Eurographics 98

Introduction to Programming with Java3D

Lecturers

David R. Nadeau (Organizer)
nadeau@sdsc.edu
http://www.sdsc.edu/~nadeau
San Diego Supercomputer Center
University of California at San Diego

Henry A. Sowizral
henry.sowizral@eng.sun.com
Sun Microsystems, Inc.

Tutorial notes sections

Introduction to Programming with Java 3D

Abstract

Java 3D is a new cross-platform API for developing 3D graphics applications in Java. Java 3D's feature set has been designed to enable quick development of complex 3D applications, and at the same time enable fast and efficient implementations on a variety of platforms, from PCs to workstations. Using Java 3D, software developers can build cross-platform applications that build 3D scenes programmatically, or via loading 3D content from VRML, OBJ, and/or other external files. The Java 3D API includes a rich feature set for building shapes, composing behaviors, interacting with the user, and controlling rendering details.

Participants in this tutorial will learn the concepts behind Java 3D, the Java 3D class hierarchy, typical usage patterns, ways of avoiding common mistakes, animation and scene design techniques, and tricks for increasing performance and realism.

Introduction to Programming with Java 3D

Preface

Welcome to the *Introduction to Programming with Java 3D* tutorial notes! These tutorial notes have been written to give you a quick, practical, example-driven overview of *Java 3D*, the cross-platform 3D graphics API for Java. To do this, we've included almost 500 pages of tutorial material with nearly 100 images and over 50 Java 3D examples.

To use these tutorial notes you will need:

- An HTML Web browser
- Java Development Kit (JDK) 1.2 or later
- Java 3D 1.2 or later

Information on Java JDKs and Java 3D is available at:

http://www.javasoft.com

What's included in these notes

These tutorial notes primarily contain two types of information:

1. General information, such as this preface
2. Tutorial slides and examples

The tutorial slides are arranged as a sequence of 400+ hyper-linked pages containing Java 3D syntax notes, Java 3D usage comments, or images of sample Java 3D applications. Clicking on the file name underneath an image brings up a window showing the Java source file that generated the image. The Java source files contain extensive comments providing information about the techniques the file illustrates.

Compiling and executing the Java example file from the command-line brings up a Java application illustrating a Java 3D feature. Most such applications include menus and other interaction options with which you can explore Java 3D features.

The tutorial notes provide a necessarily terse overview of Java 3D. We recommend that you invest in a Java 3D book to get thorough coverage of the language. Two of the course lecturers are authors of the Java 3D specification, available from Addison-Wesley: *The Java 3D API Specification*, ISBN 0-201-32576-4, 1997.

Use of these tutorial notes

We are often asked if there are any restrictions on use of these tutorial notes. The answer is:

If you use these tutorial notes, we ask that you:

1. Give us credit for the original material
2. Tell us since we like hearing about the use of our material!

If you find bugs in the notes, please tell us. We have worked hard to try and make the notes bug-free, but if something slipped by, we’d like to fix it before others are confused by our mistake.

Contact

David R. Nadeau
University of California
NPACI/SDSC, MC 0505
9500 Gilman Drive
La Jolla, CA 92093-0505

(619) 534-5062
FAX: (619) 534-5152

nadeau@sdsc.edu
http://www.sdsc.edu/~nadeau

Introduction to Programming with Java 3D

Lecturer information

David R. Nadeau (Organizer)

Title	Principal Scientist
Affiliation	San Diego Supercomputer Center (SDSC) University of California, San Diego (UCSD)
Address	NPACI/SDSC, MC 0505 9500 Gilman Drive La Jolla, CA 92093-0505
Email	nadeau@sdsc.edu
Home page	http://www.sdsc.edu/~nadeau

Dave Nadeau is a principal scientist at the San Diego Supercomputer Center (SDSC), a national research center specializing in computational science and engineering, located on the campus of the University of California, San Diego (UCSD). Specializing in scientific visualization and virtual reality, he is the author of technical papers and articles on 3D graphics and VRML and is a co-author of two books on VRML (*The VRML Sourcebook* and *The VRML 2.0 Sourcebook*, published by John Wiley & Sons). He is the founder and lead librarian for *The VRML Repository* and *The Java 3D Repository*, principal Web sites for information on VRML, Java 3D, and related software.

Dave has taught VRML at multiple conferences including SIGGRAPH 96-97, WebNet 96-97, VRML 97-98, WMC/SCS 98, Eurographics 97, and Visualization 97. He has taught Java 3D at Visualization 97, SIGGRAPH 98, and courses at SDSC and Sun Microsystems. He was a co-chair for the *VRML Behavior Workshop* in October 1995, the first workshop on VRML behavior technology, and a co-chair for the *VRML 95* conference in December 1995, the first conference on VRML. He was on the program committees for VRML 97 and VRML 98 and is SDSC's representative to the VRML Consortium.

Dave holds a B.S. in Aerospace Engineering from the University of Colorado, Boulder, an M.S. in Mechanical Engineering from Purdue University, and is in the Ph.D. program in Electrical and Computer Engineering at the University of California, San Diego.

Henry A. Sowizral

Title	Senior Staff Engineer
Affiliation	Sun Microsystems, Inc.
Address	901 San Antonio Road, MS UMPK14-202 Palo Alto, CA 94303-4900 UPS, Fed Ex: 14 Network Circle Menlo Park, CA, 94025
Email	henry.sowizral@eng.sun.com

Henry Sowizral is a Senior Staff Engineer at Sun Microsystems where he is the chief architect of the Java 3D API. His areas of interest include virtual reality, large model visualization, and distributed and concurrent simulation. He has taught numerous tutorials on topics including expert systems and virtual reality at conferences including COMPCON, Supercomputing, VRAIS, and SIGGRAPH. Henry has taught Java 3D at SIGGRAPH 97-98, JavaOne, and other conferences.

Henry is a co-author of the book *The Java 3D API Specification*, published by Addison-Wesley. He holds a B.S. in Information and Computer Science from the University of California, Irvine, and an M.Phil. and Ph.D. in Computer Science from Yale University.

Caveat

Beta releases

These tutorial notes were last updated in late July 1998. As of this date, the most current version of Java 3D is *version 1.1 beta 1*, the first beta release after three public alpha releases. This release of Java 3D is dependent upon sound, image, and font features found in JDK 1.2, whose most current release as of July is *version 1.2 beta 3*!

The good news is that Java 3D and JDK 1.2 are leading-edge technologies and represent the forefront of 3D application development technology. The bad news is that these tutorial notes are based upon *beta* releases, and as such are as unstable as those early software releases.

As with all beta product releases, functionality in both Java 3D and the JDK is not yet complete or fully stable. This is natural and appropriate for early releases. The impact on these tutorial notes is that some Java 3D features could not be adequately discussed or example applications generated. For instance, the beta 1 Java 3D release is missing stable support for:

- Compressed geometry
- 3D text geometry and fonts
- Texture images with alpha channels
- Some behavior wakeup conditions
- Full OBJ file loading
- Background geometry
- MIDI sound files
- Cone sounds
- Sound localization, reverb, and doppler shift

Some JDK 1.2 features do not mesh well with Java 3D as of the JDK beta 3 release, including support for:

- Mixing swing lightweight and Java 3D components
- Mixing Java Media Framework classes and Java 3D sound classes
- Reading image files other than GIF and JPEG

It is inevitable that some Java 3D or JDK 1.2 features will change between July 1998 and the final release of these APIs. This may cause some of the example applications in these notes to fail to compile or execute properly with the final APIs. Wherever possible, we have avoided use of features we felt were unstable and likely to change. This should enhance the chance that the early examples included here will work with the final release of Java 3D and JDK 1.2.

Getting the latest tutorial notes

As we near the final release of Java 3D and JDK 1.2, expected to be late in 1998, we will will continue to evolve these tutorial notes, updating examples and slide coverage.

Updated versions of these notes, and their Java source code, are available for free downloading from:

- Sun’s Java 3D site
- The Java 3D Repository
- Dave Nadeau’s home page

We **strongly** encourage you to check and download updated copies as they become available. We appologize for the inconvenience. We feel this situation is natural and unavoidable as we track a moving technology target and seek to get you the latest and most complete information we can about Java 3D.

Thank you.
-- the lecturers

Introduction to Programming with Java 3D

Using the Java examples

These tutorial notes include dozens of separate Java applications illustrating the use of Java 3D. The source code for these applications is included in files with `.java` file name extensions. Compiled byte-code for these Java files is *not included*! To use these examples, you will need to compile the applications first.

Compiling Java

The source code for all Java 3D examples is in the `examples` folder. Images, sound, and geometry files used by these examples are also contained within the same folder. A `README.txt` file in the folder lists the Java 3D applications included therein.

To compile the Java examples, you will need:

- The Java 3D API class files
- The JDK 1.2 class files
- A Java compiler

The JDK 1.2 class files are available for free from JavaSoft at http://www.javasoft.com.

The Java 3D class files are available for free from Sun Microsystems at http://www.sun.com/desktop/java3d.

There are multiple Java compilers available for most platforms. JavaSoft provides the Java Development Kit (JDK) for free from its Web site at http://www.javasoft.com. The JDK includes the `javac` compiler and instructions on how to use it. Multiple commercial Java development environments are available from Microsoft, Silicon Graphics, Symantec, and others. An up to date list of available Java products is available at Developer.com's Web site at http://www.developer.com/directories/pages/dir.java.html.

Once you have the Java API class files and a Java compiler, you may compile the supplied Java files. Unfortunately, we can't give you explicit directions on how to do this. Each platform and Java compiler is different. You'll have to consult your software's manuals.

Running the Java 3D Examples

To run a Java application, you must run the Java interpreter and give it the Java class file as an argument, like this:

```
java MyClass
```

The Java interpreter looks for the file `MyClass.class` in the current directory and loads it, and any additional files needed by that class.

Title Page

Introduction to Programming with Java 3D

Table of contents

1

Welcome

Introduction to Programming with Java 3D

Welcome to the course!

Welcome

Speakers

Henry Sowizral
Michael Deering
Sun Microsystems

Dave Nadeau
Mike Bailey
SDSC / UCSD

Schedule for the Day

8:30 - 10:00	Introduction, Overview, Shapes, Appearance
10:00 - 10:15	*Java break*
10:15 - 12:00	Groups, Transforms, Texture mapping
12:00 - 1:30	*Lunch*
1:30 - 3:00	Viewing, Behaviors, Interpolators, Picking, Input
3:00 - 3:15	*Java break*
3:15 - 5:00	Lighting, Backgrounds, Fog, Sound

Tutorial Scope

- This tutorial will:
    - Introduce general Java 3D concepts
    - Give an overview of Java 3D classes and methods
    - Show how to design a scene graph
    - Explain how to write a Java 3D program
    - Discuss typical usage patterns (the good, the bad, and the tricks)

6

Introduction

What is Java 3D?

- Java 3D is a runtime API for developing applications and applets

- "Write once, run anywhere"
  - Multiple platforms
  - Multiple display environments

Java 3D API Goals

- A high-level 3D capability for Java
- High performance
- Platform independence
- Rich but minimal feature set

A Higher Level API

- Raise the programming floor

- Think
  - Objects --- not vertices
  - Content --- not rendering process

Why Use Java 3D?

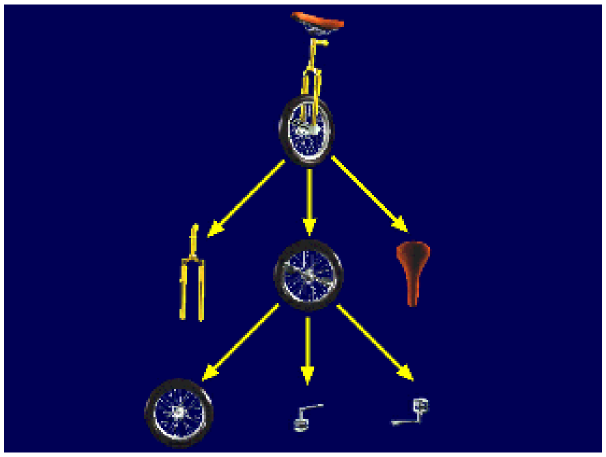
- Integration with other Java APIs
- Portability
- Consistency across platforms
- Scalability

Programming Paradigm

- Scene-graph specification
  - Objects
  - Placement
  - Grouping

- Java-based execution

A Conceptual Scene Graph



Overview of Java 3D Architecture

- Designed for parallelism

- Independent asynchronous components
  - Automatic rendering
  - Behavior and sound scheduling
  - Event generation (collision detection)
  - Input device management

- Java 3D renderer chooses traversal order
  - Neither left-to-right nor top-to-bottom
  - Except spatially bounded attributes

Resources for developing Java 3D

- JDK 1.2
- Java 3D class files
    - http://www.sun.com/desktop/java3d
- Read the README file
- Look at the example code in the distribution
- Utility classes

Summary

- General-purpose, interactive, rendering environment
- Broad range of features
- Cross-platform performance

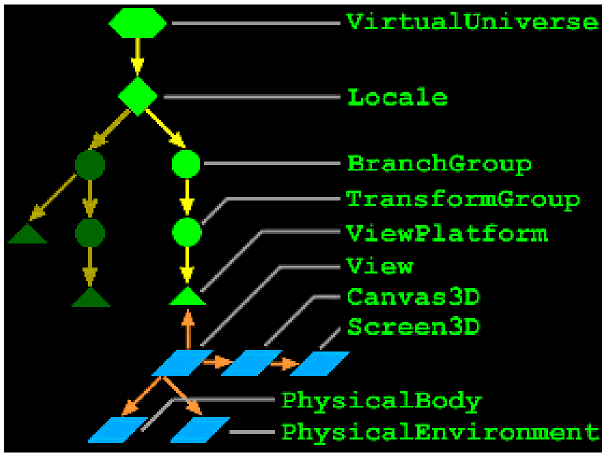
Creating a Java 3D application or applet

Constructing an Application

- Your application creates 3D content, arranged in a *scene graph*
  - Groups of content within groups, and so on to create a family tree of 3D content
  - The *scene graph* contains everything to draw
- A scene graph can be represented by a tree-like diagram, typically with two *branches*
  - *View branch* for viewing controls
  - *Content branch* for 3D shapes, lights, etc.

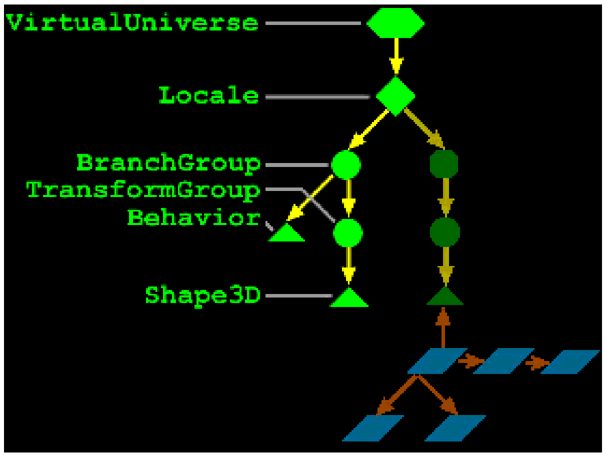
Creating a Java 3D application or applet

The View Branch



Creating a Java 3D application or applet

The Content Branch



Constructing Content

- Java 3D provides an assortment of classes for making different types of 3D content
  - Shapes and geometry
  - Lights and sounds
  - Fog and backgrounds
  - Groups and animations

- Components of a scene graph are derived from base classes:
  - **`SceneGraphObject`**
  - **`Node`**
  - **`NodeComponent`**

SceneGraphObject Class Hierarchy

- **`SceneGraphObject`** is the base class for all objects that may be included in a scene graph
  - Subclasses create shapes, sounds, lights, etc.

Class Hierarchy

```
java.lang.Object
 └─  javax.media.j3d.SceneGraphObject
      ├─  javax.media.j3d.Node
      └─  javax.media.j3d.NodeComponent
```

SceneGraphObject Capabilities

- Adding a node into a scene graph makes it "live" (drawable)
  - When live, a node may be modified only in predetermined ways
  - The fewer the ways, the more optimizations Java 3D can perform

- Modifications are enabled by turning on *Capabilities*
  - Every node class has different capabilities
  - The **`SceneGraphObject`** provides methods to turn on these capabilities

- Sample capabilities:
  - Allow read/write of transforms, geometry coordinates, shape colors, texture transforms, group content

SceneGraphObject Class Methods

Method	*Default*
`void setCapability( int bit )`	Unset
`void clearCapability( int bit )`	-
`boolean getCapability( int bit )`	-
`boolean isCompiled( )`	-
`boolean isLive( )`	-
`void setUserData( Object userData )`	Null
`Object getUserData( )`	-

Nodes and NodeComponents

- **`Node`** is the base class for all items on the rendering path through a scene graph
  - Shapes, Groups, Sounds, Lights, etc.

- **`NodeComponent`** is the base class for attributes associated with nodes
  - Shape geometry, Shape appearance, Shape texture, etc.

Node and NodeComponent Class Hierarchy

- **`Node`** and **`NodeComponent`** share the traits of **`SceneGraphObject`**

Class Hierarchy

```
java.lang.Object
 └─  javax.media.j3d.SceneGraphObject
      ├─  javax.media.j3d.Node
      └─  javax.media.j3d.NodeComponent
```

Node Attributes

- Every node has:
    - A parent node
    - A location within the virtual world
    - A bounding volume (usually automatically computed)
        - Enables quick culling of nodes out of view

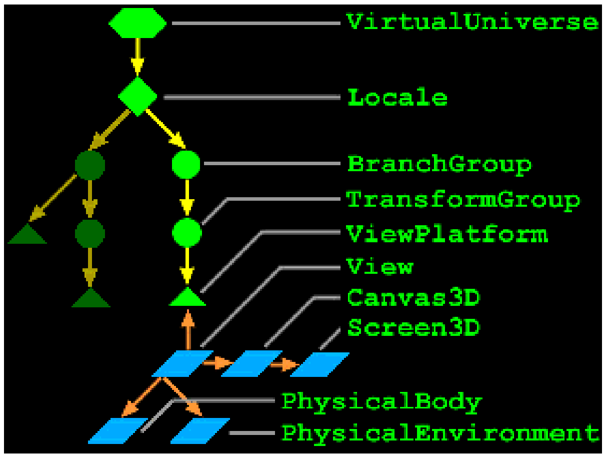
Node Class Methods

Method	*Default*
`Node getParent( )`	None
`void getLocalToVworld( Transform3D t )`	-
`void getLocalToVworld( SceneGraphPath path, Transform3D t )`	-
`void setBounds( Bounds bounds )`	Auto
`Bounds getBounds( )`	Auto
`void setAutoComputeBounds( boolean autoCompute )`	true
`boolean getAutoComputeBounds( )`	true

Creating a Java 3D application or applet

The View Branch

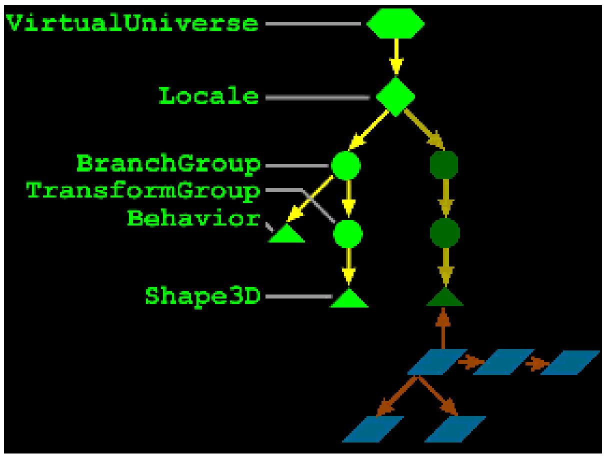
- To build an "HelloUniverse" example recall that we’ll need to set up the view branch . . .



Creating a Java 3D application or applet

The Content Branch

- . . . and the content branch



HelloUniverse Example Code

- Set up the frame to draw into

```
public static void main( String[] args )
{
    new MainFrame( new HelloUniverse( ), 640, 480 );
}
```

Creating a Java 3D application or applet

HelloUniverse Example Code

- Construct the view and content branches
  - Use **`SimpleUniverse`** to make a typical view branch

```
public HelloUniverse( ) {
    Canvas3D c = new Canvas3D( null );
    add( "Center", c );

    // Create View Branch
    SimpleUniverse u = new SimpleUniverse( c );

    // Create Content Branch
    BranchGroup scene = createSceneGraph( );
    u.addBranchGraph( scene );
}
```

HelloUniverse Example Code

- Scene content creation

```
public BranchGroup createSceneGraph( )
{
    BranchGroup objRoot = new BranchGroup( );

    // Create the transform group node and
    // enable transform write by behavior
    TransformGroup objTrans = new TransformGroup( );
    objTrans.setCapability( TransformGroup.ALLOW_TRANSFORM_W
    objRoot.addChild( objTrans );

    // Create a cube shape and add to scene
    objTrans.addChild( new ColorCube( 0.4 ) );
    . . .
```

HelloUniverse Example Code

- Rotation behavior setup

```
// Create Behavior to rotate shape
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 );
. . .
```

HelloUniverse Example Code

- Compile and done

```
    // Optimize scene
    objRoot.compile( );

    return objRoot;
}
```

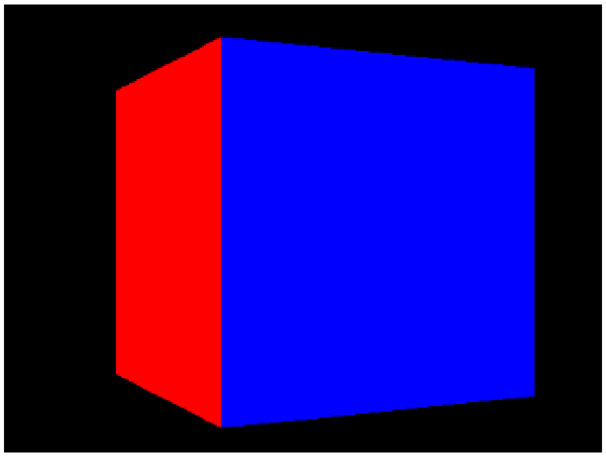
HelloUniverse Example Code

- Typical import statements:

```
import javax.media.j3d.*;
import javax.vecmath.*;
import java.applet.Applet;
import java.awt.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.behaviors.*;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.applet.MainFrame;
```

Creating a Java 3D application or applet

HelloUniverse Example



[`HelloUniverse`]

Summary

Class	*Attributes*
`SceneGraphObject`	Capabilities, User data, Liveness
├ `Node`	Bounds, LocalToVworld
└ `NodeComponent`	-

Building 3D Shapes

Motivation

- Shapes are the fundamental means of describing objects
- A **`Shape3D`** leaf node contains:

```
Node: Shape3D
Geometry
Appearance
```

- Geometry describes the form, or structure of a shape
  - Several types of geometry available
- Appearance describes the coloration and shading of that shape

Building 3D Shapes

Example

[**`GearBox`**]

Shape3D Class Hierarchy

- The **`Shape3D`** class extends the **`Leaf`** class

Class Hierarchy

```
java.lang.Object
 └─  javax.media.j3d.SceneGraphObject
      └─  javax.media.j3d.Node
           └─  javax.media.j3d.Leaf
                └─  javax.media.j3d.Shape3D
```

Shape3D Class Methods

Method	*Default*
`Shape3D( )`	-
`void setGeometry( Geometry geometry )`	None
`void setAppearance( Appearance appearance )`	None

Shape Geometry

- Shape geometry is described by classes extended from the **`Geometry`** class
  - **`GeometryArray`** builds points, lines, triangles, and quads
  - **`Text3D`** builds 3D extruded text
  - **`Raster`** builds image sprites
  - **`CompressedGeometry`** builds compressed forms of points, lines, triangles, and quads

GeometryArray Class Hierarchy

- The **`GeometryArray`** subclasses provide the basis for constructing points, lines, triangles, and quads

