constructs a new RotPosPathInterpolator that varies the rotation and translation of the target TransformGroup's transform. The `axisOfRotPos` parameter specifies the transform that defines the local coordinate system in which this interpolator operates. The `knots` parameter specifies an array of knot values that specifies a spline. The `quats` parameter specifies an array of quaternion values at the knots. The `positions` parameter specifies an array of position values at the knots.

Methods

```
public int getArrayLengths()
```

This method retrieves the lengths of the interpolator's knots, positions, and quats arrays.

```
public void setQuat(int index, Quat4f quat)
public void getQuat(int index, Quat4f quat)
```

These two methods set and get the interpolator's indexed quaternion value.

```
public void setPosition(int index, Point3f position)
public void getPosition(int index, Point3f position)
```

These two methods set and get the interpolator's indexed position.

```
public void setKnot(int index, float knot)
public float getKnot(int index)
```

These two methods set and get the interpolator's indexed knot value.

```
public void setAxisOfRotPos(Transform3D axisOfRotPos)
public Transform3D getAxisOfRotPos()
```

These two methods set and get the interpolator's axis of rotation and translation.

```
public void setTarget(TransformGroup target)
public TransformGroup getTarget()
```

These two methods set and get the interpolator's target TransformGroup object.

```
public void processStimulus(Enumeration criteria)
```

This method is invoked by the behavior scheduler every frame. It maps the alpha value that corresponds to the current time into translation and rotation values, computes a transform based on these values, and updates the specified TransformGroup node with this new transform.

9.6.13 RotPosScalePathInterpolator Object

The RotPosScalePathInterpolator class extends Interpolator. It varies the rotational, translational, and scale components of its target TransformGroup by linearly interpolating among a series of predefined knot/position, knot/orientation, and knot/scale pairs (using the value generated by the specified Alpha object). The interpolated position, orientation, and scale are used to generate a transform in the local coordinate system of this interpolator.

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

Constructors

```
public RotPosScalePathInterpolator(Alpha alpha,
       TransformGroup target, Transform3D axisOfRotPosScale,
       float knots[], Quat4f quats[], Point3f positions[],
       float scales[])
```

This constructor constructs a new RotPosScalePathInterpolator that varies the rotation, translation, and scale of the target TransformGroup's transform. The `axisOfRotPosScale` parameter specifies the transform that defines the local coordinate system in which this interpolator operates. The `knots` parameter specifies an array of knot values that specifies a spline. The `quats` parameter specifies an array of quaternion values at the knots. The `positions` parameter specifies an array of position values at the knots. The `scale` parameter specifies the scale component value.

Methods

`public int getArrayLengths()`

This method retrieves the lengths of the interpolator's knots and positions arrays.

`public void setScale(int index, float scale)`
`public float getScale(int index)`

These two methods set and get the interpolator's indexed scale value.

`public void setQuat(int index, Quat4f quat)`
`public void getQuat(int index, Quat4f quat)`

These two methods set and get the interpolator's indexed quaternion value.

`public void setPosition(int index, Point3f position)`
`public void getPosition(int index, Point3f position)`

These two methods set and get the interpolator's indexed position.

`public void setKnot(int index, float knot)`
`public float getKnot(int index)`

These two methods set and get the interpolator's indexed knot value.

`public void setAxisOfRotPosScale(Transform3D axisOfRotPosScale)`
`public Transform3D getAxisOfRotPosScale()`

These two methods set and get the interpolator's axis of rotation, translation, and scale.

`public void setTarget(TransformGroup target)`
`public TransformGroup getTarget()`

These two methods set and get the interpolator's target TransformGroup object.

`public void processStimulus(Enumeration criteria)`

This method is invoked by the behavior scheduler every frame. It maps the alpha value that corresponds to the current time into translation, rotation, and scale values, computes a transform based on these values, and updates the specified TransformGroup node with this new transform.

9.6.14 RotationPathInterpolator Object

The RotationPathInterpolator class extends the Interpolator class. It varies the rotational component of its target TransformGroup by linearly interpolating among a series of predefined knot/orientation pairs (using the value generated by the specified Alpha object). The interpolated orientation is used to generate a rotation transform in the local coordinate system of this interpolator.

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

Constructors

```
public RotationPathInterpolator(Alpha alpha,
       TransformGroup target, Transform3D axisOfRotation,
       float knots[], Quat4f quats[])
```

This constructor constructs a new RotationPathInterpolator object that varies the target TransformGroup node's transform. The `axisOfRotation` parameter specifies the transform that defines the local coordinate system in which this interpolator operates. The `knots` parameter specifies an array of knot values that specifies a spline. The `quats` parameter specifies an array of quaternion values at the knots.

Methods

```
public int getArrayLengths()
```

This method retrieves the lengths of the interpolator's knots and positions arrays.

```
public void setQuat(int index, Quat4f quat)
public void getQuat(int index, Quat4f quat)
```

These two methods set and get the interpolator's indexed quaternion value.

```
public void setKnot(int index, float knot)
public float getKnot(int index)
```

These two methods set and get the interpolator's indexed knot value.

```
public void setAxisOfRotation(Transform3D axisOfRotation)
public Transform3D getAxisOfRotation()
```

These two methods set and get the interpolator's axis of rotation.

```
public void setTarget(TransformGroup target)
public TransformGroup getTarget()
```

These two methods set and get the interpolator's target TransformGroup object.

```
public void processStimulus(Enumeration criteria)
```

This method is invoked by the behavior scheduler every frame. It maps the alpha value that corresponds to the current time into a rotation angle, computes a transform based on this angle, and updates the specified TransformGroup node with this new transform.

9.7 Level-of-Detail Behaviors

The LOD (Level of Detail) leaf node is an abstract behavior class that operates on a list of Switch group nodes to select one of the children of the Switch nodes. Specializations of the LOD abstract behavior node implement various level-of-detail policies.

9.7.1 LOD Object

The DistanceLOD behavior node implements a distance-based LOD policy.

Constructors

```
public LOD()
```

Constructs and initializes a new LOD node.

Methods

The LOD node class defines the following methods.

```
public final void addSwitch(Switch switchNode)
public final void setSwitch(Switch switchNode, int index)
public final void insertSwitch(Switch switchNode, int index)
public final void removeSwitch(int index)
public final Switch getSwitch(int index)
public final int numSwitches()
```

The `addSwitch` method appends the specified Switch node to this LOD's list of switches. The `setSwitch` method replaces the specified Switch node with the Switch node provided. The `insertSwitch` method inserts the specified Switch node at the specified index. The `removeSwitch` method removes the Switch node at the specified index. The `getSwitch` method returns the Switch node specified by the index. The `numSwitches` method returns a count of this LOD's switches.

```
public final Enumeration getAllSwitches()
```

This method returns the Enumeration object of all switches.

9.7.2 DistanceLOD Object

The DistanceLOD behavior node implements a distance-based LOD policy. The DistanceLOD behavior selects one of the Switch node's children based on distance from the viewer. For distances 0 through *n*—where distance[0] is the most detail, and *n* is least—the DistanceLOD selects child *n* when the viewer is > distance[*n*+1] and ≤ distance[*n*] from the center of the bounds of the DistanceLOD node. The LOD distances are defined in the local coordinate system of this node.

Constructors

```
public DistanceLOD()
public DistanceLOD(float distances[])
```

Construct and initialize a new DistanceLOD node. The `distances` parameter specifies a vector of doubles representing LOD cutoff distances.

Methods

```
public final int numDistances()
public final double getDistance(int whichLOD)
public final void setDistance(int whichLOD, double distance)
```

The `numDistances` method returns a count of the number of LOD distance cutoff parameters. The `getDistance` method returns a particular LOD cutoff distance. The `setDistance` method sets a particular LOD cutoff distance.

`public void initialize()`

This method sets up the initial wakeup criteria.

`public void processStimulus(Enumeration criteria)`

This method computes the appropriate level of detail.

9.8 Billboard Behavior

The Billboard behavior node operates on a TransformGroup node to specify a transform that always aligns itself perpendicular to a specified world-coordinate axis or to a viewer's view vector—regardless, in either case, of transforms above the specified transform node in the scene graph.

Billboard nodes provide the most benefit for complex, roughly symmetric objects. A typical use might consist of a quadrilateral that contains a texture map of a tree.

Constants

The Billboard class adds the following new constants.

`public static final int ROTATE_ABOUT_AXIS`

Specifies that rotation should be about the specified axis.

`public static final int ROTATE_ABOUT_POINT`

Specifies that rotation should be about the specified point and that the children's Y-axis should match the ViewPlatform's *Y*-axis.

Constructors

The Billboard class specifies the following constructors.

`public Billboard()`

Constructs a Billboard behavior node with `ROTATE_ABOUT_AXIS` rotation with an axis pointing along the *Y*-axis.

```
public Billboard(TransformGroup tg)
public Billboard(TransformGroup tg, int mode, Vector3f axis)
public Billboard(TransformGroup tg, int mode, Point3f point)
```

The first constructor constructs a Billboard behavior node with default parameters that operates on the specified target `TransformGroup` node. The default alignment mode is `ROTATE_ABOUT_AXIS`, with the axis along the *Y*-axis. The next two constructors construct a Billboard behavior node with the specified axis and mode that operates on the specified TransformGroup node. The `axis` parameter specifies the ray about which the billboard rotates. The `point` parameter specifies the position about which the billboard rotates. The `mode` parameter is the alignment mode and is either `ROTATE_ABOUT_AXIS` or `ROTATE_ABOUT_POINT`.

Methods

The Billboard class defines the following methods.

```
public final void setAlignmentMode(int mode)
public final int getAlignmentMode()
```

These methods, if enabled by the appropriate flag, permit an application to either retrieve or set the Billboard node's alignment mode, one of `ROTATE_ABOUT_AXIS` or `ROTATE_ABOUT_POINT`.

```
public final void setAlignmentAxis(Vector3f axis)
public final void setAlignmentAxis(float x, float y, float z)
public final void getAlignmentAxis(Vector3f axis)
```

These methods, if enabled by the appropriate flag, permit an application to set or retrieve the Billboard node's alignment axis.

```
public final void setTarget(TransformGroup tg)
public final TransformGroup getTarget()
```

These methods set or retrieve the target TransformGroup node for this Billboard object.

```
public final void setRotationPoint(float x, float y, float z)
public final void setRotationPoint(Point3f point)
public final void getRotationPoint(Point3f point)
```

The first two methods set the rotation point. The third method gets the rotation point and sets the parameter to this value.

```
public void initialize()
```

This method sets up the initial wakeup criteria.

```
public void processStimulus(Enumeration criteria)
```

This method computes the appropriate transform.

CHAPTER 10

Input Devices and Picking

JAVA 3D provides access to keyboards and mice using the standard Java API for keyboard and mouse support. Additionally, Java 3D provides access to a variety of continuous-input devices such as six-degrees-of-freedom (6DOF) trackers and joysticks.

Continuous-input devices like 6DOF trackers and joysticks have well defined continuous inputs. Trackers produce a position and orientation that Java 3D stores internally as a transformation matrix. Joysticks produce two continuous values in the range [–1.0, 1.0] that Java 3D stores internally as a transformation matrix with an identity rotation (no rotation) and one of the joystick values as the X translation and the other as the Y translation component.

Unfortunately, continuous-input devices do not have the same level of consistency when it comes to their associated switches or buttons. Still, the number of buttons or switches attached to a particular sensing element remains constant across all sensing elements associated with a single device.

10.1 InputDevice Interface

The InputDevice interface specifies an abstract input device that a developer can use in implementing a device driver for a particular device. All implementations of an InputDevice interface must implement all of its methods. Java 3D's input device scheduler uses these methods to interact with specific devices and incorporate their input. In addition to the generic methods that all InputDevices must provide, implementations of an InputDevice will contain whatever device-specific information and methods are necessary to maintain that device's proper functioning.

All input devices consist of a number of Sensor objects that have a direct one-to-one relationship with that device's physical detectors. Sensor objects serve double duty. They not only represent actual physical detectors but they also serve as abstract six-degrees-of-freedom transformations that a Java 3D application can access. The Sensor class is described in more detail in Section 10.2.3, "The Sensor Object."

10.1.1 The Abstract Interface

All input devices implement a consistent interface that allows the initialization, processing of input, and finalization of a particular input device. A device-driver programmer would implement the following methods in whatever device-specific manner is necessary to perform the specified operations.

Constants

`public static final int POLLED`
`public static final int STREAMING`

These flags specify whether the associated device works in polled mode or streaming mode.

Methods

`public abstract boolean initialize()`

This method initializes the device. It returns `true` if initialization succeeded, `false` otherwise.

`public abstract void setProcessingMode(int mode)`
`public abstract int getProcessingMode()`

These methods set and retrieve this device's processing mode.

`public int getSensorCount()`

This method returns the number of Sensor objects associated with this device.

`public Sensor getSensor(int sensorIndex)`

This method returns the specified Sensor associated with this device.

`public abstract void setNominalPositionAndOrientation()`

This method sets the device's current position and orientation as the device's nominal position and orientation (that is, establishes its reference frame relative to the "tracker base" reference frame). This method is most useful in defining a nominal pose in immersive head-tracked situations.

`public abstract void pollAndProcessInput()`

This method first polls the device for data values and then processes the values received from the device.

`public abstract void processStreamInput()`

This method processes the device's streaming input.

`public abstract void close()`

This method closes the device.

10.1.2 Instantiating and Registering a New Device

A browser or applications developer must instantiate whatever system-specific input devices that he or she needs and that exist on the system. This available-device information typically exists in a site configuration file. The browser or application will instantiate the viewing environment as requested by the end user.

The API for instantiating devices is site-specific, but it consists of a device object with a constructor and at least all of the methods specified in the Input-Device interface.

Once instantiated, the browser or application must register the device with the Java 3D input device scheduler. The API for registering devices is specified in Section 8.7, "The View Object." The `addInputDevice` method introduces new devices to the Java 3D environment and the `allInputDevices` method produces an enumeration that allows examination of all available devices within a Java 3D environment.

10.2 Sensors

The Java 3D API provides only an abstract concept of a device. Rather than focusing on issues of devices and device models, it instead defines the concept of a sensor. A sensor consists of a timestamped sequence of input values and the state of the buttons or switches at the time that Java 3D sampled the value. A sensor also contains a hotspot offset specified in that sensor's local coordinate system. If not specified, the hotspot is (0.0, 0.0, 0.0).

Since a typical hardware environment contains multiple sensing elements, Java 3D maintains an array of sensors. Users can access a sensor directly from their Java code or they can assign a sensor to one of Java 3D's predefined 6DOF entities such as UserHead.

10.2.1 Using and Assigning Sensors

Using a sensor is as easy as accessing an object. The application developer writes Java code to extract the associated sensor value from the array of sensors. The developer can then directly apply that value to an element in a scene graph or process the sensor values in whatever way necessary.

Java 3D includes three special six-degrees-of-freedom (6DOF) entities. These include UserHead, DominantHand, and NondominantHand. An application developer can assign or change which sensor drives one of these predefined entities. Java 3D uses the specified sensor to drive the 6DOF entity—most visibly the View. Application developers should use this facility carefully. It is quite easy to get the effect of a WristCam—and very disconcerting as well.

10.2.2 Behind the (Sensor) Scenes

Java 3D does not provide raw tracker or joystick-generated data in a sensor. At a minimum, Java 3D normalizes the raw data using the registration and calibration parameters either provided by or provided for the end user. It additionally may filter and process the data to remove noise and improve latency. The application programmer can suppress this latter effect on a sensor-by-sensor basis.

Unfortunately, tracker or sensor hardware may not always be available or be operational. Thus, Java 3D provides both an available and an enable flag on a per-sensor basis.

10.2.3 The Sensor Object

Java 3D stores its sensor array in the PhysicalEnvironment object. Each Sensor in the array consists of a fixed number of SensorRead objects. Also associated with each SensorRead is its timestamp and the state of that sensor's buttons.

Constants

The Sensor object specifies the following constants.

```
public static final int PREDICT_NONE
public static final int PREDICT_NEXT_FRAME_TIME
```

These flags define the Sensor's predictor type. The first flag defines no prediction. The second flag specifies to generate the value to correspond with the next frame time.

```
public static final int NO_PREDICTOR
public static final int HEAD_PREDICTOR
public static final int HAND_PREDICTOR
```

These flags define the Sensor's predictor policy. The first flag specifies to use no prediction policy. The second flag specifies to assume that the sensor is predicting head position or orientation. The third flag specifies to assume that the sensor is predicting hand position or orientation.

```
public static final int DEFAULT_SENSOR_READ_COUNT
```

This constant specifies the default number of SensorRead objects constructed when no SensorRead count is specified.

Constructors

The Sensor object specifies the following constructors.

```
public Sensor(InputDevice device)
public Sensor(InputDevice device, int sensorReadCount)
public Sensor(InputDevice device, int sensorReadCount,
       int sensorButtonCount)
```

These methods construct a new Sensor object associated with the specified device and consisting of either a default number of SensorReads or `sensorReadCount` number of SensorReads and a hot spot at (0.0, 0.0, 0.0) specified in the sensor's local coordinate system. The default for `sensorButtonCount` is zero.

```
public Sensor(InputDevice device, Point3d hotspot)
public Sensor(InputDevice device, int sensorReadCount,
       Point3d hotspot)
public Sensor(InputDevice device, int sensorReadCount,
       int sensorButtonCount, Point3d hotspot)
```

These methods construct a new Sensor object associated with the specified device and consisting of either `sensorReadCount` number of SensorReads or a default number of SensorReads and an offset defining the sensor's hot spot in the sensor's local coordinate system. The default for `sensorButtonCount` is zero.

Methods

```
public void setSensorReadCount(int count)
public final int getSensorReadCount()
public final int getSensorButtonCount()
```

These methods set and retrieve the number of SensorRead objects associated with this sensor and the number of buttons associated with this sensor. Both the number of SensorRead objects and the number of buttons are determined at Sensor construction time.

```
public void getHotspot(Point3d hotspot)
public void setHotspot(Point3d hotspot)
```

These methods set and retrieve the sensor's hotspot offset. The hotspot is specified in the sensor's local coordinate system.

```
public void lastRead(Transform3D read)
public void lastRead(Transform3D read, int kth)
```

These methods extract the most recent sensor reading and the *k*th most recent sensor reading from the Sensor object. In both cases, the methods copy the sensor value into the specified argument.

```
public void getRead(Transform3D read)
public void getRead(Transform3D read, long deltaT)
```

The first method computes the sensor reading consistent with the prediction policy and copies that value into the `read` matrix. The second method computes the sensor reading consistent as of time `deltaT` in the future and copies that value into the `read` matrix. All times are in milliseconds.

```
public long lastTime()
public long lastTime(int k)
```

These methods return the time associated with the most recent sensor reading and with the *k*th most recent sensor reading, respectively.

```
public int lastButtons()
public int lastButtons(int k)
```

These methods return the state of the buttons associated with the most recent sensor reading and the *k*th most recent sensor reading, respectively.

```
public void setPredictor(int predictor)
public int getPredictor()
```

These methods set and retrieve the sensor's predictor type. The predictor type is one of the following: `NO_PREDICTOR`, `HEAD_PREDICTOR`, or `HAND_PREDICTOR`.

```
public void setPredictionPolicy(int policy)
public int getPredictionPolicy()
```

These methods set and retrieve the sensor's predictor policy. The predictor policy is either `PREDICT_NONE` or `PREDICT_NEXT_FRAME_TIME`.

```
public void setDevice(InputDevice device)
public InputDevice getDevice()
```

These methods set and retrieve the sensor's input device.

```
public SensorRead getCurrentSensorRead()
```

This method returns the current number of SensorRead objects per sensor.

```
public void setNextSensorRead(long time, Transform3D transform,
       int buttons)
```

This method sets the next SensorRead object to the specified values, including the next SensorRead's associated time, transformation, and button state array.

10.2.4 The SensorRead Object

A SensorRead object encapsulates all the information associated with a single reading of a sensor.

Constants

```
public final static int MAXIMUM_SENSOR_BUTTON_COUNT
```

This flag determines the maximum number of sensor-attached buttons tracked on a per-sensor basis.

Constructors

The SensorRead object specifies the following constructor.

```
public SensorRead()
```

Creates a new SensorRead object.

Methods

```
public final void set(Transform3D t1)
public final void get(Transform3D result)
```

These methods set and retrieve the SensorRead object's transform. They allow a device to store a new rotation and orientation value into the SensorRead object, and a consumer of that value to access it.

```
public final void setTime(long time)
public final long getTime()
```

These methods set and retrieve the SensorRead object's timestamp. They allow a device to store a new timestamp value into the SensorRead object, and a consumer of that value to access it.

```
public final void setButtons(int values)
public final int getButtons()
```

These methods set and retrieve the SensorRead object's button values. They allow a device to store an integer that encodes the button values into the SensorRead object, and a consumer of those values to access the state of the buttons.

10.3 Picking

Behavior nodes provide the means for building developer-specific picking semantics. An application developer can define custom picking semantics using Java 3D's behavior mechanism (see Chapter 9, "Behaviors and Interpolators"). The developer might wish to define pick semantics that use a mouse to shoot a ray into the virtual universe from the current viewpoint, find the first object along that ray, and highlight that object when the end user releases the mouse button. A typical scenario follows:

1. The application constructs a Behavior node that arms itself to awaken when AWT detects a left-mouse-button-down event.
2. Upon awakening from a left-mouse-button-down event, the behavior
   a. Updates a Switch node to draw a ray that emanates from the center of the screen.
   b. Changes that ray's TransformGroup node so that the ray points in the direction of the current mouse position.
   c. Declares its interest in mouse-move or left-mouse-button-up events.
3. Upon awakening from a mouse-move event, the behavior
   a. Changes that ray's TransformGroup node so that the ray points in the direction of the current mouse position.
   b. Declares its interest in mouse-move or left-mouse-button-up events.
4. Upon awakening from a left-mouse-button-up event, the behavior
   a. Changes that ray's TransformGroup node so that the ray points in the direction of the current mouse position.

b. Intersects the ray with all the objects in the virtual universe to find the first object that the ray intersects.

c. Changes the appearance component of that object's shape node to highlight the selected object.

d. Declares its interest in left-mouse-button-down events.

Java 3D includes helping functions that aid in intersecting various geometric objects with objects in the virtual universe by

- Intersecting an oriented ray with all the objects in the virtual universe. That function can return the first object intersected along that ray, all the objects that intersect that ray, or a list of all the objects along that ray sorted by distance from the ray's origin.
- Intersecting a volume with all the objects in the virtual universe. That function returns a list of all the objects contained in that volume.
- Discovering which vertex within an object is closest to a specified ray.

10.3.1 SceneGraphPath Object

The SceneGraphPath object represents a path from an object to a BranchGroup or Locale object. During picking and intersection tests, the user specifies the subtree of the scene graph that should be tested. The whole tree for a Locale is searched by providing the Locale to the picking or intersection tests.

The SceneGraphPath object represents all the components in the subgraph that have the capability `ENABLE_PICK_REPORTING` set between the root of the subtree and the picked or intersected object. All Link nodes are implicitly enabled for picking.

Constructors

```
public SceneGraphPath()
public SceneGraphPath(Locale root, Node object)
public SceneGraphPath(Locale root, Node nodes[], Node object)
```

These construct and initialize a new SceneGraphPath object. The first form uses default values. The second form specifies the path's Locale object and the object in question. The third form includes an array of nodes that fall in between the Locale and the object in question, and which nodes have their `ENABLE_PICK_REPORTING` capability bit set. The object parameter may be a Group, Shape3D, or Morph node. If any other type of leaf node is specified, an `IllegalArgumentException` is thrown.

Methods

```
public final void set(SceneGraphPath newPath)
public final void setLocale(Locale newLocale)
public final void setObject(Node object)
public final void setNode(int index, Node newNode)
public final void setNodes(Node nodes[])
```

These methods set the path's values. The first method sets the path's interior values. The second method sets the path's Locale to the specified Locale. The third method sets the path's object to the specified object (a Group node, or a Shape3D or Morph leaf node). The fourth method replaces the link node associated with the specified index with the specified newLink. The last method replaces all of the link nodes with the new list of link nodes.

```
public final Locale getLocale()
public final Node getObject()
```

The first method returns the path's Locale. The second method returns the path's object.

```
public final int nodeCount()
public final Node getNode(int index)
```

The first method returns the number of intermediate nodes in this path. The second method returns the node associated with the specified index.

```
public final Transform3D getTransform()
```

This method returns a copy of the transform associated with this SceneGraph-Path. The method returns null if there is no transform associated. If this SceneGraphPath was returned by a Java 3D picking and collision method, the local-coordinate-to-virtual-coordinate transform for this scene graph object at the time of the pick or collision is recorded.

```
public final boolean isSamePath(SceneGraphPath testPath)
```

This method determines whether two SceneGraphPath objects represent the same path in the scene graph. Either object might include a different subset of internal nodes; only the internal link nodes, the Locale, and the Node itself are compared. The paths are not validated for correctness or uniqueness.

```
public boolean equals(SceneGraphPath testPath)
```

This method returns `true` if all of the data members of path `testPath` are equal to the corresponding data members in this SceneGraphPath.

`public int hashCode()`

This method returns a hash number based on the data values in this object. Two different SceneGraphPath objects with identical data values (that is, `trans.equals(SceneGraphPath)` returns `true`) will return the same hash number. Two paths with different data members may return the same hash value, although this is not likely.

`public String toString()`

This method returns a string representation of this object. The string contains the class names of all nodes in the SceneGraphPath.

10.3.2 BranchGroup Node and Locale Node Pick Methods

The following methods are in both the BranchGroup node class and the Locale node class.

```
public final SceneGraphPath[] pickAll(PickShape pickShape)
public final SceneGraphPath[] pickAllSorted(PickShape pickShape)
public final SceneGraphPath pickClosest(PickShape pickShape)
public final SceneGraphPath pickAny(PickShape pickShape)
```

These methods return either an array of SceneGraphPath objects or a single SceneGraphPath object. A SceneGraphPath object describes the entire path from a Locale to an object that intersects the specified PickShape (see Section 10.3.3, “PickShape Object”). The methods that return an array either return all the picked objects or all the picked objects in sorted order starting with the objects “closest” to the eyepoint and ending with the objects farthest from the eyepoint. Methods that return a single SceneGraphPath return a single path object that specifies either the object closest to the eyepoint or any picked object (this latter method also implements the fastest pick operation possible). All ties in testing for closest objects intersected result in an indeterminate order.

10.3.3 PickShape Object

The PickShape object is an abstract class for describing a shape that can be used with the BranchGroup and Locale pick methods. The PickShape object is extended by PickPoint, PickRay, and PickSegment objects.

10.3.4 PickPoint Object

The PickPoint object provides a point to supply to the BranchGroup and Locale pick methods. See also Section 10.3.2, "BranchGroup Node and Locale Node Pick Methods."

Constructors

```
public PickPoint()
public PickPoint(Point3d location)
```

The first constructor creates a PickPoint initialized to (0,0,0). The second constructor creates a PickPoint at the specified location.

Methods

```
public void set(Point3d location)
public void get(Point3d location)
```

These methods set and retrieve the position of this PickPoint.

10.3.5 PickRay Object

The PickRay object is an encapsulation of a ray that is passed to the pick methods in BranchGroup and Locale. See also Section 10.3.2, "BranchGroup Node and Locale Node Pick Methods."

Constructors

```
public PickRay()
public PickRay(Point3d origin, Vector3d direction)
```

The first constructor creates a PickRay initialized with an origin and direction of (0,0,0). The second constructor creates a PickRay cast from the specified `origin` and `direction`.

Methods

```
public void set(Point3d origin, Vector3d direction)
public void get(Point3d origin, Vector3d direction)
```

These methods set and retrieve the ray to point from the specified `origin` in the specified `direction`.

10.3.6 PickSegment Object

The PickSegment object is an encapsulation of a segment that is passed to the pick methods in BranchGroup and Locale. See also Section 10.3.2, “BranchGroup Node and Locale Node Pick Methods.”

Constructors

```
public PickSegment()
public PickSegment(Point3d start, Point3d end)
```

The first constructor creates a PickSegment object with the start and end of the segment initialized to (0,0,0). The second constructor creates a PickSegment object from the specified `start` and `end` points.

Methods

```
public void set(Point3d start, Point3d end)
public void get(Point3d start, Point3d end)
```

These methods set and return the line segment from the `start` point to the `end` point.

CHAPTER 11

Audio Devices

A Java 3D application running on a particular machine could have one of several options available to it for playing the audio image created by the sound renderer. Perhaps the machine on which Java 3D is executing has more than one sound card (for example, one that is a wave table synthesis card and the other with accelerated sound spatialization hardware). Furthermore, suppose there are Java 3D audio device drivers that execute Java 3D audio methods on each of these specific cards. The application would therefore have at least two audio device drivers through which the audio could be produced. For such a case the Java 3D application must choose the audio device driver with which sound rendering is to be performed. Once this audio device is chosen, the application can additionally select the type of audio playback on which device the rendered sound image is to be output. The playback device (headphones or speaker(s)) is physically connected to the port to which the selected device driver outputs.

11.1 AudioDevice Interface

The selection of this device driver is done through methods in the PhysicalEnvironment object (see Section C.9, "The PhysicalEnvironment Object"). The application queries how many audio devices are available. For each device, the user can get the AudioDevice object that describes it and query its characteristics. Once a decision is made about which of the available audio devices to use for a PhysicalEnvironment, the particular device is set into this PhysicalEnvironment's fields. Each PhysicalEnvironment object may use only a single audio device.

The AudioDevice object interface specifies an abstract audio device that creators of Java 3D class libraries would implement for a particular device. Java 3D uses several methods to interact with specific devices. Since all audio devices implement this consistent interface, the user could have a portable means of initializing, setting particular audio device elements, and querying generic characteristics for any audio device.

Constants

```
public final static int HEADPHONES
```

Specifies that audio playback will be through stereo headphones.

```
public final static int MONO_SPEAKER
```

Specifies that audio playback will be through a single speaker some distance away from the listener.

```
public final static int STEREO_SPEAKERS
```

Specifies that audio playback will be through stereo speakers some distance away from, and at some angle to, the listener.

11.1.1 Initialization

Each audio device driver must be initialized. The chosen device driver should be initialized before any Java 3D Sound methods are executed because the implementation of the Sound methods, in general, is potentially device-driver dependent.

Methods

```
public abstract boolean initialize()
```

Initialize the audio device. Exactly what occurs during initialization is implementation dependent. This method provides explicit control by the user over when this initialization occurs.

```
public abstract boolean close()
```

Closes the audio device, releasing resources associated with this device.

11.1.2 Audio Playback

Methods to set and retrieve the audio playback parameters are part of the AudioDevice object. The audio playback information specifies that playback will be through one of the following:

- Stereo headphones.

- A monaural speaker.
- A pair of speakers, equally distant from the listener, both at some angle from the head coordinate system *Z* axis. It's assumed that the speakers are at the same elevation and oriented symmetrically about the listener.

The type of playback chosen affects the sound image generated. Cross-talk cancellation is applied to the audio image if playback over stereo speakers is selected.

Methods

The following methods affect the playback of sound processed by the Java 3D sound renderer.

```
public abstract void setAudioPlaybackType(int type)
public abstract int getAudioPlaybackType()
```

These methods set and retrieve the type of audio playback device (`HEADPHONES`, `MONO_SPEAKER`, or `STEREO_SPEAKERS`) used to output the analog audio from rendering Java 3D Sound nodes.

```
public abstract void setCenterEarToSpeaker(float distance)
public abstract float getCenterEarToSpeaker()
```

These methods set and retrieve the distance in meters from the center ear (the midpoint between the left and right ears) and one of the speakers in the listener's environment. For monaural speaker playback, a typical distance from the listener to the speaker in a workstation cabinet is 0.76 meters. For stereo speakers placed at the sides of the display, this might be 0.82 meters.

```
public abstract void setAngleOffsetToSpeaker(float angle)
public abstract float getAngleOffsetToSpeaker()
```

These methods set and retrieve the angle, in radians, between the vectors from the center ear to each of the speaker transducers and the vectors from the center ear parallel to the head coordinate's Z axis. Speakers placed at the sides of the computer display typically range between 0.175 and 0.350 radians (between 10 and 20 degrees).

```
public abstract PhysicalEnvironment getPhysicalEnvironment()
```

This method returns a reference to the AudioDevice's PhysicalEnvironment object.

11.1.3 Device-Driver-Specific Data

While the sound image created for final output to the playback system is either only monaural or stereo (for this version of Java 3D), most device-driver implementations will mix the left and right image signals generated for each rendered sound source before outputting the final playback image. Each sound source will use *N* input channels of this internal mixer.

Each implemented Java 3D audio device driver will have its own limitations and driver-specific characteristics. These include channel availability and usage (during rendering). Methods for querying these device-driver-specific characteristics are provided below.

Methods

`public abstract int getTotalChannels()`

This method retrieves the maximum number of channels available for Java 3D sound rendering for all sound sources.

`public abstract int getChannelsAvailable()`

During rendering, when Sound nodes are playing, this method returns the number of channels still available to Java 3D for rendering additional Sound nodes.

`public abstract int getChannelsUsedForSound(Sound node)`

This is a deprecated method. This method is now part of the Sound class.

11.2 Instantiating and Registering a New Device

A browser or applications developer must instantiate whatever system-specific audio devices that he or she needs and that exist on the system. This device information typically exists in a site configuration file. The browser or application will instantiate the physical environment as requested by the end user.

The API for instantiating devices is site-specific, but it consists of a device object with a constructor and at least all of the methods specified in the AudioDevice interface.

Once instantiated, the browser or application must register the device with the Java 3D sound scheduler by associating this device with a PhysicalEnvironment object. The `setAudioDevice` method introduces new devices to the Java 3D environment and the `allAudioDevices` method produces an enumeration that allows examination of all available devices within a Java 3D environment. See Section C.9, "The PhysicalEnvironment Object," for more details.

11.3 AudioMixerDevice Interface

The `AudioMixerDevice` interface extension is under construction until the Version 1.1 Java 3D API Specification is frozen. Check the javadoc for details.

The intent is for this interface to be implemented by AudioDevice driver developers. The `AudioMixerDevice` interface methods should not be called by an application. Eventually this interface (when it is stable) will be used by those implementing their own Audio Devices.

C H A P T E R 12

Execution and Rendering Model

JAVA 3D's execution and rendering model assumes the existence of a VirtualUniverse object and an attached scene graph. This scene graph can be minimal and not noticeable from an application's perspective when using immediate-mode rendering, but it must exist.

Java 3D's execution model intertwines with its rendering modes and with behaviors and their scheduling. This chapter first describes the three rendering modes, then describes how an application starts up a Java 3D environment, and finally, it discusses how the various rendering modes work within this framework.

12.1 Three Major Rendering Modes

Java 3D supports three different modes for rendering scenes: immediate mode, retained mode, and compiled-retained mode. These three levels of API support represent a potentially large variation in graphics processing speed and in on-the-fly restructuring.

12.1.1 Immediate Mode

Immediate mode allows maximum flexibility at some cost in rendering speed. The application programmer can either use or ignore the scene graph structure inherent in Java 3D's design. The programmer can choose to draw geometry directly or to define a scene graph. Immediate mode can either be used independently or mixed with retained and/or compiled-retained mode rendering. The immediate-mode API is described in Chapter 13, "Immediate-Mode Rendering."

12.1.2 Retained Mode

Retained mode allows a great deal of the flexibility provided by immediate mode while also providing a substantial increase in rendering speed. All objects defined in the scene graph are accessible and manipulable. The scene graph itself is fully manipulable. The application programmer can rapidly construct the scene graph, create and delete nodes, and instantly “see” the effect of edits. Retained mode also allows maximal access to objects through a general pick capability.

Java 3D’s retained mode allows a programmer to construct objects, insert objects into a database, compose objects, and add behaviors to objects.

In retained mode, Java 3D knows that the programmer has defined objects, knows how the programmer has combined those objects into compound objects or scene graphs, and knows what behaviors or actions the programmer has attached to objects in the database. This knowledge allows Java 3D to perform many optimizations. It can construct specialized data structures that hold an object’s geometry in a manner that enhances the speed at which the Java 3D system can render it. It can compile object behaviors so that they run at maximum speed when invoked. It can flatten transformation manipulations and state changes where possible in the scene graph.

12.1.3 Compiled-retained Mode

Compiled-retained mode allows the Java 3D API to perform an arbitrarily complex series of optimizations including, but not restricted to, geometry compression, scene graph flattening, geometry grouping, and state change clustering.

Compiled-retained mode provides hooks for end-user manipulation and picking. Pick operations return the closest object (in scene graph space) associated with the picked geometry.

Java 3D’s compiled-retained mode ensures effective graphics rendering speed in yet one more way. A programmer can request that Java 3D compile an object or a scene graph. Once compiled, the programmer has minimal access to the internal structure of the object or scene graph. Capability flags provide access to specified components that the application program may need to modify on a continuing basis.

A compiled object or scene graph consists of whatever internal structures Java 3D wishes to create to ensure that objects or scene graphs render at maximal rates. Because Java 3D knows that the majority of the compiled object’s or scene graph’s components will not change, it can perform an extraordinary number of optimizations, including the fusing of multiple objects into one conceptual object, turning an object into compressed geometry, or even breaking an object up into like-kind components and reassembling the like-kind components into new "conceptual objects."

12.2 Instantiating the Render Loop

From an application's perspective, Java 3D's render loop runs continuously. Whenever an application adds a scene branch to the virtual world, that scene branch is instantly visible. This high-level view of the render loop permits concurrent implementations of Java 3D as well as serial implementations. The remainder of this section describes the Java 3D render loop bootstrap process from a serialized perspective. Differences that would appear in concurrent implementations are noted as well.

12.2.1 An Application-level Perspective

First the application must construct its scene graphs. It does this by constructing scene graph nodes and component objects and linking them into self-contained trees with a BranchGroup node as a root. The application next must obtain a reference to any constituent nodes or objects within that branch that it may wish to manipulate. It sets the capabilities of all the objects to match their anticipated use and only then compiles the branch using the BranchGroup's `compile` method. Whether or not it compiles the branch, the application can add it to the virtual universe by adding the BranchGroup to a Locale object. The application repeats this process for each branch it wishes to create. Note that for concurrent Java 3D implementations, whenever an application adds a branch to the active virtual universe, that branch becomes visible.

12.2.2 Retained and Compiled-retained Rendering Modes

This initialization process is identical for retained and compiled-retained modes. In both modes, the application builds a scene graph. In compiled-retained mode, the application then compiles the scene graph. Then the application inserts the (possibly compiled) scene graph into the virtual universe.

CHAPTER 13

Immediate-Mode Rendering

JAVA 3D is fundamentally a scene graph–based API. Most of the constructs in the API are biased toward retained mode and compiled-retained mode rendering. However, there are some applications that want both the control and the flexibility that immediate-mode rendering offers.

Immediate-mode applications can either use or ignore Java 3D's scene graph structure. By using immediate mode, end-user applications have more freedom, but this freedom comes at the expense of performance. In immediate mode, Java 3D has no high-level information concerning graphical objects or their composition. Because it has minimal global knowledge, Java 3D can only perform localized optimizations on behalf of the application programmer.

13.1 Two Styles of Immediate-Mode Rendering

Use of Java 3D's immediate mode falls into one of two categories: pure immediate-mode rendering and mixed-mode rendering in which immediate mode and retained or compiled-retained mode interoperate and render to the same canvas. The Java 3D renderer is idle in pure immediate mode, distinguishing it from mixed-mode rendering.

13.1.1 Pure Immediate-Mode Rendering

Pure immediate-mode rendering provides for those applications and applets that do not want Java 3D to do any automatic rendering of the scene graph. Such applications may not even wish to build a scene graph to represent their graphical data. However, they use Java 3D's attribute objects to set graphics state and Java 3D's geometric objects to render geometry.

A pure immediate mode application must create a minimal set of Java 3D objects before rendering. In addition to a Canvas3D object, the application must create a View object, with its associated PhysicalBody and PhysicalEnvironment objects, and the following scene graph elements: a VirtualUniverse object, a high-resolution Locale object, a BranchGroup node object, a TransformGroup node object with associated transform and, finally, a ViewPlatform leaf node object that defines the position and orientation within the virtual universe that generates the view (see Figure 13-1).

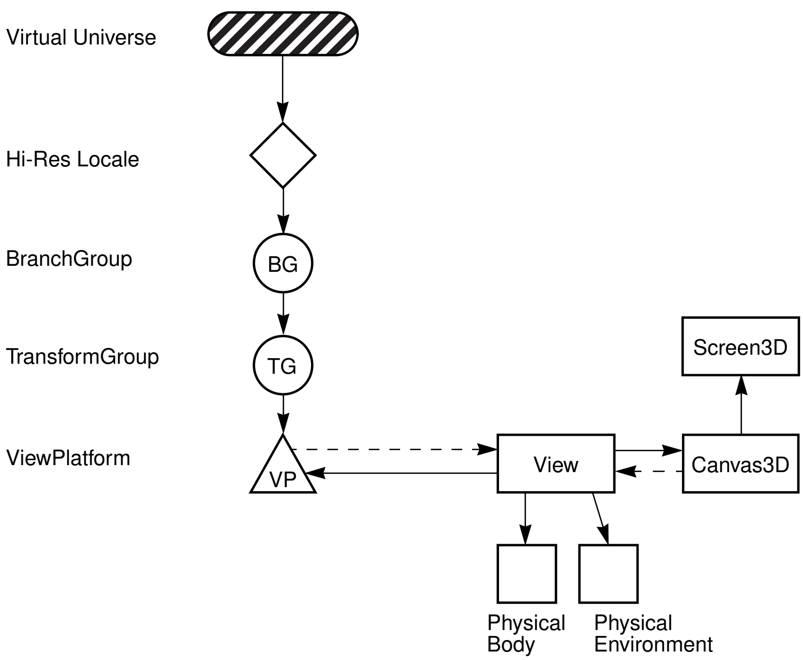


Figure 13-1 Minimal Immediate-Mode Structure

Java 3D provides utility functions that create much of this structure on behalf of a pure immediate-mode application, making it less noticeable from the application's perspective—but the structure must exist.

