Introduction to Programming with Java 3D

Lecturers

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

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

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. Its feature set is designed to enable quick development of complex 3D applications and, at the same time, enable fast and efficient implementation 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.

In this tutorial, participants 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 these 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 600 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 JDK 1.2 (Java 2 Platform) or later
- Java 3D 1.1 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 600+ 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. One of the course lecturers is an author 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:

You are free to use these tutorial notes in whole or in part to help you teach your own Java 3D tutorial. You may translate these notes into other languages and you may post copies of these notes on your own Web site, as long as the above copyright notice is included as well. You may not, however, sell these tutorial notes for profit or include them on a CD-ROM or other media product without written permission.

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

Henry A. Sowizral (Organizer)

Title	Distinguished 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 Distinguished 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 tutorials on topics including expert systems and virtual reality at conferences including COMPCON, Supercomputing, VRAIS, and SIGGRAPH. Henry has taught Java 3D at SIGGRAPH, Eurographics, Visualization, JavaOne, VRAIS, 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.

David R. Nadeau

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, a national research center specializing in computational science and engineering, located on the campus of the University of California, San Diego. His areas of interest include scientific visualization and virtual reality, He has taught Java 3D and VRML at multiple conferences including SIGGRAPH, Eurographics, Supercomputing, WebNet, WMC/SCS, VRAIS, and Visualization.

Dave is a co-author of *The VRML 2.0 Sourcebook* published by John Wiley & Sons. He 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.

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 1.1 class files (or later)
- The Java JDK 1.2 (Java 2 Platform) class files (or later)
- 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, 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

Morning

Section 1 - Introduction, Scene graphs, Shapes, Appearance

Section 2 - Groups, Transforms, Texture mapping, Lighting

Afternoon

Section 3 - Universes, Viewing, Input, Behaviors

Section 4 - Interpolators, Picking, Backgrounds, Fog

Extended notes

Section 5 - Text geometry, Raster geometry, Advanced texture mapping

1

Welcome

Introduction to Programming with Java 3D

Welcome to the tutorial!

Tutorial schedule

Morning

Section 1 Introduction, Scene graphs, Shapes, Appearance

Section 2 Groups, Transforms, Texture mapping, Lighting

Afternoon

Section 3 Universes, Viewing, Input, Behaviors

Section 4 Interpolators, Picking, Backgrounds, Fog

Extended notes

Section 5 Text geometry, Advanced texture mapping, Sound, Sound environment

Tutorial scope

- This tutorial will:
  - Introduce Java 3D concepts and terminology
  - Discuss important Java 3D classes
  - Illustrate how to write a Java 3D application or applet
  - Discuss typical usage patterns, techniques, and tricks

Introduction

What is Java 3D?

- Java 3D is an interactive 3D graphics *Application Programming Interface* (API) for building applications and applets in Java

- A means for developing and presenting 3D content

- Designed for *Write once, run anywhere*
  - Multiple platforms (processors and pipes)
  - Multiple display environments
  - Multiple input devices

What is Java 3D?

- Raise the programming floor
- *Think objects* . . . not vertices
- *Think content* . . . not rendering process

What does Java 3D do?

- Provide a vendor-neutral, platform-independent API within Java
  - Integrates with other Java APIs: image processing, fonts, 2D drawing, user interfaces, etc.

- Enable high level application development
  - Authors focus upon content, not rendering
  - Java 3D handles optimal rendering

What does Java 3D do?

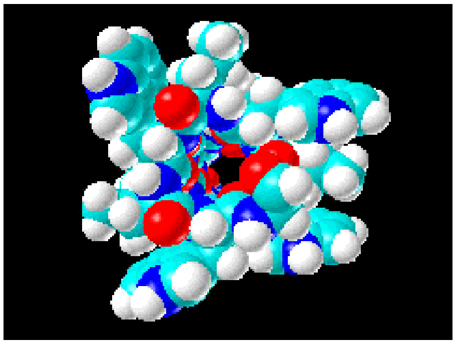
- Perform rendering optimizations
  - Scene management
  - Content culling based upon visibility (frustum)
  - Efficient pipeline use (sorting, batching)
  - Parallel rendering
  - Scene compilation (reorganization, combination, etc.)

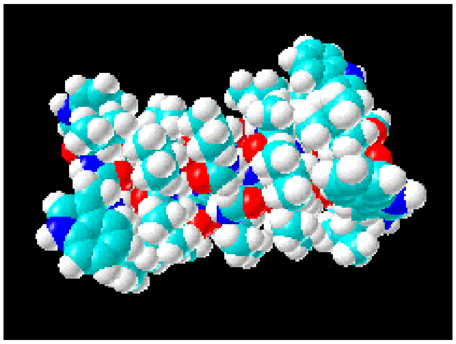
- And achieve high performance
  - Draw via OpenGL/Direct3D
  - Uses 3D graphics hardware acceleration where available

What application areas can use Java 3D?

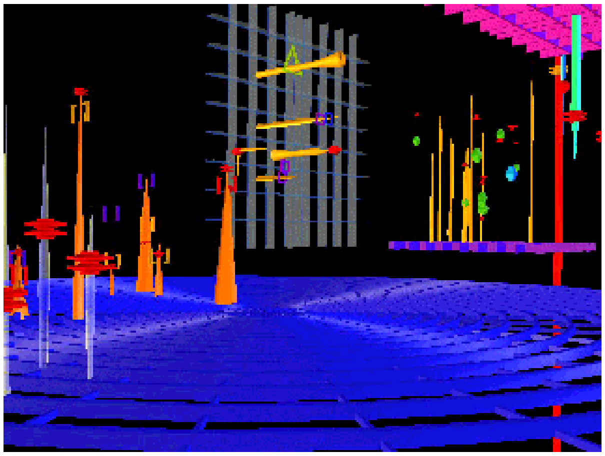
- Scientific visualization
- Information visualization
- Medical visualization
- Geographical information systems (GIS)
- Computer-aided design (CAD)
- Animation
- Education

Examples: Scientific Visualization

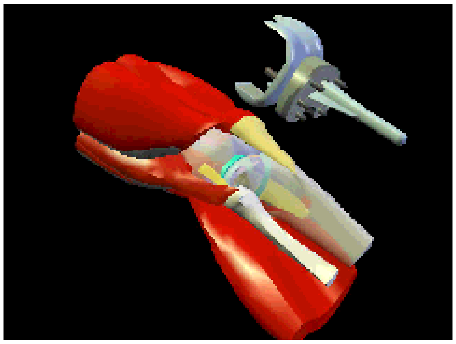


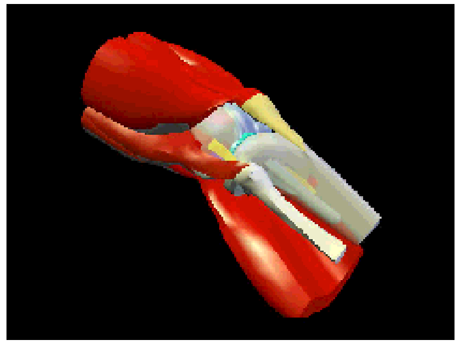


Examples: Abstract Data (Financial)

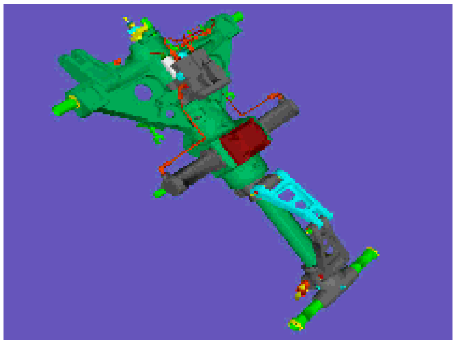


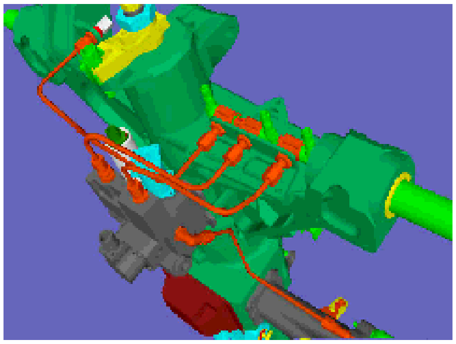
Examples: Medical Education





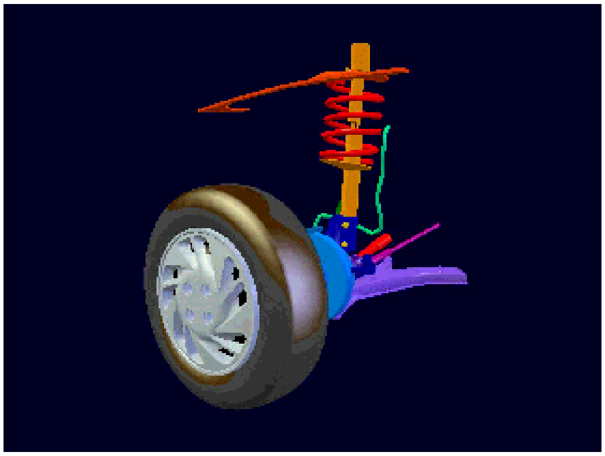
Examples: CAD





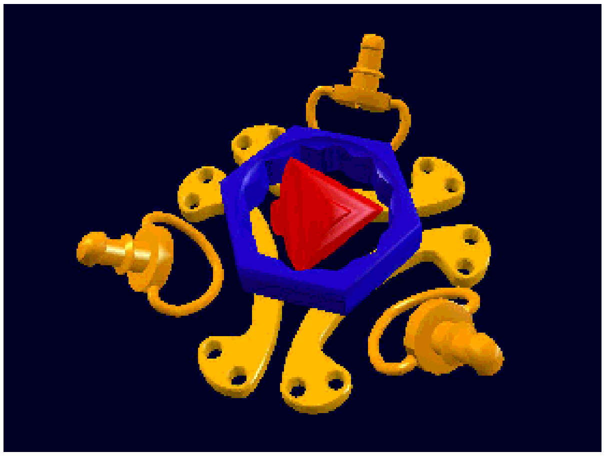
Introduction

Examples: Analysis

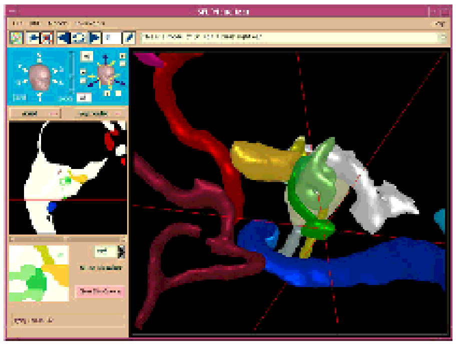


Examples: Animations

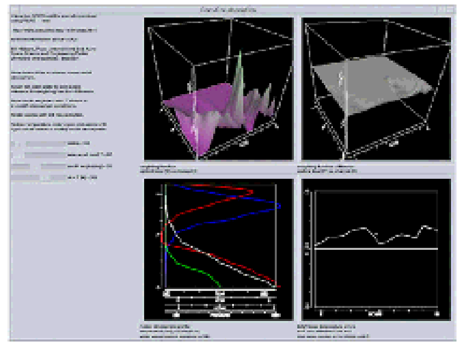
Examples: 3D Logos



Examples: Scientific Visualization



Anatomy Browser
University of Massachusets
and
Brigham and Women's
Hospital



Collaborative Visualization
Space Science and
Engineering Center (SSEC)

What software do I need to use Java 3D?

- Java development kit
  - Java 2 platform
  - Free from http://java.sun.com

- Java 3D development kit
  - Java 3D 1.1
  - Free from http://www.sun.com/desktop/java3D

- Sun provides Windows 9x/NT and Solaris ports

- Linux port is available

- Other ports come from platform vendors

What hardware do I need to use Java 3D?

- You will need a 3D graphics accelerator

  - On PCs:
    - PC cards are widely available
    - Should support OpenGL 1.1 features
    - A Direct3D version is in progress
    - Linux port uses Mesa

  - On Suns:
    - Creator 3D or Elite 3D hardware
    - Support OpenGL 1.2

How do I run a Java 3D application/applet?

- Java 3D applications:
  - Run like any other Java application

```
prompt> java myapplication
```

- Java 3D applets:
  - Use the *Java plug-in* in Netscape or Internet Explorer
  - Embeds the applet in a Web page
  - Java plug-in automatically downloads JDK and Java 3D if not already installed

How does Java 3D compare with other APIs?

- "Older" APIs enable only low-level hardware state control
  - Provide *and require* detailed control
  - OpenGL, Direct3D, low-level game engines

- "Newer" APIs focus upon high-level content control
  - Provide some rendering optimization
  - Java 3D
  - VRML
  - SGI OpenInventor, Optimizer/Cosmo3D (being phased out)
  - SGI-Microsoft "Fahrenheit"

Summary

- Java 3D is a high-level API for building interactive 3D applications and applets in Java
- Write once, run anywhere . . . *in 3D*

Building 3D content with a scene graph

Building a scene graph

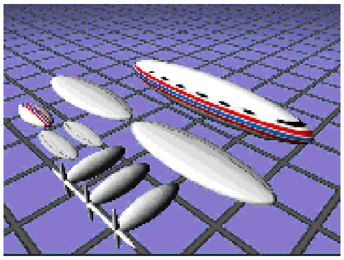
- A *scene graph* is a "family tree" containing scene data
  - "Children" are shapes, lights, sounds, etc.
  - "Parents" are groups of children and other parents
  - This defines a *hierarchical* grouping of shapes

- The application builds a scene graph using Java 3D classes and methods

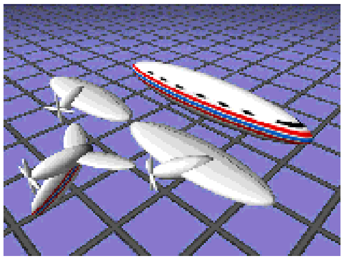
- Java 3D renders that scene graph onto the screen

Scene graph example

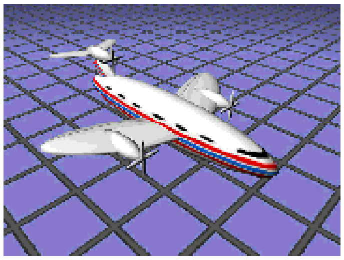
- For example, imagine building a toy airplane:



Start with parts on the table



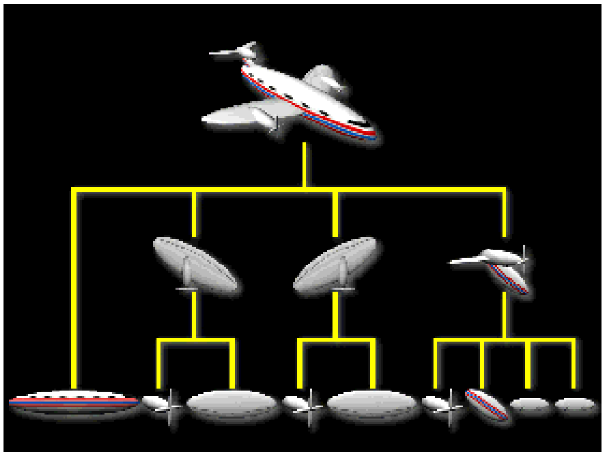
Assemble related parts



Assemble those into the final plane

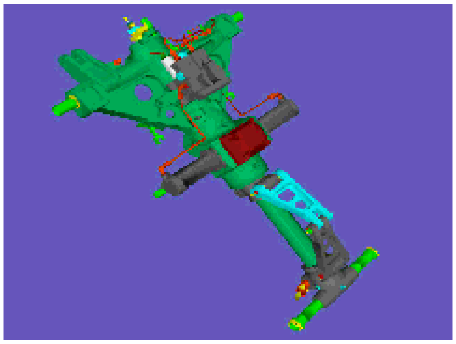
Sketching a scene graph diagram

- Sketching a scene graph diagram can clarify a design and ease software development

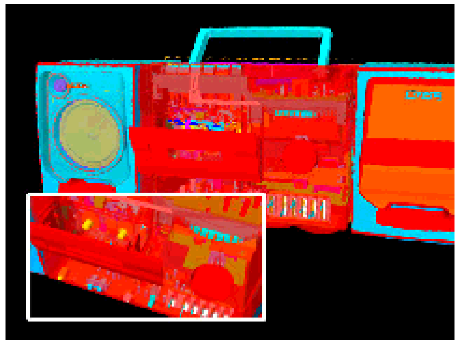


Examples of creating large scenes

- Java 3D scene graphs may include large numbers of shapes



Landing gear
192 shapes



Boom box
11,000 shapes

Building a scene graph

- Scene graphs are built from components including:
  - Shapes (geometry and appearance)
  - Groups and transforms
  - Lights
  - Fog and backgrounds
  - Sounds and sound environments (reverb)
  - Behaviors
  - View platforms (viewpoints)

Processing a scene graph

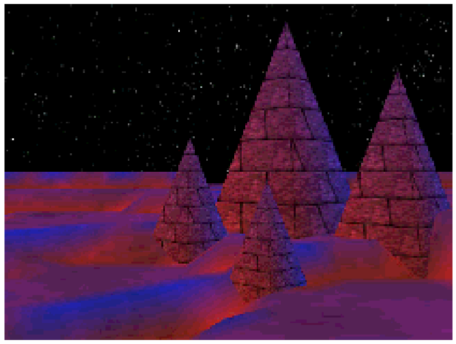
- Java 3D renders the scene graph
  - Scene graph specifies content, not rendering order
  - Rendering order is up to Java 3D

- Java 3D uses separate, independent and asynchronous threads
  - Graphics rendering
  - Sound "rendering"
  - Animation "behavior execution"
  - Input device management
  - Event generation (collision detection)

Building 3D content with a scene graph

Examples of Java 3D features

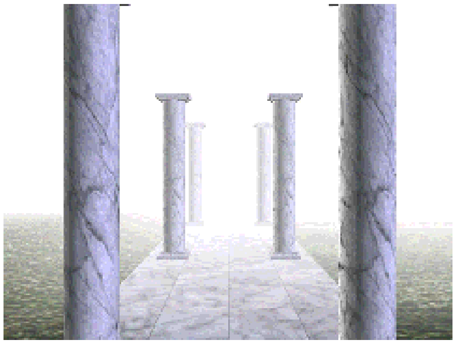
- You can control shape coloration and texture . . .



Examples of Java 3D features

. . . lighting and fog effects . . .

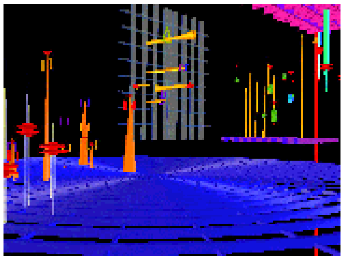
Monument



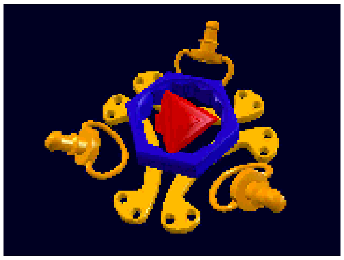
Colonade

Examples of Java 3D features

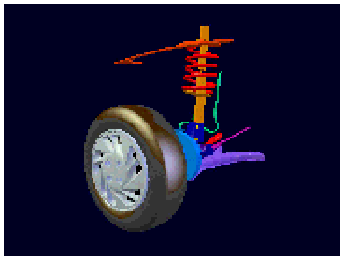
. . . shape position, orientation, and size and how those change over time, and more



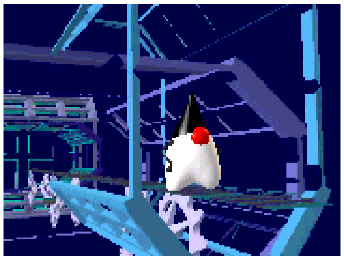
Jetsons-Vis



Logo



Car Suspension



Duke Treadmill

Using scene graph terminology

- But first, some terminology . . .

- *Node*: an item in a scene graph
    - *Leaf nodes*: nodes with no children
        - Shapes, lights, sounds, etc.
        - Animation behaviors
    - *Group nodes*: nodes with children
        - Transforms, switches, etc.

- *Node component*: a bundle of attributes for a node
    - Geometry of a shape
    - Color of a shape
    - Sound data to play

Scene graph base class hierarchy

- Leaf nodes, group nodes, node components, and different types of all of these lead to . . . *a Java 3D class hierarchy*

Class Hierarchy
java.lang.Object
└─ **javax.media.j3d.SceneGraphObject**
├─ **javax.media.j3d.Node**
├─ **javax.media.j3d.Group**
└─ **javax.media.j3d.Leaf**
└─ **javax.media.j3d.NodeComponent**

Building a scene graph

- Build nodes by instantiating Java 3D classes

```
Shape3D myShape1 = new Shape3D( myGeom1, myAppear1 );
Shape3D myShape2 = new Shape3D( myGeom2 );
```

- Modify nodes by calling methods on an instance

```
myShape2.setAppearance( newAppear2 );
```

- Build groups of nodes

```
Group myGroup = new Group( );
myGroup.addChild( myShape1 );
myGroup.addChild( myShape2 );
```

Building a scene graph

- We need to assemble chunks of content, each in its own scene graph
  - Build components separately

  - Assemble them into a common container: a *virtual universe*
    - A way to combine scene graphs
    - A place to root the scene graph

Using universe terminology

- *Virtual universe*: a collection of scene graphs
  - Typically one universe per application

- *Locale*: a position in the universe at which to put scene graphs
  - Typically one locale per universe

- *Branch graph*: a scene graph
  - Typically several branch graphs per locale

Using branch terminology

- Scene graphs are typically divided into two types of branch graphs:

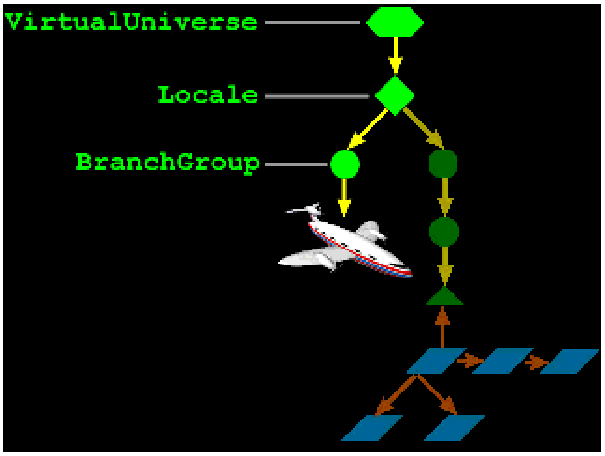
  - *Content branch*: shapes, lights, and other content
    - Typically multiple branches per locale

  - *View branch*: viewing information
    - Typically one per universe

- This division is optional:
  - Content and viewing information can be interleaved in the same branch (and sometimes should be)

Sketching a universe diagram

- A universe builds superstructure to contain scene graphs



Superstructure class hierarchy

- Universes and locales are superstructure classes for organizing content

Class Hierarchy
java.lang.Object

```
java.lang.Object
 ├─ javax.media.j3d.VirtualUniverse
 ├─ javax.media.j3d.Locale
 └─ javax.media.j3d.Node
     └─ javax.media.j3d.Group
         └─ javax.media.j3d.BranchGroup
```

Building a universe

- Build a universe

```
VirtualUniverse myUniverse = new VirtualUniverse( );
```

- Build a locale

```
Locale myLocale = new Locale( myUniverse );
```

- Build a branch group

```
BranchGroup myBranch = new BranchGroup( );
```

Building a universe

- Build nodes and groups of nodes

```
Shape3D myShape = new Shape3D( myGeom, myAppear );
Group myGroup = new Group( );
myGroup.addChild( myShape );
```

- Add them to the branch group

```
myBranch.addChild( myGroup );
```

- Add the branch graph to the locale

```
myLocale.addBranchGraph( myBranch );
```

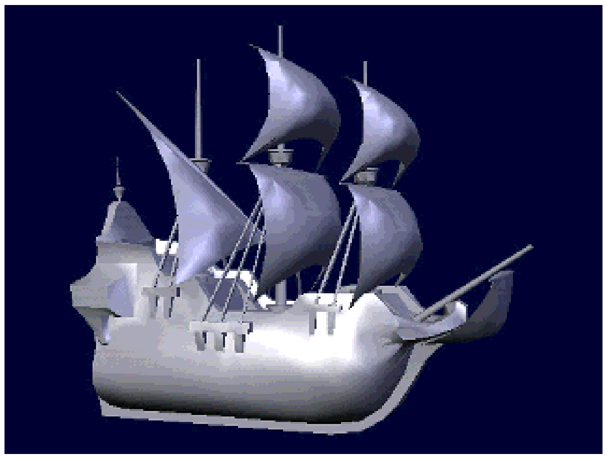
Building scene content

- Java 3D’s rich feature set enables you to build complex 3D content
  - Build content directly within your Java application
  - Load content from files
  - Do both

- *File loader* classes enable reading content from files in standard formats
  - VRML (Virtual Reality Modeling Language)
  - OBJ (Alias|Wavefront object)
  - LW3D (Lightwave 3D scene)
  - *others* . . .

Loading scene content from files

- Load an OBJ file describing a ship



[**`A3DApplet`**]

Building scene graph superstructure

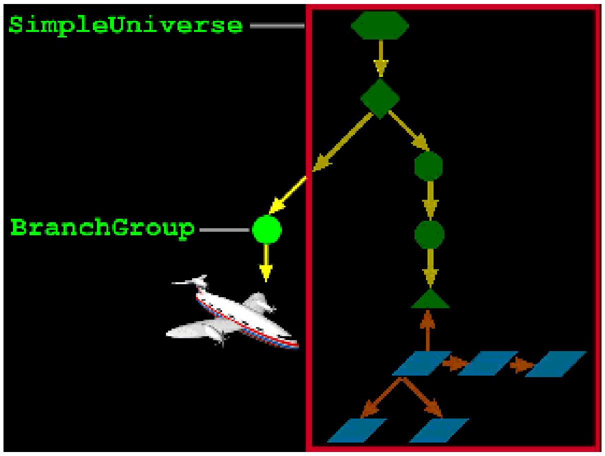
- *Utility* classes help automate common operations
  - Implemented atop Java 3D

- The **`SimpleUniverse`** utility builds a common arrangement of a universe, locale, and viewing classes

```
SimpleUniverse mySimple = new SimpleUniverse( myCanvas
mySimple.addBranchGraph( myBranch );
```

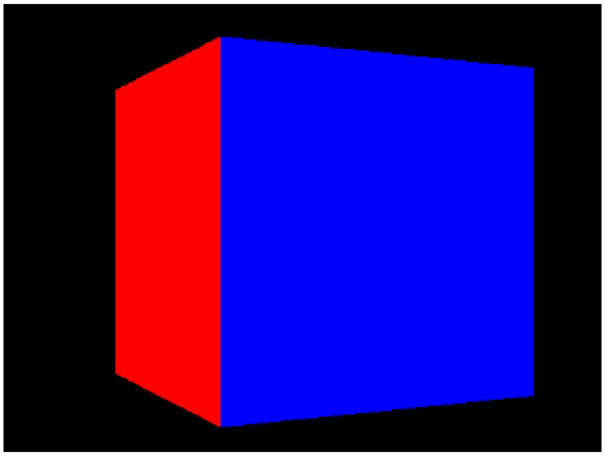
Sketching a simple universe diagram

- A **`SimpleUniverse`** encapsulates a common superstructure



HelloWorld example

- Let's build a multi-colored 3D cube and spin it about the vertical axis



[**`HelloWorld`**]

HelloWorld example code

- Import the Java 3D classes . . .

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

public class HelloWorld
{
. . .
}
```

HelloWorld example code

- Build a frame, 3D canvas, and simple universe . . .

```
public static void main( String[] args ) {
    Frame frame = new Frame( );
    frame.setSize( 640, 480 );
    frame.setLayout( new BorderLayout( ) );

    Canvas3D canvas = new Canvas3D( null );
    frame.add( "Center", canvas );

    SimpleUniverse univ = new SimpleUniverse( canvas )
    univ.getViewingPlatform( ).setNominalViewingTransf

    BranchGroup scene = createSceneGraph( );
    scene.compile( );
    univ.addBranchGraph( scene );

    frame.show( );
}
```

HelloWorld example code

- Build 3D shapes within a **`BranchGroup`** . . .

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

    // Make a changeable 3D transform
    TransformGroup trans = new TransformGroup( );
    trans.setCapability( TransformGroup.ALLOW_TRANSFOI
    branch.addChild( trans );

    // Make a shape
    ColorCube demo = new ColorCube( 0.4 );
    trans.addChild( demo );

    . . .
```

HelloWorld example code

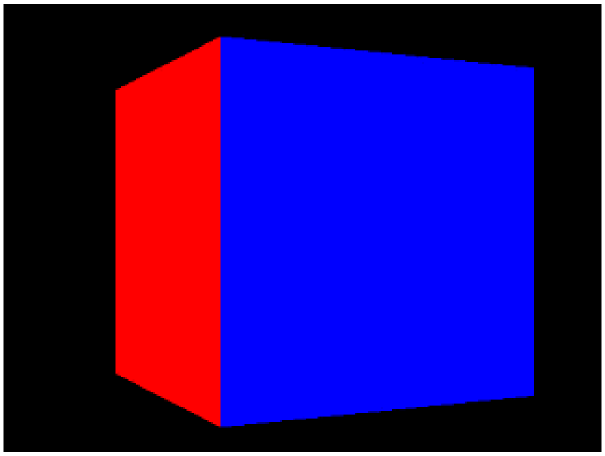
- Set up an animation behavior to spin the shapes . . .

```
        // Make a behavor to spin the shape
        Alpha spinAlpha = new Alpha( -1, 4000 );
        RotationInterpolator spinner =
            new RotationInterpolator( spinAlpha, trans );
        spinner.setSchedulingBounds(
                new BoundingSphere( new Point3d( ), 1000.0
        trans.addChild( spinner );

        return branch;
    }
```

HelloWorld example

- Which produces a spinning multi-colored 3D cube . . .