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.NodeComponent
         └─ javax.media.j3d.Geometry
             └─ javax.media.j3d.GeometryArray
                 ├─ javax.media.j3d.GeometryStripArray
                 │   ├─ javax.media.j3d.LineStripArray
                 │   ├─ javax.media.j3d.TriangleFanArray
                 │   └─ javax.media.j3d.TriangleStripArray
                 ├─ javax.media.j3d.IndexedGeometryArray
                 │   ├─ javax.media.j3d.IndexedGeometryStripArray
                 │   │   ├─ javax.media.j3d.IndexedLineStripArray
                 │   │   ├─ javax.media.j3d.IndexedTriangleFanArray
                 │   │   └─ javax.media.j3d.IndexedTriangleStripArray
                 │   ├─ javax.media.j3d.IndexedLineArray
                 │   ├─ javax.media.j3d.IndexedPointArray
                 │   ├─ javax.media.j3d.IndexedQuadArray
                 │   └─ javax.media.j3d.IndexedTriangleArray
                 ├─ javax.media.j3d.LineArray
                 ├─ javax.media.j3d.PointArray
                 ├─ javax.media.j3d.QuadArray
                 └─ javax.media.j3d.TriangleArray
```

Default Coordinate System and Units

- Geometry is described using 3D coordinates in a coordinate system:
    - Right-handed system
    - When facing the screen:
        - +X is to the right
        - +Y is up
        - +Z is towards the viewer

- Angles are in radians

- Distance is in meters

Facedness

- Ideal triangles and quads
- One-sided primitives
- Winding order determines surface normal
- Counterclockwise winding => surface normal faces out
  - Right-hand rule

GeometryArrays

- **`GeometryArray`** is extended to build:
  - Simple geometry:
    - **`PointArray`**, **`LineArray`**, **`TriangleArray`**, and **`QuadArray`**
  - Strip geometry:
    - **`LineStripArray`**, **`TriangleStripArray`**, and **`TriangleFanArray`**

GeometryArray Class Components

- The **`GeometryArray`** parent class contains:

Node: **`Shape3D`**

`GeometryArray`

  - Coordinates
  - Colors
  - Normals
  - Texture Coordinates

`Appearance`

- Classes extend **`GeometryArray`** to build specific geometries

IndexedGeometryArrays

- **`IndexedGeometryArray`** is extended to build:
  - Indexed simple geometry:
    - **`IndexedPointArray`**, **`IndexedLineArray`**, **`IndexedTriangleArray`**, and **`IndexedQuadArray`**
  - Indexed stripped geometry:
    - **`IndexedLineStripArray`**, **`IndexedTriangleStripArray`**, and **`IndexedTriangleFanArray`**

IndexedGeometryArray Class Components

- The **`IndexedGeometryArray`** parent class extends the **`GeometryArray`** class and adds:

Node: **`Shape3D`**

`IndexedGeometryArray`
- Coordinate Indices
- Color Indices
- Normal Indices
- Texture Coordinate Indices

`Appearance`

- Classes extend **`IndexedGeometryArray`** to build specific geometries

GeometryArray Class Methods

Method
`void setCoordinate( int index, * coordinate )`
`void setCoordinates( int index, * coordinate )`
`void setColor( int index, * color )`
`void setColors( int index, * color )`
`void setNormal( int index, * normal )`
`void setNormals( int index, * normal )`
`void setTextureCoordinate( int index, * texCoord )`
`void setTextureCoordinates( int index, * texCoord )`

- Variants accept byte, float, double, **`Point3f`**, **`Point3d`**, **`Color3f`**, **`Color4f`**, **`Color3b`**, **`Color4b`**, and **`Vector3f`** (as appropriate)

IndexedGeometryArray Class Methods

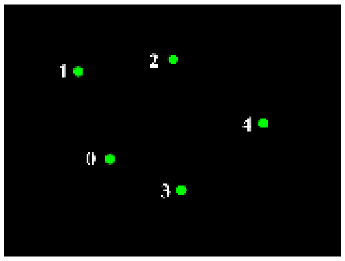
Method
`void setCoordinateIndex( int index, int value )`
`void setCoordinateIndices( int index, int[] value )`
`void setColorIndex( int index, int value )`
`void setColorIndices( int index, int[] value )`
`void setNormalIndex( int index, int value )`
`void setNormalIndices( int index, int[] value )`
`void setTextureCoordinateIndex( int index, int value )`
`void setTextureCoordinateIndices( int index, int[] value )`

Building 3D Shapes

Gear Train Example

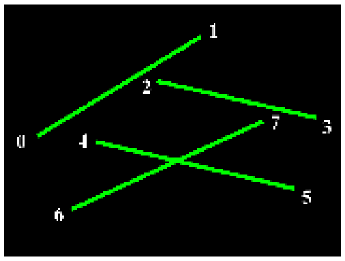
[**`GearBox`**]

Building a PointArray



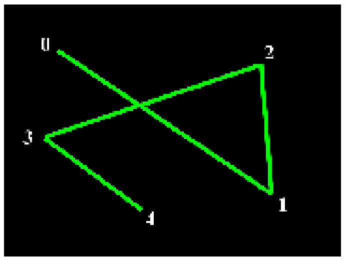
```
PointArray points = new PointArray(
    vertexCount,
    GeometryArray.COORDINATES );
points.setCoordinates( 0, coords );
```

Building a LineArray



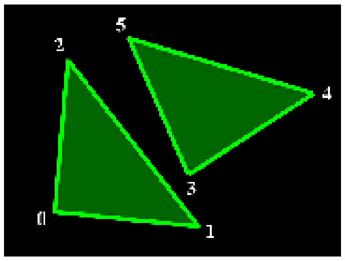
```
LineArray lines = new LineArray(
    vertexCount,
    GeometryArray.COORDINATES );
lines.setCoordinates( 0, coords );
```

Building a LineStripArray



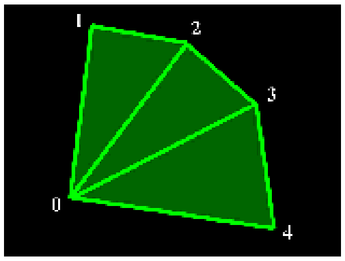
```
LineStripArray mlines = new LineStripArray(
    vertexCount,
    GeometryArray.COORDINATES,
    stripVertexCounts[] );
mlines.setCoordinates( 0, coords );
```

Building a TriangleArray



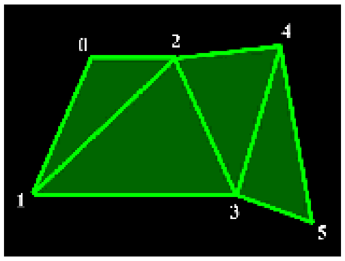
```
TriangleArray tris = new TriangleArray(
    vertexCount,
    GeometryArray.COORDINATES |
    GeometryArray.NORMALS );
tris.setCoordinates( 0, coords );
tris.setNormals( 0, normals );
```

Building a TriangleFanArray



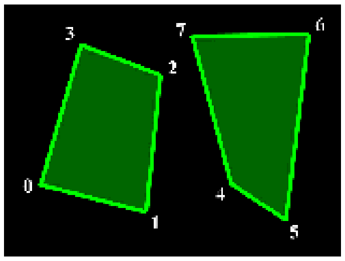
```
TriangleFanArray fans = new TriangleFanArray(
    vertexCount,
    GeometryArray.COORDINATES |
    GeometryArray.NORMALS,
    stripVertexCounts[] );
fans.setCoordinates( 0, coords );
fans.setNormals( 0, normals );
```

Building a TriangleStripArray



```
TriangleStripArray strips = new TriangleStripArray(
    vertexCount,
    GeometryArray.COORDINATES |
    GeometryArray.NORMALS,
    stripVertexCounts[] );
strips.setCoordinates( 0, coords );
strips.setNormals( 0, normals );
```

Building a QuadArray



```
QuadArray quads = new QuadArray(
    vertexCount,
    GeometryArray.COORDINATES |
    GeometryArray.NORMALS );
quads.setCoordinates( 0, coords );
quads.setNormals( 0, normals );
```

Summary

Class	*Attributes*
`GeometryArray`	Geometry base class
├ `LineArray`	Individual lines
├ `PointArray`	Individual points
├ `QuadArray`	Individual quads
└ `TriangleArray`	Individual triangles

Summary

Class	*Attributes*
`GeometryArray`	Geometry base class
└ `GeometryStripArray`	Strip geometry base class
├ `LineStripArray`	Strip of connected lines
├ `TriangleStripArray`	Strip of connected triangles
└ `TriangleFanArray`	Fan of connected triangles

Summary

Class	***Attributes***
`GeometryArray`	Geometry base class
└─ `IndexedGeometryArray`	Indexed geometry base class
├─ `IndexedLineArray`	Individual indexed lines
├─ `IndexedPointArray`	Individual indexed points
├─ `IndexedQuadArray`	Individual indexed quads
└─ `IndexedTriangleArray`	Individual indexed triangles

Summary

Class	*Attributes*
`GeometryArray`	Geometry base class
└ **`IndexedGeometryArray`**	Indexed geometry base class
└ **`IndexedGeometryStripArray`**	Indexed stripped geometry base class
├ **`IndexedLineStripArray`**	Strip of indexed lines
├ **`IndexedTriangleStripArray`**	Strip of indexed triangles
└ **`IndexedTriangleFanArray`**	Fan of indexed triangles

Building Text Shapes

Motivation

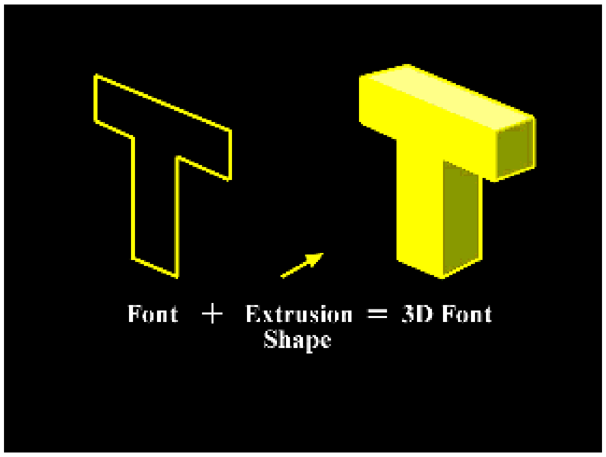
- 3D content may contain text shapes for annotation, signs, etc.
- **`Text3D`** creates 3D text geometry for a **`Shape3D`**
- Compact representation

Building 3D Text

- 3D text geometry is built from a text string and a 3D font
  - Select a 2D font with **`java.awt.Font`**
  - Describe a 2D extrusion shape with **`java.awt.Shape`** in a **`FontExtrusion`**
  - Create a 3D font by extruding the 2D font along the extrusion shape with a **`Font3D`**
  - Create 3D text using a string and a **`Font3D`** in a **`Text3D`**

Building Text Shapes

Building a 3D Font



FontExtrusion and Font3D Class Hierarchy

- **`FontExtrusion`** specifies an extrusion shape and **`Font3D`** specifies a font

Class Hierarchy
`java.lang.Object` ├─ **`javax.media.j3d.FontExtrusion`** └─ **`javax.media.j3d.Font3D`**

Text3D Class Hierarchy

- **`Text3D`** extends the **`Geometry`** class to describe 3D text geometry

Class Hierarchy

```
java.lang.Object
 └─  javax.media.j3d.SceneGraphObject
      └─  javax.media.j3d.NodeComponent
           └─  javax.media.j3d.Geometry
                └─  javax.media.j3d.Text3D
```

Text3D Class Components

- **`Text3D`** components describe the text string, positioning, and 3D font:

Node: **`Shape3D`**

```
Text3D
   String
   Position
   Alignment
   Path
   Character Spacing
   Font3D
      Font
      FontExtrusion
          Shape
Appearance
```

FontExtrusion and Font3D Class Methods

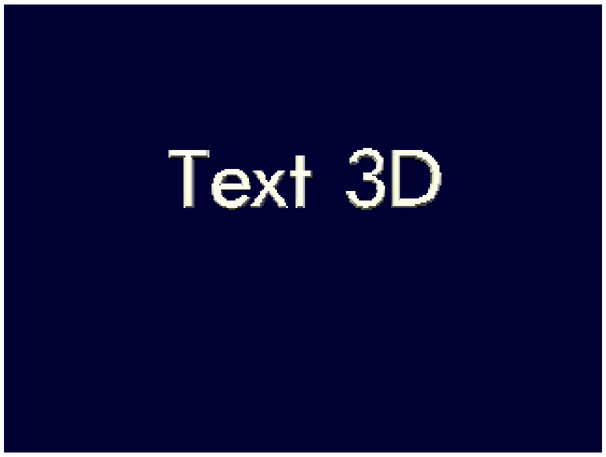
Method
`FontExtrusion( )`
`void setExtrusionShape( Shape extrusionShape )`

Method
`Font3D( Font font, FontExtrusion shape )`
`GeometryStripArray[] getAsTriangles( int glyphCode )`
`Bounds getBounds( int glyphCode )`

Text3D Class Methods

Method	*Default*
`Text3D( )`	**-**
`void setFont3D( Font3d font )`	None
`void setString( String string )`	None
`void setPosition( Point3f position )`	0.0 0.0 0.0
`void setAlignment( int alignment )`	**`ALIGN_FIRST`**
`void setPath( int Path )`	**`PATH_RIGHT`**
`void setCharacterSpacing( float spacing )`	0.0
`void getBoundingBox( BoundingBox Bounds )`	**-**

3D Text Example



Summary

Class	***Attributes***
`Geometry`	-
└ **`Text3D`**	3D font, String, Position, Alignment, Placement path, Character spacing
`Font3D`	2D Font, Extrusion path
`FontExtrude`	Extrusion 2D shape

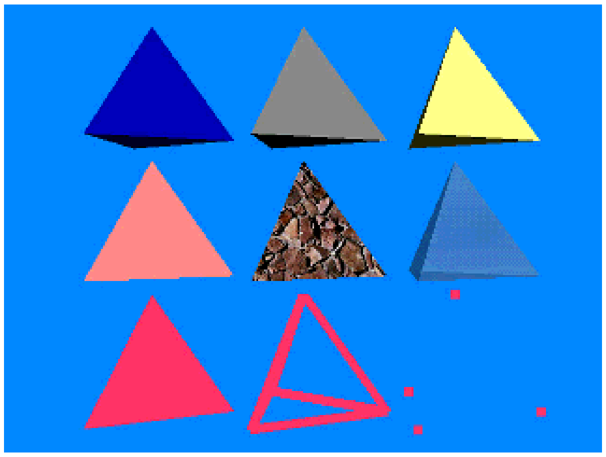
Controlling appearance

Motivation

- Control how Java 3D renders **`Geometry`**
  - Color
  - Transparency
  - Shading model
  - Line thickness
  - And lots more

- All appearance control is encapsulated within the **`Appearance`** class, and its components

Example



[**ExAppearance**]

Appearance Class Hierarchy

- The **`Appearance`** class specifies how to render its **`Geometry`** sibling

Class Hierarchy

```
java.lang.Object
  └─  javax.media.j3d.SceneGraphObject
        └─  javax.media.j3d.NodeComponent
              └─  javax.media.j3d.Appearance
```

Appearance Class Components

- Appearance attributes are grouped into separate node components for color and shading, rendering control, and texture mapping

Node: **`Shape3D`**

```
Geometry
Appearance
  ColoringAttributes
  LineAttributes
  PointAttributes
  PolygonAttributes
  RenderingAttributes
  TextureAttributes
  TransparencyAttributes
  Material
  TexCoordGeneration
  Texture
```

Appearance Attributes Class Hierarchy

- The various appearance attributes extend the **`NodeComponent`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.NodeComponent
         ├─ javax.media.j3d.ColoringAttributes
         ├─ javax.media.j3d.LineAttributes
         ├─ javax.media.j3d.PointAttributes
         ├─ javax.media.j3d.PolygonAttributes
         ├─ javax.media.j3d.RenderingAttributes
         ├─ javax.media.j3d.TextureAttributes
         ├─ javax.media.j3d.TransparencyAttributes
         ├─ javax.media.j3d.Material
         ├─ javax.media.j3d.TexCoordGeneration
         └─ javax.media.j3d.Texture
```

Appearance Class Methods

Method
`Appearance( )`
`void setColoringAttributes( ColoringAttributes coloringAttributes )`
`void setMaterial( Material material )`
`void setTransparencyAttributes( TransparencyAttributes transparencyAttributes )`

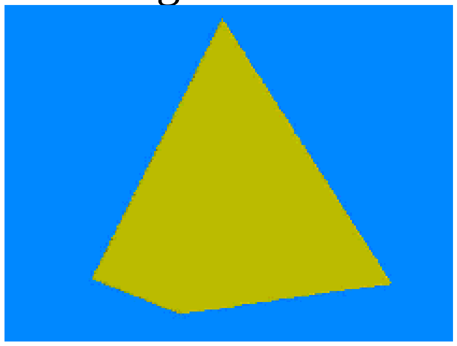
Appearance Class Methods

Method
`void setLineAttributes( LineAttributes lineAttributes )`
`void setPointAttributes( PointAttributes pointAttributes )`
`void setPolygonAttributes( PolygonAttributes polygonAttributes )`
`void setRenderingAttributes( RenderingAttributes renderingAttributes )`

Coloring Attributes

- A **`ColoringAttributes`** node component controls:
  - Intrinsic color (used when lighting disabled)
  - Shading model (flat or Gouraud)

- Primarily use coloring attributes when a shape *isn't* shaded
  - Emissive points, lines, and polygons
  - Avoids expensive shading calculations



ColoringAttributes Class Methods

Method	*Default*
`ColoringAttributes( )`	**`-`**
`void setColor( Color3f color )`	1.0 1.0 1.0
`void setShadeModel( int model )`	**`SHADE_GOURAUD`**

- Shade models include: **`SHADE_FLAT`** and **`SHADE_GOURAUD`**
- The **`FASTEST`** and **`NICEST`** shade models automatically select the fastest, and highest quality models available

ColoringAttributes Example Code

- Set the intrinsic color and shading model

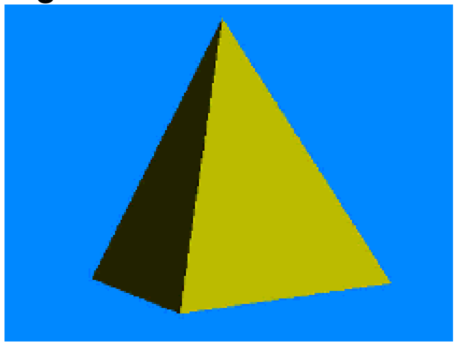
```
Appearance app = new Appearance( );
ColoringAttributes ca = new ColoringAttributes( );
ca.setColor( 1.0f, 1.0f, 0.0f );
ca.setShadeModel( ColoringAttributes.SHADE_GOURAUD );
app.setColoringAttributes( ca );
```

Controlling appearance

Material Attributes

- A **`Material`** node component controls:
  - Ambient, emissive, diffuse, and specular color
  - Shininess factor

- Primarily use materials when a shape *is* shaded
  - Most scene shapes
  - Overrides **`ColoringAttributes`** intrinsic color



Controlling appearance

Material Class Methods

Method	*Default*
`Material( )`	-
`void setAmbientColor( Color3f color )`	0.2 0.2 0.2
`void setEmissiveColor( Color3f color )`	0.0 0.0 0.0
`void setDiffuseColor( Color3f color )`	1.0 1.0 1.0
`void setSpecularColor( Color3f color )`	1.0 1.0 1.0
`void setShininess( float shininess )`	0.0
`void setLightingEnable( boolean state )`	true

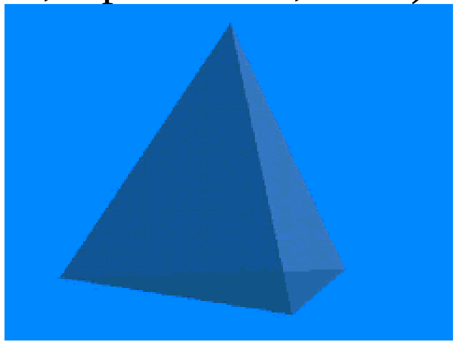
Material Attributes Example Code

- Set ambient, emissive, diffuse, and specular colors

```
Appearance app = new Appearance( );
Material mat = new Material( );
mat.setAmbientColor(  0.3f, 0.3f, 0.3f );
mat.setDiffuseColor(  1.0f, 0.0f, 0.0f );
mat.setEmissiveColor( 0.0f, 0.0f, 0.0f );
mat.setSpecularColor( 1.0f, 1.0f, 1.0f );
mat.setShininess( 80.0f );
app.setMaterial( mat );
```

Transparency Attributes

- A **`TransparencyAttributes`** node component controls:
  - Transparency amount (0.0 = opaque, 1.0 = invisible)
  - Mode (screen-door, alpha-blend, none)



TransparencyAttributes Class Methods

Method	*Default*
`TransparencyAttributes( )`	**-**
`void setTransparencyMode( int mode )`	**`FASTEST`**
`void setTransparency( float transparency )`	0.0

- Transparency modes include: **`SCREEN_DOOR`**, **`BLENDED`**, and **`NONE`**
- The **`FASTEST`** and **`NICEST`** transparency modes automatically select the fastest, and highest quality modes available

TransparencyAttributes Example Code

- Set transparency and the transparency mode

```
Appearance app = new Appearance( );
TransparencyAttributes ta = new TransparencyAttributes( );
ta.setTransparency( 0.5f );
ta.setTransparencyMode( TransparencyAttributes.BLENDED );
app.setTransparencyAttributes( ta );
```

Point and Line Attributes

- A **`PointAttributes`** node component controls:
  - Point size
  - Point anti-aliasing

- A **`LineAttributes`** node component controls:
  - Line width and pattern
  - Line anti-aliasing

Controlling appearance

PointAttributes Class Methods

Method	*Default*
`PointAttributes( )`	-
`void setPointSize( float size )`	1.0
`void setPointAntialiasingEnable( boolean state )`	false

LineAttributes Class Methods

Method	*Default*
`LineAttributes( )`	-
`void setLineWidth( float width )`	1.0
`void setLinePattern( int pattern )`	`PATTERN_SOLID`
`void setLineAntialiasingEnable( boolean state )`	false

- Line patterns include: **`PATTERN_SOLID`**, **`PATTERN_DASH`**, **`PATTERN_DOT`**, and **`PATTERN_DASH_DOT`**

PointAttributes Example Code

- Set point size and anti-aliasing mode

```
Appearance app = new Appearance( );
PointAttributes pta = new PointAttributes( );
pta.setPointSize( 10.0f );
pta.setPointAntialiasingEnable( false );
app.setPointAttributes( pta );
```

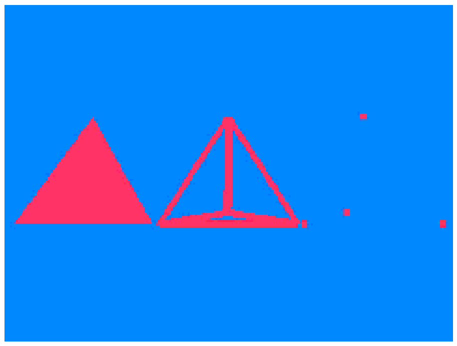
LineAttributes Example Code

- Set line width, pattern, and anti-aliasing mode

```
Appearance app = new Appearance( );
LineAttributes lta = new LineAttributes( );
lta.setLineWidth( 10.0f );
lta.setLineAntialiasingEnable( false );
lta.setLinePattern( LineAttributes.PATTERN_SOLID );
app.setLineAttributes( lta );
```

Polygon Attributes

- A **`PolygonAttributes`** node component controls:
- Polygon attributes are set with a **`PolygonAttributes`** node
    - Face culling (front, back, neither)
    - Fill mode (point, line, fill)
    - Z offset



PolygonAttributes Class Methods

Method	*Default*
`PolygonAttributes( )`	**-**
`void setCullFace( int cullface )`	`CULL_BACK`
`void setPolygonMode( int mode )`	`POLYGON_FILL`
`void setPolygonOffset( float offset )`	0.0

- Face culling modes include: **`CULL_NONE`**, **`CULL_BACK`**, and **`CULL_FRONT`**
- Polygon modes include: **`POLYGON_POINT`**, **`POLYGON_LINE`**, and **`POLYGON_FILL`**

PolygonAttributes Example Code

- Set culling and polygon modes

```
Appearance app = new Appearance( );
PolygonAttributes pa = new PolygonAttributes( );
pa.setCullFace( PolygonAttributes.CULL_NONE );
pa.setPolygonMode( PolygonAttributes.POLYGON_FILL );
app.setPolygonAttributes( pa );
```

Rendering Attributes

- A **`RenderingAttributes`** node component controls:
  - Depth buffer use and write enable
  - Alpha buffer test function and value

RenderingAttributes Class Methods

Method	*Default*
`RenderingAttributes( )`	-
`void setDepthBufferEnable( boolean state )`	true
`void setDepthBufferWriteEnable( boolean state )`	true
`void setAlphaTestFunction( int func )`	**`ALWAYS`**
`void setAlphaTestValue( float value )`	0.0

- Alpha test functions include: **`ALWAYS`**, **`NEVER`**, **`EQUAL`**, **`NOT_EQUAL`**, **`LESS`**, **`LESS_OR_EQUAL`**, **`GREATER`**, and **`GREATER_OR_EQUAL`**

RenderingAttributes Example Code

- Set depth buffer and alpha modes

```
Appearance app = new Appearance( );
RenderingAttributes ra = new RenderingAttributes( );
ra.setDepthBufferEnable( true );
ra.setalphaTestFunction( RenderingAttributes.ALWAYS );
app.setRenderingAttributes( ra );
```

Summary of Shading Components

- `Appearance` class shading components control shape color, transparency, and the shading model

Node: **`Shape3D`**

- **`Geometry`**
- **`Appearance`**
  - **`ColoringAttributes`**
    - Base Color
    - Shade Model
  - **`LineAttributes`**
  - **`Material`**
    - Ambient Color
    - Emissive Color
    - Diffuse Color
    - Specular Color
    - Shininess
    - Lighting Enable
  - **`PointAttributes`**
  - **`PolygonAttributes`**
  - **`RenderingAttributes`**
  - **`TextureAttributes`**
  - **`TexCoordGeneration`**
  - **`Texture`**
  - **`TransparencyAttributes`**
    - Transparency
    - Transparency Mode