All rendering is done completely under user control. It is necessary for the user to clear the 3D canvas, render all geometry, and swap the buffers. Additionally, rendering the right and left eye for stereo viewing becomes the sole responsibility of the application.

In pure immediate mode, the user must stop the Java 3D renderer, via the Canvas3D object `stopRenderer()` method, prior to adding the Canvas3D object to an active View object (that is, one that is attached to a live ViewPlatform object).

13.1.2 Mixed-Mode Rendering

Mixing immediate mode and retained or compiled-retained mode requires more structure than pure immediate mode. In mixed mode, the Java 3D renderer is running continuously, rendering the scene graph into the canvas. The basic Java 3D *stereo* rendering loop, executed for each Canvas3D, is as follows:

```
clear canvas (both eyes)
call preRender()                         // user-supplied method
set left eye view
render opaque scene graph objects
call renderField(FIELD_LEFT)             // user-supplied method
render transparent scene graph objects
set right eye view
render opaque scene graph objects again
call renderField(FIELD_RIGHT)            // user-supplied method
render transparent scene graph objects again
call postRender()                        // user-supplied method
synchronize and swap buffers
call postSwap()                          // user-supplied method
```

The basic Java 3D *monoscopic* rendering loop is as follows:

```
clear canvas
call preRender()                         // user-supplied method
set view
render opaque scene graph objects
call renderField(FIELD_ALL)              // user-supplied method
render transparent scene graph objects
call postRender()                        // user-supplied method
synchronize and swap buffers
call postSwap()                          // user-supplied method
```

In both cases, the entire loop, beginning with clearing the canvas and ending with swapping the buffers, defines a frame. The application is given the opportunity to render immediate-mode geometry at any of the clearly identified spots in the rendering loop. A user specifies his or her own rendering methods by extending the Canvas3D class and overriding the `preRender`, `postRender`, `postSwap`, and/or `renderField` methods.

13.2 Canvas3D Methods

The Canvas3D methods that directly affect immediate-mode rendering are described here.

When a Canvas3D object is created, it is initially marked as being started. This means that as soon as the Canvas3D is added to an active View object, the rendering loop will render the scene graph to the canvas. In pure immediate mode the renderer must be stopped (via a call to `stopRenderer`) prior to adding the canvas to an active View object.

Constants

```
public static final int FIELD_LEFT
public static final int FIELD_RIGHT
public static final int FIELD_ALL
```

These constants specify the field that the rendering loop for this Canvas3D is rendering. The `FIELD_LEFT` and `FIELD_RIGHT` values indicate the left and right fields of a field-sequential stereo rendering loop, respectively. The `FIELD_ALL` value indicates a monoscopic or single-pass stereo rendering loop.

Methods

```
public final GraphicsContext3D getGraphicsContext3D()
```

This method retrieves the immediate-mode 3D graphics context associated with this Canvas3D. It creates a new graphics context if one does not already exist. It returns a GraphicsContext3D object that can be used for immediate mode rendering to this Canvas3D.

```
public void preRender()
```

Applications that wish to perform operations in the rendering loop prior to any actual rendering must override this method. The Java 3D rendering loop invokes this method after clearing the canvas and before any rendering has been done for this frame.

```
public void postRender()
```

Applications that wish to perform operations in the rendering loop following any actual rendering must override this method. The Java 3D rendering loop invokes this method after completing all rendering to the canvas for this frame and before the buffer swap.

`public void postSwap()`

Applications that wish to perform operations at the very end of the rendering loop must override this method. The Java 3D rendering loop invokes this method after completing all rendering to this canvas, and all other canvases associated with the current view, for this frame following the buffer swap.

`public void renderField(int fieldDesc)`

Applications that wish to perform operations during the rendering loop must override this function. The Java 3D rendering loop invokes this method, possibly twice, during the loop. It is called once for each field (once per frame on a monoscopic system or once each for the right eye and left eye on a field-sequential stereo system). This method is called after all opaque objects are rendered and before any transparent objects are rendered (subject to restrictions imposed by OrderedGroup nodes). This is intended for use by applications that want to mix retained/compiled-retained mode rendering with some immediate-mode rendering. The `fieldDesc` parameter is the field description: `FIELD_LEFT`, `FIELD_RIGHT`, or `FIELD_ALL`. Applications that wish to work correctly in stereo mode should render the same image for both `FIELD_LEFT` and `FIELD_RIGHT` calls. If Java 3D calls the renderer with `FIELD_ALL`, the immediate-mode rendering only needs to be done once.

`public final void startRenderer()`
`public final void stopRenderer()`

These methods start or stop the Java 3D renderer for this Canvas3D object. If the Java 3D renderer is currently running when `stopRenderer` is called, the rendering will be synchronized before being stopped. No further rendering will be done to this canvas by Java 3D until the renderer is started again. If the Java 3D renderer is not currently running when `startRenderer` is called, any rendering to other Canvas3D objects sharing the same View will be synchronized before this Canvas3D's renderer is (re)started.

`public final void swap()`

This method synchronizes and swaps buffers on a double-buffered canvas for this Canvas3D object. This method may only be called if the Java 3D renderer has been stopped. In the normal case, the renderer automatically swaps the buffer. If the application invokes this method and the canvas has a running Java 3D renderer, a `RestrictedAccessException` exception is thrown.

13.3 API for Immediate Mode

The Java 3D immediate mode allows an application to directly set attributes and draw three-dimensional geometry using the same objects as in Java 3D scene graphs. An immediate-mode application renders by passing these objects to the `set` and `draw` methods of a GraphicsContext3D object.

13.3.1 GraphicsContext3D

The GraphicsContext3D object is used for immediate-mode rendering into a 3D canvas. It is created by, and associated with, a specific Canvas3D object. A GraphicsContext3D class defines methods that manipulate 3D graphics state attributes and draw 3D geometric primitives.

Constructors

There are no publicly accessible constructors of GraphicsContext3D. An application obtains a 3D graphics context object from the Canvas3D object into which the application wishes to render by using the `getGraphicsContext3D` method.

The Canvas3D object creates a new GraphicsContext3D the first time an application invokes `getGraphicsContext3D`. A new GraphicsContext3D initializes its state variables to the following defaults:

Background object: null
Fog object: null
Appearance object: null
List of Light objects: empty
High-Res coordinates: (0, 0, 0)
modelTransform: identity
AuralAttributes object: null
List of Sound objects: empty

Methods

```
public final Canvas3D getCanvas3D()
```

This method gets the Canvas3D that created this GraphicsContext3D.

```
public final void setAppearance(Appearance appearance)
public final Appearance getAppearance()
```

These methods access or modify the current Appearance component object used by this 3D graphics context. The graphics context stores a reference to the specified Appearance object. This means that the application may modify individual appearance attributes by using the appropriate methods on the Appearance object (see Section 7.1.1, "Appearance Object"). The Appearance component object must not be part of a live scene graph, nor may it subsequently be made part of a live scene graph—an `IllegalSharingException` is thrown in such cases. If the Appearance object is `null`, default values will be used for all appearance attributes—it is as if an Appearance node were created using the default constructor.

```
public final void setBackground(Background background)
public final Background getBackground()
```

These methods access or modify the current Background leaf node object used by this 3D graphics context. The graphics context stores a reference to the specified Background node. This means that the application may modify the background color or image by using the appropriate methods on the Background node object (see Section 5.4, "Background Node"). The Background node must not be part of a live scene graph, nor may it subsequently be made part of a live scene graph—an `IllegalSharingException` is thrown in such cases. If the Background object is `null`, the default background color of black (0,0,0) is used to clear the canvas prior to rendering a new frame. The Background node's application region is ignored for immediate-mode rendering.

```
public final void setFog(Fog fog)
public final Fog getFog()
```

These methods access or modify the current Fog leaf node object used by this 3D graphics context. The graphics context stores a reference to the specified Fog node. This means that the application may modify the fog attributes using the appropriate methods on the Fog node object (see Section 5.6, "Fog Node"). The Fog node must not be part of a live scene graph, nor may it subsequently be made part of a live scene graph—an `IllegalSharingException` is thrown in such cases. If the Fog object is `null`, fog is disabled. Both the region of influence and the hierarchical scope of the Fog node are ignored for immediate-mode rendering.

```
public final void addLight(Light light)
public final void insertLight(Light light, int index)
public final void setLight(Light light, int index)
public final Light getLight(int index)
public final void removeLight(int index)
```

```
public final int numLights()
public final Enumeration getAllLights()
```

These methods access or modify the list of lights used by this 3D graphics context. The `addLight` method adds a new light to the end of the list of lights. The `insertLight` method inserts a new light before the light at the specified index. The `setLight` method replaces the light at the specified index with the light provided. The `removeLight` method removes the light at the specified index. The `numLights` method returns a count of the number of lights in the list. The `getLight` method returns the light at the specified index. The `getAllLights` method retrieves the Enumeration object of all lights.

The graphics context stores a reference to each light object in the list of lights. This means that the application may modify the light attributes for any of the lights using the appropriate methods on that Light node object (see Section 5.7, "Light Node"). None of the Light nodes in the list of lights may be part of a live scene graph, nor may they subsequently be made part of a live scene graph—an `IllegalSharingException` is thrown in such cases. Adding a `null` Light object to the list will result in a `NullPointerException`. Both the region of influence and the hierarchical scope of all lights in the list are ignored for immediate-mode rendering.

```
public void setHiRes(int x[], int y[], int z[])
public void setHiRes(HiResCoord hiRes)
public void getHiRes(HiResCoord hiRes)
```

These methods access or modify the high-resolution coordinates of this graphics context to the location specified by the parameters provided. In the first method, the parameters `x`, `y`, and `z` are arrays of eight 32-bit integers that specify the high-resolution coordinates point.

```
public void setModelTransform(Transform3D t)
public void multiplyModelTransform(Transform3D t)
public void getModelTransform(Transform3D t)
```

These methods access or modify the current model transform. The `multiplyModelTransform` method multiplies the current model transform by the specified transform and stores the result back into the current model transform. The specified transformation must be affine. A `BadTransformException` is thrown (see Section D.1, "BadTransformException") if an attempt is made to specify an illegal Transform3D.

`public final void setAuralAttributes(AuralAttributes attributes)`
`public final AuralAttributes getAuralAttributes()`

These methods access or modify the current AuralAttributes component object used by this 3D graphics context. The graphics context stores a reference to the specified AuralAttributes object. This means that the application may modify individual audio attributes by using the appropriate methods in the Aural-Attributes object (see Section 7.1.15, “AuralAttributes Object”). The Aural-Attributes component object must not be part of a live scene graph, nor may it subsequently be made part of a live scene graph—an `IllegalSharingException` is thrown in such cases. If the AuralAttributes object is `null`, default values will be used for all audio attributes—it is as if an AuralAttributes object were created using the default constructor.

`public final void readRaster(Raster raster)`

This method reads an image from the frame buffer and copies it into the Image-Component or DepthComponent objects referenced by the specified Raster object. All parameters of the Raster object and the component ImageComponent or DepthComponent objects must be set to the desired values prior to calling this method. These values determine the location, size, and format of the pixel data that is read.

`public final void clear()`

This method clears the canvas to the color or image specified by the current Background leaf node object.

`public final void draw(Geometry geometry)`
`public final void draw(Shape3D shape)`

The first `draw` method draws the specified Geometry component object using the current state in the graphics context. The second `draw` method draws the specified Shape3D leaf node object. This is a convenience method that is identical to calling the `setAppearance(Appearance)` and `draw(Geometry)` methods passing the Appearance and Geometry component objects of the specified Shape3D nodes as arguments.

`public final void addSound(Sound sound)`
`public final void insertSound(Sound sound, int index)`
`public final void setSound(Sound sound, int index)`
`public final Sound getSound(int index)`
`public final void removeSound(int index)`
`public final int numSounds()`

```
public final boolean isSoundPlaying(int index)
public final Enumeration getAllSounds()
```

These methods access or modify the list of sounds used by this 3D graphics context. The `addSound` method appends the specified sound to this graphics context's list of sounds. The `insertSound` method inserts the specified sound at the specified index location. The `setSound` method replaces the specified sound with the sound provided. The `removeSound` method removes the sound at the specified index location. The `numSounds` method retrieves the current number of sounds in this graphics context. The `getSound` method retrieves the index-selected sound. The `isSoundPlaying` method retrieves the sound-playing flag. The `getAllSounds` method retrieves the Enumeration object of all the sounds.

The graphics context stores a reference to each sound object in the list of sounds. This means that the application may modify the sound attributes for any of the sounds by using the appropriate methods on that Sound node object (see Section 5.8, "Sound Node"). None of the Sound nodes in the list of sounds may be part of a live scene graph, nor may they subsequently be made part of a live scene graph—an `IllegalSharingException` is thrown in such cases. Adding a `null` Sound object to the list results in a `NullPointerException`. If the list of sounds is empty, sound rendering is disabled.

Adding or inserting a sound to the list of sounds implicitly starts the sound playing. Once a sound is finished playing, it can be restarted by setting the sound's enable flag to `true`. The scheduling region of all sounds in the list is ignored for immediate-mode rendering.

APPENDIX A

Math Objects

MATHEMATICAL objects allow Java 3D users to represent and manipulate low-level mathematical constructs such as vectors and matrices. Math objects also define specific operations that allow users to manipulate them in appropriate ways.

Java 3D needs these vector and matrix math classes. It uses them internally and also makes them available to applications for their use. However, they are not part of Java 3D. Rather, they are defined here for convenience. These classes will become more widely distributed, which is why Java 3D defines them as a separate `javax.vecmath` package. Figure A-1 shows the math object hierarchy.

A.1 Tuple Objects

Java 3D uses tuple objects to represent and manipulate two-, three-, and four-element values.

A.1.1 Tuple2f Class

The Tuple2f class is a generic two-element tuple mostly used for specifying points and vectors made up of single-precision floating-point *x*,*y* coordinates.

Variables

The component values of a Tuple2f are directly accessible through the public variables `x` and `y`. To access the `x` component of a Tuple2f called `upperLeftCorner`, a programmer would write `upperLeftCorner.x`. The programmer would access the `y` component similarly.

Tuple Objects

- Tuple2f
  - Point2f
  - TexCoord2f
  - Vector2f
- Tuple3b
  - Color3b
- Tuple3d
  - Point3d
  - Vector3d
- Tuple3f
  - Color3f
  - Point3f
  - TexCoord3f
  - Vector3f
- Tuple4b
  - Color4b
- Tuple4d
  - Point4d
  - Quat4d
  - Vector4d
- Tuple4f
  - Color4f
  - Point4f
  - Quat4f
  - Vector4f
- AxisAngle4d
- AxisAngle4f
- GVector

Matrix Objects

- Matrix3f
- Matrix3d
- Matrix4f
- Matrix4d
- GMatrix

Figure A-1 Math Object Hierarchy

```
public float x
public float y
```

The *x* and *y* coordinates, respectively.

Constructors

```
public Tuple2f(float x, float y)
public Tuple2f(float t[])
```

```
public Tuple2f(Tuple2f t1)
public Tuple2f()
```

These four constructors each return a new Tuple2f. The first constructor generates a Tuple2f from two floating-point numbers *x* and *y*. The second constructor generates a Tuple2f from the first two elements of array `t`. The third constructor generates a Tuple2f from the tuple `t1`. The final constructor generates a Tuple2f with the value of (0.0, 0.0).

Methods

```
public final void set(float x, float y)
public final void set(float t[])
public final void set(Tuple2f t1)
public final void get(float t[])
```

The `set` methods set the value of tuple `this` to the values provided. The `get` method copies the values of the elements of this tuple into the array `t`.

```
public final void add(Tuple2f t1, Tuple2f t2)
public final void add(Tuple2f t1)
public final void sub(Tuple2f t1, Tuple2f t2)
public final void sub(Tuple2f t1)
```

The first `add` method computes the element-by-element sum of tuples `t1` and `t2`, placing the result in `this`. The second `add` method computes the element-by-element sum of this tuple and tuple `t1`, placing the result in `this`. The first `sub` method performs an element-by-element subtraction of tuple `t2` from tuple `t1` and places the result in `this` (this = t1 – t2). The second `sub` method performs an element-by-element subtraction of `t1` from `this` and places the result in `this` (this = this – t1).

```
public final void negate(Tuple2f t1)
public final void negate()
```

The first `negate` method sets the values of this tuple to the negative of the values from tuple `t1`. The second `negate` method negates the tuple `this` and places the resulting tuple back into `this`.

```
public final void scale(float s, Tuple2f t1)
public final void scale(float s)
public final void scaleAdd(float s, Tuple2f t1)
public final void scaleAdd(float s, Tuple2f t1, Tuple2f t2)
```

The first `scale` method multiplies each element of the tuple `t1` by the scale factor `s` and places the resulting scaled tuple into `this`. The second `scale` method multiplies each element of this tuple by the scale factor `s` and places the resulting scaled tuple into `this`. The first `scaleAdd` method scales this tuple by the scale factor `s`, adds the result to tuple `t1`, and places the result into the tuple `this` (this = s*this + t1). The second `scaleAdd` method scales tuple `t1` by the scale factor `s`, adds the result to tuple `t2`, then places the result into the tuple `this` (this = s*t1 + t2)

```
public final void absolute()
public final void absolute(Tuple2f t)
```

The first `absolute` method sets each component of this tuple to its absolute value. The second `absolute` method sets each component of this tuple to the absolute value of the corresponding component in tuple `t`.

```
public final void clamp(float min, float max)
public final void clamp(float min, float max, Tuple2f t)
public final void clampMin(float min)
public final void clampMin(float min, Tuple2f t)
public final void clampMax(float max)
public final void clampMax(float max, Tuple2f t)
```

The first `clamp` method clamps this tuple to the range [`min`, `max`]. The second `clamp` method clamps the values from tuple `t` to the range [`min`, `max`] and assigns these clamped values to this tuple. The first `clampMin` method clamps each value of this tuple to the `min` parameter. The second `clampMin` method clamps each value of the tuple `t` and assigns these clamped values to this tuple. The first `clampMax` method clamps each value of this tuple to the `max` parameter. The second `clampMax` method clamps each value of tuple `t` to the `max` parameter and assigns these clamped values to this tuple. In each method the values of tuple `t` remain unchanged.

```
public final void interpolate(Tuple2f t1, Tuple2f t2, float alpha)
public final void interpolate(Tuple2f t1, float alpha)
```

The first method linearly interpolates between tuples `t1` and `t2` and places the result into this tuple (this = alpha * t1 + (1 – alpha) * t2). The second method linearly interpolates between this tuple and tuple `t1` and places the result into this tuple (this = alpha * this + (1 – alpha) * t1).

```
public boolean equals(Tuple2f t1)
```

This method returns `true` if all of the data members of tuple `t1` are equal to the corresponding data members in this tuple.

`public boolean epsilonEquals(Tuple2f t1, float epsilon)`

This method returns `true` if the L_∞ distance between this tuple and tuple `t1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2)]$$

`public int hashCode()`

The `hashCode` method returns a hash number based on the data values in this object. Two Tuple2f objects with identical data values (that is, `equals(Tuple2f)` returns `true`) will return the same hash number. Two objects with different data members may return the same hash number, although this is not likely.

`public String toString()`

This method returns a string that contains the values of this Tuple2f.

A.1.1.1 Point2f Class

The Point2f class extends Tuple2f. The Point2f is a two-element point represented by single-precision floating-point *x*,*y* coordinates.

Constructors

```
public Point2f(float x, float y)
public Point2f(float p[])
public Point2f(Point2f p1)
public Point2f(Tuple2f t1)
public Point2f()
```

These four constructors each return a new Point2f. The first constructor generates a Point2f from two floating-point numbers *x* and *y*. The second constructor generates a Point2f from the first two elements of array `p`. The third constructor generates a Point2f from the point `p1`. The fourth constructor generates a Point2f from the Tuple2f `t1`. The final constructor generates a Point2f with the value of (0.0, 0.0).

Methods

```
public final float distanceSquared(Point2f p1)
public final float distance(Point2f p1)
```

The `distanceSquared` method computes the square of the Euclidean distance between this point and point `p1` and returns the result. The `distance` method computes the Euclidean distance between this point and point `p1` and returns the result.

`public final float distanceL1(Point2f p1)`

This method computes the L_1 (Manhattan) distance between this point and point `p1`. The L_1 distance is equal to

$$\text{abs}(x1 - x2) + \text{abs}(y1 - y2)$$

`public final float distanceLinf(Point2f p1)`

This method computes the L_∞ distance between this point and point `p1`. The L_∞ distance is equal to

$$\text{MAX}[\text{abs}(x1 - x2), \text{abs}(y1 - y2)]$$

A.1.1.2 Vector2f Class

The Vector2f class extends Tuple2f. The Vector2f is a two-element vector represented by single-precision floating-point *x*,*y* coordinates.

Constructors

```
public Vector2f(float x, float y)
public Vector2f(float v[])
public Vector2f(Vector2f v1)
public Vector2f(Tuple2f t1)
public Vector2f()
```

These four constructors each return a new Vector2f. The first constructor generates a Vector2f from two floating-point numbers `x` and `y`. The second constructor generates a Vector2f from the first two elements of array `v`. The third constructor generates a Vector2f from the vector `v1`. The fourth constructor generates a Vector2f from the specified Tuple2f. The final constructor generates a Vector2f with the value of (0.0, 0.0).

Methods

`public final float dot(Vector2f v1)`

The `dot` method computes the dot product between this vector and vector `v1` and returns the resulting value.

```
public final float lengthSquared()
public final float length()
```

The `lengthSquared` method computes the square of the length of the vector `this` and returns its length as a single-precision floating-point number. The `length` method computes the length of the vector `this` and returns its length as a single-precision floating-point number.

```
public final void normalize(Vector2f v1)
public final void normalize()
```

The first `normalize` method normalizes the vector `v1` to unit length and places the result in `this`. The second `normalize` method normalizes the vector `this` and places the resulting unit vector back into `this`.

```
public final float angle(Vector2f v1)
```

This method returns the angle, in radians, between this vector and vector `v1`. The return value is constrained to the range $[0, \pi]$.

A.1.1.3 TexCoord2f Class

The TexCoord2f class is a subset of Tuple2f. The TexCoord2f is a two-element vector represented by single-precision floating-point x,y coordinates.

Constructors

```
public TexCoord2f(float x, float y)
public TexCoord2f(float v[])
public TexCoord2f(TexCoord2f v1)
public TexCoord2f(Tuple2f t1)
public TexCoord2f()
```

These four constructors each return a new TexCoord2f. The first constructor generates a TexCoord2f from two floating-point numbers `x` and `y`. The second constructor generates a TexCoord2f from the first two elements of array `v`. The third constructor generates a TexCoord2f from the TexCoord2f `v1`. The fourth constructor generates a TexCoord2f from the Tuple2f `t1`. The final constructor generates a TexCoord2f with the value of (0.0, 0.0).

A.1.2 Tuple3b Class

The Tuple3b class is used for colors. This class represents a three-byte tuple.

Variables

The component values of a Tuple3b are directly accessible through the public variables `x`, `y`, and `z`. To access the `x` (red) component of a Tuple3b called `myColor`, a programmer would write `myColor.x`. The programmer would access the `y` (green) and `z` (blue) components similarly.

Note: Java defines a byte as a signed integer in the range [−128, 127]. However, colors are more typically represented by values in the range [0, 255]. Java 3D recognizes this and, in those cases where Color3b is used to represent color, treats the bytes as if the range were [0, 255].

```
public byte x
public byte y
public byte z
```

The red, green, and blue values, respectively.

Constructors

```
public Tuple3b(byte b1, byte b2, byte b3)
public Tuple3b(byte t[])
public Tuple3b(Tuple3b t1)
public Tuple3b()
```

These four constructors each return a new Tuple3b. The first constructor generates a Tuple3b from three bytes `b1`, `b2`, and `b3`. The second constructor generates a Tuple3b from the first three elements of array `t`. The third constructor generates a Tuple3b from the byte-precision Tuple3b `t1`. The final constructor generates a Tuple3b with the value of (0.0, 0.0, 0.0).

Methods

```
public String toString()
```

This method returns a string that contains the values of this Tuple3b.

```
public final void set(byte t[])
public final void set(Tuple3b t1)
public final void get(byte t[])
public final void get(Tuple3b t1)
```

The first `set` method sets the values of the *x*, *y*, and *z* data members of this Tuple3b to the values in the array `t` of length three. The second `set` method sets the values of the *x*, *y*, and *z* data members of this Tuple3b to the values in the argument tuple `t1`. The first `get` method places the values of the *x*, *y*, and *z* components of this Tuple3b into the array `t` of length three. The second `get` method places the values of the *x*, *y*, and *z* components of this Tuple3b into the tuple `t1`.

```
public boolean equals(Tuple3b t1)
```

This method returns `true` if all of the data members of Tuple3b `t1` are equal to the corresponding data members in this tuple.

```
public int hashCode()
```

This method returns a hash number based on the data values in this object. Two different Tuple3b objects with identical data values (that is, `equals(Tuple3b)` returns `true`) will return the same hash number. Two tuples with different data members may return the same hash value, although this is not likely.

A.1.2.1 Color3b Class

The Color3b class extends Tuple3b and represents three-byte color values.

Constructors

```
public Color3b(byte c1, byte c2, byte c3)
public Color3b(byte c[])
public Color3b(Color3b c1)
public Color3b(Tuple3b t1)
public Color3b()
```

These four constructors each return a new Color3b. The first constructor generates a Color3b from three bytes `c1`, `c2`, and `c3`. The second constructor generates a Color3b from the first three elements of array `c`. The third constructor generates a Color3b from the byte-precision Color3b `c1`. The fourth constructor generates a Color3b from the tuple `t1`. The final constructor generates a Color3b with the value of (0.0, 0.0, 0.0).

A.1.3 Tuple3d Class

The Tuple3d class is a generic three-element tuple represented by double-precision floating-point *x*, *y*, and *z* coordinates.

Variables

The component values of a Tuple3d are directly accessible through the public variables `x`, `y`, and `z`. To access the `x` component of a Tuple3d called `upperLeftCorner`, a programmer would write `upperLeftCorner.x`. The programmer would access the `y` and `z` components similarly.

```
public double x
public double y
public double z
```

The *x*, *y*, and *z* coordinates, respectively.

Constructors

```
public Tuple3d(double x, double y, double z)
public Tuple3d(double t[])
public Tuple3d(Tuple3d t1)
public Tuple3d(Tuple3f t1)
public Tuple3d()
```

These five constructors each return a new Tuple3d. The first constructor generates a Tuple3d from three floating-point numbers `x`, `y`, and `z`. The second constructor generates a Tuple3d from the first three elements of array `t`. The third constructor generates a Tuple3d from the double-precision Tuple3d `t1`. The fourth constructor generates a Tuple3d from the single-precision Tuple3f `t1`. The final constructor generates a Tuple3d with the value of (0.0, 0.0, 0.0).

Methods

```
public final void set(double x, double y, double z)
public final void set(double t[])
public final void set(Tuple3d t1)
public final void set(Tuple3f t1)
public final void get(double t[])
public final void get(Tuple3d t)
```

The four `set` methods set the value of tuple `this` to the values specified or to the values of the specified vectors. The two `get` methods copy the `x`, `y`, and `z` values into the array `t` of length three.

`public final void add(Tuple3d t1, Tuple3d t2)`
`public final void add(Tuple3d t1)`
`public final void sub(Tuple3d t1, Tuple3d t2)`
`public final void sub(Tuple3d t1)`

The first `add` method computes the element-by-element sum of tuples `t1` and `t2` and places the result in `this`. The second `add` method computes the element-by-element sum of this tuple and tuple `t1` and places the result into `this`. The first `sub` method performs an element-by-element subtraction of tuple `t2` from tuple `t1` and places the result in `this` (this = t1 – t2). The second `sub` method performs an element-by-element subtraction of tuple `t1` from this tuple and places the result in `this` (this = this – t1).

`public final void negate(Tuple3d t1)`
`public final void negate()`

The first `negate` method sets the values of this tuple to the negative of the values from tuple `t1`. The second `negate` method negates the tuple `this` and places the resulting tuple back into `this`.

`public final void scale(double s, Tuple3d t1)`
`public final void scale(double s)`
`public final void scaleAdd(double s, Tuple3f t1)`
`public final void scaleAdd(double s, Tuple3d t1, Tuple3d t2)`

The first `scale` method multiplies each element of the tuple `t1` by the scale factor `s` and places the resulting scaled tuple into `this`. The second `scale` method multiplies each element of `this` tuple by the scale factor `s` and places the resulting scaled tuple back into `this`. The first `scaleAdd` method scales this tuple by the scale factor `s`, adds the result to tuple `t1`, and places the result into tuple `this` (this = s*this + t1). The second `scaleAdd` method scales the tuple `t1` by the scale factor `s`, adds the result to the tuple `t2`, and places the result into the tuple `this` (this = s*t1 + t2).

`public String toString()`

This method returns a string that contains the values of this Tuple3d. The form is `(x, y, z)`.

`public int hashCode()`

This method returns a hash number based on the data values in this object. Two different Tuple3d objects with identical data values (that is, `equals(Tuple3d)` returns `true`) will return the same hash number. Two tuples with different data members may return the same hash value, although this is not likely.

`public boolean equals(Tuple3d v1)`

This method returns `true` if all of the data members of Tuple3d `v1` are equal to the corresponding data members in this Tuple3d.

`public boolean epsilonEquals(Tuple3d t1, double epsilon)`

This method returns `true` if the L_∞ distance between this tuple and tuple `t1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2), \mathrm{abs}(z1 - z2)]$$

`public final void absolute()`
`public final void absolute(Tuple3d t)`

The first `absolute` method sets each component of this tuple to its absolute value. The second `absolute` method sets each component of this tuple to the absolute value of the corresponding component in tuple `t`.

`public final void clamp(float min, float max)`
`public final void clamp(float min, float max, Tuple3d t)`
`public final void clampMin(float min)`
`public final void clampMin(float min, Tuple3d t)`
`public final void clampMax(float max)`
`public final void clampMax(float max, Tuple3dt)`

The first `clamp` method clamps this tuple to the range [`min`, `max`]. The second `clamp` method clamps the values from tuple `t` to the range [`min`, `max`] and assigns these clamped values to this tuple. The first `clampMin` method clamps each value of this tuple to the `min` parameter. The second `clampMin` method clamps each value of the tuple `t` and assigns these clamped values to this tuple. The first `clampMax` method clamps each value of this tuple to the `max` parameter. The second `clampMax` method clamps each value of tuple `t` to the `max` parameter and assigns these clamped values to this tuple. In each method, the values of tuple `t` remain unchanged.

`public final void interpolate(Tuple3d t1, Tuple3d t2, float alpha)`
`public final void interpolate(Tuple3d t1, float alpha)`

The first `interpolate` method linearly interpolates between tuples `t1` and `t2` and places the result into this tuple (this = alpha * t1 + (1 – alpha) * t2). The second `interpolate` method linearly interpolates between this tuple and tuple `t1` and places the result into this tuple (this = alpha * this + (1 – alpha) * t1).

A.1.3.1 Point3d Class

The Point3d class extends Tuple3d. The Point3d is a three-element point represented by double-precision floating-point x, y, and z coordinates.

Constructors

```
public Point3d(double x, double y, double z)
public Point3d(double p[])
public Point3d(Point3d p1)
public Point3d(Point3f p1)
public Point3d(Tuple3d t1)
public Point3d(Tuple3f t1)
public Point3d()
```

These five constructors each return a new Point3d. The first constructor generates a Point3d from three floating-point numbers `x`, `y`, and `z`. The second constructor generates a Point3d from the first three elements of array `p`. The third constructor generates a Point3d from the double-precision Point3d `p1`. The fourth constructor generates a Point3d from the single-precision Point3f `p1`. The fifth and sixth constructors generate a Point3d from the tuple `t1`. The final constructor generates a Point3d with the value of (0.0, 0.0, 0.0).

Methods

```
public final double distanceSquared(Point3d p1)
public final double distance(Point3d p1)
```

The `distanceSquared` method computes the square of the Euclidean distance between this Point3d and the Point3d `p1` and returns the result. The `distance` method computes the Euclidean distance between this Point3d and the Point3d `p1` and returns the result.

```
public final float distanceL1(Point3d p1)
```

This method computes the L_1 (Manhattan) distance between this point and point `p1`. The L_1 distance is equal to

$$\text{abs}(x1 - x2) + \text{abs}(y1 - y2) + \text{abs}(z1 - z2)$$

```
public final float distanceLinf(Point3d p1)
```

This method computes the L_∞ distance between this point and point `p1`. The L_∞ distance is equal to

$$\text{MAX}[\text{abs}(x1 - x2), \text{abs}(y1 - y2), \text{abs}(z1 - z2)]$$

```
public final void project(Point4d p1)
```

This method multiplies each of the `x`, `y`, and `z` components of the Point4d parameter `p1` by 1/`w` and places the projected values into this point.

A.1.3.2 Vector3d Class

The Vector3d class extends Tuple3d. The Vector3d is a three-element vector represented by double-precision floating-point *x*, *y*, and *z* coordinates. If this value represents a normal, it should be normalized.

Constructors

```
public Vector3d(double x, double y, double z)
public Vector3d(double v[])
public Vector3d(Vector3d v1)
public Vector3d(Vector3f v1)
public Vector3d(Tuple3d t1)
public Vector3d(Tuple3f t1)
public Vector3d()
```

These five constructors each return a new Vector3d. The first constructor generates a Vector3d from three floating-point numbers `x`, `y`, and `z`. The second constructor generates a Vector3d from the first three elements of array `v`. The third constructor generates a Vector3d from the double-precision vector `v1`. The fourth constructor generates a Vector3d from the single-precision vector `v1`. The fifth and sixth constructors generate a Vector3d from the tuple `t1`. The final constructor generates a Vector3d with the value of (0.0, 0.0, 0.0).

Methods

```
public final void cross(Vector3d v1, Vector3d v2)
```

The `cross` method computes the vector cross-product of vectors `v1` and `v2` and places the result in `this`.

```
public final void normalize(Vector3d v1)
public final void normalize()
```

The first `normalize` method normalizes the vector `v1` to unit length and places the result in `this`. The second `normalize` method normalizes the vector `this` and places the resulting unit vector back into `this`.

`public final double dot(Vector3d v1)`

The `dot` method returns the dot product of this vector and vector `v1`.

`public final double lengthSquared()`
`public final double length()`

The `lengthSquared` method returns the squared length of this vector. The `length` method returns the length of this vector.

`public final double angle(Vector3d v1)`

This method returns the angle, in radians, between this vector and the vector `v1` parameter. The return value is constrained to the range [0, π].

A.1.4 Tuple3f Class

The Tuple3f class is a generic three-element tuple represented by single-precision floating-point *x*, *y*, and *z* coordinates.

Variables

The component values of a Tuple3f are directly accessible through the public variables `x`, `y`, and `z`. To access the `x` component of a Tuple3f called `upperLeftCorner`, a programmer would write `upperLeftCorner.x`. The programmer would access the `y` and `z` components similarly.

`public float x`
`public float y`
`public float z`

The *x*, *y*, and *z* coordinates, respectively.

Constructors

`public Tuple3f(float x, float y, float z)`
`public Tuple3f(float t[])`
`public Tuple3f(Tuple3d t1)`
`public Tuple3f(Tuple3f t1)`
`public Tuple3f()`

These five constructors each return a new Tuple3f. The first constructor generates a Tuple3f from three floating-point numbers `x`, `y`, and `z`. The second constructor generates a Tuple3f from the first three elements of array `t`. The third constructor generates a Tuple3f from the double-precision Tuple3d `t1`. The fourth constructor generates a Tuple3f from the single-precision Tuple3f `t1`. The final constructor generates a Tuple3f with the value of (0.0, 0.0, 0.0).

Methods

```
public String toString()
```

This method returns a string that contains the values of this Tuple3f.

```
public final void set(float x, float y, float z)
public final void set(float t[])
public final void set(Tuple3f t1)
public final void set(Tuple3d t1)
public final void get(float t[])
public final void get(Tuple3f t)
```

The four `set` methods set the value of vector `this` to the coordinates provided or to the values of the vectors provided. The first `get` method gets the value of this vector and copies the values into the array `t`. The second `get` method gets the value of this vector and copies the values into tuple `t`.

```
public final void add(Tuple3f t1, Tuple3f t2)
public final void add(Tuple3f t1)
public final void sub(Tuple3f t1, Tuple3f t2)
public final void sub(Tuple3f t1)
```

The first `add` method computes the element-by-element sum of tuples `t1` and `t2`, placing the result in `this`. The second `add` method computes the element-by-element sum of `this` and tuple `t1` and places the result in `this`. The first `sub` method performs an element-by-element subtraction of tuple `t2` from tuple `t1` and places the result in `this` (this = t1 – t2). The second `sub` method performs an element-by-element subtraction of tuple `t1` from this tuple and places the result into `this` (this = this – t1).

```
public final void negate(Tuple3f t1)
public final void negate()
```

The first `negate` method sets the values of this tuple to the negative of the values from tuple `t1`. The second `negate` method negates the vector `this` and places the resulting tuple back into `this`.

```
public final void scale(float s, Tuple3f t1)
public final void scale(float s)
public final void scaleAdd(float s, Tuple3f t1)
public final void scaleAdd(float s, Tuple3f t1, Tuple3f t2)
```

The first `scale` method multiplies each element of the vector `t1` by the scale factor `s` and places the resulting scaled vector into `this`. The second `scale` method multiples the vector `this` by the scale factor `s` and replaces `this` with the scaled value. The first `scaleAdd` method scales this tuple by the scale factor `s`, adds the result to tuple `t1`, and places the result into tuple `this` (this = s*this + t1). The second `scaleAdd` method scales the tuple `t1` by the scale factor `s`, adds the result to the tuple `t2`, and places the result into the tuple `this` (this = s*t1 + t2).

```
public boolean equals(Tuple3f t1)
```

This method returns `true` if all of the data members of tuple `t1` are equal to the corresponding data members in this Tuple3f.

```
public boolean epsilonEquals(Tuple3f t1, float epsilon)
```

This method returns `true` if the L_∞ distance between this tuple and tuple `t1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2), \mathrm{abs}(z1 - z2)]$$

```
public final void absolute()
public final void absolute(Tuple3f t)
```

The first `absolute` method sets each component of this tuple to its absolute value. The second `absolute` method sets each component of this tuple to the absolute value of the corresponding component in tuple `t`.

```
public final void clamp(float min, float max)
public final void clamp(float min, float max, Tuple3f t)
public final void clampMin(float min)
public final void clampMin(float min, Tuple3f t)
public final void clampMax(float max)
public final void clampMax(float max, Tuple3f t)
```

The first `clamp` method clamps this tuple to the range [`min`, `max`]. The second `clamp` method clamps the values from tuple `t` to the range [`min`, `max`] and assigns these clamped values to this tuple. The first `clampMin` method clamps each value of this tuple to the `min` parameter. The second `clampMin` method clamps each value of the tuple `t` and assigns these clamped values to this tuple. The first `clampMax` method clamps each value of this tuple to the `max` parameter. The second `clampMax` method clamps each value of tuple `t` to the `max` parameter and assigns these clamped values to this tuple. In each method the values of tuple `t` remain unchanged.

```
public final void interpolate(Tuple3f t1, Tuple3f t2, float alpha)
public final void interpolate(Tuple3f t1, float alpha)
```

The first method linearly interpolates between tuples `t1` and `t2` and places the result into this tuple (this = alpha * t1 + (1 – alpha) * t2). The second method linearly interpolates between this tuple and tuple `t1` and places the result into this tuple (this = alpha * this + (1–alpha) * t1).

```
int hashCode()
```

This method returns a hash number based on the data values in this object. Two different Tuple3f objects with identical data values (that is, `equals(Tuple3f)` returns `true`) will return the same hash number. Two tuples with different data members may return the same hash value, although this is not likely.

A.1.4.1 Point3f Class

The Point3f class extends Tuple3f. The Point3f is a three-element point represented by single-precision floating-point *x*, *y*, and *z* coordinates.

Constructors

```
public Point3f(float x, float y, float z)
public Point3f(float p[])
public Point3f(Point3d p1)
public Point3f(Point3f p1)
public Point3f(Tuple3d t1)
public Point3f(Tuple3f t1)
public Point3f()
```

These five constructors each return a new Point3f. The first constructor generates a point from three floating-point numbers `x`, `y`, and `z`. The second constructor (`Point3f(float p[])`) generates a point from the first three elements of array `p`. The third constructor generates a point from the double-precision point `p1`. The fourth constructor generates a point from the single-precision point `p1`. The fifth and sixth constructors generate a Point3f from the tuple `t1`. The final constructor generates a point with the value of (0.0, 0.0, 0.0).