[**`HelloWorld`**]

Making a node live

- Adding a branch graph into a locale (or simple universe) makes its nodes *live* (drawable)

```
BranchGroup myBranch = new BranchGroup( );
myBranch.addChild( myShape );
myLocale.addBranchGraph( myBranch ); // make live!
```

- Removing the branch graph from the locale reverses the effect

```
myLocale.removeBranchGraph( myBranch );// not live
```

Checking if a node is live

- A method on **`SceneGraphObject`** queries if a node is live

Method
`boolean isLive( )`

Compiling a scene graph

- A method on **`BranchGroup`** compiles the branch, optimizing it for faster rendering

Method
`void compile( )`

Compiling a scene graph

- Compile a branch graph *before* making it live

```
BranchGroup myBranch = new BranchGroup( );
myBranch.addChild( myShape );
myBranch.compile( );
myLocale.addBranchGraph( myBranch );
```

Controlling access capabilities

- Node *capabilities* (permissions) control read and write access
  - Read or write any attribute *before* a node is live or compiled
  - Capabilities control access *while* a node is live or compiled
- Keep the number of capabilities small so Java 3D can make more optimizations during compilation

Controlling access capabilities

- Methods on the **`SceneGraphObject`** set/clear capabilities

Method
`void setCapability( int bit )`
`void clearCapability( int bit )`
`boolean getCapability( int bit )`

Controlling access capabilities

- Each node has its own read and write capabilities
  - Usually a separate capability for each attribute of a node
  - Node's also inherit parent class capabilities
  - Each capability has an upper-case name

- For example, **`Shape3D`** capabilities include:
  - **`ALLOW_APPEARANCE_READ`**
  - **`ALLOW_APPEARANCE_WRITE`**
  - **`ALLOW_GEOMETRY_READ`**
  - **`ALLOW_GEOMETRY_WRITE`**
  - **`ALLOW_COLLISION_BOUNDS_READ`**
  - **`ALLOW_COLLISION_BOUNDS_WRITE`**
  - Plus capabilities from the parent **`Node`** class, including:
    - **`ALLOW_BOUNDS_READ`**
    - **`ALLOW_BOUNDS_WRITE`**
    - **`ALLOW_PICKABLE_READ`**
    - **`ALLOW_PICKABLE_WRITE`**
    - . . . *and others*

Controlling access capabilities

- Set capabilities while you build your content

```
Shape3D myShape = new Shape3D( myGeom, myAppear );
myShape.setCapability( Shape3D.ALLOW_APPEARANCE_WRITE
```

- After a node is live, change attributes that have enabled capabilities

```
myShape.setAppearance( newAppear );  // allowed
```

- But you cannot change attributes for which you do not have capabilities set

```
myShape.setGeometry( newGeom );  // error!
```

Summary

- A *scene graph* is a hierarchy of groups of shapes, lights, sounds, etc.
- Your application builds the scene graph using Java 3D classes and methods
- The Java 3D implementation uses the scene graph behind the scene to render shapes, play sounds, execute animations, etc.

Summary

- A *virtual universe* holds everything
- A *locale* positions a *branch graph* in a universe
- A *branch graph* is a scene graph
- A *node* is an item in a scene graph
- A *node component* is a bundle of attributes for a node

Summary

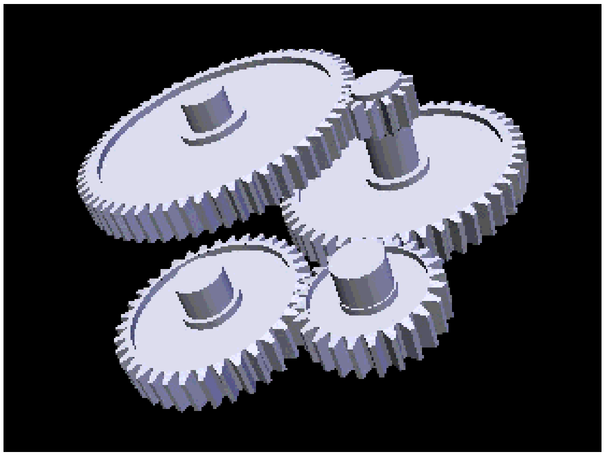
- Adding a branch graph to a locale makes it *live* and drawable
- *Compiling* a branch graph optimizes it for faster rendering
- *Capabilities* control access to node attributes after a node is *live* or *compiled*
    - Fewer capabilities enables more optimizations

Building 3D shapes

Motivation

- A **`Shape3D`** leaf node builds a 3D object with:
  - *Geometry*:
    - The form or structure of a shape
  - *Appearance*:
    - The coloration, transparency, and shading of a shape
- Java 3D supports multiple geometry and appearance features
- We'll talk about geometry first, then appearance

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

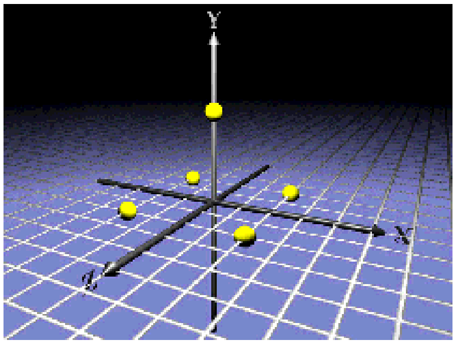
- Methods on **`Shape3D`** set geometry and appearance attributes

Method
`Shape3D( )`
`Shape3D( Geometry geometry, Appearance appearance )`
`void setGeometry( Geometry geometry )`
`void setAppearance( Appearance appearance )`

Building geometry using coordinates

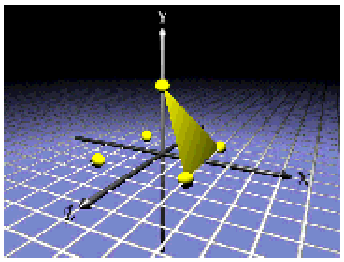
- Building shape geometry is like a 3D connect-the-dots game
  - Place "dots" at 3D *coordinates*
  - Connect-the-dots to form 3D shapes

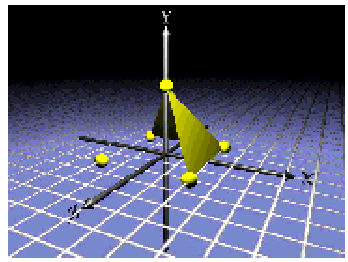
- For example, to build a pyramid start with five coordinates

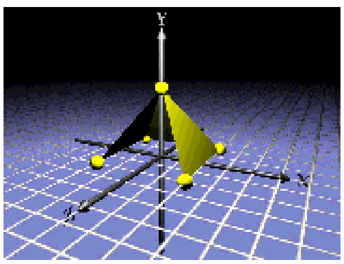


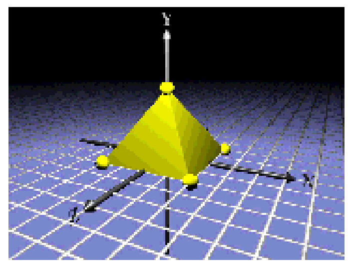
Building geometry using coordinates

- Finish the pyramid by connecting the dots to form triangles

1. 

2. 

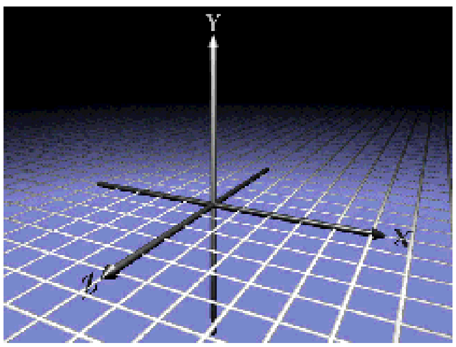
3. 

4. 

Using a right-handed coordinate system

- 3D coordinates are given in a *right-handed* coordinate system
  - X = left-to-right
  - Y = bottom-to-top
  - Z = back-to-front

- Distances are conventionally in meters



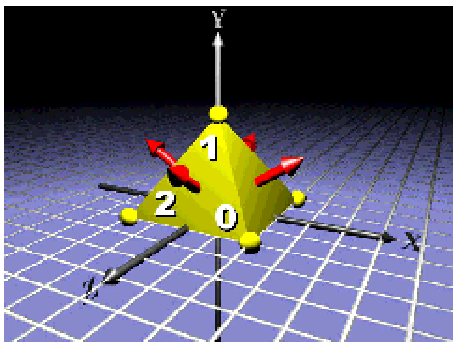
Using coordinate order

- Polygons have a front and back:
  - By default, only the *front* side of a polygon is rendered
  - A polygon's *winding order* determines which side is the front

  - Most polygons only need one side rendered
  - You can turn on double-sided rendering, at a performance cost

Using coordinate order

- Use the *right-hand rule*:
  - Curl your right-hand fingers around the polygon perimeter in the order vertices are given (counter-clockwise)
  - Your thumb sticks out the front of the polygon



Defining vertices

- A *vertex* describes a polygon corner and contains:
  - A 3D coordinate
  - A color
  - A texture coordinate
  - A lighting *normal vector*

- The 3D coordinate in a vertex is required, the rest are optional

Defining vertices

- A vertex normal defines surface information for *lighting*
  - But the coordinate winding order defines the polygon's front and back, and thus the side that is drawn

- If you want to light your geometry, you must specify vertex lighting normals
  - Lighting normals must be *unit length*

Building geometry

- Java 3D has multiple types of geometry that use 3D coordinates:
  - Points, lines, triangles, and quadrilaterals
  - 3D extruded text
  - Raster image sprites

- Geometry constructors differ in what they build, and how you tell Java 3D to build them

- Let’s look at points, lines, triangles, and quadrilaterals first . . .

GeometryArray class hierarchy

- All geometry types are derived from **`Geometry`**
  - **`GeometryArray`** extends it to build points, lines, triangles, and quadrilaterals

Class Hierarchy
java.lang.Object └─ **javax.media.j3d.SceneGraphObject**   └─ **javax.media.j3d.NodeComponent**   └─ **javax.media.j3d.Geometry**   └─ **javax.media.j3d.GeometryArray**

GeometryArray class methods

- Generic methods on **`GeometryArray`** set coordinates and normals

Method
`void setCoordinate( int index, * coordinate )`
`void setCoordinates( int index, * coordinate )`
`void setNormal( int index, * normal )`
`void setNormals( int index, * normal )`

- Coordinate method variants accept **`float`**, **`double`**, **`Point3f`**, and **`Point3d`**
- Coordinate method variants accept **`float`** and **`Vector3f`**

GeometryArray class methods

- Generic methods on **`GeometryArray`** also set colors and texture coordinates
    - Discussed in the section on shape appearance

Method
`void setColor( int index, * color )`
`void setColors( int index, * color )`
`void setTextureCoordinate( int index, * texCoord )`
`void setTextureCoordinates( int index, * texCoord )`

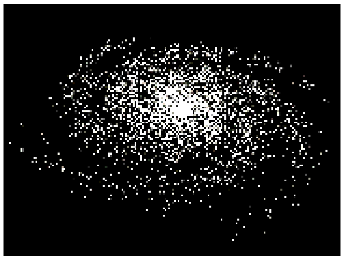
- Color method variants accept **`byte`**, **`float`**, **`Color3f`**, **`Color4f`**, **`Color3b`**, **`Color4b`**, and **`Vector3f`**
- Texture coordinate method variants accept **`float`**, **`Point2f`**, and **`Point3f`**

Building different types of geometry

- There are *14* different geometry array types grouped into:
    - Simple geometry:
        - **`PointArray`**, **`LineArray`**, **`TriangleArray`**, and **`QuadArray`**
    - Strip geometry:
        - **`LineStripArray`**, **`TriangleStripArray`**, and **`TriangleFanArray`**
    - Indexed simple geometry:
        - **`IndexedPointArray`**, **`IndexedLineArray`**, **`IndexedTriangleArray`**, and **`IndexedQuadArray`**
    - Indexed stripped geometry:
        - **`IndexedLineStripArray`**, **`IndexedTriangleStripArray`**, and **`IndexedTriangleFanArray`**

- Let’s look at simple geometry types first . . .

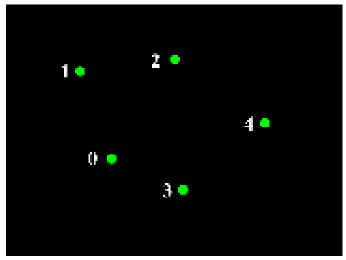
Building a PointArray



- A **`PointArray`** builds points
  - One point at each vertex
  - Point size may be controlled by shape appearance attributes

Class Hierarchy
`java.lang.Object` └─ **`javax.media.j3d.SceneGraphObject`**   └─ **`javax.media.j3d.NodeComponent`**   └─ **`javax.media.j3d.Geometry`**   └─ **`javax.media.j3d.GeometryArray`**   └─ **`javax.media.j3d.PointArray`**

PointArray example code



- Create a list of 3D coordinates for the vertices

```
Point3f[] myCoords = {
    new Point3f(  0.0f, 0.0f, 0.0f ),
    . . .
}
```

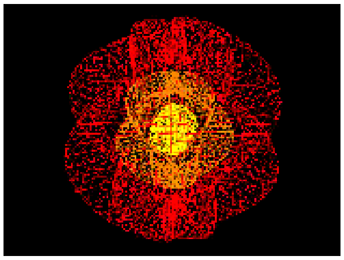
- Create a **`PointArray`** and set the vertex coordinates

```
PointArray myPoints = new PointArray(
    myCoords.length,
    GeometryArray.COORDINATES );
myPoints.setCoordinates( 0, myCoords );
```

- Assemble the shape

```
Shape3D myShape = new Shape3D( myPoints, myAppear );
```

Building a LineArray

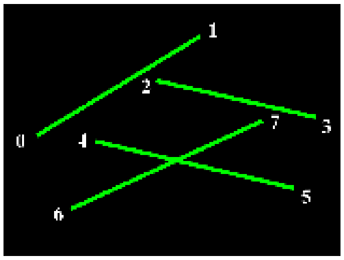


- A **`LineArray`** builds lines
  - Between each *pair* of vertices
  - Line width and style may be controlled by shape appearance attributes

Class Hierarchy

```
java.lang.Object
 └ javax.media.j3d.SceneGraphObject
    └ javax.media.j3d.NodeComponent
       └ javax.media.j3d.Geometry
          └ javax.media.j3d.GeometryArray
             └ javax.media.j3d.LineArray
```

LineArray example code



- Create a list of 3D coordinates for the vertices

```
Point3f[] myCoords = {
    new Point3f(  0.0f, 0.0f, 0.0f ),
    . . .
}
```

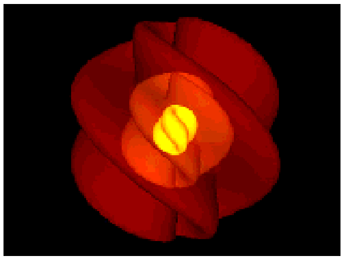
- Create a **`LineArray`** and set the vertex coordinates

```
LineArray myLines = new LineArray(
    myCoords.length,
    GeometryArray.COORDINATES );
myLines.setCoordinates( 0, myCoords );
```

- Assemble the shape

```
Shape3D myShape = new Shape3D( myLines, myAppear );
```

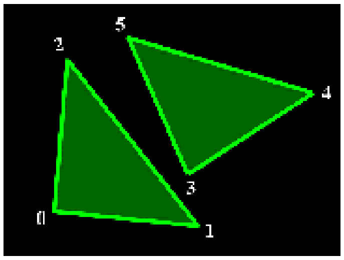
Building a TriangleArray



- A **`TriangleArray`** builds triangles
  - Between each *triple* of vertices
  - Rendering may be controlled by shape appearance attributes

Class Hierarchy
`java.lang.Object`
└ **`javax.media.j3d.SceneGraphObject`**
└ **`javax.media.j3d.NodeComponent`**
└ **`javax.media.j3d.Geometry`**
└ **`javax.media.j3d.GeometryArray`**
└ **`javax.media.j3d.TriangleArray`**

TriangleArray example code



- Create lists of 3D coordinates and normals for the vertices

```
Point3f[] myCoords = {
    new Point3f(  0.0f, 0.0f, 0.0f ),
    . . .
}
Vector3f[] myNormals = {
    new Vector3f(  0.0f, 1.0f, 0.0f ),
    . . .
}
```

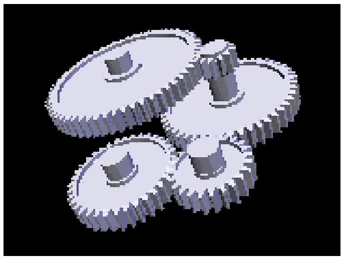
- Create a `TriangleArray` and set the vertex coordinates and normals

```
TriangleArray myTris = new TriangleArray(
    myCoords.length,
    GeometryArray.COORDINATES |
    GeometryArray.NORMALS );
myTris.setCoordinates( 0, myCoords );
myTris.setNormals( 0, myNormals );
```

- Assemble the shape

```
Shape3D myShape = new Shape3D( myTris, myAppear );
```

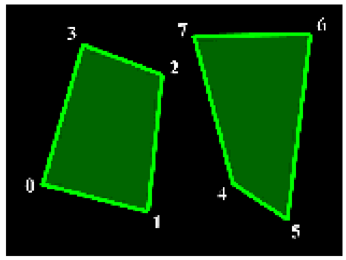
Building a QuadArray



- A **`QuadArray`** builds quadrilaterals
  - Between each *quadruple* of vertices
  - Rendering may be controlled by shape appearance attributes

Class Hierarchy
`java.lang.Object` └─ **`javax.media.j3d.SceneGraphObject`**   └─ **`javax.media.j3d.NodeComponent`**   └─ **`javax.media.j3d.Geometry`**   └─ **`javax.media.j3d.GeometryArray`**   └─ **`javax.media.j3d.QuadArray`**

QuadArray example code



- Create lists of 3D coordinates and normals for the vertices

```
Point3f[] myCoords = {
    new Point3f(  0.0f, 0.0f, 0.0f ),
    . . .
}
Vector3f[] myNormals = {
    new Vector3f(  0.0f, 1.0f, 0.0f ),
    . . .
}
```

- Create a **QuadArray** and set the vertex coordinates and normals

```
QuadArray myQuads = new QuadArray(
    myCoords.length,
    GeometryArray.COORDINATES |
    GeometryArray.NORMALS );
myQuads.setCoordinates( 0, myCoords );
myQuads.setNormals( 0, myNormals );
```

- Assemble the shape

```
Shape3D myShape = new Shape3D( myQuads, myAppear );
```

Building geometry strips

- Simple geometry types use vertices in . . .
  - pairs, triples, and quadruples to build lines, triangles, and quadrilaterals one at a time

- *Strip* geometry uses multiple vertices in . . .
  - A chain to build multiple lines and triangles

  - You provide a coordinate list (as always)

  - You provide lighting normal, color, and texture coordinate lists (optionally)

  - You provide a strip length list
    - Each list entry gives the number of consecutive vertices to chain together

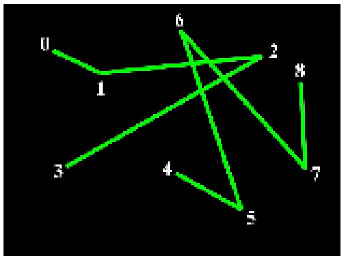
GeometryStripArray class hierarchy

- **`GeometryStripArray`** extends **`GeometryArray`** to build strips of lines and triangles

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
```

Building a LineStripArray



- Create a list of 3D coordinates for the vertices

```
Point3f[] myCoords = {
    new Point3f(  0.0f, 0.0f, 0.0f ),
    . . .
}
```

- Create a list of vertex strip lengths

```
int[] stripLengths = { 4, 5 };
```

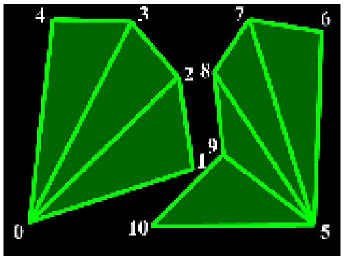
- Create a **LineStripArray** and set the vertex coordinates

```
LineStripArray myLines = new LineStripArray(
    myCoords.length,
    GeometryArray.COORDINATES,
    stripLengths );
myLines.setCoordinates( 0, myCoords );
```

- Assemble the shape

```
Shape3D myShape = new Shape3D( myLines, myAppear );
```

Building a TriangleFanArray



- Create lists of 3D coordinates and lighting normals for the vertices

```
Point3f[] myCoords = {
    new Point3f(  0.0f, 0.0f, 0.0f ),
    . . .
}
Vector3f[] myNormals = {
    new Vector3f(  0.0f, 1.0f, 0.0f ),
    . . .
}
```

- Create a list of vertex fan lengths

```
int[] fanLengths = { 5, 6 };
```

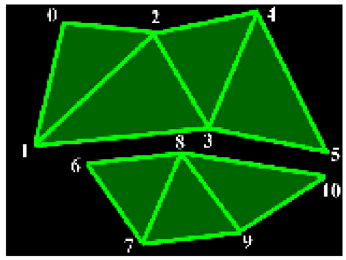
- Create a `TriangleFanArray` and set vertex coordinates and lighting normals

```
TriangleFanArray myFans = new TriangleFanArray(
    myCoords.length,
    GeometryArray.COORDINATES |
    GeometryArray.NORMALS,
    fanLengths );
myFans.setCoordinates( 0, myCoords );
myFans.setNormals( 0, myNormals );
```

- Assemble the shape

```
Shape3D myShape = new Shape3D( myFans, myAppear );
```

Building a TriangleStripArray



- Create lists of 3D coordinates and lighting normals for the vertices

```
Point3f[] myCoords = {
    new Point3f(  0.0f, 0.0f, 0.0f ),
    . . .
}
Vector3f[] myNormals = {
    new Vector3f(  0.0f, 1.0f, 0.0f ),
    . . .
}
```

- Create a list of vertex strip lengths

```
int[] stripLengths = { 6, 5 };
```

- Create a `TriangleStripArray` and set vertex coordinates and lighting normals

```
TriangleStripArray myTris = new TriangleStripArray(
    myCoords.length,
    GeometryArray.COORDINATES |
    GeometryArray.NORMALS,
    stripLengths );
myTris.setCoordinates( 0, myCoords );
myTris.setNormals( 0, myNormals );
```

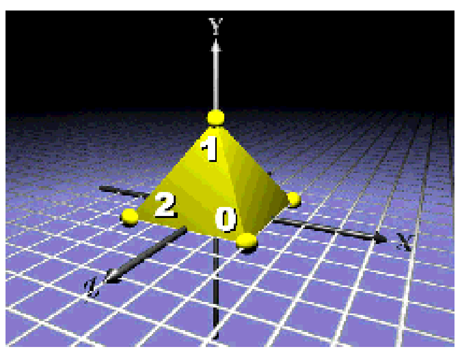
- Assemble the shape

```
Shape3D myShape = new Shape3D( myTris, myAppear );
```

Building indexed geometry

- For surfaces, the same vertices are used for adjacent lines and triangles
  - Simple and strip geometry require *redundant* coordinates, lighting normals, colors, and texture coordinates

- *Indexed* geometry uses *indices* along with the usual lists of coordinates, lighting normals, etc.
  - Indices select coordinates to use from your list
  - Use a coordinate multiple times, but give it only once
  - Indices also used for lighting normals, colors, and texture coordinates



IndexedGeometryArray class hierarchy

- **`IndexedGeometryArray`** extends **`GeometryArray`** to build indexed points, lines, triangles, and quadrilaterals
- **`IndexedGeometryStripArray`** extends **`IndexedGeometryArray`** to build indexed strips of lines and triangles

Class Hierarchy

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

IndexedGeometryArray class methods

- Generic methods on **`IndexedGeometryArray`** set coordinate and lighting normal indices

Method
`void setCoordinateIndex( int index, int value )`
`void setCoordinateIndices( int index, int[] value )`
`void setNormalIndex( int index, int value )`
`void setNormalIndices( int index, int[] value )`

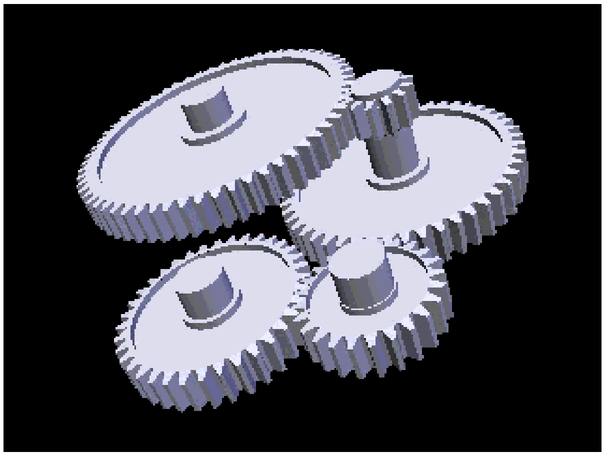
IndexedGeometryArray class methods

- Generic methods on **`IndexedGeometryArray`** also set colors and texture coordinate indices
  - Discussed in the section on shape appearance

Method
`void setColorIndex( int index, int value )`
`void setColorIndices( int index, int[] value )`
`void setTextureCoordinateIndex( int index, int value )`
`void setTextureCoordinateIndices( int index, int[] value )`

Building 3D shapes

Gearbox example



[**GearBox**]

Summary

- A 3D shape is described by:
  - *Geometry:* form or structure
  - *Appearance:* coloration, transparency, shading

- Java 3D has multiple geometry types that all use vertices containing:
  - *Coordinates:* 3D XYZ locations
  - *Normals:* 3D direction vectors
  - *Colors:* red-green-blue mix colors
  - *Texture coordinates:* 2D ST texture image locations

Summary

- Simple geometry types build points, lines, triangles, and quadrilaterals
    - Automatically using vertices in sets of 1, 2, 3, or 4

- Strip geometry types build lines and triangles
    - Using vertices in user-defined chains

- Indexed geometry types build points, lines, triangles, and quadrilaterals
    - Using coordinates, lighting normals, etc. selected by indices

Summary

- Java 3D also provides a couple more geometry types, including:
    - *Raster geometry*, discussed later this morning
    - *Text geometry*, discussed in the extended notes, but not during the tutorial

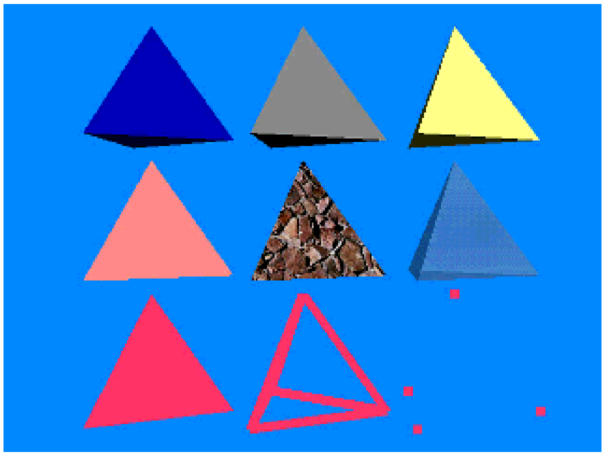
Controlling appearance

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 a shape's geometry

Class Hierarchy
java.lang.Object
└ **javax.media.j3d.SceneGraphObject**
└ **javax.media.j3d.NodeComponent**
└ **javax.media.j3d.Appearance**

Introducing appearance attributes

- Appearance attributes are grouped into several node components:
  - Color and transparency control
    - **`Material`**
    - **`ColoringAttributes`**
    - **`TransparencyAttributes`**

  - Rendering control
    - **`PointAttributes`**
    - **`LineAttributes`**
    - **`PolygonAttributes`**
    - **`RenderingAttributes`**

  - Texture control (discussed later)
    - **`Texture`**
    - **`TextureAttributes`**
    - **`TexCoordGeneration`**

Appearance attributes class hierarchy

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

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

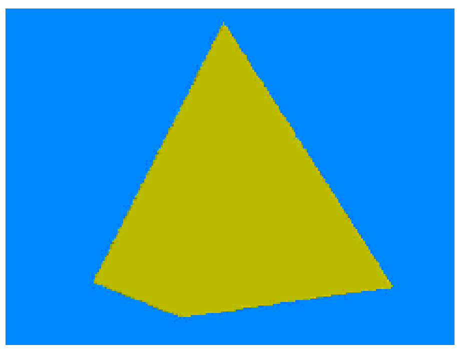
- Methods on **`Appearance`** just set which attributes to use
  - Set only the ones you need, leaving the rest at their default values

Method
`Appearance( )`
`void setColoringAttributes( ColoringAttributes coloringAttributes )`
`void setMaterial( Material material )`
`void setTransparencyAttributes( TransparencyAttributes transparencyAttributes )`
`void setLineAttributes( LineAttributes lineAttributes )`
`void setPointAttributes( PointAttributes pointAttributes )`
`void setPolygonAttributes( PolygonAttributes polygonAttributes )`
`void setRenderingAttributes( RenderingAttributes renderingAttributes )`

Using coloring attributes

- `ColoringAttributes` controls:
  - Intrinsic color (used when lighting is disabled)
  - Shading model (flat or Gouraud)

- Use coloring attributes when a shape *is not* shaded
  - Emissive points, lines, and polygons
  - Avoids expensive shading calculations



ColoringAttributes class methods

- Methods on **`ColoringAttributes`** select the color and shading model
    - The default color is white, and the default shading model **`SHADE_GOURAUD`**

Method
`ColoringAttributes( )`
`void setColor( Color3f color )`
`void setShadeModel( int model )`

- Shade models include: **`SHADE_FLAT`** and **`SHADE_GOURAUD`** (default)

- The **`FASTEST`** and **`NICEST`** shade models automatically select the fastest, and highest quality models available

ColoringAttributes example code

- Create **`ColoringAttributes`** to set an intrinsic color and shading model

```
ColoringAttributes myCA = new ColoringAttributes( );
myCA.setColor( 1.0f, 1.0f, 0.0f );
myCA.setShadeModel( ColoringAttributes.SHADE_GOURAUD
```

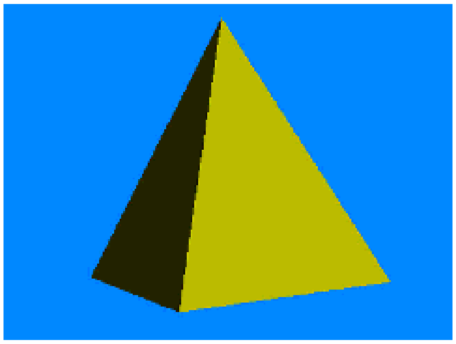
- Create **`Appearance`**, set the coloring attributes, and assemble the shape

```
Appearance myAppear = new Appearance( );
myAppear.setColoringAttributes( myCA );
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

Using material attributes

- `Material` controls:
  - Ambient, emissive, diffuse, and specular color
  - Shininess factor

- Use materials when a shape *is* shaded
  - Most scene shapes
  - Overrides `ColoringAttributes` intrinsic color (when lighting is enabled)

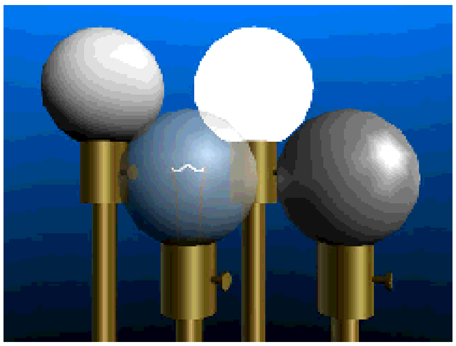


Using material colors

- *Diffuse color* sets the main shading color, giving a dull, matte finish (upper-left)

