Java 3D joy ride

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Should I take this tutorial?

This tutorial is intended for Java programmers who do not have any 3D programming experience. We'll start with some basic 3D concepts, and explore how to build a scene graph -- the fundamental object used to describe the scene we want to render. Then we'll get to some more powerful features of Java 3D. The emphasis will be on using some of the Java 3D utility classes to jump-start your programming.

Getting help

The Java 3D API is not typically included with your JDK. You can download the Java 3D API for Windows and Solaris platforms from the *Sun Java 3D Web site*. Links to other versions (AIX, HP-UX, IRIX, and Linux) are also located on this site, as well as many tutorial and learning resources.

Another useful Web site is the *Java 3D Community site*. It includes an extremely useful FAQ, which covers a lot of common problems that many Java 3D programmers may encounter.

For technical questions about the content of this tutorial, contact the author, Suzy Deffeyes at *suzyq@us.ibm.com*. Suzy Deffeyes is a 3D software engineer at IBM in Austin, Texas. She is the developer responsible for the original release of *Java 3D for AIX*, and represents IBM on the Java expert group for *3D Media Utilities*. She is currently a member of IBM's Linux Technology Center. Her past projects included OpenGL API design and development, Direct 3D driver development, and C++ scene graph technologies. She also did the AIX ports for Quake and Quake 2, and ensured that they were thoroughly tested.

A word about the samples

All of the images except the Quake screenshot in this tutorial were generated using Java 3D, and captured using the NCSA Java 3D Portfolio (see Resources on page 26 .) Additionally, most of the images are VRML files that were loaded using the VRML loader from the *Web3D Consortium*. The Quake2 image was captured with Quake2 for AIX, using a GXT6000P graphics adapter. (Quake 2 uses OpenGL instead of Java 3D, but the basic 3D concepts are the same.)

The code samples all borrow from the Java 3D SDK samples. In most of the samples, the "meat" is in the **`createSceneGraph()`** method.

Section 2. Life from a Java 3D point of view

Philosophy

The design of the Java 3D API is a significant departure from previous popular 3D graphics APIs, like OpenGL and Direct3D, which were low-level procedural APIs that were closely tied to the design of 3D hardware. Java 3D is a powerful, object-oriented API that provides a lot of functionality beyond what we think of as a "3D graphics API." Java programmers will likely find the Java 3D programming environment to be familiar and comfortable. Java 3D does a lot to manage your graphics data for you, allowing you to concentrate on other parts of your programming.

With Java 3D, you first set up all your graphics objects (also called *geometry objects*) in a *scene graph*, which is a hierarchical model containing all the information about the objects in your scene and how they will be rendered. Then, you hand the scene graph over to Java 3D for rendering. You don't have to write any code to handle displaying your data -- Java 3D does that for you. You get to program at a higher level with the many built-in power tools.