Methods

```
public final float distance(Point3f p1)
public final float distanceSquared(Point3f p1)
```

The `distance` method computes the Euclidean distance between this point and the point `p1` and returns the result. The `distanceSquared` method computes the square of the Euclidean distance between this point and the point `p1` and returns the result.

```
public final float distanceL1(Point3f p1)
```

This method computes the L_1 (Manhattan) distance between this point and point `p1`. The L_1 distance is equal to

$$\mathrm{abs}(x1 - x2) + \mathrm{abs}(y1 - y2) + \mathrm{abs}(z1 - z2)$$

```
public final float distanceLinf(Point3f p1)
```

This method computes the L_∞ distance between this point and point `p1`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2), \mathrm{abs}(z1 - z2)]$$

```
public final void project(Point4f p1)
```

This method multiplies each of the `x`, `y`, and `z` components of the Point4f parameter `p1` by 1/`w` and places the projected values into this point.

A.1.4.2 Vector3f Class

The Vector3f class extends Tuple3f. The Vector3f is a three-element vector represented by single-precision floating-point *x*, *y*, and *z* coordinates.

Constructors

```
public Vector3f(float x, float y, float z)
public Vector3f(float v[])
public Vector3f(Vector3d v1)
public Vector3f(Vector3f v1)
public Vector3f(Tuple3d t1)
Public Vector3f(Tuple3f t1)
public Vector3f()
```

These five constructors each return a new Vector3f. The first constructor generates a Vector3f from three floating-point numbers `x`, `y`, and `z`. The second constructor generates a Vector3f from the first three elements of array `v`. The third constructor generates a Vector3f from the double-precision Vector3d `v1`. The fourth constructor generates a Vector3f from the single-precision Vector3f `v1`. The fifth and sixth constructors generate a Vector3f from the tuple `t1`. The final constructor generates a Vector3f with the value of (0.0, 0.0, 0.0).

Methods

```
public final float length()
public final float lengthSquared()
```

The `length` method computes the length of the vector `this` and returns its length as a single-precision floating-point number. The `lengthSquared` method computes the square of the length of the vector `this` and returns its length as a single-precision floating-point number.

```
public final void cross(Vector3f v1, Vector3f v2)
```

The `cross` method computes the vector cross-product of `v1` and `v2` and places the result in `this`.

```
public final float dot(Vector3f v1)
```

The `dot` method computes the dot product between this vector and the vector `v1` and returns the resulting value.

```
public final void normalize(Vector3f v1)
public final void normalize()
```

The first `normalize` method normalizes the vector `v1` to unit length and places the result in `this`. The second `normalize` method normalizes the vector `this` and places the resulting unit vector back into `this`.

```
public final float angle(Vector3f v1)
```

This method returns the angle, in radians, between this vector and the vector parameter. The return value is constrained to the range [0, π].

A.1.4.3 TexCoord3f Class

The TexCoord3f class extends Tuple3f. The TexCoord3f is a three-element texture coordinate represented by single-precision floating-point *x*, *y*, and *z* coordinates.

Constructors

```
public TexCoord3f(float x, float y, float z)
public TexCoord3f(float v[])
public TexCoord3f(TexCoord3f v1)
public TexCoord3f(Tuple3d t1)
public TexCoord3f(Tuple3f t1)
public TexCoord3f()
```

These four constructors each return a new TexCoord3f. The first constructor generates a texture coordinate from three floating-point numbers `x`, `y`, and `z`. The second constructor generates a texture coordinate from the first three elements of array `v`. The third constructor generates a texture coordinate from the single-precision TexCoord3f `v1`. The fourth and fifth constructors generate a texture coordinate from tuple `t1`. The final constructor generates a texture coordinate with the value of (0.0, 0.0, 0.0).

A.1.4.4 Color3f Class

The Color3f class extends Tuple3f. The Color3f is a three-element color value represented by single-precision floating-point *x*, *y*, and *z* values. The *x*, *y*, and *z* values represent the red, blue, and green color values, respectively. Color components should be in the range [0.0, 1.0].

Constructors

```
public Color3f(float x, float y, float z)
public Color3f(float v[])
public Color3f(Color3f v1)
public Color3f(Tuple3d t1)
public Color3f(Tuple3f t1)
public Color3f()
```

These four constructors each return a new Color3f. The first constructor generates a Color3f from three floating-point numbers `x`, `y`, and `z`. The second constructor (`Color3f(float v[]`) generates a Color3f from the first three elements of array `v`. The third constructor generates a Color3f from the single-precision color `v1`. The fourth and fifth constructors generate a Color3f from the tuple `t1`. The final constructor generates a Color3f with the value of (0.0, 0.0, 0.0).

A.1.5 Tuple4b Class

The Tuple4b class represents four-byte tuples.

Variables

The component values of a Tuple4b are directly accessible through the public variables `x`, `y`, `z`, and `w`. The `x`, `y`, `z`, and `w` values represent the red, green, blue, and alpha values, respectively. To access the `x` (red) component of a Tuple4b called `backgroundColor`, a programmer would write `backgroundColor.x`. The programmer would access the `y` (green), `z` (blue), and `w` (alpha) components similarly.

Note: Java defines a byte as a signed integer in the range [–128, 127]. However, colors are more typically represented by values in the range [0, 255]. Java 3D recognizes this and, in those cases where Color4b is used to represent color, treats the bytes as if the range were [0, 255].

```
public byte x
public byte y
public byte z
public byte w
```

The red, green, blue, and alpha values, respectively.

Constructors

```
public Tuple4b(byte b1, byte b2, byte b3, byte b4)
public Tuple4b(byte t[])
public Tuple4b(Tuple4b t1)
public Tuple4b()
```

These four constructors each return a new Tuple4b. The first constructor generates a Tuple4b from four bytes `b1`, `b2`, `b3`, and `b4`. The second constructor (`Tuple4b(byte t[])`) generates a Tuple4b from the first four elements of array `t`. The third constructor generates a Tuple4b from the byte-precision Tuple4b `t1`. The final constructor generates a Tuple4b with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public String toString()
```

This method returns a string that contains the values of this Tuple4b.

```
public final void set(byte b[])
public final void set(Tuple4b t1)
public final void get(byte b[])
public final void get(Tuple4b t1)
```

The first `set` method sets the value of the data members of this Tuple4b to the value of the array `b`. The second `set` method sets the value of the data members of this Tuple4b to the value of the argument tuple `t1`. The first `get` method places the values of the `x`, `y`, `z`, and `w` components of this Tuple4b into the byte array `b`. The second `get` method places the values of the `x`, `y`, `z`, and `w` components of this Tuple4b into the Tuple4b `t1`.

```
public boolean equals(Tuple4b t1)
```

This method returns `true` if all of the data members of Tuple4b `t1` are equal to the corresponding data members in this Tuple4b.

```
public int hashCode()
```

This method returns a hash number based on the data values in this object. Two different Tuple4b objects with identical data values (that is, `equals(Tuple4b)` returns `true`) will return the same hash number. Two Tuple4b objects with different data members may return the same hash value, although this is not likely.

A.1.5.1 Color4b Class

The Color4b class extends Tuple4b. The Color4b is a four-byte color value (red, green, blue, and alpha).

Constructors

```
public Color4b(byte b1, byte b2, byte b3, byte b4)
public Color4b(byte c[])
public Color4b(Color4b c1)
public Color4b(Tuple4b t1)
public Color4b()
```

These four constructors each return a new Color4b. The first constructor generates a Color4b from four bytes `b1`, `b2`, `b3`, and `b4`. The second constructor generates a Color4b from the first four elements of byte array `c`. The third constructor generates a Color4b from the byte-precision Color4b `c1`. The fourth constructor generates a Color4b from the tuple `t1`. The final constructor generates a Color4b with the value of (0.0, 0.0, 0.0, 0.0).

A.1.6 Tuple4d Class

The Tuple4d class represents a four-element tuple represented by double-precision floating-point *x*, *y*, *z*, and *w* coordinates.

Variables

The component values of a Tuple4d are directly accessible through the public variables `x`, `y`, `z`, and `w`. To access the `x` component of a Tuple4d called `upperLeftCorner`, a programmer would write `upperLeftCorner.x`. The programmer would access the `y`, `z`, and `w` components similarly.

```
public double x
public double y
public double z
public double w
```

The *x*, *y*, *z*, and *w* coordinates, respectively.

Constructors

```
public Tuple4d(double x, double y, double z, double w)
public Tuple4d(double t[])
public Tuple4d(Tuple4d t1)
public Tuple4d(Tuple4f t1)
public Tuple4d()
```

These five constructors each return a new Tuple4d. The first constructor generates a Tuple4d from four floating-point numbers `x`, `y`, `z`, and `w`. The second constructor (`Tuple4d(double t[])`) generates a Tuple4d from the first four elements of array `t`. The third constructor generates a Tuple4d from the double-precision tuple `t1`. The fourth constructor generates a Tuple4d from the single-precision tuple `t1`. The final constructor generates a Tuple4d with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public final void set(double x, double y, double z, double w)
public final void set(double t[])
public final void set(Tuple4d t1)
public final void set(Tuple4f t1)
public final void get(double t[])
public final void get(Tuple4d t)
```

These methods set the value of the tuple `this` to the values specified or to the values of the specified tuples. The first `get` method retrieves the value of this tuple and places it into the array `t` of length four, in *x*, *y*, *z*, *w* order. The second `get` method retrieves the value of this tuple and places it into tuple `t`.

```
public final void add(Tuple4d t1, Tuple4d t2)
public final void add(Tuple4d t1)
public final void sub(Tuple4d t1, Tuple4d t2)
public final void sub(Tuple4d t1)
```

The first `add` method computes the element-by-element sum of the tuple `t1` and the tuple `t2`, placing the result in `this`. The second `add` method computes the element-by-element sum of this tuple and the tuple `t1` and places the result in `this`. The first `sub` method performs an element-by-element subtraction of tuple `t2` from tuple `t1` and places the result in `this`. The second `sub` method performs an element-by-element subtraction of tuple `t1` from this tuple and places the result in `this`.

```
public final void negate(Tuple4d t1)
public final void negate()
```

The first `negate` method sets the values of this tuple to the negative of the values from tuple `t1`. The second `negate` method negates the tuple `this` and places the resulting tuple back into `this`.

```
public final void scale(double s, Tuple4d t1)
public final void scale(double s)
public final void scaleAdd(double s, Tuple4d t1)
public final void scaleAdd(double s, Tuple4d t1, Tuple4d t2)
```

The first `scale` method multiplies each element of the tuple `t1` by the scale factor `s` and places the resulting scaled tuple into `this`. The second `scale` method multiples the tuple `this` by the scale factor `s` and replaces `this` with the scaled value. The first `scaleAdd` method scales this tuple by the scale factor `s`, adds the result to tuple `t1`, and places the result into tuple `this` (this = s*this + t1). The second `scaleAdd` method scales the tuple `t1` by the scale factor `s`, adds the result to the tuple `t2`, and places the result into the tuple `this` (this = s*t1 + t2).

```
public void interpolate(Tuple4d t1, Tuple4d t2, float alpha)
public void interpolate(Tuple4d t1, float alpha)
```

The first `interpolate` method linearly interpolates between tuples `t1` and `t2` and places the result into this tuple (this = alpha * t1 + (1 – alpha) * t2). The second `interpolate` method linearly interpolates between this tuple and tuple `t1` and places the result into this tuple (this = alpha * this + (1 – alpha) * t1).

```
public String toString()
```

This method returns a string that contains the values of this tuple. The form is `(x, y, z, w)`.

```
public boolean equals(Tuple4d v1)
```

This method returns `true` if all of the data members of tuple `v1` are equal to the corresponding data members in this tuple.

```
public boolean epsilonEquals(Tuple4d t1, double epsilon)
```

This method returns `true` if the L_∞ distance between this Tuple4d and Tuple4d `t1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2), \mathrm{abs}(z1 - z2), \mathrm{abs}(w1 - w2)]$$

```
public final void absolute()
public final void absolute(Tuple4d t)
```

The first `absolute` method sets each component of this tuple to its absolute value. The second `absolute` method sets each component of this tuple to the absolute value of the corresponding component in tuple `t`.

```
public final void clamp(float min, float max)
public final void clamp(float min, float max, Tuple4d t)
public final void clampMin(float min)
public final void clampMin(float min, Tuple4d t)
public final void clampMax(float max)
public final void clampMax(float max, Tuple4d t)
```

The first `clamp` method clamps this tuple to the range [`min`, `max`]. The second `clamp` method clamps this tuple to the range [`min`, `max`] and places the values into tuple `t`. The first `clampMin` method clamps the minimum value of this tuple to the `min` parameter. The second `clampMin` method clamps the minimum value of this tuple to the `min` parameter and places the values into the tuple `t`. The first `clampMax` method clamps the maximum value of this tuple to the `max` parameter. The second `clampMax` method clamps the maximum value of this tuple to the `max` parameter and places the values into the tuple `t`.

```
public int hashCode()
```

This method returns a hash number based on the data values in this object. Two different Tuple4d objects with identical data values (that is, `equals(Tuple4d)` returns `true`) will return the same hash number. Two Tuple4d objects with different data members may return the same hash value, although this is not likely.

A.1.6.1 Point4d Class

The Point4d class extends Tuple4d. The Point4d is a four-element point represented by double-precision floating-point x, y, z, and w coordinates.

Constructors

```
public Point4d(double x, double y, double z, double w)
public Point4d(double p[])
public Point4d(Point4d p1)
public Point4d(Point4f p1)
public Point4d(Tuple4d t1)
public Point4d(Tuple4f t1)
public Point4d()
```

These five constructors each return a new Point4d. The first constructor generates a Point4d from four floating-point numbers `x`, `y`, `z`, and `w`. The second constructor (`Point4d(double p[])`) generates a Point4d from the first four elements of array `p`. The third constructor generates a Point4d from the double-precision point `p1`. The fourth constructor generates a Point4d from the single-precision point `p1`. The fifth and sixth constructors generate a Point4d from tuple `t1`. The final constructor generates a Point4d with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public final double distance(Point4d p1)
public final double distanceSquared(Point4d p1)
```

The `distance` method computes the Euclidean distance between this point and the point `p1` and returns the result. The `distanceSquared` method computes the square of the Euclidean distance between this point and the point `p1` and returns the result.

public final float distanceL1(Point4d p1)

This method computes the L_1 (Manhattan) distance between this point and point `p1`. The L_1 distance is equal to

$$\mathrm{abs}(x1 - x2) + \mathrm{abs}(y1 - y2) + \mathrm{abs}(z1 - z2) + \mathrm{abs}(w1 - w2)$$

public final float distanceLinf(Point4d p1)

This method computes the L_∞ distance between this point and point `p1`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2), \mathrm{abs}(z1 - z2), \mathrm{abs}(w1 - w2)]$$

public final void project(Point4d p1)

This method multiplies each of the `x`, `y`, and `z` components of the point `p1` by $1/w$, places the projected values into this point, and places a 1 into the `w` parameter of this point.

A.1.6.2 Vector4d Class

The Vector4d class extends Tuple4d. The Vector4d is a four-element vector represented by double-precision floating-point *x*, *y*, *z*, and *w* coordinates.

Constructors

```
public Vector4d(double x, double y, double z, double w)
public Vector4d(double v[])
public Vector4d(Vector4d v1)
public Vector4d(Vector4f v1)
public Vector4d(Tuple4d t1)
public Vector4d(Tuple4f t1)
public Vector4d()
```

These five constructors each return a new Vector4d. The first constructor generates a Vector4d from four floating-point numbers `x`, `y`, `z`, and `w`. The second constructor generates a Vector4d from the first four elements of array `v`. The third constructor generates a Vector4d from the double-precision Vector4d `v1`. The fourth constructor generates a Vector4d from the single-precision Vector4f `v1`. The fifth and sixth constructors generate a Vector4d from tuple `t1`. The final constructor generates a Vector4d with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public final double length()
public final double lengthSquared()
```

The `length` method computes the length of the vector `this` and returns its length as a double-precision floating-point number. The `lengthSquared` method computes the square of the length of the vector `this` and returns its length as a double-precision floating-point number.

```
public final void dot(Vector4d v1)
```

This method returns the dot product of this vector and vector `v1`.

```
public final void normalize(Vector4d v1)
public final void normalize()
```

The first `normalize` method normalizes the vector `v1` to unit length and places the result in `this`. The second `normalize` method normalizes the vector `this` and places the resulting unit vector back into `this`.

```
public final double angle(Vector4d v1)
```

This method returns the (four-space) angle, in radians, between this vector and the vector `v1` parameter. The return value is constrained to the range [0, π].

A.1.6.3 Quat4d Class

The Quat4d class extends Tuple4d. The Quat4d is a four-element quaternion represented by double-precision floating-point *x*, *y*, *z*, and *w* values.

Constructors

```
public Quat4d(double x, double y, double z, double w)
public Quat4d(double q[])
public Quat4d(Quat4d q1)
public Quat4d(Quat4f q1)
public Quat4d(Tuple4d t1)
public Quat4d(Tuple4f t1)
public Quat4d()
```

These five constructors each return a new Quat4d. The first constructor generates a quaternion from four floating-point numbers `x`, `y`, `z`, and `w`. The second constructor generates a quaternion from the first four elements of array `q` of length four. The third constructor generates a quaternion from the double-precision quaternion `q1`. The fourth constructor generates a quaternion from the single-precision quaternion q1. The fifth and sixth constructors generate a Quat4d from tuple t1. The final constructor generates a quaternion with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public final void conjugate(Quat4d q1)
public final void conjugate()
```

The first `conjugate` method sets the values of this quaternion to the conjugate of quaternion `q1`. The second `conjugate` method negates the value of each of this quaternion's *x*, *y*, and *z* coordinates in place.

```
public final void mul(Quat4d q1, Quat4d q2)
public final void mul(Quat4d q1)
```

The first `mul` method sets the value of this quaternion to the quaternion product of quaternions `q1` and `q2` (this = q1 * q2). Note that this is safe for aliasing (that is, `this` can be `q1` or `q2`). The second `mul` method sets the value of this quaternion to the quaternion products of itself and `q1` (this = this * q1).

```
public final void mulInverse(Quat4d q1, Quat4d q2)
public final void mulInverse(Quat4d q1)
```

The first `mulInverse` method multiplies quaternion `q1` by the inverse of quaternion `q2` and places the value into this quaternion. The values of both quaternion arguments are preserved (this = q1 * $q2^{-1}$). The second `mulInverse` method multiplies this quaternion by the inverse of quaternion `q1` and places the value into this quaternion. The value of the argument `q1` is preserved (this = this * $q1^{-1}$).

```
public final void inverse(Quat4d q1)
public final void inverse()
```

The first `inverse` method sets the value of this quaternion to the quaternion inverse of quaternion `q1`. The second `inverse` method sets the value of this quaternion to the quaternion inverse of itself.

```
public final void normalize(Quat4d q1)
public final void normalize()
```

The first `normalize` method sets the value of this quaternion to the normalized value of quaternion `q1`. The second `normalize` method normalizes the value of this quaternion in place.

```
public final void set(Matrix4f m1)
public final void set(Matrix4d m1)
public final void set(Matrix3f m1)
public final void set(Matrix3d m1)
public final void set(AxisAngle4f a)
public final void set(AxisAngle4d a)
```

These `set` methods set the value of this quaternion to the rotational component of the passed matrix.

```
public final void interpolate(Quat4d q1, double alpha)
public final void interpolate(Quat4d q1, Quat4d q2, double alpha)
```

The first method performs a great circle interpolation between this quaternion and the quaternion parameter and places the result into this quaternion. The second method performs a great circle interpolation between quaternion `q1` and quaternion `q2` and places the result into this quaternion.

A.1.7 Tuple4f Class

The Tuple4f class represents a four-element tuple represented by single-precision floating-point *x*, *y*, *z*, and *w* values.

Variables

The component values of a Tuple4f are directly accessible through the public variables `x`, `y`, `z`, and `w`. To access the `x` component of a Tuple4f called `upperLeftCorner`, a programmer would write `upperLeftCorner.x`. The programmer would access the `y`, `z`, and `w` components similarly.

```
public double x
public double y
public double z
public double w
```

The *x*, *y*, *z*, and *w* values, respectively.

Constructors

```
public Tuple4f(float x, float y, float z, float w)
public Tuple4f(float t[])
public Tuple4f(Tuple4d t1)
```

```
public Tuple4f(Tuple4f t1)
public Tuple4f()
```

These five constructors each return a new Tuple4f. The first constructor generates a Tuple4f from four floating-point numbers `x`, `y`, `z`, and `w`. The second constructor (`Tuple4f(float t[])`) generates a Tuple4f from the first four elements of array `t`. The third constructor generates a Tuple4f from the double-precision tuple `t1`. The fourth constructor generates a Tuple4f from the single-precision tuple `t1`. The final constructor generates a Tuple4f with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public final void set(float x, float y, float z, float w)
public final void set(float t[])
public final void set(Tuple4f t1)
public final void set(Tuple4d t1)
public final void get(float t[])
public final void get(Tuple4f t)
```

The first `set` method sets the value of this tuple to the specified `x`, `y`, `z`, and `w` values. The second `set` method sets the value of this tuple to the specified coordinates in the array. The next two methods set the value of tuple `this` to the value of tuple `t1`. The `get` methods copy the value of this tuple into the tuple `t`.

```
public final void add(Tuple4f t1, Tuple4f t2)
public final void add(Tuple4f t1)
public final void sub(Tuple4f t1, Tuple4f t2)
public final void sub(Tuple4f t1)
```

The first `add` method computes the element-by-element sum of tuples `t1` and `t2` and places the result in `this`. The second `add` method computes the element-by-element sum of this tuple and tuple `t1` and places the result in `this`. The first `sub` method performs the element-by-element subtraction of tuple `t2` from tuple `t1` and places the result in `this` (this = t1 – t2). The second `sub` method performs the element-by-element subtraction of tuple `t1` from this tuple and places the result in `this` (this = this – t1).

```
public final void negate(Tuple4f t1)
public final void negate()
```

The first `negate` method sets the values of this tuple to the negative of the values from tuple `t1`. The second `negate` method negates the tuple `this` and places the resulting tuple back into `this`.

public final void scale(float s, Tuple4f t1)
public final void scale(float s)
public final void scaleAdd(float s, Tuple4f t1)
public final void scaleAdd(float s, Tuple4f t1, Tuple4f t2)

The first `scale` method multiplies each element of the tuple `t1` by the scale factor `s` and places the resulting scaled tuple into `this`. The second `scale` method multiples the tuple `this` by the scale factor `s`, replacing `this` with the scaled value. The first `scaleAdd` method scales this tuple by the scale factor `s`, adds the result to tuple `t1`, and places the result into tuple `this` (this = s*this + t1). The second `scaleAdd` method scales the tuple `t1` by the scale factor `s`, adds the result to the tuple `t2`, and places the result into the tuple `this` (this = s*t1 + t2).

public String toString()

This method returns a string that contains the values of this Tuple4f. The form is `(x, y, z, w)`.

public boolean equals(Tuple4f t1)

This method returns `true` if all of the data members of Tuple4f `t1` are equal to the corresponding data members in this Tuple4f.

public boolean epsilonEquals(Tuple4f t1, float epsilon)

This method returns `true` if the L_∞ distance between this Tuple4f and Tuple4f `t1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2), \mathrm{abs}(z1 - z2), \mathrm{abs}(w1 - w2)]$$

public final void absolute()
public final void absolute(Tuple4f t)

The first `absolute` method sets each component of this tuple to its absolute value. The second `absolute` method sets each component of this tuple to the absolute value of the corresponding component in tuple `t`.

```
public final void clamp(float min, float max)
public final void clamp(float min, float max, Tuple4f t)
public final void clampMin(float min)
public final void clampMin(float min, Tuple4f t)
public final void clampMax(float max)
public final void clampMax(float max, Tuple4f t)
```

The first `clamp` method clamps this tuple to the range [`min`, `max`]. The second `clamp` method clamps this tuple to the range [`min`, `max`] and places the values into tuple `t`. The first `clampMin` method clamps the minimum value of this tuple to the `min` parameter. The second `clampMin` method clamps the minimum value of this tuple to the `min` parameter and places the values into the tuple `t`. The first `clampMax` method clamps the maximum value of this tuple to the `max` parameter. The second `clampMax` method clamps the maximum value of this tuple to the `max` parameter and places the values into the tuple `t`.

```
public void interpolate(Tuple4f t1, Tuple4f t2, float alpha)
public void interpolate(Tuple4f t1, float alpha)
```

The first `interpolate` method linearly interpolates between tuples `t1` and `t2` and places the result into this tuple (this = alpha * t1 + (1 – alpha) * t2). The second `interpolate` method linearly interpolates between this tuple and tuple `t1` and places the result into this tuple (this = alpha * this + (1 – alpha) * t1).

```
public int hashCode()
```

This method returns a hash number based on the data values in this object. Two different Tuple4f objects with identical data values (that is, `equals(Tuple4f)` returns `true`) will return the same hash number. Two Tuple4f objects with different data members may return the same hash value, although this is not likely.

A.1.7.1 Point4f Class

The Point4f class extends Tuple4f. The Point4f is a four-element point represented by single-precision floating-point x, y, z, and w coordinates.

Constructors

```
public Point4f(float x, float y, float z, float w)
public Point4f(float p[])
public Point4f(Point4d p1)
public Point4f(Point4f p1)
public Point4f(Tuple4d t1)
public Point4f(Tuple4f t1)
public Point4f()
```

These five constructors each return a new Point4f. The first constructor generates a Point4f from four floating-point numbers `x`, `y`, `z`, and `w`. The second constructor (`Point4f(float p[])`) generates a Point4f from the first four elements of array `p`. The third constructor generates a Point4f from the double-precision point `p1`. The fourth constructor generates a Point4f from the single-precision point `p1`. The fifth and sixth constructors generate a Point4f from tuple `t1`. The final constructor generates a Point4f with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public final float distanceSquared(Point4f p1)
public final float distance(Point4f p1)
```

The `distanceSquared` method computes the square of the Euclidean distance between this point and the point `p1` and returns the result. The `distance` method computes the Euclidean distance between this point and the point `p1` and returns the result.

```
public final float distanceL1(Point4f p1)
```

This method computes the L_1 (Manhattan) distance between this point and point `p1`. The L_1 distance is equal to

$$\mathrm{abs}(x1 - x2) + \mathrm{abs}(y1 - y2) + \mathrm{abs}(z1 - z2) + \mathrm{abs}(w1 - w2)$$

```
public final float distanceLinf(Point4f p1)
```

This method computes the L_∞ distance between this point and point `p1`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2), \mathrm{abs}(z1 - z2), \mathrm{abs}(w1 - w2)]$$

```
public final void project(Point4f p1)
```

This method multiplies each of the `x`, `y`, and `z` components of the point `p1` by $1/w$, places the projected values into this point, and places a 1 into the `w` parameter of this point.

A.1.7.2 Color4f Class

The Color4f class extends Tuple4f. The Color4f is a four-element color value represented by single-precision floating-point *x*, *y*, *z*, and *w* values. The *x*, *y*, *z*, and *w* values represent the red, blue, green, and alpha color values, respectively. Color and alpha components should be in the range [0.0, 1.0].

Constructors

```
public Color4f(float x, float y, float z, float w)
public Color4f(float c[])
public Color4f(Color4f c1)
public Color4f(Tuple4d t1)
public Color4f(Tuple4f t1)
public Color4f()
```

These four constructors each return a new Color4f. The first constructor generates a Color4f from four floating-point numbers `x`, `y`, `z`, and `w`. The second constructor generates a Color4f from the first four elements of array `c`. The third constructor generates a Color4f from the single-precision color `c1`. The fourth and fifth constructors generate a Color4f from tuple `t1`. The final constructor generates a Color4f with the value of (0.0, 0.0, 0.0, 0.0).

A.1.7.3 Vector4f Class

The Vector4f class extends Tuple4f. The Vector4f is a four-element vector represented by single-precision floating-point *x*, *y*, *z*, and *w* coordinates.

Constructors

```
public Vector4f(float x, float y, float z, float w)
public Vector4f(float v[])
public Vector4f(Vector4d v1)
public Vector4f(Vector4f v1)
public Vector4f(Tuple4d t1)
public Vector4f(Tuple4f t1)
public Vector4f()
```

These five constructors each return a new Vector4f. The first constructor generates a Vector4f from four floating-point numbers `x`, `y`, `z`, and `w`. The second constructor generates a Vector4f from the first four elements of array `v`. The third constructor generates a Vector4f from the double-precision Vector4d `v1`. The fourth constructor generates a Vector4f from the single-precision Vector4f `v1`. The fifth and sixth constructors generate a Vector4f from tuple `t1`. The final constructor generates a Vector4f with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public final float length()
public final float lengthSquared()
```

The `length` method computes the length of the vector `this` and returns its length as a single-precision floating-point number. The `lengthSquared` method computes the square of the length of the vector `this` and returns its length as a single-precision floating-point number.

```
public final float dot(Vector4f v1)
```

The `dot` method computes the dot product between this vector and the vector `v1` and returns the resulting value.

```
public final void normalize(Vector4f v1)
public final void normalize()
```

The first `normalize` method sets the value of this vector to the normalization of vector `v1`. The second `normalize` method normalizes this vector in place.

```
public final float angle(Vector4f v1)
```

This method returns the (four-space) angle, in radians, between this vector and the vector `v1` parameter. The return value is constrained to the range [0, π].

A.1.7.4 Quat4f Class

The Quat4f class extends Tuple4f. The Quat4f is a four-element quaternion represented by single-precision floating-point *x*, *y*, *z*, and *w* coordinates.

Constructors

```
public Quat4f(float x, float y, float z, float w)
public Quat4f(float q[])
public Quat4f(Quat4d q1)
public Quat4f(Quat4f q1)
public Quat4f(Tuple4d t1)
public Quat4f(Tuple4f t1)
public Quat4f()
```

These five constructors each return a new Quat4f. The first constructor generates a quaternion from four floating-point numbers `x`, `y`, `z`, and `w`. The second constructor generates a quaternion from the four floating-point numbers of array `q` of length four. The third constructor generates a quaternion from the double-precision quaternion `q1`. The fourth constructor generates a quaternion from the single-precision quaternion `q1`. The fifth and sixth constructors generate a quaternion from tuple `t1`. The final constructor generates a quaternion with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public final void conjugate(Quat4f q1)
public final void conjugate()
```

The first `conjugate` method sets the value of this quaternion to the conjugate of quaternion `q1`. The second `conjugate` method sets the value of this quaternion to the conjugate of itself.

```
public final void mul(Quat4f q1, Quat4f q2)
public final void mul(Quat4f q1)
```

The first `mul` method sets the value of this quaternion to the quaternion product of quaternions `q1` and `q2` (this = q1 * q2). Note that this is safe for aliasing (that is, `this` can be `q1` or `q2`). The second `mul` method sets the value of this quaternion to the quaternion product of itself and `q1` (this = this * q1).

public final void mulInverse(Quat4f q1, Quat4f q2)
public final void mulInverse(Quat4f q1)

The first `mulInverse` method multiplies quaternion `q1` by the inverse of quaternion `q2` and places the value into this quaternion. The value of both argument quaternions is preserved (this = q1 * $q2^{-1}$). The second `mulInverse` method multiplies this quaternion by the inverse of quaternion `q1` and places the value into this quaternion. The value of the argument quaternion is preserved (this = this * $q1^{-1}$).

public final void inverse(Quat4f q1)
public final void inverse()

The first `inverse` method sets the value of this quaternion to the quaternion inverse of quaternion `q1`. The second `inverse` method sets the value of this quaternion to the quaternion inverse of itself.

public final void normalize(Quat4f q1)
public final void normalize()

The first `normalize` method sets the value of this quaternion to the normalized value of quaternion `q1`. The second `normalize` method normalizes the value of this quaternion in place.

public final void set(Matrix4f m1)
public final void set(Matrix4d m1)
public final void set(Matrix3f m1)
public final void set(Matrix3d m1)
public final void set(AxisAngle4f a)
public final void set(AxisAngle4d a)

These `set` methods set the value of this quaternion to the rotational component of the passed matrix.

public final void interpolate(Quat4f q1, float alpha)
public final void interpolate(Quat4f q1, Quat4f q2, float alpha)

The first method performs a great circle interpolation between this quaternion and quaternion `q1` and places the result into this quaternion. The second method performs a great circle interpolation between quaternion `q1` and quaternion `q2` and places the result into this quaternion.

A.1.8 AxisAngle4d Class

The AxisAngle4d class represents a four-element axis-angle represented by double-precision floating-point *x*, *y*, *z* coordinates and an angle of rotation in radians. An axis-angle is a rotation of `angle` radians about the vector `x`,`y`,`z`.

Variables

The component values of an AxisAngle4d are directly accessible through the public variables `x`, `y`, `z`, and `angle`. To access the `x` component of an AxisAngle4d called `myRotation`, a programmer would write `myRotation.x`. The programmer would access the `y`, `z`, and `angle` components similarly.

```
public double x
public double y
public double z
public double angle
```

The *x*, *y*, and *z* coordinates and the rotational angle, respectively. The rotation angle is expressed in radians.

Constructors

```
public AxisAngle4d(double x, double y, double z, double angle)
public AxisAngle4d(double a[])
public AxisAngle4d(AxisAngle4d a1)
public AxisAngle4d(AxisAngle4f a1)
public AxisAngle4d()
```

These five constructors each return a new AxisAngle4d. The first constructor generates an axis-angle from four floating-point numbers `x`, `y`, `z`, and `angle`. The second constructor generates an axis-angle from the first four elements of array `a`. The third constructor generates an axis-angle from the double-precision axis-angle `a1`. The fourth constructor generates an axis-angle from the single-precision axis-angle `a1`. The final constructor generates an axis-angle with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public final void set(double x, double y, double z, double angle)
public final void set(double a[])
public final void set(Matrix4f m1)
public final void set(Matrix4d m1)
public final void set(Matrix3f m1)
public final void set(Matrix3d m1)
```

```
public final void set(AxisAngle4f a1)
public final void set(AxisAngle4d a1)
public final void set(Quat4f q1)
public final void set(Quat4d q1)
public final void get(double a[])
```

The first `set` method sets the value of this axis-angle to the specified `x`, `y`, `z`, and `angle` coordinates. The second `set` method sets the value of this axis-angle to the specified *x*,*y*,*z* angle. The next four `set` methods set the value of this axis-angle to the rotational component of the passed matrix `m1`. The next two `set` methods set the value of this axis-angle to the value of axis-angle `a1`. The last two `set` methods set the value of this axis-angle to the value of the passed quaternion `q1`. The `get` method retrieves the value of this axis-angle and places it into the array `a` of length four in `x`,`y`,`z`,`angle` order.

```
public String toString()
```

This method returns a string that contains the values of this AxisAngle4d. The form is `(x, y, z, angle)`.

```
public boolean equals(AxisAngle4d v1)
```

This method returns `true` if all of the data members of AxisAngle4d `v1` are equal to the corresponding data members in this axis-angle.

```
public boolean epsilonEquals(AxisAngle4d a1, double epsilon)
```

This method returns `true` if the L_∞ distance between this axis-angle and axis-angle `a1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2), \mathrm{abs}(z1 - z2), \mathrm{abs}(angle1 - angle2)]$$

```
public int hashCode()
```

This method returns a hash number based on the data values in this object. Two different AxisAngle4d objects with identical data values (that is, `equals(AxisAngle4d)` returns `true`) will return the same hash number. Two AxisAngle4d objects with different data members may return the same hash value, although this is not likely.

A.1.9 AxisAngle4f Class

The AxisAngle4f class represents a four-element axis-angle represented by single-precision floating-point x, y, and z coordinates and an angle of rotation in radians. An axis-angle is a rotation of `angle` radians about the vector `x`,`y`,`z`.

Variables

The component values of an AxisAngle4f are directly accessible through the public variables `x`, `y`**,** `z`**,** and `angle`. To access the `x` component of an AxisAngle4f called `myRotation`, a programmer would write `myRotation.x`. The programmer would access the `y`, `z`, and `angle` components similarly.

```
public float x
public float y
public float z
public float angle
```

The *x*, *y*, and *z* coordinates and the rotational angle, respectively. The rotation angle is expressed in radians.

Constructors

```
public AxisAngle4f(float x, float y, float z, float angle)
public AxisAngle4f(float a[])
public AxisAngle4f(AxisAngle4f a1)
public AxisAngle4f(AxisAngle4d a1)
public AxisAngle4f()
```

These five constructors each return a new AxisAngle4f. The first constructor generates an axis-angle from four floating-point numbers `x`, `y`, `z`, and `angle`. The second constructor generates an axis-angle from the first four elements of array `a`. The third constructor generates an axis-angle from the single-precision axis-angle `a1`. The fourth constructor generates an axis-angle from the double-precision axis-angle `a1`. The final constructor generates an axis-angle with the value of (0.0, 0.0, 0.0, 0.0).

Methods

```
public final void set(float x, float y, float z, float angle)
public final void set(float a[])
public final void set(Matrix4f m1)
public final void set(Matrix4d m1)
public final void set(Matrix3f m1)
public final void set(Matrix3d m1)
public final void set(AxisAngle4f a1)
public final void set(AxisAngle4d a1)
public final void set(Quat4f q1)
public final void set(Quat4d q1)
public final void get(float a[])
```

The first `set` method sets the value of this axis-angle to the specified `x`, `y`, `z`, and `angle` coordinates. The second `set` method sets the value of this axis-angle to the specified coordinates in the array `a`. The next four `set` methods set the value of this axis-angle to the rotational component of the passed matrix `m1`. The next two `set` methods set the value of this axis-angle to the value of axis-angle `a1`. The last two `set` methods set the value of this axis-angle to the value of the passed quaternion `q1`. The `get` method retrieves the value of this axis-angle and places it into the array `a` of length four in `x`,`y`,`z`,`angle` order.

```
public String toString()
```

This method returns a string that contains the values of this axis-angle. The form is `(x, y, z, angle)`.

```
public boolean equals(AxisAngle4f a1)
```

This method returns `true` if all of the data members of axis-angle `a1` are equal to the corresponding data members in this axis-angle.

```
public boolean epsilonEquals(AxisAngle4f a1, float epsilon)
```

This method returns `true` if the L_∞ distance between this axis-angle and axis-angle `a1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2), \mathrm{abs}(z1 - z2), \mathrm{abs}(angle1 - angle2)]$$

```
public int hashCode()
```

This method returns a hash number based on the data values in this object. Two different AxisAngle4f objects with identical data values (that is, `equals(AxisAngle4f)` returns `true`) will return the same hash number. Two

AxisAngle4f objects with different data members may return the same hash value, although this is not likely.

A.1.10 GVector Class

The GVector class represents a double-precision, general, dynamically resizable, one-dimensional vector class. Index numbering begins with zero.

Constructors

```
public GVector(int length)
public GVector(double vector[])
public GVector(GVector vector)
public GVector(Tuple2f tuple)
public GVector(Tuple3f tuple)
public GVector(Tuple3d tuple)
public GVector(Tuple4f tuple)
public GVector(Tuple4d tuple)
public GVector(double vector[], int length)
```

These eight constructors each return a new GVector. The first constructor generates a generalized mathematical vector with all elements set to 0.0: `length` represents the number of elements in the vector. The second and third constructors generate a generalized mathematical vector and copy the initial value from the parameter `vector`. The next four constructors generate a generalized mathematical vector and copy the initial value from the tuple parameter `tuple`. The final method generates a generalized mathematical vector by copying `length` elements from the array parameter. The parameter `length` must be less than or equal to `vector.length`.