- *Specular color* and *shininess factor* make a shape appear shiny (lower-right)

- *Emissive color* makes a shape appear to glow (upper-right)



Material class methods

- Methods on **`Material`** set shading colors and turn on/off lighting effects
  - Defaults include white diffuse and specular colors, a black emissive color, (0.2,0.2,0.2) ambient color, shininess of 64.0, and lighting enabled

Method
`Material( )`
`void setAmbientColor( Color3f color )`
`void setEmissiveColor( Color3f color )`
`void setDiffuseColor( Color3f color )`
`void setSpecularColor( Color3f color )`
`void setShininess( float shininess )`
`void setLightingEnable( boolean state )`

Material attributes example code

- Create **`Material`** to set shape colors

```
Material myMat = new Material( );
myMat.setAmbientColor(  0.3f, 0.3f, 0.3f );
myMat.setDiffuseColor(  1.0f, 0.0f, 0.0f );
myMat.setEmissiveColor( 0.0f, 0.0f, 0.0f );
myMat.setSpecularColor( 1.0f, 1.0f, 1.0f );
myMat.setShininess( 64.0f );
```

- Create **`Appearance`**, set the material, and assemble the shape

```
Appearance myAppear = new Appearance( );
myAppear.setMaterial( myMat );
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

Using coordinate colors

- You may also set a color for each geometry coordinate in a **`GeometryArray`**
  - Coordinate colors override coloring attributes or a material's diffuse color

Method
`void setColor( int index, * color )`
`void setColors( int index, * color )`

- Method variants accept **`byte`**, **`float`**, **`Color3f`**, **`Color4f`**, **`Color3b`**, and **`Color4b`**

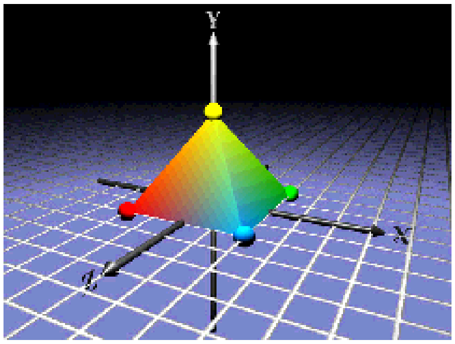
Using coordinate color indices

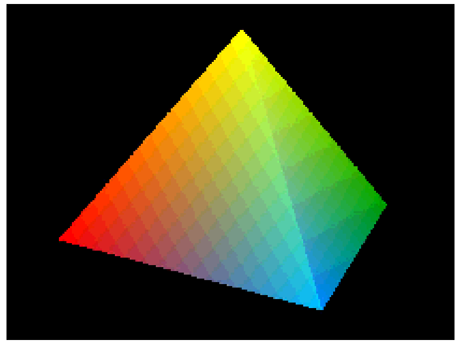
- For indexed geometry, you may select color indices in an **`IndexedGeometryArray`**

Method
`void setColorIndex( int index, int value )`
`void setColorIndices( int index, int[] value )`

Coloring coordinates

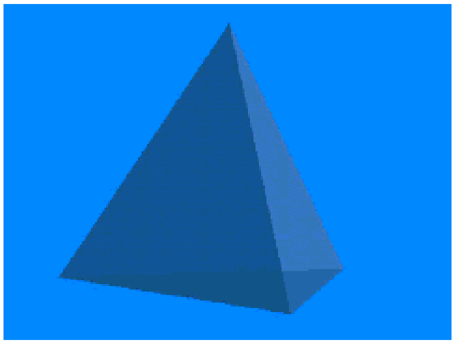
- Coordinate colors are interpolated along lines or across polygons

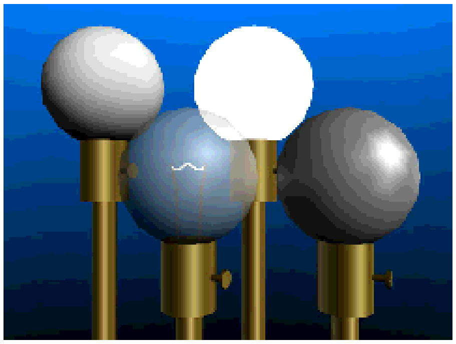




Using transparency attributes

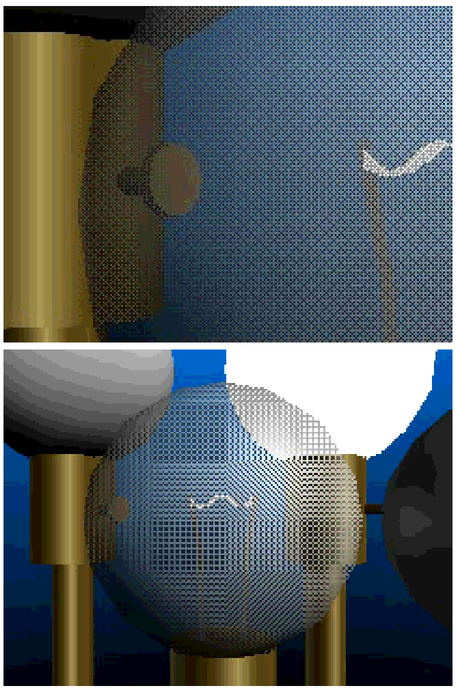
- `TransparencyAttributes` controls:
  - Transparency amount (0.0 = opaque, 1.0 = invisible)
  - Transparency mode (screen-door, alpha-blend, none)





Using transparency modes

- The transparency mode selects between **SCREEN_DOOR** or **BLENDED** transparency



SCREEN_DOOR

BLENDED

TransparencyAttributes class methods

- Methods on **`TransparencyAttributes`** set the transparency
  - By default, transparency is 0.0 (opaque) with a **`FASTEST`** transparency mode

Method
`TransparencyAttributes( )`
`void setTransparency( float transparency )`
`void setTransparencyMode( int mode )`

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

TransparencyAttributes example code

- Create **`TransparencyAttributes`** to set the transparency amount and mode

```
TransparencyAttributes myTA = new TransparencyAttribu
myTA.setTransparency( 0.5f );
myTA.setTransparencyMode( TransparencyAttributes.BLEN
```

- Create **`Appearance`**, set the transparency attributes, and assemble the shape

```
Appearance myAppear = new Appearance( );
myAppear.setTransparencyAttributes( myTA );
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

Using point and line attributes

- **`PointAttributes`** controls:
  - Point size (in pixels)
  - Point anti-aliasing

- **`LineAttributes`** controls:
  - Line width (in pixels)
  - Line dot/dash pattern
  - Line anti-aliasing

PointAttributes class methods

- Methods on **`PointAttributes`** select the way points are rendered
  - By default, the point size is 1.0 and anti-aliasing is disabled

Method
`PointAttributes( )`
`void setPointSize( float size )`
`void setPointAntialiasingEnable( boolean state )`

LineAttributes class methods

- Methods on **`LineAttributes`** select the way lines are rendered
    - By default, the line width is 1.0, the pattern is **`PATTERN_SOLID`**, and anti-aliasing is disabled

Method
`LineAttributes( )`
`void setLineWidth( float width )`
`void setLinePattern( int pattern )`
`void setLineAntialiasingEnable( boolean state )`

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

PointAttributes example code

- Create **`PointAttributes`** to set the point size and anti-aliasing

```
PointAttributes myPA = new PointAttributes( );
myPA.setPointSize( 10.0f );
myPA.setPointAntialiasingEnable( true );
```

- Create **`Appearance`**, set the point attributes, and assemble the shape

```
Appearance myAppear = new Appearance( );
myAppear.setPointAttributes( myPA );
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

LineAttributes example code

- Create **LineAttributes** to set the line width, pattern, and anti-aliasing

```
LineAttributes myLA = new LineAttributes( );
myLA.setLineWidth( 10.0f );
myLA.setLinePattern( LineAttributes.PATTERN_SOLID );
myLA.setLineAntialiasingEnable( true );
```

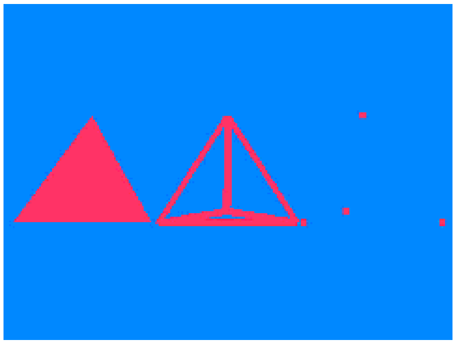
- Create **Appearance**, set the line attributes, and assemble the shape

```
Appearance myAppear = new Appearance( );
myAppear.setLineAttributes( myLA );
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

Controlling appearance

Using polygon attributes

- **`PolygonAttributes`** controls:
  - Face culling (front, back, neither)
  - Fill mode (point, line, fill)
  - Z offset



PolygonAttributes class methods

- Methods on **`PolygonAttributes`** select the way polygons are rendered
  - By default, back faces are culled, polygons are filled, and the offset is 0.0

Method
`PolygonAttributes( )`
`void setCullFace( int cullface )`
`void setPolygonMode( int mode )`
`void setPolygonOffset( float offset )`

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

PolygonAttributes example code

- Create **`PolygonAttributes`** to set the culling mode and fill style

```
PolygonAttributes myPA = new PolygonAttributes( );
myPA.setCullFace( PolygonAttributes.CULL_NONE );
myPA.setPolygonMode( PolygonAttributes.POLYGON_FILL )
```

- Create **`Appearance`**, set the polygon attributes, and assemble the shape

```
Appearance myAppear = new Appearance( );
myAppear.setPolygonAttributes( myPA );
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

Using rendering attributes

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

RenderingAttributes class methods

- Methods on **`RenderingAttributes`** control the way everything is rendered
  - By default, the depth buffer is enabled and writable, and the alpha test function is **`ALWAYS`** with a 0.0 alpha test value

Method
`RenderingAttributes( )`
`void setDepthBufferEnable( boolean state )`
`void setDepthBufferWriteEnable( boolean state )`
`void setAlphaTestFunction( int func )`
`void setAlphaTestValue( float value )`

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

RenderingAttributes example code

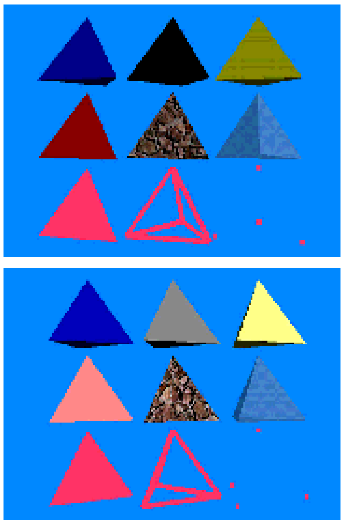
- Create **`RenderingAttributes`** to set the depth and alpha modes

```
RenderingAttributes myRA = new RenderingAttributes( )
myRA.setDepthBufferEnable( false );
myRA.setAlphaTestFunction( RenderingAttributes.NEVER
```

- Create **`Appearance`**, set the rendering attributes, and assemble the shape

```
Appearance myAppear = new Appearance( );
myAppear.setRenderingAttributes( myRA );
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

Appearance example



[**ExAppearance**]

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

Summary

- **`Appearance`** groups together appearance attributes for a **`Shape3D`**

- Color and transparency control
    - **`ColoringAttributes`**
        - Non-shading color and shading model
    - **`Material`**
        - Ambient, diffuse, emissive, and specular colors
        - Lighting enable/disable
    - **`GeometryArray`** and **`IndexedGeometryArray`**
        - Color per coordinate
    - **`TransparencyAttributes`**
        - Transparency amount and mode

Summary

- Rendering control
  - **`PointAttributes`**
    - Point size and anti-aliasing
  - **`LineAttributes`**
    - Line width, pattern, and anti-aliasing
  - **`PolygonAttributes`**
    - Polygon culling and draw style
  - **`RenderingAttributes`**
    - Depth and alpha buffer use

Grouping shapes

Motivation

- Recall that a scene graph is a hierarchy of groups
  - Shapes, lights, sounds, etc.
  - Groups of groups of groups of . . .

- Java 3D has several types of groups
  - Some simply group their children
  - Others provide added functionality

Introducing grouping types

- Java 3D's grouping nodes include:
  - **`Group`**
  - **`BranchGroup`**
  - **`OrderedGroup`**
  - **`DecalGroup`**
  - **`Switch`**
  - **`SharedGroup`**
  - **`TransformGroup`**

- All groups manage a list of children nodes

- For most groups, Java 3D may render children *in any order*

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`**

Creating groups

- **`Group`** is the most general-purpose grouping node

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

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

Group class methods

- Methods on **`Group`** control group content

Method
`Group( )`
`void addChild( Node child )`
`void setChild( Node child, int index )`
`void insertChild( Node child, int index )`
`void removeChild( int index )`

Group example code

- Build a shape

```
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

- Add it to a group

```
Group myGroup = new Group( );
myGroup.addChild( myShape );
```

Creating branch groups

- **`BranchGroup`** extends **`Group`** and creates a *branch graph*, a major branch in the scene graph
  - Can be attached to a **`Locale`** (or **`SimpleUniverse`**)
  - Can be compiled
  - Can be a child of any grouping node
  - Can detach itself from its parent (if that parent has appropriate *capabilities* enabled)

- Adding a **`BranchGroup`** to a **`Locale`** makes it *live*
  - Once live or compiled, changes are constrained to those enabled by *capabilities*

BranchGroup class methods

- In addition to **`Group`**'s methods, **`BranchGroup`** provides compilation and membership control

Method
`BranchGroup( )`
`void compile( )`
`void detach( )`

BranchGroup example code

- Build a locale in a universe

```
Locale myLocale = new Locale( myUniverse );
```

- Build a shape

```
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

- Add the shape to a branch group

```
BranchGroup myBranch = new BranchGroup( );
myBranch.addChild( myShape );
```

- Add the branch group to the locale

```
myLocale.addBranchGraph( myBranch );
```

Summary

- All groups can have children set, added, inserted, and removed
- All groups can have any number of children
- **`Group`** does nothing more
  - All children rendered
  - Rendered in any order
- **`BranchGroup`** can compile its children for faster rendering
  - All children rendered
  - Rendered in any order

Transforming shapes

Motivation

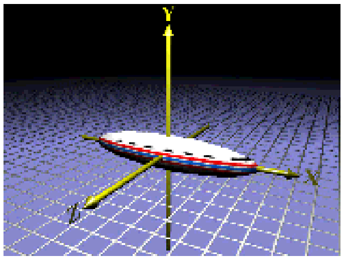
- By default, all shapes are built within a shared *world coordinate system*

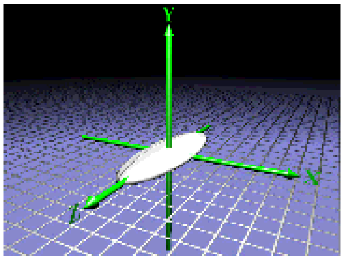
- A **`TransformGroup`** builds a new coordinate system for its children, *relative* to its parent
    - *Translate* to change relative position
    - *Rotate* to change relative orientation
    - *Scale* to change relative size
    - Use in combination

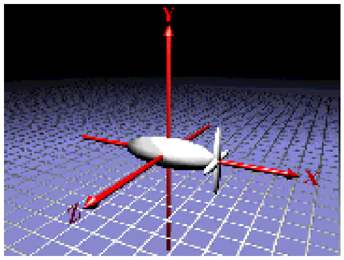
- Shapes built in the new coordinate system are relative to it
    - If you translate the coordinate system, the shapes move too

Using coordinate systems

- Recall the toy airplane . . . its parts are each built in their own coordinate system

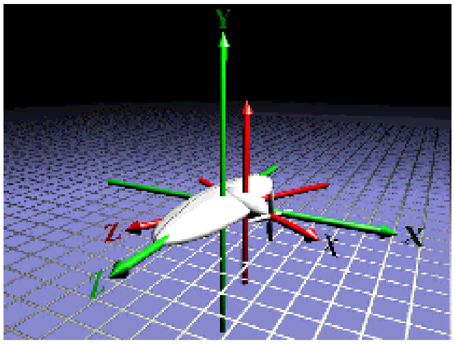


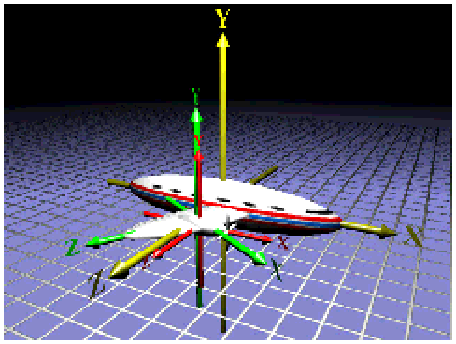




Using coordinate systems

- Those parts are assembled, bringing a child shape into a parent's coordinate system

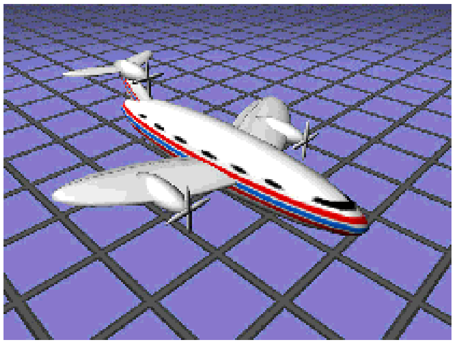


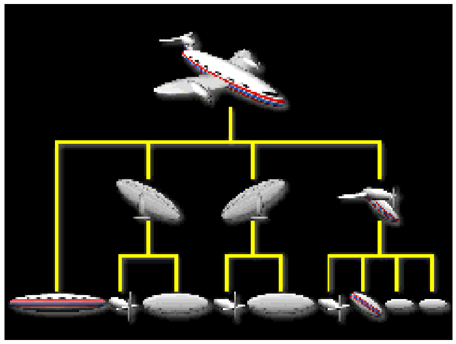


Transforming shapes

Using coordinate systems

- And so on, to build the full toy airplane





Creating transform groups

- Transforms can be arbitrarily *nested* to include one **`TransformGroup`** within another
- Transforms "closer" to the geometry (deeper nesting in the scene graph) apply first

TransformGroup class hierarchy

- **`TransformGroup`** extends **`Group`** and builds a transformed coordinate system for its children

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

- In addition to `Group`'s methods, `TransformGroup` adds a 3D transform
  - The default transform is *identity*, which does no translation, rotation, or scaling

Method
`TransformGroup( )`
`void setTransform( Transform3D xform )`

Creating a 3D transform

- A `Transform3D` describes the actual translation, rotation, and scaling

- 3D transforms are internally represented as a 4x4 matrix
  - You can set the matrix directly

  - Most people will use helper methods to do translation, rotation, and scaling

Transform3D class hierarchy

- **`Transform3D`** extends **`Object`**

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

Transform3D class methods

- At the most basic level, methods on **`Transform3D`** create and set the underlying 4x4 matrix

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

Abiding by Transform3D restrictions

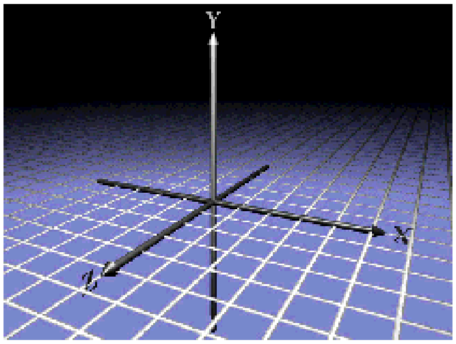
- A 3D transform must be *affine*
  - No perspective-like homogeneous division, such as for hyperbolic spaces

- A 3D transform must be *congruent* if used in a **`TransformGroup`** above a **`ViewPlatform`**
  - No non-uniform scaling of the viewpoint
  - **`ViewPlatform`** is discussed later in the tutorial

Resetting a transform

- Setting the transform to identity does a reset
  - Zero translation in X, Y, and Z
  - No rotation
  - Scale factor of 1.0 in X, Y, and Z

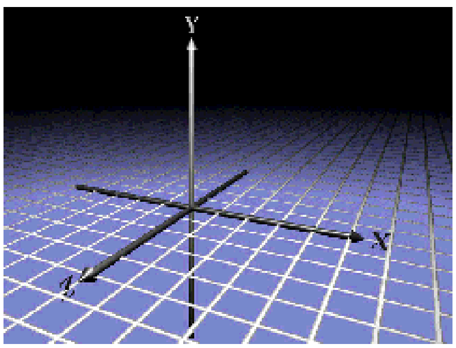
Method
`void setIdentity( )`

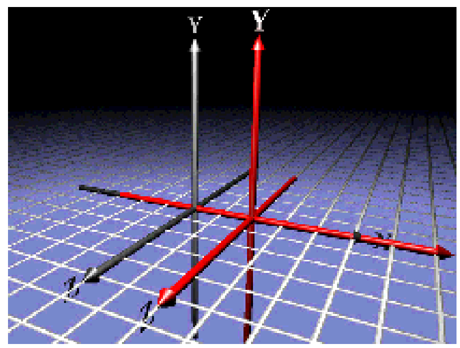


Translating a coordinate system

- Translation moves the coordinate system and its shapes
  - A direction **`Vector3d`** gives X, Y, and Z distances

Method
`void set( Vector3d trans )`





TransformGroup example code

- Build a shape

```
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

- Create a 3D transform for a +1.0 translation in X

```
Transform3D myTrans3D = new Transform3D( );
myTrans3D.set( new Vector3d( 1.0, 0.0, 0.0 ) );
```

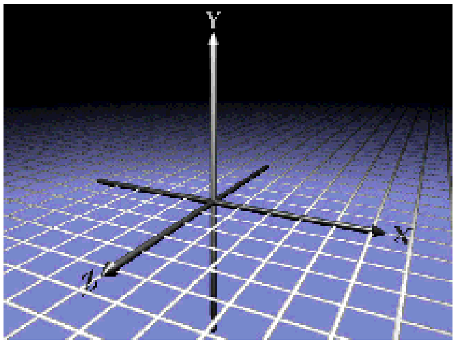
- Create a transform group, set the transform, and add the shape

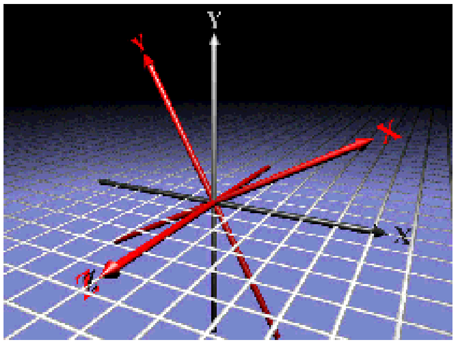
```
TransformGroup myGroup = new TransformGroup( );
myGroup.setTransform( myTrans3D );
myGroup.addChild( myShape );
```

Rotating a coordinate system

- Rotation orients the coordinate system and its shapes
  - Rotate about X, Y, or Z by an angle
  - Rotate about an arbitrary axis

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





TransformGroup example code

- Build a shape, as before

```
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

- Create a 3D transform for a Z-axis rotation by 30 degrees (0.52 radians)

```
Transform3D myTrans3D = new Transform3D( );
myTrans3D.rotZ( 0.52 ); // 30 degrees
```

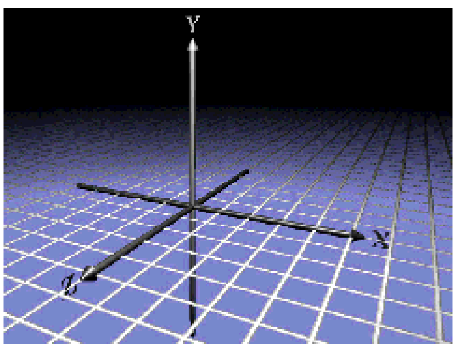
- Create a transform group, set the transform, and add the shape

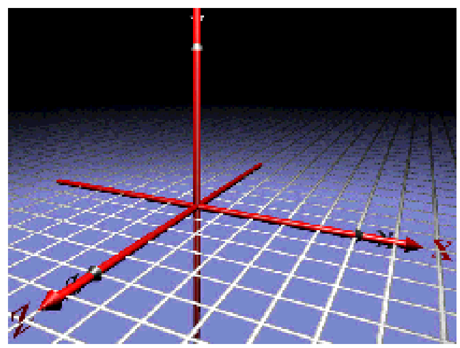
```
TransformGroup myGroup = new TransformGroup( );
myGroup.setTransform( myTrans3D );
myGroup.addChild( myShape );
```

Scaling a coordinate system

- Scaling grows or shrinks the coordinate system and its shapes
  - Use a single scale factor for uniform scaling
  - Use X, Y, and Z scale factors for non-uniform scaling

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





TransformGroup example code

- Build a shape, as before

```
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

- Create a 3D transform for scaling by 1.5 in X, Y, and Z

```
Transform3D myTrans3D = new Transform3D( );
myTrans3D.set( 1.5 );
```

- Create a transform group, set the transform, and add the shape

```
TransformGroup myGroup = new TransformGroup( );
myGroup.setTransform( myTrans3D );
myGroup.addChild( myShape );
```

Modifying parts of transforms

- Modify *parts* of an existing transform
  - Leave the rest of the transform unaffected
  - Used to combine translation, rotation, and scaling

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 )`

Summary

- `Transform3D` describes translation, rotation, and scaling

- A transform may be built from a 4x4 matrix, or by helper methods

- `TransformGroup` creates a new coordinate system for its children, transformed by a `Transform3D`
  - All children rendered
  - Rendered in any order

Using special-purpose groups

Motivation

- Java 3D includes several more types of groups
  - `Group`
  - `BranchGroup`
  - `OrderedGroup`
  - `DecalGroup`
  - `Switch`
  - `SharedGroup`
  - `TransformGroup`

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`**

Creating ordered groups

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

Creating decal groups

- **`DecalGroup`** extends **`OrderedGroup`** and renders children in *first-to-last 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. airport runway markings)
  - Good for avoiding Z-fighting artifacts

OrderedGroup and DecalGroup class methods

- Neither class provides methods beyond the basics

Method
`OrderedGroup( )`

Method
`DecalGroup( )`

DecalGroup example code

- Build an underlying surface shape, and decal shapes

```
Shape3D underly = new Shape3D( geom0, app0 );
Shape3D decal_1 = new Shape3D( geom1, app1 );
Shape3D decal_2 = new Shape3D( geom2, app2 );
```

- Add them to a decal group, starting with the underlying surface

```
DecalGroup myDecals = new DecalGroup( );
myDecals.addChild( underly );  // First!
myDecals.addChild( decal_1 );
myDecals.addChild( decal_2 );
```

Creating switch groups

- **`Switch`** extends **`Group`** and selects zero, one, or multiple children to render or process
  - Child choice can be by number, or by a bit mask
  - Only selected children are rendered (shapes) or processed (lights, fog, backgrounds, behaviors)

- Similar to a Java "switch" statement

- Java 3D is still free to render children in any order

Switch class methods

- In addition to **`Group`**'s methods, **`Switch`** enables child rendering control

Method
`Switch( )`
`void setWhichChild( int index )`
`void setChildMask( BitSet mask )`

- Remember to use . . .