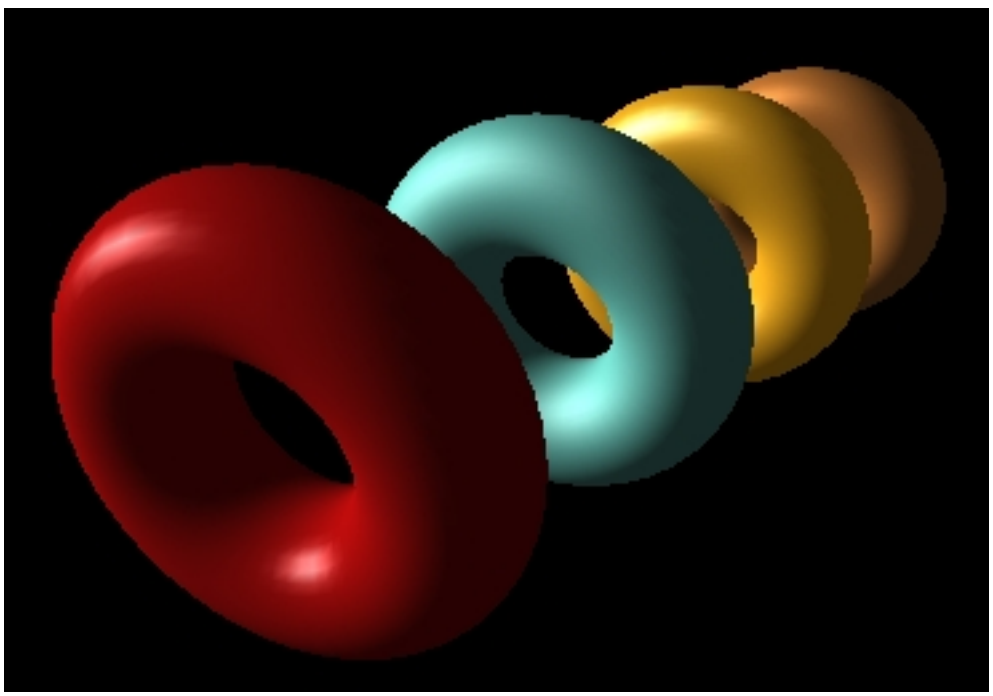
The need for speed

3D programmers are pretty fussy about performance, and with good reason -- their applications tend to be very performance sensitive. 3D application users notice very quickly if a spinning CAD model isn't spinning smoothly, or if they can't interactively grab an object and move it.

Thankfully, Java 3D can take advantage of any 3D acceleration that your graphics adapter provides. Java 3D ultimately generates OpenGL calls in a JNI layer that can be accelerated by your graphics card. OpenGL accelerated adapters are common in newer workstations, so your Java 3D programs should be hardware accelerated.

So what's the Point?

3D graphics will add a whole new dimension to your life: the dimension of **z**. In a three-dimensional **(x,y,z)** coordinate, the **Z** component specifies distance from the viewer. Java 3D uses **z** values to remove non-visible surfaces of distant obscured objects. The **z** values of the red torus in the image below are small because it is close to the viewer. It will obscure portions of the blue torus when the **Z** values of both tori are compared during rendering.



A 3D object contains a collection of coordinates rendered together (see The Primitive class on page 10). You can render them as points, lines, and polygons. A game programmer might want to use points to simulate a spray of bullets as a monster charges toward a player. A CAD designer might want to render using lines in order to see more detail about the very precise object she is designing. And after most of a car object has been designed, the designer can render it using filled polygons instead of wire-frame. While filled polygons will look more realistic, the image doesn't allow the designer to easily work with the nitty-gritty data describing the surface of the object.

In this tutorial, we'll stick to drawing polygons, because that's where most of the fun is.

Moving things around

After we have created the objects we want to display, we can move and scale them by using 3D *transformations*, in essence animating the objects. For example, when you're playing Quake, the bad guy charges toward you when the game manipulates his 3D transformation. The location, direction, and orientation of your view (before you are fragged!) is called the *viewpoint*. As you sneak around looking for more ammo, a transformation changes your viewpoint.

Transformations are specified as matrices in the powerful **Transform3D** class. **Transform3D** has many helper functions for specifying common transformations, such as translations, rotations, and scaling.