Methods

```
public final void add(GVector v1)
public final void add(GVector v1, GVector v2)
public final void sub(GVector v1)
public final void sub(GVector v1, GVector v2)
```

The first `add` method computes the element-by-element sum of this GVector and GVector `v1` and places the result in `this`. The second `add` method computes the element-by-element sum of GVectors `v1` and `v2` and places the result in `this`. The first `sub` method performs the element-by-element subtraction of GVector `v1` from this GVector and places the result in `this` (this = this – v1). The second `sub` method performs the element-by-element subtraction of GVector `v2` from GVector `v1` and places the result in `this` (this = v1 – v2).

```
public final void mul(GMatrix m1, GVector v1)
public final void mul(GVector v1, GMatrix m1)
```

The first `mul` method multiplies matrix `m1` times vector `v1` and places the result into this vector (this = m1 * v1). The second `mul` method multiplies the transpose of vector `v1` (that is, `v1` becomes a row vector with respect to the multiplication) times matrix `m1` and places the result into this vector (this = transpose(v1) * m1). The result is technically a row vector, but the GVector class only knows about column vectors, so the result is stored as a column vector.

```
public final void negate()
```

This method negates the vector `this` and places the resulting vector back into `this`.

```
public final void zero()
```

This method sets all the values in this vector to zero.

```
public final void setSize(int length)
public final void int getSize()
```

This method changes the size of this vector dynamically. If the size is increased, no data values are lost. If the size is decreased, only those data values whose vector positions were eliminated are lost.

```
public final void set(double v[])
public final void set(GVector v)
public final void set(Tuple2f t)
public final void set(Tuple3f t)
public final void set(Tuple3d t)
public final void set(Tuple4f t)
public final void set(Tuple4d t)
```

The first `set` method sets the values of this vector to the values found in the array `v`: The array should be at least equal in length to the number of elements in the vector. The second `set` method sets the values of this vector to the values in vector `v`. The last five `set` methods set the value of this vector to the values in tuple `t`.

```
public final double getElement(int index)
public final void setElement(int index, double value)
```

These methods set and retrieve the specified index value of this vector.

```
public final double norm()
public final double normSquared()
```

The `norm` method returns the square root of the sum of the squares of this vector (its length in *n*-dimensional space). The `normSquared` method returns the sum of the squares of this vector (its length in *n*-dimensional space).

```
public final void normalize(GVector v1)
public final void normalize()
```

The first `normalize` method sets the value of this vector to the normalization of vector `v1`. The second `normalize` method normalizes this vector in place.

```
public final void scale(double s, GVector v1)
public final void scale(double s)
public final void scaleAdd(double s, GVector v1, GVector v2)
```

The first `scale` method sets the value of this vector to the scalar multiplication of the scale factor `s` with the vector `v1`. The second `scale` method scales this vector by the scale factor `s`. The `scaleAdd` method scales the vector `v1` by the scale factor `s`, adds the result to the vector `v2`, and places the result into this vector (this = s*v1 + v2).

```
public String toString()
```

This method returns a string that contains the values of this vector.

```
public int hashCode()
```

This method returns a hash number based on the data values in this object. Two different GVector objects with identical data values (that is, `equals(GVector)` returns `true`) will return the same hash number. Two objects with different data members may return the same hash value, although this is not likely.

```
public boolean equals(GVector vector1)
```

This method returns `true` if all of the data members of GVector `vector1` are equal to the corresponding data members in this GVector.

```
public boolean epsilonEquals(GVector v1, double epsilon)
```

This method returns `true` if the L_∞ distance between this vector and vector `v1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\mathrm{MAX}[\mathrm{abs}(x1 - x2), \mathrm{abs}(y1 - y2), \ldots]$$

```
public final double dot(GVector v1)
```

This method returns the dot product of this vector and vector `v1`.

```
public final void SVDBackSolve(GMatrix U, GMatrix W, GMatrix V,
       GVector x)
public final void LUDBackSolve(GMatrix LU, GVector b,
       GVector permutation)
```

The first method solves for x in $\mathbf{A}x = b$, where x is `this` vector ($n \times 1$), b is an $m \times 1$ vector, and $\mathbf{A}$ is an $m \times n$ matrix, defined as $\mathbf{A} = \mathbf{U} * \mathbf{W} * \text{transpose}(\mathbf{V})$. `U`, `W`, and `V` must be precomputed and can be found by taking the singular value decomposition (SVD) of **A**. The second method takes the `LU` matrix and the permutation vector produced by the GMatrix method `LUD` and solves the equation $\mathbf{LU} * x = b$ by placing the solution to the set of linear equations into `this` vector (x).

```
public final double angle(GVector v1)
```

This method returns the (n-space) angle, in radians, between this vector and the vector `v1` parameter . The return value is constrained to the range $[0, \pi]$.

```
public final void interpolate(GVector v1, GVector v2, float alpha)
public final void interpolate(GVector v1, float alpha)
```

Deprecated methods. See the next two methods.

```
public final void interpolate(GVector v1, GVector v2, double alpha)
public final void interpolate(GVector v1, double alpha)
```

The first method linearly interpolates between vectors `v1` and `v2` and places the result into this vector (this = alpha * v1 + (1 – alpha) * v2). The second method linearly interpolates between this vector and vector `v1` and places the result into this vector (this = alpha * this + (1 – alpha) * v1).

A.2 Matrix Objects

Java 3D uses matrix objects to represent rotations and full 3D transformations. The matrix classes (as well as the associated Tuple and AxisAngle classes) include code for accessing, manipulating, and updating the matrix, vector, and AxisAngle classes. Java 3D further subdivides the matrix classes into 3×3 matrices (mainly to store rotations) and 4×4 matrices (mainly to store more complex 3D transformations). These two classes in turn provide support for both single-precision floating-point representations and for double-precision floating-point representations.

Matrix operations try to minimize gratuitous allocation of memory, thus all matrix operations update an existing object. To multiply two matrices together and store the result in a third, a Java 3D application or applet would write `matrix3.mul(matrix1, matrix2)`. Here `matrix3` receives the results of multiplying `matrix1` with `matrix2`.

The Java 3D model for 3 × 3 transformations is

$$\begin{bmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

$$\begin{aligned} x' &= m00 \cdot x + m01 \cdot y + m02 \cdot z \\ y' &= m10 \cdot x + m11 \cdot y + m12 \cdot z \\ z' &= m20 \cdot x + m21 \cdot y + m22 \cdot z \end{aligned}$$

The Java 3D model for 4 × 4 transformations is

$$\begin{bmatrix} m00 & m01 & m02 & m03 \\ m10 & m11 & m12 & m13 \\ m20 & m21 & m22 & m23 \\ m30 & m31 & m32 & m33 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

$$\begin{aligned} x' &= m00 \cdot x + m01 \cdot y + m02 \cdot z + m03 \cdot w \\ y' &= m10 \cdot x + m11 \cdot y + m12 \cdot z + m13 \cdot w \\ z' &= m20 \cdot x + m21 \cdot y + m22 \cdot z + m23 \cdot w \\ w' &= m30 \cdot x + m31 \cdot y + m32 \cdot z + m33 \cdot w \end{aligned}$$

Note: When transforming a Point3f or a Point3d, the input w is set to 1. When transforming a Vector3f or Vector3d, the input w is set to 0.

A.2.1 Matrix3f Class

The Matrix3f class serves to contain 3 × 3 matrices mainly for storing and manipulating 3D rotation matrices. The class includes five different constructors for creating matrices and several operators for manipulating these matrices.

Variables

The component values of a Matrix3f are directly accessible through the public variables `m00`, `m01`, `m02`, `m10`, `m11`, `m12`, `m20`, `m21`, and `m22`. To access the element in row 2 and column 0 of matrix `rotate`, a programmer would write `rotate.m20`. A programmer would access the other values similarly.

```
public float m00
public float m01
public float m02
public float m10
public float m11
public float m12
public float m20
public float m21
public float m22
```

These public variables are the elements of the matrix.

Constructors

```
public Matrix3f(float m00, float m01, float m02, float m10,
       float m11, float m12, float m20, float m21, float m22)
public Matrix3f(float v[])
public Matrix3f(Matrix3d m1)
public Matrix3f(Matrix3f m1)
public Matrix3f()
```

These constructors each return a new Matrix3f object. The first constructor generates a 3×3 matrix from the nine values provided. The second constructor generates a 3×3 matrix from the first nine values in the array `v`. The third and fourth constructors generate a new matrix with the same values as the passed matrix `m1`. The final constructor generates a 3×3 matrix with all nine values set to 0.0.

Methods

```
public final void set(Quat4d q1)
public final void set(Quat4f q1)
```

These two `set` methods set the value of the matrix `this` to the matrix conversion of the quaternion argument `q1`.

```
public final void set(AxisAngle4d a1)
public final void set(AxisAngle4f a1)
```

These two `set` methods set the value of the matrix `this` to the matrix conversion of the axis and angle argument `a1`.

```
public final void set(float scale)
public final void set(float m[])
```

The first method sets the value of this matrix to a scale matrix with the passed `scale` amount. The second method sets the values of this matrix to the row-major array parameter (that is, the first three elements of the array are copied into the first row of this matrix, and so forth).

```
public final void setElement(int row, int column, float value)
public final float getElement(int row, int column)
```

The `setElement` and `getElement` methods provide a means for accessing a single element within a 3×3 matrix using indices. This is not a preferred method of access, but Java 3D provides these methods for functional completeness. The `setElement` method takes a row index `row` (where a value of 0 represents the first row and a value of 2 represents the third row), a column index `column` (where a value of 0 represents the first column and a value of 2 represents the third column), and a value. It sets the corresponding element in matrix `this` to the specified value. The `getElement` method also takes a row index `row` and a column index `column`. It returns the element at the corresponding locations as a floating-point value.

```
public final void setRow(int row, float x, float y, float z)
public final void setRow(int row, Vector3f v)
public final void setRow(int row, float v[])
public final void getRow(int row, Vector3f v)
public final void getRow(int row, float v[])
```

The three `setRow` methods provide a means for constructing a 3×3 matrix on a row basis. The row parameter `row` determines which row the method invocation affects. A row value of 0 represents the first row and a value of 2 represents the third row. The first `setRow` method specifies the three new values as independent floating-point values. The second `setRow` method uses the values in the Vector3f `v` to update the matrix. The third `setRow` method uses the first three values in the array `v` to update the matrix. In all three cases the matrix affected is the matrix `this`. The two `getRow` methods copy the matrix values in the specified row into the vector or array parameter, respectively.

```
public final void setColumn(int column, float x, float y, float z)
public final void setColumn(int column, Vector3f v)
public final void setColumn(int column, float v[])
public final void getColumn(int column, Vector3f v)
public final void getColumn(int column, float v[])
```

The three `setColumn` methods provide a means for constructing a 3×3 matrix on a column basis. The `column` parameter determines which column the method invocation affects. A column value of 0 represents the first column and a value of 2 represents the third column. The first `setColumn` method specifies the three new values as independent floating-point values. The second `setColumn` method uses the values in the Vector3f `v` to update the matrix. The third `setColumn` method uses the first three values in the array `v` to update the matrix. In all three cases the matrix affected is the matrix `this`. The two `getColumn` methods copy the matrix values in the specified column into the vector or array parameter, respectively.

```
public final void setZero()
```

This method sets this matrix to all zeros.

```
public final void setIdentity()
```

This method sets this Matrix3f to identity.

```
public final void add(Matrix3f m1, Matrix3f m2)
public final void add(Matrix3f m1)
public final void sub(Matrix3f m1, Matrix3f m2)
public final void sub(Matrix3f m1)
```

The first `add` method adds the matrix `m1` to the matrix `m2` and places the result into the matrix `this`. The second `add` method adds the matrix `this` to the matrix `m1` and places the result into the matrix `this`. The first `sub` method performs an element-by-element subtraction of matrix `m2` from matrix `m1` and places the result into the matrix `this`. The second `sub` method performs an element-by-element subtraction of the matrix `m1` from the matrix `this` and places the result into the matrix `this`.

```
public final void transform(Tuple3f t)
public final void transform(Tuple3f t, Tuple3f result)
```

The first method multiplies this matrix by the tuple `t` and places the result back into the tuple (t = this*t). The second method multiplies this matrix by the tuple `t` and places the result into the tuple `result` (result = this*t).

```
public final void transpose()
public final void transpose(Matrix3f m1)
```

The first method transposes this matrix in place. The second method sets the value of this matrix to the transpose of the matrix `m1`.

```
public final void invert()
public final void invert(Matrix3f m1)
```

The first method inverts this matrix in place. The second method sets the value of this matrix to the inverse of the matrix `m1`.

```
public final float determinant()
```

The `determinant` method computes the determinant of the matrix `this` and returns the computed value.

```
public final void rotX(float angle)
public final void rotY(float angle)
public final void rotZ(float angle)
```

The three `rot` methods construct rotation matrices that rotate in a clockwise direction around the axis specified as the last letter of the method name. The constructed matrix replaces the value of the matrix `this`. The rotation angle is expressed in radians.

```
public final void mul(Matrix3f m1, Matrix3f m2)
public final void mul(Matrix3f m1)
```

The first `mul` method multiplies matrix `m1` with matrix `m2` and places the result into the matrix `this`. The second `mul` method multiplies the matrix `this` with the matrix `m1` and places the result into matrix `this`.

```
public final void mulNormalize(Matrix3f m1)
public final void mulNormalize(Matrix3f m1, Matrix3f m2)
```

The first `mulNormalize` method multiplies this matrix by matrix `m1`, performs an SVD normalization of the result, and places the result back into this matrix (this = SVDnorm(`this` · `m1`)). The second `mulNormalize` method multiplies matrix `m1` by matrix `m2`, performs an SVD normalization of the result, and places the result into this matrix (this = SVDnorm(`m1` · `m2`)).

public final void mulTransposeBoth(Matrix3f m1, Matrix3f m2)
public final void mulTransposeRight(Matrix3f m1, Matrix3f m2)
public final void mulTransposeLeft(Matrix3f m1, Matrix3f m2)

The `mulTransposeBoth` method multiplies the transpose of matrix `m1` (left) times the transpose of matrix `m2` (right) and places the result into this matrix. The `mulTransposeRight` method multiplies matrix `m1` times the transpose of matrix `m2` and places the result back into this matrix. The `mulTransposeLeft` method multiplies the transpose of matrix `m1` times matrix `m2` and places the result into this matrix.

public final void normalize()
public final void normalize(Matrix3f m1)

The first `normalize` method performs a singular value decomposition normalization of this matrix. The second `normalize` method performs a singular value decomposition normalization of matrix `m1` and places the normalized values into `this`.

public final void normalizeCP()
public final void normalizeCP(Matrix3f m1)

The first `normalizeCP` method performs a cross-product normalization of this matrix. The second `normalizeCP` method performs a cross-product normalization of matrix `m1` and places the normalized values into `this`.

public boolean equals(Matrix3f m1)

The `equals` method returns `true` if all of the data members of Matrix3f `m1` are equal to the corresponding data members in this Matrix3f.

public boolean epsilonEquals(Matrix3f m1, float epsilon)

This method returns `true` if the L_∞ distance between this Matrix3f and Matrix3f `m1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\text{MAX}[i = 0,1,2, \ldots n; j = 0,1,2,\ldots n; \text{abs}(\text{this.m}(i,j) - \text{m1.m}(i,j)]$$

public final void negate()
public final void negate(Matrix3f m1)

The first method negates the value of this matrix in place (this = –this). The second method sets the value of this matrix equal to the negation of the matrix `m1` (this = –m1).

`public final float getScale()`

This method performs an SVD normalization of this matrix to calculate and return the uniform scale factor.

`public final void setScale(float scale)`

This method sets the scale component of the current matrix by factoring out the current scale (by doing an SVD) and multiplying by the new scale.

`public final void add(float scalar)`

This method adds a scalar to each component of this matrix.

`public final void add(float scalar, Matrix3f m1)`

This method adds a scalar to each component of the matrix `m1` and places the result into `this`. Matrix `m1` is not modified.

`public final void mul(float scalar, Matrix3f m1)`

This method multiplies each component of the matrix `m1` by a scalar and places the result into `this`. Matrix `m1` is not modified.

`public final void mul(float scalar)`

This method multiplies each element of this matrix by a scalar.

`public final void transform(Tuple3f t)`
`public final void transform(Tuple3f t, Tuple3f result)`

The first method multiplies this matrix by the tuple `t` and places the result back into the tuple (`t` = `this`*`t`). The second method multiplies this matrix by the tuple `t` and places the result into the tuple `result` (result = `this`*`t`).

`public int hashCode()`

The `hashCode` method returns a hash number based on the data values in this object. Two different Matrix3f objects with identical data values (that is, `equals(Matrix3f)` returns `true`) will return the same hash number. Two Matrix3f objects with different data members may return the same hash value, although this is not likely.

`public String toString()`

The `toString` method returns a string that contains the values of this Matrix3f.

A.2.2 Matrix3d Class

The Matrix3d class serves to contain 3×3 matrices mainly for storing and manipulating 3D rotation matrices. The class includes five different constructors for creating matrices and several operators for manipulating these matrices.

Variables

The component values of a Matrix3d are directly accessible through the public variables `m00`, `m01`, `m02`, `m10`, `m11`, `m12`, `m20`, `m21`, and `m22`. To access the element in row 2 and column 0 of the matrix named `rotate`, a programmer would write `rotate.m20`. Other matrix values are accessed similarly.

```
public double m00
public double m01
public double m02
public double m10
public double m11
public double m12
public double m20
public double m21
public double m22
```

These public variables are the elements of the matrix.

Constructors

```
public Matrix3d(double m00, double m01, double m02, double m10,
       double m11, double m12, double m20, double m21, double m22)
public Matrix3d(double v[])
public Matrix3d()
public Matrix3d(Matrix3d m1)
public Matrix3d(Matrix3f m1)
```

These constructors each return a new Matrix3d object. The first constructor generates a 3×3 matrix from the nine values provided. The second constructor generates a 3×3 matrix from the first nine values in the array `v`. The third constructor generates a 3×3 matrix with all nine values set to 0.0. The fourth and fifth constructors generate a 3×3 matrix with the same values as the matrix `m1` parameter.

Methods

`public final void set(Matrix3f m1)`

This method sets the value of the matrix `this` to the float value of the rotational components of the passed matrix `m1`.

`public final void set(double scale)`
`public final void set(double m[])`

These methods set the value of the matrix `this` to a scale matrix with the passed scale amount.

`public final void set(AxisAngle4d a1)`
`public final void set(AxisAngle4f a1)`

These two `set` methods set the value of the matrix `this` to the matrix conversion of the axis and angle argument `a1`.

`public final void set(Quat4d q1)`
`public final void set(Quat4f q1)`

These two `set` methods set the value of the matrix `this` to the matrix conversion of the quaternion argument `q1`.

`public final void setElement(int row, int column, double value)`
`public final double getElement(int row, int column)`

The `setElement` and `getElement` methods provide a means for accessing a single element within a 3×3 matrix using indices. This is not a preferred method of access, but Java 3D provides these methods for functional completeness. The `setElement` method takes a row index `row` (where a value of 0 represents the first row and a value of 2 represents the third row), a column index `column` (where a value of 0 represents the first column and a value of 2 represents the third column), and a value. It sets the corresponding element in matrix `this` to the specified value. The `getElement` method also takes a row index `row` and a column index `column` and returns the element at the corresponding locations as a floating-point value.

```
public final void setRow(int row, double x, double y, double z)
public final void setRow(int row, Vector3d v)
public final void setRow(int row, double v[])
public final void getRow(int row, Vector3d v)
public final void getRow(int row, double v[])
```

The three `setRow` methods provide a means for constructing a 3×3 matrix on a row basis. The `row` parameter determines which row the method invocation affects. A row value of 0 represents the first row and a value of 2 represents the third row. The first `setRow` method specifies the three new values as independent floating-point values. The second `setRow` method uses the values in the Vector3d `v` to update the matrix. The third `setRow` method uses the first three values in the array `v` to update the matrix. In all three cases the matrix affected is the matrix `this`. The two `getRow` methods copy the matrix values in the specified row into the array or vector parameter, respectively.

```
public final void setColumn(int column, double x, double y,
       double z)
public final void setColumn(int column, Vector3d v)
public final void setColumn(int column, double v[])
public final void getColumn(int column, Vector3d v)
public final void getColumn(int column, double v[])
```

The three `setColumn` methods provide a means for constructing a 3×3 matrix on a column basis. The `column` parameter determines which column the method invocation affects. A column value of 0 represents the first column and a value of 2 represents the third column. The first `setColumn` method specifies the three new values as independent floating-point values. The second `setColumn` method uses the values in the Vector3d `v` to update the matrix. The third `setColumn` method uses the first three values in the array `v` to update the matrix. In all three cases the matrix affected is the matrix `this`. The two `getColumn` methods copy the matrix values in the specified column into the array or vector parameter, respectively.

```
public final void add(Matrix3d m1, Matrix3d m2)
public final void add(Matrix3d m1)
public final void sub(Matrix3d m1, Matrix3d m2)
public final void sub(Matrix3d m1)
```

The first `add` method adds the matrix `m1` to the matrix `m2` and places the result into the matrix `this`. The second `add` method adds the matrix `this` to the matrix `m1` and places the result into the matrix `this`. The first `sub` method performs an element-by-element subtraction of matrix `m2` from matrix `m1` and places the result into the matrix `this`. The second `sub` method performs an element-by-element subtraction of the matrix `m1` from the matrix `this` and places the result into the matrix `this`.

`public final void add(double scalar)`

This method adds a scalar to each component of this matrix.

`public final void add(double scalar, Matrix3d m1)`

This method adds a scalar to each component of the matrix `m1` and places the result into `this`. Matrix `m1` is not modified.

`public final void transform(Tuple3d t)`
`public final void transform(Tuple3d t, Tuple3d result)`

The first method multiplies this matrix by the tuple `t` and places the result back into the tuple (t = this*t). The second method multiplies this matrix by the tuple `t` and places the result into the tuple `result` (result = this*t).

`public final void transpose()`
`public final void transpose(Matrix3d m1)`

The first method transposes this matrix in place. The second method sets the value of this matrix to the transpose of the matrix `m1`.

`public final void invert()`
`public final void invert(Matrix3d m1)`

The first method inverts this matrix in place. The second method sets the value of this matrix to the inverse of the matrix `m1`.

`public final double determinant()`

The `determinant` method computes the determinant of the matrix `this` and returns the computed value.

`public final void rotX(double angle)`
`public final void rotY(double angle)`
`public final void rotZ(double angle)`

The three `rot` methods construct rotation matrices that rotate in a clockwise direction around the axis specified by the final letter of the method name. The constructed matrix replaces the value of the matrix `this`. The rotation angle is expressed in radians.

public final void mul(Matrix3d m1, Matrix3d m2)
public final void mul(Matrix3d m1)

The first `mul` method multiplies matrix `m1` with matrix `m2` and places the result into the matrix `this`. The second `mul` method multiplies matrix `this` with matrix `m1` and places the result into the matrix `this`.

public final void mulNormalize(Matrix3d m1)
public final void mulNormalize(Matrix3d m1, Matrix3d m2)

The first `mulNormalize` method multiplies this matrix by matrix `m1`, performs an SVD normalization of the result, and places the result back into this matrix (this = SVDnorm(`this` · `m1`)). The second `mulNormalize` method multiplies matrix `m1` by matrix `m2`, performs an SVD normalization of the result, and places the result into this matrix (this = SVDnorm(`m1` · `m2`)).

public final void mulTransposeBoth(Matrix3d m1, Matrix3d m2)
public final void mulTransposeRight(Matrix3d m1, Matrix3d m2)
public final void mulTransposeLeft(Matrix3d m1, Matrix3d m2)

The `mulTransposeBoth` method multiplies the transpose of matrix `m1` (left) times the transpose of matrix `m2` (right) and places the result into this matrix. The `mulTransposeRight` method multiplies matrix `m1` times the transpose of matrix `m2` and places the result back into this matrix. The `mulTransposeLeft` method multiplies the transpose of matrix `m1` times matrix `m2` and places the result into this matrix.

public final void normalize()
public final void normalize(Matrix3d m1)

The first `normalize` method performs a singular value decomposition normalization of this matrix. The second `normalize` method performs a singular value decomposition normalization of matrix `m1` and places the normalized values into `this`.

public final void normalizeCP()
public final void normalizeCP(Matrix3d m1)

The first `normalizeCP` method performs a cross-product normalization of this matrix. The second `normalizeCP` method performs a cross-product normalization of matrix `m1` and places the normalized values into `this`.

public boolean equals(Matrix3d m1)

The `equals` method returns `true` if all of the data members of Matrix3d `m1` are equal to the corresponding data members in this Matrix3d.

```
public boolean epsilonEquals(Matrix3d m1, double epsilon)
```

This method returns `true` if the L_∞ distance between this Matrix3d and Matrix3d `m1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

MAX[i = 0,1,2,; j = 0,1,2,; abs(this.m(i,j) – m1.m(i,j)]

```
public final void negate()
public final void negate(Matrix3d m1)
```

The first method negates the value of this matrix in place (this = –this). The second method sets the value of this matrix equal to the negation of the matrix `m1` (this = –m1).

```
public final double getScale()
```

This method performs an SVD normalization of this matrix to calculate and return the uniform scale factor.

```
public final void setScale(double scale)
```

This method sets the scale component of the current matrix by factoring out the current scale (by doing an SVD) and multiplying by the new scale.

```
public final void mul(double scalar, Matrix3d m1)
```

This method multiplies each component of the matrix `m1` by a scalar and places the result into `this`. Matrix `m1` is not modified.

```
public final void mul(double scalar)
```

This method multiplies each element of this matrix by a scalar.

```
public final void transform(Tuple3d t)
public final void transform(Tuple3d t, Tuple3d result)
```

The first method multiplies this matrix by the tuple `t` and places the result back into the tuple (`t = this*t`). The second method multiplies this matrix by the tuple `t` and places the result into the tuple `result` (`result = this*t`).

```
public final void setZero()
```

This method sets this matrix to all zeros.

```
public final void setIdentity()
```

This method sets this Matrix3d to identity.

`public int hashCode()`

The `hashCode` method returns a hash number based on the data values in this object. Two different Matrix3d objects with identical data values (that is, `equals(Matrix3d)` returns `true`) will return the same hash number. Two Matrix3d objects with different data members may return the same hash value, although this is not likely.

`public String toString()`

The `toString` method returns a string that contains the values of this Matrix3d.

A.2.3 Matrix4f Class

The Matrix4f class serves to contain 4 × 4 matrices mainly for storing and manipulating 3D transformation matrices. The class includes seven different constructors for creating matrices and several operators for manipulating these matrices.

Variables

The component values of a Matrix4f are directly accessible through the public variables `m00`, `m01`, `m02`, `m03`, `m10`, `m11`, `m12`, `m13`, `m20`, `m21`, `m22`, `m23`, `m30`, `m31`, `m32`, and `m33`. To access the element in row 2 and column 0 of matrix `rotate`, a programmer would write `rotate.m20`. A programmer would access the other values similarly.

```
public float m00
public float m01
public float m02
public float m03
public float m10
public float m11
public float m12
public float m13
public float m20
public float m21
public float m22
public float m23
public float m30
public float m31
public float m32
public float m33
```

These public variables are the elements of the matrix.

Constructors

```
public Matrix4f(float m00, float m01, float m02, float m03,
       float m10, float m11, float m12, float m13,
       float m20, float m21, float m22, float m23,
       float m30, float m31, float m32, float m33)
public Matrix4f(float v[])
public Matrix4f(Quat4f q1, Vector3f t1, float s)
public Matrix4f(Matrix4d m1)
public Matrix4f(Matrix4f m1)
public Matrix4f(Matrix3f m1, Vector3f t1, float s)
public Matrix4f()
```

These constructors each return a new Matrix4f object. The first constructor generates a 4×4 matrix from the 16 values provided. The second constructor generates a 4×4 matrix from the first 16 values in the array `v`. The third constructor generates a 4×4 matrix from the quaternion, translation, and scale values. The scale is applied only to the rotational components of the matrix (upper 3×3) and not to the translational components. The fourth and fifth constructors generate a 4×4 matrix with the same values as the passed matrix `m1`. The sixth constructor generates a 4×4 matrix from the rotation matrix, translation, and scale values. The scale is applied only to the rotational components of the matrix (upper 3×3) and not to the translational components of the matrix. The final constructor generates a 4×4 matrix with all 16 values set to 0.0.

Methods

```
public final void set(Quat4f q1)
public final void set(Quat4d q1)
public final void set(Quat4f q1, Vector3f t1, float s)
public final void set(Quat4d q1, Vector3d t1, double s)
public final void set(Matrix4d m1)
public final void set(Matrix4f m1)
public final void set(AxisAngle4f a1)
public final void set(AxisAngle4d a1)
```

The first two `set` methods set the value of this matrix to the matrix conversion of the quaternion argument `q1`. The next two `set` methods set the value of this matrix from the rotation expressed by the quaternion `q1`, the translation `t1`, and the scale `s`. The next two `set` methods set the value of this matrix to a copy of the passed matrix `m1`. The last two `set` methods set the value of this matrix to the matrix conversion of the axis and angle argument `a1`.

```
public final void set(Matrix3f m1)
public final void set(Matrix3d m1)
```

These methods set the rotational component (upper 3 × 3) of this matrix to the matrix values in the `m1` argument. The other elements of this matrix are initialized as if this were an identity matrix (that is, an affine matrix with no translational component).

```
public final void set(float scale)
public final void set(float m[])
```

The first method sets the value of this matrix to a scale matrix with the passed scale amount. The second method sets the value of this matrix to the row-major array parameter (that is, the first four elements of the array are copied into the first row of this matrix, and so forth).

```
public final void set(Vector3f v1)
```

This method sets the value of this matrix to a translation matrix with the passed translation value.

```
public final void set(float scale, Vector3f t1)
public final void set(Vector3f t1, float scale)
```

These methods set the value of this matrix to a scale and translation matrix. In the first method, the scale is not applied to the translation, and all of the matrix values are modified. In the second method, the translation is scaled by the scale factor, and all of the matrix values are modified.

```
public final void set(Matrix3f m1, Vector3f t1, float scale)
public final void set(Matrix3d m1, Vector3d t1, double scale)
```

These two methods set the value of this matrix from the rotation expressed by the rotation matrix `m1`, the translation `t1`, and the scale `scale`. The translation is not modified by the scale.

```
public final void get(Matrix3d m1)
public final void get(Matrix3f m1)
public final float get(Matrix3f m1, Vector3f t1)
public final void get(Quat4f q1)
public final void get(Vector3f trans)
```

The first two methods perform an SVD normalization of this matrix in order to acquire the normalized rotational component. The values are placed into the matrix parameter `m1`. The third method performs an SVD normalization of this matrix to calculate the rotation as a 3 × 3 matrix, the translation, and the scale.

None of the matrix values in this matrix are modified. The fourth method performs an SVD normalization of this matrix to acquire the normalized rotational component. The values are placed into the quaternion `q1`. The final method retrieves the translational components of this matrix and copies them into the vector `trans`.

```
public final void setElement(int row, int column, float value)
public final float getElement(int row, int column)
```

The `setElement` and `getElement` methods provide a means for accessing a single element within a 4×4 matrix using indices. This is not a preferred method of access, but Java 3D provides these methods for functional completeness. The `setElement` method takes a row index `row` (where a value of 0 represents the first row and a value of 3 represents the fourth row), a column index `column` (where a value of 0 represents the first column and a value of 3 represents the fourth column), and a value. It sets the corresponding element in matrix `this` to the specified value. The `getElement` method also takes a row index `row` and a column index `column` and returns the element at the corresponding locations as a floating-point value.

```
public final void getRotationScale(Matrix3f m1)
```

This method retrieves the upper 3×3 values of this matrix and places them into the matrix `m1`.

```
public final void setScale(float scale)
public final float getScale()
```

The first method sets the scale component of the current matrix by factoring out the current scale (by doing an SVD) and multiplying by the new scale. The second method performs an SVD normalization of this matrix to calculate and return the uniform scale factor.

```
public final void add(float scalar)
```

This method adds a scalar to each component of this matrix.

```
public final void add(float scalar, Matrix4f m1)
```

This method adds a scalar to each component of the matrix `m1` and places the result into `this`. Matrix `m1` is not modified.

```
public final void mul(float scalar, Matrix4f m1)
```

This method multiplies each component of the matrix `m1` by a scalar and places the result into `this`. Matrix `m1` is not modified.

```
public final void mul(float scalar)
```

This method multiplies each element of this matrix by a scalar.

```
public final void setRow(int row, float x, float y, float z,
       float w)
public final void setRow(int row, Vector4f v)
public final void setRow(int row, float v[])
public final void getRow(int row, Vector4f v)
public final void getRow(int row, float v[])
```

The three `setRow` methods provide a means for constructing a 4×4 matrix on a row basis. The row parameter `row` determines which row the method invocation affects. A row value of 0 represents the first row and a value of 3 represents the fourth row. The first `setRow` method specifies the four new values as independent floating-point values. The second `setRow` method uses the values in the Vector4f `v` to update the matrix. The third `setRow` method uses the first four values in the array `v` to update the matrix. In all three cases the matrix affected is the matrix `this`. The two `getRow` methods copy the matrix values in the specified row into the array or vector parameter, respectively.

```
public final void setColumn(int column, float x, float y, float z,
       float w)
public final void setColumn(int column, Vector4f v)
public final void setColumn(int column, float v[])
public final void getColumn(int column, Vector4f v)
public final void getColumn(int column, float v[])
```

The three `setColumn` methods provide a means for constructing a 4×4 matrix on a column basis. The `column` parameter determines which column the method invocation affects. A column value of 0 represents the first column and a value of 3 represents the fourth column. The first `setColumn` method specifies the four new values as independent double-precision floating-point values. The second `setColumn` method uses the values in the Vector4f `v` to update the matrix. The third `setColumn` method uses the first four values in the array `v` to update the matrix. In all three cases the matrix affected is the matrix `this`. The two `getColumn` methods copy the matrix values in the specified column into the array or vector parameter, respectively.

```
public final void setRotation(Matrix3d m1)
public final void setRotation(Matrix3f m1)
public final void setRotation(Quat4f q1)
public final void setRotation(Quat4d q1)
public final void setRotation(AxisAngle4f a1)
```

These methods set the rotational component (upper 3 × 3) of this matrix to the matrix values in the passed argument. The other elements of this matrix are unchanged. In the first two methods, a singular value decomposition is performed on this object's upper 3 × 3 matrix to factor out the scale, then this object's upper 3 × 3 matrix components are replaced by the passed rotation components, and finally the scale is reapplied to the rotational components. In the next two methods, a singular value decomposition is performed on this object's upper 3 × 3 matrix to factor out the scale, then this object's upper 3 × 3 matrix components are replaced by the matrix equivalent of the quaternion, and finally the scale is reapplied to the rotational components. In the last method, a singular value decomposition is performed on this object's upper 3 × 3 matrix to factor out the scale, then this object's upper 3 × 3 matrix components are replaced by the matrix equivalent of the axis-angle, and finally the scale is reapplied to the rotational components.

```
public final void setRotationScale(Matrix3f m1)
```

This method replaces the upper 3 × 3 matrix values of this matrix with the values in the matrix `m1`.

```
public final void setTranslation(Vector3f trans)
```

This method modifies the translational components of this matrix to the values of the vector `trans`. The other values of this matrix are not modified.

```
public final void setIdentity()
```

This method sets this Matrix4f to identity.

```
public final void setZero()
```

This method sets this matrix to all zeros.

```
public final void add(Matrix4f m1, Matrix4f m2)
public final void add(Matrix4f m1)
public final void sub(Matrix4f m1, Matrix4f m2)
public final void sub(Matrix4f m1)
```

The first `add` method adds the matrix `m1` to the matrix `m2` and places the result into the matrix `this`. The second `add` method adds the matrix `this` to the matrix `m1` and places the result into the matrix `this`. The first `sub` method performs an element-by-element subtraction of matrix `m2` from matrix `m1` and places the result into the matrix `this`. The second `sub` method performs an element-by-element subtraction of the matrix `m1` from the matrix `this` and places the result into the matrix `this`.

`public final void transpose(Matrix4f m1)`
`public final void transpose()`

The first `transpose` method transposes the matrix `m1` and places the result into the matrix `this`. The second `transpose` method transposes the matrix `this` and places the result back into the matrix `this`.

`public final void transform(Point3f point)`
`public final void transform(Point3f point, Point3f pointOut)`

The first `transform` method postmultiplies this matrix by the Point3f `point` and places the result back into `point`. The multiplication treats the three-element point as if its fourth element were 1. The second `transform` method postmultiplies this matrix by the Point3f `point` and places the result into `pointOut`.

`public final void transform(Vector3f normal)`
`public final void transform(Vector3f normal, Vector3f normalOut)`

The first `transform` method postmultiplies this matrix by the Vector3f `normal` and places the result back into `normal`. The multiplication treats the three-element vector as if its fourth element were 0. The second `transform` method postmultiplies this matrix by the Vector3f `normal` and places the result into `normalOut`.

`public final void transform(Tuple4f vec)`
`public final void transform(Tuple4f vec, Tuple4f vecOut)`

The first `transform` method postmultiplies this matrix by the tuple `vec` and places the result back into `vec`. The second `transform` method postmultiplies this matrix by the tuple `vec` and places the result into `vecOut`.

`public final void negate()`
`public final void negate(Matrix4f m1)`

The first method negates the value of this matrix in place (this = –this). The second method sets the value of this matrix equal to the negation of the matrix `m1` (this = –m1).

```
public final void invert()
public final void invert(Matrix4f m1)
```

The first method inverts this matrix in place. The second method sets the value of this matrix to the inverse of the matrix `m1`.

```
public final float determinant()
```

The `determinant` method computes the determinant of the matrix `this` and returns the computed value.

```
public final void rotX(float angle)
public final void rotY(float angle)
public final void rotZ(float angle)
```

The three `rot` methods construct rotation matrices that rotate in a clockwise direction around the axis specified as the last letter of the method name. The constructed matrix replaces the value of the matrix `this`. The rotation angle is expressed in radians.

```
public final void mul(Matrix4f m1, Matrix4f m2)
public final void mul(Matrix4f m1)
```

The first `mul` method multiplies matrix `m1` with matrix `m2` and places the result into the matrix `this`. The second `mul` method multiplies the matrix `this` with matrix `m1` and places the result in matrix `this`.

```
public final void mulTransposeBoth(Matrix4f m1, Matrix4f m2)
public final void mulTransposeRight(Matrix4f m1, Matrix4f m2)
public final void mulTransposeLeft(Matrix4f m1, Matrix4f m2)
```

The `mulTransposeBoth` method multiplies the transpose of matrix `m1` (left) times the transpose of matrix `m2` (right) and places the result into this matrix. The `mulTransposeRight` method multiplies matrix `m1` times the transpose of matrix `m2` and places the result back into this matrix. The `mulTransposeLeft` method multiplies the transpose of matrix `m1` times matrix `m2` and places the result into this matrix.

```
public boolean equals(Matrix4f m1)
```

The `equals` method returns `true` if all of the data members of Matrix4f `m1` are equal to the corresponding data members in this Matrix4f.

`public boolean epsilonEquals(Matrix4f m1, float epsilon)`

This method returns `true` if the L_∞ distance between this Matrix4f and Matrix4f `m1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

MAX[i = 0,1,2,3; j = 0,1,2,3; abs(this.m(i,j) – m1.m(i,j))]

`public int hashCode()`

The `hashCode` method returns a hash number based on the data values in this object. Two different Matrix4f objects with identical data values (that is, `equals(Matrix4f)` returns `true`) will return the same hash number. Two Matrix4f objects with different data members may return the same hash value, although this is not likely.

`public String toString()`

The `toString` method returns a string that contains the values of this Matrix4f.

A.2.4 Matrix4d Class

The Matrix4d class serves to contain 4 × 4 matrices mainly for storing and manipulating 3D transformation matrices. The class includes nine different constructors for creating matrices and several operators for manipulating these matrices.

Variables

The component values of a Matrix4d are directly accessible through the public variables `m00`, `m01`, `m02`, `m03`, `m10`, `m11`, `m12`, `m13`, `m20`, `m21`, `m22`, `m23`, `m30`, `m31`, `m32`, and `m33`. To access the element in row 2 and column 0 of matrix `rotate`, a programmer would write `rotate.m20`. A programmer would access the other values similarly.

```
public double m00
public double m01
public double m02
public double m03
public double m10
public double m11
public double m12
public double m13
public double m20
public double m21
public double m22
public double m23
public double m30
public double m31
public double m32
public double m33
```

These public variables are the elements of the matrix.

Constructors

```
public Matrix4d(double m00, double m01, double m02, double m03,
       double m10, double m11, double m12, double m13, double m20,
       double m21, double m22, double m23, double m30, double m31,
       double m32, double m33)
public Matrix4d(double v[])
public Matrix4d(Quat4d q1, Vector3d t1, double s)
public Matrix4d(Quat4f q1, Vector3d t1, double s)
public Matrix4d(Matrix3d m1, Vector3d t1, double s)
public Matrix4d(Matrix3f m1, Vector3d t1, double s)
public Matrix4d(Matrix4d m1)
public Matrix4d(Matrix4f m1)
public Matrix4d()
```

These constructors each return a new Matrix4d object. The first constructor generates a 4×4 matrix from the 16 values provided. The second constructor generates a 4×4 matrix from the first 16 values in the array `v`. The third through sixth constructors generate a 4×4 matrix from the quaternion, translation, and scale values. The scale is applied only to the rotational components of the matrix (upper 3×3) and not to the translational components. The seventh and eighth constructors generate a 4×4 matrix with the same values as the passed matrix. The final constructor generates a 4×4 matrix with all 16 values set to 0.0.