```
setCapability( Switch.ALLOW_SWITCH_WRITE );
```

  . . . to enable the switch value to be changed while it is live or compiled

Selecting switch children

- Select which child to render by:
  - Passing its child index to **`setWhichChild( )`**
  - Or by passing in a special value:
    - 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 example code

- Build children

```
Shape3D zero = new Shape3D( geom0, app0 );
Shape3D one  = new Shape3D( geom1, app1 );
Shape2D two  = new Shape2D( geom2, app2 );
```

- Add them to the switch group

```
Switch mySwitch = new Switch( );
mySwitch.setCapability( Switch.ALLOW_SWITCH_WRITE );
mySwitch.addChild( zero );
mySwitch.addChild( one );
mySwitch.addChild( two );
```

Switch example code

- Select a single child of the switch group

```
mySwitch.setWhichChild( 2 );
```

- Select all children of the switch group

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

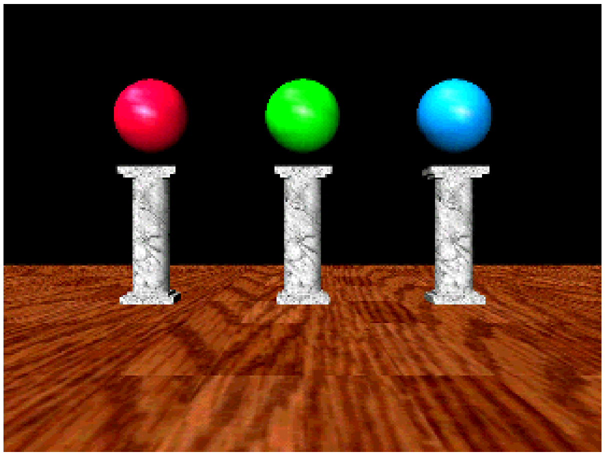
Switch example code

- Select a set of children of the switch group

```
BitSet mask = new BitSet(3);
mask.set( 0 );
mask.set( 2 );

mySwitch.setWhichChild( Switch.CHILD_MASK );
mySwitch.setChildMask( mask );
```

Switch example



[**ExSwitch**]

Creating shared groups

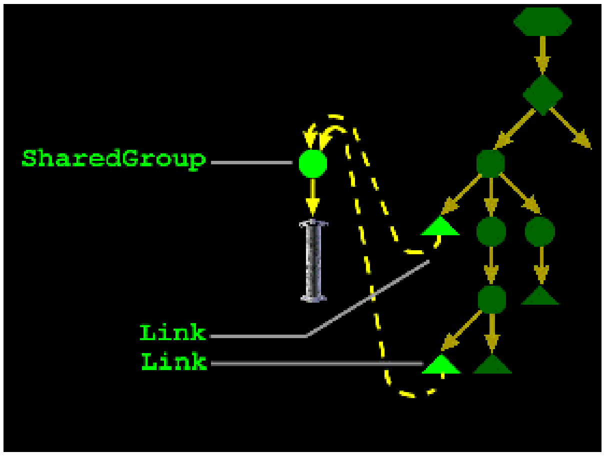
- **`SharedGroup`** extends **`Group`** to create a group of shapes that can be *shared* (used multiple times throughout a scene graph)
  - It contains shapes, like other groups
  - It is *never* added into the scene graph directly
  - It is referenced by one or more **`Link`** leaf nodes
- Changes to a **`SharedGroup`** affect all references to it
- Can be compiled prior to referencing it from a **`Link`** node

Example

[**ExLinearFog**]

Linking to shared groups

- In the example, the column is in a **`SharedGroup`**
- Each visible column uses a **`Link`** to that group



SharedGroup and Link class hierarchy

- **`Link`** extends **`Leaf`** to point to a **`SharedGroup`**

Class Hierarchy
java.lang.Object
└─ **javax.media.j3d.SceneGraphObject**
└─ **javax.media.j3d.Node**
└─ **javax.media.j3d.Leaf**
└─ **javax.media.j3d.Link**

SharedGroup class methods

- In addition to **`Group`**'s methods, **`SharedGroup`** adds a compilation method

Method
`SharedGroup( )`
`void compile( )`

Link class methods

- Methods on **`Link`** select the shared group to link to

Method
`Link( )`
`Link( SharedGroup group )`
`void setSharedGroup( SharedGroup group )`

SharedGroup example code

- Build one or more shapes to share

```
Shape3D myShape = new Shape3D( myGeom, myAppear );
```

- Create a **`SharedGroup`** and add the shapes to it

```
SharedGroup myShared = new SharedGroup( );
myShared.addChild( myShape );
```

- Compile the **`SharedGroup`** for maximum performance

```
myShared.compile( );
```

- Use **`Link`** nodes to point to the group from another group

```
Link myLink = new Link( myShared );
TransformGroup myGroup = new TransformGroup( );
myGroup.addChild( myLink );
```

Using special-purpose groups

SharedGroup example

[**ExLinearFog**]

Summary

- All groups can have children set, added, inserted, and removed

- All groups can have any number of children

- `Group` does nothing more
  - All children rendered
  - Rendered in any order

- `BranchGroup` can compile its children for faster rendering
  - All children rendered
  - Rendered in any order

Summary

- **`OrderedGroup`** forces a rendering order
  - All children rendered
  - Rendered in first-to-last order

- **`DecalGroup`** forces a rendering order for shapes atop an underlying shape
  - All children rendered
  - Rendered in first-to-last order

- **`Switch`** selects zero, one, or multiple children to render or process
  - Selected children rendered
  - Rendered in any order

Summary

- **`SharedGroup`** creates a group of shared shapes
    - All children rendered if the group is referenced by a live link node
    - Rendered in any order

- **`SharedGroup`** nodes are *never* placed directly in a live scene graph

- **`Link`** points to a shared group from a live scene graph
    - Any number of links to the same shared group

Introducing texture mapping

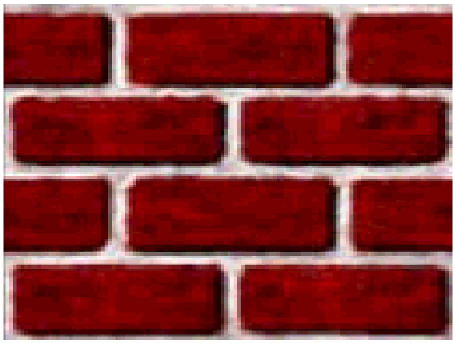
Motivation

- You could model every detail of every 3D shape in your scene
  - This requires an enormous amount of modeling effort
  - More shapes means more to draw and worse interactivity

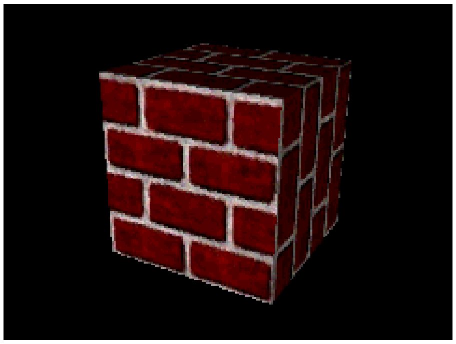
- Instead, create the *illusion* of detail:
  - Take a photograph of the "real thing"
  - Paste that photo onto simple 3D geometry

- Increases realism without increasing the amount of geometry to draw

Example



Texture image



[**`ExTexture`**]

Using texture appearance attributes

- Recall that **`Appearance`** is a container for multiple visual attributes for a shape
  - Color and transparency control (discussed earlier)
    - **`Material`**
    - **`ColoringAttributes`**
    - **`TransparencyAttributes`**

  - Rendering control (discussed earlier)
    - **`PointAttributes`**
    - **`LineAttributes`**
    - **`PolygonAttributes`**
    - **`RenderingAttributes`**

  - Texture control
    - **`Texture`**
    - **`TextureAttributes`**
    - **`TexCoordGeneration`**

Using texture appearance attributes

- Texture control attributes are divided among a few node components
  - **`Texture`**
    - Select a texture image and control basic mapping attributes
  - **`TextureAttributes`**
    - Control advanced mapping attributes
  - **`TexCoordGeneration`**
    - Automatically generate texture coordinates if you do not provide your own (most people provide their own)

Texture class hierarchy

- **`Texture`** is the base class for two node components that select the image to use
  - **`Texture2D`**: a standard 2D image
  - **`Texture3D`**: a 3D volume of images

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 class methods

- Methods on **`Texture`** and **`Texture2D`** select the image, and turn texture mapping on and off

Method
`Texture( )`
`Texture2D( )`
`void setImage( int level, ImageComponent2D image )`
`void setEnable( boolean onOff )`

Texture2D example code

- Load a texture image (discussed later)

```
TextureLoader myLoader = new TextureLoader( "brick.jpg
ImageComponent2D myImage = myLoader.getImage( );
```

- Create a **`Texture2D`** using the image, and turn it on

```
Texture2D myTex = new Texture2D( );
myTex.setImage( 0, myImage );
myTex.setEnable( true );
```

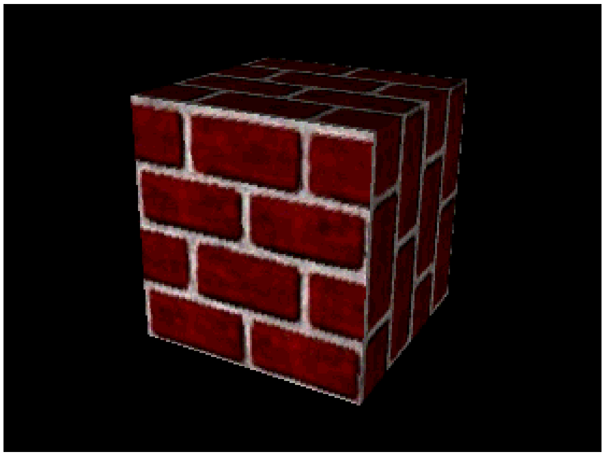
- Create an **`Appearance`** and set the texture in it

```
Appearance myAppear = new Appearance( );
myAppear.setTexture( myTex );
```

- Assemble the shape

```
Shape3D myShape = new Shape3D( myText, myAppear );
```

Texture example



[**`ExTexture`**]

Preparing for texture mapping

- Getting a texture requires:
  - A file to load from disk or the Web
  - A **`TextureLoader`** to load that file
  - An **`ImageComponent`** to hold the loaded image
    - Which in turn uses a standard **`BufferedImage`**

ImageComponent class hierarchy

- **`ImageComponent`** is the base class for two image containers:
  - **`ImageComponent2D`** holds a 2D image
  - **`ImageComponent3D`** holds a 3D volume of images

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

- Methods on **`ImageComponent2D`** set the image it is holding

Method
`ImageComponent2D( int format, BufferedImage image )`
`void set( BufferedImage image )`

Loading texture images

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

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

TextureLoader example code

- Load a texture image

```
TextureLoader myLoader = new TextureLoader( "brick.jpg
ImageComponent2D myImage = myLoader.getImage( );
```

- Create a **Texture2D** using the image, and turn it on

```
Texture2D myTex = new Texture2D( );
myTex.setImage( 0, myImage );
myTex.setEnable( true );
```

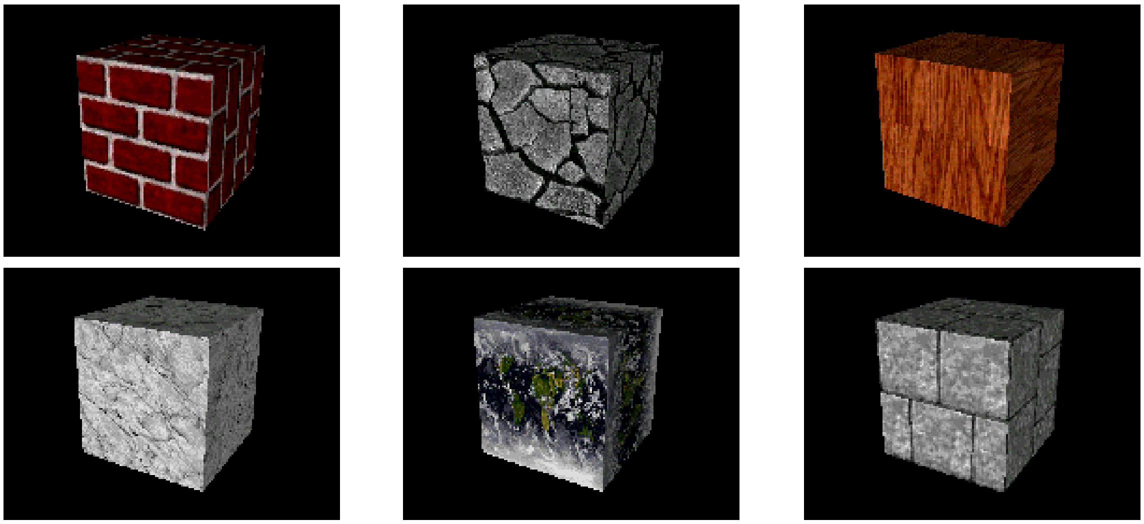
- Create an **Appearance** and set the texture in it

```
Appearance myAppear = new Appearance( );
myAppear.setTexture( myTex );
```

- Assemble the shape

```
Shape3D myShape = new Shape3D( myText, myAppear );
```

TextureLoader example



[ExTexture]

Summary

- A *texture* is an image pasted onto a shape to create the illusion of detail

- Texture mapping is controlled by node components in a shape's **`Appearance`** including **`Texture2D`**
  - Enables texture mapping using an image in an **`ImageComponent2D`**

- **`TextureLoader`** gets an image from disk or the Web, returning an **`ImageComponent`**

- **`ImageComponent2D`** holds 2D image data

Using texture coordinates

Motivation

- We need a mapping from parts of a texture to parts of a shape
  - Define a "texture cookie cutter" to cut out a texture piece
  - Translate, rotate, and scale the cookie cutter before cutting out the piece
  - Map the cut out texture "cookie" onto your shape
- *Texture coordinates* describe the 2D shape of that cookie cutter

Using a texture coordinate system

- Texture images have a *true size* and a *logical size*

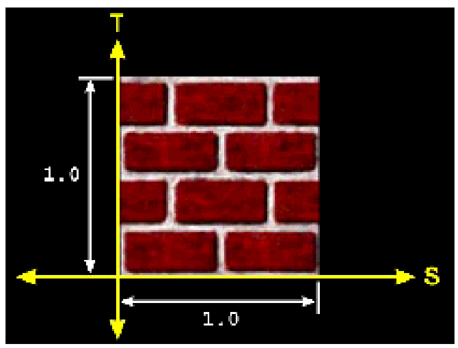
  - True size is the width and height of the image in pixels
    - Must be powers of 2
    - Width and height need not be the same

  - Logical size is a generic treatment of image dimensions
    - *Always* a width of 1.0
    - *Always* a height of 1.0

Using a texture coordinate system

- Textures can be visualized as in a 2D *texture coordinate system*
  - The horizontal dimension is *S*
  - The vertical dimension is *T*

- An image extends from 0.0 to 1.0 in S and T, regardless of the true size



Specifying texture coordinates

- Texture coordinates define a 2D shape atop the texture image
  - A "texture cookie cutter"

- There must be one ST pair for each shape coordinate
  - Give texture coordinates to **`GeometryArray`**, and texture coordinate indices to **`IndexedGeometryArray`**

GeometryArray class methods

- Methods on **`GeometryArray`** set texture coordinates

Method
`void setTextureCoordinate( int index, * texCoord )`
`void setTextureCoordinates( int index, * texCoord )`

- Method variants accept **`float`**, **`Point2f`**, and **`Point3f`**

IndexedGeometryArray class methods

- Methods on **`IndexedGeometryArray`** set texture coordinate indices

Method
`void setTextureCoordinateIndex( int index, int value )`
`void setTextureCoordinateIndices( int index, int[] value )`

Texture coordinates example code

- Create lists of 3D coordinates, lighting normals, and texture coordinates for the vertices

```
Point3f[] myCoords = {
    new Point3f( 0.0f, 0.0f, 0.0f ),
    . . .
}
Vector3f[] myNormals = {
    new Vector3f( 0.0f, 1.0f, 0.0f ),
    . . .
}
Point2f[] myTexCoords = {
    new Point2f( 0.0f, 0.0f ),
    . . .
}
```

- Create a **QuadArray** and set the vertex coordinates, lighting normals, and texture coordinates

```
QuadArray myQuads = new QuadArray(
    myCoords.length,
    GeometryArray.COORDINATES |
    GeometryArray.NORMALS |
    GeometryArray.TEXTURE_COORDINATE_2 );
myQuads.setCoordinates( 0, myCoords );
myQuads.setNormals( 0, myNormals );
myQuads.setTextureCoordinates( 0, myTexCoords );
```

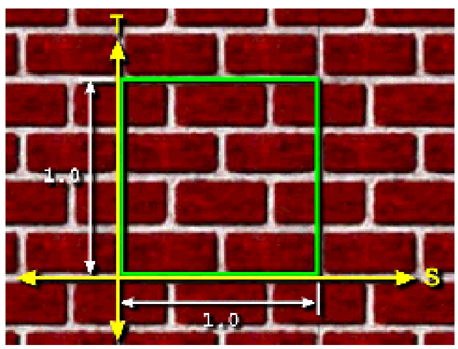
- Assemble the shape

```
Shape3D myShape = new Shape3D( myQuads, myAppear );
```

Transforming texture coordinates

- The "texture cookie cutter" can be transformed to translate, rotate, and scale it before cutting out a piece of texture

- Scaling is the most important
  - Scale up and coordinates *wrap* around image boundaries
  - Similar to imagining an infinite amount of texture cookie dough



TextureAttributes class hierarchy

- **`TextureAttributes`** control how a texture is mapped, including use of a texture coordinates transform

Class Hierarchy
java.lang.Object └─ **javax.media.j3d.SceneGraphObject**   └─ **javax.media.j3d.NodeComponent**   └─ **javax.media.j3d.TextureAttributes**

TextureAttributes class methods

- Methods on **`TextureAttributes`** set a **`Transform3D`** to transform texture coordinates

Method
`TextureAttributes( )`
`void setTextureTransform( Transform3D trans )`

Texture rotation example code

- Create **`TextureAttributes`**

```
TextureAttributes myTA = new TextureAttributes( );
```

- Create a rotation transform (Z sticks out of the ST plane)

```
Transform3D myTrans = new Transform3D( );
myTrans.rotZ( Math.PI/4.0 ); // 45 degrees
myTA.setTextureTransform( myTrans );
```

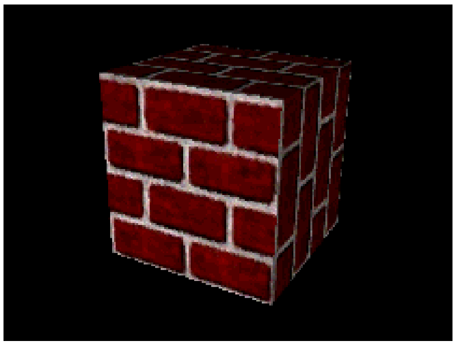
- Set the texture attributes on an **`Appearance`**

```
Appearance myAppear = new Appearance( );
myAppear.setTextureAttributes( myTA );
```

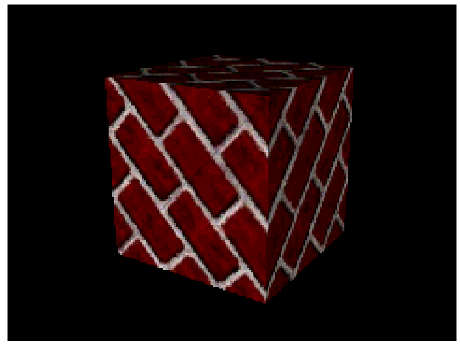
- Assemble the shape

```
Shape3D myShape = new Shape3D( myText, myAppear );
```

Texture rotation example



No rotation



Rotate 45 degrees

Texture scaling example code

- Create **`TextureAttributes`**

```
TextureAttributes myTA = new TextureAttributes( );
```

- Create a scaling transform

```
Transform3D myTrans = new Transform3D( );
myTrans.set( 4.0 );
myTA.setTextureTransform( myTrans );
```

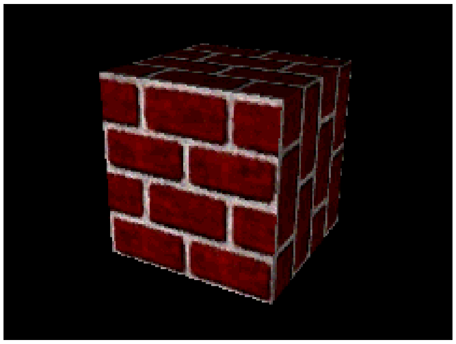
- Set the texture attributes on an **`Appearance`**

```
Appearance myAppear = new Appearance( );
myAppear.setTextureAttributes( myTA );
```

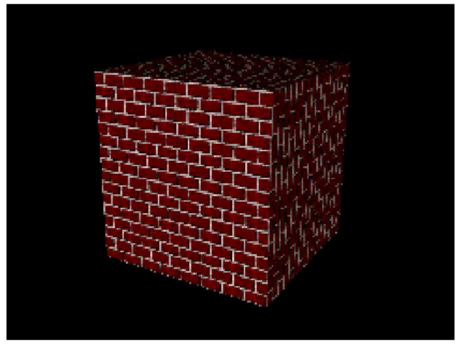
- Assemble the shape

```
Shape3D myShape = new Shape3D( myText, myAppear );
```

Texture scaling example



Scale by 1.0



Scale by 4.0

Texture translation example code

- Create **TextureAttributes**

```
TextureAttributes myTA = new TextureAttributes( );
```

- Create a translation transform

```
Transform3D myTrans = new Transform3D( );
myTrans.set( new Vector3f( 0.25f, 0.0f, 0.0f ) );
myTA.setTextureTransform( myTrans );
```

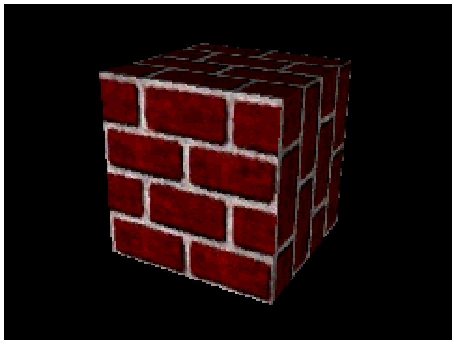
- Set the texture attributes on an **Appearance**

```
Appearance myAppear = new Appearance( );
myAppear.setTextureAttributes( myTA );
```

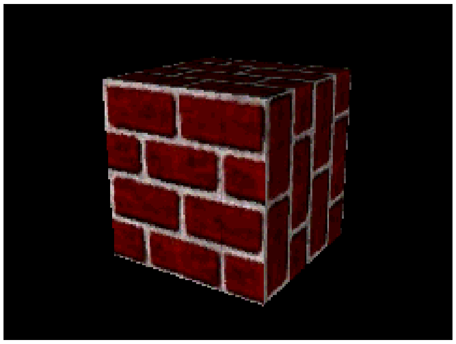
- Assemble the shape

```
Shape3D myShape = new Shape3D( myText, myAppear );
```

Texture translation example



No translation



Translate by 0.25 in S, 0.0 in T

Using texture boundary modes

- *But* . . . when texture coordinates extend past the edge of the image they can:
  - *Wrap* to create a repeating pattern (as before)
  - Or *Clamp* to prevent repeatition

Texture class methods

- Methods on **`Texture`** select **`WRAP`** or **`CLAMP`** boundary modes in S and T
  - **`WRAP`** is the default in both S and T

Method
`void setBoundaryModeS( int mode )`
`void setBoundaryModeT( int mode )`

Texture boundary mode example code

- Load a texture image

```
TextureLoader myLoader = new TextureLoader( "brick.jp
ImageComponent2D myImage = myLoader.getImage( );
```

- Create a **`Texture2D`** using the image, and turn it on

```
Texture2D myTex = new Texture2D( );
myTex.setImage( 0, myImage );
myTex.setEnable( true );
```

- Set the boundary modes and color

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

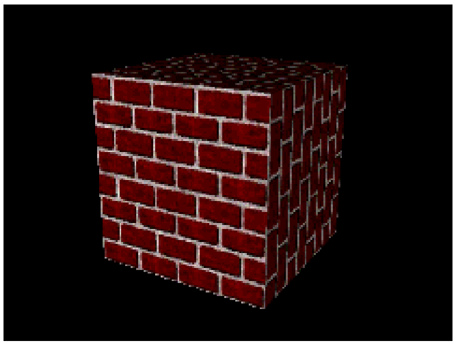
- **`Create an Appearance and set the texture in it`**

```
Appearance myAppear = new Appearance( );
myAppear.setTexture( myTex );
```

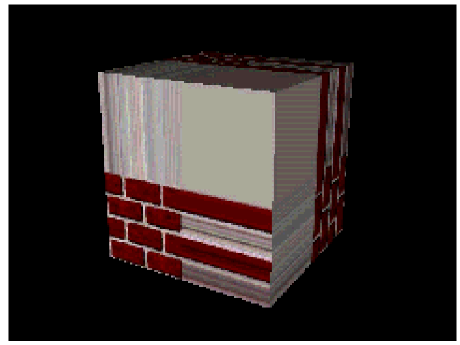
- **`Assemble the shape`**

```
Shape3D myShape = new Shape3D( myText, myAppear );
```

Texture boundary mode example



Wrap



Clamp

Summary

- Textures are in a logical coordinate system with S (horizontal) and T (vertical) directions

- Regardless of true size, all textures have logical width and height of 1.0

- *Texture coordinates* describe the shape of a texture cookie cutter
  - Provide texture coordinates to **`GeometryArray`**
  - Provide texture coordinate indices to **`IndexedGeometryArray`**

Summary

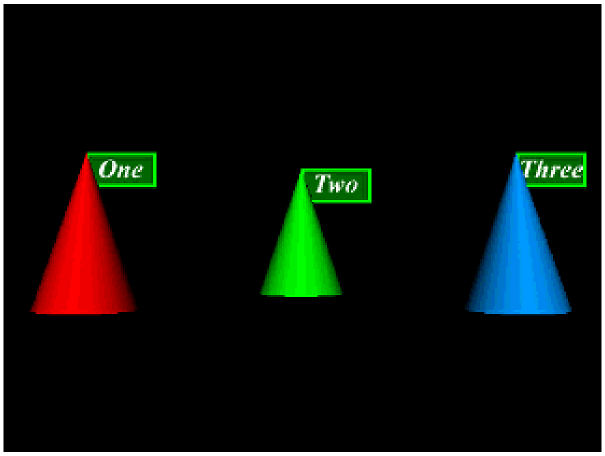
- A *Texture transform* translates, rotates, and scales texture coordinates
- When texture coordinates extend past the image boundary they can *wrap* or be *clamped*
  - When clamped, the rest of the texture cookie is set to a *boundary color*
- Boundary modes are set in **`Texture`**
- Texture transforms are set in **`TextureAttributes`**

Using raster geometry

Motivation

- We would like to position a 2D image in the 3D scene
  - Anchor it to a 3D point in model coordinates
  - Make its size independent of the distance from the user to the shape
- Useful for annotation text, sprites, etc.
- We call this *raster geometry*

Example



[**ExRaster**]

Raster class hierarchy

- **`Raster`** extends **`Geometry`**

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

Building 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

Raster class methods

- Methods on **`Raster`** set the image data and type

Method
`Raster( )`
`void setImage( ImageComponent2D image )`
`void setDepthComponent( DepthComponent depth )`
`void setType( int flag )`

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

Raster class methods

- Methods on **`Raster`** also set the anchor position and image region to use

Method
`void setPosition( Point3f pos )`
`void setSize( int width, int height )`
`void setOffset( int x, int y )`
`void readRaster( Raster raster )`

- Reading from a **`Raster`** only may be done in immediate mode

Raster example code

- Load a texture image

```
TextureLoader myLoader = new TextureLoader( "brick.jpg
ImageComponent2D myImage = myLoader.getImage( );
```

- Create a **`Raster`**

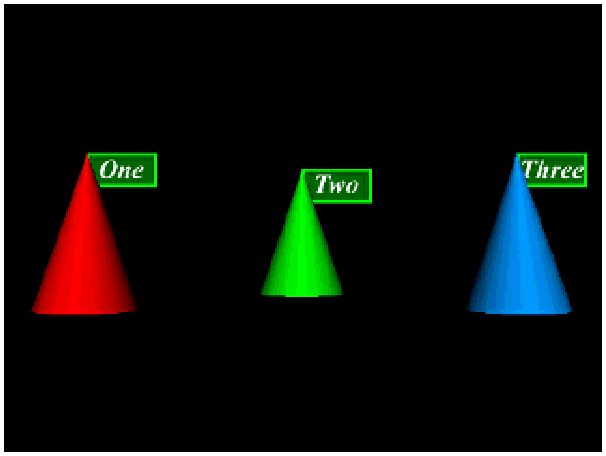
```
Raster myRaster = new Raster( );
myRaster.setPosition( new Point3f( 1.0f, 0.0f, 0.0f )
myRaster.setType( Raster.RASTER_COLOR );
myRaster.setImage( myImage );
myRaster.setOffset( 0, 0 );
myRaster.setSize( 256, 256 );
```

- Assemble the shape

```
Shape3D myShape = new Shape3D( myRaster, myAppear );
```

Using raster geometry

Raster Example



[**ExRaster**]

Summary

- **`Raster`** creates an image sprite by placing a 2D image at a screen position controlled by a 3D anchor position

Lighting the environment

Motivation

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

- Java 3D provides four types of lights to illuminate your scene:
  - Ambient
  - Directional
  - Point
  - Spot

Example

[**ExHenge**]

Light class hierarchy

- All lights share attributes inherited from **`Light`**

Class Hierarchy

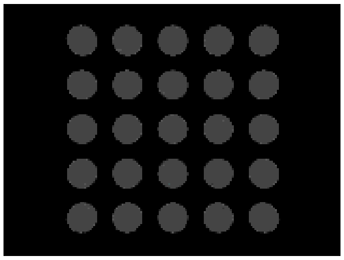
```
java.lang.Object
 └─ javax.media.j3d.SceneGraphObject
     └─ javax.media.j3d.Node
         └─ javax.media.j3d.Leaf
             └─ javax.media.j3d.Light
                 ├─ javax.media.j3d.AmbientLight
                 ├─ javax.media.j3d.DirectionalLight
                 └─ javax.media.j3d.PointLight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

Light class methods

- Methods on **`Light`** control attributes common to all light types:
  - An on/off enable state
  - A color
  - A bounding volume and scope controlling the range of shapes they illuminate

Method
`void setEnable( boolean OnOff )`
`void setColor( Color3f color )`

Creating ambient lights



[**`ExAmbientLight`**]

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

Method
`AmbientLight( )`

AmbientLight example code

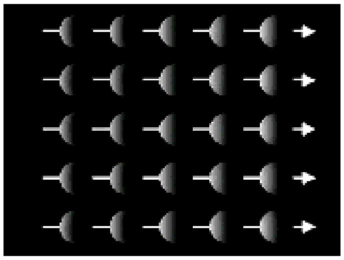
- Create a light

```
AmbientLight myLight = new AmbientLight( );
myLight.setEnable( true );
myLight.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
```

- Set its influencing bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myLight.setInfluencingBounds( myBounds );
```

Creating directional lights



[**ExDirectionalLight**]

- **DirectionalLight** extends **Light**
- Light rays are parallel and aim in one direction

Method
DirectionalLight( )
void setDirection(Vector3f dir)

- The default aim direction is (0.0, 0.0, -1.0)

DirectionalLight example code

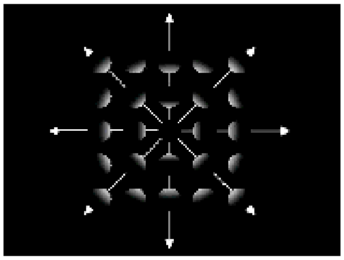
- Create a light

```
DirectionalLight myLight = new DirectionalLight( );
myLight.setEnable( true );
myLight.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
myLight.setDirection( new Vector3f( 1.0f, 0.0f, 0.0f
```

- Set its influencing bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myLight.setInfluencingBounds( myBounds );
```

Creating point lights



[**`ExPointLight`**]

- **`PointLight`** extends **`Light`**
- Light rays emit radially from a point in all directions

Method
`PointLight( )`
`void setPosition( Point3f pos )`

Using point light attenuation

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

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

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

Method
`void setAttenuation( Point3f atten )`

PointLight example code

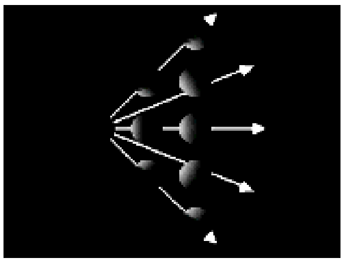
- Create a light

```
PointLight myLight = new PointLight( );
myLight.setEnable( true );
myLight.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
myLight.setPosition( new Point3f( 0.0f, 1.0f, 0.0f )
myLight.setAttenuation( new Point3f( 1.0f, 0.0f, 0.0f
```

- Set its influencing bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myLight.setInfluencingBounds( myBounds );
```

Creating spot lights



[**ExSpotLight**]

- **SpotLight** extends **PointLight**
- 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

Method
SpotLight( )
void setDirection(Vector3f dir)

- The default aim direction is (0.0, 0.0, -1.0)

SpotLight class methods

- Methods on **`SpotLight`** also set the cone spread angle and concentration

Method
`void setSpreadAngle( float angle )`
`void setConcentration( float concen )`

- Spread angle varies from 0.0 to PI/2.0 radians
  - A value of PI radians makes the light a **`PointLight`**
  - The default is PI

- Concentrations vary from 0.0 (unfocused) to 128.0 (focused)
  - The default is 0.0

SpotLight example code

- Create a light

```
SpotLight myLight = new SpotLight( );
myLight.setEnable( true );
myLight.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
myLight.setPosition( new Point3f( 0.0f, 1.0f, 0.0f )
myLight.setAttenuation( new Point3f( 1.0f, 0.0f, 0.0f
myLight.setDirection( new Vector3f( 1.0f, 0.0f, 0.0f
myLight.setSpreadAngle( 0.785f ); // 45 degrees
myLight.setConcentration( 3.0f ); // Unfocused
```

- Set its influencing bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myLight.setInfluencingBounds( myBounds );
```

Using 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 class methods

- Methods on **`Light`** set the influencing bounds

Method
`void setInfluencingBounds( Bounds bounds )`
`void setInfluencingBoundingLeaf( BoundingLeaf leaf )`