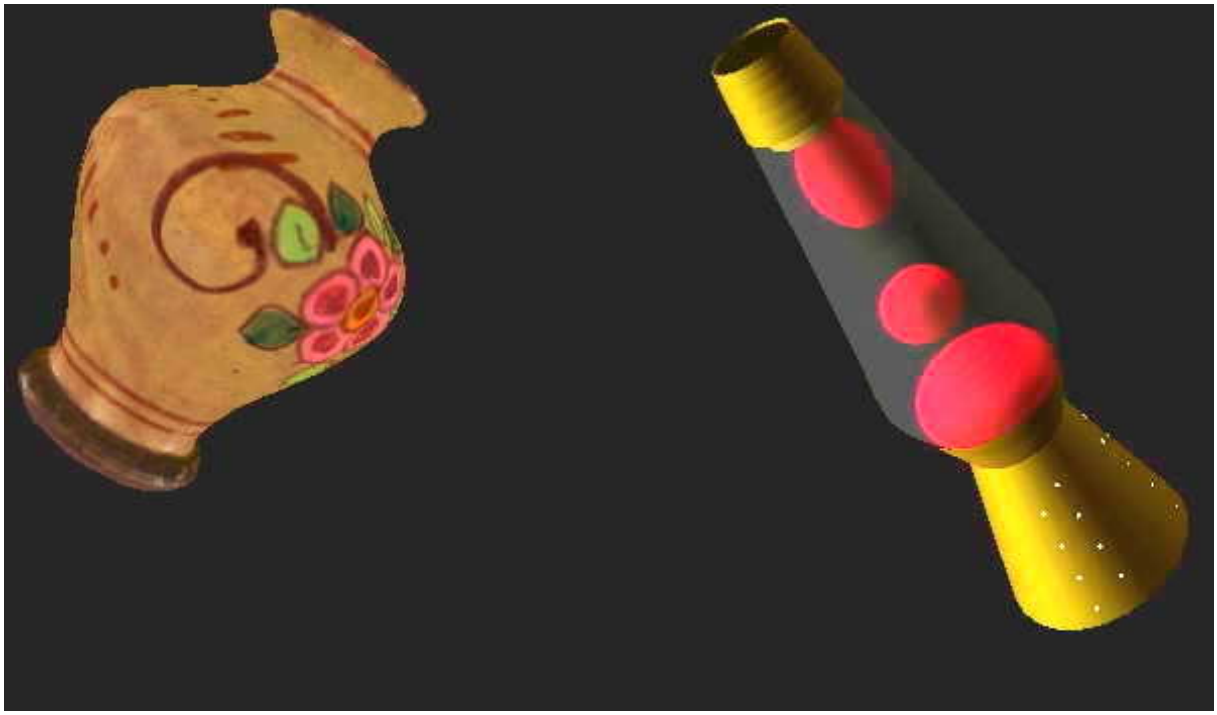
Lighting and other effects

In addition to specifying what objects appear in our scene, we can also control how they appear by specifying lighting effects (see The Light class on page 16 for more details on lighting). You can specify the type of lighting effect, like a spotlight, and the color of the light. You can also apply fog effects to your scene and set up automated behaviors of your objects.

Texture mapping (commonly referred to as *wallpapering*) is used to provide more realism to a scene. For instance, you can apply a wood grain image on an object's surface to simulate an oak table top.

Geometry objects don't have to be opaque; they can be transparent or translucent. The lava

lamp in the image below uses both transparency and lighting effects, and the pottery is using texture mapping.



Section 3. Scene graph nuts and bolts

The scene graph tree

A Java 3D scene graph is a tree with two parts, or branches: content and view. The view branch contains all the gory details of the complex Java 3D viewing model, and it defines the viewpoint. The good news is that for most simple applications we can use the universe utility classes, specifically the aptly-named **`SimpleUniverse`**, to handle most of the complexity of view management.

The content branch describes *what* you see in your scene. It contains all your graphics objects (spheres, boxes, or more complicated geometry objects), the transformations that move them around, lights, behaviors, group nodes, and fog. We will focus most of our effort on the content branch.

Group nodes

`Group` objects make up the interior *nodes* of the content branch of the scene graph. You can use **`Group`** nodes to organize your scene graph into related pieces. Each **`Group`** node contains a number of children that will be rendered when the node is processed. **`Switch`** nodes and **`TransformGroup`** nodes are specialized **`Group`** nodes that allow you to exert futher control over your scene graph.

`Switch` nodes limit which children are visited during rendering, giving you control over which portions of your scene graph are rendered. Using our Quake example, you could group all the different weapons under one **`Switch`** node, allowing you to render only the current weapon being used.

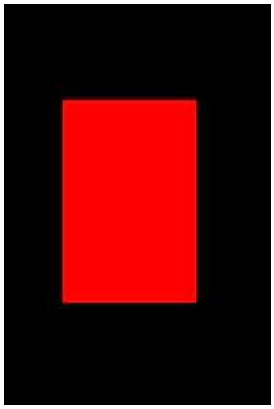
`TransformGroup` nodes apply a 3D transformation -- such as translation, scaling, or rotation -- to their children during processing, allowing you to move, rotate, or scale entire portions of your scene graph.

Capability bits

Java 3D will optimize the rendering of your scene graph by precalculating values where possible. If you want to change certain aspects of the scene during rendering, you must indicate what data you will want to change later by using capability bits. For example, animating an object requires changing the transformation that affects the object. To do this, you would enable the **`ALLOW_TRANSFORM_WRITE`** bit in your **`TransformGroup`** like this:

```
suzySpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
```

You can make any changes you like to scene graph data before you attach it to the universe. Any changes after Java 3D starts rendering, however, are allowed only on objects with the proper capability bits set.