Methods

```
public final void get(Matrix3d m1)
public final void get(Matrix3f m1)
public final double get(Matrix3d m1, Vector3d t1)
public final double get(Matrix3f m1, Vector3d t1)
public final void get(Quat4f q1)
public final void get(Quat4d q1)
public final void get(Vector3d trans)
```

The first two methods perform an SVD normalization of this matrix in order to acquire the normalized rotational component. The values are placed into the passed parameter. The next two methods perform an SVD normalization of this matrix to calculate the rotation as a 3×3 matrix, the translation, and the scale. None of the matrix values are modified. The next two methods perform an SVD normalization of this matrix to acquire the normalized rotational component. The last two methods retrieve the translational components of this matrix.

```
public final void setElement(int row, int column, double value)
public final double getElement(int row, int column)
```

The `setElement` and `getElement` methods provide a means for accessing a single element within a 4×4 matrix using indices. This is not a preferred method of access, but Java 3D provides these methods for functional completeness. The `setElement` method takes a row index `row` (where a value of 0 represents the first row and a value of 3 represents the fourth row), a column index `column` (where a value of 0 represents the first column and a value of 3 represents the fourth column), and a value. It sets the corresponding element in matrix `this` to the specified value. The `getElement` method also takes a row index `row` and a column index `column` and returns the element at the corresponding locations as a floating-point value.

```
public final void setRow(int row, double x, double y, double z,
       double w)
public final void setRow(int row, Vector4d v)
public final void setRow(int row, double v[])
public final void getRow(int row, Vector4d v)
public final void getRow(int row, double v[])
```

The three `setRow` methods provide a means for constructing a 4×4 matrix on a row basis. The `row` parameter determines which row the method invocation affects. A row value of 0 represents the first row and a value of 3 represents the fourth row. The first `setRow` method specifies the four new values as independent floating-point values. The second `setRow` method uses the values in the Vector4d `v` to update the matrix. The third `setRow` method uses the first four values in the array `v` to update the matrix. In all three cases the matrix affected is the matrix `this`. The two `getRow` methods copy the matrix values in the specified row into the array or vector parameter, respectively.

```
public final void setColumn(int column, double x, double y,
       double z, double w)
public final void setColumn(int column, Vector4d v)
public final void setColumn(int column, double v[])
public final void getColumn(int column, Vector4d v)
public final void getColumn(int column, double v[])
```

The three `setColumn` methods provide a means for constructing a 4×4 matrix on a column basis. The `column` parameter determines which column the method invocation affects. A column value of 0 represents the first column and a value of 3 represents the fourth column. The first `setColumn` method specifies the four new values as independent double-precision floating-point values. The second `setColumn` method uses the values in the Vector4d `v` to update the matrix. The third `setColumn` method uses the first four values in the array `v` to update the matrix. In all three cases the matrix affected is the matrix `this`. The two `getColumn` methods copy the matrix values in the specified column into the array or vector parameter, respectively.

```
public final void setRotation(Matrix3f m1)
public final void setRotation(Matrix3d m1)
```

These methods set the rotational component (upper 3×3) of this matrix to the matrix values in the passed argument. The other elements of this matrix are unchanged. A singular value decomposition is performed on this object's upper 3×3 matrix to factor out the scale, then this object's upper 3×3 matrix components are replaced by the passed rotation components, and finally the scale is reapplied to the rotational components.

```
public final void setRotation(Quat4f q1)
public final void setRotation(Quat4d q1)
```

These methods set the rotational component (upper 3×3) of this matrix to the matrix values in the passed argument. The other elements of this matrix are unchanged. A singular value decomposition is performed on this object's upper 3×3 matrix to factor out the scale, then this object's upper 3×3 matrix components are replaced by the matrix equivalent of the quaternion, and finally the scale is reapplied to the rotational components.

`public final void setRotation(AxisAngle4d a1)`

This method sets the rotational component (upper 3 × 3) of this matrix to the equivalent values in the passed argument. The other elements of this matrix are unchanged. A singular value decomposition is performed on this object's upper 3 × 3 matrix to factor out the scale, then this object's upper 3 × 3 matrix components are replaced by the matrix equivalent of the axis-angle, and finally the scale is reapplied to the rotational components.

`public final void getRotationScale(Matrix3f m1)`
`public final void getRotationScale(Matrix3d m1)`
`public final void setRotationScale(Matrix3d m1)`
`public final void setRotationScale(Matrix3f m1)`

The two `get` methods retrieve the upper 3 × 3 values of this matrix and place them into the matrix `m1`. The two `set` methods replace the upper 3 × 3 matrix values of this matrix with the values in the matrix `m1`.

`public final void setTranslation(Vector3d trans)`

This method modifies the translational components of this matrix to the values of the Vector3d argument. The other values of this matrix are not modified.

`public final void setScale(double scale)`
`public final double getScale()`

The first method sets the scale component of the current matrix by factoring out the current scale (by doing an SVD) and multiplying by the new scale. The second method performs an SVD normalization of this matrix to calculate and return the uniform scale factor.

`public final void add(double scalar)`

This method adds a scalar to each component of this matrix.

`public final void add(double scalar, Matrix4d m1)`

This method adds a scalar to each component of the matrix `m1` and places the result into `this`. Matrix `m1` is not modified.

`public final void mul(double scalar, Matrix4d m1)`

This method multiplies each component of the matrix `m1` by a scalar and places the result into `this`. Matrix `m1` is not modified.

public final void mul(double scalar)

This method multiplies each element of this matrix by a scalar.

public final void add(Matrix4d m1, Matrix4d m2)
public final void add(Matrix4d m1)
public final void sub(Matrix4d m1, Matrix4d m2)
public final void sub(Matrix4d m1)

The first `add` method adds the matrix `m1` to the matrix `m2` and places the result into the matrix `this`. The second `add` method adds the matrix `this` to the matrix `m1` and places the result into the matrix `this`. The first `sub` method performs an element-by-element subtraction of matrix `m2` from matrix `m1` and places the result into the matrix `this`. The second `sub` method performs an element-by-element subtraction of the matrix `m1` from the matrix `this` and places the result into the matrix `this`.

public final void set(double m[])

This method sets the value of this matrix to the row-major array parameter (that is, the first four elements of the array will be copied into the first row of this matrix, and so forth).

public final void set(Matrix3f m1)
public final void set(Matrix3d m1)

These methods set the rotational component (upper 3×3) of this matrix to the matrix values in the matrix argument. The other elements of this matrix are initialized as if this were an identity matrix (that is, an affine matrix with no translational component).

public final void set(Matrix4f m1)
public final void set(Matrix4d m1)

These methods set the value of this matrix to the value of the passed matrix `m1`.

public final void set(Quat4d q1)
public final void set(Quat4f q1)

These methods set the value of this matrix to the matrix conversion of the quaternion argument.

public final void set(AxisAngle4d a1)
public final void set(AxisAngle4f a1)

These methods set the value of this matrix to the matrix conversion of the axis and angle argument.

public final void set(Vector3d v1)

This method sets the value of this matrix to a translation matrix by the passed translation value.

public final void set(Quat4d q1, Vector3d t1, double s)
public final void set(Quat4f q1, Vector3d t1, double s)
public final void set(Quat4f q1, Vector3f t1, float s)

These methods set the value of this matrix to the rotation expressed by the quaternion `q1`, the translation `t1`, and the scale `s`.

public final void set(double scale)

This method sets the value of this matrix to a scale matrix with the passed scale amount.

public final void set(double scale, Vector3d v1)

This method sets the value of this matrix to a scale and translation matrix. The scale is not applied to the translation, and all of the matrix values are modified.

public final void set(Vector3d v1, double scale)

This method sets the value of this matrix to a scale and translation matrix. The translation is scaled by the scale factor, and all of the matrix values are modified.

public final void set(Matrix3f m1, Vector3f t1, float scale)
public final void set(Matrix3d m1, Vector3d t1, double scale)

These methods set the value of this matrix from the rotation expressed by the rotation matrix `m1`, the translation `t1`, and the scale `s`.

public final void negate(Matrix4d m1)
public final void negate()

The first method sets the value of this matrix to the negation of the `m1` parameter. The second method negates the value of this matrix (this = –this).

public final void transpose(Matrix4d m)
public final void transpose()

The first `transpose` method transposes the matrix `m` and places the result into the matrix `this`. The second `transpose` method transposes the matrix `this` and places the result back into the matrix `this`.

```
public final void transform(Tuple4d vec)
public final void transform(Tuple4f vec)
public final void transform(Tuple4d vec, Tuple4d vecOut)
public final void transform(Tuple4f vec, Tuple4f vecOut)
```

The first two `transform` methods postmultiply this matrix by the tuple `vec` and place the result back into `vec`. The last two `transform` methods postmultiply this matrix by the tuple `vec` and place the result into `vecOut`.

```
public final void transform(Point3d point)
public final void transform(Point3f point)
public final void transform(Point3d point, Point3d pointOut)
public final void transform(Point3f point, Point3f pointOut)
```

The first two `transform` methods postmultiply this matrix by the point argument `point` and place the result back into `point`. The multiplication treats the three-element point as if its fourth element were 1. The last two `transform` methods postmultiply this matrix by the point argument `point` and place the result into `pointOut`.

```
public final void transform(Vector3d normal)
public final void transform(Vector3f normal)
public final void transform(Vector3d normal, Vector3d normalOut)
public final void transform(Vector3f normal, Vector3f normalOut)
```

The first two `transform` methods postmultiply this matrix by the vector argument `normal` and place the result back into `normal`. The multiplication treats the three-element vector as if its fourth element were 0. The last two `transform` methods postmultiply this matrix by the vector argument `normal` and place the result into `normalOut`.

```
public final void invert()
public final void invert(Matrix4d m1)
```

The first method inverts this matrix in place. The second method sets the value of this matrix to the inverse of the matrix `m1`.

```
public final double determinant()
```

The `determinant` method computes the determinant of the matrix `this` and returns the computed value.

```
public final void rotX(double angle)
public final void rotY(double angle)
public final void rotZ(double angle)
```

The `rot` methods construct rotation matrices that rotate in a clockwise direction around the axis specified as the last letter of the method name. The constructed matrix replaces the value of the matrix `this`. The rotation angle is expressed in radians.

```
public final void mul(Matrix4d m1, Matrix4d m2)
public final void mul(Matrix 4d m1)
```

The first `mul` method multiplies matrix `m1` with matrix `m2` and places the result into the matrix `this`. The second `mul` method multiplies matrix `this` with matrix `m1` and places the result into the matrix `this`.

```
public final void mulTransposeBoth(Matrix4d m1, Matrix4d m2)
public final void mulTransposeRight(Matrix4d m1, Matrix4d m2)
public final void mulTransposeLeft(Matrix4d m1, Matrix4d m2)
```

The `mulTransposeBoth` method multiplies the transpose of matrix `m1` (left) times the transpose of matrix `m2` (right) and places the result into this matrix. The `mulTransposeRight` method multiplies matrix `m1` times the transpose of matrix `m2` and places the result back into this matrix. The `mulTransposeLeft` method multiplies the transpose of matrix `m1` times matrix `m2` and places the result into this matrix.

```
public final void setZero()
```

This method sets this matrix to all zeros.

```
public final void setIdentity()
```

This method sets this Matrix4d to identity.

```
public boolean equals(Matrix4d m1)
```

The `equals` method returns `true` if all of the data members of Matrix4d `m1` are equal to the corresponding data members in this Matrix4d.

```
public boolean epsilonEquals(Matrix4d m1, float epsilon)
```

This method returns `true` if the L_∞ distance between this Matrix4d and Matrix4d `m1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\text{MAX}[i = 0,1,2,3;\ j = 0,1,2,3;\ \text{abs}(\text{this.m}(i,j) - \text{m1.m}(i,j))]$$

`public int hashCode()`

The `hashCode` method returns a hash number based on the data values in this object. Two different Matrix4d objects with identical data values (that is, `equals(Matrix4d)` returns `true`) will return the same hash number. Two Matrix4d objects with different data members may return the same hash value, although this is not likely.

`public String toString()`

The `toString` method returns a string that contains the values of this Matrix4d.

A.2.5 GMatrix Class

The GMatrix class serves to contain a double-precision, general, and dynamically resizeable $M \times N$ matrix. Row and column numbering begins with zero. The representation is row major.

The GMatrix data members are not public, thus allowing efficient implementations of sparse matrices. However, the data members can be modified through public accessors. The class includes three different constructors for creating matrices and several operators for manipulating these matrices.

Constructors

```
public GMatrix(int nRow, int nCol)
public GMatrix(int nRow, int nCol, double matrix[])
public GMatrix(GMatrix matrix)
```

These constructors each return a new GMatrix. The first constructor generates an `nRow` by `nCol` identity matrix. The second constructor generates an `nRow` by `nCol` matrix initialized to the values in the array `matrix`. The last constructor generates a new GMatrix and copies the initial values from the parameter `matrix` argument.

Methods

```
public final void mul(GMatrix m1, GMatrix m2)
public final void mul(GMatrix m1)
```

The first `mul` method multiplies matrix `m1` with matrix `m2` and places the result into `this`. The second `mul` method multiplies this matrix with matrix `m1` and places the result into `this`.

public final void add(GMatrix m1)
public final void add(GMatrix m1, GMatrix m2)
public final void sub(GMatrix m1)
public final void sub(GMatrix m1, GMatrix m2)

The first `add` method adds this matrix to matrix `m1` and places the result back into `this`. The second `add` method adds matrices `m1` and `m2` and places the result into `this`. The first `sub` method subtracts matrix `m1` from the matrix `this` and places the result into `this`. The second `sub` method subtracts matrix `m2` from matrix `m1` and places the result into the matrix `this`.

public final void negate()
public final void negate(GMatrix m1)

The first method negates the value of this matrix in place (this = –this). The second method sets the value of this matrix to the negation of the matrix `m1` (this = –m1).

public final void invert()
public final void invert(GMatrix m1)

The first method inverts this matrix in place. The second method sets the value of this matrix to the inverse of the matrix `m1`.

public final void setIdentity()

This method sets this GMatrix to the identity matrix.

public final void setZero()

This method sets all the values in this matrix to zero.

public final void identityMinus()

This method subtracts this matrix from the identity matrix and puts the values back into `this` (this = I – this).

public final void copySubMatrix(int rowSource, int colSource, int numRow, int numCol, int rowDest, int colDest, GMatrix target)

This method copies a submatrix derived from this matrix into the target matrix. The `rowSource` and `colSource` parameters define the upper left of the submatrix. The `numRow` and `numCol` parameters define the number of rows and columns in the submatrix. The submatrix is copied into the target matrix starting at (`rowDest`, `colDest`). The `target` parameter is the matrix into which the submatrix will be copied.

public final void setSize(int nRow, int nCol)

This method changes the size of this matrix dynamically. If the size is increased, no data values will be lost. If the size is decreased, only those data values whose matrix positions were eliminated will be lost.

public final void set(double matrix[])
public final void set(GMatrix m1)
public final void set(Matrix3f m1)
public final void set(Matrix3d m1)
public final void set(Matrix4f m1)
public final void set(Matrix4d m1)

The first `set` method sets the values of this matrix to the values found in the `matrix` array parameter. The values are copied in one row at a time, in row-major fashion. The array should be at least equal in length to the number of matrix rows times the number of matrix columns in this matrix. The second `set` method sets the values of this matrix to the values found in matrix `m1`. The last four `set` methods set the values of this matrix to the values found in matrix `m1`.

public final void get(Matrix3d m1)
public final void get(Matrix3f m1)
public final void get(Matrix4d m1)
public final void get(Matrix4f m1)
public final void get(GMatrix m1)

The first two methods place the values in the upper 3×3 of this matrix into the matrix `m1`. The next two methods place the values in the upper 4×4 of this matrix into the matrix `m1`. The final method places the values in this matrix into the matrix `m1`. Matrix `m1` should be at least as large as this matrix.

public final int getNumRow()
public final int getNumCol()

The `getNumRow` method returns the number of rows in this matrix. The `getNum-Col` method returns the number of columns in this matrix.

public final void setElement(int row, int column, double value)
public final double getElement(int row, int column)

These methods set and retrieve the value at the specified `row` and `column` of this matrix.

public final void setRow(int row, double array[])
public final void setRow(int row, GVector vector)
public final void getRow(int row, double array[])

public final void getRow(int row, GVector vector)
public final void setColumn(int col, double array[])
public final void setColumn(int col, GVector vector)
public final void getColumn(int col, double array[])
public final void getColumn(int col, GVector vector)

The `setRow` methods copy the values from the array into the specified row of this matrix. The `getRow` methods place the values of the specified row into the array or vertex. The `setColumn` methods copy the values from the array into the specified column of this matrix or vector. The `getColumn` methods place the values of the specified column into the array or vector.

public final void setScale(double scale)

This method sets this matrix to a uniform scale matrix, and all of the values are reset.

public final void setZero()

Sets all the values in this matrix to zero.

public final void mulTransposeBoth(GMatrix m1, GMatrix m2)
public final void mulTransposeRight(GMatrix m1, GMatrix m2)
public final void mulTransposeLeft(GMatrix m1, GMatrix m2)

The `mulTransposeBoth` method multiplies the transpose of matrix `m1` (left) times the transpose of matrix `m2` (right) and places the result into this matrix. The `mulTransposeRight` method multiplies matrix `m1` times the transpose of matrix m2 and places the result back into this matrix. The `mulTransposeLeft` method multiplies the transpose of matrix `m1` times matrix `m2` and places the result into this matrix.

public final void transpose()
public final void transpose(GMatrix m1)

The first `transpose` method transposes this matrix in place. The second `transpose` method places the matrix values of the transpose of matrix `m1` into this matrix.

public String toString()

This method returns a string that contains the values of this GMatrix.

public int hashCode()

This method returns a hash number based on the data values in this object. Two different `GMatrix` objects with identical data values (that is, `equals(GMatrix)` returns `true`) will return the same hash number. Two objects with different data members may return the same hash value, although this is not likely.

public boolean equals(GMatrix m1)

This method returns `true` if all of the data members of GMatrix `m1` are equal to the corresponding data members in this GMatrix.

public boolean epsilonEquals(GMatrix m1, float epsilon)

Deprecated method. See the next method.

public boolean epsilonEquals(GMatrix m1, double epsilon)

This method returns `true` if the L_∞ distance between this GMatrix and GMatrix `m1` is less than or equal to the `epsilon` parameter. Otherwise, this method returns `false`. The L_∞ distance is equal to

$$\text{MAX}[i = 0,1,2, \ldots n; j = 0,1,2,\ldots n; \text{abs}(\text{this.m}(i,j) - \text{m1.m}(i,j))]$$

public final double trace()

This method returns the trace of this matrix.

public final int SVD(GMatrix U, GMatrix W, GMatrix V)

The `SVD` method finds the singular value decomposition (SVD) of this matrix such that `this` $= \mathbf{U} * \mathbf{W} * \mathbf{V}^T$, and returns the rank of this matrix. The values of `U`, `W`, and `V` are all overwritten. Note that the matrix `V` is output as $\mathbf{V}$ and not $\mathbf{V}^T$. If this matrix is $m \times n$, then $\mathbf{U}$ is $m \times m$, $\mathbf{W}$ is a diagonal matrix that is $m \times n$, and $\mathbf{V}$ is $n \times n$. The inverse of this matrix is $\texttt{this}^{-1} = \mathbf{V} * \mathbf{W}^{-1} * \mathbf{U}^T$, where $\mathbf{W}^{-1}$ is a diagonal matrix computed by taking the reciprocal of each of the diagonal elements of matrix $\mathbf{W}$.

public final int LUD(GMatrix LU, GVector permutation)

The `LUD` method performs an LU decomposition. This matrix must be a square matrix, and the `LU` parameter must be the same size as this matrix. The diagonal elements of **L** (unity) are not stored. The `permutation` parameter records the row permutation affected by the partial pivoting, and is used as a parameter to the GVector `LUDBackSolve` method to solve sets of linear equations. This method returns +1 or –1, depending on whether the number of row interchanges was even or odd, respectively.

APPENDIX B

3D Geometry Compression

JAVA 3D allows programmers to specify geometry using a binary geometry compression format. This compression format is used with APIs other than just Java 3D, and can be used both as a runtime in-memory format for describing geometry, as well as a storage and network format. Eventually the full specification of the geometry compression format described in this section will be part of its own stand-alone specification, but for completeness it is included as an appendix to the early specification of the Java 3D API.

Java 3D uses a geometry compression format that allows 3D geometry to be represented in an order of magnitude less space than most traditional 3D representations, with very little loss in object quality. The compression is achieved through several layers of techniques.

B.1 Compression

First, the geometry to be compressed is converted into a generalized mesh form, which allows a triangle to be, on average, specified by 0.80 vertices.

Next the data for each vertex component of the geometry is converted to the most efficient representation format for its type and then quantized to as few bits as possible.

These quantized bits are differenced between successive vertices, and the results are modified Huffman encoded into self-describing variable-bit-length data elements.

Finally, these variable-length elements are strung together using Java 3D's eight geometry commands into a final compressed geometry block.

B.2 Decompression

Upon receipt, compressed geometry blocks are decompressed into the local host's preferred geometry format by reversing the above process.

B.3 Appendix Organization

Before the bit details of the compression can be specified, several of the concepts used in geometry compression need elaboration. The first several sections are an expansion of our SIGGRAPH '95 paper on geometry compression.[1]

- *Generalized Triangle Strip*. This section is a refresher on the concept and semantics of a generalized triangle strip.
- *Generalized Triangle Mesh*. This section introduces the concept and semantics of a generalized triangle mesh.
- *Color Representation and Quantization*. This section describes the fixed-point format used for 3D positional representation.
- *Color Representation and Quantization*. This section describes the fixed-point format used for color representation.
- *Normal Representation and Quantization*. This section describes a novel folded table based representation of surface normals, and the fixed-point format of the resultant normals.
- *Modified Huffman Encoding*. This section describes the variant of Huffman delta encoding used for geometry compression.
- *Geometry Compression Commands*. This section gives an overview of the eight geometry compression commands.

B.4 Generalized Triangle Strip

A generalized triangle strip is a generalization of the concept of a "zig-zag" and "star" triangle strip. It is a sequence of vertices in which each vertex contains a two-bit replacement code. This replacement code defines how the present vertex is to be combined with previous vertices to form the next triangle. The replacement bits can also be thought of as a generalization of the "move/draw" bit used for lines.

1. Deering, Michael. "Geometry Compression." *Computer Graphics Proceedings*, Annual Conference Series, 1995, ACM SIGGRAPH, pp 13–19.

A stack of the last three vertices used to form a triangle is kept. The three vertices are labeled oldest, middle, and newest. An incoming vertex of type `replace_oldest` causes the oldest vertex to be replaced by the middle, the middle to be replaced by the newest, and the incoming vertex to become the newest. This corresponds to a PHIGS PLUS triangle strip (sometimes called a "zig-zag" strip). The replacement type `replace_middle` leaves the oldest vertex unchanged, replaces the middle vertex by the newest, and the incoming vertex becomes the newest. This corresponds to a triangle star or fan.

The replacement type `restart` marks the oldest and middle vertices as invalid, and the incoming vertex becomes the newest. Generalized triangle strips must always start with this code. A triangle will be output only when a replacement operation results in three valid vertices.

`Restart` corresponds to a "move" operation in polylines, and allows multiple unconnected variable-length triangle strips to be described by a single data structure passed in by the user, reducing the overhead. The generalized triangle strip's ability to effectively change from "strip" to "star" mode in the middle of a strip allows more complex geometry to be represented compactly, and requires less input data bandwidth. The restart capability allows several pieces of disconnected geometry to be passed as one data block. Figure B-1 shows a single generalized triangle strip and the associated replacement codes.

Triangles are normalized such that the front face is always defined by a clockwise vertex order after transformation. To support this, there are two flavors of restart: `restart_clockwise` and `restart_counterclockwise`. The vertex order is reversed after every `replace_oldest`, but remains the same after every `replace_middle`.

B.5 Generalized Triangle Mesh

The first stage of geometry compression is to convert triangle data into an efficient linear strip form: the *generalized triangle mesh*. This is a near-optimal representation of triangle data, given fixed storage.

The existing concept of a generalized triangle strip structure allows for compact representation of geometry while maintaining a linear data structure. That is, the geometry can be extracted by a single monotonic scan over the vertex array data structure. This is very important for pipelined hardware implementations, a data format that requires random access back to main memory during processing is very problematic.

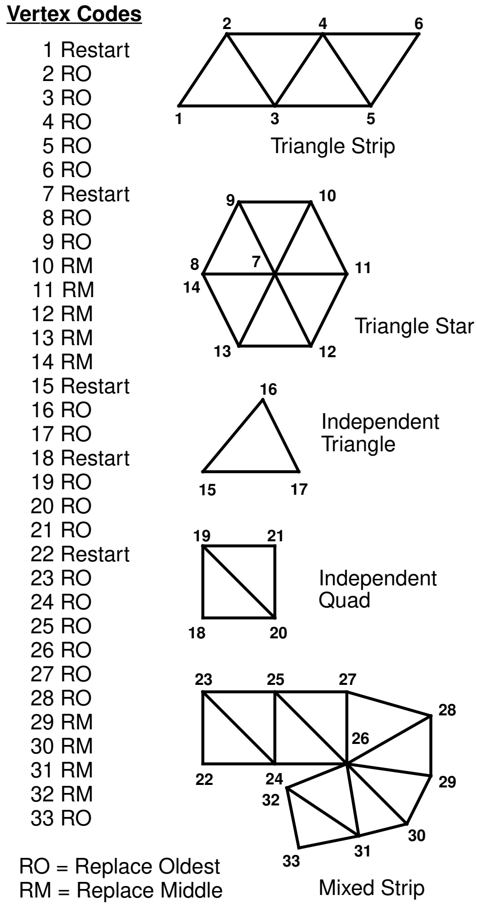
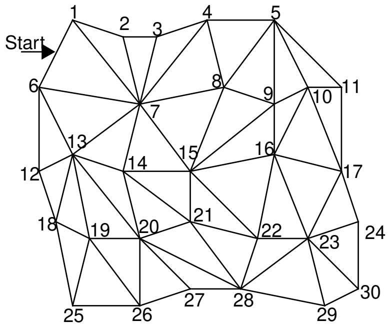


Figure B-1 A Generalized Triangle Strip

However, by confining itself to linear strips, the generalized triangle strip format leaves a potential factor of two (in space) on the table. Consider the geometry in Figure B-2.

While it can be represented by one triangle strip, many of the interior vertices appear twice in the strip. This is inherent in any approach wishing to avoid references to old data. Some systems have tried using a simple regular mesh buffer to support reuse of old vertices, but there is a problem with this approach in practice: In general, geometry does not come in a perfectly regular rectangular mesh structure.



Generalized Triangle Strip:
R6, O1, O7, O2, O3, M4, M8, O5, O9, O10, M11,
M17, M16, M9, O15, O8, O7, M14, O13, M6,
O12, M18, M19, M20, M14, O21, O15, O22, O16,
O23, O17, O24, M30, M29, M28, M22, O21, M20,
M27, O26, M19, O25, O18

Generalized Triangle Mesh:
R6p, O1, O7p, O2, O3, M4, M8p, O5, O9p, O10, M11,
M17p, M16p, M-3, O15p, O-5, O6, M14p, O13p, M-9,
O12, M18p, M19p, M20p, M-5, O21p, O-7, O22p, O-9,
O23, O-10, O-7, M30, M29, M28, M-1, O-2, M-3,
M27, O26, M-4, O25, O-5

Legend:
First letter: R = Restart, O = Replace Oldest, M = Replace Middle
Trailing “p” = push into mesh buffer
Number is vertex number, -number is mesh buffer reference
where -1 is most recent pushed vertex.

Figure B-2 A Generalized Triangle Mesh

The generalized technique employed by geometry compression addresses this problem. Old vertices are *explicitly* pushed into a queue, and then explicitly referenced in the future when the old vertex is desired again. This fine control supports irregular meshes of nearly any shape. Any viable technique must recognize that storage is finite; thus the maximum queue length is fixed at 16, requiring a four-bit index. We refer to this queue as the *mesh buffer.* The combination of generalized triangle strips and mesh buffer references is referred to as a *generalized triangle mesh*.

The fixed mesh buffer size requires all tessellators or restrippers for compressed geometry to break up any runs longer than 16 unique references. Since geometry compression is not meant to be programmed directly at the user level, but rather by sophisticated tessellators or reformatters, this is not too onerous a restriction. Sixteen old vertices allow up to 94 percent of the redundant geometry to avoid being respecified. Figure B-2 also contains an example of a general mesh buffer representation of the surface geometry.

The language of geometry compression supports the four vertex replacement codes of generalized triangle strips (replace oldest, replace middle, restart clockwise, and restart counterclockwise), and adds another bit in each vertex header to indicate if this vertex should be pushed into the mesh buffer or not. The mesh buffer reference command has a four-bit field to indicate which old vertex should be rereferenced, along with the two-bit vertex replacement code. Mesh buffer reference commands do *not* contain a mesh buffer push bit; old vertices can only be recycled once.

Geometry rarely is composed purely of positional data; generally a normal and/or color are also specified per vertex. Therefore, mesh buffer entries contain storage for all associated per-vertex information (specifically including normal and color).

For maximum space efficiency, when a vertex is specified in the data stream, (per-vertex) normal and/or color information should be directly bundled with the position information. This bundling is controlled by two state bits: *bundle normals with vertices* (`bnv`), and *bundle colors with vertices* (`bcv`). When a vertex is pushed into the mesh buffer, these bits control whether its bundled normal and/or color are pushed as well. During a mesh buffer reference command, this process is reversed. The two bits specify if a normal and/or color should be inherited from the mesh buffer storage, or inherited from the *current normal* or *current color*.

There are explicit commands for setting these two current values. An important exception to this rule occurs when an explicit "set current normal" command is followed by a mesh buffer reference, with the `bnv` state bit active. In this case, the former overrides the mesh buffer normal. This allows compact representation of hard edges in surface geometry. The analogous semantics are also defined for colors, allowing compact representation of hard edges in textures.

B.6 Position Representation and Quantization

The 8-bit exponent of 32-bit IEEE floating-point numbers allows positions literally to span the known universe: from a scale of 100 billion light years, down to the radius of subatomic particles. However, for any given tessellated object the exponent is really specified just once by the current modeling matrix; within a given modeling space, the object geometry is effectively described with only the 24-bit fixed-point mantissa. Visually, in many cases far fewer bits are needed; thus the language of geometry compression supports variable quantization of position data down to as little as one bit. The maximum number of bits supported is at most 16 bits of precision per component of position.

We still assume that the position and scale of the local modeling spaces are specified by full 32-bit or 64-bit floating-point coordinates. If sufficient numerical care is taken, multiple such modeling spaces can be stitched together without cracks, forming seamless geometry coordinate systems with much greater than 16-bit positional precision.

Most geometry is local, so within the 16-bit (or less) modeling space (of each object), the delta difference between one vertex and the next in the generalized mesh buffer stream is very likely to be less than 16 bits in significance. Indeed one can histogram the bit length of neighboring position deltas in a batch of geometry and, based on this histogram, assign a variable-length code to compactly represent the vertices. The typical coding used in many other similar situations is customized Huffman code; this is the case for geometry compression. The details of the coding of position deltas will be postponed until later, where they can be discussed in the context of color and normal delta coding as well.

B.7 Color Representation and Quantization

We treat colors similar to positions, but without using negative values. Thus RGBα color data is first quantized to 15-bit unsigned fraction components, and a zero sign bit added to form a 16-bit signed number. These are absolute linear reflectivity values, with 1.0 representing 100 percent reflectivity. An additional parameter allows color data to be quantized effectively to any amount less than 16 bits; that is, the colors can all be within a 5-5-5 RGB color space. (The α field is optional, controlled by the *color alpha present* (`cap`) state bit.) Note that this decision does *not* necessarily cause mach banding on the final rendered image; individual pixel colors are still interpolated between these quantized vertex colors, and vertices also are subject to lighting.

The same delta coding is used for color components as is used for positions. Compression of color data is where geometry compression and traditional image compression face the most similar problem. However, many of the more advanced techniques for image compression were rejected for geometry color compression because of the difference in focus.

Image compression makes several assumptions about the viewing of the decompressed data that *cannot* be made for geometry compression. In image compression, it is known a priori that the pixels appear in a perfectly rectangular array, and that when viewed, each pixel subtends a narrow range of visual angles. In geometry compression, one has almost no idea what the relationship between the viewer and the rasterized geometry will be.

In image compression, it is known that the spatial frequency of the displayed pixels on the viewer's eyes is likely higher than the human visual system's color acuity. This is why colors are usually converted to yuv space, so that the uv color components can be represented at a lower spatial frequency than the y (intensity) component.

Usually the digital bits representing the subsampled uv components are split up among two or more pixels. Geometry compression cannot take advantage of this because the display scale of the geometry relative to the viewer's eye is not fixed. Also, given that compressed triangle vertices are connected to four to eight or more other vertices in the generalized triangle mesh, there is no consistent way of sharing "half" the color information across vertices.

Similar arguments apply for the more sophisticated transforms used in traditional image compression, such as the discrete cosine transform. These transforms assume a regular (rectangular) sampling of pixel values, and require a large amount of random access during decompression.

B.8 Normal Representation and Quantization

Probably the most innovative concept in geometry compression is the method of compressing surface normals. Traditionally, 96-bit normals (three 32-bit IEEE floating-point numbers) are used in calculations to determine 8-bit color intensities. Theoretically, 96 bits of information could be used to represent 2^{96} different normals, spread evenly over the surface of a unit sphere. This is a normal every 2^{-46} radians in any direction. Such angles are so exact that spreading out angles evenly in every direction from earth, you could point out any rock on Mars with subcentimeter accuracy.

But for normalized normals, the exponent bits are effectively unused. Given the constraint $|N| = 1$, at least one of N_x, N_y, or N_z must be in the range of 0.5 to 1.0. During rendering, this normal will be transformed by a composite modeling orientation matrix T: $N' = N \cdot T$.

Assuming the typical implementation in which lighting is performed in world coordinates, the view transform is not involved in the processing of normals. If the normals have been prenormalized, then to avoid redundant renormalization of the normals, the composite modeling transformation matrix T is typically prenormalized to divide out any scale changes, and thus

$$\mathbf{T}_{0,0}^2 + \mathbf{T}_{1,0}^2 + \mathbf{T}_{2,0}^2 = 1, \text{ etc.}$$

During the normal transformation, floating-point arithmetic hardware effectively truncates all additive arguments to the accuracy of the largest component. The result is that for a normalized normal being transformed by a scale-preserving modeling orientation matrix, the numerical accuracy of the transformed normal value is reduced to no more than 24-bit fixed-point accuracy in all but a few special cases.

Even 24-bit normal components are still much higher in angular accuracy than the (repaired) Hubble space telescope. After empirical tests, it was determined that an angular density of 0.01 radians between normals gave results that were not visually distinguishable from finer representations. This works out to about 100,000 normals distributed over the unit sphere. In rectilinear space, these normals still require high accuracy of representation; we chose to use 16-bit components that include one sign and one guard bit.

This still requires 48 bits to represent a normal. But since we are only interested in 100,000 specific normals, in theory a single 17-bit index could denote any of these normals. The next section shows how it is possible to take advantage of this observation.

B.8.1 Normals as Indices

The most obvious hardware implementation for converting an index of a normal on the unit sphere back into an N_x N_y N_z value is by table look-up. The problem is the size of the table. Fortunately, several symmetry tricks can be applied to greatly reduce the size of the table (by a factor of 48).

First, the unit sphere is symmetrical in the eight quadrants by sign bits. In other words, if we let three of the normal representation bits be the three sign bits of the XYZ components of the normal, then we only need to find a way to represent one eighth of the unit sphere.

Second, each octant of the unit sphere can be split up into six identical pieces by folding about the planes $X = Y$, $X = Z$, and $Y = Z$. (See Figure B-3.) The six possible sextants are encoded with another three bits. Now only 1/48 of the sphere remains to be represented.

This reduces the 100,000-entry look-up table by a factor of 48, requiring only about 2,000 entries, small enough to fit into an on-chip ROM look-up table. This table needs 11 address bits to index into it, so including our previous two 3-bit fields, the result is a grand total of 17 bits for all three normal components.

Representing a finite set of unit normals is equivalent to positioning points on the surface of the unit sphere. While no perfectly equal angular density distribution exists for large numbers of points, many near-optimal distributions exist. Thus in theory one of these with the same sort of 48-way symmetry described above could be used for the decompression look-up table. However, several additional constraints mandate a different choice of encoding:

- We desire a scalable density distribution in which zeroing more and more of the low-order address bits to the table still results in fairly even density of normals on the unit sphere. Otherwise a different look-up table for every encoding density would be required.
- We desire a delta-encodable distribution. Statistically, adjacent vertices in geometry will have normals that are nearby on the surface of the unit sphere. Nearby locations on the 2D space of the unit-sphere surface are most succinctly encoded by a 2D offset. We desire a distribution where such a metric exists.
- Finally, while the computational cost of the normal encoding process is not too important, in general, distributions with lower encoding costs are preferred.

For all these reasons, we decided to use a regular grid in the angular space within one sextant as our distribution. Thus, rather than a monolithic 11-bit index, all normals within a sextant are much more conveniently represented as two 6-bit orthogonal angular addresses, revising our grand total to 18 bits. Just as for positions and colors, if more quantization of normals is acceptable, then these 6-bit indices can be reduced to fewer bits, and thus absolute normals can be represented using anywhere from 18 to as few as 6 bits. But as will be seen, we can delta-encode this space, further reducing the number of bits required for high-quality representation of normals.

B.8.2 Normal Encoding Parameterization

Points on a unit radius sphere are parameterized by two angles, θ and ϕ, using spherical coordinates. θ is the angle about the Y-axis; ϕ is the longitudinal angle from the $y = 0$ plane. The mapping between rectangular and spherical coordinates is as follows:

$$x = \cos\theta \cdot \cos\phi \qquad y = \sin\phi \qquad z = \sin\theta \cdot \cos\phi \tag{B.1}$$

Points on the sphere are folded first by octant, and then by sort order of *xyz* into one of six sextants. All the table encoding takes place in the positive octant, in the region bounded by the half spaces:

$$x \ge z \qquad z \ge y \qquad y \ge 0$$

This triangular-shaped patch runs from 0 to $\pi/4$ radians in θ, and from 0 to as much as 0.615479709 radians in ϕ: ϕ_{max}.

Quantized angles are represented by two n-bit integers $\hat{\theta}_n$ and $\hat{\phi}_n$, where n is in the range of 0 to 6. For a given n, the relationship between these indices θ and ϕ is

$$\begin{aligned} \theta(\hat{\theta}_n) &= \operatorname{asin}\tan(\phi_{max} \cdot (n - \hat{\theta}_n)/2^n) \\ \phi(\hat{\phi}_n) &= \phi_{max} \cdot \hat{\phi}_n/2^n \end{aligned} \tag{B.2}$$

These two equations show how values of $\hat{\theta}_n$ and $\hat{\phi}_n$ can be converted to spherical coordinates θ and ϕ, which in turn can be converted to rectilinear normal coordinate components via equation B.1.

To reverse the process, for example, to encode a given normal n into $\hat{\theta}_n$ and $\hat{\phi}_n$, one cannot just invert equation B.2. Instead, the n must first be folded into the canonical octant and sextant, resulting in n'. Then n' must be dotted with all quantized normals in the sextant. For a fixed n, the values of $\hat{\theta}_n$ and $\hat{\phi}_n$ that result in the largest (nearest unity) dot product define the proper encoding of n.

Now the complete bit format of absolute normals can be given. The uppermost three bits specify the octant, the next three bits the sextant, and finally two n-bit fields specify $\hat{\theta}_n$ and $\hat{\phi}_n$. The three-bit sextant field takes on one of six values, the binary codes for which are shown in Figure B-3.

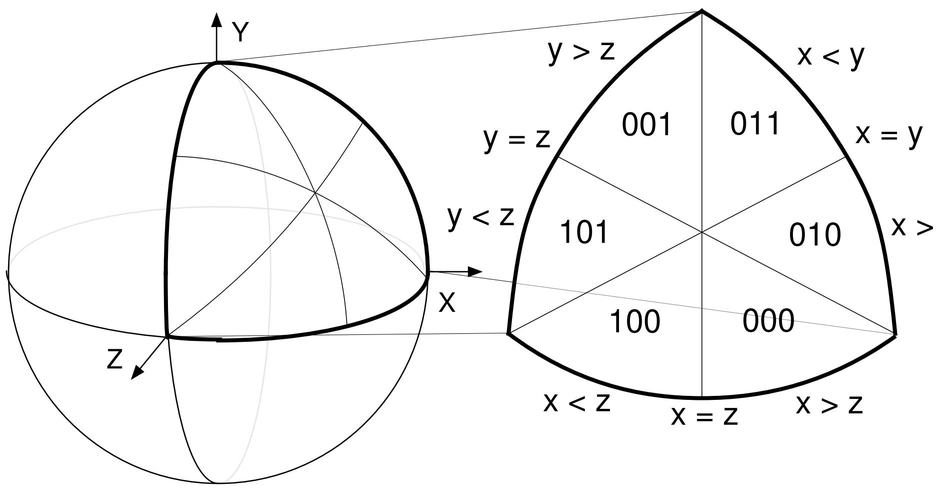


Figure B-3 Encoding of the Six Sextants of Each Octant of a Sphere

This discussion has ignored some details. In particular, the three normals at the corners of the canonical patch are multiply represented (6, 8, and 12 times). By employing the two unused values of the sextant field, these normals can be uniquely encoded as special normals. The `normal` subcommand describes the special encoding used for two of these corner cases (14 total special normals).

This representation of normals is amenable to delta encoding, at least within a sextant. (With some additional work, this can be extended to sextants that share a common edge.) The delta code between two normals is simply the difference in $\hat{\theta}_n$ and $\hat{\phi}_n$: $\Delta\hat{\theta}_n$ and $\Delta\hat{\phi}_n$.

B.9 Modified Huffman Encoding

There are many techniques known for minimally representing variable-length bit fields. For geometry compression, we have chosen a variation of the conventional Huffman technique.

The Huffman compression algorithm takes in a set of symbols to be represented, along with frequency of occurrence statistics (histograms) of those symbols. From this, variable-length, uniquely identifiable bit patterns are generated that allow these symbols to be represented with a near-minimum total number of bits, assuming that symbols do occur at the frequencies specified.

Many compression techniques, including JPEG, create unique symbols as tags to indicate the length of a variable-length data field that follows. This data field is typically a specific-length delta value. Thus the final binary stream consists of (self-describing length) variable-length tag symbols, each immediately followed by a data field whose length is associated with that unique tag symbol.