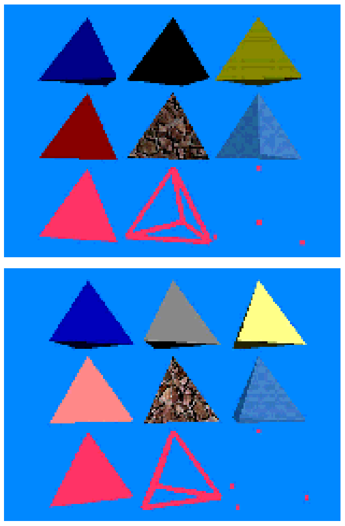
Summary of Rendering Components

- **`Appearance`** class rendering components control how shapes are drawn

Node: **`Shape3D`**

- **`Geometry`**
- **`Appearance`**
  - **`ColoringAttributes`**
  - **`LineAttributes`**
    - Line Width
    - Line Pattern
    - Line Antialiasing
  - **`Material`**
  - **`PointAttributes`**
    - Point Size
    - Point Antialiasing
  - **`PolygonAttributes`**
    - Face Culling
    - Polygon Mode
    - Polygon Offset
  - **`RenderingAttributes`**
    - Depth Buffer Enables
    - Alpha Test and Value
  - **`TextureAttributes`**
  - **`TexCoordGeneration`**
  - **`Texture`**
  - **`TransparencyAttributes`**

Appearance Example



[**ExAppearance**]

Diffuse	Specular	Diffuse & Specular
Shaded	Textured	Transparent
Unlit polygons	Unlit lines	Unlit points

Summary

Class	*Attributes*
`Appearance`	Multiple attribute components
`ColoringAttributes`	Intrinsic color, Shading model
`Material`	Ambient color, Emissive color, Diffuse color, Specular color, Shininess
`TransparencyAttributes`	Transparency, Mode

Summary

Class	*Attributes*
`PointAttributes`	Point size, Anti-aliasing
`LineAttributes`	Line width, Pattern, Anti-aliasing
`PolygonAttributes`	Culling, Polygon mode, Z offset
`RenderingAttributes`	Depth and Alpha modes

Loading pre-authored content

Creating Content Loaders

- Load authored content from files:
  - Objects
  - Scenes
  - Animations

- Use existing modeling tools to create content

- Provide a uniform interface

Loading pre-authored content

Typical Usage

- Construct a loader (or use a loader utility)
  - Open and parse a content file
  - Convert content into Java 3D objects

- Retrieve the "loaded" objects
  - Objects
  - Lights
  - Sounds
  - Backgrounds

- Insert them into a universe

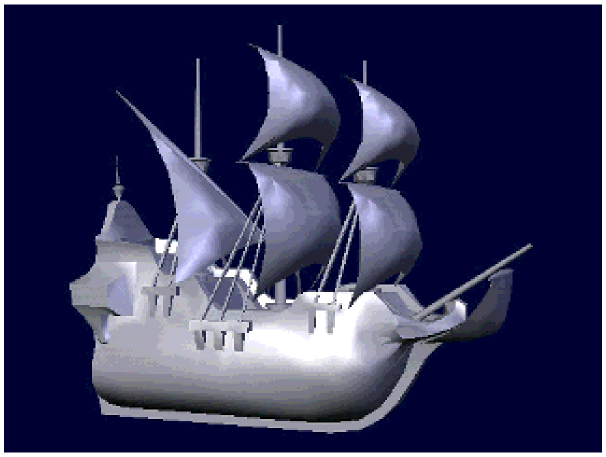
Object Loading Example Code

- **`ObjectFile`** extends **`Group`** and sets its children to content from an OBJ file

```
TransformGroup group = new TransformGroup( );
. . .
ObjectFile f = new ObjectFile( flags, creaseRadians );
Scene s = f.load( filename );
. . .
group.addChild( s.getSceneGroup() );
```

Loading pre-authored content

Object Loading Example



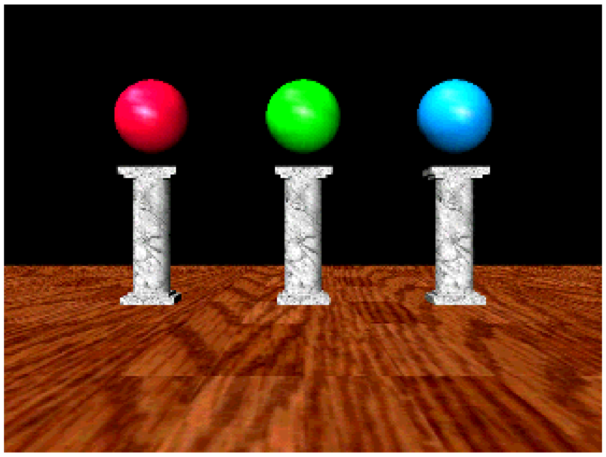
[**`A3DApplet`**]

Grouping Shapes

Motivation

- We want to be able to "clump" elements of the scene in certain ways
- There are several different types of groups, depending on what properties you want the clumped elements to have
- Each group can have any number of children
- With one exception, Java 3D reserves the right to render the children *in any order*

Example



[**`ExSwitch`**]

Types of Groups

- Java 3D provides several types of groups:
  - **`Group`**
  - **`BranchGroup`**
  - **`OrderedGroup`**
  - **`DecalGroup`**
  - **`SharedGroup`**
  - **`Switch`**
  - **`TransformGroup`**
- All groups manage a list of children nodes
  - Shapes, lights, sounds, behaviors

Group Class Hierarchy

- All groups share attributes inherited from the **`Group`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Group
             ├─ javax.media.j3d.BranchGroup
             ├─ javax.media.j3d.OrderedGroup
             │   └─ javax.media.j3d.DecalGroup
             ├─ javax.media.j3d.SharedGroup
             ├─ javax.media.j3d.Switch
             └─ javax.media.j3d.TransformGroup
```

Groups

- The `Group` class is the most general-purpose of the group nodes
  - Has exactly one parent
  - Can have any number of children

- You can add, insert, remove, and get children in a group
  - Children are implicitly numbered starting with 0

- Child rendering order is up to Java 3D!
  - Java 3D can sort shapes for better rendering efficiency

Group Class Methods

Method	*Default*
`Group( )`	-
`void addChild( Node child )`	None
`void setChild( Node child, int index )`	None
`void insertChild( Node child, int index )`	None
`void removeChild( int index )`	None

BranchGroups

- A **`BranchGroup`** is a major sub-tree of the scene graph
  - Can be attached to a **`Locale`**
  - Can be compiled as a unit
  - Can be included as a child of a group node in another subgraph
  - Can be reparented or removed at runtime

- Adding a **`BranchGroup`** to a **`Locale`** makes the group *live*
  - Once live, changes are constrained to those enabled by *capabilities*

BranchGroup Class Hierarchy

- **`BranchGroup`** extends the **`Group`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Group
             ├─ javax.media.j3d.BranchGroup
             ├─ javax.media.j3d.OrderedGroup
             │   └─ javax.media.j3d.DecalGroup
             ├─ javax.media.j3d.SharedGroup
             ├─ javax.media.j3d.Switch
             └─ javax.media.j3d.TransformGroup
```

BranchGroup Class Methods

Method	*Default*
`BranchGroup( )`	-
`void compile( )`	None
`void detach( )`	None

BranchGroup Example Code

```
Locale locale = new Locale( universe );
Shape3D shape = new Shape3D( geom, app );
. . .
BranchGroup branch = new BranchGroup( );
branch.addChild( shape );
branch.compile( );
. . .
locale.addBranchGraph( branch );
```

OrderedGroups

- An **`OrderedGroup`** guarantees children are rendered in first-to-last order
  - Unlike **`Group`**, **`BranchGroup`**, etc.

DecalGroups

- **`DecalGroup`** extends **`OrderedGroup`** and renders children in order
  - Children must be co-planar
  - All polygons must be facing the same way
  - First child is the underlying surface
  - The underlying surface must encompass all other children
  - Use for rendering *decal* geometry
    - Text
    - Texture Decals (eg, insignia on an airplane wing)
  - Good for avoiding Z-fighting artifacts

OrderedGroup and DecalGroup Class Hierarchy

- **`OrderedGroup`** extends the **`Group`** class
- **`DecalGroup`** extends the **`OrderedGroup`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Group
             ├─ javax.media.j3d.BranchGroup
             ├─ javax.media.j3d.OrderedGroup
             │   └─ javax.media.j3d.DecalGroup
             ├─ javax.media.j3d.SharedGroup
             ├─ javax.media.j3d.Switch
             └─ javax.media.j3d.TransformGroup
```

OrderedGroup and DecalGroup Class Methods

Method	*Default*
`OrderedGroup( )`	-

Method	*Default*
`DecalGroup( )`	-

- **`OrderedGroup`** adds no additional methods beyond those of the **`Group`** class
- **`DecalGroup`** adds no additional methods beyond those of the **`OrderedGroup`** class

DecalGroup Example Code

```
Shape3D encompass = new Shape3D( geom,  app );
Shape3D decal1    = new Shape3D( geom1, app1 );
Shape3D decal2    = new Shape3D( geom2, app2 );
. . .
DecalGroup decals = new DecalGroup( );
decals.addChild( encompass );
decals.addChild( decal1 );
decals.addChild( decal2 );
```

Switches

- A **`Switch`** selects zero, one, or multiple children to render
  - Which child to render can be an integer child number, or bits in a mask

- Similar to a Java "switch" statement

- This node is only a switch, it does not imply any rendering order

Switch Class Hierarchy

- **`Switch`** extends the **`Group`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Group
             ├─ javax.media.j3d.BranchGroup
             ├─ javax.media.j3d.OrderedGroup
             │   └─ javax.media.j3d.DecalGroup
             ├─ javax.media.j3d.SharedGroup
             ├─ javax.media.j3d.Switch
             └─ javax.media.j3d.TransformGroup
```

Switch Class Methods

Method	*Default*
`Switch( )`	**`-`**
`void setWhichChild( int index )`	**`CHILD_NONE`**
`void setChildMask( BitSet mask )`	None

- Remember to use **`setCapability( Switch.ALLOW_SWITCH_WRITE );`** to enable the switch value to be changed

Selecting Children

- Select which child to render by passing its child index to **`setWhichChild( )`**
  - Special values:
    - Render no children: **`CHILD_NONE`**
    - Render all children: **`CHILD_ALL`**

- *Or* select a set of children with a bit mask
  - A value of **`CHILD_MASK`** enables mask use
  - Set a member of a Java **`BitSet`** for each child to render

Switch Creation Example Code

```
Shape3D zero = new Shape3D( geom0, app0 );
Shape3D one  = new Shape3D( geom1, app1 );
Shape2D two  = new Shape2D( geom2, app2 );
. . .
Switch sw = new Switch( );
sw.setCapability( Switch.ALLOW_SWITCH_WRITE );
sw.addChild( zero );
sw.addChild( one );
sw.addChild( two );
```

Switch Use Example Code

- Select a single child of the switch group:

```
sw.setWhichChild( 1 );
```

- Select all children of the switch group:

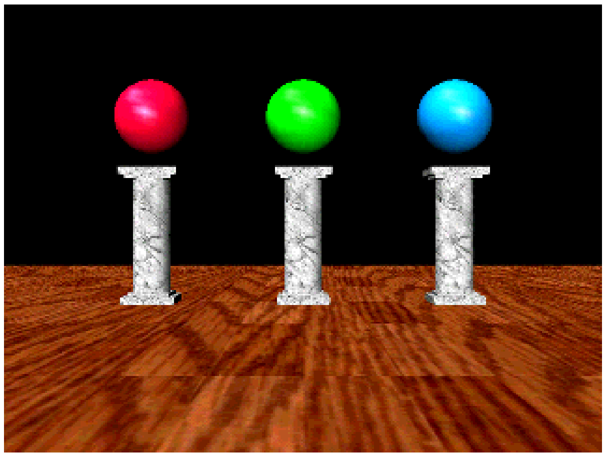
```
sw.setWhichChild( Switch.CHILD_ALL );
```

Switch Use Example Code

- Select a set of children of the switch group:

```
group.setWhichChild( Switch.CHILD_MASK );
BitSet mask = new BitSet(3);
mask.set( 0 );
mask.set( 2 );
group.setChildMask( mask );
```

Switch Group Example



[**`ExSwitch`**]

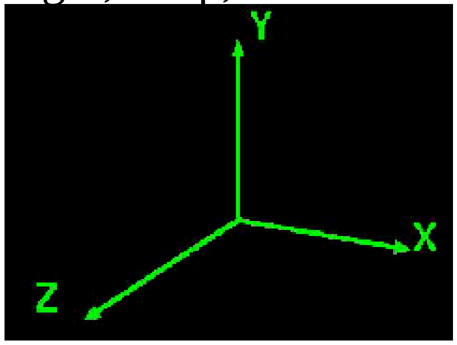
Summary

Class	*Attributes*
`Group`	Add and remove children
├ **`BranchGroup`**	Compile, Detach
├ **`OrderedGroup`**	-
└ **`DecalGroup`**	-
└ **`Switch`**	Which child, Child mask

Transforming Shapes

Motivation

- By default, all shapes are built within a *world coordinate system*
  - Right-handed (X right, Y up, Z toward viewer)



Motivation

- A `TransformGroup` enables you to build a new coordinate system relative to a *parent*
  - Translate
  - Rotate
  - Scale

- Shapes built in the new coordinate system are relative to it
  - If you translate the coordinate system, the shapes move too

TransformGroups

- A *transform* describes translation, rotation, and scale effects
- Transformations accumulate as the scene graph is traversed
- Transformations lower (closest to the geometry) in the scene graph appear to happen before those farther up in the scene graph

TransformGroup Class Hierarchy

- **`TransformGroup`** extends the **`Group`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Group
             ├─ javax.media.j3d.BranchGroup
             ├─ javax.media.j3d.OrderedGroup
             │   └─ javax.media.j3d.DecalGroup
             ├─ javax.media.j3d.SharedGroup
             ├─ javax.media.j3d.Switch
             └─ javax.media.j3d.TransformGroup
```

TransformGroup Class Methods

Method	*Default*
`TransformGroup( )`	-
`void setTransform( Transform3D xform )`	Identity

Transform3D

- The **`Transform3D`** class defines a transform
  - Represented as a 4x4 double precision matrix
  - Can represent translation, rotation, scaling, and shear in any combinations
  - Can represent rotations in a number of ways: principle axis rotations, arbitrary axis rotations, quaternions
  - Can represent scaling as uniform or non-uniform

- A **`Transform3D`** is attached to a **`TransformGroup`** to transform its children

Transform3D Restrictions

- Must be *affine* (ie, no perspective-like homogeneous division)
- Transforms must be *congruent* if used in a **`TransformGroup`** that is an ancestor of a **`ViewPlatform`** (ie, no non-uniform scaling)

Transform3D Class Hierarchy

- **`Transform3D`** extends the **`Object`** class

Class Hierarchy
`java.lang.Object` └ **`javax.media.j3d.Transform3D`**

Transform3D Class Methods

Method
`Transform3D( )`
`Transform3D( Matrix4d mat )`
`Transform3D( Matrix3d rot, Vector3d trans, double scale )`
`void set( Matrix4d mat )`
`void set( Matrix3d rot, Vector3d trans, double scale )`

Building Transforms

- Transforms may be built by:
  - Creating matrices by hand
  - Using helper methods to build **`Matrix4d`** objects
  - Using helper methods to build **`Transform3D`** objects

- Helper methods can:
  - Set the transform to identity
  - Set the translation
  - Set the rotation for X, Y, and Z axes
  - Set the scale (uniform and non-uniform)

Transform3D Class Methods

- Setting the transform to identity

Method
`void setIdentity( )`

- Setting the transform to a translation

Method
`void set( Vector3d trans )`

Transform3D Class Methods

- Setting the transform to a rotation

Method
`void rotX( double angle )`
`void rotY( double angle )`
`void rotZ( double angle )`
`void set( AxisAngle4d axang )`
`void set( Matrix3d rot )`

Transform3D Class Methods

- Setting the transform to a scale factor (uniform or XYZ)

Method
`void set( double scale )`
`void setScale( Vector3d scale )`

Transform3D Class Methods

- Modifying *parts* of an existing transform
  - Leaves the rest of the transform unaffected

Method
`void setTranslation( Vector3d trans )`
`void setRotation( AxisAngle4d axang )`
`void setRotation( Matrix3d rot )`
`void setEuler( Vector3d rollPitchYaw )`
`void setScale( double scale )`

Transforming Vectors and Points

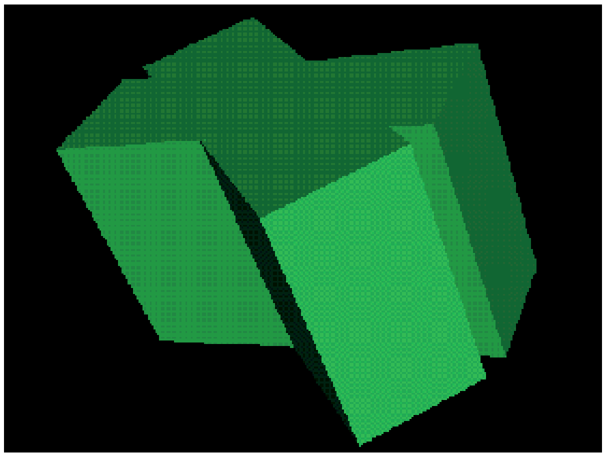
- During rendering, Java 3D processes geometry coordinates and vectors through each **`Transform3D`**
- You can use **`Transform3D`** methods to do this processing on your own points and vectors

Method
`void transform( Point3d inout )`
`void transform( Point3d in, Point3d out )`
`void transform( Vector3d inout )`
`void transform( Vector3d in, Vector3d out )`

TransformGroup Example Code

```
Shape3D shape = new Shape3D( geom, app );
. . .
Transform3D transLeft = new Transform3D( );
transLeft.set( new Vector3d( -1.0, 0.0, 0.0 ) );
. . .
TransformGroup group = new TransformGroup( );
group.setTransform( transLeft );
group.addChild( shape );
```

TransformGroup Example



[**`ExTransform`**]

Summary

Class	*Attributes*
`TransformGroup`	Transform3D
`Transform3D`	Matrix4d (for translation, rotation, scale)

Sharing Groups of Shapes

Motivation

- Some shapes occur over and over in a scene
  - Marble columns in a colonnade
  - Basic user interface components

- Sharing enables you to create one master definition of a shape then link to it multiple times in the scene

SharedGroups

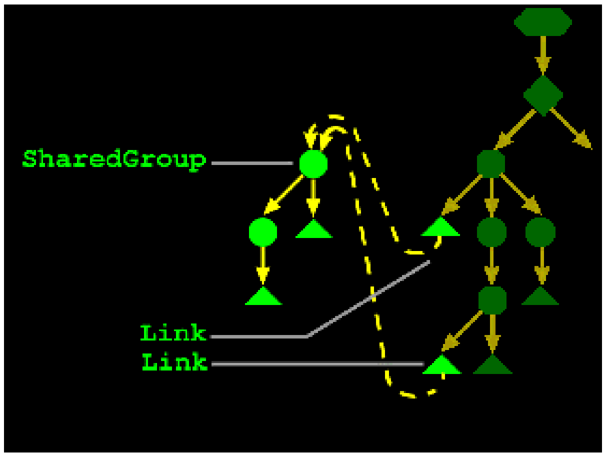
- A **`SharedGroup`** extends **`Group`**
  - It contains shapes, like other groups
  - It is never added into the scene graph directly
  - It is referenced by one or more **`Link`** leaf nodes

- Changes to a **`SharedGroup`** affect all references to it

- Can be compiled by calling its **`compile( )`** method prior to referencing it from a **`Link`** node

Sharing Groups of Shapes

Linking to Shared Groups



SharedGroup Class Hierarchy

- **`SharedGroup`** extends the **`Group`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Group
             ├─ javax.media.j3d.BranchGroup
             ├─ javax.media.j3d.OrderedGroup
             │   └─ javax.media.j3d.DecalGroup
             ├─ javax.media.j3d.SharedGroup
             ├─ javax.media.j3d.Switch
             └─ javax.media.j3d.TransformGroup
```

Sharing Groups of Shapes

SharedGroup Class Methods

Method	*Default*
`SharedGroup( )`	-
`void compile( )`	None

Link Class Hierarchy

- **`Link`** extends the **`Leaf`** class

Class Hierarchy

```
java.lang.Object
  └─ javax.media.j3d.SceneGraphObject
       └─ javax.media.j3d.Node
            └─ javax.media.j3d.Leaf
                 └─ javax.media.j3d.Link
```

Sharing Groups of Shapes

Link Class Methods

Method	*Default*
`Link( )`	-
`Link( SharedGroup group )`	-
`void setSharedGroup( SharedGroup group )`	None

SharedGroup Example Code

```
TransformGroup group = new TransformGroup( );
Shape3D shape0 = new Shape3D( geom0, app0 );
Shape3D shape1 = new Shape3D( geom1, app1 );
. . .
SharedGroup shared = new SharedGroup( );
shared.addChild( shape0 );
shared.addChild( shape1 );
shared.compile( );
. . .
Link link = new Link( shared );
. . .
group.addChild( link );
```

Summary

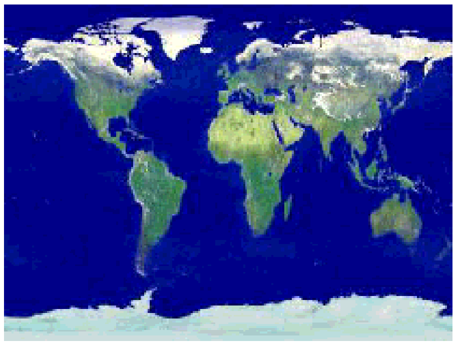
Class	*Attributes*
`SharedGroup`	Compile
`Link`	Shared group

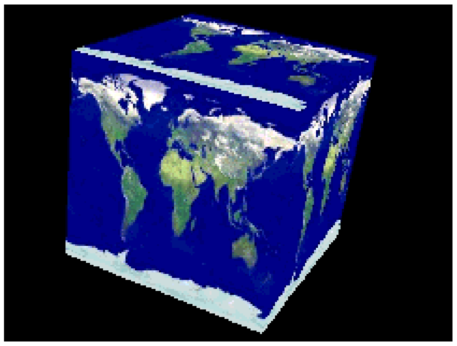
Introducing Texture Mapping

Motivation

- To reduce actual scene detail without reducing *apparent* scene detail
  - Stretch and paste an image onto 3D geometry
  - Increases realism without increasing the amount of geometry

Example


Texture image


[**`ExTexture`**]

Texture Class Hierarchy

- All textures share attributes inherited from the **`Texture`** class

Class Hierarchy

```
java.lang.Object
 └─  javax.media.j3d.SceneGraphObject
      └─  javax.media.j3d.NodeComponent
           └─  javax.media.j3d.Texture
                ├─  javax.media.j3d.Texture2D
                └─  javax.media.j3d.Texture3D
```

Texture Mapping Appearance Class Components

- Texture mapping is controlled by attributes of an **`Appearance`** node component

```
            Node: Shape3D
Geometry
Appearance
  ColoringAttributes
  LineAttributes
  Material
  PointAttributes
  PolygonAttributes
  RenderingAttributes
  TextureAttributes
  TexCoordGeneration
  Texture
  TransparencyAttributes
```

Textures

- The **`Texture`** base class is extended for 2D and 3D textures using **`Texture2D`** and **`Texture3D`**
    - These classes enable/disable texture mapping, and select the image to use

Node: **`Shape3D`**

```
Geometry
Appearance
  TextureAttributes
  Texture2D
    Enable
    Image
    . . .
  . . .
```

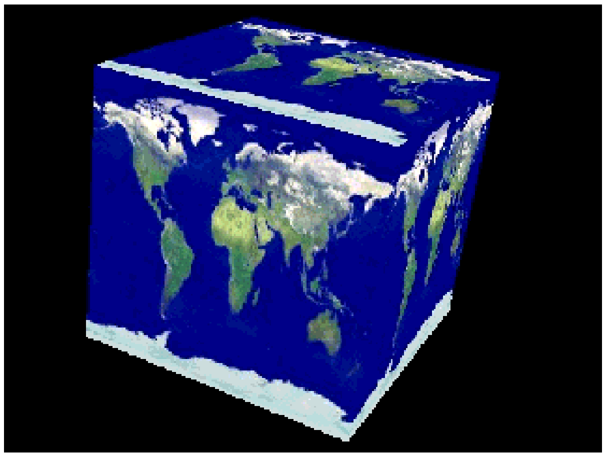
Texture Class Methods

Method	*Default*
`void setEnable( boolean onOff )`	false
`void setImage( int level, ImageComponent2D image )`	None