UglyCube example

Take a look at UglyCube.java on page 28 , the source for this example. But let me warn you, it's really boring.

UglyCube just displays a cube. It's chopped down from the **HelloUniverse** sample that ships with the Java 3D SDK. You create a **Canvas3D** to draw on, and create a **SimpleUniverse** to handle all the view management for you. The top of the content branch is always a **BranchGroup**. We add the cube as a child, and hand over the **BranchGroup** for rendering.

After you have added the **BranchGroup** to the **SimpleUniverse**, Java 3D will take over rendering in a continuous loop. We'll add different types of nodes, such as lights and behaviors, later.

Here are the important lines from the example:

```
Canvas3D c = new Canvas3D(
    SimpleUniverse.getPreferredConfiguration());
setLayout(new BorderLayout());
add("Center", canvas);
BranchGroup scene = new BranchGroup();
scene.addChild(new ColorCube(0.4));
SimpleUniverse u = new SimpleUniverse(c);
u.getViewingPlatform().setNominalViewingTransform();
u.addBranchGraph(scene);
```

See, I told you it was boring.

Scene graph key points

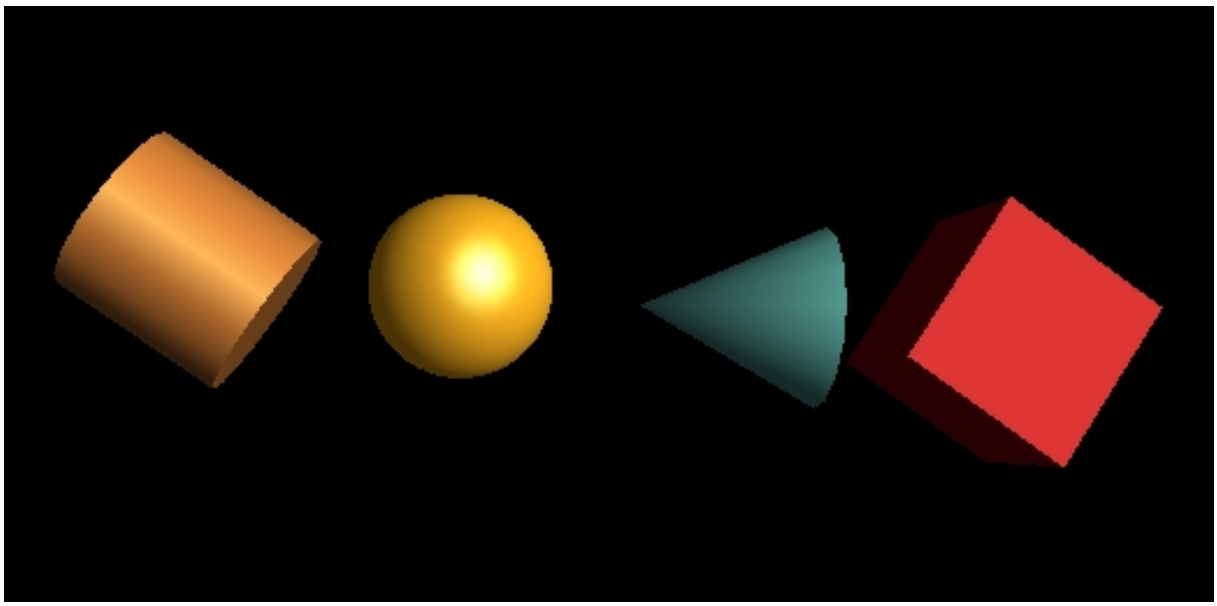
* **SimpleUniverse** creates the view branch of the scene graph.
* **Group** nodes give hierarchical structure onto your scene graph.
* Capability bits allow access to data in your scene graph nodes.

Section 4. Let's make some shapes

The Primitive class

`Primitive` is an abstract class for geometry objects that can be used as simple building blocks in your scene graph. Java 3D includes several useful concrete subclasses of **`Primitive`** -- **`Sphere`**, **`Box`**, **`Cone`**, and **`Cylinder`** -- that allow you to create basic objects easily without having to specify a lot of data. For example, when using the **`Sphere`** class, you simply specify a radius, and all the vertex data is generated for you.