Influencing bounds example code

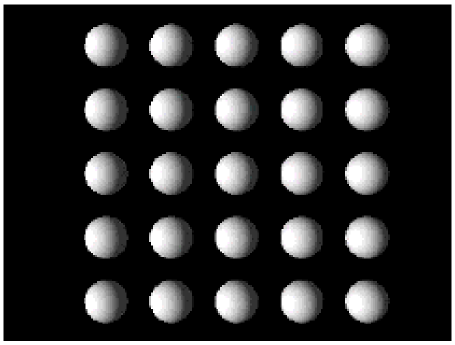
- Set bounds relative to the light's coordinate system

```
PointLight myLight = new PointLight( );
myLight.setInfluencingBounds( myBounds );
```

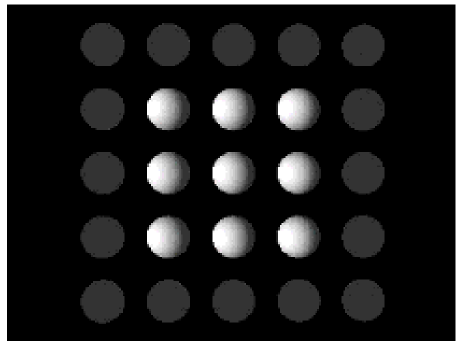
- Or relative to a bounding leaf's coordinate system

```
TransformGroup myGroup = new TransformGroup( );
BoundingLeaf myLeaf = new BoundingLeaf( myBounds );
myGroup.addChild( myLeaf );
. . .
PointLight myLight = new PointLight( );
myLight.setInfluencingBoundingLeaf( myLeaf );
```

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 class methods

- Methods on **`Light`** control the scope list

Method
`void setScope( Group group, int index )`
`void addScope( Group group )`
`void insertScope( Group group, int index )`
`void removeScope( int index )`

Scoping example code

- Build a group of shapes

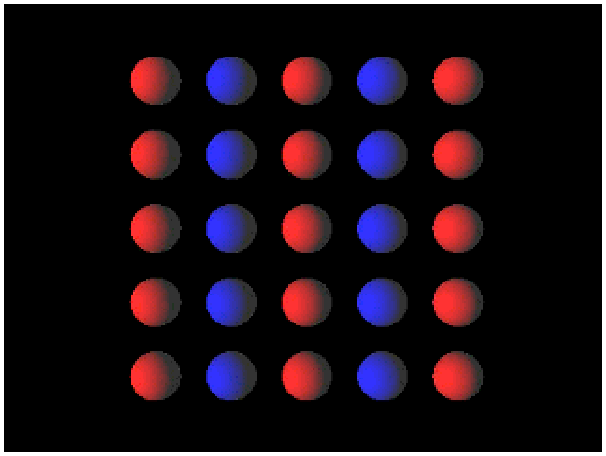
```
TransformGroup myLightable = new TransformGroup( );
Shape3D myShape = new Shape3D( myGeom, myAppear );
myLightable.addChild( myShape );
```

- Create a light and add the group to its scope list

```
DirectionalLight myLight = new DirectionalLight( );
myLight.addScope( myLightable );
```

Lighting the environment

Scoping Example



[**ExLightScope**]

Summary

- Java 3D provides four types of lights:
  - **`AmbientLight`**
  - **`DirectionalLight`**
  - **`PointLight`**
  - **`SpotLight`**

- All lights have a color, can be turned on/off, and have influencing bounds and a scope list

- Directional lights have an aim direction

- Point lights have a position and attenuation

- Spot lights have an aim direction, position, attenuation, and a cone spread angle and 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)

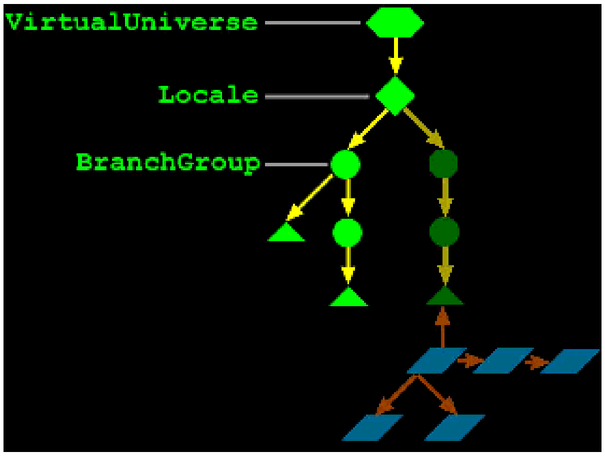
Building a virtual universe

Motivation

- We need to assemble large chunks of content
  - Build components separately
  - Assemble them into a *virtual universe*
- We need scene graph *superstructure*

Looking at the content branch

- The virtual universe superstructure includes the upper portion of the scene graph



Terminology

- Recall some terminology we introduced at the start of this tutorial

- *Virtual universe*: a collection of scene graphs
  - Typically one universe per application

- *Locale*: a position in the universe at which to put scene graphs
  - Typically one locale per universe

- *Branch graph*: a scene graph
  - Typically several branch graphs per locale
  - Content and view branchs are both branch graphs

Scene graph superstructure class hierarchy

- Universes and locales are built using superstructure classes

Class Hierarchy
java.lang.Object
├─ **javax.media.j3d.VirtualUniverse**
├─ **javax.media.j3d.Locale**
└─ **javax.media.j3d.Node**
└─ **javax.media.j3d.Group**
└─ **javax.media.j3d.BranchGroup**

VirtualUniverse class methods

- Methods on **`VirtualUniverse`** access its list of **`Locale`**s

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

Locale class methods

- Methods on **`Locale`** position it within a **`VirtualUniverse`**

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

Locale class methods

- `Locale` methods also manage a list of branch graphs

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

Building a universe example code

- Build a universe

```
VirtualUniverse myUniverse = new VirtualUniverse( );
```

- Build a locale

```
Locale myLocale = new Locale( myUniverse );
```

- Build a branch group

```
BranchGroup myBranch = new BranchGroup( );
```

Building a universe example code

- Build nodes and groups of nodes

```
Shape3D myShape = new Shape3D( myGeom, myAppear );
Group myGroup = new Group( );
myGroup.addChild( myShape );
```

- Add them to the branch group

```
myBranch.addChild( myGroup );
```

- Add the branch graph to the locale

```
myLocale.addBranchGraph( myBranch );
```

Summary

- A **`VirtualUniverse`** holds everything within one or more **`Locale`**s
- A **`Locale`** positions in a universe one or more **`BranchGroup`**s
- A **`BranchGroup`** holds a scene graph, often with separate branchs for content and viewing information

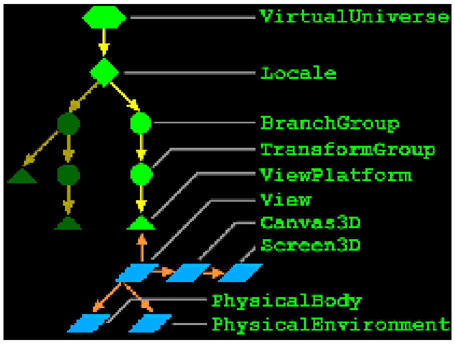
Introducing the view model

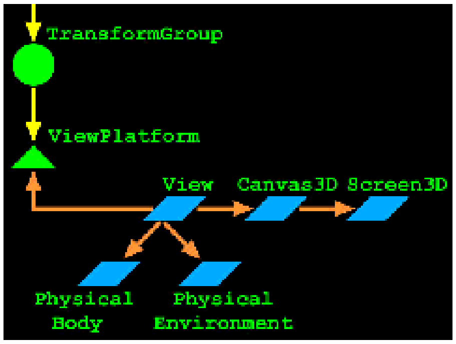
Motivation

- We need control over the user's virtual position and orientation
  - Navigate their viewpoint using the mouse, or any other input device
  - Or move the viewpoint automatically in a guided tour
  - We call such a user viewpoint a *view platform*
- We also need a careful abstraction from hardware gadgetry
  - Support different display configurations
    - Stereo, HMDs, multi-screen portals
  - Support head tracking

Looking at the view branch

- Viewing controls are typically placed in a parallel *view branch* of the scene graph





Coexisting in the physical and virtual worlds

- Shapes, branch groups, locales, and the virtual universe define the *virtual world*
- A user *co-exists* in this virtual world and in the physical world
  - The user has a position and orientation in the *virtual world*
  - The user, and their display, have positions and orientations in the *physical world*
- The Java 3D view model handles mapping between virtual and physical worlds

Understanding constraints and policies

- A chain of relationships control this mapping between worlds
  - Eye locations relative to the user’s head
  - Head location relative to a head tracker
  - Head tracker relative to the tracker base
  - Tracker base relative to an image plate (display)
  - . . . *and so on, with variations*

- A *constraint system* defines these relationships
  - For a given environment and usage, some relationships are constants, while others vary

- Java 3D *policies* select among standard constraint systems and control how they adapt to changes

Understanding view policies

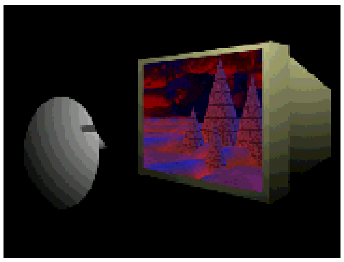
- The *view policy* selects one of two constraint systems

  - *Room-mounted displays*
    - Displays whose locations are fixed
    - CRTs, video projectors, multi-screen walls, portals

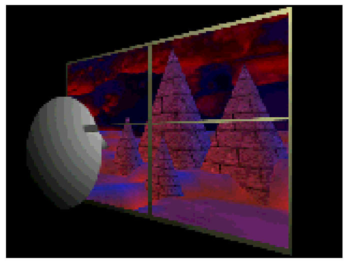
  - *Head-mounted displays*
    - Displays whose locations change as the user moves
    - HMDs

Understanding room-mounted displays

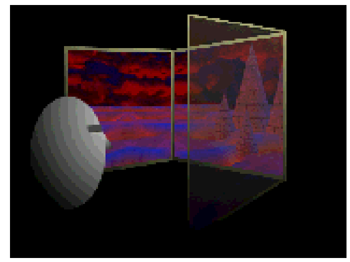
- In a *room-mounted display*, the user looks at a display with a fixed location relative to the physical world



Desktop CRT



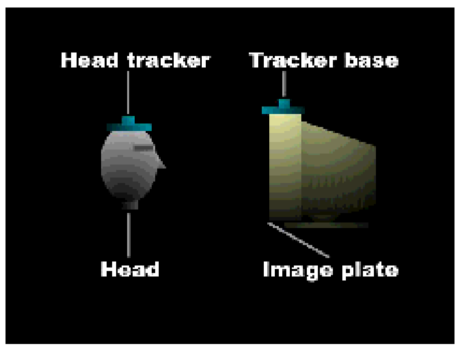
Video wall



Portal

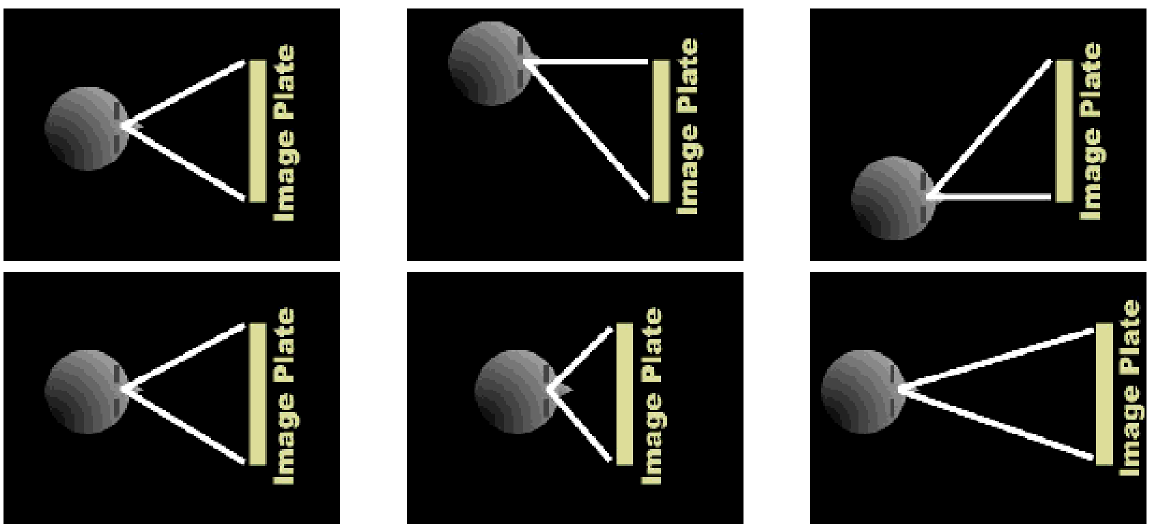
Understanding room-mounted displays

- Physical world components include:
  - *Head* - the user!
  - *Eye* - a "center eye" on the user's head
  - *Image plate* - the physical display
  - *Head tracker* - the tracked point on a user's head
  - *Tracker base* - the tracking system's emitter or reference point



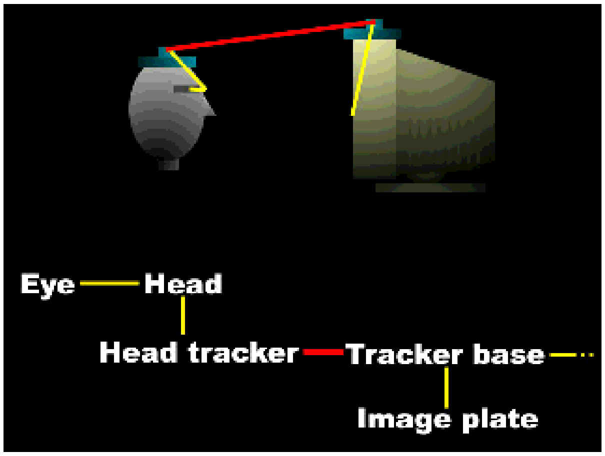
Understanding room-mounted displays

- The constraint system uses the eye location relative to the image plate to compute a correct view frustum
  - When using head tracking, the eyepoint is computed automatically
  - When not using head tracking, the eyepoint may be set manually

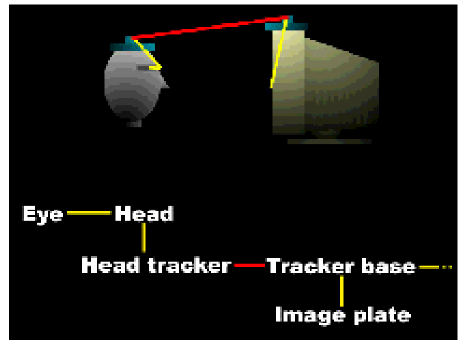


Understanding room-mounted displays

- To map from eye to image plate, the constraint system uses a chain of coordinate system mappings



Understanding room-mounted displays

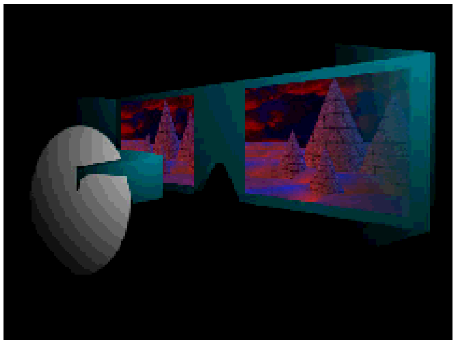


- Configuration constants: (yellow)
  - Physical body
    - Eye-to-head
    - Head-to-head tracker
  - Screen
    - Tracker base-to-image plate

- Vary during use: (red)
  - Head tracker-to-tracker base

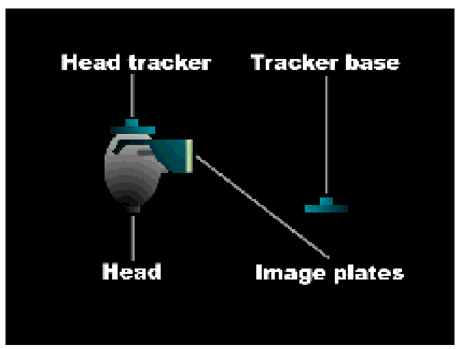
Understanding head-mounted displays

- In a *head-mounted display*, each eye looks at its own display with a fixed location relative to the user's head



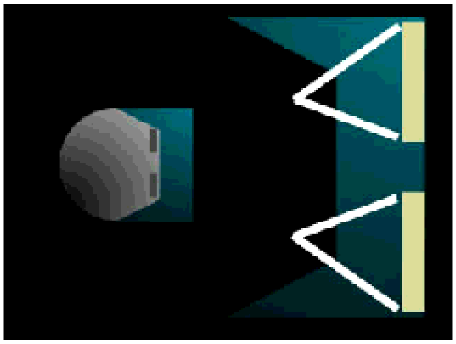
Understanding head-mounted displays

- Physical world components include:
  - *Head* - the user!
  - *Eyes* - left and right eyes on the user’s head
  - *Image plates* - a physical display per eye
  - *Head tracker* - the tracked point on a user’s head
  - *Tracker base* - the tracking system’s emitter or reference point



Understanding head-mounted displays

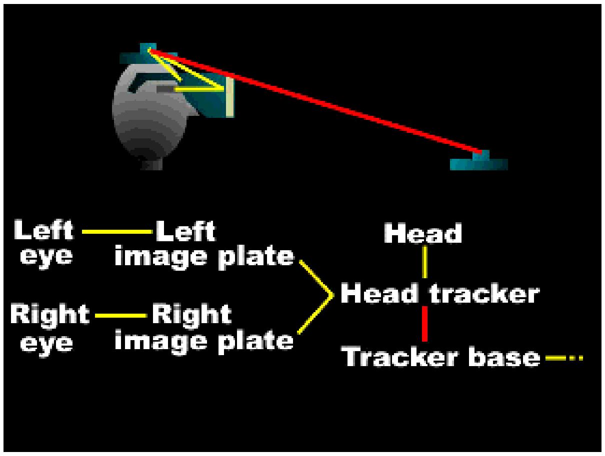
- The constraint system uses the left and right eye locations relative to the left and right image plates to compute correct view frustums



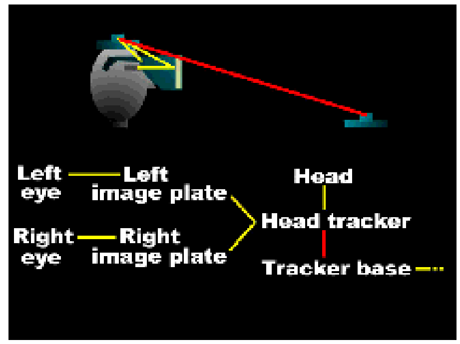
Introducing the view model

Understanding head-mounted displays

- To map from left and right eyes to their image plates, the constraint system uses a chain of coordinate system mappings



Understanding head-mounted displays



- Configuration constants: (yellow)
  - Physical body
    - Left/Right eye-to-head mapping
    - Head-to-head tracker
  - Screen
    - Head tracker-to-left/right image plate

- Vary during use: (red)
  - Head tracker-to-tracker base

Understanding physical to virtual mappings

- Recall that the user *co-exists* in the virtual and physical worlds
  - The user has a physical position and orientation
  - The user also has a virtual position and orientation

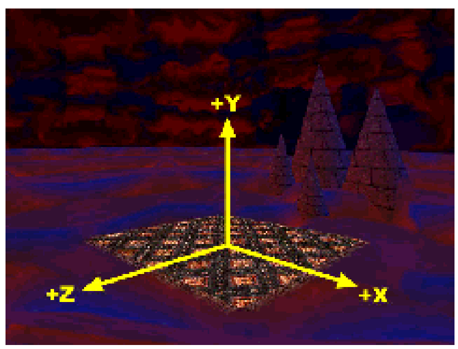
- Room- and head-mounted display view policies handle mapping from the user's physical body to a tracker base and image plates

- To map from this physical world to the virtual world, we add to the constraint chain:
  - Tracker base to coexistance
  - Coexistance to view platform
  - View platform to locale
  - Locale to virtual universe

Understanding physical to virtual mappings

- For example, in a virtual world imagine the view platform is a magic carpet
  - The user can walk about on the carpet
  - The carpet flys about under application control

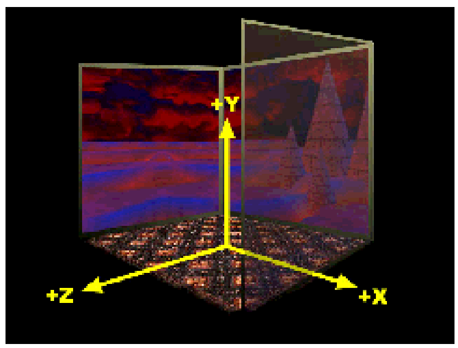
  - Define the view platform origin at "ground level", at carpet center



Understanding physical to virtual mappings

- In the physical world, imagine the user is standing in a portal
  - Images of the virtual world are rendered on three sides
  - The user's position is tracked within the portal

  - Define the portal origin at ground level, at the portal center



Understanding physical to virtual mappings

- Physical device configurations and a room-mounted view policy establish:
  - Mappings from eye to head, to head tracker, to tracker base, to image plate (portal screen)

  - A *tracker base to coexistence transform* maps from the tracker base to the portal center
    - Or whatever reference point you prefer

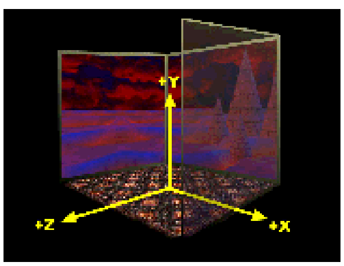
- As the user moves about, their location is computable relative to this coexistence frame of reference - the portal center

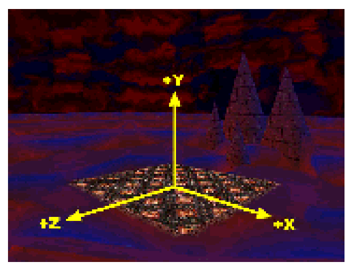
Understanding physical to virtual mappings

- On the virtual side, the scene graph establishes:
  - Mappings from view platform center, to locale, to virtual universe

  - The view platform's center *co-exists* with the center of the portal (or wherever the coexistence transform selects)

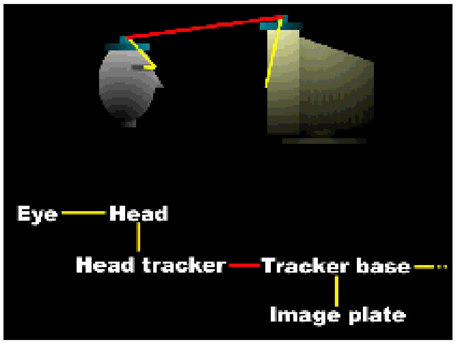
- Together, these physical and virtual mappings establish coexistence
  - Movement in the physical world gives proper corresponding movement in the virtual world

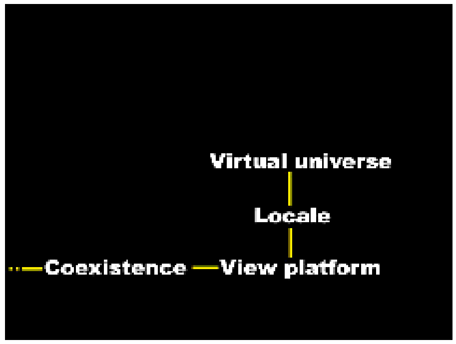




Putting it all together

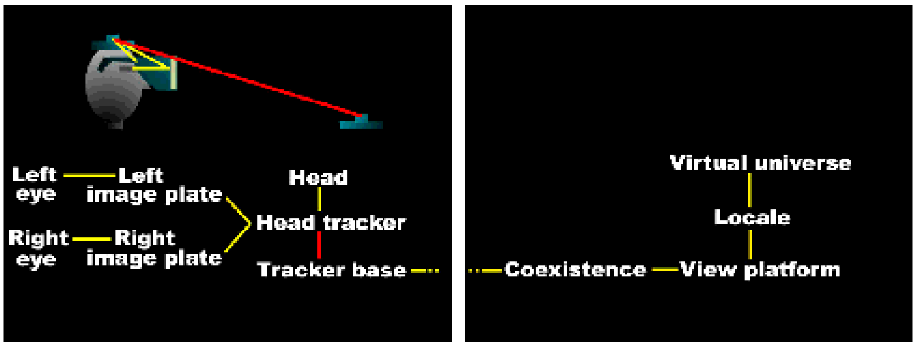
- The room-mounted display view policy:





Putting it all together

- The head-mounted display view policy:

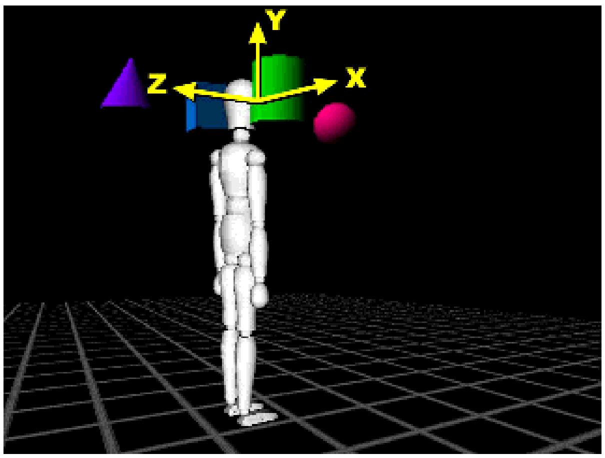


Using view attach policies

- The *view attach policy* establishes how the view platform origin is placed relative to the user (i.e., how it is *attached* to the user's view)
    - *Nominal head*
    - *Nominal feet*
    - *Nominal screen*

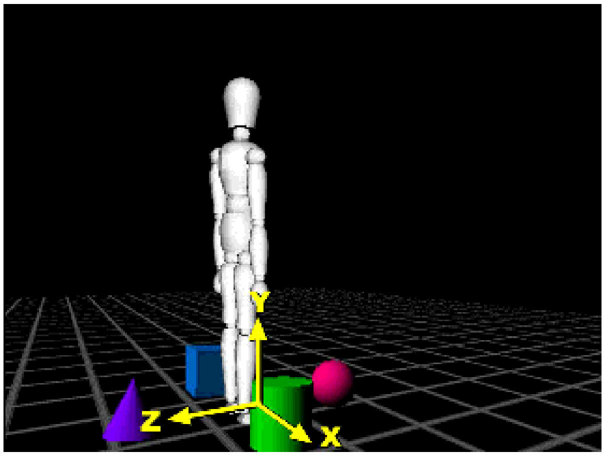
Using the head view attach policy

- *Nominal head* places the view platform origin at the user's head
  - Convenient for arrangement of content around the user's head for a heads-up display
  - Most like "older" view models



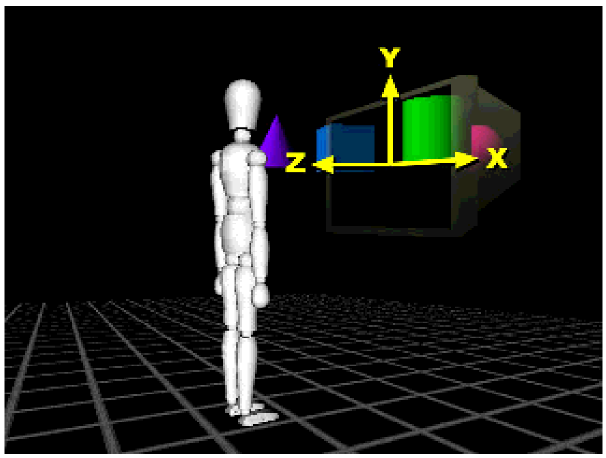
Using the feet view attach policy

- *Nominal feet* places the view platform origin at the user’s feet, at the ground plane
  - Convenient for walk-throughs where the user’s feet should touch the virtual ground



Using the screen view attach policy

- *Nominal screen* places the view platform origin at the screen center
  - Enables the user to view objects from an optimal viewpoint



Using the Java 3D viewing model

- So, the *view model* is composed of:
  - A *view policy* to choose a room- or head-mounted constraint system

  - A set of physical body, physical environment, and screen configuration parameters

  - A set of policies to guide the chosen constraint system
    - Including the view attach policy

Using the Java 3D viewing model

- The physical world policies and parameters are set up when the system is installed and initially configured
  - Application programmers rarely need to deal with these

- The virtual world policies and parameters are set up when the application initializes

- The constraint system then maintains proper coexistence relationships automatically as the user moves

Looking at view model classes

- Let's look at which classes are involved in the view model
- A **`VirtualUniverse`** defines the universe coordinate system
- A **`Locale`** places a scene graph branch within that universe
- A **`ViewPlatform`** (and a **`Transform3D`** above it) defines a view point within that locale
  - It defines a frame of reference for the user's position and orientation in the virtual world
  - Think of it as a magic carpet
  - There can be many **`ViewPlatform`**s in a scene graph

Looking at view model classes

- A **`View`** is the virtual user standing on a **`ViewPlatform`**
  - There can be many **`View`**s on the same **`ViewPlatform`**

- A **`PhysicalBody`** describes the user's dimensions for use by a **`View`**
  - There is always one **`PhysicalBody`** for a **`View`**

- A **`PhysicalEnvironment`** describes the user's environment for use by a **`View`**
  - There is always one **`PhysicalEnvironment`** for a **`View`**

Looking at view model classes

- A **`Canvas3D`** selects a screen area on which to draw a **`View`**
  - Every **`View`** has one or more **`Canvas3D`**s

- A **`Screen3D`** describes the physical display device (image plate) drawn onto by a **`Canvas3D`**
  - A **`Canvas3D`** always has a **`Screen3D`** to draw onto

Looking at what is where

- And now, the view model policies and parameters are found in these classes
- The virtual user's location and orientation is controlled by a **`ViewPlatform`**:
  - A **`Transform3D`** above the **`ViewPlatform`** moves the platform about
  - The *view attach policy* aligns the platform origin with the user's screen, head, or feet

Looking at what is where

- Viewing policies and parameters are controlled by a **`View`**
  - The *projection policy* selects perspective or parallel projection
  - The *view policy* selects the room- or head-mounted display constraint systems
  - Various *window policies* control how the view frustum adapts to viewing parameter changes

Looking at what is where

- The user’s physical dimensions are described by a **`PhysicalBody`**
    - Parameters set the left and right eye and ear positions
    - Parameters also set the nominal head height from the ground, and the nominal eye offset from the nominal screen
    - A transform describes the head to head tracker relationship

Looking at what is where

- The user's display, input sensors, and sound environment are described by a **`PhysicalEnvironment`**
  - A transform describes the coexistence to tracker base relationship
  - A set of abstract input sensors provide access to trackers
  - An audio device enables sound playback

Looking at what is where

- The drawing area is selected by a **`Canvas3D`**
- The physical screen device is described by a **`Screen3D`** (image plate)
    - A transform describes the tracker base to image plate relationship
    - Parameters set the display's physical width and height (in meters)

Summary

- Virtual world:
  - **`ViewPlatform`** controls the user’s virtual position and orientation
  - **`View`** sets the view policy, etc.