Texture Example Code

```
Appearance app = new Appearance( );
. . .
TextureLoader loader = new TextureLoader( "earth.jpg", this
ImageComponent2D image = loader.getImage( );
. . .
Texture2D tex = new Texture2D( );
tex.setImage( 0, image );
tex.setEnable( true );
app.setTexture( tex );
```

Texture Example



[**`ExTexture`**]

Creating Texture Images

- The **`Texture`** classes use an **`ImageComponent`** to hold image data
- **`ImageComponent2D`** extends **`ImageComponent`** for 2D images with width and height

ImageComponent Class Hierarchy

- All image components share attributes inherited from the **`ImageComponent`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.NodeComponent
         └─ javax.media.j3d.ImageComponent
             ├─ javax.media.j3d.ImageComponent2D
             └─ javax.media.j3d.ImageComponent3D
```

ImageComponent2D Class Methods

Method	*Default*
`ImageComponent2D( int format, BufferedImage image )`	-
`void set( BufferedImage image )`	None

Loading Texture Images

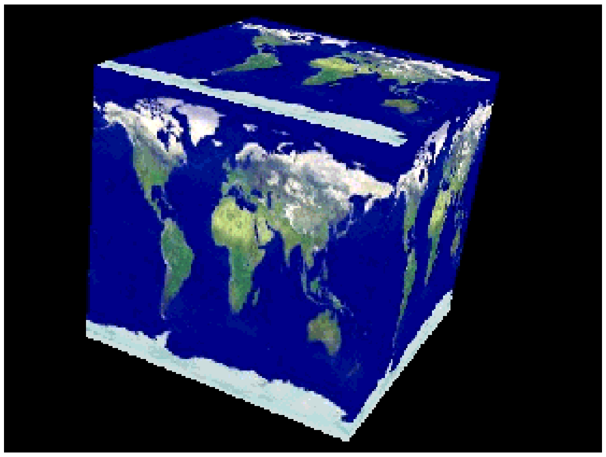
- The **`TextureLoader`** utility class loads an image from a file or URL, and returns an **`ImageComponent`**

Method
`TextureLoader( String path, Component observer )`
`TextureLoader( String path, String format, Component observer )`
`ImageComponent2D getImage( )`
`Texture getTexture( )`

TextureLoader Example Code

```
Appearance app = new Appearance( );
. . .
TextureLoader loader = new TextureLoader( "earth.jpg", this
ImageComponent2D image = loader.getImage( );
. . .
Texture2D tex = new Texture2D( );
tex.setImage( 0, image );
tex.setEnable( true );
. . .
app.setTexture( tex );
```

TextureLoader Example



[**`ExTexture`**]

Summary

Class	***Attributes***
`Texture`	Enable, Image
├ **`Texture2D`**	-
└ **`Texture3D`**	-

Summary

Class	***Attributes***
`ImageComponent`	-
├ **`ImageComponent2D`**	Format, Image
└ **`ImageComponent3D`**	Format, Images
`TextureLoader`	String, Image, Texture

Using Texture Coordinates

Motivation

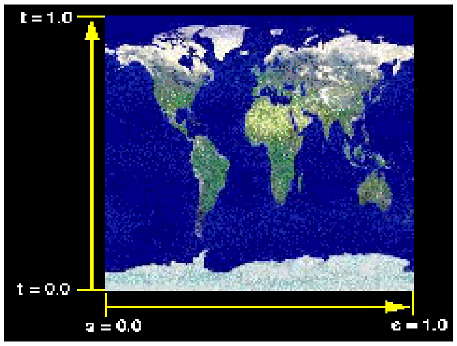
- We need a mapping from portions of a texture image to portions of a shape
  - Placement of an image (translation)
  - Orientation of an image (rotation)
  - Size of an image (scale)
  - Repetition of an image (scale)

Using Texture Coordinates

Texture Images

- Texture images must have dimensions that are a power of 2
  - Width and height don't have to be the *same* power of 2

- Regardless of the real size of the image, **`s`** and **`t`** run from 0.0 to 1.0



Specifying Texture Coordinates

```
TriangleArray ta = new TriangleArray( vertexCount,
    Geometry.COORDINATES |
    Geometry.NORMALS |
    Geometry.TEXTURE_COORDINATE_2 );
ta.setCoordinate( index, new Point3f( x, y, z ) );
ta.setTextureCoordinate( index, new Point2f( s, t ) );
. . .
```

Using Texture Boundary Modes

- The **`Texture`** class controls the boundary mode in S (horizontal) and T (vertical) texture directions
  - **`CLAMP`**
    - Clamp texture coordinates to 0.0 - 1.0
    - Use boundary color for surface parts outside this range
  - **`WRAP`**
    - Use the fractional part of texture coordinates, so that values outside 0.0 - 1.0 repeat
    - Creates a repeating pattern, such as a checkerboard or bricks

Texture Class Components

Node: **`Shape3D`**

```
Geometry
Appearance
  TextureAttributes
  Texture2D
```

- Enable
- Image
- Boundary Modes
- Boundary Color
- . . .

. . .

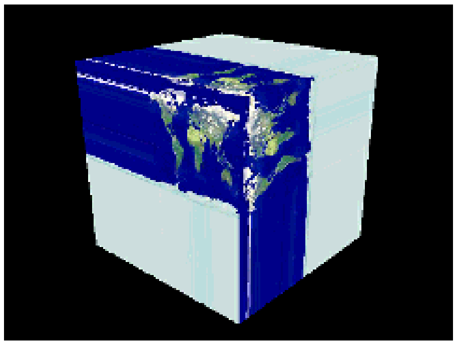
Texture Class Methods

Method	*Default*
`void setBoundaryModeS( int mode )`	**`WRAP`**
`void setBoundaryModeT( int mode )`	**`WRAP`**

Texture Boundary Example

```
Appearance app = new Appearance( );
. . .
TextureLoader loader = new TextureLoader( "earth.jpg", this
ImageComponent2D image = loader.getImage( );
. . .
Texture2D tex = new Texture2D( );
tex.setImage( 0, image );
tex.setEnable( true );
tex.setBoundaryModeS( Texture.CLAMP );
tex.setBoundaryModeT( Texture.CLAMP );
. . .
app.setTexture( tex );
```

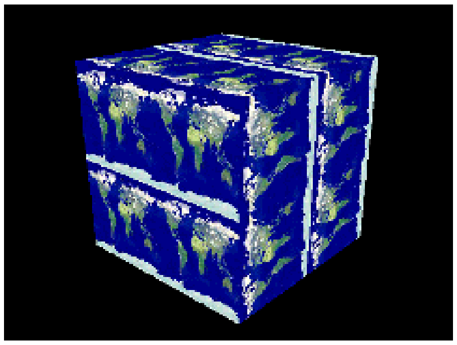
The CLAMP Texture Boundary Mode



```
setBoundaryModeS( Texture.CLAMP );
setBoundaryModeT( Texture.CLAMP );
```

- Any texture coordinate < 0.0 is clamped to 0.0
- Any texture coordinate > 1.0 is clamped to 1.0

The WRAP Texture Boundary Mode



```
setBoundaryModeS( Texture.WRAP );
setBoundaryModeT( Texture.WRAP );
```

- Any texture coordinate < 0.0 is modulo’d with 1.0
- Any texture coordinate > 1.0 is modulo’d with 1.0

Making a Brick Wall with WRAPing

```
QuadArray qa = new QuadArray( nCoordinates,
    Geometry.COORDINATES |
    Geometry.NORMALS |
    Geometry.TEXTURE_COORDINATE_2 );
qa.setCoordinate(        0, new Point3f( 0.f, 0.f, 0.f ) );
qa.setTextureCoordinate( 0, new Point2f( 0.f, 0.f )     );

qa.setCoordinate(        1, new Point3f( 10.f, 0.f, 0.f ) );
qa.setTextureCoordinate( 1, new Point2f( 100.f, 0.f )    );

qa.setCoordinate(        2, new Point3f( 10.f, 5.f, 0.f ) );
qa.setTextureCoordinate( 2, new Point2f( 100.f, 50.f )   );

qa.setCoordinate(        3, new Point3f( 0.f, 5.f, 0.f ) );
qa.setTextureCoordinate( 3, new Point2f( 0.f, 50.f )    );
```

Using Texture Transforms

- A *texture transform* translates, rotates, and scales texture coordinates (s,t)
- Translate to slide an image across a shape
- Rotate to orient an image on a shape
- Scale to repeat an image multiple times across a shape

TextureAttributes Class Hierarchy

- The **`TextureAttributes`** class extends **`NodeComponent`**

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.NodeComponent
         └─ javax.media.j3d.TextureAttributes
```

Texture Attribute Class Components

- Texture transforms are controlled by the **`TextureAttributes`** class as part of shape appearance

Node: **`Shape3D`**

```
Geometry
Appearance
  TextureAttributes
    Texture Transform
    . . .
  Texture2D
  . . .
```

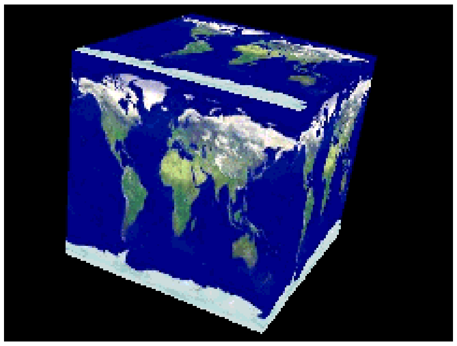
TextureAttributes Class Methods

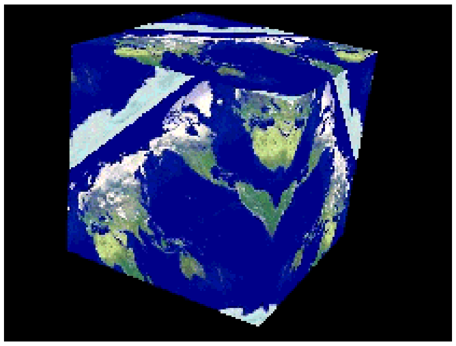
Method	*Default*
`TextureAttributes( )`	-
`void setTextureTransform( Transform3D trans )`	None

Texture Rotation Example Code

```
Appearance app = new Appearance( );
. . .
TextureAttributes tatt = new TextureAttributes( );
Transform3D trans = new Transform3D( );
trans.rotZ( Math.PI/4.0 );
tatt.setTextureTransform( trans );
. . .
app.setTextureAttributes( tatt );
```

Texture Rotation Example


`trans.rotZ(0.0)`

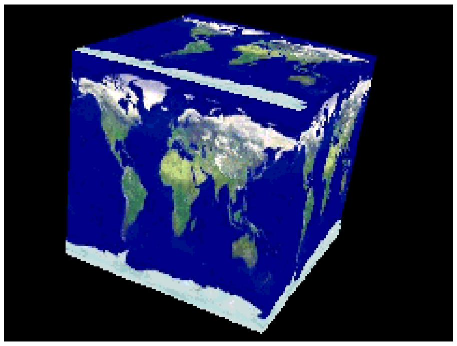

`trans.rotZ(Math.PI/4.0)`

- The boundary mode here is **`WRAP`**

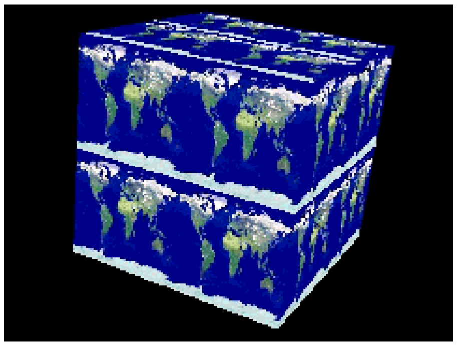
Texture Scaling Example Code

```
Appearance app = new Appearance( );
. . .
TextureAttributes tatt = new TextureAttributes( );
Transform3D trans = new Transform3D( );
trans.setScale( 2.0 );
tatt.setTextureTransform( trans );
. . .
app.setTextureAttributes( tatt );
```

Texture Scaling Example



`trans.setScale( 1.0 )`



`trans.setScale( 2.0 )`

- The boundary mode here is **`WRAP`**

Summary

Class	*Attributes*
`Texture`	Enable, Image, S and T boundary modes
├ **`Texture2D`**	-
└ **`Texture3D`**	R boundary mode
`TextureAttributes`	Texture transform

Controlling the Appearance of Texture Mapping

Controlling the Appearance of Texture Mapping

Motivation

- We’d like to control how texture and shape color are combined
    - Texture filtering
    - Texture Mip-mapping

Texture Image Contents

- A texture can contain information on:
  - Color (red-green-blue)
  - Transparency (alpha)

What Is Alpha Blending?

- Alpha Blending is a linear blending from one value to another as *alpha* goes from 0.0 to 1.0:

```
Value = (1.0-alpha)*Value0 + alpha*Value1
```

- In texturing, it will be used for blending color components
- In RGBA texturing, the texture’s alpha value is sometimes used to specify blending
- In texturing, sometimes the texture’s RGB color will be used as 3 alpha values to give spectral color filtering

Using Texture Modes

- The *Texture mode* in the **`TextureAttributes`** class controls how texture pixels affect shape color:

`REPLACE`	Texture color completely replaces the shape’s material color
`DECAL`	Texture color is blended as a decal on top of the shape’s material color
`MODULATE`	Texture color modulates (filters) the shape’s material color
`BLEND`	Texture color blends the shape’s material color with an arbitrary *blend color*

- Note: the shape’s material color undergoes 3D lighting calculations

Using Texture Modes

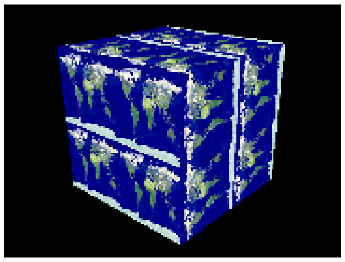
Mode	P'_{rgb}	P'_{a}
REPLACE	T_{rgb}	T_a
DECAL	$P_{rgb}*(1-T_a)+T_{rgb}*T_a$	P_a
MODULATE	$P_{rgb}*T_{rgb}$	P_a*T_a
BLEND	$P_{rgb}*(1-T_{rgb})+B_{rgb}*T_{rgb}$	P_a*T_a

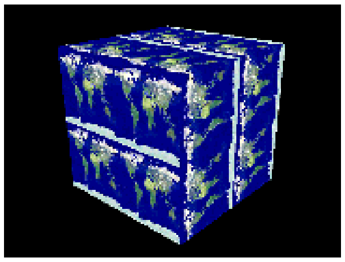
- Where:

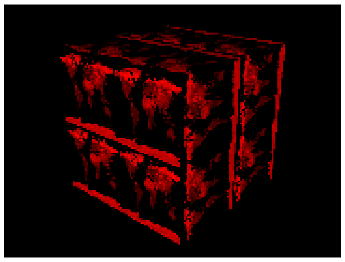
  P_{rgb} is the color of the pixel being texture mapped, as if no texture-mapping was to take place
  P_a is the alpha of the pixel being texture mapped, as if no texture-mapping was to take place
  T_{rgb} is the texture color
  T_a is the texture alpha
  B_{rgb} is the blend color
  B_a is the blend alpha
  Non-primed quantities denote *before*
  Primed quantities denote *after*

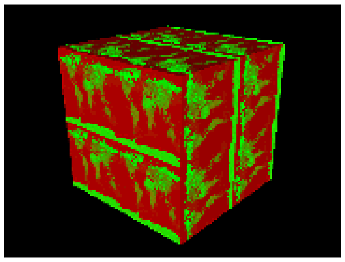
Controlling the Appearance of Texture Mapping

Using Texture Modes


REPLACE


DECAL


MODULATE


BLEND

- Shape color = red
- Blend color = green

Using Texture Mip-map Modes

- *Mip-mapping* is an anti-aliasing technique that uses different texture versions at different distances from the viewer
  - You can have any number of *levels*
  - Level 0 is the base image used when the viewer is close

- Mip-maps can be computed automatically from a base image:
  - Use a mip-mapping mode of **`BASE_LEVEL`**

- *Or* you can specify each image level explicitly:
  - Use a mip-mapping mode of **`MULTI_LEVEL_MIPMAP`**

Using Texture Minification Filters

- A *Minification filter* controls the way the texture is interpolated when a pixel in the scene maps to more than one texel

`FASTEST`	Use fastest method
`NICEST`	Use best looking method
`BASE_LEVEL_POINT`	Use nearest texel in level 0 map
`BASE_LEVEL_LINEAR`	Bilinearly interpolate 4 nearest texels in level 0 map
`MULTI_LEVEL_POINT`	Use nearest texel in mip-mapped maps
`MULTI_LEVEL_LINEAR`	Bilinearly interpolate 4 nearest texels in mip-mapped maps

Using Texture Magnification Filters

- A *Magnification filter* controls the way the texture is interpolated when a pixel in the scene maps to less than one texel

`FASTEST`	Use fastest method
`NICESET`	Use best looking method
`BASE_LEVEL_POINT`	Use nearest texel in level 0 map
`BASE_LEVEL_LINEAR`	Bilinearly interpolate 4 nearest texels in level 0 map

Texture Class Components

- Mip-mapping and filters for a texture image are controlled by the **`Texture`** class

Node: **`Shape3D`**

```
Geometry
Appearance
  TextureAttributes
  Texture2D
    Enable
    Image
    Boundary Modes
    Boundary Color
    Mipmap Mode
    Minification Filters
    Magnification Filters
  . . .
```

Texture Class Methods

Method	*Default*
`void setMipMapMode( int mode )`	`BASE_LEVEL`
`void setMinFilter( int minFilter )`	`BASE_LEVEL_POINT`
`void setMagFilter( int maxFilter )`	`BASE_LEVEL_POINT`

TextureAttributes Class Components

- The texture mode and blend color are controlled by the **`TextureAttributes`** class

Node: **`Shape3D`**

```
Geometry
Appearance
  TextureAttributes
    Texture Transform
    Texture Mode
    Blend Color
    . . .
  Texture2D
  . . .
```

TextureAttributes Class Methods

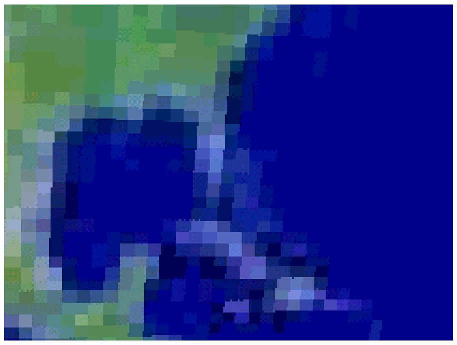
Method	*Default*
`void setTextureMode( int mode )`	**`MODULATE`**
`void setTextureBlendColor( Color4f color )`	0.0 0.0 0.0 0.0

- Texture modes include: **`MODULATE`**, **`DECAL`**, **`BLEND`**, and **`REPLACE`**

Texture Filter Example Code

```
Appearance app = new Appearance( );
. . .
TextureLoader loader = new TextureLoader( "earth.jpg", this
ImageComponent2D image = loader.getImage( );
. . .
Texture2D tex = new Texture2D( );
tex.setImage( 0, image );
tex.setEnable( true );
tex.setBoundaryModeS( Texture.CLAMP );
tex.setBoundaryModeT( Texture.CLAMP );
tex.setMagFilter( Texture.BASE_LEVEL_POINT );
tex.setMinFilter( Texture.BASE_LEVEL_POINT );
. . .
app.setTexture( tex );
```

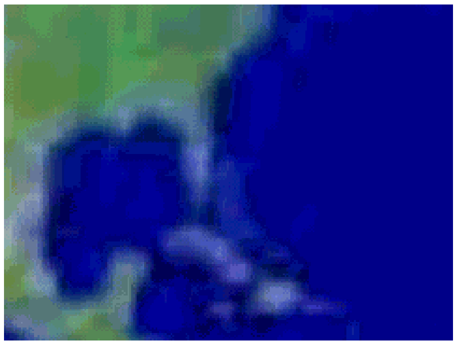
The BASE_LEVEL_POINT Filter



```
setMagFilter( Texture.BASE_LEVEL_POINT );
setMinFilter( Texture.BASE_LEVEL_POINT );
```

- Selects the nearest texel in the level 0 texture map

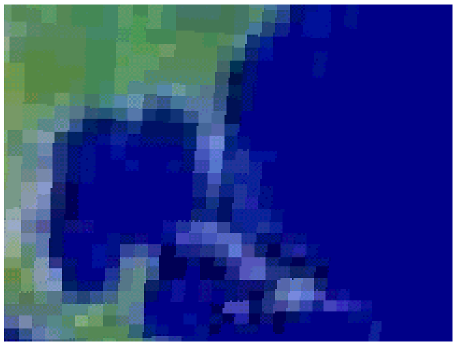
The BASE_LEVEL_LINEAR Filter



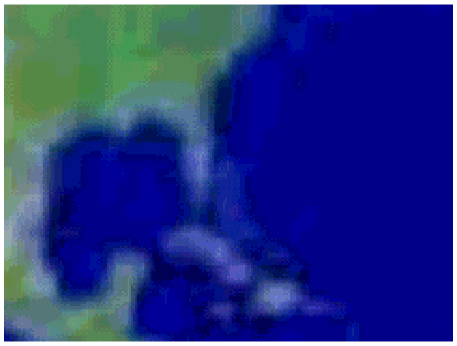
```
setMagFilter( Texture.BASE_LEVEL_LINEAR );
setMinFilter( Texture.BASE_LEVEL_LINEAR );
```

- Interpolates the 4 nearest texels in the level 0 texture map

Summary of Texture Filters



BASE_LEVEL_POINT



BASE_LEVEL_LINEAR

Summary

Class	*Attributes*
`Texture`	Enable, Image, S and T boundary modes, MipMapMode, Minification filter, Magnification Filter
├─ **`Texture2D`**	-
└─ **`Texture3D`**	R boundary mode
`TextureAttributes`	Texture mode, Blend color, Texture transform

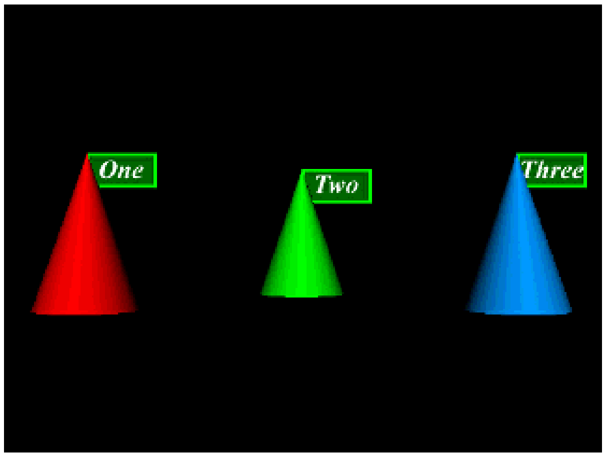
Using Raster Geometry

Motivation

- We’d like to position a 2D image in the 3D scene
  - Anchor it to a 3D point in model coordinates
- Useful for annotation text, sprites, etc.