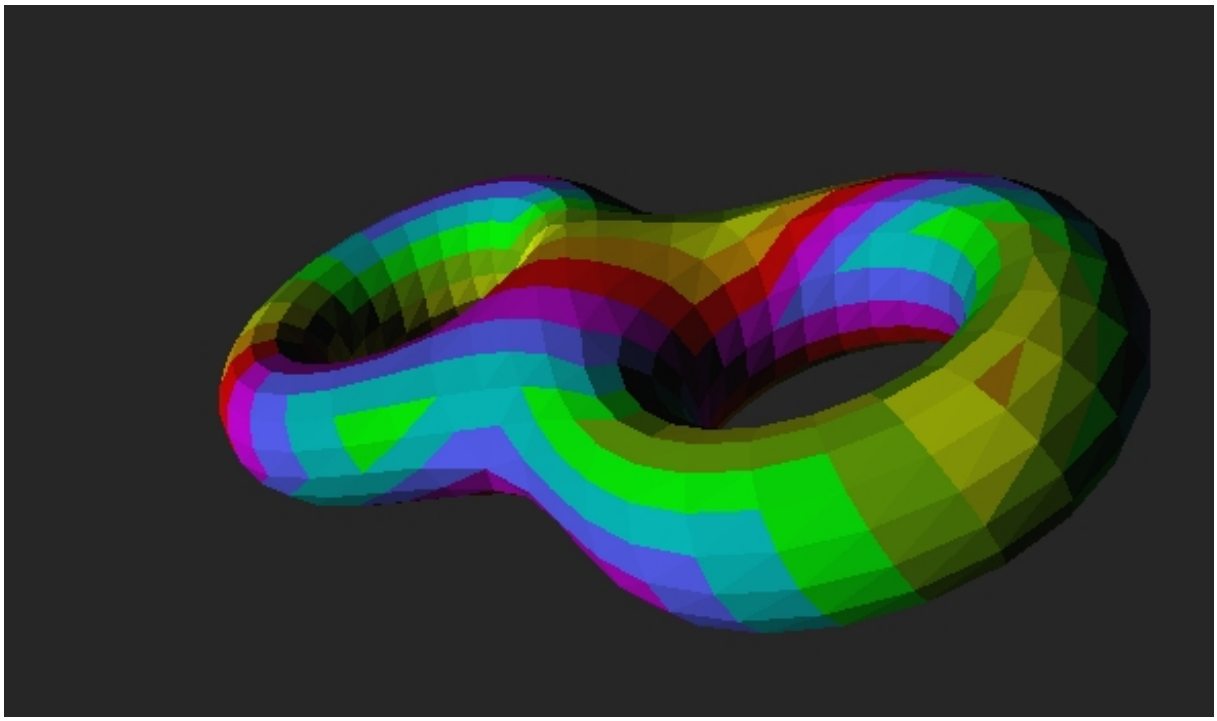
We'll use **`Primitive`**s exclusively in this tutorial so you don't get bogged down with the details of having to specify all the graphics data.



The Shape3D class

If you are not using one of the **`Primitive`** classes, you'll have to use the **`Shape3D`** class to specify all of the vertex data. You can specify data as triangles, quadrilaterals, lines, and points. A geometric representation of a sphere would be defined as a polygonal mesh, typically using strips of connected triangles or quads.

The image below should give you an idea of how strips are joined to make a mesh. The strips are different colors to make them stand out. In this **`Shape3D`** object, each strip of triangles shares common vertices with the adjacent triangle strip, which makes a mesh surface when it is rendered.



Per-vertex data

At a minimum, every vertex must have a location value (coordinate). In addition to location values, you can specify other items for each vertex, such as a color value, normal vector, and texture coordinates. Normal vectors are used for lighting effects, and texture coordinates are used when applying textures to the surface via texture mapping. Additionally, each vertex could also have an *alpha*, or transparency, value specified with its color.

Fortunately, vertex normals and texture coordinates are generated for you when using the **`Primitive`** classes. We will explore vertex normals and texture coordinates, which are the most common per-vertex attributes.