- Physical world:
  - **`PhysicalBody`** describes the user
  - **`PhysicalEnvironment`** describes the user’s environment
  - **`Canvas3D`** selects a region to draw into
  - **`Screen3D`** describes the screen device

Viewing the scene

Motivation

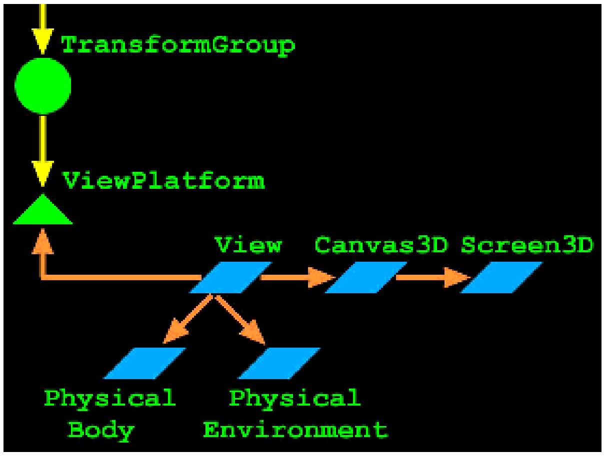
- Now we can look deeper at the view model classes and methods

- Everything has reasonable default values

- For complex display systems, a system manager’s configuration establishes the default values
  - Thereafter, applications need not be aware of the configuration’s details

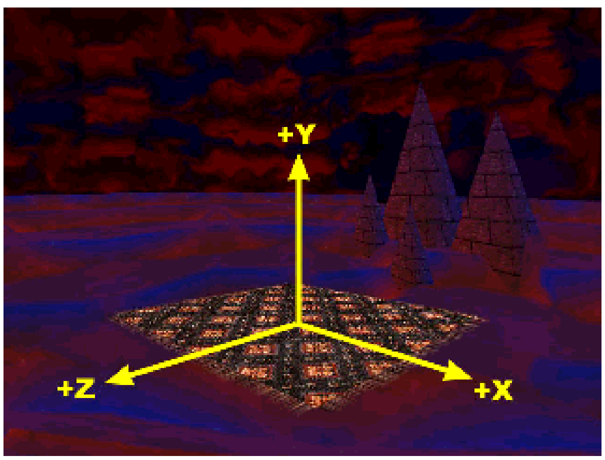
Looking at the view branch

- Let's start with the **`ViewPlatform`**, and work through the viewing objects



Creating a ViewPlatform

- A **`ViewPlatform`** defines a view point within the scene
  - It defines a frame of reference for the user's position and orientation in the virtual world
  - Think of it as a magic carpet on which the user stands/sits
  - There can be many **`ViewPlatform`**s in a scene graph



Using ViewPlatforms

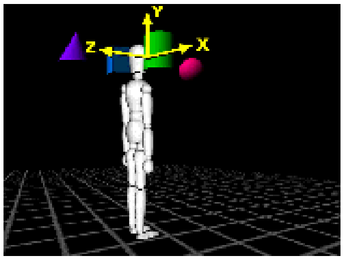
- A **`ViewPlatform`** is a leaf in the scene graph
  - It can be transformed by a **`TransformGroup`** parent
  - User interface and animation features can modify that **`TransformGroup`** to move the platform (fly the magic carpet)

Setting the activation radius

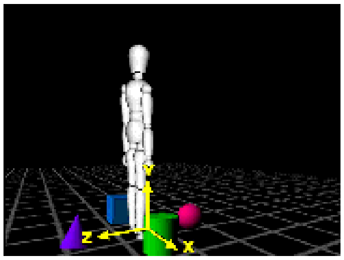
- Each `ViewPlatform` has an *activation radius* that defines a region of interest
  - Animation behaviors, sounds, backgrounds, fog, and other nodes have bounding volumes
  - When the activation radius intersects those bounds, those nodes are active
    - Backgrounds or fog are activated
    - Sounds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

Using view attach policies

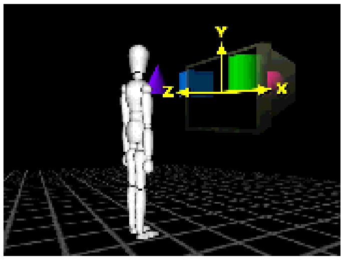
- Each **`ViewPlatform`** has a *view attach policy* that determines how the user's **`View`** is placed relative to the **`ViewPlatform`**'s origin



`NOMINAL_HEAD`
origin at user's head (default)



`NOMINAL_FEET`
origin at user's feet



`NOMINAL_SCREEN`
origin at screen center

ViewPlatform class methods

- Methods on **`ViewPlatform`** set the activation radius and attach policy

Method
`ViewPlatform( )`
`void setActivationRadius( float radius )`
`void setViewAttachPolicy( int policy )`

- Policy values include: **`NOMINAL_SCREEN`**, **`NOMINAL_HEAD`** (default), and **`NOMINAL_FEET`**

ViewPlatform example code

- Create a **`TransformGroup`** to steer the platform

```
TransformGroup viewGroup = new TransformGroup( );
viewGroup.setCapability( TransformGroup.ALLOW_TRANSFORM_W
```

- Add a **`ViewPlatform`**

```
ViewPlatform myPlatform = new ViewPlatform( );
myPlatform.setActivationRadius( 1000.0f );
myPlatform.setViewAttachPolicy( View.NOMINAL_HEAD );
viewGroup.addChild( myPlatform );
```

- Add them to a **`BranchGroup`** view branch

```
BranchGroup viewBranch = new BranchGroup( );
viewBranch.addChild( viewGroup );
myLocale.addBranchGraph( viewBranch );
```

Using views

- A **View** represents the user on a **ViewPlatform**
  - It manages the rendering of the scene into a screen region from the user's viewpoint
  - That screen region is a **Canvas3D** (extends AWT **Canvas**)

- Typically, add a **Canvas3D** to a Java **Frame**, then point a **View** at that canvas

Method
View( )
void attachViewPlatform(ViewPlatform vp)
void setCanvas3D(Canvas3D c3d)

Setting the view projection policy

- Rendering through a **`View`** can use **`PERSPECTIVE_PROJECTION`** (default) or **`PARALLEL_PROJECTION`**

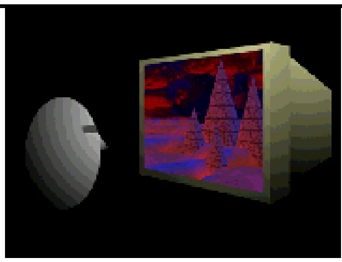
- You can also control front and back clip planes

Method
`void setProjectionPolicy( int policy )`
`void setFrontDistance( double distance )`
`void setBackDistance( double distance )`

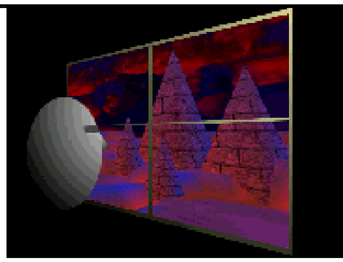
Setting the view policy

- A **View**'s *view policy* selects the constraint system to use for the display configuration
  - **SCREEN_VIEW**: room-mounted displays (default)
  - **HMD_VIEW**: head-mounted displays

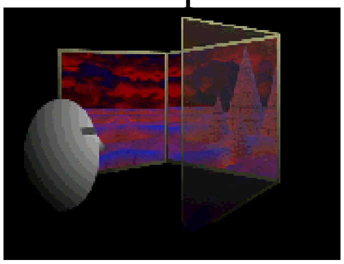
Method
void setViewPolicy(int policy)



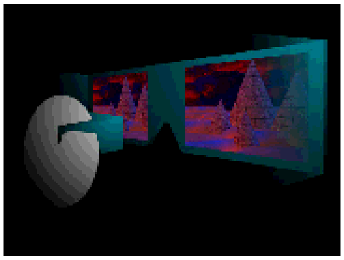
Desktop CRT



Video wall



Portal



HMD

Setting physical data for a view

- **`View`** methods select the physical body and environment to use with the view policy

Method
`void setPhysicalBody( PhysicalBody pb )`
`void setPhysicalEnvironment( PhysicalEnvironment pe )`

Using a Canvas3D

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

- A **`Canvas3D`** describes the region of a **`Screen3D`** in which to draw a **`View`**

- A **`Screen3D`** describes the physical screen device (image plate)

Canvas3D class methods

- Methods on **`Canvas3D`** configure the use of the underlying **`Screen3D`**, including support for stereo

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

Canvas3D class methods

- When not using head tracking, methods on **`Canvas3D`** also manually set the left and right eye locations relative to the image plate

Method
`void setLeftManualEyeInImagePlate( Point3d position )`
`void setRightManualEyeInImagePlate( Point3d position )`

Using a Screen3D

- Methods on **`Screen3D`** describe the physical device and the tracker base to image plate transform

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

Using a Screen3D

- Methods on **`Screen3D`** also set transforms to place the tracker base relative to the single image plate (for room-mounted displays) or to the left and right image plates (for head-mounted displays)

Method
`void setTrackerBaseToImagePlate( Transform3D trans )`
`void setTrackerBaseToLeftImagePlate( Transform3D trans )`
`void setTrackerBaseToRightImagePlate( Transform3D trans )`

Describing the user's physical body

- Methods on **`PhysicalBody`** set the eye and ear positions, and the user's height

Method
`PhysicalBody( )`
`void setLeftEarPosition( Point3d position )`
`void setRightEarPosition( Point3d position )`
`void setLeftEyePosition( Point3d position )`
`void setRightEyePosition( Point3d position )`
`void setNominalEyeHeightFromGround( double height )`

Describing the user's physical body

- Methods on **`PhysicalBody`** also set the head tracker's position relative to the head, and the screen's position relative to the eye

Method
`void setHeadToHeadTracker( Transform3D trans )`
`void setNominalEyeOffsetFromNominalScreen( double offset )`

Describing the physical environment

- Methods on **`PhysicalEnvironment`** set the coexistence to tracker base transform

Method
`PhysicalEnvironment( )`
`void setCoexistenceToTrackerBase( Transform3D trans )`

- The **`PhysicalEnvironment`** also describes the set of available input sensors, discussed in a later section

View example code

- Create a **`Canvas3D`** with a default configuration (automatically creating a **`Screen3D`**)

```
Canvas3D myCanvas = new Canvas3D( null );
```

- Create a **`View`** and give it the **`Canvas3D`**

```
View myView = new View( );
myView.setCanvas3D( myCanvas );
```

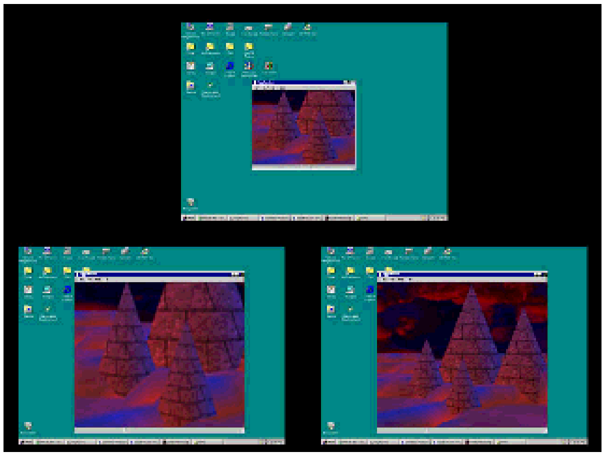
- And attach the **`ViewPlatform`** to the **`View`**

```
myView.attachViewPlatform( myPlatform );
```

- Use defaults for the physical body, physical environment, and miscellaneous transforms

Using view window policies

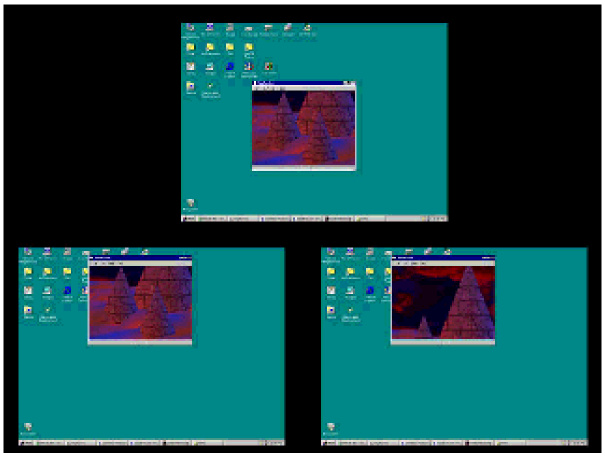
- A **View**’s *resize policy* sets how the view changes on a window resize



PHYSICAL_WORLD	**VIRTUAL_WORLD**
Same view fills window	View changes to see more/less

Using view window policies

- A **View**'s *movement policy* sets how the view changes on a window move

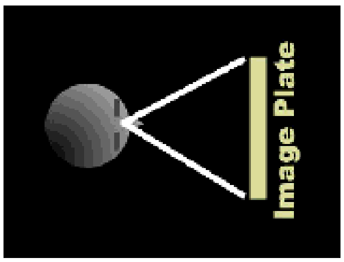


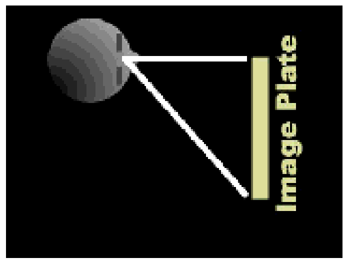
PHYSICAL_WORLD
Same view fills window

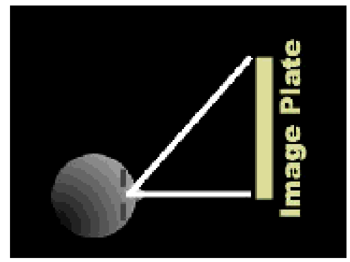
VIRTUAL_WORLD
View shifts to see left/right/above/below

Using view window policies

- When using head tracking, the constraint system automatically changes the view frustum as the users head moves

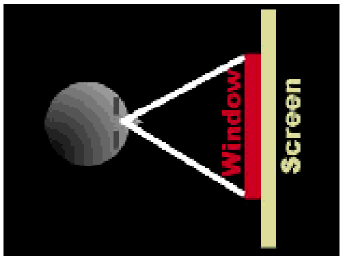


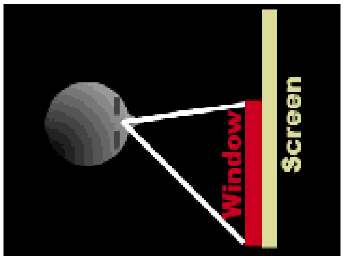




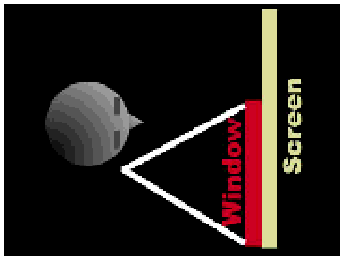
Using view window policies

- When *not* using head tracking, a **`View`**'s *eyepoint policy* sets how the view frustum changes on a window move





`RELATIVE_TO_SCREEN`
Frustum changes



`RELATIVE_TO_WINDOW`
Frustum doesn't change

- **`RELATIVE_TO_FIELD_OF_VIEW`** (default) enables the application to set the field of view directly. The eyepoint changes accordingly.

View class methods

- **`View`** methods set these window policies

Method
`void setWindowEyepointPolicy( int policy )`
`void setWindowMovementPolicy( int policy )`
`void setWindowResizePolicy( int policy )`

View class methods

- When using a **`RELATIVE_TO_FIELD_OF_VIEW`** window eyepoint policy, you can set the **`View`**'s field of view

Method
`void setFieldOfView( double fovx )`

Setting the view screen scale policy

- A **View**'s *screen scale policy* selects how a view's scale factor is chosen:
  - **SCALE_EXPLICIT**: set it using **setScreenScale**
  - **SCALE_SCREEN_SIZE**: derive it from the screen's physical size (default)

Method
void setScreenScalePolicy(int policy)
void setScreenScale(double scale)

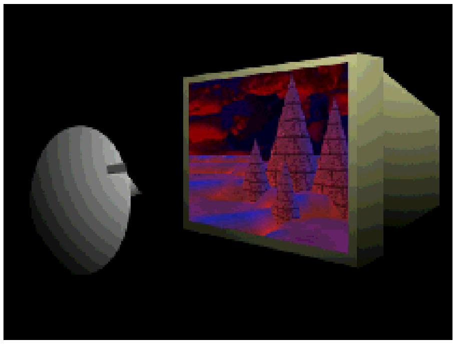
Setting the view monoscopic policy

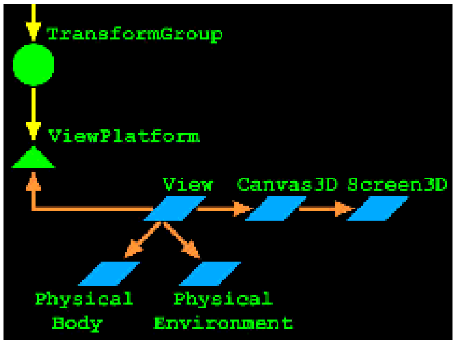
- A **View**'s *monoscopic view policy* selects how a single-image view is created when a **Canvas3D** is not in stereo mode
  - **LEFT_EYE_VIEW**: render from the left eye
  - **RIGHT_EYE_VIEW**: render from the right eye
  - **CYCLOPEAN_EYE_VIEW**: render from a "center" eye midway between left and right eyes (default)

Method
void setMonoscopicViewPolicy(int policy)

Using a desktop configuration

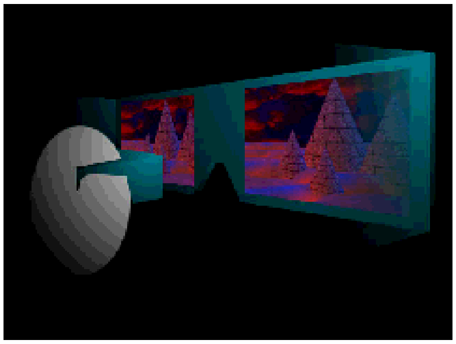
- Use a single **`Canvas3D`** for a single drawing surface in a desktop configuration

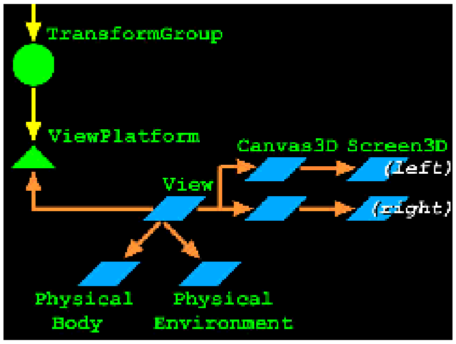




Using an HMD configuration

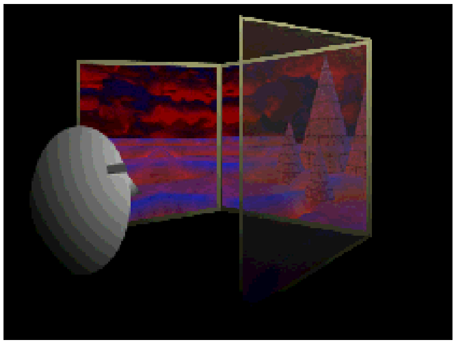
- Use two **`Canvas3D`**s for left and right drawing surfaces in an HMD configuration

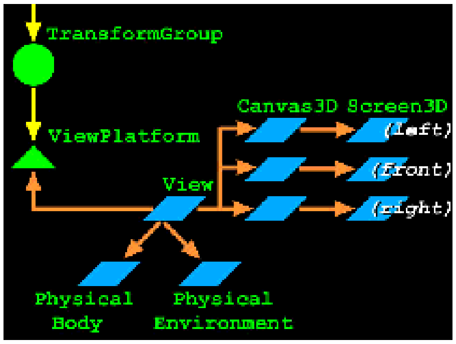




Using a portal configuration

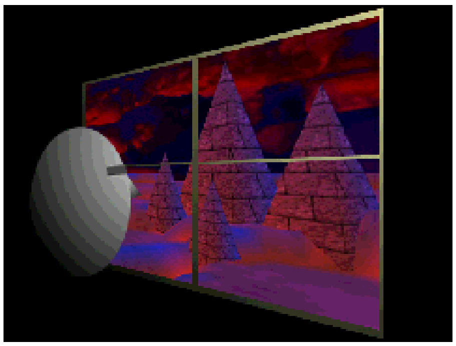
- Use three **`Canvas3D`**s for left, front, and right drawing surfaces in a portal configuration

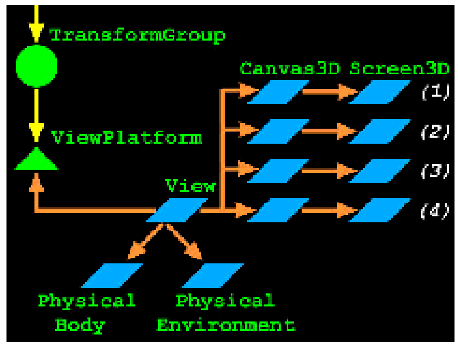




Using a wall configuration

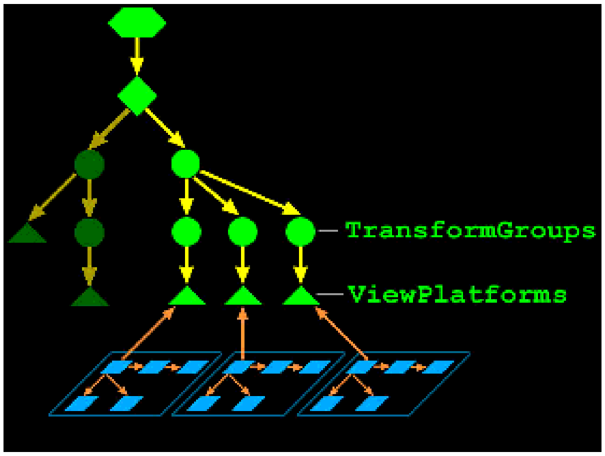
- Use four or more **`Canvas3D`**s for a multi-screen drawing surface in a wall configuration





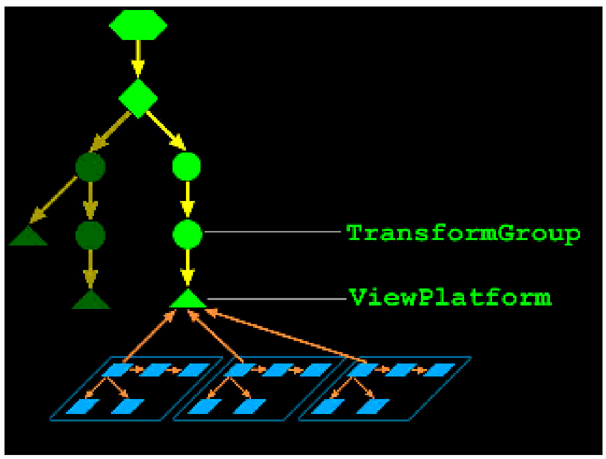
Using multiple view platforms

- A scene graph may contain multiple **`ViewPlatform`**s
  - When a **`View`** is attached to a platform, the scene is rendered from that viewpoint
  - Moving a **`View`** from one platform to another "teleports" the user to a new viewpoint



Using multiple views

- A **ViewPlatform** may have multiple **View**s attached
  - Each **View** renders the same scene from that platform
  - You could track multiple users, each with their own **View** on that platform



Immersive workbench example code

- For an immersive workbench, use a single canvas and screen

```
myView.setCanvas3D( myCanvas );
```

- Use a room-mounted display view policy:

```
myView.setViewPolicy( View.SCREEN_VIEW );
```

- Attach the view to the user’s head:

```
myViewPlatform.setViewAttachPolicy( View.NOMINAL_HEAD );
```

- Use virtual-world window policies and a screen-relative eyepoint:

```
myView.setWindowResizePolicy( View.VIRTUAL_WORLD );
myView.setWindowMovementPolicy( View.VIRTUAL_WORLD );
myView.setWindowEyePointPolicy( RELATIVE_TO_SCREEN );
```

Immersive workbench example code

- Enable head-tracking and place co-existence at the tracker base:

```
myView.setTrackingEnable( true );
myPhysEnv.setCoexistenceToTrackerBase( ident );
```

- Locate the tracker base relative to the workbench:

```
Screen3D myScreen = myCanvas.getScreen3D( );
myScreen.setTrackerBaseToImagePlate( transform );
```

- And configure the screen's size and scale policy:

```
myScreen.setPhysicalScreenHeight( height );
myScreen.setPhysicalScreenWidth( width );
myScreen.setScreenScalePolicy( View.SCALE_EXPLICIT );
```

Summary

- A **`ViewPlatform`** positions a user's **`View`** of the scene
- A **`View`** controls how to render the scene
- A **`Canvas3D`** selects the region of the screen in which a **`View`** should render
- A **`Screen3D`** describes that screen
- A **`PhysicalBody`** describes the user
- A **`PhysicalEnvironment`** describes the user's environment

Building a simple universe

Motivation

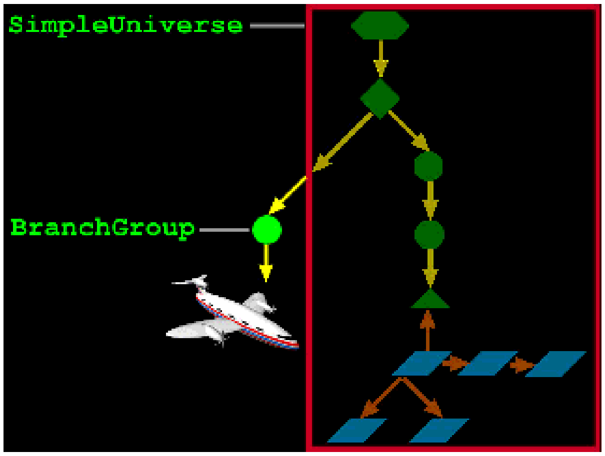
- You can create universes, locales, branchs, view platforms, views, and so forth by yourself

- *Or* you can use the **`SimpleUniverse`** utility to create a standard set for you
  - Far easier and appropriate for most applications

Building a simple universe

Using SimpleUniverse

- A **`SimpleUniverse`** encapsulates a common superstructure



SimpleUniverse class methods

- Methods on **`SimpleUniverse`** build the universe and attach content to it

Method
`SimpleUniverse( Canvas3D canvas )`
`void addBranchGraph( BranchGroup group )`

SimpleUniverse example code

- Create a **`Canvas3D`** with a default configuration (automatically creating a **`Screen3D`**)

```
Canvas3D myCanvas = new Canvas3D( null );
```

- Create a **`SimpleUniverse`** and give it the **`Canvas3D`**

```
SimpleUniverse myUniverse = new SimpleUniverse( myCanvas
```

- And attach your content branch

```
myUniverse.addBranchGraph( myBranch );
```

Summary

- A **SimpleUniverse** handles building standard infrastructure and viewing components
  - **VirtualUniverse**
  - **Locale**
  - **BranchGroup** for viewing objects
  - **TransformGroup** for moving the view platform
  - **ViewPlatform**
  - **View**
  - **PhysicalBody**
  - **PhysicalEnvironment**

Using input devices

Motivation

- There are more input devices besides the mouse:
  - Joysticks
  - Six-degree-of-freedom devices (6DOF) such as a Polhemus, Bird, SpaceBall, Magellan, Ultrasonic tracker, *etc.*
  - Button, knobs, sliders

- Read from any physical input device:
  - Use the serial-device standard extension
  - Use the networking API
  - Use the Java-to-C interface

- Java 3D provides an input device abstraction to:
  - Encapsulate device-specific details behind a generic interface
  - Enable painless integration of new input devices within existing Java applications

Looking at input device components

- An implementation of the **`InputDevice`** interface provides:
  - A description of a continuous device
  - Initialization, prompt for a value, get a value, close, *etc.*
  - Construction of one or more **`Sensor`**s for abstract access to the physical detectors

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

InputDevice interface methods

- Implement the **`InputDevice`** interface for a new input device
  - Supply methods to initialize the device, and get data
  - The principal result is one or more new **`Sensor`**s that abstract the device for generic use elsewhere in Java 3D

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

Using sensors

- **`Sensor`** represents an abstract 6DOF input and any associated buttons/knobs
- The **`Sensor`** abstraction enables a separation between physical and virtual worlds
  - Maps physical position, orientation, and state to an abstract 6DOF value and state
  - Provides generic methods for accessing these values
- Available sensors are managed by the **`PhysicalEnvironment`**
- Sensors are built by low-level **`InputDevice`** implementations

Using sensors

- **`PhysicalEnvironment`** maintains a list of sensors
  - Plugboard model: The application assigns input device **`Sensor`**s to positions in the sensor array
  - Each one is specially identified by an array index
- The application can associate sensor indices with:
  - **`HeadIndex`**
  - **`LeftHandIndex`**
  - **`RightHandIndex`**
  - **`DominantHandIndex`**
  - **`NonDominantHandIndex`**
- Whatever sensor is at the **`HeadIndex`** is used for head tracking, and so forth

Using sensors

- A **`Sensor`** manages 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`**

Using 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 **`Object`**

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

Sensor class methods

- Methods on **`Sensor`** get access to the input device . . .