The binary format for geometry compression uses this technique to represent position, normal, and color data fields. For geometry compression, these <tag, data> fields are immediately preceded by (a more conventional computer instruction set) opcode field. These fields, plus potential additional operand bits, are referred to as *geometry instructions* (see Figure B-4).

Traditionally, each value to be compressed is assigned its own associated label; for example, an XYZ delta position would be represented by three tag/value pairs. However, the delta XYZ values are not uncorrelated, and we can get both a denser and simpler representation by taking advantage of this fact.

In general, the XYZ deltas statistically point equally in all directions in space. This means that if the number of bits to represent the largest of these deltas is n, then statistically the other two delta values require an average of n – 1.4 bits for their representation. Thus we made the decision to use a single field-length tag to indicate the bit length of ΔX, ΔY, and ΔZ.

This also means that we cannot take advantage of another Huffman technique that saves somewhat less than one more bit per component, but our bit savings by not having to specify two additional tag fields (for ΔY and ΔZ) outweigh this. A single tag field also means that a hardware decompression engine can decompress all three fields in parallel, if desired.

Similar arguments hold for deltas of RGBα values, and so here also a single field-length tag indicates the bit-length of the ΔR, ΔG, ΔB, and Δα (if present) fields.

Both absolute and delta normals are also parameterized by a single value (n), which can be specified by a single tag.

We chose to limit the length of the Huffman tag field to the relatively small value of six bits. This was done to facilitate high-speed, low-cost hardware implementations. (A 64-entry tag look-up table allows decoding of tags in one clock cycle.) Three such tables exist: one each for positions, normals, and colors. The tables contain the length of the tag field, the length of the data field(s), a data normalization coefficient, and an absolute/relative bit.

One additional complication was required to enable reasonable hardware implementations. As will be seen in a later section, all instructions are broken up into an eight-bit header and a variable-length body. Sufficient information is present in the header to determine the length of the body. But to give the hardware time to process the header information, the header of one instruction must be placed in the stream before the body of the previous instruction. Thus the sequence … B0 H1B1 H2B2 H3 … has to be encoded as follows:

… H1 B0 H2 B1 H3 B2 …

B.10 Geometry Compression Commands

Java 3D's geometry compression protocol defines eight commands to be used in specifying 3D geometry and certain affiliated attributes. This section gives a brief overview of these commands and some of their semantics. More detail of these commands, including their bit layout, is given in the following sections.

`vertex`

The primary command is `vertex`. A `vertex` command always specifies a 3D position, two generalized triangle strip replacement bits (`rep`), and a mesh buffer push (mbp) bit, and may optionally specify a normal and/or a color (or texture map coordinate). The presence of normal or color data within a vertex command is controlled by two state bits known as the bundling bits: bnv and bcv, respectively.

normal, color

There are also two stand-alone commands for specifying normals and colors: normal and color. These commands may be freely interspersed with vertex commands, and semantically have (nearly) the same effect as normals or colors bundled directly with a normal.

Once a color or normal value is specified, either directly or bundled with a vertex command, that color or normal will remain in effect as the current color or normal until a new value is specified. In this fashion, for example, a constant material color may be specified to apply to a forthcoming sequence of non-color-bundled vertices.

setState

The setState command updates the value of the three state bits. Two of these bits are the normal and color bundling bits; the other one will be described later.

meshBufferReference

The meshBufferReference command allows any of the 16 vertices most recently pushed into the mesh buffer to be reused in place of a vertex command at this point. Two vertex replacement bits are also present.

setTable

The setTable command allows a range of entries in one of the three Huffman decompression tables all to be set to the same new value.

passthrough

The passthrough command allows other data to be embedded in the compression stream.

NOP

The variable length no-operation NOP command allows the compression bit stream to be padded by a specified number of bits. This allows portions of the compression data to be 32-bit aligned when desired.

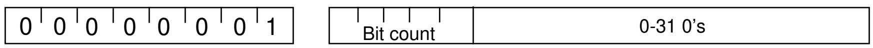
B.11 Bit Layout of Geometry Decompression Commands

Figure B-4 shows the bit-level layout of the eight geometry decompression commands. Each command has a unique opcode, and then some (possible variable) number of arguments. The actual bit length of many of the components may vary, and if so, a unique (dynamic) Huffman tag at the very start of any variable-length argument delimits the size of the argument.

B.12 Geometry Decompression Command Bit Details

The following subsections describe the bit details of several of the geometry decompression commands, and much of their associated semantics.

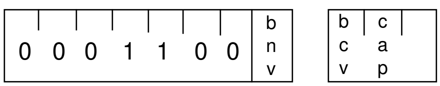
B.12.1 NOP



The variable length no-operation (NOP) command has an 8-bit opcode, a 5-bit count field, and a 0- to 31-bit field of zeros. The total length of the variable-length no-operation command is between 13 and 44 bits.

The variable-length NOP command's primary use is to align geometry decompression commands to word boundaries, when desired. This is useful if one wishes to "patch" a decompression instruction in the middle of a stream without having to bit-align the patch.

B.12.2 setState



The `setState` command has a 7-bit opcode, 3 bits of state to be set, and a spare, for a total length of 11 bits. The first and second state bits indicate if normals and/or colors will be bundled with `vertex` commands, respectively. The third state bit indicates if colors will contain an alpha value, in addition to the standard RGB. The final state bit is unused, and reserved for future use.

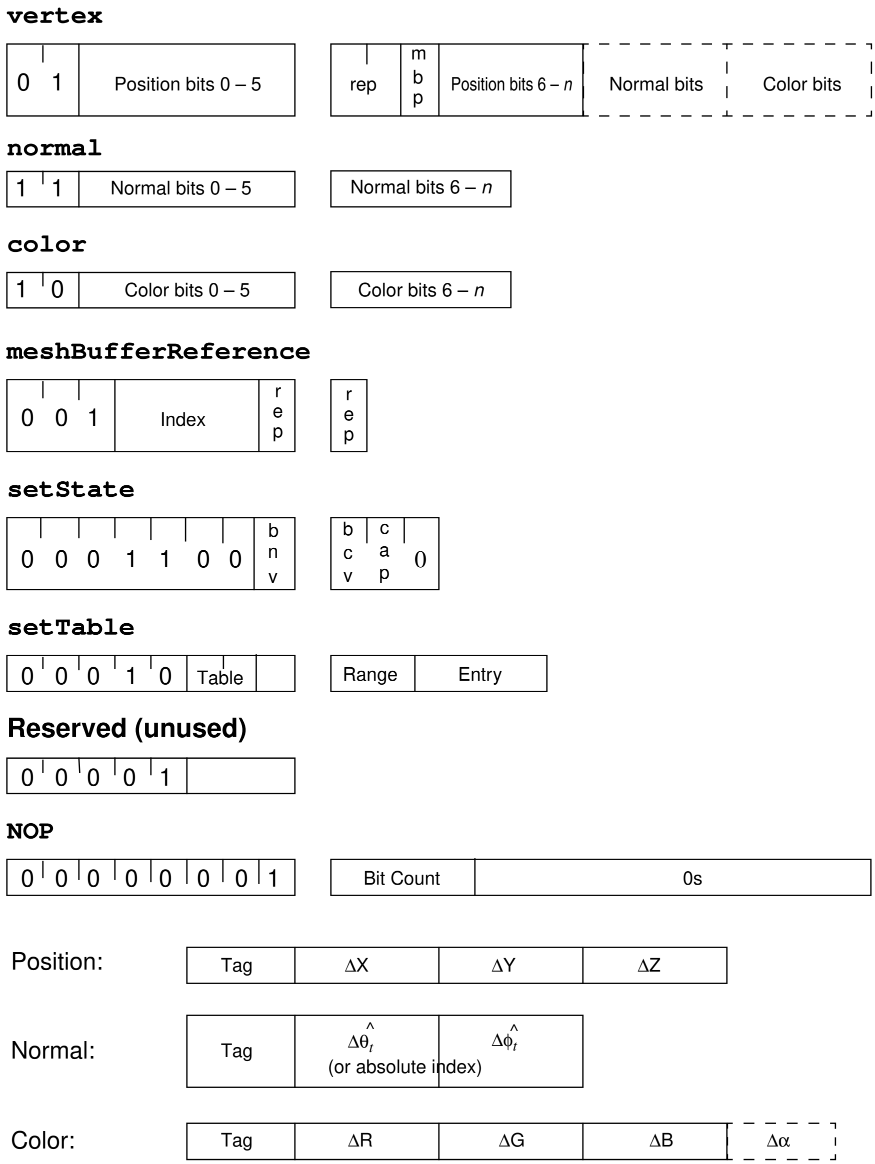
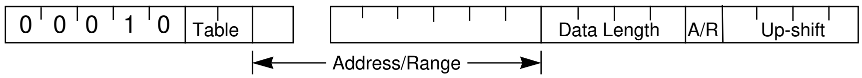


Figure B-4 Bit Layout of Geometry Compression Commands

B.12.3 setTable



The `setTable` command has a 5-bit op code, a 2-bit table field, a 7-bit address/range field, a 4-bit data length field, an absolute/relative bit, and a 4-bit up-shift field. The total instruction length is fixed at 23 bits. The table and address/range fields specify which decompression table entries to update; the remaining fields comprise the values to which to update the table entries.

The two-bit table specifies for which of the three decompression tables this update is targeted:

00	Position
01	Color
10	Normal
11	Unused—reserved for future use

The seven-bit address/range field specifies which entries in the specified table are to be set to the values in the following fields.

Address/Range	Semantics	Implicit Tag Length
$1a_5a_4a_3a_2a_1a_0$	set table entry $a_5a_4a_3a_2a_1a_0$	6
$01a_5a_4a_3a_2a_1$	set table entry $a_5a_4a_3a_2a_10$ through $a_5a_4a_3a_2a_11$	5
$001a_5a_4a_3a_2$	set table entry $a_5a_4a_3a_200$ through $a_5a_4a_3a_211$	4
$0001a_5a_4a_3$	set table entry $a_5a_4a_3000$ through $a_5a_4a_3111$	3
$00001a_5a_4$	set table entry a_5a_40000 through a_5a_41111	2
$000001a_5$	set table entry a_500000 through a_511111	1
0000001	set table entry 000000 through 111111	0

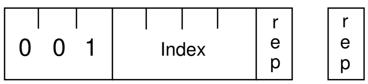
The idea is that table settings are made in aligned power-of-two ranges. The position of the first ‘1’ bit in the address/range field indicates how many entries are to be consecutively set; the remaining bits after the first ‘1’ are the upper address bits of the base of the table entries to be set. This also sets the length of the “tag” that this entry defines as equal to the number of address bits (if any) after the first ‘1’ bit.

The data length specifies how large the delta values to be associated with this tag are; a data length of 12 implies that the upper 4 bits are to be sign extensions of the incoming delta value. Note that the data length describes not the length of the delta value coming in, but the final position of the delta value for reconstruction. In other words, the data length field is the sum of the actual delta bits to be read in plus the up-shift amount. For the position and color tables, the data length values of 1 to 15 correspond to lengths of 1 to 15, but the data length value of 0 encodes an actual length of 16, as a length of 0 makes no sense for positions and colors. For normals, a length of 0 is sometimes appropriate, and the maximum length needed is only 7. Thus for normals, the values 0 to 7 map through 0 to 7, and 8 to 15 are not used.

The up-shift value is the number of bits that the delta values described by these tags will be shifted up before being added to the current value.

The absolute/relative flag indicates whether this table entry describes values that are to be interpreted as an absolute reference or a relative delta. Note that for normals, absolute references will have an additional six leading bits describing the absolute octant and sextant.

B.12.4 meshBufferReference



The `meshBufferReference` command has a 3-bit opcode, a 4-bit mesh buffer index field, and a 2-bit vertex replacement field, for a total length of nine bits.

The index specifies which element of the mesh buffer should be used to define the current vertex. A value of 0 indicates to use the most recent vertex that has been pushed into the mesh buffer (before this command). Larger values indicate successively less recent pushes. Only the most recent 16 pushes are addressable.

The two-bit vertex replacement field has the same triangle semantics as it does within the `vertex` command:

0 0	Restart clockwise
0 1	Restart counterclockwise
1 0	Replace middle
1 1	Replace oldest

There is no mesh buffer re-push bit; mesh buffer contents may be referenced multiple times until 16 newer vertices have been pushed; if a vertex is still needed it must be resent.

B.12.5 Position Subcommand

	0–6	1–16	1–16	1–16
position:	Tag	ΔX	ΔY	ΔZ

The `position` subcommand can only appear within a geometry decompression `vertex` command, and always as the first subcommand. The tag field can be between 0 and 6 bits in length; all three delta (or absolute) fields will have the same length, between 1 and 16 bits, for a range of lengths between 3 and 54 bits. As usual, any subcommand with a total length of less than 6 bits has trailing zeros added to pad the length to a minimum of 6 bits.

As usual, the first six bits of the subcommand are actually forwarded ahead of the rest of the command. Depending on the length of the tag and delta fields, the first 6 bits might only contain the tag, or the tag and some of the *X* field bits, or any subset up to the entire subcommand, if short enough.

For clarity, because it is by far the most typical case, the three coordinate bit fields are labeled ΔX ΔY ΔZ, though more properly they are X, Y, and Z fields; their actual interpretation is absolute or relative depending on the setting of that bit in the decompression table entry corresponding to the tag field. In both cases the fields are signed two's-complement numbers.

B.12.6 Color Subcommand

	0–6	1–16	1–16	1–16	1–16
color:	Tag	ΔR	ΔG	ΔB	Δα

The `color` subcommand can appear within either a geometry decompression `vertex` command or `color` command. The tag field can be between 0 and 6 bits in length; all three (or four) delta (or absolute) fields will have the same length, between 1 and 16 bits, for a range of lengths between 3 and 54 (or 70) bits. As usual, any subcommand with a total length of less than 6 bits has trailing zeros added to pad the length to a minimum of 6 bits.

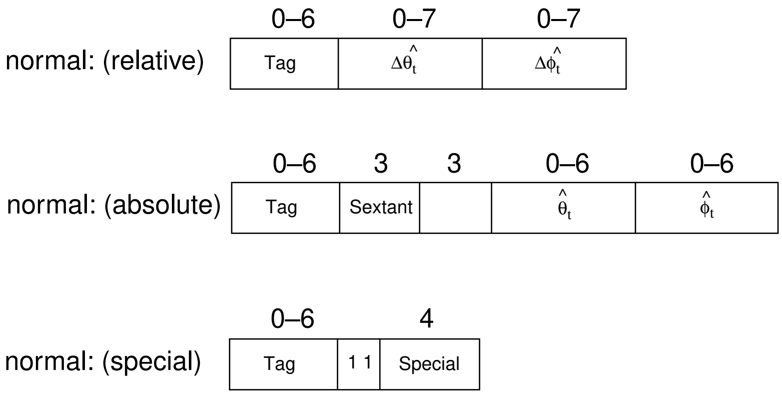
As usual, the first six bits of the subcommand are actually forwarded ahead of the rest of the command. Depending on the length of the tag and delta fields, the first six bits might only contain the tag, or the tag and some of the R field bits, or any subset up to the entire subcommand, if short enough.

For clarity, because it is by far the most typical case, the coordinate bit-fields are labeled ΔR ΔG ΔB (Δα), though more properly they are R, G, and B fields; their actual interpretation is absolute or relative depending on the setting of that bit in the decompression table entry corresponding to the tag field. In both cases the fields are signed two's-complement numbers.

If the most recent setting of the `cap` bit by a `setState` command is zero, then no fourth (alpha) field will be expected, and must not be present. If the `cap` bit was set, then the alpha field will be processed and must be present.

The rest of the graphics pipeline and frame buffer following the geometry decompression stage may choose not to use all (up to) 16 bits of color component information; in this case it is acceptable to truncate the trailing bits during decompression. What the geometry decompression format does require is that color setting of any size up to 16 bits be supported, even if all the bits are not used.

B.12.7 Normal Subcommand



The normal subcommand can appear within either a geometry decompression `vertex` command or `normal` command. The tag field can be between 0 and 6 bits in length; the last two delta (or absolute) fields will have the same length, between 1 and 7 bits. Six more bits are always present for absolute normals. The range of sizes for a relative normal can be from 6 to 20 bits, and an absolute normal can be from 6 to 24 bits. (As usual, any subcommand with a total length of less than 6 bits has trailing zeros added to pad the length to a minimum of 6 bits.)

As usual, the first six bits of the subcommand are actually forwarded ahead of the rest of the command. Depending on the length of the tag and delta fields, the first six bits might only contain the tag, or the tag and some of the other field bits, or any subset up to the entire subcommand, if short enough.

A `normal` subcommand is interpreted as relative or absolute depending on the current setting of that bit in the decompression table entry corresponding to the tag field. Unlike the `position` and `color` subcommands, the number of fields of a `normal` command differ between the absolute and relative types.

When the subcommand is relative, there are two delta angle fields after the tag field, both of the same length, up to seven bits. These two fields are signed two's-complement numbers. If after delta addition the resulting angle is outside the current sextant or octant, the sextant/octant wrapping rules (described elsewhere) apply.

When the subcommand is absolute, four bit fields follow the tag. The first is a three-bit (fixed-length) absolute sextant field, indicating in which of six sextants of an octant of the unit sphere this normal resides. The second field is also fixed at three bits, and indicates in which octant of the unit sphere the normal resides. The last two fields are absolute angles within the sextant, and are unsigned positive numbers, up to six bits in length.

Fourteen special absolute normals are encoded by the unused two settings within the three sextant bits. This is indicated by specifying the angle fields to have a length of zero (not present), the first two bits of the sextant field to both have a value of 1, and the trailing bit after the octant field to have a value of 0. Table B-1 lists the 14 special normals

Table B-1 The 14 Special Normals

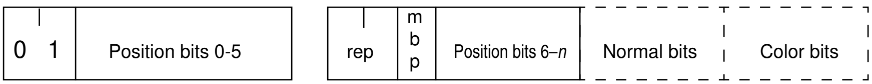
Special	NX	NY	NZ	Comment
0000	1.0	0.0	0.0	+X axis
0010	–1.0	0.0	0.0	–X axis
0100	0.0	1.0	0.0	+Y axis
0110	0.0	–1.0	0.0	–Y axis
1000	0.0	0.0	1.0	+Z axis
1010	0.0	0.0	–1.0	–Z axis
0001	$1/\sqrt{3}$	$1/\sqrt{3}$	$1/\sqrt{3}$	+X +Y +Z
0011	$1/\sqrt{3}$	$1/\sqrt{3}$	$1/\sqrt{3}$	+X +Y –Z
0101	$1/\sqrt{3}$	$1/\sqrt{3}$	$1/\sqrt{3}$	+X –Y +Z

Table B-1 The 14 Special Normals (Continued)

Special	NX	NY	NZ	Comment
0111	$1/\sqrt{3}$	$1/\sqrt{3}$	$1/\sqrt{3}$	+X –Y –Z
1001	$1/\sqrt{3}$	$1/\sqrt{3}$	$1/\sqrt{3}$	–X +Y +Z
1011	$1/\sqrt{3}$	$1/\sqrt{3}$	$1/\sqrt{3}$	–X +Y –Z
1101	$1/\sqrt{3}$	$1/\sqrt{3}$	$1/\sqrt{3}$	–X –Y +Z
1111	$1/\sqrt{3}$	$1/\sqrt{3}$	$1/\sqrt{3}$	–X –Y –Z

The rest of the graphics pipeline and frame buffer following the geometry decompression stage may choose not to use all (up to) 16 bits of normal component information; in this case it is acceptable to truncate the trailing bits during decompression. What the geometry decompression format does require is that normal settings of any size up to 18-bit absolute normals be supported, even if all the decompressed bits are not used.

B.12.8 vertex



The `vertex` command has a two-bit opcode, a `position` subcommand (always), a two-bit vertex replacement field, a mesh buffer push bit, and, optionally, a `normal` subcommand and/or a `color` command, depending on the current setting of the state bundling bits. The two-bit vertex replacement field has the same triangle semantics as it does within the `meshBufferReference` command:

0 0	Restart clockwise
0 1	Restart counterclockwise
1 0	Replace middle
1 1	Replace oldest

The mesh buffer push bit indicates whether this vertex should be pushed into the mesh buffer so as to be eligible for later re-reference.

The `position`, `normal`, and `color` subcommands have the semantics documented in their individual sections.

B.12.9 normal

1	1	Normal bits 0–5	Normal bits 6–*n*

The `normal` command has a two-bit opcode, and a `normal` subcommand.

The `normal` subcommand has the semantics documented in Section B.12.7, “Normal Subcommand.”

If a `normal` command is present immediately before a `meshBufferReference` command, then the new normal value overrides the normal data present in the mesh buffer for that particular mesh buffer reference.

B.12.10 color

1	0	Color bits 0–5	Color bits 6–*n*

The `color` command has a two-bit opcode, and a `color` subcommand. The `color` subcommand semantics are documented in Section B.12.6, “Color Subcommand.”

If a `color` command is present immediately before a `meshBufferReference` command, then the new color value overrides the color data present in the mesh buffer for that particular mesh buffer reference.

B.13 Semantics of Geometry Decompression Commands

The formal semantics of the compression format is best described by a state description of the decompression process. It must be emphasized that these state descriptions are given to show the formal semantics, not an efficient implementation.

The next few sections will present such a state description. While this description is intended to be a complete and unambiguous description of the compression format and decompression semantics, in practice studying both the compression process and the decompression process, and studying code examples for both, is a better approach for the human understanding process.

B.13.1 Header and Body to Variable-Length Command

Geometry decompression commands have a minimum length of eight bits (six bits for subcommands). This allows all geometry decompression commands to be split into two physically separate bit sequences within the compressed stream. The first bit sequence is always of eight bits in length (six for subcommands), the second bit sequence contains the remaining bits of the decompression command (if any). Thus a logical stream of *N* geometry decompression commands, where each command is split into two bit sequences H_i and B_i (i being from 0 to $N - 1$) is physically represented as:

$$H_0\ B_{-1}\ H_1\ B_0\ H_2\ B_1\ \ldots\ H_{n-1}\ B_{n-2}\ H_n\ B_{n-1}$$

OK, so what is this "B_{-1}"? All compressed geometry sequences have an implied (not physically present) H_{-1} of a variable-length no-op opcode, thus B_{-1} is always present (starting at the eighth bit of the stream) as any valid variable-length no-op body. (Just five zeros, the minimum-length no-op, is a good default.) Thus the implied no-op opcode "jump starts" the header-forwarded decompression process. This process is reversed at the end of the stream. H_n is a variable-length no-op opcode, but no body is present, as B_{n-1} is the last bits of the stream.

This is viable because all compressed geometry streams are presented along with a total bit length of their contents, so no explicit end-of-stream marker is needed. Streams *must* be rounded up to the nearest full 64-bit word multiple by use of additional variable length no-ops (within the body of the stream, that is, their headers appear before H_n).

This "header-forwarding" shuffled representation is necessary for hardware decompressors to operate efficiently. While this is not an issue for purely software-based decompressor implementations, in order to have one canonical format for both hard and soft decompressors, all decompressors must operate only on the header-forwarded representation; this is the only "official" compression bit-format specified. For a software decompressor, the extra unshuffling adds only slightly to the overall overhead of decompression; for hardware, it is essential.

Thus the first stage in the decompression process is to put the two separate bit sequences for each command back together. The next paragraph describes the flavor of this process, going around the loop approximately one and one-half times. The actual process is more accurately described in state machine semantics.

First the fixed-length eight- (or six-) bit header for the next full command (or subcommand) to be processed is detached from the current head of the compressed stream. Next, the variable-length body bits for the previous command (or subcommand) are detached from the compressed stream and combined with the already extracted header for the previous command; the previous command is now complete and can be processed. Now the fixed-length header for the command after the next is detached from the bit stream, and then finally the variable-length body for the next full command can be detached; the next command is now complete and can be processed.

```
// pseudocode for converting bitstream into sequences of
// commands
decompress(stream) {
    previous_header <- no-op
    while (not_empty(stream)) {
        current_header <- get_8_bits(stream)
        previous_body <- get_n_bits(stream,
                             body_length(previous_header))
        process_command(previous_header, previous_body)
        previous_header <- current_header
    }
}
```

One slight complexity: the `get_8_bits()` only extracts six bits of header for the `color` or `normal` subcommand of a `vertex` command. It extracts a full eight bits of header in all other cases.

B.13.2 Variable-Length Command to Command

The three decompression tables contain entries for each different numeric tag describing whether the value in the stream is absolute or relative, and length and shift constants describing how to convert the variable-length bit field back into a fixed-length value. The fixed-length value for position and color components is 16 bits in length (sign, unit, 14 fraction); the fields for normal angles are 7 bits (signed), and 3 each for sextant and octant (if present).

B.13.3 Delta Position to Position

```
absolute_position(x, y, z):
   cur_x ← x, cur_y ← y, cur_z ← z
```

```
relative_position(Δx, Δy, Δz):
   cur_x ← cur_x + Δx, cur_y ← cur_y + Δy, cur_z ← cur_z + Δz
```

B.13.4 Delta Color to Color

```
absolute_color(r, g, b {, α}):
   cur_r ← r, cur_g ← g, cur_b ← b, {cur_α ← α }

relative_color(Δr, Δg, Δb {, Δα}):
   cur_r ← cur_r + Δr, cur_g ← cur_g + Δg, cur_b ← cur_b + Δb,
   {cur_α ← cur_α + Δα }
```

B.13.5 Encoded Delta Normal to Encoded Normal

State: `cur_oct`, `cur_sex`, `cur_u`, `cur_v`

```
absolute_normal(oct, sex, u, v):
   cur_oct ← oct, cur_sex ← sex, cur_u ← u, cur_v ← v,

relative_normal(Δu, Δv):
```

```
   cur_u ← cur_u + Δu, cur_v ← cur_v + Δv,
   if (cur_u < 0)
      cur_u ← -cur_u, cur_sex ← flip_u[cur_sex]
   else if (cur_v < 0)
      cur_v ← -cur_v, cur_oct ← cur_oct <xor> flip_v[cur_sex]
   else if (cur_u + cur_v > 64)
      cur_u ← 64 - cur_u, cur_v ← 64 - cur_v,
      cur_sex ← flip_uv[cur_sex]

   flip_u[6]  = { 4, 5, 3, 2, 0, 1 }
   flip_v[6]  = { 2, 4, 1, 1, 2, 4 }
   flip_uv[6] = { 2, 3, 0, 1, 5, 4 }
```

B.13.6 Encoded Normal to Rectilinear Normal

```
   nx ← norms[v,u].nx,    ny ← norms[v,u].ny,    nz ←
   norms[v,u].nz,
   if (cur_sex & 4) t ← nx, nx ← nz, nz ← t
   if (cur_sex & 2) t ← ny, ny ← nz, nz ← t
   if (cur_sex & 1) t ← nx, nx ← ny, ny ← t
   if (cur_oct & 1) nz ← -nz
```

```
if (cur_oct & 2) ny ← -ny
if (cur_oct & 4) nx ← -nx
```

The contents of the `norms[]` table is exactly specified, and the next revision of this specification will contain an exact listing of the values.

B.14 Semantics of Vertices

The formal semantics of the vertex processing is best described by a state description of the decompression process. Once again it must be emphasized that these state descriptions are given to show the formal semantics, not an efficient implementation.

B.14.1 Command to Vertex

This section describes the state change semantics caused by each command to generate the next output vertex, prior to assembly into triangles. The internal state consists of the six mode bits, a current normal and current color, `normal_override` and `color_override` bits, the 16 mesh buffer vertices, and the current mesh index.

```
normal(n):
   current_normal ← n, normal_override ← 1

color(c):
   current_color ← c, color_override ← 1

vertex(rep, mbp, p {, n} {, c}):
```

```
   current_position ← p,
   if (bnv) current_normal ← n,
   if (bcv) current_color ← c,
   output_vertex(rep, current_position, current_normal, current_
   color)
   if (mbp) mesh_buffer[oldest_mesh_index].position ← p
   if (mbp && bnv) mesh_buffer[mesh_index].normal ← n
   if (mbp && bcv) mesh_buffer[mesh_index].color ← c
   if (mbp) mesh_index ← (mesh_index+1) & 15
   normal_override ← 0, color_override ← 0
```

```
mesh buffer reference(rep, i):
```

```
current_position ←
          mesh_buffer[(mesh_index - i - 1) & 15].position
if (bnv && !normal_override)
   current_normal ←mesh_buffer[(mesh_index - i - 1) & 15].normal
if (bcv && !color_override)
   current_color ← mesh_buffer[(mesh_index - i - 1) & 15].color
output_vertex(rep, current_position, current_normal, current_
color)
```

```
set state(new_bnv, new_bcv, new_cap, new_tex):
```

```
bnv ← new_bnv,
bcv ← new_bcv,
cap ← new_cap,
tex ← new_tex
```

```
set table(address, range, entry):
        ...

passthrough(data):
        (null)

vnop(length):
        (null)
```

B.14.2 Vertex to Intermediate Triangle

This section describes the formal semantics of assembling vertices with replacement commands into nearly finished triangles: the semantics of generalized triangle strips.

```
output_vertex(restart clockwise, newv):
   newest ← newv, number_of_vertices ← 1, ccw = 0

output_vertex(restart counterclockwise, newv):
   newest ← newv, number_of_vertices ← 1, ccw = 1
```

`output_vertex(replace_middle, newv):`

```
if (number_of_vertices < 2)
   midlest ← newest, newest ← newv, number_of_vertices++
else if (number_of_vertices < 3)
   oldest ← midlest, midlest ← newest, newest ← newv,
   number_of_vertices++,
   intermediate_triangle(ccw, oldest, midlest, newest)
else if (number_of_vertices == 3)
   midlest ← newest, newest ← newv,
   intermediate_triangle(ccw, oldest, midlest, newest)
```

`output_vertex(replace_oldest, newv):`

```
if (number_of_vertices < 2)
   midlest ← newest, newest ← newv, number_of_vertices++
else if (number_of_vertices < 3)
   oldest ← midlest, midlest ← newest, newest ← newv,
   number_of_vertices++,
   intermediate_triangle(ccw, oldest, midlest, newest)
else if (number_of_vertices == 3)
   oldest ← midlest, midlest ← newest, newest ← newv,
   ccw = 1 - ccw,
   intermediate_triangle(ccw, oldest, midlest, newest)
```

B.14.3 Intermediate Triangle to Final Triangle

The final stage is to take into account the current `rnt` and `rct` mode bits settings. These control the semantics of the normal and color vertex data. The semantics of the counterclockwise bit also can be expressed here; thus, the final triangles can always be assumed to be front facing when their vertices appear in counterclockwise order.

`intermediate_triangle(ccw, v1, v2, v3):`

```
if (ccw)
    final_triangle(v1.position, v1.normal, v1.color,
                   v2.position, v2.normal, v2.color,
                   v3.position, v3.normal, v3.color)
else if (!ccw)
    final_triangle(v2.position, v2.normal, v2.color,
                   v1.position, v1.normal, v1.color,
                   v3.position, v3.normal, v3.color)
```

B.15 Outline of Geometry Process

Java 3D only formally defines the geometry compression format and the decompression semantics. Authoring tools are free to employ whatever geometry compression algorithms they choose, as long as the results adhere to the specifications described in the previous sections.

However, to further document the semantics of the geometry compression format, an overview of one particular geometry compression algorithm is given here.

B.15.1 Compressing Geometry Data

Group the geometry to be compressed into separate rigid objects. Typically such objects will be individually culled during rendering, so you should not join objects too extensively prior to compression. In optimized systems, the granularity of object splitting will be computed by an algorithm that takes culling optimization into account.

B.15.2 Convert to Generalized Mesh Format

Once a group of geometry has been identified, it is next converted into generalized mesh format. This is a complex step, and a number of topological analysis-based algorithms have been applied to it.

The next step is the quantization of the geometry positions, colors, normals, and/or texture map coordinates. All these quantizations can be varied within the geometry, but for simplicity a single fixed quantization of each is assumed here.

B.15.3 Position

Normalize the position data.

The containing bounding box for the object is computed. This is the minimal box such that all geometry vertices are contained within it. The vertices are then all normalized to be contained within this bounding box by first subtracting the XYZ location of the bounding box center from the vertex XYZ and then dividing all the XYZ vertex values by the half length of the longest side of the bounding box. Thus all normalized positions will be within the ±1 unit cube. A constant matrix transform corresponding to an offset to the center of the bounding box, and an inverse scale by the half length of the longest side of the bounding box are created as a prologue for the geometry data.

Quantize the position data.

Assuming that position data is to be quantized to n bits, each vertex position component should be multiplied by the value of 2^n and then rounded to the nearest integer.

B.15.4 Normals

Normalize the normals.

Each normal should be normalized to unit length.

Quantize the XYZ components of the normal to 14 bits accuracy

Each normal component should be multiplied by 2^{14}, rounded to the nearest integer, and then converted back to floating-point representation and divided by 2^{14}.

Quantize the XYZ components of the normal to 14 bits accuracy

If an XYZ component of the normal is negative, invert it and save the original sign bits as a three-bit octant value:

```
oct = 0;
if(nx < 0.0) oct |= 4, nx = -nx
if(ny < 0.0) oct |= 2, ny = -ny
if(nz < 0.0) oct |= 1, nz = -nz
```

Fold the normal to the nX > nZ > nY sextant

Check (in exactly the following order):

```
sex = 0;
if (nx < ny) t = nx, nx = ny, ny = t, sex |= 1
if (nz < ny) t = ny, ny = nz, nz = t, sex |= 2
if (nx < nz) t = nx, nx = nz, nz = t, sex |= 4
```

Match the nearest quantized normal representation

Take the dot product of the normal with each of the quantized reference normals in the table for the specified number of quantized normal bits. That UV normal index for the reference normal that gives the greatest (nearest unity) dot product result is the new quantized normal representation (along with the octant and sextant representation).

Check for special normals

B.15.5 Colors

The colors are assumed to be in a 0.0 to 1.0 representation to begin with.

Quantize the color values.

Assuming that color data is to be quantized to n bits, each vertex color component (R, G, B, and optionally α) should be multiplied by the value of 2^n and then rounded to the nearest integer.

Texture map coordinates may appear in place of color components within the compression stream, as controlled by the `tex` state bit. If 2D texture mapping is desired, then the UV texture coordinate values take the place of the RG color components in the compression stream (B is not present).

B.15.6 Collect Delta Code Statistics

Make a pass in generalized mesh order through all vertices in the geometry. For each successive pair of vertices, compute the difference between their component values, compute the bit length of this (signed) difference, and histogram this bit length. Specifics for each component type are detailed in the next sections.

B.15.7 Position Delta Code Statistics

Compute ΔX, ΔY, and ΔZ. Determine which of these has the greatest magnitude. Compute the number of bits for this component, including one sign bit. This is the length to be histogrammed for positions.

B.15.8 Color Delta Code Statistics

Compute ΔR, ΔG, ΔB, and $\Delta\alpha$ (if present). Determine which of these has the greatest magnitude. Compute the number of bits for this component, including one sign bit. This is the length to be histogrammed for colors.

B.15.9 Normal Delta Code Statistics

For a given pair of normals, check to see if they have the same octant and sextant. If so, compute ΔU and ΔV. Determine which of these has the greatest magnitude. Compute the number of bits for this component, including one sign bit. This is the length to be histogrammed for this pair of normals.

If the normals have different sextants and/or octants, check to see if their sextants share an edge. Depending on what type of edge they share, the delta including the change in edges is encoded in one of three ways: $U + \Delta U < 0$, $V + \Delta V < 0$, and $U + \Delta U + V + \Delta V > 64$. Each case is discussed in the paragraphs below. The sextant numbers are from the binary codes shown in Figure B-3.

Sextants 0 and 4, 1 and 5, and 2 and 3 share the U = 0 edge. When crossing this boundary, ΔU becomes ~U – `last_u`. This will generate a negative `cur_u` value during decompression, which causes the decompressor to invert `cur_u` and look up the new sextant in a table.

Sextants 0 and 2, 1 and 3, and 4 and 5 share the U + V = 64 edge. ΔU becomes 64 – U – `last_u` and ΔV becomes 64 – V – `last_v`. When `cur_u` + `cur_v` > 64, the decompressor sets `cur_u` = 64 – `cur_u` and `cur_v` = 64 – `cur_v`, and a table lookup determines the new sextant.

Each sextant shares the V = 0 edge with its corresponding sextant in another octant. When in sextants 1 or 5, the normal moves across the X-axis, across the Y-axis for sextants 0 or 4, and across the Z-axis for sextants 2 or 3. ΔV becomes ~V – `last_v`. The decompressor inverts a negative `cur_v` and performs a table lookup for a mask to exclusive-OR with the current octant value.

Otherwise the normals cannot be delta encoded, and so the second (target) normal must be represented by an absolute reference to its three octant, three sextant, and 2 N-bit U V addresses. This is the length to be histogrammed for this pair of normals.

B.15.10 Assign Huffman Tags

Encode data into variable-bit length decompression commands.

One can use an algorithm similar to the one used by the JPEG image compression standard. The main differences are how codes are reassigned when their lengths exceed the maximum code length and how the data bits are encoded in the compressed data stream.

The frequencies of the data lengths are used as leaf nodes in a binary tree. The algorithm used to generate the tree places the less frequent codes deeper in the tree. After the tree is built, the traversal path to a leaf node becomes its Huffman code, and the depth in the tree becomes its code length.

Codes generated with a length greater than six, the maximum code length, must be shortened. These nodes are merged with more frequent nodes by increasing the number of sign bits included with the smaller data length.

B.15.11 Assemble the Pieces into a Bit Stream

Given the sequence of variable-bit-length decompression commands, shuffle the first eight (six) bits of each command ahead of its predecessor's body.

APPENDIX C

View Model Details

AN application programmer writing a 3D graphics program that will deploy on a variety of platforms must anticipate the likely end-user environments and must carefully construct the view transformations to match those characteristics using a low-level API. This appendix addresses many of the issues an application must face and describes the sophisticated features that Java 3D's advanced view model provides.

C.1 An Overview of the Java 3D View Model

Both camera-based and Java 3D–based view models allow a programmer to specify the shape of a view frustum and, under program control, to place, move, and re-orient that frustum within the virtual environment. However, how they do this varies enormously. Unlike the camera-based system, the Java 3D view model allows slaving the view frustum's position and orientation to that of a six-degrees-of-freedom tracking device. By slaving the frustum to the tracker, Java 3D can automatically modify the view frustum so that the generated images match the end-user's viewpoint exactly.

Java 3D must handle two rather different head-tracking situations. In one case, we rigidly attach a tracker's *base*, and thus its coordinate frame, to the display environment. This corresponds to placing a tracker base in a fixed position and orientation relative to a projection screen within a room, relative to a computer display on a desk, or relative to the walls of a multiple-wall projection display. In the second head-tracking situation, we rigidly attach a tracker's *sensor*, not its base, to the display device. This corresponds to rigidly attaching one of that tracker's sensors to a head-mounted display and placing the tracker base somewhere within the physical environment.

C.2 Physical Environments and Their Effects

Imagine an application where the end user sits on a magic carpet. The application flies the user through the virtual environment by controlling the carpet's location and orientation within the virtual world. At first glance, it might seem that the application also controls what the end user will see—and it does, but only superficially.

The following two examples show how end-user environments can significantly affect how an application must construct viewing transformations.

C.2.1 A Head-mounted Example

Imagine that the end user sees the magic carpet and the virtual world with a head-mounted display and head tracker. As the application flies the carpet through the virtual world, the user may turn to look to the left, right, or even toward the rear of the carpet. Because the head tracker keeps the renderer informed of the user's gaze direction, it might not need to draw the scene directly in front of the magic carpet. The view that the renderer draws on the head-mount's display must match what the end user would see had the experience occurred in the real world.

C.2.2 A Room-mounted Example

Imagine a slightly different scenario, where the end user sits in a darkened room in front of a large projection screen. The application still controls the carpet's flight path; however, the position and orientation of the user's head barely influences the image drawn on the projection screen. If a user looks left or right, then he or she only sees the darkened room. The screen does not move. It's as if the screen represents the magic carpet's "front window" and the darkened room represents the "dark interior" of the carpet.

By adding a left and right screen, we give the magic carpet rider a more complete view of the virtual world surrounding the carpet. Now our end user sees the view to the left or right of the magic carpet by turning left or right.

C.2.3 Impact of Head Position and Orientation on the Camera

In the head-mounted example, the user's head position and orientation significantly affects a camera model's camera position and orientation but has hardly any effect on the projection matrix. In the room-mounted example, the user's head position and orientation contributes little to a camera model's camera position and orientation; however, it does affect the projection matrix.

From a camera-based perspective, the application developer must construct the camera's position and orientation by combining the virtual-world component (the position and orientation of the magic carpet) and the physical-world component (the user's instantaneous head position and orientation).

Java 3D's view model incorporates the appropriate abstractions to compensate automatically for such variability in end-user hardware environments.

C.3 The Coordinate Systems

The basic view model consists of eight or nine coordinate systems, depending on whether the end-user environment consists of a room-mounted display or a head-mounted display. First we define the coordinate systems used in a room-mounted display environment. Next we define the added coordinate system introduced when using a head-mounted display system.

C.3.1 Room-mounted Coordinate Systems

The room-mounted coordinate system is divided into the virtual coordinate system and the physical coordinate system. Figure C-1 shows these coordinate systems graphically. The coordinate systems within the grayed area exist in the virtual world; those outside exist in the physical world. Note that the coexistence coordinate system exists in both worlds.