Using Raster Geometry

Example



[ExRaster]

Raster Class Hierarchy

- The **`Raster`** class extends the **`Geometry`** class

Class Hierarchy

```
java.lang.Object
 └─  javax.media.j3d.SceneGraphObject
      └─  javax.media.j3d.NodeComponent
           └─  javax.media.j3d.Geometry
                └─  javax.media.j3d.Raster
```

Raster Geometry

- **`Raster`** describes geometry for a **`Shape3D`**, including
  - A 3D anchor position
    - Placement of upper-left corner of image
  - An image and its type
    - Color image, depth, or both
  - A region of the image to copy to the screen

Node: **`Shape3D`**

`Raster`
- Image Component
- Depth Component
- Image Type
- 3D Position
- Region Size
- Region Offset

`Appearance`

Raster Class Methods

Method	*Default*
`Raster( )`	**-**
`void setImage( ImageComponent2D image )`	None
`void setDepthComponent( DepthComponent depth )`	None
`void setType( int flag )`	**`RASTER_COLOR`**

- Raster image types include: **`RASTER_COLOR`**, **`RASTER_DEPTH`**, and **`RASTER_COLOR_DEPTH`**

Raster Class Methods

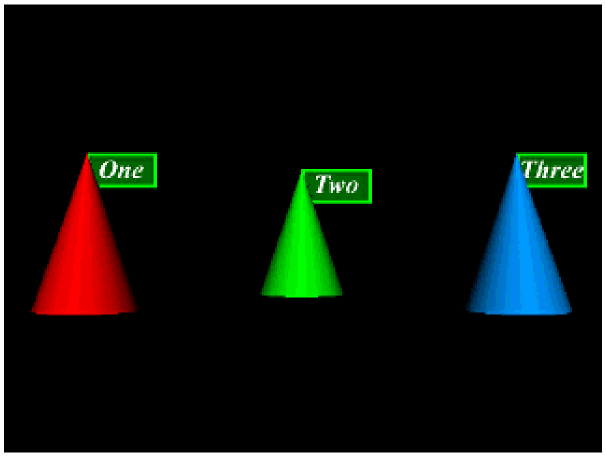
Method	*Default*
`void setPosition( Point3f pos )`	0.0 0.0 0.0
`void setSize( int width, int height )`	0, 0
`void setOffset( int x, int y )`	0, 0

Raster Example Code

```
TransformGroup group = new TransformGroup( );
. . .
Raster raster = new Raster( );
raster.setPosition( new Point3f( 1.0f, 0.0f, 0.0f ) );
raster.setType( Raster.RASTER_COLOR );
raster.setOffset( 0, 0 );
raster.setSize( 256, 256 );
raster.setImage( image );
. . .
Shape3D shape = new Shape3D( );
shape.setGeometry( raster );
shape.setAppearance( app );
group.addChild( shape );
```

Using Raster Geometry

Raster Example



[**`ExRaster`**]

Reading from Raster Geometry

- The screen at the raster geometry location can be read while in *Immediate Mode*

Method
`void readRaster( Raster raster )`

Summary

Class	*Attributes*
`Raster`	Position, Image, Image type, Image region offset and size

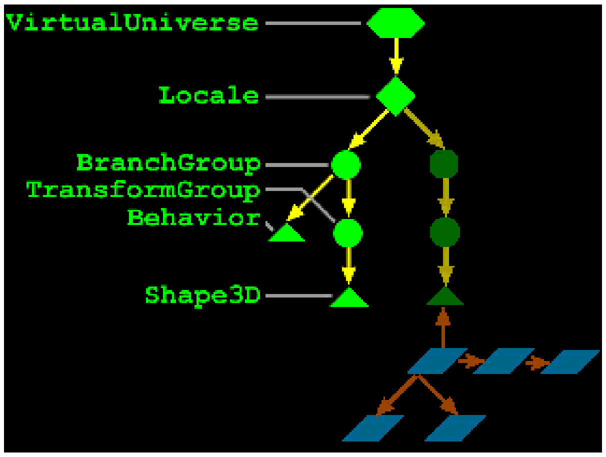
Viewing the Scene

Motivation

- Need to be able to position the viewer in the virtual scene
- Need to be able to handle a multitude of head tracking and display environments
- Viewing is controlled by nodes added to the *View branch* of the scene graph

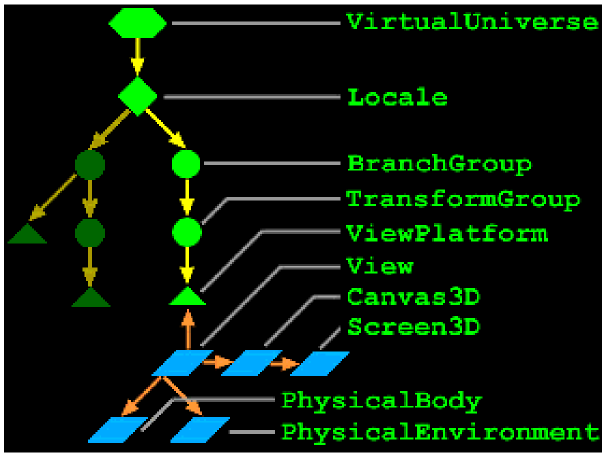
Viewing the Scene

The Content Branch



Viewing the Scene

The View Branch



Terminology

- *VirtualUniverse*
  - A **`VirtualUniverse`** object
  - A container for all 3D content and viewing controls
  - Typically one universe per application

Terminology

- *Locale*
  - A **`Locale`** object
  - An anchor position in the universe at which to attach 3D content
  - There can be multiple locales per universe
  - There can be multiple branches of 3D content per locale

Terminology

- *Branch graph*
  - A scene graph rooted in a **`BranchGroup`** node
  - Typically one branch for viewing-related nodes (the **`ViewPlatform`**, etc.)
  - Typically one branch for 3D content (shapes, sounds, etc.)

VirtualUniverse and Locale Class Hierarchy

- All Superstructure Objects inherit from **`Object`**

Class Hierarchy
`java.lang.Object`
├─ **`javax.media.j3d.VirtualUniverse`**
└─ **`javax.media.j3d.Locale`**

VirtualUniverse Class Methods

- Each universe manages one or more locales

Method
`VirtualUniverse( )`
`Enumeration getAllLocales( )`
`int numLocales( )`

Locale Class Methods

- Every locale is positioned within a single universe

Method
`Locale( VirtualUniverse universe )`
`Locale( VirtualUniverse universe, HiResCoord hiRes )`
`VirtualUniverse getVirtualUniverse( )`
`void setHiRes( HiResCoord hiRes )`

Locale Class Methods

- Every locale manages one or more branch graphs for 3D content and viewing controls

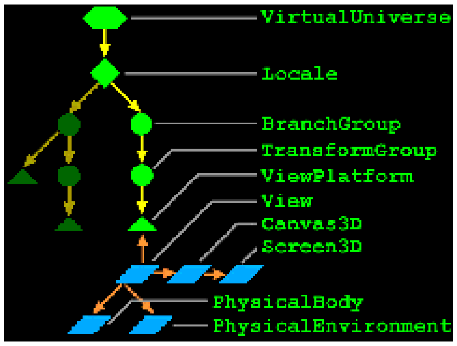
Method
`void addBranchGraph( BranchGroup branchGroup )`
`void removeBranchGraph( BranchGroup branchGroup )`
`void replaceBranchGraph( BranchGroup oldGroup, BranchGroup newGroup )`
`int numBranchGraphs( )`
`Enumeration getAllBranchGraphs( )`

The Java 3D Viewing Model

- Most 3D APIs transform the scene
  - Eg, OpenGL’s **`gluLookAt( )`**
- Java 3D’s viewing model transforms the eye
- The content scene graph isn’t modified when the eye moves
- Cleanly separates the virtual and physical worlds

The Java 3D Viewing Model

- The **`ViewPlatform`** tells from *where* a scene is being viewed
- The **`View`** tells *how* a scene is being viewed



ViewPlatforms

- The **`ViewPlatform`** object locates the viewer
  - It appears as a leaf node of the scene graph

- Translation, rotation, and scaling of the viewer in the virtual universe are controlled by a **`TransformGroup`** above the **`ViewPlatform`** in the scene graph

The Activation Radius

- Each **`ViewPlatform`** has an *activation radius* that expresses a bounded region of interest
  - Behaviors, sounds, backgrounds, fog, and other nodes have bounding volumes
  - When the activation radius intersects those bounds, those nodes are active
    - Backgrounds or fog are turn on
    - Sounds and behaviors are scheduled

ViewPlatform Class Methods

Method	*Default*
`ViewPlatform( )`	-
`void setViewAttachPolicy( int policy )`	**`NOMINAL_ SCREEN_ SCALED`**
`void setActivationRadius( float radius )`	-

- Attach policies include: **`NOMINAL_HEAD`**, **`NOMINAL_FEET`**, **`NOMINAL_SCREEN`**, and **`NOMINAL_SCREEN_SCALED`**

View Class Methods

Method
`void attachViewPlatform( ViewPlatform vp )`
`void setCanvas3D( Canvas3D c3d )`
`void setLocalEyeLightEnable( boolean flag )`
`void setWindowResizePolicy( int policy )`
`void setWindowMovementPolicy( int policy )`

- Resize and movement policies include: **`PHYSICAL_WORLD`**, and **`VIRTUAL_WORLD`**

View Class Methods Relating to Projection

Method	*Default*
`void setProjectionPolicy( int policy )`	**`PARALLEL_ PROJECTION`**
`void setFieldOfView( double fovx )`	Math.PI/4.0 radians

- Projection policies include: **`PARALLEL_PROJECTION`**, and **`PERSPECTIVE_PROJECTION`**

View Class Methods Relating to Clipping

Method
`void setFrontClipPolicy( int policy )`
`void setBackClipPolicy( int policy )`
`void setFrontDistance( double distance )`
`void setBackDistance( double distance )`

- Clip policies include: **`PHYSICAL_EYE`**, **`PHYSICAL_SCREEN`**, **`VIRTUAL_EYE`**, and **`VIRTUAL_SCREEN`**

View Class Methods

Method
`void setDepthBufferFreezeTransparent( boolean flag )`
`void setPhysicalBody( PhysicalBody pb )`
`void setPhysicalEnvironment( PhysicalEnvironment pe )`

PhysicalBody and PhysicalEnvironment Class Methods

Method
`PhysicalBody( )`

Method
`PhysicalEnvironment( )`

Canvas3Ds

- **`Canvas3D`** extends the AWT **`Canvas`** class to support
  - Stereo
  - Double buffering
  - A **`Screen3D`**

Canvas3D Class Methods

Method
`Canvas3D( Configuration gc )`
`boolean getStereoAvailable( )`
`void setStereoEnable( boolean flag )`
`boolean getDoubleBufferAvailable( )`
`void setDoubleBufferEnable( boolean flag )`

Screen3D Class Methods

Method
`void setPhysicalScreenWidth( double width )`
`double getPhysicalScreenWidth( )`
`void setPhysicalScreenHeight( double height )`
`double getPhysicalScreenHeight( )`

Summary

Class	*Attributes*
`VirtualUniverse`	Root of all Java 3D scenes
`Locale`	Basic Placement

Summary

Class	*Attributes*
`ViewPlatform`	Attach policy, Activation radius
`View`	View platform, Canvas3D, Local eye lighting enable, Window resize and movement policy, Projection policy, Field of View, Front and back clip policy and distance, Depth buffer freeze transparent, Physical body, Physical Environment

Summary

Class	*Attributes*
`Canvas3D`	Stereo enable, Double buffer enable
`Screen3D`	-
`PhysicalBody`	-
`PhysicalEnvironment`	-

Creating Behaviors

Motivation

- Programmability
  - Behaviors are just Java methods
- Efficiency
  - Java 3D schedules behaviors to run only when necessary
- Composability
  - Behaviors attached to one set of objects can co-exist with dynamically loaded behaviors controlling other objects
- Pre-support of networked shared worlds
  - Closed function of time interpolators support dead reckoning

Behavior Class Hierarchy

- **`Behavior`** extends the **`Leaf`** class

Class Hierarchy

```
java.lang.Object
 └─  javax.media.j3d.SceneGraphObject
         └─  javax.media.j3d.Node
                 └─  javax.media.j3d.Leaf
                         └─  javax.media.j3d.Behavior
```

Behaviors

- Every behavior contains:
  - An **`initialize`** method

  - Code to run, contained within a **`processStimulus`** method

  - Wakeup conditions controlling when to run next
    - Respecified on each wakeup

  - Scheduling bounds controlling scheduling
    - When the viewer’s activation radius intersects the bounds, the behavior is scheduled

Behaviors

- A behavior can do anything
  - Perform computations
  - Update its internal state
  - Modify the scene graph
  - Start a thread

- For example, a behavior to rotate a radar dish to track an object:
  - Wakeup frequently
  - Get the object's location
  - Create transform to re-orient radar dish
  - Set **`TransformGroup`** of radar dish
  - Return

Behavior Class Methods

Method
`Behavior( )`
`void initialize( )`
`void processStimulus( )`
`void setEnable( boolean onOff )`

Behavior Scheduling Bounds

- A behavior only needs to be scheduled if the viewer is nearby
  - The viewer’s activation radius intersects it’s *scheduling bounds*

- Behavior bounding enables costly behaviors to be skipped if they aren’t nearby

Creating Scheduling Bounds

- A behavior’s scheduling bounds is a bounded volume
  - Sphere, box, polytope, or combination
  - To make a global behavior, use a huge bounding sphere

- By default, behaviors have no scheduling bounds and are never executed!
  - ***Common error:*** forgetting to set scheduling bounds

Anchoring Scheduling Bounds

- A behavior’s bounding volume can be relative to:
    - The behavior’s coordinate system
        - Volume centered on origin
        - As origin moves, so does volume

    - A *Bounding leaf*’s coordinate system
        - Volume centered on leaf node elsewhere in scene graph
        - As that leaf node moves, so does volume
        - If behavior’s origin moves, volume does not

Behavior Class Methods

Method	*Default*
`void setSchedulingBounds( Bounds bounds )`	None
`void setSchedulingBoundingLeaf( BoundingLeaf leaf )`	None

Scheduling Bounds Example Code

- Set bounds relative to the behavior’s coordinate system

```
Behavior behavior = new MyBehavior( );
behavior.setSchedulingBounds( bounds );
```

- Or relative to a bounding leaf’s coordinate system

```
TransformGroup group = new TransformGroup( );
BoundingLeaf leaf = new BoundingLeaf( bounds );
group.addChild( leaf );
. . .
Behavior behavior = new MyBehavior( );
behavior.setSchedulingBoundingLeaf( leaf );
```

Waking up a Behavior

- Even when scheduled, a behavior runs only when *wakeup criterion* are met
  - A number of frames or milliseconds have elapsed
  - A behavior or AWT posts an event
  - A transform changes in a **`TransformGroup`**
  - A shape collides with another shape
  - A view platform or sensor gets close

- Multiple criteria can be and/or-ed to form *wakeup conditions*

WakeupCriterion Class Hierarchy

- **`WakeupCriterion`** extends the **`WakeupCondition`** base class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.WakeupCondition
     └─ javax.media.j3d.WakeupCriterion
         ├─ javax.media.j3d.WakeupOnActivation
         ├─ javax.media.j3d.WakeupOnAWTEvent
         ├─ javax.media.j3d.WakeupOnBehaviorPost
         ├─ javax.media.j3d.WakeupOnCollisionEntry
         ├─ javax.media.j3d.WakeupOnCollisionExit
         ├─ javax.media.j3d.WakeupOnDeactivation
         ├─ javax.media.j3d.WakeupOnElapsedFrames
         ├─ javax.media.j3d.WakeupOnElapsedTime
         ├─ javax.media.j3d.WakeupOnSensorEntry
         ├─ javax.media.j3d.WakeupOnSensorExit
         ├─ javax.media.j3d.WakeupOnTransformChange
         ├─ javax.media.j3d.WakeupOnViewPlatformEntry
         └─ javax.media.j3d.WakeupOnViewPlatformExit
```

WakeupCriterion Class Methods

Method
`WakeupCriterion( )`
`boolean hasTriggered( )`

- Each of the subclasses provide constructors and methods to specify wakeup criterion

AWT Event Wakeup

- A behavior can wakeup on a specified AWT event
  - To use the mouse to rotate geometry:
    - Wake up a behavior on mouse press, release, and drag
    - On each drag event, compute distance mouse has moved since press and map to rotation angle
    - Create a rotation transform and write to a **`TransformGroup`**

Method
`WakeupOnAWTEvent( int AWTid )`
`AWTEvent getAWTEvent( )`

Collision Wakeup

- A behavior can wakeup when a **`Shape3D`**'s geometry:
  - Enters/exits collision with another shape
  - Moves while collided with another shape

- Collision detection can be approximate and fast by using bounding volumes, not geometry
- Or it can be exact, but slower, by using the geometry itself

Method
`WakeupOnCollisionEntry( SceneGraphPath armingpath )`
`WakeupOnCollisionExit( SceneGraphPath armingpath )`
`WakeupOnCollisionMovement( SceneGraphPath armingpath )`
`SceneGraphPath getArmingPath( )`
`SceneGraphPath getTriggeringPath( )`

Proximity Wakeup

- Viewer proximity can wakeup a behavior on:
  - Entry/exit of the `ViewPlatform` in a region

- Sensor proximity can wakeup a behavior in the same way on:
  - Entry/exit of the sensor in a region

Method
`WakeupOnViewPlatformEntry( Bounds region )`
`WakeupOnViewPlatformExit( Bounds region )`
`WakeupOnSensorEntry( Bounds region )`
`WakeupOnSensorExit( Bounds region )`
`Bounds getBounds( )`

Elapsed Time Wakeup

- A behavior can wakeup after a number of elapsed frames or milliseconds

Method
`WakeupOnElapsedFrames( int frameCount )`
`int getElapsedFrameCount( )`

Method
`WakeupOnElapsedTime( long milliseconds )`
`long getElapsedFrameTime( )`

Composing Wakeup Criterion

- A behavior can wake up when a set of criterion occur:
  - Criterion are ANDed and ORed together to form *wakeup conditions*

- For example:
  - Wakeup on any of several AWT events (mouse press, release, or drag)
  - Wakeup on viewer proximity OR after some time has elapsed

Composing Wakeup Criterion

- Wakeup conditions can be complex and changing, for example:
    - In a game, the user must press two buttons within a time limit to open a door
    - Behavior's initial wakeup conditions are:
        - Viewer near button 1 or viewer near button 2
    - After button 1 is pressed, conditions become:
        - Viewer near button 2 or time elapsed
    - If time elapses, conditions revert back to the initial one
    - If button 2 is pressed in time, behavior sends event to wakeup door-opening behavior, then exits without rescheduling

WakeupCondition Class Hierarchy

- **`WakeupCondition`** extends the **`Object`** class

Class Hierarchy

```
java.lang.Object
 └─  javax.media.j3d.WakeupCondition
      ├─  javax.media.j3d.WakeupAnd
      ├─  javax.media.j3d.WakeupAndOfOrs
      ├─  javax.media.j3d.WakeupOr
      └─  javax.media.j3d.WakeupOrOfAnds
```

WakeupCondition Class Methods

Method
`WakeupCondition( )`
`Enumeration allElements( )`
`Enumeration triggeredElements( )`

Creating Behaviors

WakeupCondition Subclass Methods

Method
`WakeupAnd( WakeupCriterion[] conditions )`
`WakeupAndOfOrs( WakeupOr[] conditions )`
`WakeupOr( WakeupCriterion[] conditions )`
`WakeupOrOfAnds( WakeupAnd[] conditions )`

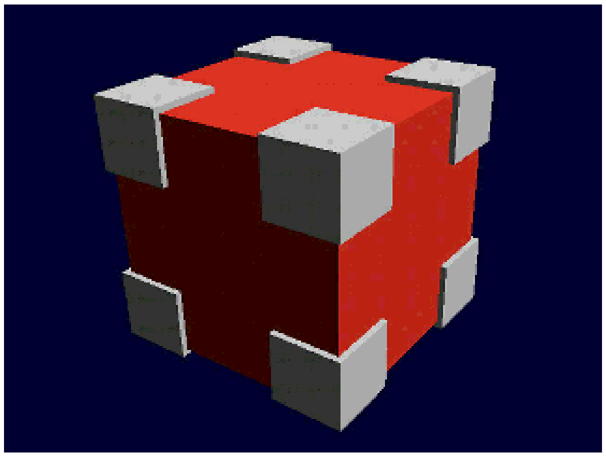
WakeupCondition Example Code

```
Behavior behavior = new MyBehavior( );
. . .
WakeupCriterion[] onMouseEvents =
    new WakeupCriterion[2];

onMouseEvents[0] =
    new WakeupOnAWTEvent( MouseEvent.MOUSE_PRESSED );
onMouseEvents[1] =
    new WakeupOnAWTEvent( MouseEvent.MOUSE_RELEASED );

WakeupCondition onMouse =
    new WakeupOr( onMouseEvents );
. . .
behavior.wakeupOn( onMouse );
behavior.setSchedulingBounds( bounds );
```

WakeupCondition Example



[**`Drag`**]

Summary

Class	*Attributes*
`Behavior`	processStimulus, Enable, Wakeup conditions, Scheduling bounds
`WakeupCondition`	One or more criterion
`WakeupCriterion`	Elapsed time, frames, event to wait for, etc.

Creating Interpolators

Motivation

- Many simple behaviors can be expressed as interpolators
  - Vary a parameter from a starting to an ending value during a time interval
    - Transforms, colors, switches

- Java 3D provides special-case *interpolator* behaviors
  - Enables optimized implementations
  - Since they are closed functions of time, they can be used for dead-reckoning over a network

Interpolator Mapping

- An interpolator uses two mappings:
  - Time-to-*Alpha*
    - *Alpha* is a generalized value that varies from 0.0 to 1.0 over a time interval

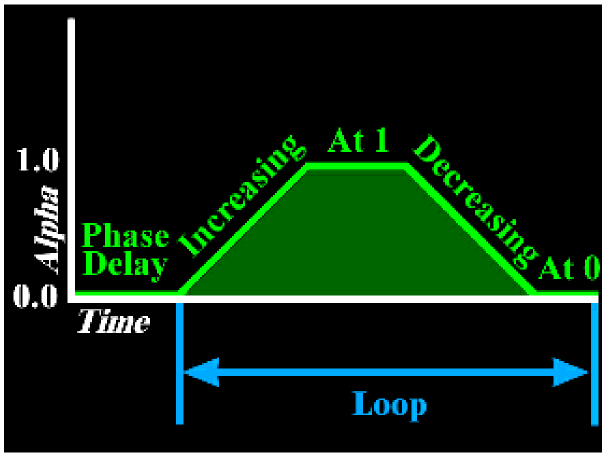
  - Alpha-to-*Value*
    - Different interpolator types map to different values, such as transforms, colors, switches

Mapping Time to Alpha

- *Alpha* is a generalized value that varies from 0.0 to 1.0 over a time interval

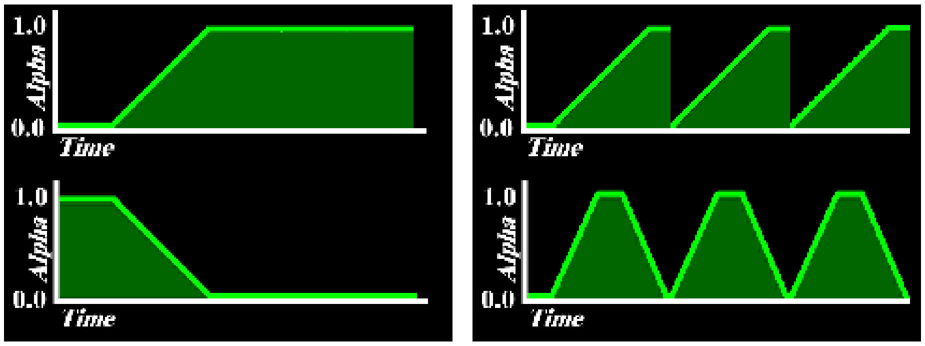
- An *Alpha generator* computes alpha based upon parameters that control
  - Trigger time
  - *Phase Delay* before initial alpha change (attack)
  - *Increasing Ramp* time for increasing alpha
  - *At-One* time for constant high alpha
  - *Decreasing Ramp* time for decreasing alpha
  - *At-Zero* time for constant low alpha

Mapping Time to Alpha



Alpha One-Shot and Cyclic Behaviors

- This model of alpha generalizes to several different types of one-shot and cyclic behaviors



Alpha Class Hierarchy

- **`Alpha`** extends the **`Object`** class

Class Hierarchy
`java.lang.Object` └ **`javax.media.j3d.Alpha`**

Alpha Class Methods

- **`Alpha`** methods construct and control alpha start and looping, or get the current value

Method
`Alpha( )`
`void setLoopCount( int count )`
`void setMode( int mode )`
`void setStartTime( long millisecs )`
`void setTriggerTime( long millisecs )`
`float value( )`
`float value( long millisecs )`

Alpha Class Methods

- Methods set alpha stage durations

Method
`void setAlphaAtOneDuration( long millisecs )`
`void setAlphaAtZeroDuration( long millisecs )`
`void setDecreasingAlphaDuration( long millisecs )`
`void setDecreasingAlphaRampDuration( long millisecs )`
`void setIncreasingAlphaDuration( long millisecs )`
`void setIncreasingAlphaRampDuration( long millisecs )`
`void setPhaseDelayDuration( long millisecs )`

Types of Interpolators

- Simple interpolators map alpha to a value between a start and end value
    - Single transforms
        - **`PositionInterpolator`**, **`RotationInterpolator`** , and **`ScaleInterpolator`**
    - Colors and transparency
        - **`ColorInterpolator`** and **`TransparencyInterpolator`**
    - **`Switch`** group values
        - **`SwitchValueInterpolator`**

Types of Interpolators

- *Path* interpolators map alpha to a value along a path of two or more values
  - Single transforms
    - **`PositionPathInterpolator`** and **`RotationPathInterpolator`**
  - Combined transforms
    - **`RotPosPathInterpolator`** and **`RotPosScalePathInterpolator`**

Interpolator Class Hierarchy

- **`Interpolator`** extends the **`Behavior`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Behavior
                 └─ javax.media.j3d.Interpolator
                     ├─ javax.media.j3d.ColorInterpolator
                     ├─ javax.media.j3d.PathInterpolator
                     │   ├─ javax.media.j3d.PositionPathInterpolator
                     │   ├─ javax.media.j3d.RotationPathInterpolator
                     │   ├─ javax.media.j3d.RotPosPathInterpolator
                     │   └─ javax.media.j3d.RotPosScalePathInterpolat
                     ├─ javax.media.j3d.PositionInterpolator
                     ├─ javax.media.j3d.RotationInterpolator
                     ├─ javax.media.j3d.ScaleInterpolator
                     ├─ javax.media.j3d.SwitchValueInterpolator
                     └─ javax.media.j3d.TransparencyInterpolator
```

Using Interpolators

- All interpolators specify a *target* into which to write new values
  - Transform interpolators use a **`TransformGroup`** target
  - A **`ColorInterpolator`** uses a **`Material`** target
  - And so forth