Appearance objects

While you can can specify a great deal of data with each vertex, many of your graphics effects are applied using the **`Appearance`** object. This object describes the overall attributes of an object's surface. Each **`Shape3D`** and **`Primitive`** object will have its own **`Appearance`** object, and each **`Appearance`** object contains several attribute objects. For example, an **`Appearance`** object can contain both a **`ColoringAttributes`** object and a **`RenderingAttributes`** object.

As you can see, with all the various types of graphics data, Java 3D applications can get complicated in a hurry. For simplicity's sake, we will look only at the **`Texture`** and **`Material`** attribute classes.

Pop quiz

Let's test your knowledge so far with this quick quiz. The **`Primitive`** base class is used for:

1. High-level geometry objects generated by Java 3D
2. Defining spheres, cones, cylinders, and boxes
3. Specifying prehistoric cave drawings
4. Both 1 and 2

Section 5. Transform your life (or at least your geometry)

The Transform3D class

Transformations allow you to move, rotate, or resize geometry objects in your scene and can be used to affect how the scene is viewed. You will use **Transform3D** objects regularly when doing Java 3D programming. A **Transform3D** object represents a transformation matrix. **Transform3D** objects are commonly used by the **TransformGroup** class. We aren't going to go through the dozens of methods in **Transform3D** -- just some of the helper functions -- and thankfully (for me anyway) we won't be getting into linear algebra.

Move me, zoom me, spin me

Let's begin by looking at some of the helper functions in the **Transform3D** class:

* **setTranslation(Vector3f trans)**: Translates (moves) an object. Replaces the translate values of this transform with the **x**, **y**, and **z** values in the **trans** argument.

* **setScale(double scale)**: Sets the scale of this transform. Use this function to resize an object.

* **rotX(double angle)**: Sets the rotational component to a counterclockwise rotation around the X axis. Note that calling **rotX()** wipes out non-rotational components of your matrix. The methods **rotY()** and **rotZ()** are also useful. Angles are specified in radians, so use **Math.toRadians(degrees)** to convert from degrees if necessary.

The OrbitBehavior class

The **OrbitBehavior** class allows you to easily move the view around using the mouse. It will translate, rotate, and zoom your scene as the user moves. The following code snippet adds the **OrbitBehavior** to your **ViewingPlatform**. No other code is needed. (We will go over bounding spheres and behaviors later.)

```
ViewingPlatform viewingPlatform = universe.getViewingPlatform();
orbit = new OrbitBehavior(canvas);
BoundingSphere bounds =
    new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 100.0);
orbit.setSchedulingBounds(bounds);
viewingPlatform.setViewPlatformBehavior(orbit);
```

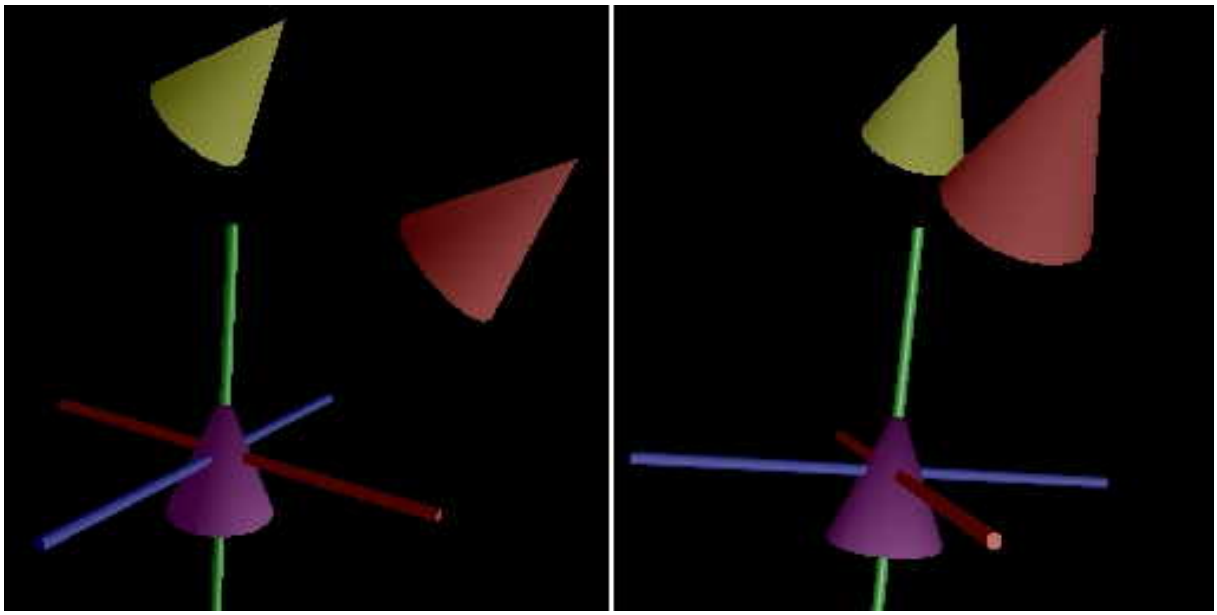
Note that the **OrbitBehavior** is manipulating the view side of the scene graph using the **SimpleUniverse** object. Also notice that most of the Java 3D SDK samples throw in an **OrbitBehavior**, and you should do the same in your programs too -- it will help to be able to move things around and look at them. We will spend the rest of the tutorial back on the content side of the scene graph.