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

Sensor class methods

- . . . and get the latest 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

- Methods also set a prediction policy 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`** (default) and **`PREDICT_NEXT_FRAME_TIME`**
- Predictors include: **`NO_PREDICTOR`**(default) , **`HEAD_PREDICTOR`**, and **`HAND_PREDICTOR`**

SensorRead class hierarchy

- **`SensorRead`** extends **`Object`** and encapsulates the latest data from an input device

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

SensorRead class methods

- Methods on **`SensorRead`** get the current button state and 3D transform

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

Summary

- To use a new input gadget, implement the **`InputDevice`** interface and supply methods to read that gadget
- Provide high-level generic access to that device through a **`Sensor`**
- A **`SensorRead`** contains a reading from the **`Sensor`**
- Use methods on **`SensorRead`** to get the associated transform and button state

Creating behaviors

Motivation

- A *Behavior* is a Java class that makes changes to a scene graph
- In a broad sense, your entire Java application is a behavior
- Java 3D also provides a **`Behavior`** class as a base class for smaller components that change the scene
    - Often one behavior for each shape being animated

Motivation

- Java 3D behavior support:
  - Supports arbitrary content changes - behaviors are just Java methods
  - Schedules behaviors to run only when necessary
  - Enables composability where independent behaviors may run in parallel without interfering with each other
  - Provides basic dead reckoning for animation execution independent of host speed

Behavior class hierarchy

- **`Behavior`** extends **`Leaf`**

- Your application extends **`Behavior`** further to create one or more behaviors to change scene content

Class Hierarchy
java.lang.Object   └─ **javax.media.j3d.SceneGraphObject**   └─ **javax.media.j3d.Node**   └─ **javax.media.j3d.Leaf**   └─ **javax.media.j3d.Behavior**

Creating behaviors

- Every behavior contains:
  - An **`initialize`** method called when the behavior is made live
  - A **`processStimulus`** method called when the behavior wakes up
  - Wakeup conditions controlling when to wakeup next
    - Respecified on each wakeup
  - Scheduling bounds controlling scheduling
    - When the viewer's activation radius intersects the bounds, the behavior is scheduled

Creating 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:
  - On initialization, set initial wakeup criteria
  - Get the object’s location
  - Create a transform to re-orient the radar dish
  - Set a `TransformGroup` of the radar dish
  - Set the next wakeup criteria
  - Return

Behavior class methods

- Methods on **`Behavior`** include those your subclass provides, and a generic method to enable or disable the behavior

Method
`Behavior( )`
`void initialize( )`
`void processStimulus( Enumeration criteria )`
`void setEnable( boolean onOff )`
`void wakeupOn( WakeupCondition criteria )`

Behavior example code

- Extend the **`Behavior`** class and fill in the **`initialize`** and **`processStimulus`** methods

```
public class MyBehavior extends Behavior {
    private WakeupCriterion criteria;
    public MyBehavior( ) {
        // Do something on construction
        . . .
        criteria = new WakeupOnAWTEvent( . . . );
    }
    public void initialize( ) {
        // Do something at startup
        . . .
        wakeupOn( criteria );
    }
    public void processStimulus( Enumeration criteria ) {
        // Do something on a wakeup
        . . .
        wakeupOn( criteria );
    }
}
```

Creating behavior scheduling bounds

- A behavior only needs to be scheduled if the viewer is nearby
  - The viewer's activation radius intersects its *scheduling bounds*
  - Behavior bounding enables costly behaviors to be skipped if they aren't nearby

- 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

- Methods on **`Behavior`** set the scheduling bounds

Method
`void setSchedulingBounds( Bounds bounds )`
`void setSchedulingBoundingLeaf( BoundingLeaf leaf )`

Scheduling bounds example code

- Set bounds relative to the behavior's coordinate system

```
Behavior myBeh = new MyBehavior( );
myBeh.setSchedulingBounds( myBounds );
```

- Or relative to a bounding leaf's coordinate system

```
TransformGroup myGroup = new TransformGroup( );
BoundingLeaf myLeaf = new BoundingLeaf( bounds );
myGroup.addChild( myLeaf );
. . .
Behavior myBeh = new MyBehavior( );
myBeh.setSchedulingBoundingLeaf( myLeaf );
```

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/ORed to form *wakeup conditions*

WakeupCriterion class hierarchy

- **`WakeupCriterion`** extends **`WakeupCondition`** to provide multiple ways to wakeup a behavior

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

- The **`WakeupCriterion`** base class only provides a method to ask if the wakeup has been triggered

- Each of the subclasses provide constructors and methods for specific wakeup criterion

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

Waking up on an AWT event

- 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 the distance the mouse has moved since the press and map it to a rotation angle
  - Create a rotation transform and write to a **`TransformGroup`**

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

Waking up on elapsed time

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

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

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

Waking up on shape collision

- 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

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

Waking up on viewer proximity

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

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

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

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

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 **`Object`** and provides several subclasses to group wakeup criterion

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

- Methods on the **`WakeupCondition`** base class only ask about the grouped wakeup criterion

- Each of the subclasses provide constructors and methods for specific wakeup groupings

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

WakeupCondition subclass methods

- The **`WakeupCondition`** subclasses have constructions that use arrays of **`WakeupCriterion`** or other **`WakeupCondition`**s

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

WakeupCondition example code

- Create AWT event wakeup criterion

```
WakeupCriterion[] onMouseEvents =
    new WakeupCriterion[2];
onMouseEvents[0] =
    new WakeupOnAWTEvent( MouseEvent.MOUSE_PRESSED );
onMouseEvents[1] =
    new WakeupOnAWTEvent( MouseEvent.MOUSE_RELEASED );
```

- Combine together those criterion

```
WakeupCondition onMouse =
    new WakeupOr( onMouseEvents );
```

WakeupCondition example code

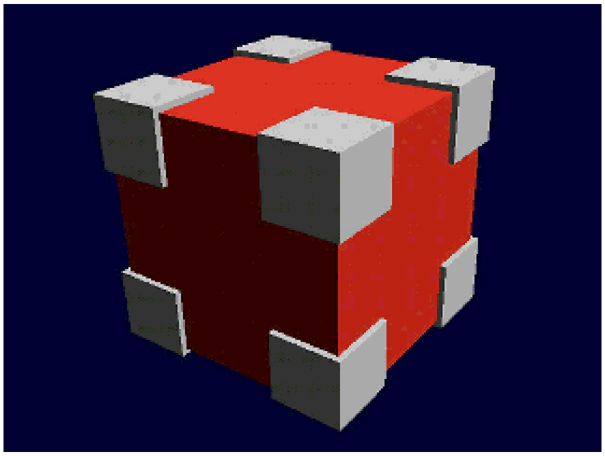
- Create the behavior

```
Behavior myBeh = new MyBehavior( );
```

- And set the behavior’s wakup conditions and scheduling bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myBeh.setSchedulingBounds( myBounds );
```

WakeupCondition example



[**Drag**]

Summary

- A **`Behavior`** is a base class extended to hold:
    - An **`initialize`** method called when made live
    - A **`processStimulus`** method called at wakeup

- A **`WakeupCriterion`** defines a specific condition for behavior wakeup, including elapsed time, AWT events, etc.

- A **`WakeupCondition`** combines together multiple **`WakeupCriterion`**

- Behaviors are schedulable (if enabled) when the viewer's activation radius intersects the behavior's scheduling bounds
    - Default is *no scheduling bounds*, so nothing is scheduled!

Creating interpolator behaviors

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

Using interpolator value mappings

- 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

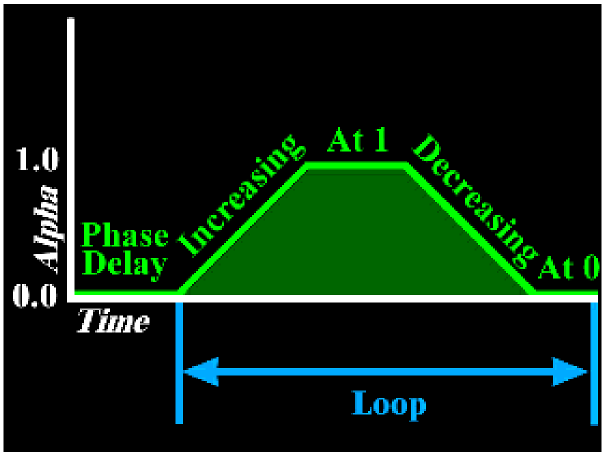
  - Alpha-to-*Value*
    - Different interpolator types map to different values, such as transforms, colors, switches

Mapping time to alpha

- An *Alpha generator* computes alpha using:
  - Trigger time
  - *Phase Delay* before initial alpha change
  - *Increasing* time for increasing alpha
  - *At-One* time for constant high alpha
  - *Decreasing* time for decreasing alpha
  - *At-Zero* time for constant low alpha

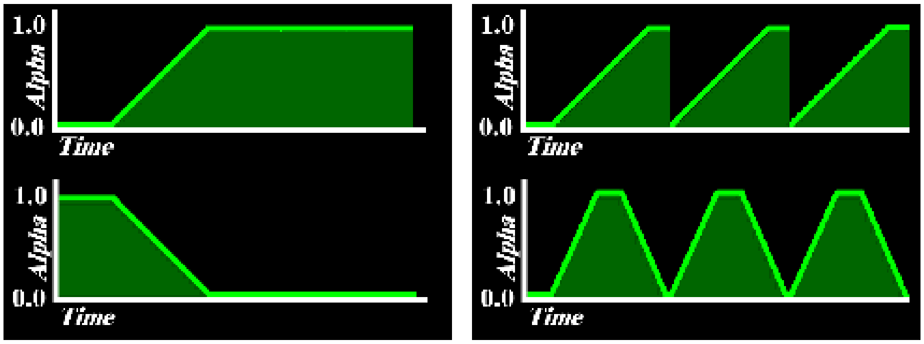
- Increasing and decreasing phases may be individually enabled or disabled and their acceleration controlled
  - *Increasing ramp* controls increasing acceleration
  - *Decreasing ramp* controls decreasing acceleration

Mapping time to alpha



Building 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 **`Object`**

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

- Alpha modes include **`INCREASING_ENABLE`** and **`DECREASING_ENABLE`** to enable use of increasing and/or decreasing portions of the alpha envelope

Alpha class methods

- **`Alpha`** methods also set stage parameters

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 start and end values
  - 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`**

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
  - A **`TransparencyInterpolator`** uses a **`TransparencyAttributes`** target
  - A **`SwitchValueInterpolator`** uses a **`Switch`** target
  - And so forth

Interpolator class hierarchy

- **`Interpolator`** extends **`Behavior`**, and is further extended for the different types of interpolators

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
```

Interpolator class methods

- Methods on **`Interpolator`** just set the alpha generator to use
- The subclasses of **`Interpolator`** add methods for specific types of interpolators

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

- Let's look at simple interpolators first . . . (they are all pretty much the same)

PositionInterpolator class methods

- **`PositionInterpolator`** linearly interpolations a position from a starting position to an ending position
- Methods on **`PositionInterpolator`** set the translation axis, value range, and target
  - Sets the translation in a **`TransformGroup`**

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

RotationInterpolator class methods

- **`RotationInterpolator`** linearly interpolations a rotation from a starting angle to an ending angle

- Methods on **`RotationInterpolator`** set the rotation axis, value range, and target
  - Sets the rotation in a **`TransformGroup`**

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

ScaleInterpolator class methods

- **`ScaleInterpolator`** linearly interpolations a scale value from a starting value to an ending value

- Methods on **`ScaleInterpolator`** set the scale axis, value range, and target
  - Sets the scale in a **`TransformGroup`**

Method
`ScaleInterpolator( Alpha alpha, TransformGroup target )`
`void setAxisOfScale( Transform3D axis )`
`void setMaximumScale( float scale )`
`void setMinimumScale( float scale )`
`void setTarget( TransformGroup target )`

ColorInterpolator class methods

- **`ColorInterpolator`** linearly interpolates a diffuse color (in a red-green-blue color space) from a starting color to an ending color

- Methods on **`ColorInterpolator`** set the value range and target
  - Sets the diffuse color in a **`Material`**

Method
`ColorInterpolator( Alpha alpha, Material target )`
`void setStartColor( Color3f color )`
`void setEndColor( Color3f color )`
`void setTarget( Material target )`

TransparencyInterpolator class methods

- **`TransparencyInterpolator`** linearly interpolates a transparency value from a starting value to an ending value

- Methods on **`TransparencyInterpolator`** set the value range and target
  - Sets the transparency in a **`TransparencyAttributes`**

Method
`TransparencyInterpolator( Alpha alpha, TransparencyAttributes target )`
`void setMaximumTransparency( float trans )`
`void setMinimumTransparency( float trans )`
`void setTarget( TransparencyAttributes target )`

SwitchValueInterpolator class methods

- **SwitchValueInterpolator** linearly interpolates a child index value from a starting index to an ending index

- Methods on **SwitchValueInterpolator** set the value range and target
  - Sets the child choice in a **Switch**

Method
SwitchValueInterpolator(Alpha alpha, Switch target)
void setFirstChildIndex(int index)
void setLastChildIndex(int index)
void setTarget(Switch target)

- (Whew! That’s all of the simple interplators)

RotationInterpolator example code

- Create a **`TransformGroup`** to animate

```
TransformGroup myGroup = new TransformGroup( );
```

- Create an alpha generator

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

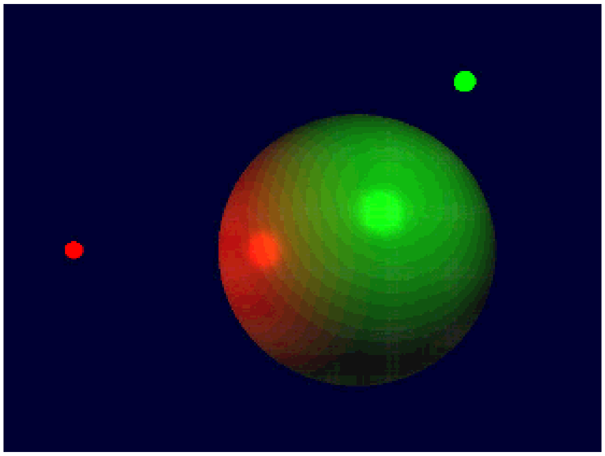
- Create and set up a rotation interpolator

```
RotationInterpolator mySpinner = new RotationInterpola
mySpinner.setAxisOfRotation( new Transform3D( ) );
mySpinner.setMinimumAngle( 0.0f );
mySpinner.setMaximumAngle( (float)(Math.PI * 2.0) );
```

- Set the scheduling bounds and add it to the scene

```
mySpinner.setSchedulingBounds( bounds );
myGroup.addChild( spinner );
```

RotationInterpolator example



[**SphereMotion**]

PathInterpolator class methods

- Methods on **`PathInterpolator`** set the alpha generator to use and the "knots" used for the path
  - Knots are specific alpha values that correspond to specific positions, rotations, etc. along a path
  - Interpolation is done between knots, then mapped to the corresponding interpolated position, rotation, etc.

- The subclasses of **`PathInterpolator`** add methods for specific types of path interpolators

Method
`PathInterpolator( Alpha alpha, float[] knots )`
`void setKnot( int index, float knot )`

- Let’s look at the various path interpolators . . . (and they too are pretty much all the same)

PositionPathInterpolator class methods

- **`PositionPathInterpolator`** interpolates a position along a path
- Methods on **`PositionPathInterpolator`** set the translation axis, path, and target
  - Sets the translation in a **`TransformGroup`**

Method
`PositionPathInterpolator( Alpha alpha, TransformGroup target, Transform3D axis, float[] knots, Point3f[] positions )`
`void setAxisOfTranslation( Transform3D axis )`
`void setPosition( int index, Point3f pos )`
`void setTarget( TransformGroup target )`

RotationPathInterpolator class methods

- **`RotationPathInterpolator`** interpolates a rotation along a path
- Methods on **`RotationPathInterpolator`** set the translation axis, path, and target
  - Sets the rotation in a **`TransformGroup`**

Method
`RotationPathInterpolator( Alpha alpha, TransformGroup target, Transform3D axis, float[] knots, Quat4f[] quats )`
`void setAxisOfRotation( Transform3D axis )`
`void setQuat( int index, Quat4f quat )`
`void setTarget( TransformGroup target )`

RotPosPathInterpolator class methods

- **`RotPosPathInterpolator`** interpolates a position and rotation along a path
- Methods on **`RotPosPathInterpolator`** set the translation axis, path, and target
  - Sets the translation and rotation in a **`TransformGroup`**

Method
`RotPosPathInterpolator( Alpha alpha, TransformGroup target, Transform3D axis, float[] knots, Quat4f[] quats, Point3f[] positions )`
`void setAxisOfRotPos( Transform3D axis )`
`void setPosition( int index, Point3f pos )`
`void setQuat( int index, Quat4f quat )`
`void setTarget( TransformGroup target )`

RotPosScalePathInterpolator class methods

- **`RotPosScalePathInterpolator`** interpolates a position, rotation, and scale along a path

- Methods on **`RotPosScalePathInterpolator`** set the translation axis, path, and target
  - Sets the translation, rotation, and scale in a **`TransformGroup`**

Method
`RotPosScalePathInterpolator( Alpha alpha, TransformGroup target, Transform3D axis, float[] knots, Quat4f[] quats, Point3f[] positions, float] scales )`
`void setAxisOfRotPosScale( Transform3D axis )`
`void setPosition( int index, Point3f pos )`
`void setQuat( int index, Quat4f quat )`
`void setScale( int index, float scale )`
`void setTarget( TransformGroup target )`

Summary

- An **`Interpolator`** behavior varies a value over time using two mappings
  - Time-to-alpha
  - Alpha-to-value

- An **`Alpha`** generator maps time to an alpha value that varies from 0.0 to 1.0 through several stages

- Specific interpolator types use an alpha generator, and a target node to vary position, rotation, color, transparency, etc.

Using specialized behaviors

Motivation

- As with interpolators, some behaviors are so common they are provided upfront by Java 3D
    - *Billboard* auto-rotation of shapes to face the viewer
    - Switching between shape *levels of detail* based upon distance to the viewer

Specialized behavior class hierarchy

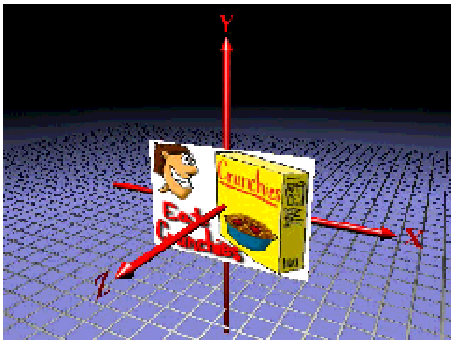
- Specialized behaviors are all extensions of **`Behavior`**

```
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
```

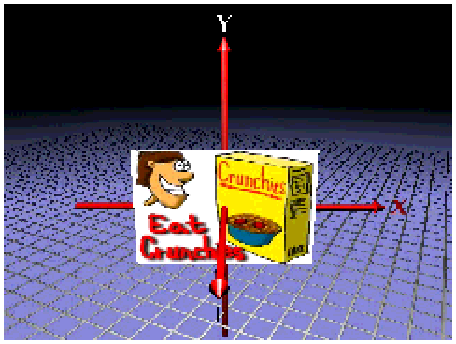
Using billboard behaviors

- 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`**

Using billboard behaviors



Viewer steps to the right . . .



. . . and the behavior immediately rotates the shape

Using billboard alignment modes

- Billboard rotation can be about:
  - An axis to pivot the `TransformGroup`
  - A point to arbitrarily rotate the `TransformGroup`
    - Rotation makes the group's Y-axis parallel to the viewer's Y-axis

Billboard class methods

- Methods on **`Billboard`** set the alignment mode, rotation axis or point, and the target
  - The default alignment mode is about the Y axis

Method
`Billboard( )`
`void setAlignmentMode( int mode )`
`void setAlignmentAxis( Vector3f axis )`
`void setRotationPoint( Point3f point )`
`void setTarget( TransformGroup group )`

- Alignment modes include **`ROTATE_ABOUT_AXIS`** (default) and **`ROTATE_ABOUT_POINT`**

Using level-of-detail behaviors

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

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

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

LOD class methods

- Methods on **`LOD`** manage a list of **`Switch`** groups to control based upon viewer distance

Method
`LOD( )`
`void setSwitch( Switch switch, int index )`
`void addSwitch( Switch switch )`
`void insertSwitch( Switch switch, int index )`
`void removeSwitch( int index )`

DistanceLOD class methods

- Methods on **`DistanceLOD`** set the distances at which detail switches should occur

Method
`DistanceLOD( )`
`void setDistance( int whichLOD, double distance )`

Summary

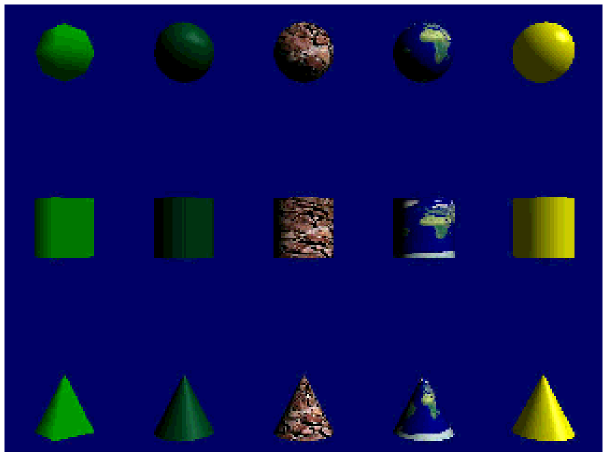
- **`Billboard`** automatically rotates a **`TransformGroup`** so that its Z-axis always points towards the viewer
- **`DistanceLOD`** automatically switches children in a **`Switch`** group based upon distance to the viewer

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
  - It supports various selection shapes
  - It can report the first, any, all, or all sorted hits

Example



[**PickWorld**]

Using the picking API

- The Java 3D API divides picking into two portions
  - Control: clicking with a 2D mouse or move a 6DOF wand
  - Selection: finding shapes that meet the search criteria

- Seperation enables interchangeable interaction methods

- The API designed for speed
  - Picking only works on bounds
  - Utilities provide more fine-grained pick support

Where is the API?

- The API is distributed among a number of classes . . .
- Enable pickability of any node via methods on **`Node`**
- Initiate a pick using methods on **`Locale`** or **`BranchGroup`**
- Pick methods take as an argument a **`PickShape`**
  - **`PickBounds`**, **`PickPoint`**, **`PickRay`**, **`PickSegment`**
- Pick methods return one or more **`SceneGraphPath`**s

Node class methods

- Methods on **`Node`** enable *pickability*

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

Locale and BranchGroup class methods

- Methods on **`Locale`** or **`BranchGroup`** initiate a pick on their children
  - 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

- **`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

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

- **`PickBounds`** checks the scene at a position, in a bounded volume
  - Pick occurs for shape bounds that intersect the bounded volume

PickShape class hierarchy

- **`PickShape`** extends **`Object`**
- This is further extended for various types of pick shapes

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

PickShape class methods

- **`PickShape`** provides no further methods
- The pick shape types extend **`PickShape`**

Method
`PickShape( )`

PickRay class methods

- Methods on **`PickRay`** set the position and aim direction used for a pick intersection

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

PickSegment class methods

- Methods on **`PickSegment`** set the starting and ending positions for the ray segment used for a pick intersection

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

PickPoint class methods

- Methods on **`PickPoint`** set the position used for a pick intersection

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

PickBounds class methods

- Methods on **`PickBounds`** set the bounding volume used for a pick intersection

Method
`PickBounds( )`
`PickBounds( Bounds bounds )`
`void set( Bounds bounds )`

Getting Pick Results

- The pick methods on **`Locale`** or **`BranchGroup`** return one or more **`SceneGraphPath`**s
- Each **`SceneGraphPath`** contains:
  - A **`Node`** for the shape that was picked
  - The **`Locale`** above it in the scene graph
  - A list of the **`Node`**s from the picked shape up to the **`Locale`**
  - The world-to-shape transform

SceneGraphPath class hierarchy

- **`SceneGraphPath`** extends **`Object`**

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

SceneGraphPath class methods

- Methods on **`SceneGraphPath`** get the shape (object) picked, the locale above it, the transform to it, and nodes on the path between the locale and the shape

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

Using the mouse for a pick

- Create a behavior that wakes up on mouse events
  - On a mouse release:
    - Construct a **`PickRay`** from the eye passing through the 2D mouse screen point
    - Initiate a pick to find all pick hits along the ray, sorted from closest to furthest
    - Get the first pick hit in the returned data
    - Do something to that picked shape
    - (Re)declare interest in mouse events

Picking example code

- Create a pick ray aimed using mouse screen data

```
PickRay myRay = new PickRay( rayOrigin, rayDirection
```

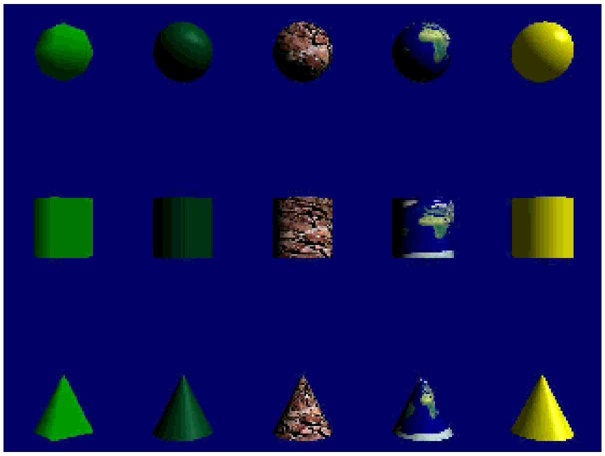
- Initiate a pick starting at a **`Locale`**

```
SceneGraphPath[ ] results = myLocale.pickAllSorted( my
```

- Get the first (closest) shape off the results

```
Node pickedObject = results[0].getObject( );
```

Picking example



[**PickWorld**]

Summary

- Picking selects a shape pointed at by the user
  - The pointing device can be anything (often the mouse)
- Pickability is enabled on a per-node basis
- Picking looks for the intersection of a **`PickShape`** with shape bounding volumes
  - **`PickBounds`**, **`PickPoint`**, **`PickRay`**, and **`PickSegment`**,
- A pick is initiated on a **`Locale`** or **`BranchGroup`**
- A pick returns one or more **`SceneGraphPath`**s for the shapes hit by the pick

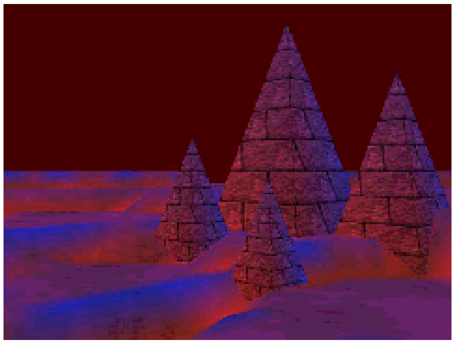
Creating backgrounds

Motivation

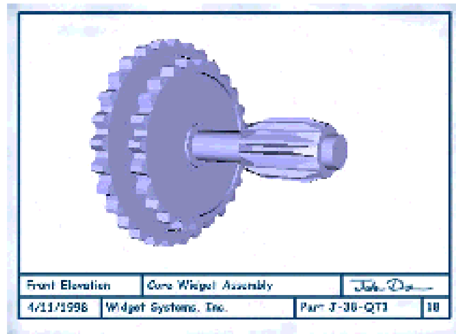
- You can add a *background* to provide context for foreground content

- 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 **`Background`**

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

Using 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

Using 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

Using 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)

Background class methods

- Methods on **`Background`** set the color, image, or geometry

Method
`Background( )`
`void setColor( Color3f color )`
`void setImage( ImageComponent2D image )`
`void setGeometry( BranchGroup group )`

Background color example code

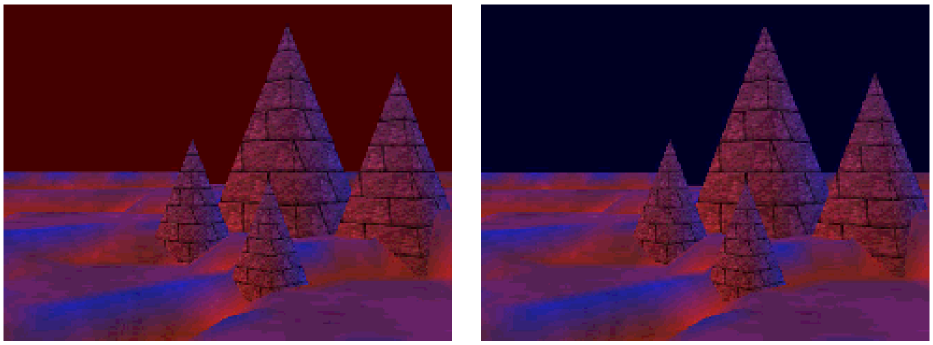
- Create a background

```
Background myBack = new Background( );
myBack.setColor( new Color3f( 0.3f, 0.0f, 0.0f ) );
```

- Set the application bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myBack.setApplicationBounds( myBounds );
```

Background color example



[**ExBackgroundColor**]

Background image example code

- Load a texture image

```
TextureLoader myLoader = new TextureLoader( "stars2.j
ImageComponent2D myImage = myLoader.getImage( );
```

- Create a background

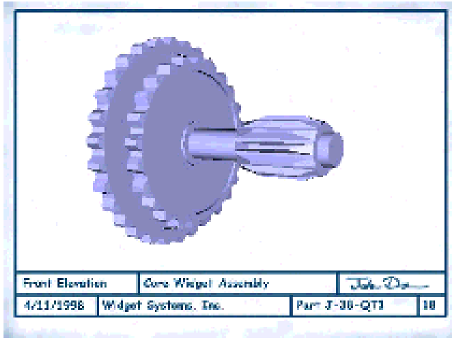
```
Background myBack = new Background( );
myBack.setImage( myImage );
```

- Set the application bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myBack.setApplicationBounds( myBounds );
```

Background image example

[**ExBackgroundImage**]



[**ExBluePrint**]

Background geometry example code

- Create background geometry

```
BranchGroup myBranch = createBackground( );
```

- Create a background

```
Background myBack = new Background( );
myBack.setGeometry( myBranch );
```

- Set the application bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myBack.setApplicationBounds( myBounds );
```

Using background application bounds

- A background is applied when:
  - The viewer's activation radius intersects its *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

- Methods on **`Background`** set the application bounds

Method
`void setApplicationBounds( Bounds bounds )`
`void setApplicationBoundingLeaf( BoundingLeaf leaf )`

Application bounds example code

- Set bounds relative to the background’s coordinate system

```
Background myBack = new Background( );
myBack.setApplicationBounds( myBounds );
```

- Or relative to a bounding leaf’s coordinate system

```
TransformGroup myGroup = new TransformGroup( );
BoundingLeaf myLeaf = new BoundingLeaf( myBounds );
myGroup.addChild( myLeaf );
. . .
Background myBack = new Background( );
myBack.setApplicationBoundingLeaf( myLeaf );
```

Summary

- `Background` sets the background color, image, or geometry

- Backgrounds are activated when the viewer's activation radius intersects the background's application bounds
  - Default is *no application bounds*, so never takes effect

Working with fog

Motivation

- Fog increases realism and declutters a scene

- Fog also obscures distant shapes, enabling you to turn them off and render the scene faster

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

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

- Both types of fog have:
  - A color (default is black)
  - A bounding volume and scope controlling the range of shapes to affect

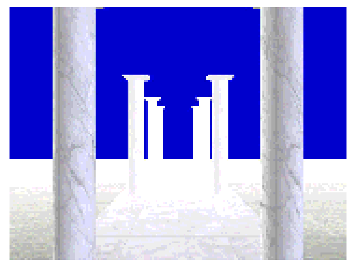
Method
`void setColor( Color3f color )`

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

Using 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

$$effect = e^{(-density * distance)}$$

$$color = effect * shapeColor + (1-effect) * fogColor$$

ExponentialFog class methods

- Methods on **`ExponentialFog`** set the fog density

Method
`ExponentialFog( )`
`void setDensity( float density )`

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

ExponentialFog example code

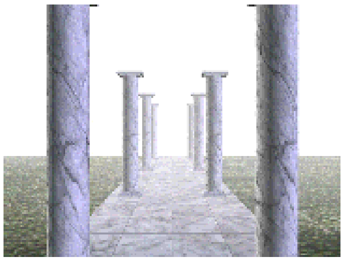
- Create fog