Interpolator Abstract Class Methods

Method
`Interpolator( )`
`void setAlpha( Alpha alpha )`

Creating Interpolators

RotationInterpolator Class Methods

Method
`RotationInterpolator( Alpha alpha, TransformGroup target )`
`void setAxisOfRotation( Transform3D axis )`
`void setMaximumAngle( float angle )`
`void setMinimumAngle( float angle )`
`void setTarget( TransformGroup target )`

PositionInterpolator Class Methods

Method
`PositionInterpolator( Alpha alpha, TransformGroup target )`
`void setAxisOfTranslation( Transform3D axis )`
`void setEndPosition( float pos )`
`void setStartPosition( float pos )`
`void setTarget( TransformGroup target )`

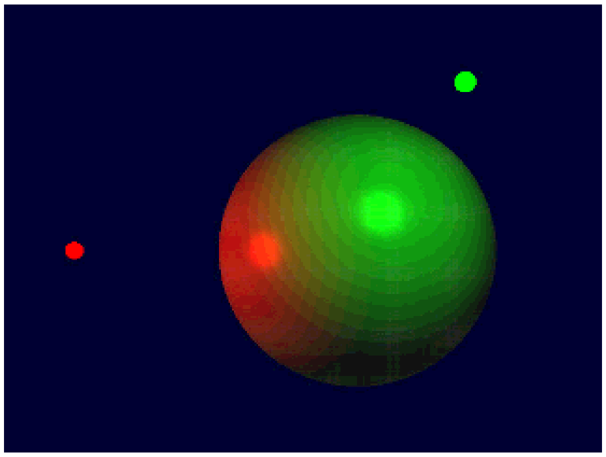
Interpolator Example Code

```
TransformGroup group = new TransformGroup( );
. . .
Alpha upRamp = new Alpha( );
upRamp.setIncreasingAlphaDuration( 10000 );
upRamp.setLoopCount( -1 );  // loop forever

RotationInterpolator spinner =
    new RotationInterpolator( upRamp, group );
spinner.setAxisOfRotation( new Transform3D( ) );
spinner.setMinimumAngle( 0.0f );
spinner.setMaximumAngle( (float)(Math.PI * 2.0) );
. . .
spinner.setSchedulingBounds( bounds );
group.addChild( spinner );
```

Creating Interpolators

Interpolator Example



[**SphereMotion**]

Summary

Class	*Attributes*
`Alpha`	Loop count, Mode, Start time, Trigger time, Stage durations
`Interpolator`	Alpha
├ `ColorInterpolator`	Target, Color range
├ `PositionInterpolator`	Target, Axis, Position range
├ `RotationInterpolator`	Target, Axis, Angle range
├ `ScaleInterpolator`	Target, Axis, Scale range
├ `SwitchValueInterpolator`	Target, Child index range
└ `TransparencyInterpolator`	Target, Transparency range

Summary

Class	*Attributes*
`Interpolator`	Alpha
└ `PathInterpolator`	Value, Knot index
└ `PositionPathInterpolator`	Target, Axis, Positions
└ `RotationPathInterpolator`	Target, Axis, Rotations
└ `RotPosPathInterpolator`	Target, Axis, Rotations, Positions
└ `RotPosScalePathInterpolator`	Target, Axis, Rotations, Positions, Scales

Using Specialized Behaviors

Motivation

- As with interpolators, some behaviors are so common they are provided upfront by Java 3D
  - Billboards
  - Level-of-detail switching

Billboard and LOD Class Hierarchy

- **`Billboard`** and **`LOD`** extend the **`Behavior`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Behavior
                 ├─ javax.media.j3d.Billboard
                 └─ javax.media.j3d.LOD
                     └─ javax.media.j3d.DistanceLOD
```

BillBoard Behaviors

- A *Billboard* is a specialized behavior that:
  - Tracks the **`ViewPlatform`**
  - Generates a rotation about an axis so that the Z-axis points at the platform
  - Writes that transform to a target **`TransformGroup`**

BillBoard Alignment Modes

- Billboard rotation can be about:
    - An object axis to pivot the `TransformGroup`

    - An object point to arbitrarily rotate the `TransformGroup`
        - Rotation makes the group’s Y-axis parallel to the viewer’s Y-axis

Billboard Class Methods

Method	*Default*
`Billboard( )`	None
`void setAlignmentMode( int mode )`	Axis
`void setAlignmentAxis( Vector3f axis )`	Y-axis
`void setRotationPoint( Point3f point )`	0.0 0.0 0.0
`void setTarget( TransformGroup group )`	None

Level-of-Detail Behaviors

- Level-of-Detail (LOD) is a specialized behavior that:
  - Tracks the **`ViewPlatform`**
  - Computes a distance from to a shape
  - Maps the distance to target **`Switch`**(s) choices

- The **`LOD`** abstract class generalizes level-of-detail behaviors

- The **`DistanceLOD`** class implements distance-based switching level-of-detail

Using Specialized Behaviors

LOD Class Methods

Method	*Default*
`LOD( )`	-
`void setSwitch( Switch switch, int index )`	None
`void addSwitch( Switch switch )`	None
`void insertSwitch( Switch switch, int index )`	None
`void removeSwitch( int index )`	None

DistanceLOD Class Methods

Method	*Default*
`DistanceLOD( )`	-
`void setDistance( int whichLOD, double distance )`	None

Picking Shapes

Motivation

- Selection is essential to interactivity
  - Without an ability to select objects you cannot manipulate them

- The picking API enables selecting objects in the scene
  - Various selection shapes
  - First, any, all, all sorted (from closest to furthest)

Overview

- The Java 3D API divides picking into two portions
  - Interface (clicking with a 2D mouse or move a 6DOF wand)
  - Semantics (finding objects that meet the search criteria)

- Enables interchangeable interaction methods

- API designed for speed
  - Only works on bounds
  - Utilities provide more fine-grained support

Where is the API?

- API is distributed among a number of objects
- Enable pickability of any node via methods of the **`Node`** class
- Initiate a pick using methods of the **`Locale`** and **`BranchGroup`** classes
- Pick methods take as an argument a **`PickShape`**:
  - **`PickPoint`**, **`PickRay`**, **`PickSegment`**
- Pick methods return one or more **`SceneGraphPath`** objects

Node Class Pick Methods

- Enable pickability on any node

Method
`void setPickable( boolean onOff )`
`boolean getPickable( )`

Locale and BranchGroup Class Pick Methods

- Initiate a pick using methods of **`Locale`** or **`BranchGroup`**
  - Methods are identical for both classes

Method
`SceneGraphPath pickAll( PickShape pickShape )`
`SceneGraphPath pickAllSorted( PickShape pickShape )`
`SceneGraphPath pickAny( PickShape pickShape )`
`SceneGraphPath pickClosest( PickShape pickShape )`

Types of PickShapes

- Picking intersects a **`PickShape`** with pickable shape bounding volumes

- **`PickPoint`** checks the scene at a position
  - Pick occurs for shape bounds that contain the position

- **`PickRay`** fires a ray from a position, in a direction
  - Pick occurs for shape bounds the ray strikes

- **`PickSegment`** fires a ray along a ray segment between two positions
  - Pick occurs for shape bounds the ray segment intersects

PickShape Abstract Class Hierarchy

- **`PickShape`** and its subclasses extend the **`Object`** class

Class Hierarchy

```
java.lang.Object
 └─   javax.media.j3d.PickShape
       ├─   javax.media.j3d.PickPoint
       ├─   javax.media.j3d.PickRay
       └─   javax.media.j3d.PickSegment
```

PickShape Abstract Class Methods

Method
`PickShape( )`

- The **`PickShape`** abstract class has no methods
- The three pick shape types extend **`PickShape`**

PickPoint Class Methods

Method
`PickPoint( )`
`PickPoint( Point3d pos )`
`void set( Point3d pos )`

PickRay Class Methods

Method
`PickRay( )`
`PickRay( Point3d pos, Vector3d dir )`
`void set( Point3d pos, Vector3d dir )`

PickSegment Class Methods

Method
`PickSegment( )`
`PickSegment( Point3d start, Point3d end )`
`void set( Point3d start, Point3d end )`

Getting Pick Results

- The pick methods on **`Locale`** or **`BranchGroup`** return one or more **`SceneGraphPath`**s

- Each **`SceneGraphPath`** contains:
  - A **`Node`** for the object that was picked
  - The **`Locale`** above it in the scene graph
  - A list of the **`Node`**s from the picked object up to the **`Locale`**
  - The world-to-object transform

SceneGraphPath Class Hierarchy

- **`SceneGraphPath`** extends the **`Object`** class

Class Hierarchy
`java.lang.Object` └ **`javax.media.j3d.SceneGraphPath`**

SceneGraphPath Class Methods

Method
`SceneGraphPath( )`
`Node getObject( )`
`Locale getLocale( )`
`Node getNode( int index )`
`int nodeCount( )`
`Transform3D getTransform( )`

Picking Example Using the Mouse

- Create a behavior that wakes up on a mouse press event
  - On mouse press:
    - Construct a **`PickRay`** from the eye passing through the 2D mouse screen point
    - Wait for mouse drag or mouse release event

Picking Example Using the Mouse

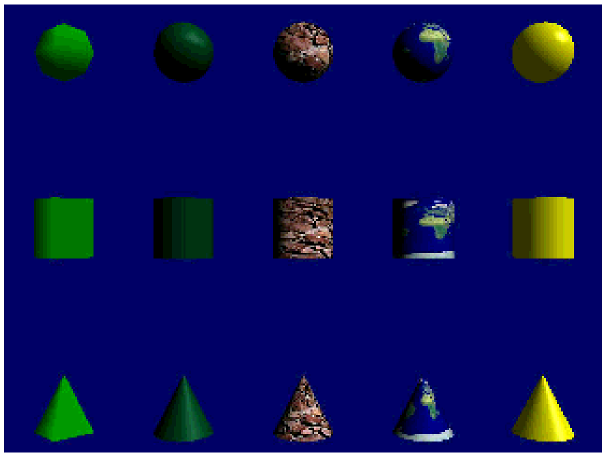
- Continuing...
  - On mouse drag:
    - Update ray to pass through new 2D mouse point
    - Wait for mouse drag or mouse release event

  - On mouse release:
    - Update ray to pass thorugh last 2D mouse point
    - Use Java 3D pick method to generate the first object intersected by the ray
    - Get the pick object and do something to it
    - (Re)declare interest in mouse press events

Picking Example Code

```
PickRay ray = new PickRay( origin, direction );
SceneGraphPath results = locale.pickAllSorted( ray );
Node pickedObject = results.getObject( );
```

Picking Example



[**`PickWorld`**]

Summary

Class	*Attributes*
`Node`	Enable pickability
`Locale`	Request a pick
`BranchGroup`	Request a pick
`PickShape`	-
├ **`PickPoint`**	Position
├ **`PickRay`**	Position, Direction
└ **`PickSegment`**	Start and end positions
`SceneGraphPath`	Object, Locale, Node list, Transform

Using Input Devices

Motivation

- Access to real-time input devices
  - Joysticks
  - Six-degree-of-freedom devices (6DOF) such as a Polhemus, Bird, SpaceBall, Magellan, Ultrasonic tracker, *etc*.
  - Access to button values and knob positions.
- Enables interchangeable input devices
- Painless integration of new input devices with existing applications

Input Device Components

- An implementation of the **`InputDevice`** interface provides:
  - An abstract description of a continuous device
  - Initialization, prompt for a value, get a value, close, *etc.*

- Each **`InputDevice`** implementation maps physical detectors onto **`Sensor`** objects

- Devices can be:
  - real (trackers, network values)
  - virtual (retrieved from a file, computationally generated)

InputDevice Interface Methods

Method
`void initialize( )`
`void close( )`
`void processStreamInput( )`
`void pollAndProcessInput( )`
`void setProcessingMode( int mode )`
`int getSensorCount( )`
`Sensor getSensor( int sensorIndex )`

Sensors

- **`Sensor`** represents an abstract 6DOF input and any associated buttons/knobs
  - Contains the last *k* read values, as **`SensorRead`** objects

- Each **`SensorRead`** contains:
  - A time-stamp
  - A 6DOF value
  - The button states

- A sensor can return a **`Transform3D`** that can be written directly to a **`TransformGroup`**

Sensors

- Sensor *prediction policies* enable a sensor to predict a future value assuming:
  - The sensor is associated with a hand (a data glove, etc.)
  - The sensor is associated with a head (HMD, etc.)

Sensor Class Hierarchy

- **`Sensor`** extends the **`Object`** class

Class Hierarchy
`java.lang.Object` └ **`javax.media.j3d.Sensor`**

Sensor Class Methods

Method
`Sensor( InputDevice device )`
`InputDevice getDevice( )`
`void setDevice( InputDevice device )`
`int getSensorButtonCount( )`

Sensor Class Methods

- Get sensor input

Method
`SensorRead getCurrentSensorRead( )`
`int getSensorReadCount( )`
`void lastRead( Transform3D read )`
`void lastRead( Transform3D read, int kth )`
`int lastButtons( )`
`int lastButtons( int kth )`
`long lastTime( )`
`long lastTime( int kth )`

Sensor Class Methods

- Use prediction and get a predicted value

Method
`void setPredictionPolilcy( int policy )`
`void setPredictor( int predictor )`
`void getRead( Transform3D read )`
`void getRead( Transform3D read, long deltaT )`

- Prediction policies include: **`PREDICT_NONE`** and **`PREDICT_NEXT_FRAME_TIME`**
- Predictors include: **`NO_PREDICTOR`**, **`HEAD_PREDICTOR`**, and **`HAND_PREDICTOR`**

SensorRead Class Hierarchy

- **`SensorRead`** extends the **`Object`** class

Class Hierarchy
`java.lang.Object` └─ **`javax.media.j3d.SensorRead`**

SensorRead Class Methods

Method
`SensorRead( )`
`void get( Transform3D result )`
`int getButtons( )`
`long getTime( )`

Summary

Class	***Attributes***
`InputDevice`	Process input, Return sensor
`Sensor`	Get transform, time, and button state, Use predictor
`SensorRead`	Get transform, time, and button state

Lighting the environment

Motivation

- Previous examples have used a default light attached to the viewer’s head

- You can add your own lights to the scene
  - Use lights to highlight features or add realism
  - Suns, light bulbs, flashlights, spotlights

Example

[**ExHenge**]

Types of Lights

- Java 3D provides four types of lights:
  - Ambient
  - Directional
  - Point
  - Spot

- All lights share common attributes:
  - An on/off enable state
  - A color
  - A bounding volume and scope controlling the range of shapes they illuminate

Light Class Hierarchy

- All lights share attributes inherited from the **`Light`** abstract class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Light
                 ├─ javax.media.j3d.AmbientLight
                 ├─ javax.media.j3d.DirectionalLight
                 └─ javax.media.j3d.PointLight
                     └─  javax.media.j3d.SpotLight
```

Light Abstract Class Methods

Method	*Default*
`void setEnable( boolean OnOff )`	true
`void setColor( Color3f color )`	1.0, 1.0, 1.0

Ambient Lights

- **`AmbientLight`** extends the **`Light`** class
  - Light rays aim in all directions, flooding an environment and illuminating shapes evenly

- Use ambient lights to raise the overall light level
  - Low ambient light makes artificial contrasty imagery
  - High ambient light looks washed out

- Usually one ambient light in the scene is enough

AmbientLight Class Hierarchy

- **`AmbientLight`** extends the **`Light`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Light
                 ├─ javax.media.j3d.AmbientLight
                 ├─ javax.media.j3d.DirectionalLight
                 └─ javax.media.j3d.PointLight
                     └─  javax.media.j3d.SpotLight
```

AmbientLight Class Methods

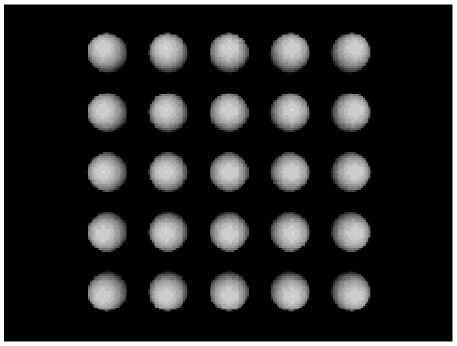
Method	*Default*
`AmbientLight( )`	-

- **`AmbientLight`** adds no additional methods beyond those of the **`Light`** class

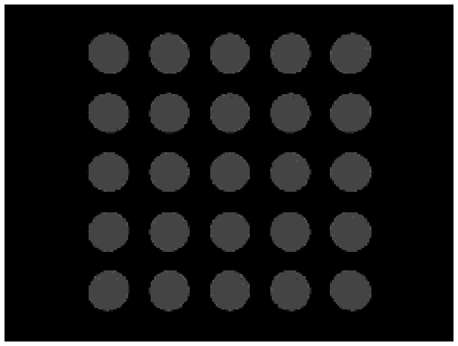
AmbientLight Example Code

```
TransformGroup group = new TransformGroup( );
. . .
AmbientLight light = new AmbientLight( );
light.setEnable( true );
light.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
. . .
light.setInfluencingBounds( bounds );
group.addChild( light );
```

AmbientLight Example



Ambient light + Headlight

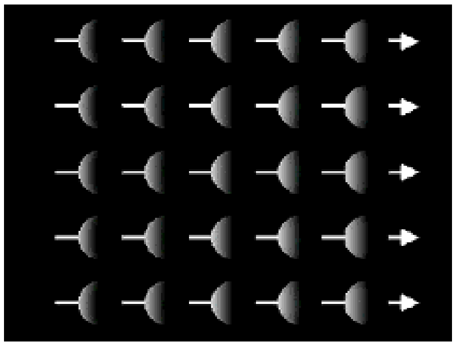


Ambient light only

[**`ExAmbientLight`**]

Directional Lights

- `DirectionalLight` extends the `Light` class
  - Light rays are parallel and aim in one direction
- Use directional lights to simulate distant lights, like the Sun



DirectionalLight Class Hierarchy

- **`DirectionalLight`** extends the **`Light`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Light
                 ├─ javax.media.j3d.AmbientLight
                 ├─ javax.media.j3d.DirectionalLight
                 └─ javax.media.j3d.PointLight
                     └─  javax.media.j3d.SpotLight
```

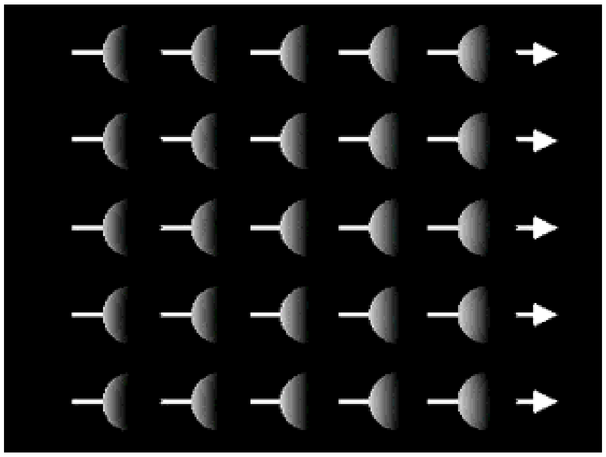
DirectionalLight Class Methods

Method	*Default*
`DirectionalLight( )`	-
`void setDirection( Vector3f dir )`	0.0, 0.0, -1.0

DirectionalLight Example Code

```
TransformGroup group = new TransformGroup( );
. . .
DirectionalLight light = new DirectionalLight( );
light.setEnable( true );
light.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
light.setDirection( new Vector3f( 1.0f, 0.0f, 0.0f ) );
. . .
light.setInfluencingBounds( bounds );
group.addChild( light );
```

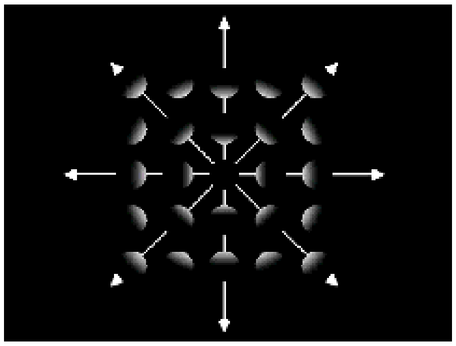
DirectionalLight Example



[**`ExDirectionalLight`**]

Point Lights

- **`PointLight`** extends the **`Light`** class
  - Light rays emit radially from a point in all directions
- Use point lights to simulate local lights, like light bulbs



Point Light Attenuation

- Point light rays are *attenuated*:
  - As distance increases, light brightness decreases

- Attenuation is controlled by three coefficients:
  - *constant*, *linear*, and *quadratic*

$$\text{brightness} = \frac{\text{lightIntensity}}{\text{constant} + \text{linear}*\text{distance} + \text{quadratic}*\text{distance}^2}$$

PointLight Class Hierarchy

- **`PointLight`** extends the **`Light`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Light
                 ├─ javax.media.j3d.AmbientLight
                 ├─ javax.media.j3d.DirectionalLight
                 └─ javax.media.j3d.PointLight
                     └─ javax.media.j3d.SpotLight
```

Lighting the environment

PointLight Class Methods

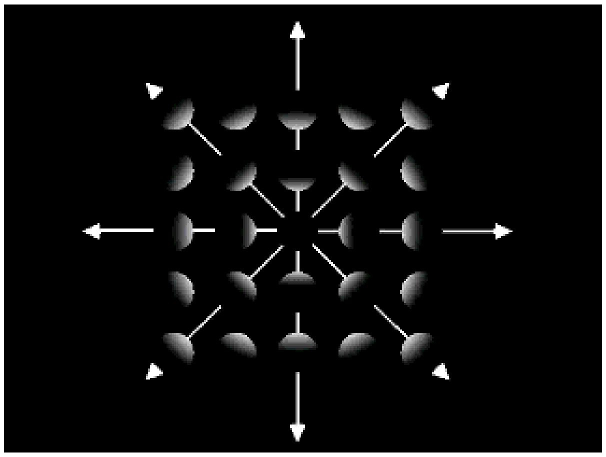
Method	*Default*
`PointLight( )`	-
`void setPosition( Point3f pos )`	0.0, 0.0, 0.0
`void setAttenuation( Point3f atten )`	1.0, 0.0, 0.0

PointLight Example Code

```
TransformGroup group = new TransformGroup( );
. . .
PointLight light = new PointLight( );
light.setEnable( true );
light.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
light.setPosition( new Point3f( 0.0f, 1.0f, 0.0f ) );
light.setAttenuation( new Point3f( 1.0f, 0.0f, 0.0f ) );
. . .
light.setInfluencingBounds( bounds );
group.addChild( light );
```

Lighting the environment

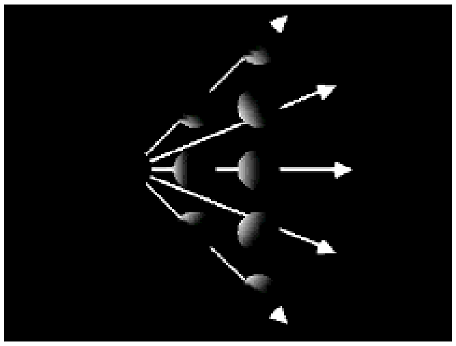
PointLight Example



[**ExPointLight**]

Spot Lights

- `SpotLight` extends the `PointLight` class
  - Light rays emit radially from a point, within a cone
- Vary the *spread angle* to widen, or narrow the cone
- Vary the *concentration* to focus the spot light



SpotLight Class Hierarchy

- **`SpotLight`** extends the **`PointLight`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Light
                 ├─ javax.media.j3d.AmbientLight
                 ├─ javax.media.j3d.DirectionalLight
                 └─ javax.media.j3d.PointLight
                     └─  javax.media.j3d.SpotLight
```

SpotLight Class Methods

Method	*Default*
`SpotLight( )`	-
`void setDirection( Vector3f dir )`	0.0, 0.0, -1.0
`void setSpreadAngle( float angle )`	PI
`void setConcentration( float concen )`	0.0

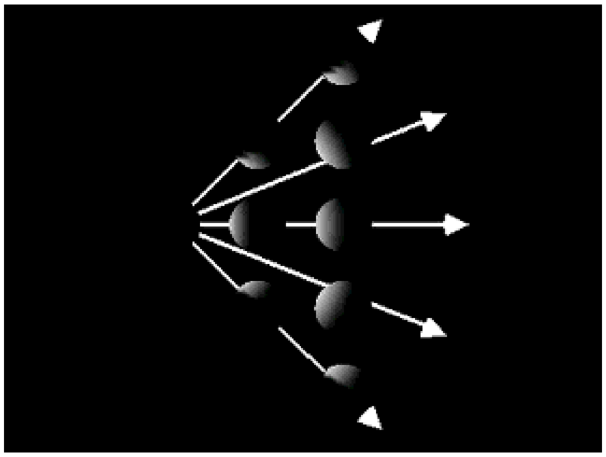
- Spread angle varies from 0.0 to PI/2.0 radians
    - A value of PI radians makes the light a **`PointLight`**
- Concentrations vary from 0.0 (unfocused) to 128.0 (focused)