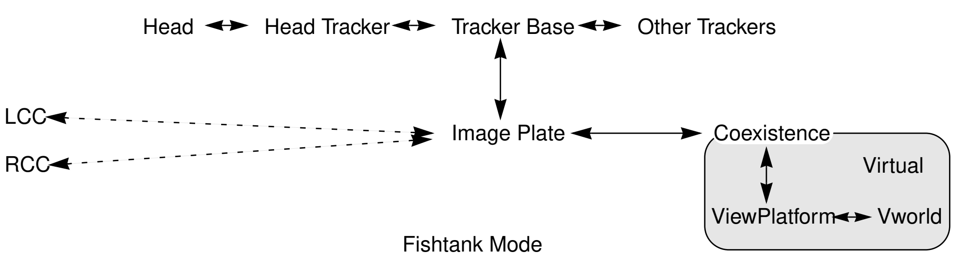


Figure C-1 Display Rigidly Attached to the Tracker Base

C.3.1.1 The Virtual Coordinate Systems

The Virtual World Coordinate System

The virtual world coordinate system encapsulates the unified coordinate system for all scene graph objects in the virtual environment. For a given View, the virtual world coordinate system is defined by the Locale object that contains the ViewPlatform object attached to the View. It is a right-handed coordinate system with +*x* to the right, +*y* up, and +*z* toward the viewer.

The ViewPlatform Coordinate System

The ViewPlatform coordinate system is the local coordinate system of the ViewPlatform leaf node to which the View is attached.

The Coexistence Coordinate System

A primary implicit goal of any view model is to map a specified local portion of the physical world onto a specified portion of the virtual world. Once established, one can legitimately ask where the user's head or hand is located within the virtual world, or where a virtual object is located in the local physical world. In this way the physical user can interact with objects inhabiting the virtual world, and vice versa. To establish this mapping, Java 3D defines a special coordinate system, called *coexistence* coordinates, that is defined to exist in *both* the physical world and the virtual world.

The coexistence coordinate system exists half in the virtual world and half in the physical world. The two transforms that go from the coexistence coordinate system to the virtual world coordinate system and back again contain all the information needed to expand or shrink the virtual world relative to the physical world, as well as the information needed to position and orient the virtual world relative to the physical world.

Modifying the transform that maps the coexistence coordinate system into the virtual world coordinate system changes what the end user can see. The Java 3D application programmer moves the end user within the virtual world by modifying this transform.

C.3.1.2 The Physical Coordinate Systems

The Head Coordinate System

The head coordinate system allows an application to import its user's head geometry. The coordinate system provides a simple consistent coordinate frame for specifying such factors as the location of the eyes and ears.

The Image Plate Coordinate System

The image plate coordinate system corresponds with the physical coordinate system of the image generator. The image plate is defined as having its origin at the lower left-hand corner of the display area and as lying in the display area's *XY* plane. Note that image plate is a different coordinate system than either left image plate or right image plate. These last two coordinate systems are defined in head-mounted environments only (see Section C.3.2, "Head-mounted Coordinate Systems").

The Head Tracker Coordinate System

The head tracker coordinate system corresponds to the six-degrees-of-freedom tracker's sensor attached to the user's head. The head tracker's coordinate system describes the user's instantaneous head position.

The Tracker Base Coordinate System

The tracker base coordinate system corresponds to the emitter associated with absolute position/orientation trackers. For those trackers that generate relative position/orientation information, this coordinate system is that tracker's initial position and orientation. In general, this coordinate system is rigidly attached to the physical world.

C.3.2 Head-mounted Coordinate Systems

Head-mounted coordinate systems divide the same virtual coordinate systems and the physical coordinate systems. Figure C-2 shows these coordinate systems graphically. As with the room-mounted coordinate systems, the coordinate systems within the grayed area exist in the virtual world; those outside exist in the physical world. Once again, the coexistence coordinate system exists in both worlds. The arrangement of the coordinate system differs from those for a room-mounted display environment. The head-mounted version of Java 3D's coordinate system differs in another way. It includes two image plate coordinate systems, one for each of an end-user's eyes.

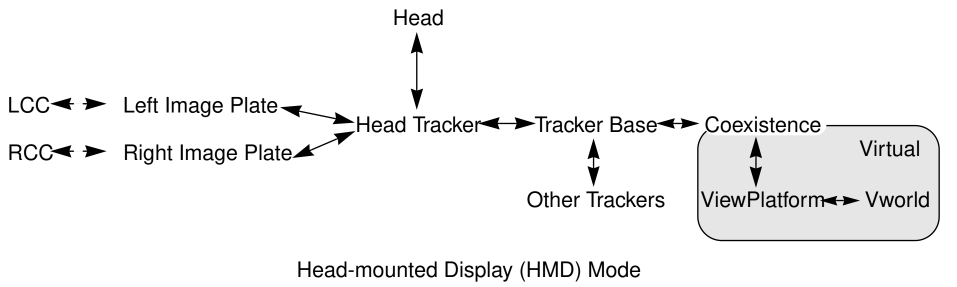


Figure C-2 Display Rigidly Attached to the Head Tracker (Sensor)

The Left Image Plate and Right Image Plate Coordinate Systems

The left image plate and right image plate coordinate systems correspond with the physical coordinate system of the image generator associated with the left and right eye, respectively. The image plate is defined as having its origin at the lower left-hand corner of the display area and lying in the display area's *XY* plane. Note that the left image plate's *XY* plane does not necessarily lie parallel to the right image plate's *XY* plane. Note that left image plate and right image plate are different coordinate systems than the room-mounted display environment's image plate coordinate system.

C.4 The ViewPlatform Object

The ViewPlatform object is a leaf object within the Java 3D scene graph. The ViewPlatform object is the only portion of Java 3D's viewing model that resides as a node within the scene graph. Changes to TransformGroup nodes in the scene graph hierarchy above a particular ViewPlatform object move the view's location and orientation within the virtual world (see Section 8.4, "ViewPlatform: A Place in the Virtual World"). The ViewPlatform object also contains a ViewAttachPolicy and an ActivationRadius (see Section 5.10, "ViewPlatform Node," for a complete description of the ViewPlatform API).

C.5 The View Object

The View object is the central Java 3D object for coordinating all aspects of a viewing situation. All parameters that determine the viewing transformation to be used in rendering on a collected set of canvases in Java 3D are either directly contained within the View object, or within objects pointed to by a View object (or pointed to by these, etc.). Java 3D supports multiple simultaneously active View objects, each of which controls its own set of canvases.

The Java 3D View object has several instance variables and methods, but most are calibration variables or user-helping functions.

Methods

`public final void setTrackingEnable(boolean flag)`
`public final boolean getTrackingEnable()`

These methods set and retrieve a flag specifying whether to enable the use of six-degrees-of-freedom tracking hardware.

`public final void getUserHeadToVworld(Transform3D t)`

This method retrieves the user-head-to-vworld coordinate system transform. This Transform3D object takes points in the user's head coordinate system and transforms them into points in the virtual world coordinate system. This value is read-only. Java 3D continually generates it, but only if enabled by using the `setUserHeadToVworldEnable` method.

`public final void setUserHeadToVworldEnable(boolean flag)`
`public final boolean getUserHeadToVworldEnable()`

These methods set and retrieve a flag that specifies whether or not to repeatedly generate the user-head-to-vworld transform (initially `false`).

`public String toString()`

This method returns a string that contains the values of this View object.

C.5.1 View Policy

The view policy informs Java 3D whether it should generate the view using the head-tracked system of transformations or the head-mounted system of transformations. These policies are attached to the Java 3D View object.

Methods

`public final void setViewPolicy(int policy)`
`public final int getViewPolicy()`

These two methods set and retrieve the current policy for view computation. The `policy` variable specifies how Java 3D uses its transforms in computing new viewpoints, as follows:

- SCREEN_VIEW: Specifies that Java 3D should compute new viewpoints using the sequence of transforms appropriate to nonattached, screen-based head-tracked display environments, such as fishtank VR, multiple-projection walls, and VR desks. This is the default setting.
- HMD_VIEW: Specifies that Java 3D should compute new viewpoints using the sequence of transforms appropriate to head-mounted display environments. This policy is not available in compatibility mode (see Section C.11, "Compatibility Mode").

C.5.2 Screen Scale Policy

The screen scale policy specifies where the screen scale comes from when the view attach policy is `NOMINAL_SCREEN_SCALED` (see Section 8.4.3, "View Attach Policy"). The policy can be one of the following:

- SCALE_EXPLICIT: Specifies that the scale used for a view attach policy of `NOMINAL_SCREEN_SCALED` is taken from the user-provided `nominalScreenScale` variable.
- SCALE_SCREEN_SIZE: Specifies that the scale used for a view attach policy of `NOMINAL_SCREEN_SCALED` is derived from the physical screen. This is the default policy.

```
public final void setScreenScalePolicy(int policy)
public final int getScreenScalePolicy()
```

These methods set and retrieve the current screen scale policy.

```
public final void setScreenScale(double scale)
public final double getScreenScale()
```

These methods set and retrieve the screen scale value. This value is used when the view attach policy is `NOMINAL_SCREEN_SCALED` and the screen scale policy is `SCALE_EXPLICIT`.

C.5.3 Window Eyepoint Policy

The window eyepoint policy comes into effect in a non-head-tracked environment. The policy tells Java 3D how to construct a new view frustum based on changes in the field of view and in the Canvas3D's location on the screen. The policy only comes into effect when the application changes a parameter that can change the placement of the eyepoint relative to the view frustum.

Constants

```
public static final int RELATIVE_TO_FIELD_OF_VIEW
```

This variable tells Java 3D that it should modify the eyepoint position so it is located at the appropriate place relative to the window to match the specified field of view. This implies that the view frustum will change whenever the application changes the field of view. In this mode, the eye position is read-only. This is the default setting.

```
public static final int RELATIVE_TO_SCREEN
```

This variable tells Java 3D to interpret the eye's position relative to the entire screen. No matter where an end user moves a window (a Canvas3D), Java 3D continues to interpret the eye's position relative to the screen. This implies that the view frustum changes shape whenever an end user moves the location of a window on the screen. In this mode, the field of view is read-only.

```
public static final int RELATIVE_TO_WINDOW
```

This variable specifies that Java 3D should interpret the eye's position information relative to the window (Canvas3D). No matter where an end user moves a window (a Canvas3D), Java 3D continues to interpret the eye's position relative to that window. This implies that the frustum remains the same no matter where the end user moves the window on the screen. In this mode, the field of view is read-only.

Methods

```
public final int getWindowEyepointPolicy()
public final void setWindowEyepointPolicy(int policy)
```

This variable specifies how Java 3D handles the predefined eyepoint in a non-head-tracked application. The variable can contain one of three values: `RELATIVE_TO_FIELD_OF_VIEW`, `RELATIVE_TO_SCREEN`, or `RELATIVE_TO_WINDOW`. The default value is `RELATIVE_TO_FIELD_OF_VIEW`.

C.5.4 Monoscopic View Policy

This policy specifies how Java 3D generates a monoscopic view.

Constants

```
public final static int LEFT_EYE_VIEW
public final static int RIGHT_EYE_VIEW
public final static int CYCLOPEAN_EYE_VIEW
```

These constants specify the monoscopic view policy. The first constant specifies that the monoscopic view should be the view as seen from the left eye. The second constant specifies that the monoscopic view should be the view as seen from the right eye. The third constant specifies that the monoscopic view should be the view as seen from the "center eye," the fictional eye half-way between the left and right eyes. This is the default setting.

Methods

```
public final void setMonoscopicViewPolicy(int policy)
public final int getMonoscopicViewPolicy()
```

These methods set and return the monoscopic view policy, respectively.

C.5.5 Sensors and Their Location in the Virtual World

```
public final void getSensorToVworld(Sensor sensor, Transform3D t)
public final void getSensorHotSpotInVworld(Sensor sensor,
       Point3d position)
public final void getSensorHotSpotInVworld(Sensor sensor,
       Point3f position)
```

The first method takes the sensor's last reading and generates a sensor-to-vworld coordinate system transform. This Transform3D object takes points in that sensor's local coordinate system and transforms them into virtual world coordinates. The next two methods retrieve the specified sensor's last hotspot location in virtual world coordinates.

C.6 The Screen3D Object

A Screen3D object represents one independent display device. The most common environment for a Java 3D application is a desktop computer with or without a head tracker. Figure C-3 shows a scene graph fragment for a display environment designed for such an end-user environment. Figure C-4 shows a display environment that matches the scene graph fragment in Figure C-3.

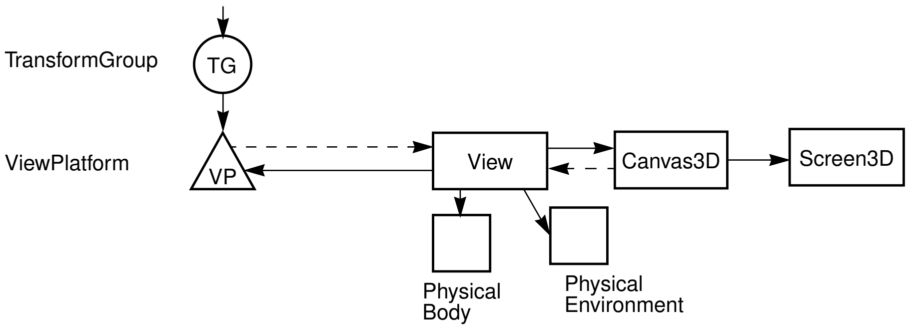


Figure C-3 A Portion of a Scene Graph Containing a Single Screen3D Object

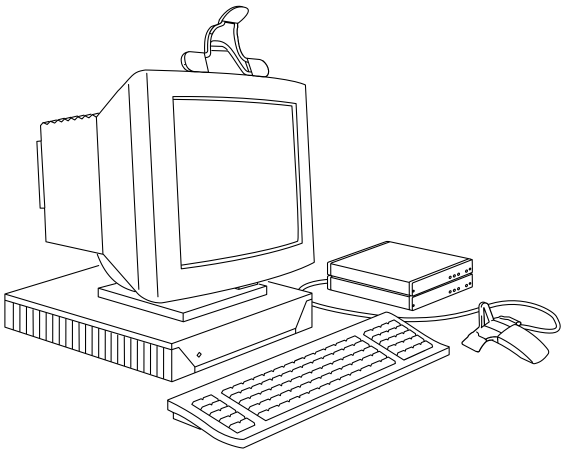


Figure C-4 A Single-Screen Display Environment

A multiple-projection wall display presents a more exotic environment. Such environments have multiple screens, typically three or more. Figure C-5 shows a scene graph fragment representing such a system and Figure C-6 shows the corresponding display environment.

A multiple-screen environment requires more care during the initialization and calibration phase. Java 3D must know how the Screen3D's are placed with respect to one another, the tracking device, and the physical portion of the coexistence coordinate system.

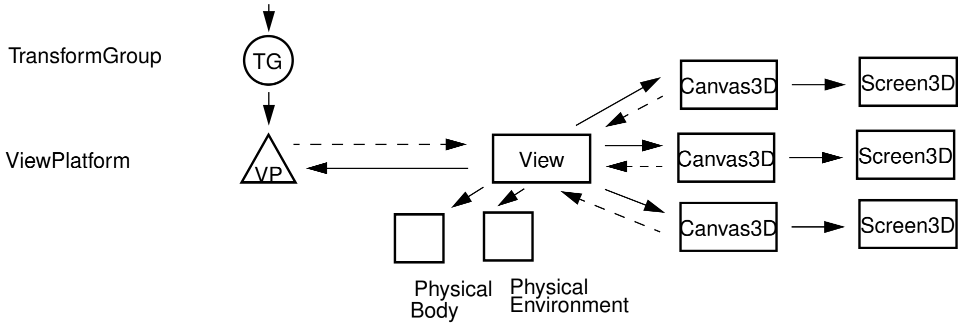


Figure C-5 A Portion of a Scene Graph Containing Three Screen3D Objects

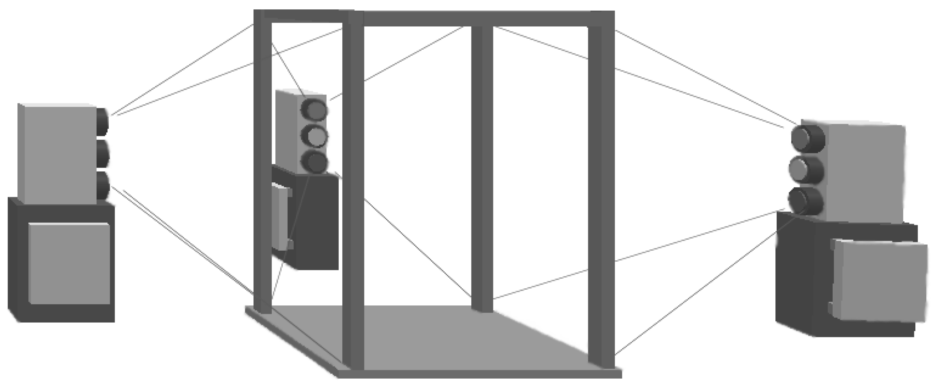


Figure C-6 A Three-Screen Display Environment

C.6.1 Screen3D Calibration Parameters

The Screen3D object is the 3D version of AWT's screen object (see Section 8.8, "The Screen3D Object"). To use a Java 3D system, someone or some program must calibrate the Screen3D object with the coexistence volume. These methods allow that person or program to inform Java 3D of those calibration parameters.

Measured Parameters

These calibration parameters are set once, typically by a browser, calibration program, system administrator, or system calibrator, not by an applet.

```
public final void setPhysicalScreenWidth(double width)
public final void setPhysicalScreenHeight(double height)
```

These methods store the screen's (image plate's) physical width and height in meters. The system administrator or system calibrator must provide these values by measuring the display's active image width and height. In the case of a head-mounted display, this should be the display's apparent width and height at the focal plane.

C.6.2 Accessing and Changing Head Tracker Coordinates

```
public void setTrackerBaseToImagePlate(Transform3D t)
public void getTrackerBaseToImagePlate(Transform3D t)
```

These methods set and get the tracker-base-to-image-plate coordinate system transform. If head tracking is enabled, this transform is a calibration constant. If head tracking is not enabled, this transform is not used. This is used only in `SCREEN_VIEW` mode. Users must recalibrate whenever the image plate moves relative to the tracker.

```
public void setHeadTrackerToLeftImagePlate(Transform3D t)
public void getHeadTrackerToLeftImagePlate(Transform3D t)
public void setHeadTrackerToRightImagePlate(Transform3D t)
public void getHeadTrackerToRightImagePlate(Transform3D t)
```

These methods set and get the head-tracker-to-left-image-plate and head-tracker-to-right-image-plate coordinate system transforms, respectively. If head tracking is enabled, these transforms are calibration constants. If head tracking is not enabled, these transforms are not used. They are used only in `HMD_VIEW` mode.

C.7 The Canvas3D Object

Java 3D provides special support for those applications that wish to manipulate an eye position even in a non-head-tracked display environment. One situation where such a facility proves useful is an application that wishes to generate a very high-resolution image composed of lower-resolution tiled images. The application must generate each tiled component of the final image from a common eye position with respect to the composite image but a different eye position from the perspective of each individual tiled element.

C.7.1 Scene Antialiasing

`public final boolean getSceneAntialiasingAvailable()`

This method returns a status flag indicating whether scene antialiasing is available.

C.7.2 Accessing and Modifying an Eye's Image Plate Position

A Canvas3D object provides sophisticated applications with access to the eye's position information in head-tracked, room-mounted runtime environments. It also allows applications to manipulate the position of an eye relative to an image plate in non-head-tracked runtime environments.

```
public final void getLeftEyeInImagePlate(Point3d position)
public final void getRightEyeInImagePlate(Point3d position)
public final void getCenterEyeInImagePlate(Point3d position)
```

These values determine eye placement when a head tracker is not in use and the application is directly controlling the eye position in image plate coordinates. In head-tracked mode or when the `windowEyepointPolicy` is `RELATIVE_TO_FIELD_OF_VIEW`, this value is derived from other values and is read-only. In head-tracked mode, Java 3D repetitively generates these values as a function of the current head position. The center eye is the fictional eye half-way between the left and right eye.

```
public final void getPixelLocationInImagePlate(int x, int y,
       Point3d position)
```

This method computes the position of the specified AWT pixel value in image plate coordinates and copies that value into the object provided.

```
public final void setLeftManualEyeInImagePlate(Point3d position)
public final void setRightManualEyeInImagePlate(Point3d position)
public final void getLeftManualEyeInImagePlate(Point3d position)
public final void getRightManualEyeInImagePlate(Point3d position)
```

These methods set and retrieve the position of the manual left and right eyes in image plate coordinates. These values determine eye placement when a head tracker is not in use and the application is directly controlling the eye position in image plate coordinates. In head-tracked mode or when the `windowEyepointPolicy` is `RELATIVE_TO_FIELD_OF_VIEW`, this value is ignored. When the `windowEyepointPolicy` is `RELATIVE_TO_WINDOW`, only the *Z* value is used.

```
public final void getVworldToImagePlate(Transform3D t)
```

This method retrieves the current virtual-world-to-image-plate coordinate system transform and places it into the specified object.

```
public final void getImagePlateToVworld(Transform3D t)
```

This method retrieves the current image-plate-to-virtual-world coordinate system transform and places it into the specified object.

C.7.3 Canvas Width and Height

```
public final double getPhysicalWidth()
public final double getPhysicalHeight()
```

These methods retrieve the physical width and height of this canvas window, in meters.

C.8 The PhysicalBody Object

The PhysicalBody object contains information concerning the physical characteristics of the end-user's body. The head parameters allow end users to specify their own head's characteristics and thus to customize any Java 3D application so that it conforms to their unique geometry. The PhysicalBody object defines head parameters in the head coordinate system. It provides a simple and consistent coordinate frame for specifying such factors as the location of the eyes and thus the interpupilary distance.

The Head Coordinate System

The head coordinate system has its origin on the head's bilateral plane of symmetry, roughly half-way between the left and right eyes. The origin of the head coordinate system is known as the *center eye*. The positive *X*-axis extends to the right. The positive *Y*-axis extends up. The positive *Z*-axis extends into the skull. Values are in meters.

Constructors

```
public PhysicalBody()
```

Constructs a default user PhysicalBody object with the following default eye and ear positions:

Left eye: –0.033, 0.0, 0.0

Right eye: 0.033, 0.0, 0.0
Left ear: –0.080, –0.030, 0.095
Right ear: 0.080, –0.030, 0.095

```
public PhysicalBody(Point3d leftEyePosition,
       Point3d rightEyePosition)
public PhysicalBody(Point3d leftEyePosition,
       Point3d rightEyePosition, Point3d leftEarPosition,
       Point3d rightEarPosition)
```

These methods construct a PhysicalBody object with the specified eye and ear positions.

Methods

```
public void getLeftEyePosition(Point3d position)
public void setLeftEyePosition(Point3d position)
public void getRightEyePosition(Point3d position)
public void setRightEyePosition(Point3d position)
```

These methods set and retrieve the position of the center of rotation of a user's left and right eyes in head coordinates.

```
public void getLeftEarPosition(Point3d position)
public void setLeftEarPosition(Point3d position)
public void getRightEarPosition(Point3d position)
public void setRightEarPosition(Point3d position)
```

These methods set and retrieve the position of the user's left and right ear positions in head coordinates.

```
public double getNominalEyeHeightFromGround()
public void setNominalEyeHeightFromGround(double height)
```

These methods set and retrieve the user's nominal eye height as measured from the ground to the center eye in the default posture. In a standard computer monitor environment, the default posture would be seated. In a multiple-projection display room environment or a head-tracked environment, the default posture would be standing.

```
public double getNominalEyeOffsetFromNominalScreen()
public void setNominalEyeOffsetFromNominalScreen(double offset)
```

These methods set and retrieve the offset from the center eye to the center of the display screen. This offset distance allows an "over the shoulder" view of the scene as seen by the end user.

`public void setHeadToHeadTracker(Transform3D t)`
`public void getHeadToHeadTracker(Transform t)`

These methods set and retrieve the head-to-head-tracker coordinate system transform. If head tracking is enabled, this transform is a calibration constant. If head tracking is not enabled, this transform is not used. This transform is used in both `SCREEN_VIEW` and `HMD_VIEW` modes.

`public String toString()`

This method returns a string that contains the values of this PhysicalBody object.

C.9 The PhysicalEnvironment Object

The PhysicalEnvironment object contains information about the local physical world of the end-user's physical environment. This includes information about audio output devices and tracking sensor hardware, if present.

Constructors

`public PhysicalEnvironment()`
`public PhysicalEnvironment(int sensorCount)`

These constructors construct and initialize a new PhysicalEnvironment object. The first constructor constructs a new PhysicalEnvironment object with the following default sensor and audio fields, and an array of `sensorCount` sensor objects.

Audio device count: 1
Audio devices: null
Input sensor count: 10
Tracking available: false
Input sensors: null

The sensor information provides real-time access to continuous-input devices such as joysticks and trackers. It also contains two-degrees-of-freedom joystick and six-degrees-of-freedom tracker information. See Section 10.2, "Sensors," for more information. Java 3D uses Java AWT's event model for noncontinuous input devices such as keyboards (see Chapter 10, "Input Devices and Picking").

Audio device information associated with the PhysicalEnvironment object allows the application a mechanism to choose a particular audio device (if more than one is available) and explicitly set the type of audio playback for sound rendered using this device. See Chapter 11, "Audio Devices," for more details on the fields and methods that set and initialize the device driver and output playback associated with the audio device.

Methods

The PhysicalEnvironment object specifies the following methods pertaining to audio output devices and input sensors.

```
public void setAudioDevice(AudioDevice device)
```

This method selects the specified AudioDevice object as the device through which audio rendering for this PhysicalEnvironment will be performed.

```
public AudioDevice getAudioDevice()
```

This retrieves the specified AudioDevice object.

```
public void setSensorCount(int count)
public int getSensorCount()
```

These methods set and retrieve the count of the number of sensors stored within the PhysicalEnvironment object. It defaults to a small number of sensors. It should be set to the number of sensors available in the end-user's environment before initializing the Java 3D API.

```
public void setCoexistenceToTrackerBase(Transform3D t)
public void getCoexistenceToTrackerBase(Transform3D t)
```

These methods set the coexistence-to-tracker-base coordinate system transform. If head tracking is enabled, this transform is a calibration constant. If head tracking is not enabled, this transform is not used. This is used in both `SCREEN_VIEW` and `HMD_VIEW` modes.

```
public boolean getTrackingAvailable()
```

This method returns a status flag indicating whether or not tracking is available.

```
public void setSensor(int index, Sensor sensor)
public Sensor getSensor(int index)
```

The first method sets the sensor specified by the index to the sensor provided. The second method retrieves the specified sensor.

```
public void setDominantHandIndex(int index)
public int getDominantHandIndex()
```

These methods set and retrieve the index of the dominant hand.

```
public void setNonDominantHandIndex(int index)
public int getNonDominantHandIndex()
```

These methods set and retrieve the index of the nondominant hand.

```
public void setHeadIndex(int index)
public int getHeadIndex()
public void setRightHandIndex(int index)
public int getRightHandIndex()
public void setLeftHandIndex(int index)
public int getLeftHandIndex()
```

These methods set and retrieve the index of the head, right hand, and left hand. The `index` parameter refers to the sensor index.

Physical Coexistence Policy

```
public int getCoexistenceCenterInPworldPolicy()
public void setCoexistenceCenterInPworldPolicy(int policy)
```

These methods set and retrieve the physical coexistence policy used in this physical environment. This policy specifies how Java 3D will place the user's eyepoint as a function of current head position during the calibration process. Java 3D permits one of three values: `NOMINAL_HEAD`, `NOMINAL_FEET`, or `NOMINAL_SCREEN`. Note: `NOMINAL_SCREEN_SCALED` is not allowed for this policy.

C.10 Viewing in Head-tracked Environments

Section 8.5, "Generating a View," describes how Java 3D generates a view for a standard flat-screen display with no head tracking. In this section, we describe how Java 3D generates a view in a room-mounted, head-tracked display environment—either a computer monitor with shutter glasses and head tracking or a multiple-wall display with head-tracked shutter glasses. Finally, we describe how Java 3D generates view matrices in a head-mounted and head-tracked display environment.

C.10.1 A Room-mounted Display with Head Tracking

When head tracking combines with a room-mounted display environment (for example, a standard flat screen display), the ViewPlatform's origin and orientation serves as a base for constructing the view matrices. Additionally, Java 3D uses the end-user's head position and orientation to compute where an end-user's eyes are located in physical space. Each eye's position serves to offset the corresponding virtual eye's position relative to the ViewPlatform's origin. Each eye's position also serves to specify that eye's frustum since the eye's position relative to a Screen3D uniquely specifies that eye's view frustum. Note that Java 3D will access the PhysicalBody object to obtain information describing the user's interpupilary distance and tracking hardware, values it needs to compute the end-user's eye positions from the head position information.

C.10.2 A Head-mounted Display with Head Tracking

In a head-mounted environment, the ViewPlatform's origin and orientation also serves as a base for constructing view matrices. And, as in the head-tracked, room-mounted environment, Java 3D also uses the end-user's head position and orientation to further modify the ViewPlatform's position and orientation. In a head-tracked, head-mounted display environment, an end-user's eyes do not move relative to their respective display screens, rather, the display screens move relative to the virtual environment. A rotation of the head by an end user can radically affect the final view's orientation. In this situation, Java 3D combines the position and orientation from the ViewPlatform with the position and orientation from the head tracker to form the view matrix. The view frustum, however, does not change since the user's eyes do not move relative to their respective display screen, so Java 3D can compute the projection matrix once and cache the result.

If any of the parameters of a View object are updated, this will effect a change in the implicit viewing transform (and thus image) of any Canvas3D that references that View object.

C.11 Compatibility Mode

A camera-based view model allows application programmers to think about the images displayed on the computer screen as if a virtual camera took those images. Such a view model allows application programmers to position and orient a virtual camera within a virtual scene, to manipulate some parameters of the virtual camera's lens (specify its field of view), and to specify the locations of the near and far clipping planes.

Java 3D allows applications to enable compatibility mode for room-mounted, non-head-tracked display environments, or to disable compatibility mode using the following methods. Camera-based viewing functions are only available in compatibility mode.

Methods

```
public final void setCompatibilityModeEnable(boolean flag)
public final boolean getCompatabilityModeEnable()
```

This flag turns compatibility mode on or off. Compatibility mode is disabled by default.

Note: Use of these view-compatibility functions will disable some of Java 3D's view model features and limit the portability of Java 3D programs. These methods are primarily intended to help jump-start porting of existing applications.

C.11.1 Overview of the Camera-based View Model

The traditional camera-based view model, shown in Figure C-7, places a virtual camera inside a geometrically specified world. The camera "captures" the view from its current location, orientation, and perspective. The visualization system then draws that view on the user's display device. The application controls the view by moving the virtual camera to a new location, by changing its orientation, by changing its field of view, or by controlling some other camera parameter.

The various parameters that users control in a camera-based view model specify the shape of a viewing volume (known as a frustum because of its truncated pyramidal shape) and locate that frustum within the virtual environment. The rendering pipeline uses the frustum to decide which objects to draw on the display screen. The rendering pipeline does not draw objects outside the view frustum and it clips (partially draws) objects that intersect the frustum's boundaries.

Though a view frustum's specification may have many items in common with those of a physical camera, such as placement, orientation, and lens settings, some frustum parameters have no physical analog. Most noticeably, a frustum has two parameters not found on a physical camera: the near and far clipping planes.

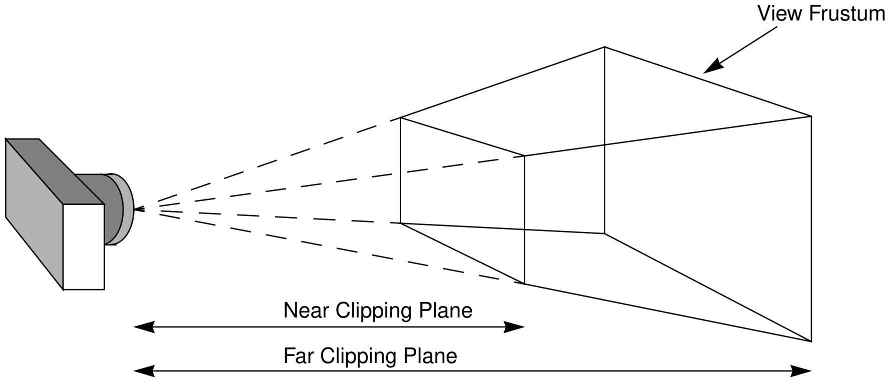


Figure C-7 The Camera-based View Model

The location of the near and far clipping planes allow the application programmer to specify which objects Java 3D should not draw. Objects too far away from the current eyepoint usually do not result in interesting images. Those too close to the eyepoint might obscure the interesting objects. By carefully specifying near and far clipping planes, an application programmer can control which objects the renderer will not be drawing.

From the perspective of the display device, the virtual camera's image plane corresponds to the display screen. The camera's placement, orientation, and field of view determine the shape of the view frustum.

C.11.2 Using the Camera-based View Model

The camera-based view model allows Java 3D to bridge the gap between existing 3D code and Java 3D's view model. By using the camera-based view model methods, a programmer retains the familiarity of the older view model but gains some of the flexibility afforded by Java 3D's new view model.

The traditional camera-based view model is supported in Java 3D by helping methods in the Transform3D object. These methods were explicitly designed to resemble as closely as possible the view functions of older packages, and thus should be familiar to most 3D programmers. The resulting Transform3D objects can be used to set compatibility-mode transforms in the View object.

C.11.2.1 Creating a Viewing Matrix

The Transform3D object provides the following method to create a viewing matrix.

```
public void lookAt(Point3d eye, Point3d center, Vector3d up)
```

This is a utility method that specifies the position and orientation of a viewing transform. It works very similarly to the equivalent function in OpenGL. The inverse of this transform can be used to control the ViewPlatform object within the scene graph. Alternatively, this transform can be passed directly to the View's `VpcToEc` transform via the compatibility-mode viewing functions (see Section C.11.2.3, "Setting the Viewing Transform").

C.11.2.2 Creating a Projection Matrix

The Transform3D object provides the following three methods for creating a projection matrix. All three map points from eye coordinates (EC) to clipping coordinates (CC). Eye coordinates are defined such that (0, 0, 0) is at the eye and the projection plane is at $z = -1$.

```
public void frustum(double left, double right, double bottom,
       double top, double near, double far)
```

The `frustum` method establishes a perspective projection with the eye at the apex of a symmetric view frustum. The transform maps points from eye coordinates to clipping coordinates. The clipping coordinates generated by the resulting transform are in a right-handed coordinate system (as are all other coordinate systems in Java 3D).

The arguments define the frustum and its associated perspective projection: `(left`, `bottom`, `-near)` and `(right`, `top`, `-near)` specify the point on the near clipping plane that maps onto the lower-left and upper-right corners of the window, respectively. The `-far` parameter specifies the far clipping plane. See Figure C-8.

```
public void perspective(double fovx, double aspect, double zNear,
       double zFar)
```

The `perspective` method establishes a perspective projection with the eye at the apex of a symmetric view frustum, centered about the *Z*-axis, with a fixed field of view. The resulting perspective projection transform mimics a standard camera-based view model. The transform maps points from eye coordinates to clipping coordinates. The clipping coordinates generated by the resulting transform are in a right-handed coordinate system.

The arguments define the frustum and its associated perspective projection: `-near` and `-far` specify the near and far clipping planes; `fovx` specifies the field of view in the *X* dimension, in radians; and `aspect` specifies the aspect ratio of the window. See Figure C-9.

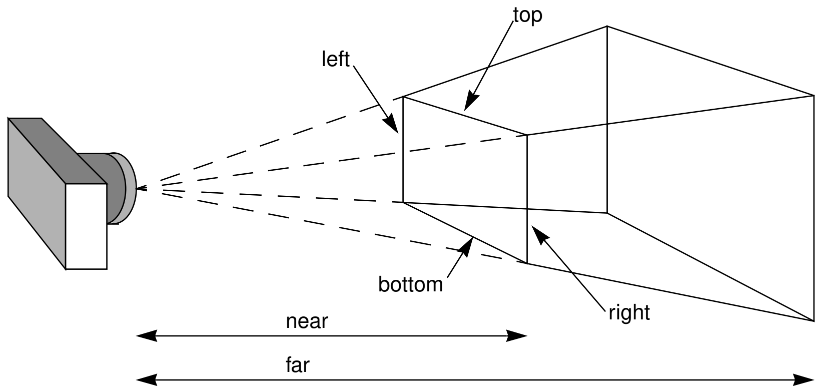


Figure C-8 A Perspective Viewing Frustum

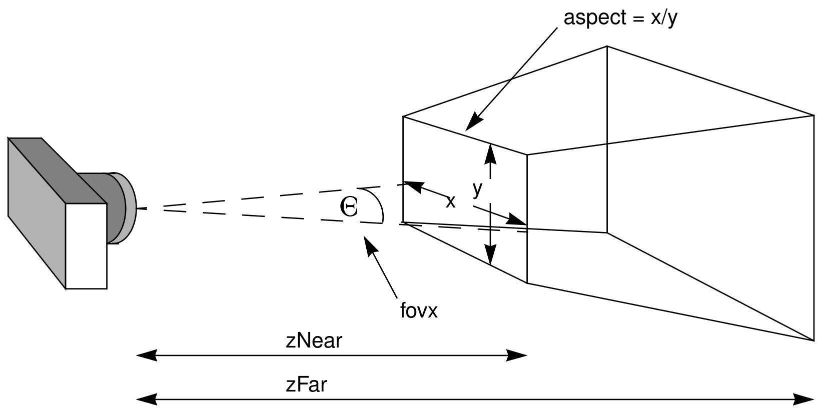


Figure C-9 Perspective View Model Arguments

```
public void ortho(double left, double right, double bottom,
       double top, double near, double far)
```

The `ortho` method establishes a parallel projection. The orthographic projection transform mimics a standard camera-based video model. The transform maps points from eye coordinates to clipping coordinates. The clipping coordinates generated by the resulting transform are in a right-handed coordinate system.

The arguments define a rectangular box used for projection: `(left`, `bottom`, `-near)` and `(right`, `top`, `-near)` specify the point on the near clipping plane that maps onto the lower-left and upper-right corners of the window, respectively. The `-far` parameter specifies the far clipping plane. See Figure C-10.

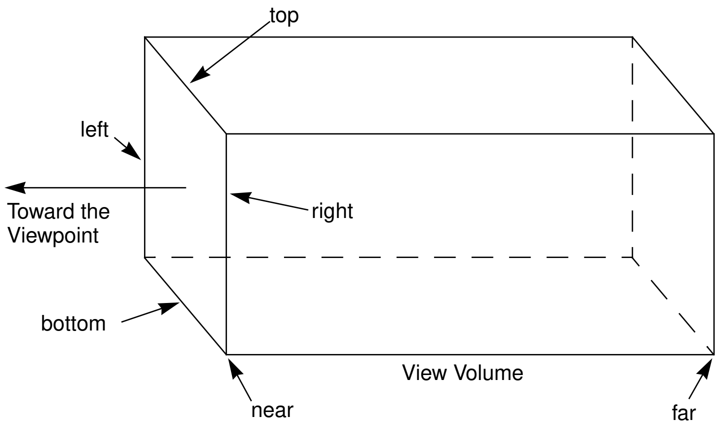


Figure C-10 Orthographic View Model

C.11.2.3 Setting the Viewing Transform

The View object provides the following compatibility-mode methods that operate on the viewing transform.

```
public final void setVpcToEc(Transform3D vpcToEc)
public final void getVpcToEc(Transform3D vpcToEc)
```

This compatibility-mode method specifies the ViewPlatform coordinates (VPC) to eye coordinates viewing transform. If compatibility mode is disabled, this transform is derived from other values and is read-only.

C.11.2.4 Setting the Projection Transform

The View object provides the following compatibility-mode methods that operate on the projection transform.

```
public final void setLeftProjection(Transform3D projection)
public final void getLeftProjection(Transform3D projection)
public final void setRightProjection(Transform3D projection)
public final void getRightProjection(Transform3D projection)
```

These compatibility-mode methods specify a viewing frustum for the left and right eye that transforms points in eye coordinates to clipping coordinates. If compatibility mode is disabled, a `RestrictedAccessException` is thrown. In monoscopic mode, only the left eye projection matrix is used.

APPENDIX D

Exceptions

THE Java 3D API uses the standard Java exception model for handling errors or exceptional conditions. In addition to using existing exception classes, such as `ArrayIndexOutOfBoundsException` and `IllegalArgumentException`, Java 3D defines several new runtime exceptions. These exceptions are thrown by various Java 3D methods or by the Java 3D renderer to indicate an error condition of some kind.

The exceptions defined by Java 3D, as part of the `javax.media.j3d` package, are described in the following sections. They all extend `RuntimeException` and, as such, need not be declared in the `throws` clause of methods that might cause the exception to be thrown. This appendix is not an exhaustive list of all exceptions expected for Java 3D. Additional exceptions will be added as the need arises.

D.1 BadTransformException

Indicates an attempt to use a Tranform3D object that is inappropriate for the object in which it is being used. For example:

- Transforms that are used in the scene graph, within a TransformGroup node, must be affine. They may optionally contain a nonuniform scale or a shear, subject to other listed restrictions.
- All transforms in the TransformGroup nodes above a ViewPlatform object must be congruent. This ensures that the Vworld-coordinates-to-ViewPlatform-coordinates transform is angle- and length-preserving with no shear and only uniform scale.
- Most viewing transforms other than those in the scene graph can only contain translation and rotation.