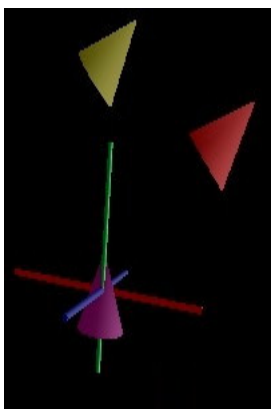
Lost in 3D space

As you build up your scene graph hierarchy, you will use a lot of **`TransformGroup`** nodes. The geometry objects that you want to draw will be located at leaves in the graph. The path of nodes connecting the root of your scene graph to a geometry leaf will generally have several **`TransformGroup`** nodes in it. All of the transformations between the root and a geometry object are applied to that shape, in order, and the ordering of the transformations will affect the final location of your geometry object.

Keep each logical operation you want to do in a seperate **`Transform3D`** object (for example, keep the rotation in one object and a translate in another). Transformations are combined as Java 3D walks down the scene graph during a rendering traversal.

Transformations are not necessarily commutative -- a rotation followed by a translation will have a different result than a translation followed by a rotation. The image below illustrates some transformations on several **Cone** primitives from two different viewpoints (the cylinders represent axes). We'll review the code that generated this image and the scene graph construction of each of these cones in the next panel.





TransformOrder code

Here is a simplified chunk of the TransformOrder.java example. (See

TransformOrder.java on page 29 .) There are two **TransformGroup** nodes and a **Cone**, one attached under the other. The only difference is the order in which they are attached, and the **Cone** color. The **objRotate TransformGroup** is above the **objTranslate** in the red cone, whereas the yellow cone has the translate on top and the rotate under it. This means there is a different ordering in the path between the root node and the cone. Here is code for creating the cones and their transformations.

```
void rotateOnTop(){
      topNode.addChild(objRotate);
      objRotate.addChild(objTranslate);
      objTranslate.addChild(redCone);
}
void translateOnTop(){
      topNode.addChild(objTranslate);
      objTranslate.addChild(objRotate);
      objRotate.addChild(yellowCone);
}
void noTransform(){
      topNode.addChild(purpleCone);
}
```

Transformation key points

* Transformations can be used to specify the location of geometric objects in the scene.
* Use an **OrbitBehavior** with your **SimpleUniverse**.
* Use **Transform3D** helper functions.
* For simplicity, keep each logical operation you want to do in a different **Transform3D** object.

Section 6. Lighting and material properties

The Light class

Lights are used to illuminate the geometry objects in your scene. There are several different light types, which are all subclasses of the abstract **`Light`** class. All lights have a color value, an on/off bit, and a bounding object that describes what areas of the scene it illuminates.

In the real world, the objects around you are lit by several different light sources. The sunlight coming in the window and the overhead light in the room will both illuminate everything in the room. Both lights will affect the color and appearance of objects in the room. In Java 3D you can simulate realistic lighting effects by using multiple light sources.

Light types

An **`AmbientLight`** is everywhere in the scene. It does not originate from a particular point, and it does not point in a particular direction.