SpotLight Example Code

```
TransformGroup group = new TransformGroup( );
. . .
SpotLight light = new SpotLight( );
light.setEnable( true );
light.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
light.setPosition( new Point3f( 0.0f, 1.0f, 0.0f ) );
light.setAttenuation( new Point3f( 1.0f, 0.0f, 0.0f ) );
light.setDirection( new Vector3f( 1.0f, 0.0f, 0.0f ) );
light.setSpreadAngle( 0.785f );  // 45 degrees
light.setConcentration( 0.0f );  // Unfocused
. . .
light.setInfluencingBounds( bounds );
group.addChild( light );
```

Lighting the environment

SpotLight Example



[**`ExSpotLight`**]

Light Influencing Bounds

- A light’s illumination is *bounded* to a region of influence
  - Shapes within the region may be lit by the light

- Light bounding:
  - Enables controlled lighting in large scenes
  - Avoids over-lighting a scene when using multiple lights
  - Saves lighting computation time

Creating Influencing Bounds

- A light region of influence is a bounded volume:
  - Sphere, box, polytope, or combination using **`Bounds`**
  - To make a global light, use a huge bounding sphere

- By default, lights have no influencing bounds and illuminate nothing!
  - ***Common error:*** forgetting to set influencing bounds

Anchoring Influencing Bounds

- A light bounding volume can be relative to:
    - The light’s coordinate system
        - Volume centered on light
        - As light moves, so does volume

    - A *Bounding leaf*’s coordinate system
        - Volume centered on a leaf node elsewhere in scene graph
        - As that leaf node moves, so does volume
        - If light moves, volume does not

Light Abstract Class Methods

Method	*Default*
`void setInfluencingBounds( Bounds bounds )`	None
`void setInfluencingBoundingLeaf( BoundingLeaf leaf )`	None

Influencing Bounds Example Code

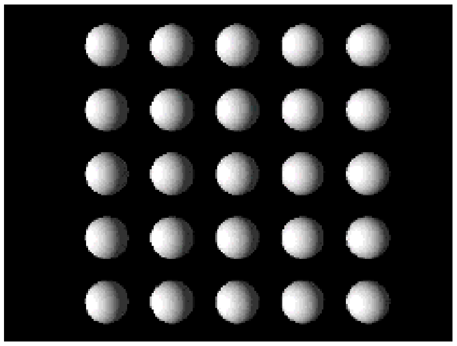
- Set bounds relative to the light’s coordinate system

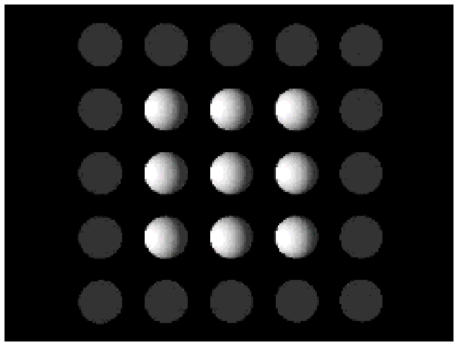
```
PointLight light = new PointLight( );
light.setInfluencingBounds( bounds );
```

- Or relative to a bounding leaf’s coordinate system

```
TransformGroup group = new TransformGroup( );
BoundingLeaf leaf = new BoundingLeaf( bounds );
group.addChild( leaf );
. . .
PointLight light = new PointLight( );
light.setInfluencingBoundingLeaf( leaf );
```

Influencing Bounds Example


Large bounds


Small bounds

[**`ExLightBounds`**]

Scoping Lights

- A light’s illumination may be *scoped* to one or more *groups* of shapes
  - Shapes within the influencing bounds *and* within those groups are lit

- By default, lights have *universal scope* and illuminate everything within their influencing bounds

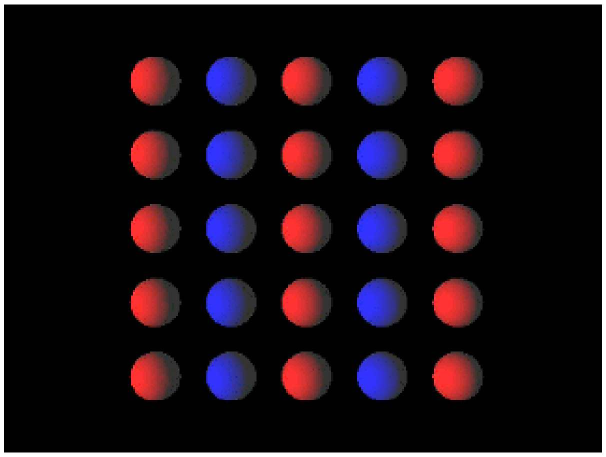
Light Abstract Class Methods

Method	*Default*
`void setScope( Group group, int index )`	None
`void addScope( Group group )`	-
`void insertScope( Group group, int index )`	-
`void removeScope( int index )`	-

Scoping Example Code

```
TransformGroup lightable = new TransformGroup( );
. . .
DirectionalLight light = new DirectionalLight( );
light.addScope( lightable );
```

Scoping Example



[**`ExLightScope`**]

Summary

Class	*Attributes*
`Light`	Enable, Color, Influencing bounds, Scope
├ `AmbientLight`	-
├ `DirectionalLight`	Direction
└ `PointLight`	Position, Attenuation
└ `SpotLight`	Direction, Spread Angle, Concentration

Summary

- Lights illuminate shapes within their influencing bounds
  - Default is *no influence*, so nothing is illuminated!
- *and* within groups on the light's scope list
  - Default is *universal scope*, so everything is illuminated (if within influencing bounds)

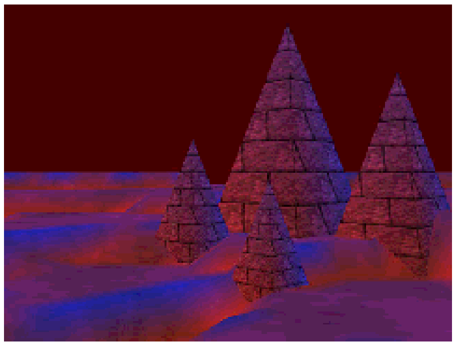
Creating Backgrounds

Motivation

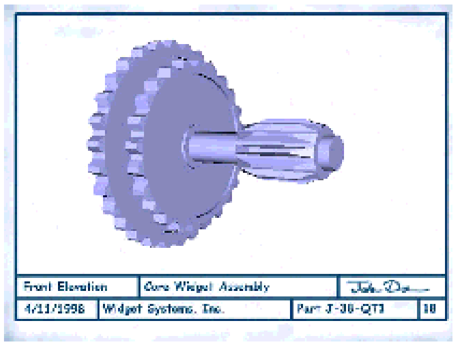
- You can add a *background* to provide context for foreground content

- Use backgrounds to:
    - Set a sky color
    - Add clouds, stars, mountains, city skylines
    - Create an environment map

Example



[**ExBackgroundColor**]



[**ExBluePrint**]

Types of Backgrounds

- Java 3D provides three types of backgrounds:
  - Constant color
  - Flat Image
  - Geometry

- All types are built with a **`Background`** node with:
  - A color, image, or geometry
  - A bounding volume controlling when the background is activated

Background Class Hierarchy

- All background features are controlled using the **`Background`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Background
```

Background Colors

- A **`Background`** node can set a single background color
  - Fills canvas with the color
  - Same color for all viewing directions and lighting levels
- If you want a color gradient, use background geometry

Background Images

- A **`Background`** node can set a background image
  - Fills canvas with the image
  - Image upper-left is at the canvas upper-left
    - To fill the canvas, use an image the size of the canvas
    - Image overrides background color

  - Same image for all viewing directions and lighting levels

- If you want an environment map, use background geometry

Background Geometry

- A **`Background`** node can set background geometry
  - Geometry surrounds the viewer at an "infinite" distance
    - As the viewer turns, they see different parts of the geometry
    - The viewer can never move closer to the geometry
  - Geometry should be on a unit sphere
  - The geometry is not lit by scene lights

- Use background geometry to:
  - Create sky and ground color gradients
  - Build mountain or city skylines
  - Do environment maps (ala QuickTimeVR)

Creating Backgrounds

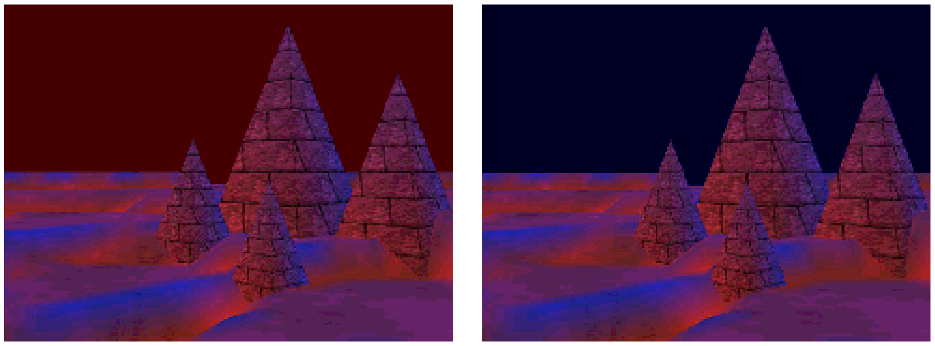
Background Class Methods

Method	*Default*
`Background( )`	-
`void setColor( Color3f color )`	0.0, 0.0, 0.0
`void setImage( ImageComponent2D image )`	None
`void setGeometry( BranchGroup group )`	None

Background Color Example Code

```
TransformGroup group = new TransformGroup( );
. . .
Background back = new Background( );
back.setColor( new Color3f( 0.3f, 0.0f, 0.0f ) );
. . .
back.setApplicationBounds( bounds );
group.addChild( back );
```

Background Color Example

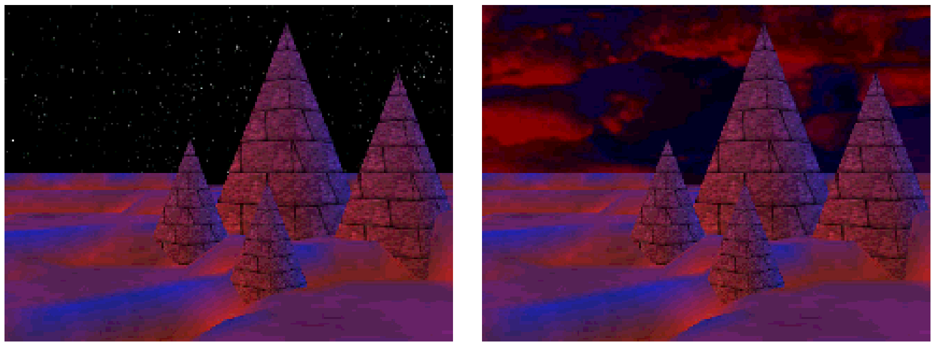


[**`ExBackgroundColor`**]

Background Image Example Code

```
TransformGroup group = new TransformGroup( );
. . .
TextureLoader loader = new TextureLoader( "stars2.jpg", this
ImageComponent2D image = loader.getImage( );
. . .
Background back = new Background( );
back.setImage( image );
. . .
back.setApplicationBounds( bounds );
group.addChild( back );
```

Background Image Example



[**`ExBackgroundImage`**]

Background Geometry Example Code

```
TransformGroup group = new TransformGroup( );
. . .
BranchGroup branch = createBackground( );
. . .
Background back = new Background( );
back.setGeometry( branch );
. . .
back.setApplicationBounds( bounds );
group.addChild( back );
```

Background Application Bounds

- A background is applied when:
  - The viewer's activation radius intersects it's *application bounds*
  - If multiple backgrounds are active, the closest is used
  - If no backgrounds are active, background is black

- Background bounding enables different backgrounds for different areas of the scene

Creating Application Bounds

- A background’s application bounds is a bounded volume
  - Sphere, box, polytope, or combination
  - To make a global background, use a huge bounding sphere

- By default, backgrounds have no application bounds and are never applied!
  - ***Common error:*** forgetting to set application bounds

Anchoring Application Bounds

- A background bounding volume can be relative to:
  - The background’s coordinate system
    - Volume centered on origin
    - As origin moves, so does volume

  - A *Bounding leaf*’s coordinate system
    - Volume is centered on leaf node elsewhere in scene graph
    - As that leaf node moves, so does volume
    - If background origin moves, volume does not

Background Class Methods

Method	*Default*
`void setApplicationBounds( Bounds bounds )`	None
`void setApplicationBoundingLeaf( BoundingLeaf leaf )`	None

Application Bounds Example Code

- Set bounds relative to the background’s coordinate system

```
Background back = new Background( );
back.setApplicationBounds( bounds );
```

- Or relative to a bounding leaf’s coordinate system

```
TransformGroup group = new TransformGroup( );
BoundingLeaf leaf = new BoundingLeaf( bounds );
group.addChild( leaf );
. . .
Background back = new Background( );
back.setApplicationBoundingLeaf( leaf );
```

Summary

Class	*Attributes*
`Background`	Color, Image, Geometry, Application bounds

- Backgrounds are activated when the viewer’s activation radius intersects the background’s application bounds

Working with Fog

Motivation

- Fog increases realism and declutters a scene:
  - Add gray fog to create virtual weather effects
  - Add black fog to create *depth cueing*

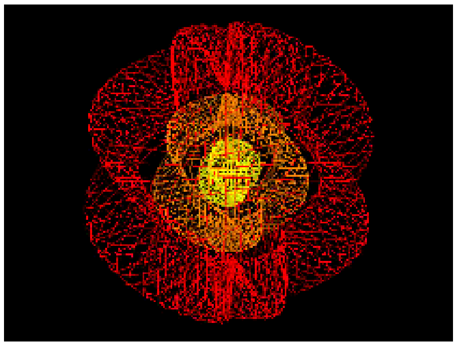
- The further the viewer can see, the more you have to model and draw

- To reduce development time and drawing time, limit the viewer's sight by using fog

Working with Fog

Example

[**`ExExponentialFog`**]



[**`ExDepthCue`**]

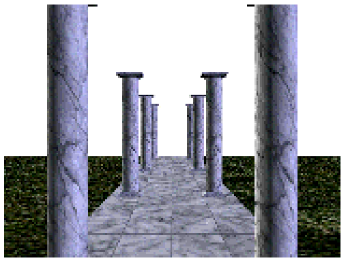
Types of Fog

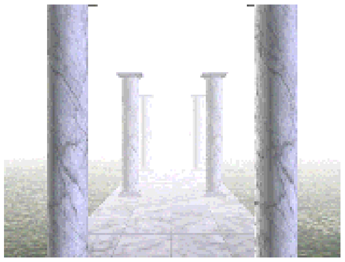
- Java 3D provides two types of fog:
  - Exponential
  - Linear

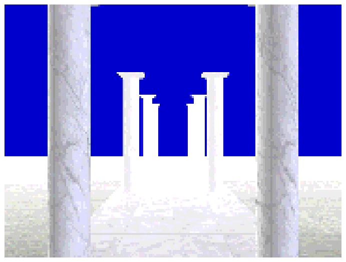
- Both types of fog have:
  - A color
  - A bounding volume and scope controlling the range of shapes to affect

Understanding Fog Effects

- Fog affects shape color, *not* shape profile
  - Distant shapes have the fog color, but still have crisp profiles
- Set the background color to the fog color or your scene will look odd!


No fog


Light fog


Fog on Background

Fog Class Hierarchy

- All fog types share attributes inherited from the **`Fog`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Fog
                 ├─ javax.media.j3d.ExponentialFog
                 └─ javax.media.j3d.LinearFog
```

Fog Class Methods

Method	*Default*
`void setColor( Color3f color )`	0.0, 0.0, 0.0

Exponential Fog

- **`ExponentialFog`** extends the **`Fog`** class
  - Thickness increases exponentially with distance

- Use exponential fog to create thick, realistic fog

- Vary fog *density* to control thickness

  effect = $e^{(\text{-density} * \text{distance})}$
  color = effect * shapeColor + (1-effect) * fogColor

ExponentialFog Class Hierarchy

- **`ExponentialFog`** extends the **`Fog`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Fog
                 ├─ javax.media.j3d.ExponentialFog
                 └─ javax.media.j3d.LinearFog
```

ExponentialFog Class Methods

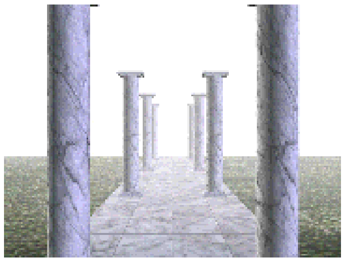
Method	*Default*
`ExponentialFog( )`	-
`void setDensity( float density )`	1.0

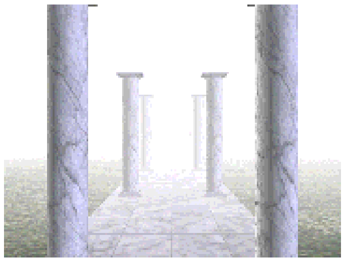
- Fog density varies from 0.0 (no fog) and up (denser fog)

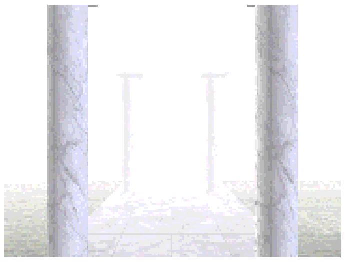
ExponentialFog Example Code

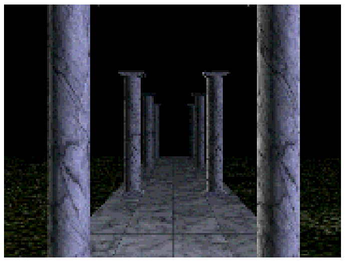
```
TransformGroup group = new TransformGroup( );
. . .
ExponentialFog fog = new ExponentialFog( );
fog.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
fog.setDensity( 1.0f );
. . .
fog.setInfluencingBounds( bounds );
group.addChild( fog );
```

ExponentialFog Example


Haze


Light fog


Heavy fog


Black fog

[**`ExExponentialFog`**]

Linear Fog

- **`LinearFog`** extends the **`Fog`** class
  - Thickness increases linearly with distance

- Use linear fog to create more easily controlled fog, though less realistic

- Set *front* and *back* distances to control density

  effect = (back - distance) / (back - front)
  color = effect * shapeColor + (1-effect) * fogColor

LinearFog Class Hierarchy

- **`LinearFog`** extends the **`Fog`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Fog
                 ├─ javax.media.j3d.ExponentialFog
                 └─ javax.media.j3d.LinearFog
```

LinearFog Class Methods

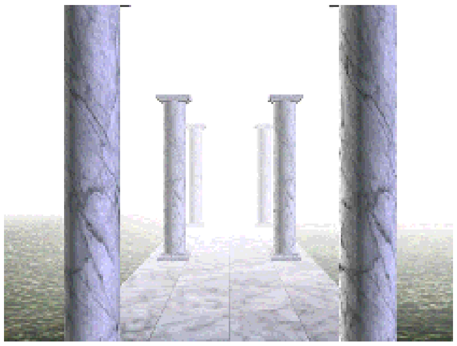
Method	*Default*
`LinearFog( )`	-
`void setFrontDistance( double front )`	0.0
`void setBackDistance( double back )`	1.0

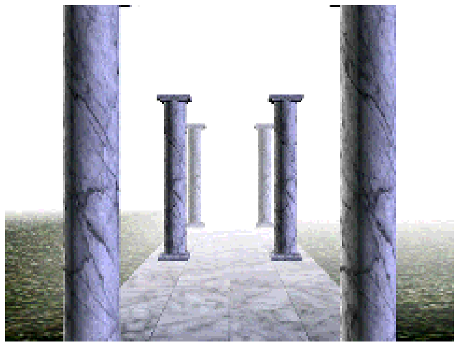
LinearFog Example Code

```
TransformGroup group = new TransformGroup( );
. . .
LinearFog fog = new LinearFog( );
fog.setColor( new Color3f( 0.5f, 0.5f, 0.5f ) );
fog.setFrontDistance( 1.0 );
fog.setBackDistance( 30.0 );
. . .
fog.setInfluencingBounds( bounds );
group.addChild( fog );
```

Working with Fog

LinearFog Example

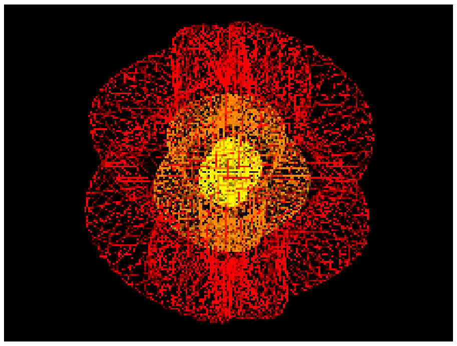

Distances wide apart

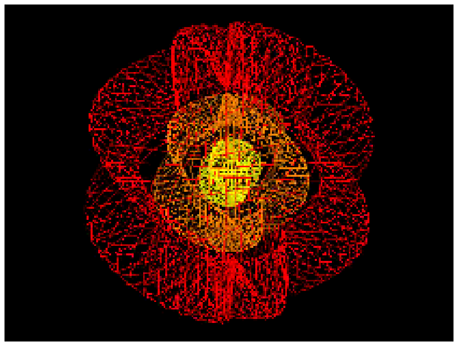

Distances close together

[**`ExLinearFog`**]

Depth Cueing Example

- For depth-cueing, use black linear fog
  - Set front distance to distance to center of shape
  - Set back distance to distance to back of shape


Depth cueing off


Depth cueing on

[**`ExDepthCue`**]

Fog Influencing Bounds and Scope

- Fog effects are bounded to a volume and scoped to a list of groups
  - Identical to light influencing bounds and scope

- By default, fog has no influencing bounds and affects nothing!
  - ***Common error:*** forgetting to set influencing bounds

- By default, fog has universal scope and affects everything within its influencing bounds

Fog Class Methods

Method	*Default*
`void setInfluencingBounds( Bounds bounds )`	None
`void setInfluencingBoundingLeaf( BoundingLeaf leaf )`	None
`void setScope( Group group, int index )`	None
`void addScope( Group group )`	-
`void insertScope( Group group, int index )`	-
`void removeScope( int index )`	-

Influencing Bounds Example Code

- Set bounds relative to the fog’s coordinate system

```
LinearFog fog = new LinearFog( );
fog.setInfluencingBounds( bounds );
```

- Or relative to a bounding leaf’s coordinate system

```
TransformGroup group = new TransformGroup( );
BoundingLeaf leaf = new BoundingLeaf( bounds );
group.addChild( leaf );
. . .
LinearFog fog = new LinearFog( );
fog.setInfluencingBoundingLeaf( leaf );
```

Clipping Foggy Shapes

- Shapes obscured by fog are still drawn

- To increase performance, you can clip away distant shapes using a **`Clip`** node
  - You can clip without using fog too
  - Fog helps cover up the abruptness of clipping

Clip Class Hierarchy

- **`Clip`** extends the **`Leaf`** class

Class Hierarchy

```
java.lang.Object
 └─   javax.media.j3d.SceneGraphObject
          └─   javax.media.j3d.Node
                   └─   javax.media.j3d.Leaf
                            └─   javax.media.j3d.Clip
```

Clipping Shapes

- Clipping chops away shapes, or parts of shapes, further away from the viewer than a *back distance*
    - Also called a *far clipping plane*

- Clipping can be obscured using linear fog
    - The fog back distance = the clip back distance

Clip Application Bounds

- A clip is applied when:
    - The viewer's activation radius intersects it's *application bounds*
    - If multiple clips are active, the closest is used
    - If no clips are active, the **`View`** object's far clip distance is used

- Clip bounding enables different clip planes for different areas of the scene

Clipping Shapes

- A clip’s application bounds is a bounded volume
  - Sphere, box, polytope, or combination
  - To make a global clip, use a huge bounding sphere

- By default, clip has no application bounds and affects nothing!
  - ***Common error:*** forgetting to set application bounds

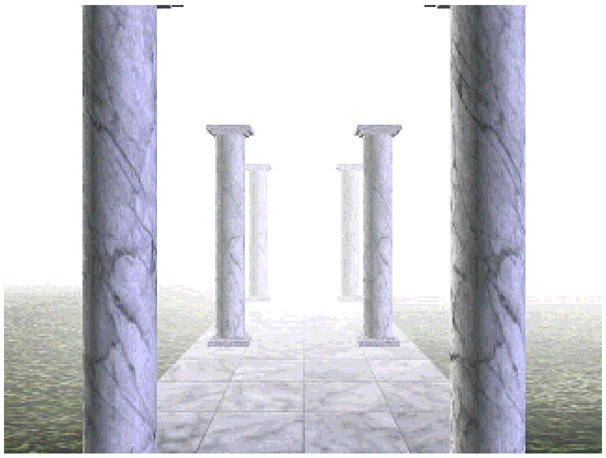
Clip Class Methods

Method	*Default*
`Clip( )`	-
`void setBackDistance( double back )`	1.0
`void setApplicationBounds( Bounds bounds )`	None
`void setApplicationBoundingLeaf( BoundingLeaf leaf )`	None

Clip Example Code

```
TransformGroup group = new TransformGroup( );
. . .
Clip clip = new Clip( );
clip.setBackDistance( 30.0 );
. . .
clip.setApplicationBounds( bounds );
group.addChild( clip );
```

Clip Example



[**ExClip**]

Summary

Class	*Attributes*
`Fog`	Color, Influencing bounds, Scope
├ **`ExponentialFog`**	Density
└ **`LinearFog`**	Front distance, Back distance
`Clip`	Back distance, Application bounds

Summary

- Fog affects shapes within the influencing bounds
  - Default is *no influence*, so nothing affected!