```
ExponentialFog myFog = new ExponentialFog( );
myFog.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
myFog.setDensity( 1.0f );
```

- Set the influencing bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myFog.setInfluencingBounds( myBounds );
```

ExponentialFog example



Haze

Light fog

Heavy fog

Black fog

[**`ExExponentialFog`**]

Using 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 methods

- Methods on **`LinearFog`** set the fog front and back distances

Method
`LinearFog( )`
`void setFrontDistance( double front )`
`void setBackDistance( double back )`

- Default front distance is 0.0
- Default back distance is 1.0

LinearFog example code

- Create fog

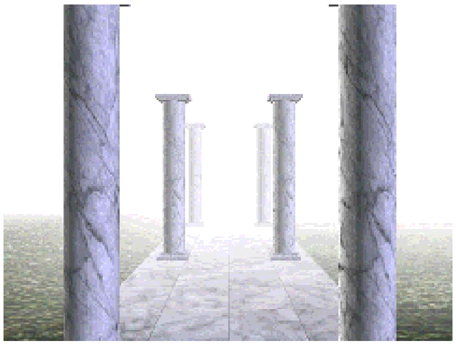
```
LinearFog myFog = new LinearFog( );
myFog.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
myFog.setFrontDistance( 1.0 );
myFog.setBackDistance( 30.0 );
```

- Set the influencing bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myFog.setInfluencingBounds( myBounds );
```

Working with fog

LinearFog example



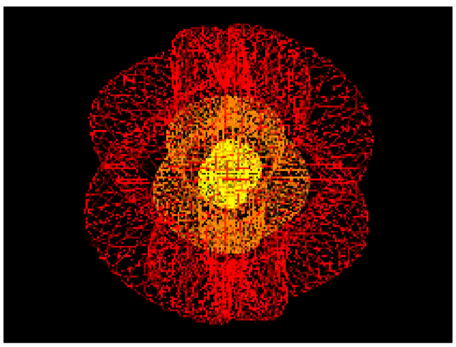
Distances wide apart

Distances close together

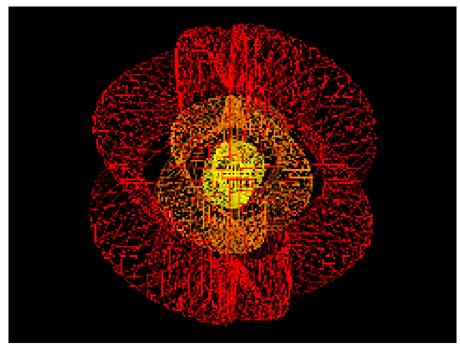
[**`ExLinearFog`**]

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**]

Using 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

- Methods on **`Fog`** set the influencing bounds and scope list

Method
`void setInfluencingBounds( Bounds bounds )`
`void setInfluencingBoundingLeaf( BoundingLeaf leaf )`
`void setScope( Group group, int index )`
`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 myFog = new LinearFog( );
myFog.setInfluencingBounds( myBounds );
```

- Or relative to a bounding leaf's coordinate system

```
TransformGroup myGroup = new TransformGroup( );
BoundingLeaf myLeaf = new BoundingLeaf( myBounds );
myGroup.addChild( myLeaf );
. . .
LinearFog myFog = new LinearFog( );
myFog.setInfluencingBoundingLeaf( myLeaf );
```

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 **`Leaf`**

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

Using clip application bounds

- A clip is applied when:
  - The viewer's activation radius intersects its *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

- Methods on **`Clip`** set the clip distance and application bounds

Method
`Clip( )`
`void setBackDistance( double back )`
`void setApplicationBounds( Bounds bounds )`
`void setApplicationBoundingLeaf( BoundingLeaf leaf )`

Clip example code

- Create a clip

```
Clip myClip = new Clip( );
myClip.setBackDistance( 30.0 );
```

- Set its application bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myClip.setApplicationBounds( myBounds );
```

Clip example

[**ExClip**]

Summary

- `ExponentialFog` creates fog that increases in density exponentially with distance to the user

- `LinearFog` creates fog that increases in density linearly with distance to the user

- Both types of fog have a fog color and influencing bounds

- `Clip` cuts away shapes beyond a clip distance and has 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)

- Clip is activated when the viewer's activation radius intersects the clip node's application bounds
  - Default is *no application bounds*, so never takes effect

Conclusions

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

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

- The latest version of these tutorial notes are available at the Sun Java 3D site

Thanks for coming!

Building text shapes

Motivation

- `Text3D` builds 3D text geometry for a `Shape3D`
  - Use to make annotation, signs, flying logos, etc.

- You could build your own 3D text from triangles and quadrilaterals
  - `Text3D` does it for you

Example

[**ExText**]

Building 3D text

- Building 3D text is a multi-step process

    1. Select a 2D font with **`java.awt.Font`**

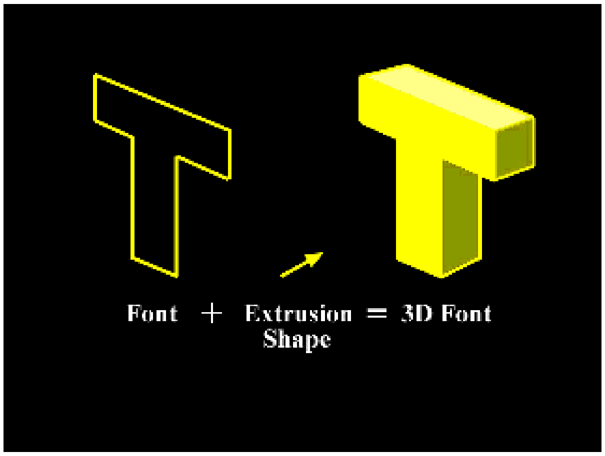
    2. Describe a 2D extrusion shape with **`java.awt.Shape`** in a **`FontExtrusion`**

    3. Create a 3D font by extruding the 2D font along the extrusion shape with a **`Font3D`**

    4. Create 3D text using a string and a **`Font3D`** in a **`Text3D`**

Building a 3D font

- Create a 3D font by sweeping a 2D font along a 2D extrusion shape



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**

FontExtrusion class methods

- Methods on **`FontExtrusion`** select the extrusion

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

FontExtrusion example code

- For a simple extrusion, use the default:

```
FontExtrusion myExtrude = new FontExtrusion( );
```

- This creates a straight-line extrusion shape 0.2 units deep

Font3D class methods

- Methods on **`Font3D`** build the 3D font from a 2D font and an extrusion

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

Font3D example code

- Get a 2D font

```
Font my2DFont = new Font(
    "Arial",       // font name
    Font.PLAIN,    // font style
    1 );           // font size
```

- Make a simple extrusion

```
FontExtrusion myExtrude = new FontExtrusion( );
```

- Then build a 3D font

```
Font3D my3DFont = new Font3D( my2DFont, myExtrude );
```

Text3D class hierarchy

- **`Text3D`** extends **`Geometry`** to describe 3D text geometry for a **`Shape3D`**

Class Hierarchy
java.lang.Object
└─ **javax.media.j3d.SceneGraphObject**
└─ **javax.media.j3d.NodeComponent**
└─ **javax.media.j3d.Geometry**
└─ **javax.media.j3d.Text3D**

Text3D class methods

- Methods on **`Text3D`** select the text string and 3D font

Method
`Text3D( )`
`void setString( String string )`
`void setFont3D( Font3d font )`

Text3D class methods

- Additional methods on **`Text3D`** select the starting position, alignment, character spacing, and character path

Method
`void setPosition( Point3f position )`
`void setAlignment( int alignment )`
`void setCharacterSpacing( float spacing )`
`void setPath( int Path )`

- Alignment types include **`ALIGN_FIRST`** (default), **`ALIGN_LAST`**, and **`ALIGN_CENTER`**
- Character paths include **`PATH_LEFT`**, **`PATH_RIGHT`** (default), **`PATH_DOWN`**, and **`PATH_UP`**

Text3D example code

- Build 3D text that says "Hello!", starting with a 2D font and extrusion to build a 3D font

```
Font my2DFont = new Font(
    "Arial",      // font name
    Font.PLAIN,   // font style
    1 );          // font size
FontExtrusion myExtrude = new FontExtrusion( );
Font3D my3DFont = new Font3D( my2DFont, myExtrude );
```

- Then build 3D text geometry using the font

```
Text3D myText = new Text3D( );
myText.setFont3D( my3DFont );
myText.setString( "Hello!" );
```

- Assemble the shape

```
Shape3D myShape = new Shape3D( myText, myAppear );
```

Building text shapes

Text3D example

[**ExText**]

Summary

- A *font extrusion* defines the depth of 3D text
- A *3D font* combines a font extrusion with a 2D font to make 3D character glyphs
- *3D text* geometry is built using a 3D font and a text string

Controlling the appearance of textures

Motivation

- Texture image colors can replace, modulate, or blend with shape color
  - Different *texture modes* are useful for different effects
  - Some are faster to draw than others

- Different texture images can be used at different distances between the shape and the user
  - Use lower resolution images for distant shapes
  - This is known as *Mip-mapping*

Combining texture and shape colors

- A texture image may contain:
  - A red-green-blue color at each pixel
  - A transparency, or *alpha* value at each pixel
- Typically, image color modulates shape color
  - Darkly shaded parts of the shape use a darkened texture, etc.

Blending textures using alpha

- *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
```

- Texture alpha values can control color blending

- Texture color values can do spectral color filtering, using color as three alpha values

Using texture modes

- The *Texture mode* in **`TextureAttributes`** 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*

Controlling the appearance of textures

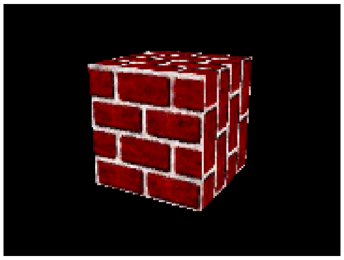
Using texture modes

Mode	Result color	Result transparency
`REPLACE`	T_{rgb}	T_a
`DECAL`	$S_{rgb}*(1-T_a)+T_{rgb}*T_a$	S_a
`MODULATE`	$S_{rgb}*T_{rgb}$	S_a*T_a
`BLEND`	$S_{rgb}*(1-T_{rgb})+B_{rgb}*T_{rgb}$	S_a*T_a

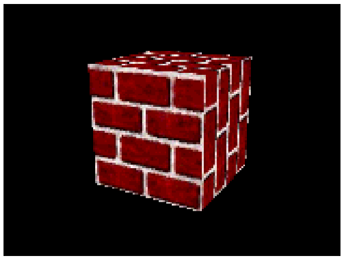
- Where:
  - S_{rgb} is the color of the shape being texture mapped
  - S_a is the alpha of the shape being texture mapped
  - T_{rgb} is the texture pixel color
  - T_a is the texture pixel alpha
  - B_{rgb} is the shape blend color
  - B_a is the shape blend alpha

Controlling the appearance of textures

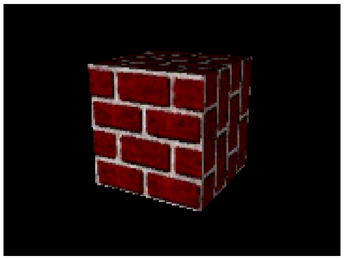
Using texture modes



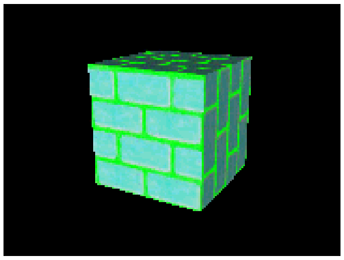
`REPLACE`



`DECAL`



`MODULATE` with white



`BLEND` with green

Using texture modes

- In typical use:
  - Use **`REPLACE`** for emissive textures
    - Glowing "neon" textures
    - Textures where lighting is painted in
  - Use **`MODULATE`** on a white shape for shaded textures
    - Most textured shaded surfaces
  - Use **`BLEND`** on a colored shape for colorized textures
    - Colorizing a grayscale woodgrain, marble, etc.

TextureAttributes class methods

- Methods on **`TextureAttributes`** set the texture mode and blend color
  - **`REPLACE`** is the default mode
  - Black is the default blend color

Method
`void setTextureMode( int mode )`
`void setTextureBlendColor( Color4f color )`

- Texture modes include **`MODULATE`**, **`DECAL`**, **`BLEND`**, and **`REPLACE`** (default)

Texture mode example code

- Create **`TextureAttributes`**

```
TextureAttributes myTA = new TextureAttributes( );
```

- Set the texture mode to **`MODULATE`**

```
myTA.setTextureMode( Texture.MODULATE );
```

- Set the texture attributes on an **`Appearance`**

```
Appearance myAppear = new Appearance( );
myAppear.setTextureAttributes( myTA );
```

Using texture mip-map modes

- *Mip-mapping* is an anti-aliasing technique that uses different texture versions (levels) at different distances from the user
  - You can have any number of *levels*
  - Level 0 is the base image used when the user 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 how a texture is interpolated when a scene pixel maps to multiple texture pixels (texels)

`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 how a texture is interpolated when a scene pixel 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 methods

- Methods on **`Texture`** control mip-mapping and filtering
  - **`BASE_LEVEL`** is the default mip-map mode
  - **`BASE_LEVEL_POINT`** is the default filter

Method
`void setMipMapMode( int mode )`
`void setMinFilter( int minFilter )`
`void setMagFilter( int maxFilter )`

Texture filter example code

- Load a texture image

```
TextureLoader myLoader = new TextureLoader( "brick.jpg
ImageComponent2D myImage = myLoader.getImage( );
```

- Create a **Texture2D** using the image, and turn it on

```
Texture2D myTex = new Texture2D( );
myTex.setImage( 0, myImage );
myTex.setEnable( true );
```

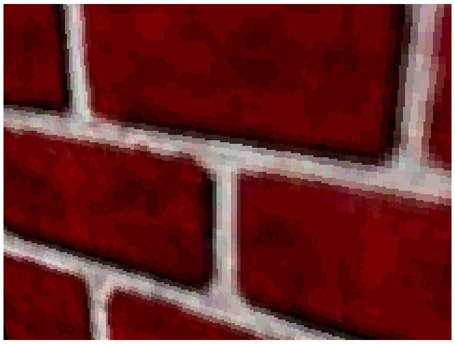
- Set the filtering types

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

- Create an **Appearance** and set the texture in it

```
Appearance myAppear = new Appearance( );
myAppear.setTexture( myTex );
```

Texture filter example



BASE_LEVEL_POINT

No interpolation

BASE_LEVEL_LINEAR

Linear interpolation of 4 nearest neighbors

Summary

- The *texture mode* controls how texture color and alpha values **`REPLACE`**, **`MODULATE`**, **`BLEND`**, or **`DECAL`** with the shape color

- *Mip-mapping* uses different versions (levels) of an image at different distances from the user

- *Minification* and *Magnification* filters control how individual, or neighboring texture pixels contribute to an image

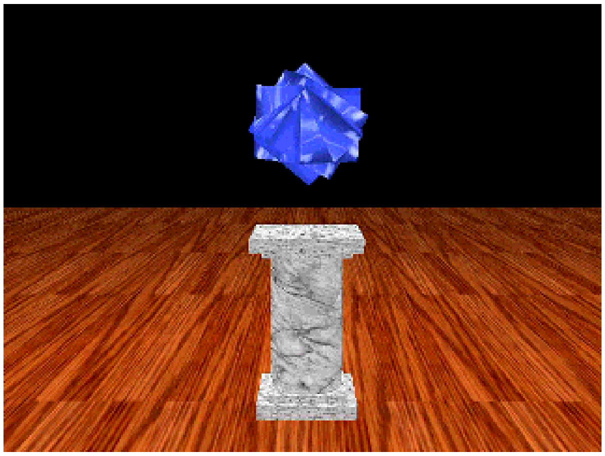
Adding sound

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 **`Sound`**

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**

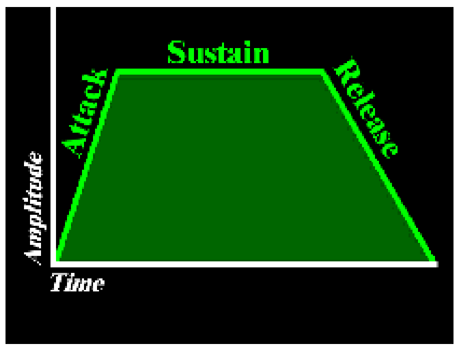
MediaContainer class methods

- Methods on **`MediaContainer`** select the file path or URL for the sound file
  - Setting the URL triggers loading of the sound

Method
`MediaContainer( )`
`void setUrl( String path )`
`void setUrl( URL url )`

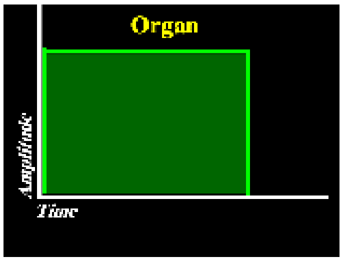
Looking at sound envelopes

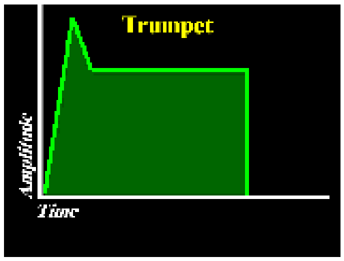
- Sound files have a built-in amplitude *Envelope* with three stages:
  - *Attack*: the start of the sound
  - *Sustain*: the body of the sound
  - *Release*: the ending decay of the sound

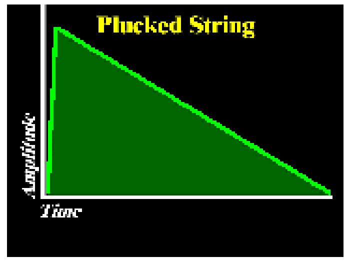


Looking at sound envelopes

- 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

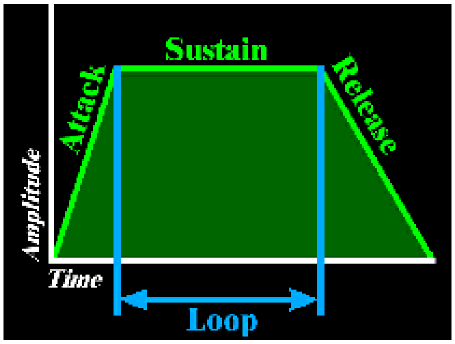






Looping sounds

- 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

- Methods on **`Sound`** select the sound data, turn on the sound, set its volume, and loop sound playback
  - By default, sounds are disabled, have a gain of 1.0, and are not looped

Method
`void setSoundData( MediaContainer sound )`
`void setEnable( boolean onOff )`
`void setInitialGain( float amplitude )`
`void setLoop( int count )`

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

Using 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 methods

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

Method
`BackgroundSound( )`

BackgroundSound example code

- Load sound data

```
MediaContainer myWave = new MediaContainer( "canon.wav
```

- Create a sound

```
BackgroundSound mySound = new BackgroundSound( );
mySound.setSoundData( myWave );
mySound.setEnable( true );
mySound.setInitialGain( 1.0f );
mySound.setLoop( -1 );  // Loop forever
```

- Set the scheduling bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
mySound.setSchedulingBounds( myBounds );
```

Using point sounds

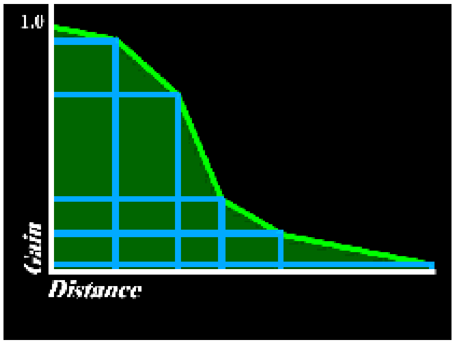
- **`PointSound`** extends the **`Sound`** class
  - Sound waves emit radially from a point in all directions
  - Similar idea as a **`PointLight`**

- 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

Varying gain 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 methods

- Methods on **`PointSound`** set the sound position and attenuation
  - The default position is (0.0,0.0,0.0) with no attenuation

Method
`PointSound( )`
`void setPosition( Point3f pos )`
`void setDistanceGain( Point2f[] atten )`

PointSound example code

- Load sound data

```
MediaContainer myWave = new MediaContainer( "willow1.\
```

- Create an attenuation array

```
Point2f[] myAtten = {
    new Point2f( 100.0f, 1.0f ),
    new Point2f( 350.0f, 0.5f ),
    new Point2f( 600.0f, 0.0f )
};
```

PointSound example code

- Create a sound

```
PointSound mySound = new PointSound( );
mySound.setSoundData( myWave );
mySound.setEnable( true );
mySound.setInitialGain( 1.0f );
mySound.setLoop( -1 );  // Loop forever
mySound.setPosition( new Point3f( 0.0f, 1.0f, 0.0f )
mySound.setDistanceGain( myAtten );
```

- Set the scheduling bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
mySound.setSchedulingBounds( myBounds );
```

Using 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

Varying gain 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

Varying gain and frequency 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 methods

- Methods on **`ConeSound`** aim the sound, set its distance gain front and back, and control angular attenuation
  - By default, cone sounds are aimed in the positive Z direction with no distance or angular attenuation

Method
`ConeSound( )`
`void setDirection( Vector3f dir )`
`void setDistanceGain( Point2f[] front, Point2f[] back )`
`void setBackDistanceGain( Point2f[] back )`
`void setAngularAttenuation( Point3f[] atten )`

- Attenuation angles are in the range 0.0 to PI radians

ConeSound example code

- Load sound data

```
MediaContainer myWave = new MediaContainer( "willow1.\
```

- Create attenuation arrays

```
Point2f[] myFrontAtten = {
    new Point2f( 100.0f, 1.0f ),
    new Point2f( 350.0f, 0.5f ),
    new Point2f( 600.0f, 0.0f )
};
Point2f[] myBackAtten = {
    new Point2f( 50.0f, 1.0f ),
    new Point2f( 100.0f, 0.5f ),
    new Point2f( 200.0f, 0.0f )
};
Point3f[] myAngular = {
    new Point3f( 0.000f, 1.0f, 20000.0f ),
    new Point3f( 0.785f, 0.5f,  5000.0f ),
    new Point3f( 1.571f, 0.0f,  2000.0f ),
};
```

ConeSound example code

- Create a sound

```
ConeSound mySound = new ConeSound( );
mySound.setSoundData( myWave );
mySound.setEnable( true );
mySound.setInitialGain( 1.0f );
mySound.setLoop( -1 );  // Loop forever
mySound.setPosition( new Point3f( 0.0f, 1.0f, 0.0f )
mySound.setDirection( new Vector3f( 0.0f, 0.0f, 1.0f
mySound.setDistanceGain( myFrontAtten, myBackAtten );
mySound.setAngularAttenuation( myAngular );
```

- Set the scheduling bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
mySound.setSchedulingBounds( myBounds );
```

Setting scheduling bounds

- A sound is hearable (if it is playing) when:
  - The viewer's activation radius intersects its *scheduling bounds*
  - Multiple sounds can be active at once
  - Identical to behavior scheduling

- Sound bounding enables different sounds for different areas of the scene

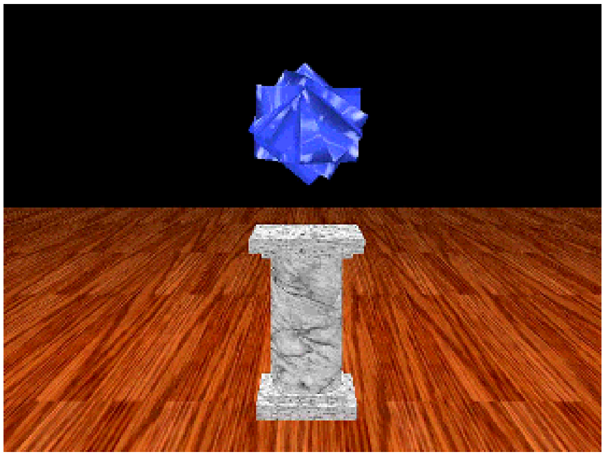
- By default, sounds have no scheduling bounds and are never hearable!
  - ***Common error:*** forgetting to set scheduling bounds

Sound class methods

- Methods on **`Sound`** set the scheduling bounds

Method
`void setSchedulingBounds( Bounds bounds )`
`void setSchedulingBoundingLeaf( BoundingLeaf leaf )`

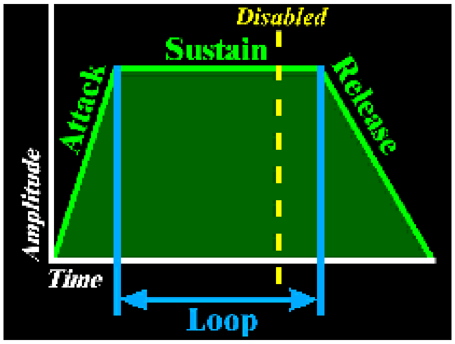
Sound example



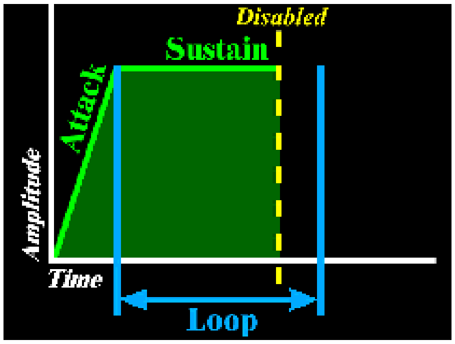
[**ExSound**]

Controlling the sound 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



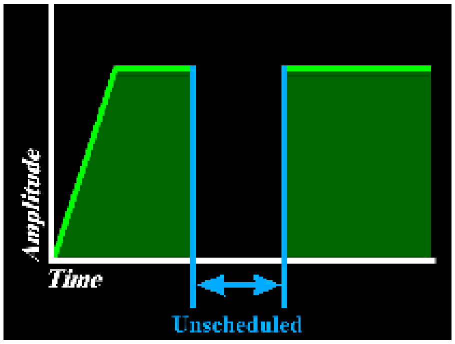
Release enabled



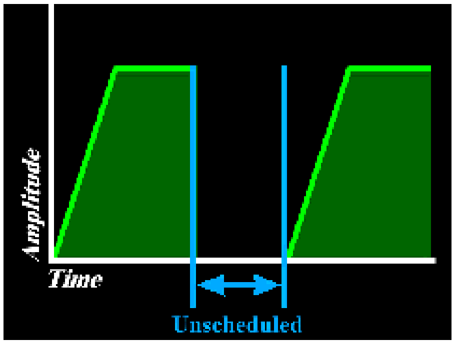
Release disabled

Enabling 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

- Methods on **`Sound`** control the release, continuous playback, and priority
  - By default, the release and continuous playback or disabled and the priority is 1.0

Method
`void setReleaseEnable( boolean onOff )`
`void setContinuousEnable( boolean onOff )`
`void setPriority( float ranking )`

Summary

- All sounds use sound data from a **`MediaContainer`**
- For all sounds you can turn them on or off, set their gain, release style, continuous playback style, looping, priority, and scheduling bounds
- **`BackgroundSound`** creates a sound that emits everywhere, flooding the area with sound
- **`PointSound`** creates a sound that emits from a position, radially in all directions, with distance attenuation
- **`ConeSound`** creates a sound that emits from a position in a forward direction, with distance and angular attenuation
- Sounds are hearable (if playing) when the viewer's activation radius intersects the sound's scheduling bounds
    - Default is *no scheduling bounds*, so nothing is hearable!

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 **`Soundscape`**

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

Setting Soundscape application bounds

- A *Soundscape* affects sound when:
  - The viewer's activation radius intersects its *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

Soundscape class methods

- Methods on **`Soundscape`** set the aural attributes and application bounds

Method
`Soundscape( )`
`void setApplicationBounds( Bounds bounds )`
`void setApplicationBoundingLeaf( BoundingLeaf leaf )`
`void setAuralAttributes( AuralAttributes aural )`

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 **`AuralAttributes`**

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

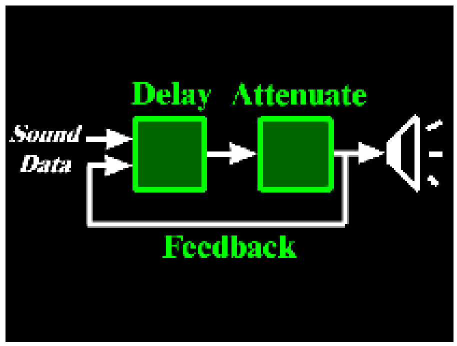
Controlling reverberation

- 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

- Methods on **`AuralAttributes`** control reverberation
  - All values are zero by default

Method
`AuralAttributes( )`
`void setReverbDelay( float delay )`
`void setReflectionCoefficient( float coeff )`
`void setReverbOrder( int order )`

- A reverb order of -1 repeats echos until they die out

Controlling 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

Controlling frequency 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

AuralAttributes class methods

- Methods on **`AuralAttributes`** control gain, filtering, and rolloff
  - By default, there is no filtering and gain and rolloff are 1.0

Method
`void setAttributeGain( float gain )`
`void setRolloff( float rolloff )`
`void setDistanceFilter( Point2f[] atten )`

Controlling 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

- Methods on **`AuralAttributes`** control frequency and velocity scaling for Doppler shift
  - By default, frequencies are scaled by 1.0 and velocity by 0.0

Method
`void setFrequencyScaleFactor( float scale )`
`void setVelocityScaleFactor( float scale )`

AuralAttributes example code

- Set up aural attributes

```
AuralAttributes myAural = new AuralAttributes( );
myAural.setReverbDelay( 2.0f );
myAural.setReverbOrder( -1 );  // Until dies out
myAural.setReflectionCoefficient( 0.2f );  // dampen
```

- Create the sound scape

```
Soundscape myScape = new Soundscape( );
myScape.setAuralAttributes( myAural );
```

- Set the application bounds

```
BoundingSphere myBounds = new BoundingSphere(
    new Point3d( ), 1000.0 );
myScape.setApplicationBounds( myBounds );
```

Summary

- **`Soundscape`** anchors a set of **`AuralAttributes`** to be applied within a bounded area
- **`AuralAttributes`** control reverberation, distance filtering, and Doppler shift within that area
- Soundscapes apply when the viewer's activation radius intersects the soundscape's application bounds
  - Default is *no application bounds*, so nothing is affected!