- The projection transform is allowed to be non-affine, but it must either be a single-point perspective projection or a parallel projection.

Constructors

```
public BadTransformException()
public BadTransformException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

D.2 CapabilityNotSetException

This exception indicates an access to a live or compiled Scene Graph object without the required capability set.

Constructors

```
public CapabilityNotSetException()
public CapabilityNotSetException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

D.3 DanglingReferenceException

This exception indicates that during a `cloneTree` call, an updated reference was requested for a node that did not get cloned. This occurs when a subgraph is duplicated via `cloneTree` and has at least one leaf node that contains a reference to a node with no corresponding node in the cloned subgraph. This results in two leaf nodes wanting to share access to the same node.

If dangling references are to be allowed during the `cloneTree` call, `cloneTree` should be called with the `allowDanglingReferences` parameter set to `true`.

Constructors

```
public DanglingReferenceException()
public DanglingReferenceException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

D.4 IllegalRenderingStateException

This exception indicates an illegal state for rendering. This includes:

- Lighting without specifying normals in a geometry array object
- Texturing without specifying texture coordinates in a geometry array object

```
public illegalRenderingStateException()
public illegalRenderingStateException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

D.5 IllegalSharingException

This exception indicates an illegal attempt to share a scene graph object. For example, the following are illegal:

- Referencing a shared subgraph in more than one virtual universe
- Using the same component object both in the scene graph and in an immediate-mode graphics context
- Including an unsupported type of leaf node within a shared subgraph
- Referencing a BranchGroup node in more than one of the following ways:
    - Attaching it to a (single) Locale
    - Adding it as a child of a Group node within the scene graph
    - Referencing it from a (single) Background leaf node as background geometry

Constructors

```
public IllegalSharingException()
public IllegalSharingException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

D.6 MismatchedSizeException

This exception indicates that an operation cannot be completed properly because of a mismatch in the sizes of the object attributes.

```
public MismatchedSizeException()
public MismatchedSizeException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

D.7 MultipleParentException

This exception extends `IllegalSharingException` and indicates an attempt to add a node that is already a child of one group node into another group node.

Constructors

```
public MultipleParentException()
public MultipleParentException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

D.8 RestrictedAccessException

This exception indicates an attempt to access or modify a state variable without permission to do so. For example, invoking a `set` method for a state variable that is currently read-only.

Constructors

```
public RestrictedAccessException()
public RestrictedAccessException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

D.9 SceneGraphCycleException

This exception indicates that one of the live scene graphs attached to a viewable Locale has a cycle in it. Java 3D scene graphs are directed acyclic graphs and, as such, do not permit cycles. This exception is either thrown by the Java 3D renderer at scene graph traversal time or when a scene graph containing a cycle is made live (added as a descendant of a Locale object).

Constructors

```
public SceneGraphCycleException()
public SceneGraphCycleException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

D.10 SingularMatrixException

This exception, in the `javax.vecmath` package, indicates that the inverse of a matrix cannot be computed.

Constructors

```
public SingularMatrixException()
public SingularMatrixException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

D.11 SoundException

This exception indicates a problem in loading or playing a sound sample.

Constructors

```
public SoundException()
public SoundException(String str)
```

These create the exception object that outputs the exception message. The first form uses the default message. The second form specifies the message string to be output.

APPENDIX E

Equations

THIS appendix contains the Java 3D equations for fog, lighting, sound, and texture mapping. Many of the equations use the following symbols:

- $\cdot$ Multiplication
- $\bullet$ Function operator for sound equations, Dot product for all other equations

E.1 Fog Equations

The ideal fog equation is as follows:

$$C' = C \cdot f + Cf \cdot (1 - f) \qquad \text{(E.1)}$$

The fog coefficient, *f*, is computed differently for linear and exponential fog. The equation for linear fog is as follows:

$$f = \frac{B - z}{B - F} \qquad \text{(E.2)}$$

The equation for exponential fog is as follows:

$$f = \mathrm{e}^{-d \cdot z} \qquad \text{(E.3)}$$

The parameters used in the fog equations are as follows:

C	=	Color of the pixel being fogged
Cf	=	Fog color
d	=	Fog density
F	=	Front fog distance, measured in eye coordinates

B = Back fog distance, measured in eye coordinates

z = The z-coordinate distance from the eyepoint to the pixel being fogged, measured in eye coordinates

f = Fog coefficient

Fallbacks and Approximations

1. An implementation may approximate per-pixel fog by calculating the correct fogged color at each vertex and then linearly interpolating this color across the primitive.
2. An implementation may approximate exponential fog using linear fog by computing values of F and B that cause the resulting linear fog ramp to most closely match the effect of the specified exponential fog function.
3. An implementation will ideally perform the fog calculations in eye coordinates, which is an affine space. However, an implementation may approximate this by performing the fog calculations in a perspective space (such as, device coordinates). As with other approximations, the implementation should match the specified function as closely as possible.

E.2 Lighting Equations

The ideal lighting equation is as follows:

$$Me + Ma \cdot \sum_{i}^{Numamb} (Lc_i) + \sum_{i}^{Numlt} (atten_i \cdot spot_i \cdot (diff_i + spec_i)) \tag{E.4}$$

$$diff_i = (L_i \bullet N) \cdot Lc_i \cdot Md \tag{E.5}$$

$$spec_i = (S_i \bullet N)^{shin} \cdot Lc_i \cdot Ms \tag{E.6}$$

Note: If $(L_i \bullet N) \leq 0$, then $diff_i$ and $spec_i$ are set to 0.

$$atten_i = 1/(Kc_i + Kl_i \cdot d_i + Kq_i \cdot d_i^2) \tag{E.7}$$

Note: For directional lights, $atten_i$ is set to 1.

$$spot_i = \max((-L_i \cdot D_i), 0)^{exp_i} \tag{E.8}$$

Note: If the vertex is outside the spot light cone, as defined by the cutoff angle, $spot_i$ is set to 0. For directional and point lights, $spot_i$ is set to 1.

This is a subset of OpenGL in that the Java 3D ambient and directional lights are not attenuated and only ambient lights contribute to ambient lighting.

The parameters used in the lighting equation are as follows:

E = Eye vector
Ma = Material ambient color
Md = Material diffuse color
Me = Material emissive color
Ms = Material specular color
N = Vertex normal
$shin$ = Material shininess

The per-light values are as follows:

d_i = Distance from vertex to light
D_i = Spot light direction
exp_i = Spot light exponent
Kc_i = Constant attenuation
Kl_i = Linear attenuation
Kq_i = Quadratic attenuation
L_i = Direction from vertex to light
Lc_i = Light color
S_i = Specular half-vector = $\| (L_i + E) \|$

Fallbacks and Approximations

1. An implementation may approximate the specular function using a different power function that produces a similar specular highlight. For example, the PHIGS+ lighting model specifies that the reflection vector (the light vector reflected about the vertex normal) is dotted with the eye vector, and

that this dot product is raised to the specular power. An implementation that uses such a model should map the shininess into an exponent that most closely matches the effect produced by the ideal equation.

2. Implementations that do not have a separate ambient and diffuse color may fall back to using an ambient intensity as a percentage of the diffuse color. This ambient intensity should be calculated using the NTSC luminance equation:

$$I = 0.30 \cdot Red + 0.59 \cdot Green + 0.11 \cdot Blue \tag{E.9}$$

E.3 Sound Equations

There are different sets of sound equations, depending on whether the application uses headphones or speakers.

E.3.1 Headphone Playback Equations

For each sound source, Java 3D calculates a separate left and right output signal. Each left and right sound image includes differences in the *interaural intensity* and an *interaural delay*. The calculation results are a set of direct and indirect (delayed) sound signals mixed together before being sent to the audio playback system's left and right transducers.

E.3.1.1 Interaural Time Difference (Delay)

For each PointSound and ConeSound source, the left and right output signals are delayed based on the location of the sound and the orientation of the listener's head. The time difference between these two signals is called the *interaural time* difference (ITD). The time delay of a particular sound reaching an ear is affected by the arc the sound must travel around the listener's head. Java 3D uses an approximation of the ITD using a spherical head model. The interaural path difference is calculated based on the following cases:

1. The signal from the sound source to only one of the ears is direct. The ear farthest from the sound is shadowed by the listener's head ($\sin\alpha \geq De/2Dh$); see Figure E-1:

$$\begin{aligned} Ec &= |Vc| \\ Ef &= |Vt| + P \end{aligned} \tag{E.10}$$

where

$$P = \frac{De}{2}\left(\frac{\pi}{2} - (\gamma - \alpha)\right)$$

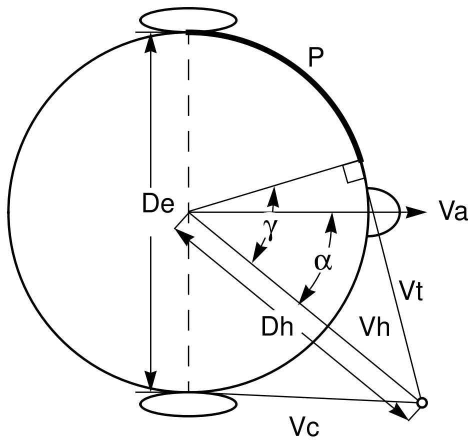


Figure E-1 Signal to Only One Ear Is Direct

2. The signals from the sound source reaches both ears by indirect paths around the head ($\sin\alpha < De/2Dh$); see Figure E-2:

$$\begin{aligned} Ec &= |Vt| + P' \\ Ef &= |Vt| + P \end{aligned} \tag{E.11}$$

where

$$P = \frac{De}{2}\left(\frac{\pi}{2} - (\gamma - \alpha)\right)$$

$$P' = \frac{De}{2}\left(\frac{\pi}{2} - (\gamma + \alpha)\right)$$

The time from the sound source to the closest ear is Ec/S, and the time from the sound source to the farthest ear is Ef/S, where S is the current AuralAttribute region's speed of sound.

If the sound is closest to the left ear, then

$$\begin{aligned} ITD_l &= Ec/S \\ ITD_r &= Ef/S \end{aligned} \tag{E.12}$$

If the sound is closest to the right ear, then

$$ITD_l = Ef / S$$
$$ITD_r = Ec / S \tag{E.13}$$

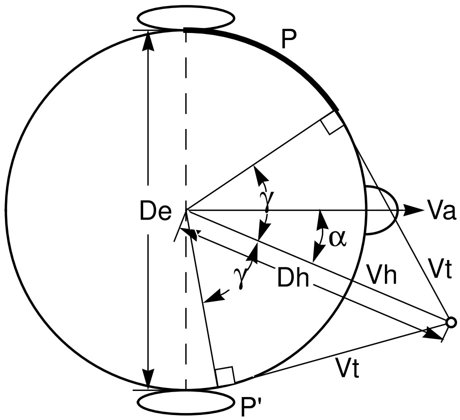


Figure E-2 Signals to Both Ears Are Indirect

The parameters used in the ITD equations are as follows:

α	=	The smaller of the angles between *Vh* (or –*Vh*) and *Va* in radians
γ	=	Angle between *Vh* and radius to tangent point on *Vt* in radians
De	=	Distance between ears (interaural distance)
Dh	=	Distance from interaural center to sound source
Ec	=	Distance from sound source to ear closest to sound
Ef	=	Distance from sound source to ear farthest from sound
P, *P*'	=	Arc path around the head an indirect signal must travel to reach an ear
S	=	Speed of sound for the current AuralAttribute region
Va	=	Vector from center ear forward parallel to *Z* axis of head coordinates
Vc	=	Vector from sound source to ear closest to sound
Vh	=	Vector from center ear to sound source
Vt	=	Vector from sound source to tangent point on the listener's head

E.3.1.2 Interaural Intensity (Gain) Difference

For each active and playing Point and ConeSound source, *i*, separate calculations for the left and right signal (based on which ear is closest and which is farthest to the source) are combined with nonspatialized BackgroundSound to create a stereo sound image. Each equation below is calculated separately for the left and right ear.

$$I(t) = \frac{\sum_{i}^{numS} [G_i \cdot (F_i \bullet [ITD_i \bullet Sample(t)])]}{maxNumS} \tag{E.14}$$

Note: For BackgroundSound sources ITD_i is an identity function so there is no delay applied to the sample for these sources.

$$G_i = Gi_i \cdot Gd_i \cdot Ga_i \cdot Gr_i \tag{E.15}$$

Note: For BackgroundSound sources $Gd_i = Ga_i = 1.0$. For PointSound sources $Ga_i = 1.0$.

$$F_i = Fd_i \bullet Fa_i \tag{E.16}$$

Note: For BackgroundSound sources Fd_i and Fa_i are identity functions. For PointSound sources Fa_i is an identity function.

If the sound source is on the right side of the head, *Ec* is used for left *G* and *F* calculations and *Ef* is used for right. Conversely, if the Sound source is on the left side of the head, *Ef* is used for left calculations and *Ec* is used for right.

Attenuation

For sound sources with a single distanceGain array defined, the intersection points of V_h (the vector from the sound source position through the listener's position) and the spheres (defined by the distanceGain array) are used to find the index *k* where $d_k \le L \le d_{k+1}$. See Figure E-3.

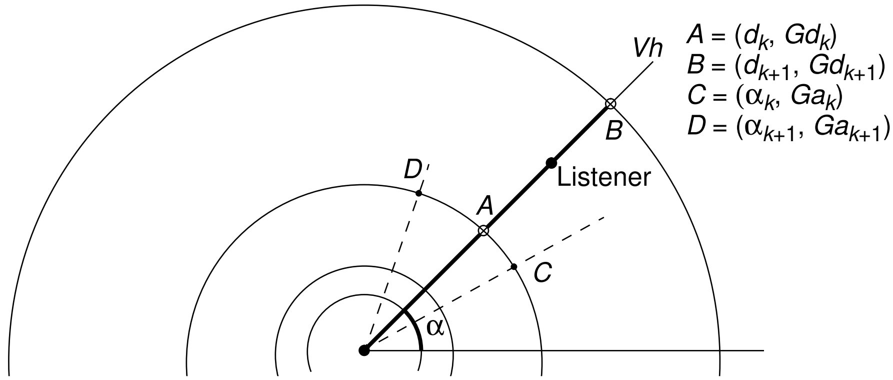


Figure E-3 ConeSound with a Single Distance Gain Attenuation Array

For ConeSound sources with two distanceGain arrays defined, the intersection points of V_h and the ellipsi (defined by both the front and back `distanceGain` arrays) closest to the listener's position are used to determine the index k. See Figure E-4.

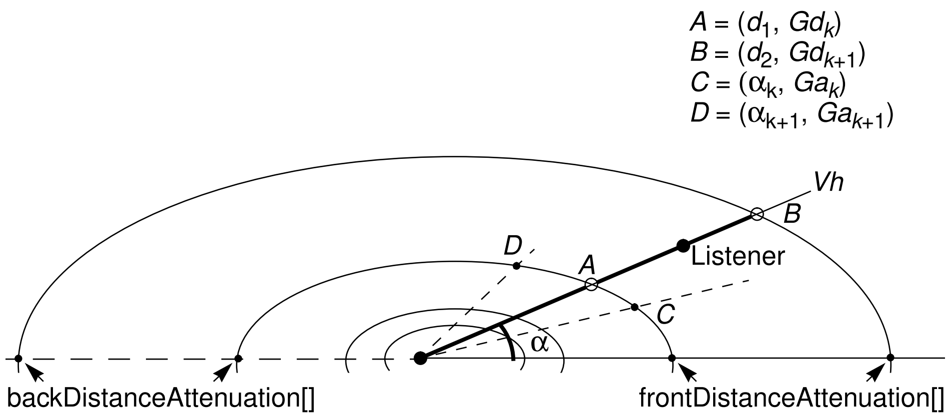


Figure E-4 ConeSound with Two Distance Attenuation Arrays

The equation for the distance gain is

$$Gd = Gd_k + \frac{(Gd_{k+1} - Gd_k) \cdot (d_2 - d_1)}{L - d_1} \tag{E.17}$$

Angular attenuation for both the spherical and elliptical cone sounds is identical. The angular distances in the attenuation array closest to α are found and define the index k into the angular attenuation array elements. The equation for the angular gain is

$$Ga = Ga_k + \frac{(Ga_{k+1} - Ga_k) \cdot (\alpha_{k+1} - \alpha_k)}{\alpha - \alpha_k} \tag{E.18}$$

Filtering

Similarly, the equations for calculating the AuralAttributes distance filter and the ConeSound angular attenuation frequency cutoff filter are

$$Fd = Fd_k + \frac{(Fd_{k+1} - Fd_k) \cdot (d_2 - d_1)}{L - d_1} \tag{E.19}$$

$$Fa = Fa_k + \frac{(Fa_{k+1} - Fa_k) \cdot (\alpha_{k+1} - \alpha_k)}{\alpha - \alpha_k} \tag{E.20}$$

An *N*-pole lowpass filter may be used to perform the simple angular and distance filtering defined in this version of Java 3D. These simple lowpass filters are meant only as an approximation for full, FIR filters (to be added in some future version of Java 3D).

Fallbacks and Approximations

1. If more than one lowpass filter is to be applied to the sound source (for example, both an angular filter and a distance filter are applied to a ConeSound source) it is only necessary to use a single filter, specifically the one that has the lowest cutoff frequency.
2. There is no requirement to support anything higher than very simple two-pole filtering. Any type of multipole lowpass filter can be used. If higher *N*-pole or compound filtering are available on the device on which sound rendering is being performed, use of these is encouraged, but not required.

The parameters used in the interaural intensity difference (IID) equations are as follows:

A, B	=	Triples containing DistanceGain linear distance, gain scale factor, and AuralAttribute cutoff frequency
C, D	=	Triples containing AngularAttenuation angular distance, gain scale factor, and cutoff frequency

α	=	Angle between *Vh* and *Va* in radians
Ec	=	Distance from sound source to ear closest to sound from the ITD equation
Ef	=	Distance from sound source to ear farthest from sound source from the ITD equation
Fa	=	Angular filter from ConeSound definition
Fd	=	Distance filter from AuralAttributes
Ga	=	Angular gain attenuation scale factor
Gd	=	Distance gain attenuation scale factor
Gi	=	Initial gain scale factor
Gr	=	Current AuralAttribute region's gain scale factor
I	=	Stereo sound image
L	=	Listener distance from sound source
maxNumS	=	Maximum number of sound sources for the audio device that the application is using for playback
numS	=	Number of sound sources
sample	=	Sound digital sample with a specific sample rate, bit precision, and an optional encoding and/or compression format
Vh	=	Vector from center ear to sound source

E.3.1.3 Doppler Effect Equations

The frequency of sound waves emanating from the source are lowered based on the speed of the source in relation to the listener, and the sound wave length, as follows:

$$S(f)' = S(f) - [Ds \cdot (Dv / W(f, Dh))] \tag{E.21}$$

The parameters used in the Doppler effect equations are as follows:

Dh	=	Distance from sound source to center ear
Ds	=	Doppler scale factor (AuralAttribute field)
Dv	=	Doppler velocity (between the listener and sound source)
f	=	Frequency
S	=	Sound source frequency

t = Time

W = Wavelength of sound source based on frequency and distance

E.3.1.4 Reverberation Equations

The overall reverberant sounds, used to give the impression of the aural space in which the active/enabled source sources are playing, is added to the stereo sound image output from equation equation E.14.

$$I'(t)_{[l,r]} = I(t)_{[l,r]} + \sum_{i}^{numS} R_i \tag{E.22}$$

Reverberation for each sound is approximated in the following:

$$R_i = \sum_{j}^{fLoop} [(Gr^j \cdot Sample(t)_i) \bullet D(t + (Tr \cdot j))] \tag{E.23}$$

Note that the reverberation calculation outputs the same image to both left and right output signals (thus there is a single monaural calculation for each sound reverberated). Correct first-order (early) reflections, based on the location of the sound source, the listener, and the active AuralAttribute's bounds, are not required for this version of Java 3D. Approximations based on the reverberation delay time, either suppled by the application or calculated as the average delay time within the selected AuralAttribute's application region, will be used.

The feedback loop is repeated until AuralAttribute's reverberation feedback loop count is reached or $Gr^j \leq 0.000976$ (effective zero amplitude, –60 dB, using the measure of –6 dB drop for every doubling of distance).

Fallbacks and Approximations

1. Reducing the number of feedback loops repeated while still maintaining the overall impression of the environment. For example, if –10 dB were used as the drop in gain for every doubling of distance, a scale factor of 0.015625 could be used as the effective zero amplitude, which can be reached in only 15 loop iterations (rather than the 25 needed to reach 0.000976).
2. Using preprogrammed "room" reverberation algorithms that allow selection of a fixed set of "reverberation types" (for example, large hall, small living room), which have implied reflection coefficients, delay times, and feedback loop durations.

The parameters used in the reverberation equations are as follows:

D	=	Delay function
fLoop	=	Reverberation feedback loop count
Gr	=	Reverberation coefficient acting as a gain scale-factor
I	=	Stereo image of unreflected sound sources
R	=	Reverberation for each sound sources
Sample	=	Sound digital sample with a specific sample rate, bit precision, and an optional encoding and/or compression format
t	=	Time
Tr	=	Reverberation delay time (approximating first-order delay in the AuralAttribute region)

E.3.2 Speaker Playback Equations

Different speaker playback equations are used depending on whether the system uses monaural or stereo speakers.

E.3.2.1 Monaural Speaker Output

The equations for headphone playback need only be modified to output a single signal, rather than two signals for left and right transducers. Although there is only one speaker, distance and filter attenuation, Doppler effect, elevation, and front and back cues can be distinguished by the listener and should be included in the sound image generated.

E.3.2.2 Stereo Speaker Output

In a two-speaker playback system, the signal from one speaker is actually heard by both ears and this affects the spectral balance and interaural intensity and time differences heard by each of the listener's ears. Cross-talk cancellation must be performed on the right and left signal to compensate for the delayed attenuated signal heard by the ear opposite the speaker. Thus a delayed attenuated signal for each of the stereo signals must be added to the output from the equations for headphone playback.

The equations for stereo speaker playback assume that the two speakers are placed symmetrically about the listener (at the same off-axis angle from the viewing axis at an equal distance from the center of the listener's head).

$$I'(t)_l = I(t)_l + [D(t) \bullet [G(P, \alpha) \cdot I(t)_r]] \tag{E.24}$$

$$I'(t)_r = I(t)_r + [D(t) \bullet [G(P, \alpha) \cdot I(t)_l]] \tag{E.25}$$

The parameters used in the cross-talk equations, expanding on the terms used for the equations for headphone playback, are as follows:

α = Angle between vectors from speaker to near and far ears

D = Delay function of signal variant over time

G = Gain attenuation scale factors function taking initial distance and angular gain scale factors into account

I = Sound image for left and right stereo signals calculated as for headphone output

P = Distance difference between near ear and far ear as defined for ITD, the speaker substituted for the sound source in equation

t = Time

E.4 Texture Mapping Equations

Texture mapping can be divided into two steps. The first step takes the transformed *s* and *t* (and possibly *r*) texture coordinates, the current texture image, and the texture filter parameters, and computes a texture color based on looking up the texture coordinates in the texture map. The second step applies the computed texture color to the incoming pixel color using the specified texture mode function.

E.4.1 Texture Lookup

The texture lookup stage maps a texture image onto a geometric polygonal primitive. The most common method for doing this is to reverse map the *s* and *t* coordinates from the primitive back onto the texture image, then filter and resample the image. In the simplest case, a point in s, t space is transformed into a u, v address in the texture image space (E.26), then this address is used to look up the nearest texel value in the image. This method, used when the selected texture filter function is `BASE_LEVEL_POINT`, is called nearest-neighbor sampling or point sampling.

$$\begin{aligned} u &= s \cdot width \\ v &= t \cdot height \end{aligned} \tag{E.26}$$

$$\begin{aligned} i &= \mathrm{trunc}(u) \\ j &= \mathrm{trunc}(v) \end{aligned} \tag{E.27}$$

$$Ct = T_{i,\,j} \tag{E.28}$$

If the texture boundary mode is `REPEAT`, then only the fractional bits of s and t are used, ensuring that both s and t are less than 1.

If the texture boundary mode is `CLAMP`, then the s and t values are clamped to be in the range [0, 1] before being mapped into u and v values. Further, if $s \geq 1$, then i is set to $width - 1$; if $t \geq 1$, then j is set to $height - 1$.

The parameters in the point-sampled texture lookup equations are as follows:

width	=	Width, in pixels, of the texture image
height	=	Height, in pixels, of the texture image
s	=	Interpolated s coordinate at the pixel being textured
t	=	Interpolated t coordinate at the pixel being textured
u	=	u coordinate in texture image space
v	=	v coordinate in texture image space
i	=	Integer row address into texture image
j	=	Integer column address into texture image
T	=	Texture image

The above equations are used when the selected texture filter function—either the minification or the magnification filter function—is `BASE_LEVEL_POINT`. Java 3D selects the appropriate texture filter function based on whether the texture image is minified or magnified when it is applied to the polygon. If the texture is applied to the polygon such that more than one texel maps onto a single pixel, then the texture is said to be minified and the minification filter function is selected. If the texture is applied to the polygon such that a single texel maps onto more than one pixel, then the texture is said to be magnified and the magnification filter function is selected. The selected function is one of the following: `BASE_LEVEL_POINT`, `BASE_LEVEL_LINEAR`, `MULTI_LEVEL_POINT`, or `MULTI_LEVEL_LINEAR`. In the case of magnification, the filter will always be one of the two base level functions (`BASE_LEVEL_POINT` or `BASE_LEVEL_LINEAR`).

If the selected filter function is `BASE_LEVEL_LINEAR`, then a weighted average of the four texels that are closest to the sample point in the base level texture image is computed.

$$\begin{aligned} i_0 &= \text{trunc}(u - 0.5) \\ j_0 &= \text{trunc}(v - 0.5) \\ i_1 &= i_0 + 1 \\ j_1 &= j_0 + 1 \end{aligned} \tag{E.29}$$

$$\begin{aligned} \alpha &= \text{frac}(u - 0.5) \\ \beta &= \text{frac}(v - 0.5) \end{aligned} \tag{E.30}$$

$$\begin{aligned} Ct = {} & (1 - \alpha) \cdot (1 - \beta) \cdot T_{i_0, j_0} + \alpha \cdot (1 - \beta) \cdot T_{i_1, j_0} \\ & + (1 - \alpha) \cdot \beta \cdot T_{i_0, j_1} + \alpha \cdot \beta \cdot T_{i_1, j_1} \end{aligned} \tag{E.31}$$

If the selected filter function is `MULTI_LEVEL_POINT` or `MULTI_LEVEL_LINEAR`, the texture image needs to be sampled at multiple levels of detail. If multiple levels of detail are needed and the texture object only defines the base level texture image, Java 3D will compute multiple levels of detail as needed.

Mipmapping is the most common filtering technique for handling multiple levels of detail. If the implementation uses mipmapping, the equations for computing a texture color based on texture coordinates are simply those used by the underlying rendering API (such as OpenGL or PEX). Other filtering techniques are possible as well.

Fallbacks and Approximations

1. If the texture boundary mode is `CLAMP`, an implementation may either use the closest boundary pixel or the constant boundary color attribute for those values of *s* or *t* that are outside the range [0, 1].
2. An implementation can choose a technique other than mipmapping to perform the filtering of the texture image when the texture minification filter is `MULTI_LEVEL_POINT` or `MULTI_LEVEL_LINEAR`.
3. If mipmapping is chosen by an implementation as the method for filtering, it may approximate trilinear filtering with another filtering technique. For example, an OpenGL implementation may choose to use `LINEAR_MIPMAP_NEAREST` or `NEAREST_MIPMAP_LINEAR` in place of `LINEAR_MIPMAP_LINEAR`.

E.4.2 Texture Application

Once a texture color has been computed, this color is applied to the incoming pixel color. If lighting is enabled, only the emissive, ambient, and diffuse components of the incoming pixel color are modified. The specular component is added into the modified pixel color after texture application.

The equations for applying that color to the original pixel color are based on the texture mode, as follows:

REPLACE Texture Mode

$$C' = Ct \tag{E.32}$$

MODULATE Texture Mode

$$C' = C \cdot Ct \tag{E.33}$$

DECAL Texture Mode

$$\begin{aligned} C'_{rgb} &= C_{rgb} \cdot (1 - Ct_{\alpha}) + Ct_{rgb} \cdot Ct_{\alpha} \\ C'_{\alpha} &= C_{\alpha} \end{aligned} \tag{E.34}$$

Note that the texture format must be either `RGB` or `RGBA`.

BLEND Texture Mode

$$\begin{aligned} C'_{rgb} &= C_{rgb} \cdot (1 - Ct_{rgb}) + Cb_{rgb} \cdot Ct_{rgb} \\ C'_{\alpha} &= C_{\alpha} \cdot Ct_{\alpha} \end{aligned} \tag{E.35}$$

Note that if the texture format is `INTENSITY`, alpha is computed identically to red, green, and blue:

$$C'_{\alpha} = C_{\alpha} \cdot (1 - Ct_{\alpha}) + Cb_{\alpha} \cdot Ct_{\alpha} \tag{E.36}$$

The parameters used in the texture mapping equations are as follows:

C = Color of the pixel being texture mapped (if lighting is enabled, then this does not include the specular component)

Ct = Texture color

Cb = Blend color

Note that C_{rgb} indicates the red, green, and blue channels of color C and that C_α indicates the alpha channel of color C. This convention applies to the other color variables as well.

If there is no alpha channel in the texture, a value of 1 is used for Ct_α in `BLEND` and `DECAL` modes.

When the texture mode is one of `REPLACE`, `MODULATE`, or `BLEND`, only certain of the red, green, blue, and alpha channels of the pixel color are modified, depending on the texture format, as described below.

- INTENSITY: All four channels of the pixel color are modified. The intensity value is used for each of Ct_r, Ct_g, Ct_b, and Ct_α in the texture application equations, and the alpha channel is treated as an ordinary color channel—the equation for C'_{rbg} is also used for C'_α.
- LUMINANCE: Only the red, green, and blue channels of the pixel color are modified. The luminance value is used for each of Ct_r, Ct_g, and Ct_b in the texture application equations. The alpha channel of the pixel color is unmodified.
- ALPHA: Only the alpha channel of the pixel color is modified. The red, green, and blue channels are unmodified.
- LUMINANCE_ALPHA: All four channels of the pixel color are modified. The luminance value is used for each of Ct_r, Ct_g, and Ct_b in the texture application equations, and the alpha value is used for Ct_α.
- RGB: Only the red, green, and blue channels of the pixel color are modified. The alpha channel of the pixel color is unmodified.
- RGBA: All four channels of the pixel color are modified.

Fallbacks and Approximations

An implementation may apply the texture to all components of the lit color, rather than separating out the specular component. Conversely, an implementation may separate out the emissive and ambient components in addition to the specular component, potentially applying the texture to the diffuse component only.

APPENDIX F

VRML Support

THIS appendix is designed to help VRML browser developers design a VRML browser and runtime environment using the Java 3D API. Sun has already prototyped a VRML 1.0 and VRML 2.0 3D-only browser. This browser has limited browsing functionality but includes the ability to load worlds, navigate, and pick.

VRML files come in one of two formats: VRML 1.0 and VRML 2.0. These two file formats are sufficiently different that developers will most likely support them separately. In general, VRML 1.0 files allow the definition of static geometry, predefined "viewpoints," linking to other worlds, and some simple browser-specific semantics (pick to transport, menu to change viewpoint). The newer VRML 2.0 file format provides an improved facility for defining static geometry and includes support for representing a broader range of information, such as active components and sound.

A developer can use Java 3D functionality to build a VRML loader and browser much as he or she would use C and OpenGL to write a VRML loader and associated browser. The combination of Java and the Java 3D API—like the combination of C and the OpenGL API—provides programmers with sufficient functionality to write complete applications. The Java-based approach additionally allows the development of complete applets.

F.1 VRML 1.0

VRML 1.0 files describe geometry, the material properties associated with that geometry, and the placement of that geometry with respect to other geometry, all within a single virtual world. The file format also includes other properties such as lights, attach points (via cameras), and links to other files.

VRML 1.0 files also assume the existence of a browser and, over time, there have been some default browser semantics that have become a de facto standard.

Specifically these include browser-constructed menus, gleaned from the VRML 1.0 file, that allow an end user to choose among a predefined set of viewpoints, browser controls that allow an end user to navigate within the virtual world, and mouse-based pick operations that allow an end user to gain access to other worlds.

F.1.1 Mapping VRML 1.0 Files onto Java 3D Objects

VRML 1.0 files do not map directly onto Java 3D objects. However, Java 3D objects support the complete functionality needed to support VRML 1.0 files. Thus, a developer can write a simple loader to parse a VRML 1.0 file, construct a Java 3D scene graph that represents the information contained in the VRML file, and let Java 3D render the scene. Unfortunately, this does not address the browser issue.

F.1.2 A VRML 1.0 Browsing Environment

VRML 1.0 browsers allow end users to navigate within a VRML world, to "pick" objects by using their mouse, to choose among viewpoints defined within the scene graph via menus, and perform other housekeeping tasks (such as loading a world).

By using Java, specifically Java's AWT package, developers can construct windows, menus, and buttons that allow end-user access to developer-written functionality. A developer would build a browser that would interface with the VRML 1.0 loader to load a VRML world and retrieve references to relevant scene graph information. The browser would interface with the Java 3D API to draw the world, to change viewpoints, to process pick operations, and to navigate within the world.

F.2 VRML 2.0

The VRML 2.0 file format allows the description of geometry, material properties, and object placement, but in an easier manner than the VRML 1.0 file format. In addition, the VRML 2.0 file format includes mechanisms for describing sensors, routes and fields, script nodes, interpolators, support for collision detection, and support for picking.

A similar approach to the one used for VRML 1.0 would have worked (a loader/browser combination) had there not been sensors, routes, fields, and script nodes. These new features require a different approach since information flow within a

VRML 2.0 file is specified via routes: Java 3D does not include a similar structure.

F.2.1 VRML Support Requires a VRML Runtime Environment

VRML 2.0 files contain both geometry and behavior components. VRML 2.0 geometry components map straightforwardly onto Java 3D objects. VRML 2.0 behaviors (routes and scripting nodes), however, do not map directory onto Java 3D behaviors. The behaviors in a VRML 2.0 scene could be mapped onto a set of Java 3D classes that mimic the scene's functionality, but, unfortunately, this does not work in all cases. VRML 2.0's runtime semantics, specifically the ability to retarget routes to other fields and the ability to access all the descendants of a referenced VRML 2.0 object, require that the same structure that is specified in the VRML 2.0 file exists within the runtime environment.

Another problem area involves script nodes based on languages other than Java. We do not anticipate developers defining interpreters for other languages that run within the Java environment. We expect that developers will support only Java-based script nodes.

This mismatch between VRML 2.0 and Java 3D is bidirectional: There are Java 3D constructs and behaviors that do not map directly onto the VRML 2.0 file format.

F.2.2 An Approach

Developers can host a VRML 2.0 file within a Java 3D environment by constructing exactly the same scene graph structure as specified in a VRML 2.0 file. They can then use the Java 3D behavior system or its mixed-mode callback features to implement field value propagation. The remainder of this section defines one such approach.

F.2.2.1 The Scene Graph Structure

A developer defines a series of thin-layer Java objects that correspond to VRML 2.0 node types. These thin-layer objects contain references to a set of Java objects that represent VRML field objects and references to underlying Java 3D objects that implement the functionality of the VRML node. These objects also include code to propagate field value changes to their underlying Java 3D objects. Thus, a VRML 2.0 scene graph is represented as a thin layer of Java objects, associated Java 3D objects, and methods that manipulate the Java 3D scene graph whenever field values change in the thin-layer object.

The thin-layer objects (VRML nodes) exist only to translate VRML semantics into appropriate actions at runtime. If a VRML scene has subgraphs that cannot be accessed during runtime (because no routes connect into nodes within that subgraph) then the developer may choose not to retain—or even create—these thin-layer VRML node objects, constructing only the underlying Java 3D objects.

F.2.2.2 The Execution Environment

The developer of a VRML 2.0 runtime environment must also include code to propagate events along the routes specified in a VRML file. Either user interactions or VRML sensors, such as timers, can generate events that change field values. When a field's value changes, the thin-layer VRML node object that contains the field first updates its underlying Java 3D object(s), and then updates the field's associated output fields with the new value. The VRML runtime code then propagates the new output field values, through routes, to other fields. The resulting event cascade moves through the VRML route structure, resulting in an updated Java 3D scene graph.

The developer must somehow integrate event generation into the Java 3D execution environment. This can be done using the Java 3D behavior mechanism to generate events. For example, a WakeupOnElapsedTime or WakeupOnElapsedFrames wakeup criteria can trigger a VRML TimeSensor event, and a mouse-button WakeupOnAWTEvent wakeup criteria can trigger a VRML TouchSensor event. Other VRML sensors can be implemented in a similar manner.

F.2.3 A Browser

Much like implementing a VRML 1.0 browser, a developer can implement a VRML 2.0 browser using Java. A VRML 2.0 browser includes specific functionality, such as Viewpoint binding and a browser interface for use by Script nodes. Most of this latter functionality involves interaction with the thin-layer VRML node objects, but the browser may call Java 3D directly as well. For example, a VRML browser developer can implement VRML Viewpoints by associating Java 3D ViewPlatform nodes with each VRML Viewpoint. Then, to change VRML Viewpoints, the browser would detach the Java 3D View object from the current Java 3D ViewPlatform object and reattach it to the new Java 3D ViewPlatform object associated with the desired VRML Viewpoint.

F.2.4 Optimizing for Viewing versus Editing

A VRML browser need not provide access to the complete VRML 2.0 scene graph and, indeed, a developer can take advantage of this to minimize the number of thin-layer VRML 2.0 objects. If there is no way to reference a VRML 2.0 node, then its thin-layer object need not exist.

An environment that allows editing must keep the entire thin-layer-object scene graph in memory.

In either the viewing-only or the editing case, a developer can straightforwardly write a VRML 2.0 file to disk by traversing the thin-layer-object representation of the scene graph.

Glossary

avatar

The software representation of a person as the person appears to others in a shared virtual universe. The avatar may or may not resemble an actual person.

branch graph

A graph rooted to a BranchGroup node. See also *scene graph* and *shared graph*.

CC

Clipping coordinates.

center ear

Midpoint between left and right ears of listener.

center eye

Midpoint between left and right eyes of viewer. This is the head coordinate system origin.

compiled

A subgraph may be compiled by an application using the `compile` method of the root node—a BranchGroup or a SharedGroup—of the graph. A compiled object is any object that is part of a compiled graph. An application can compile some or all of the subgraphs that make up a complete scene graph. Java 3D compiles these graphs into an internal format. Additionally, Java 3D provides restricted access to methods of compiled objects or graphs. See also *live*.

compiled-retained mode

One of three modes in which Java 3D objects are rendered. In this mode, Java 3D renders the scene graph, or a portion of the scene graph, that has been previously compiled into an internal format. See also *retained mode*, *immediate mode*.

DAG
Directed acyclic graph. A *scene graph*.

EC
Eye coordinates.

frustum
See *view frustum*.

group node
A node within a *scene graph* that composes, transforms, selects, and in general modifies its descendant nodes. See also *leaf node*, *root node*.

HMD
Head-mounted display.

image plate
The display area; the viewing screen or head-mounted display.

immediate mode
One of three modes in which Java 3D objects are rendered. In this mode objects are rendered directly, under user control, rather than as part of a scene graph traversal. See also *retained mode*, *compiled-retained mode*.

IID
Interaural intensity difference. The difference between the perceived amplitude (gain) of the signal from a source as it reaches the listener's left and right ears.

ITD
Interaural time difference. The difference in time in the arrival of the signal from a sound source as it reaches the listener's left and right ears.

leaf node
A node within a *scene graph* that contains the visual, auditory, and behavioral components of the scene. See also *group node*, *root node*.

live
A live graph is any graph that is attached to a Locale object, or a shared graph that is referenced by a live graph. A live object is any object that is part of a live graph. Live objects are subject to being traversed and rendered by the

Java 3D renderer. Additionally, Java 3D provides restricted access to methods of live objects or graphs. See also *compiled*.

LOD

Level of detail. A predefined Behavior that operates on a Switch node to select from among multiple versions of an object or collection of objects.

polytope

A bounding volume defined by a closed intersection of half-spaces.

retained mode

One of three modes in which Java 3D objects are rendered. In this mode, Java 3D traverses the scene graph and renders the objects that are in the graph. See also *compiled-retained mode*, *immediate mode*.

root node

A node within a *scene graph* that establishes the default environment. See also *group node*, *leaf node*.

scene graph

A collection of branch graphs rooted to a Locale. A virtual universe has one or more scene graphs. See also *branch graph* and *shared graph*.

shared graph

A graph rooted to a SharedGroup node. See also *branch graph* and *scene graph*.

stride

The part of an interleaved array that defines the length of a vertex.

three space

Three-dimensional space.

view frustum

A truncated, pyramid-shaped viewing area that defines how much of the world the viewer sees. Objects not within the view frustum are not visible. Objects that intersect the boundaries of the viewing frustum are clipped (partially drawn).

VPC
View platform coordinates.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

T

U

V

W

X

Y

Z