- *and* within groups on the fog's scope list
  - Default is *universal scope*, so everything is affected (if within influencing bounds)

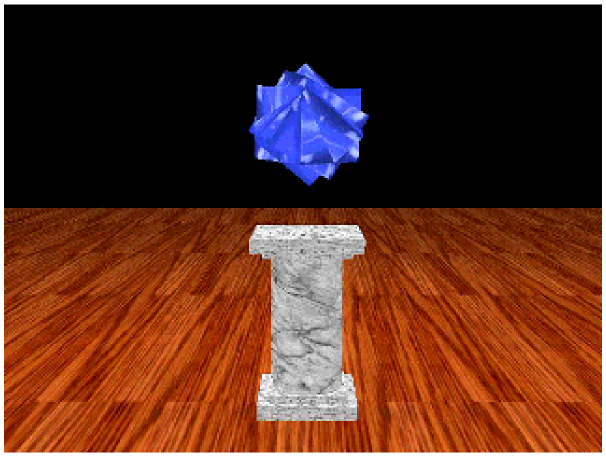
Adding Sound

Motivation

- You can add sounds to your environment:
  - Localized sounds - sounds with a position
    - User interface sounds (clicks, alerts)
    - Data sonification
    - Game sounds (laser blasters, monster growls)

  - Background sounds - sounds filling an environment
    - Presentation sounds (voice over, narration)
    - Environment sounds (ocean waves, wind)
    - Background music

Example



[ExSound]

Types of Sounds

- Java 3D provides three types of sounds:
  - Background
  - Point
  - Cone

- All three types of sounds have:
  - Sound data to play
  - An initial gain (overall volume)
  - Looping parameters
  - Playback priority
  - Scheduling bounds (like a behavior)

Sound Class Hierarchy

- All sounds share attributes inherited from the **`Sound`** abstract class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Sound
                 ├─ javax.media.j3d.BackgroundSound
                 └─ javax.media.j3d.PointSound
                     └─  javax.media.j3d.ConeSound
```

Loading Sound Data

- Sound nodes play *sound data* describing a digital waveform
  - Data loaded by a **`MediaContainer`** from
    - A file on disk or on the Web

- Typical sound file formats include:
  - **`AIF`**: standard cross-platform format
  - **`AU`**: standard Sun format
  - **`WAV`**: standard PC format

MediaContainer Class Hierarchy

- The **`MediaContainer`** class provides functionality to load sound files given a URL or file path

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.NodeComponent
         └─ javax.media.j3d.MediaContainer
```

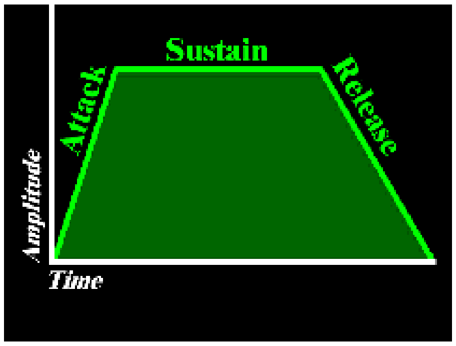
MediaContainer Class Methods

Method	*Default*
`MediaContainer( )`	-
`void setUrl( String path )`	None
`void setUrl( URL url )`	None

- The URL can be a local file path or a Web URL
- Setting the URL triggers loading of the sound

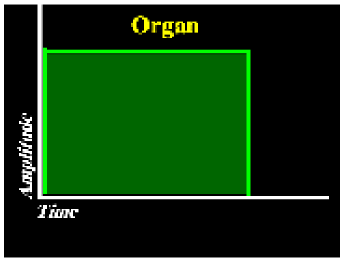
Sound Envelopes

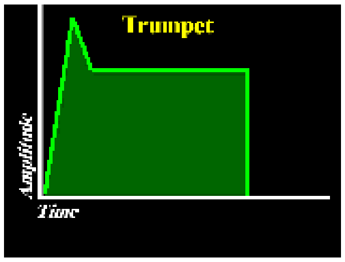
- Sound files have a built-in amplitude *Envelope* with three stages:
  - *Attack*: the start of the sound
  - *Sustain*: the body of the sound
  - *Release*: the ending decay of the sound

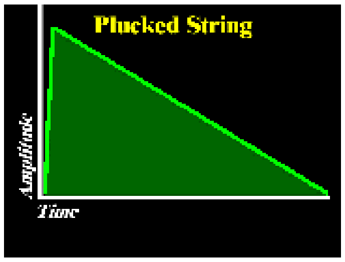


Sound Envelopes

- The envelope is part of the sound data loaded by a **`MediaContainer`**
  - Set sound envelopes using a sound editor
  - Amplitude is *not* ramped by Java 3D

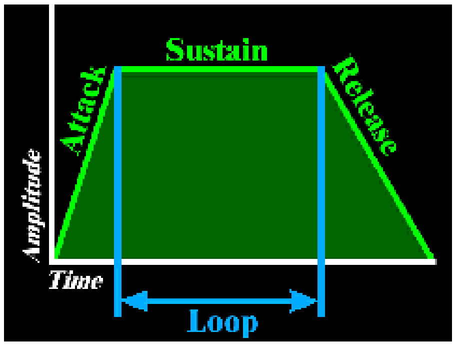






Looping Sounds

- To *sustain* a sound, you can loop between *loop points*
  - Authored using a sound editor
    - They usually bracket the *Sustain* stage
  - If no loop points, loop defaults to entire sound
  - Loops can run a number of times, or forever



Controlling Sounds

- Sounds may be enabled and disabled
  - Enabling a sound makes it *schedulable*
  - The sound will start to play if the sound’s scheduling bounds intersect the viewer’s activation radius
- Overall sound volume may be controlled with a gain multiplication factor

Sound Class Methods

Method	*Default*
`void setSoundData( MediaContainer sound )`	None
`void setLoop( int count )`	0
`void setEnable( boolean onOff )`	false
`void setInitialGain( float amplitude )`	1.0

- Special loop count values:
  - A **`0`** count loops 0 times (play once through)
  - A **`-1`** count loops forever

Background Sounds

- **`BackgroundSound`** extends the **`Sound`** class
  - Background sound waves come from all directions, flooding an environment at constant volume
  - Similar idea as an **`AmbientLight`**

- Use background sounds for:
  - Presentation sounds (voice over, narration)
  - Environment sounds (ocean waves, wind)
  - Background music

- You can have multiple background sounds playing

BackgroundSound Class Hierarchy

- **`BackgroundSound`** extends the **`Sound`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Sound
                 ├─ javax.media.j3d.BackgroundSound
                 └─ javax.media.j3d.PointSound
                     └─  javax.media.j3d.ConeSound
```

BackgroundSound Class Methods

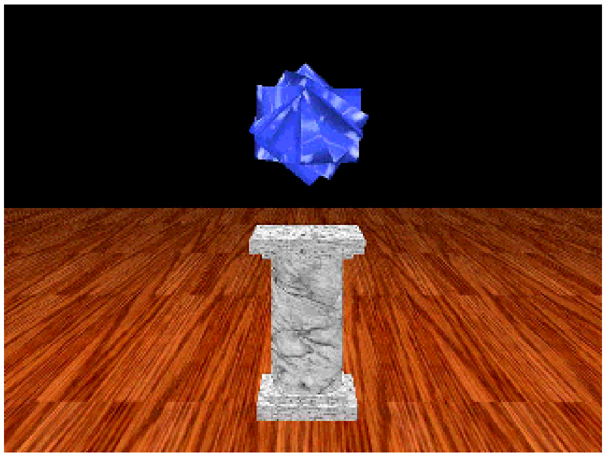
Method	*Default*
`BackgroundSound( )`	-

- **`BackgroundSound`** adds no additional methods beyond those of the **`Sound`** class

BackgroundSound Example Code

```
MediaContainer music = new MediaContainer( "canon.wav" );
. . .
TransformGroup group = new TransformGroup( );
. . .
BackgroundSound sound = new BackgroundSound( );
sound.setSoundData( music );
sound.setInitialGain( 1.0f );
sound.setEnable( true );
sound.setLoop( -1 );     // Loop forever
. . .
sound.setSchedulingBounds( bounds );
group.addChild( sound );
```

BackgroundSound Example



[ExSound]

Point Sounds

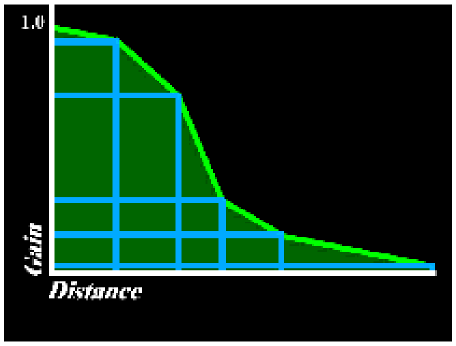
- **`PointSound`** extends the **`Sound`** class
  - Sound waves emit radially from a point in all directions
  - Similar idea as a **`PointLight`**

- Use point sounds to simulate local sounds like:
  - User interface sounds (clicks, alerts)
  - Data sonification
  - Game sounds (laser blasters, monster growls)

- You can have multiple point sounds playing

Sound Gain Variation with Distance

- Point sound waves are *attenuated*:
  - Amplitude decreases as the viewer moves away
- Attenuation is controlled by a list of value pairs:
  - *Distance* from sound position
  - *Gain* at that distance



PointSound Class Hierarchy

- **`PointSound`** extends the **`Sound`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Sound
                 ├─ javax.media.j3d.BackgroundSound
                 └─ javax.media.j3d.PointSound
                     └─  javax.media.j3d.ConeSound
```

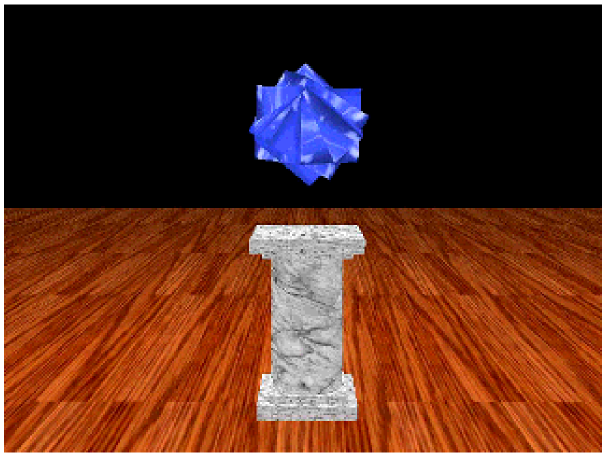
PointSound Class Methods

Method	*Default*
`PointSound( )`	-
`void setPosition( Point3f pos )`	0.0, 0.0, 0.0
`void setDistanceGain( Point2f[] atten )`	None

PointSound Example Code

```
MediaContainer effect = new MediaContainer( "willow1.wav" );
Point2f[] atten = new Point2f[. . .];
. . .
TransformGroup group = new TransformGroup( );
. . .
PointSound sound = new PointSound( );
sound.setSoundData( effect );
sound.setInitialGain( 1.0f );
sound.setEnable( true );
sound.setLoop( -1 );     // Loop forever
sound.setPosition( new Point3f( 0.0f, 0.0f, 0.0f ) );
sound.setDistanceGain( atten );
. . .
sound.setSchedulingBounds( bounds );
group.addChild( sound );
```

PointSound Example



[ExSound]

Cone Sounds

- **`ConeSound`** extends the **`PointSound`** class
  - Sound waves emit radially from a point in a direction, constrained to a cone
  - Similar idea as a **`SpotLight`**

- Use cone sounds to simulate local directed sounds like:
  - Loud speakers
  - Musical instruments

- You can have multiple cone sounds playing

Sound Gain Variation with Distance

- **`ConeSound`** extends **`PointSound`** support for attenuation
  - **`PointSound`** uses one list of distance-gain pairs that apply for all directions

  - **`ConeSound`** uses *two* lists of distance-gain pairs that apply in front and back directions
    - The cone’s aim direction is the front direction
    - If no back list is given, the front list is used

Sound Gain and Filter Variation with Angle

- Real-world sound sources emit in a direction
  - Volume (gain) and frequency content varies with angle

- **`ConeSound`** angular attenuation simulates this effect with a list of angle-gain-filter triples
  - *Angle* from the cone’s front direction
  - *Gain* at that angle
  - *Cutoff frequency* for a low-pass filter at that angle

ConeSound Class Hierarchy

- **`ConeSound`** extends the **`PointSound`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Sound
                 ├─ javax.media.j3d.BackgroundSound
                 └─ javax.media.j3d.PointSound
                     └─  javax.media.j3d.ConeSound
```

The ConeSound Class

Method	*Default*
`ConeSound( )`	-
`void setDirection( Vector3f dir )`	0.0, 0.0, 1.0
`void setDistanceGain( Point2f[] front, Point2f[] back )`	None
`void setBackDistanceGain( Point2f[] back )`	None
`void setAngularAttenuation( Point3f[] atten )`	None

- Attenuation angles are in the range 0.0 to PI radians

ConeSound Example Code

```
MediaContainer effect = new MediaContainer( "willow1.wav" );
Point2f[] front = new Point2f[. . .];
Point2f[] back  = new Point2f[. . .];
Point3f[] atten = new Point3f[. . .];
. . .
TransformGroup group = new TransformGroup( );
. . .
ConeSound sound = new ConeSound( );
sound.setSoundData( effect );
sound.setInitialGain( 1.0f );
sound.setEnable( true );
sound.setLoop( -1 );     // Loop forever
sound.setPosition( new Point3f( 0.0f, 0.0f, 0.0f ) );
sound.setDirection( new Vector3f( 0.0f, 0.0f, 1.0f ) );
sound.setDistanceGain( front, back );
sound.setAngularAttenuation( atten );
. . .
sound.setSchedulingBounds( bounds );
group.addChild( sound );
```

Sound Scheduling Bounds

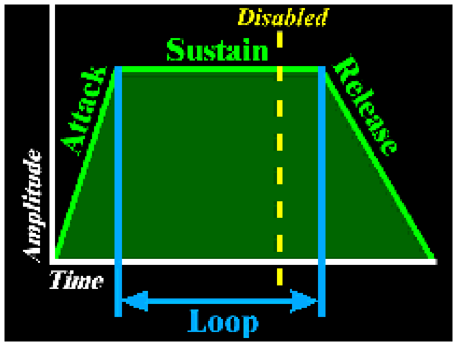
- A sound is hearable (if it is playing) when:
  - The viewer's activation radius intersects it's *scheduling bounds*
  - Multiple sounds can be active at once
  - Identical to behavior scheduling
- Sound bounding enables different sounds for different areas of the scene
- By default, sounds have no scheduling bounds and are never hearable!
  - ***Common error:*** forgetting to set scheduling bounds

Sound Class Methods

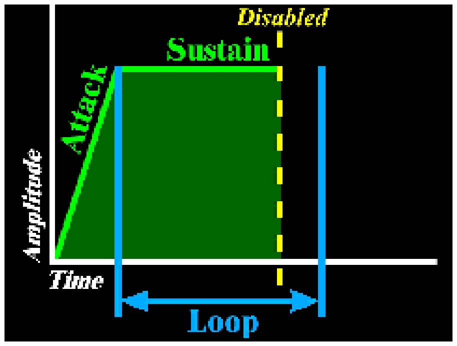
Method	*Default*
`void setSchedulingBounds( Bounds bounds )`	None
`void setSchedulingBoundingLeaf( BoundingLeaf leaf )`	None

Controlling the Release

- When you disable a sound:
  - Enable the release to let the sound finish playing, without further loops
  - Disable the release to stop it immediately



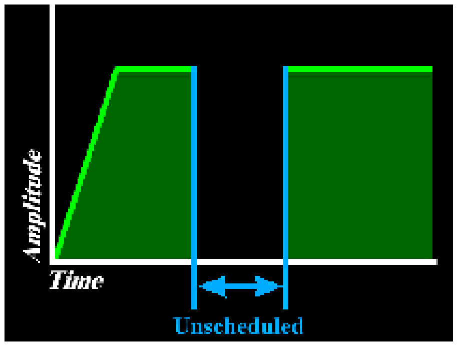
Release enabled



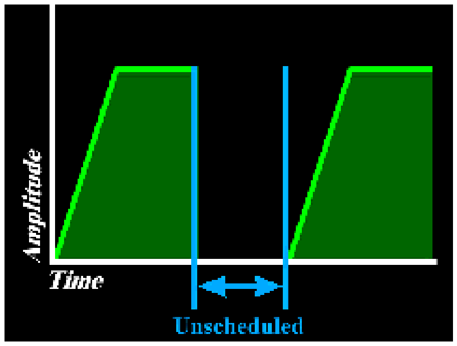
Release disabled

Controlling Continuous Playback

- When a sound is unscheduled (viewer moves out of scheduling bounds):
  - Enable *continuous* playback to keep it going silently
    - It resumes, *in progress* if scheduled again
  - Disable *continuous* playback to skip silent playback
    - It starts at the beginning if scheduled again



Continuous enabled



Continuous disabled

Prioritizing Sounds

- Sound hardware and software limits the number of simultaneous sounds
    - Worst case is 4 point/cone sounds and 7 background sounds

- You can prioritize your sounds
    - A low priority sound may be temporarily muted when a high priority sound needs to be played

Sound Class Methods

Method	*Default*
`void setReleaseEnable( boolean onOff )`	false
`void setContinuousEnable( boolean onOff )`	false
`void setPriority( float ranking )`	1.0

Summary

Class	*Attributes*
`Sound`	Enable, Sound data, Release enable, Continuous enable, Initial gain, Loop count, Priority, Scheduling bounds
├ **`BackgroundSound`**	-
└ **`PointSound`**	Position, Distance gain
└ **`ConeSound`**	Direction, Back distance gain, Angular attenuation

- Sounds are hearable (if playing) when the viewer’s activation radius intersects the sound’s scheduling bounds

Controlling the Sound Environment

Motivation

- The **Sound** classes control features of the sound

- To enhance realism, you can control features of the environment too

- Use *soundscapes* and *aural attributes* to
  - Add reverberation (echos)
  - Use different reverberation for different rooms
  - Control doppler pitch shift
  - Control frequency filtering with distance

Soundscape Class Hierarchy

- All soundscape features are controlled using the **`Soundscape`** class

Class Hierarchy

```
java.lang.Object
 └─  javax.media.j3d.SceneGraphObject
      └─  javax.media.j3d.Node
           └─  javax.media.j3d.Leaf
                └─  javax.media.j3d.Soundscape
```

Soundscape Application Bounds

- A *Soundscape* affects sound when:
  - The viewer’s activation radius intersects it’s *application bounds*
    - Identical to background application bounds
  - If multiple soundscapes active, closest one used
  - If no soundscapes active, no reverb, filtering, or doppler shift takes place

- By default, soundscapes have no application bounds and are never applied!
  - ***Common error:*** forgetting to set application bounds

Controlling the Sound Environment

Soundscape Class Methods

Method	*Default*
`Soundscape( )`	-
`void setApplicationBounds( Bounds bounds )`	None
`void setApplicationBoundingLeaf( BoundingLeaf leaf )`	None
`void setAuralAttributes( AuralAttributes aural )`	Defaults

Types of Aural Attributes

- Java 3D provides three types of aural attributes:
  - Reverberation (echo)
  - Distance filtering
  - Doppler Shift

- All aural attributes types are controlled with an **`AuralAttributes`** node

AuralAttributes Class Hierarchy

- All aural attributes features are controlled using the **`AuralAttributes`** class

Class Hierarchy

```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.NodeComponent
         └─ javax.media.j3d.AuralAttributes
```

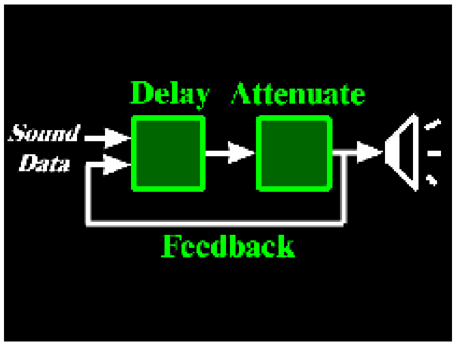
Controlling Reverberation

- In the real world, sound bounces off walls, floors, etc
  - If the bounce surface is hard, we hear a perfect echo
  - If it is soft, some frequencies are absorbed
  - The set of all echos is *Reverberation*

- Java 3D provides a simplified model of reverberation
  - Sounds echo after a *reverb delay* time
  - Echo attenuation is controlled by a *reflection coefficient*
  - Echos stop after a *reverb order* (count)

Controlling Reverberation

- Reverberation uses a feedback loop:
    - Each echo is a trip around the feedback loop



AuralAttributes Class Methods

Method	*Default*
`AuralAttributes( )`	-
`void setReverbDelay( float delay )`	0.0
`void setReflectionCoefficient( float coeff )`	0.0
`void setReverbOrder( int order )`	0

- An order of -1 continues echos until they die out

Sound Delay with Distance

- When a sound starts playing, there is a delay before it is heard
  - It takes time for sound to travel from source to listener

- The default speed of sound is 0.344 meters/millisecond
  - You can scale this up or down using *rolloff*
  - Values 0.0 <= 1.0 slow down sound
  - Values > 1.0 speed up sound
  - A 0.0 value mutes the sound

Sound Filtering with Distance

- An *attribute gain* controls overall volume
- Sound waves are *filtered*, decreasing high frequency content as the viewer moves away
- Attenuation is controlled by a list of value pairs:
  - *Distance* from sound position
  - *Cutoff frequency* for a low-pass filter at that distance

Controlling the Sound Environment

AuralAttributes Class Methods

Method	*Default*
`void setAttributeGain( float gain )`	1.0
`void setRolloff( float rolloff )`	1.0
`void setDistanceFilter( Point2f[] atten )`	None

Doppler Shift

- Doppler shift varies pitch as the sound or viewer moves
  - Set the *velocity scale factor* to scale the relative velocity between the sound and viewer
  - A *frequency scale factor* accentuates or dampens the effect

AuralAttributes Class Methods

Method	*Default*
`void setFrequencyScaleFactor( float scale )`	1.0
`void setVelocityScaleFactor( float scale )`	0.0

AuralAttributes Example Code

```
BoundingSphere bounds = new BoundingSphere(
    new Point3d( 0.0, 0.0, 0.0 ), 1000.0 );
. . .
TransformGroup group = new TransformGroup( );
. . .
AuralAttributes aural = new AuralAttributes( );
aural.setReverbDelay( 2.0f );
aural.setReverbOrder( -1 );  // Until dies out
aural.setReflectionCoefficient( 0.2f );  // dampen
. . .
Soundscape scape = new Soundscape( );
scape.setAuralAttributes( aural );
scape.setApplicationBounds( bounds );
group.addChild( scape );
```

Summary

Class	*Attributes*
`Soundscape`	Aural attributes, Application bounds
`AuralAttributes`	Attribute gain, Distance filtering, Frequency scale factor, Velocity scale factor, Reflection coefficient, Reverb delay, Reverb order, Rolloff

- Soundscapes apply when the viewer’s activation radius intersects the soundscape’s application bounds

Conclusions

Coverage

- This morning we covered:
  - Creating scene graphs
  - Building 3D shapes
  - Building point, line, triangle, and quad geometry
  - Building text geometry
  - Controlling appearance
  - Loading content from files
  - Grouping shapes
  - Transforming shapes
  - Sharing shapes
  - Controlling texture mapping
  - Building raster geometry

Coverage

- This afternoon we covered:
  - Controlling viewing and view platforms
  - Creating behaviors
  - Controlling behavior wakeup
  - Picking shapes
  - Getting input
  - Lighting the environment
  - Creating backgrounds
  - Working with fog
  - Adding sound
  - Controlling the sound environment

Where to find out more

- The Java 3D specification
  - http://www.javasoft.com/products/java-media/3D/

    Or...

  - **`The Java 3D API Specification`**
    by Henry Sowizral, Kevin Rushforth, Michael Deering
    published by Addison-Wesley

Where to find out more

- The Java 3D Site at Sun
  - http://www.sun.com/desktop/java3d

- The Java 3D Repository
  - http://java3d.sdsc.edu

Where to find out more

- The latest version of these course notes are available at
  - Sun’s Java 3D site
  - The Java 3D Repository
  - Off of Dave Nadeau’s home page, http://www.sdsc.edu/~nadeau

Introduction to Programming with Java 3D

Thanks for coming!

Henry Sowizral
Michael Deering
Sun Microsystems

Dave Nadeau
Mike Bailey
SDSC / UCSD