A **`PointLight`** radiates from a specified location in all directions, and diminishes with distance. An example of a point light is a desk lamp with no lampshade. A **`SpotLight`** is a type of point light that restricts the light to a cone shape. An example of a spot light is a flashlight.

A **`DirectionalLight`** shines in a particular direction but doesn't emanate from any particular location. All light rays of a directional light travel in parallel. While technically the sun is a point light source, sunlight can be more accurately imitated using a **`DirectionalLight`**.

The teapot in the image has ambient light (you can see that the back side is illuminated slightly), and directional light shining on the front. Both lights affect the final color of each triangle on the surface of the teapot.

Lights in the scene graph

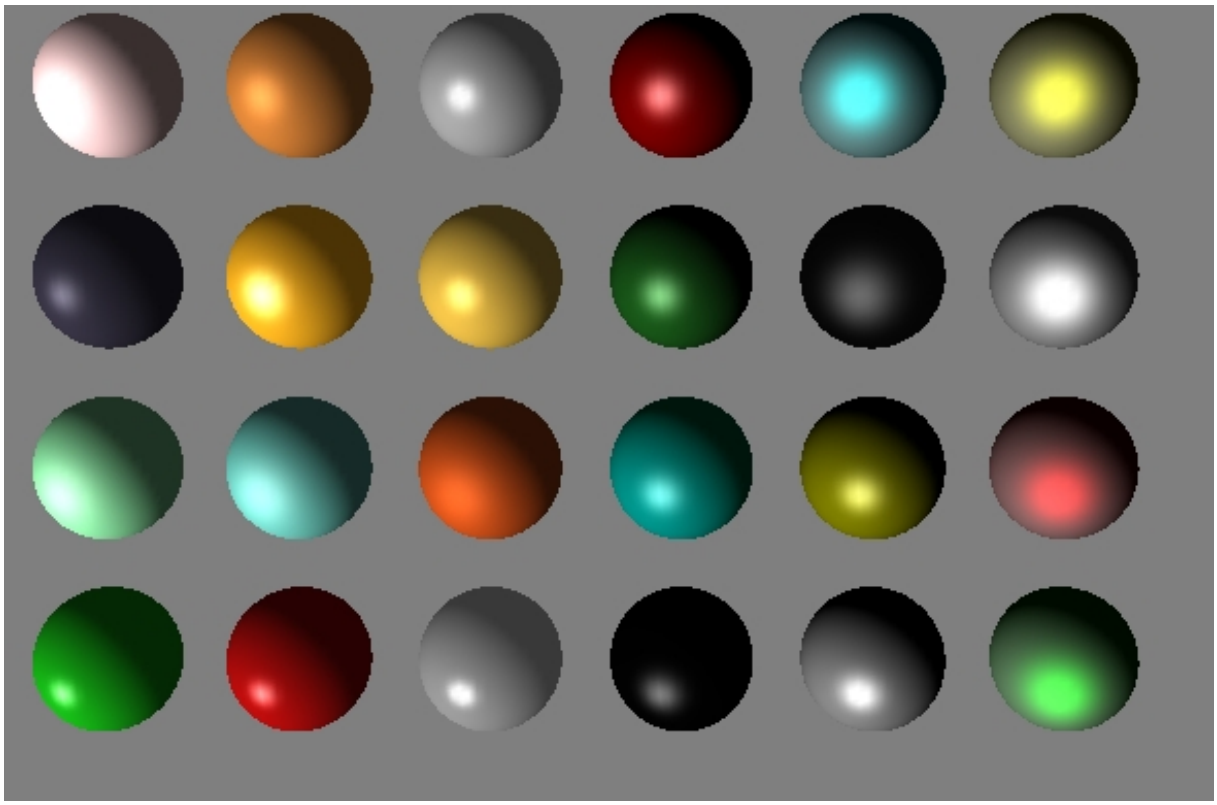
All **`Light`** nodes are leaf nodes in the scene graph. When you create them, you need to specify a **`Bounds`** object; we will be using **`BoundingSphere`**. The light will affect only those geometry objects that are inside the volume defined by the light's **`BoundingSphere`**, so we'll need to make sure the **`BoundingSphere`** is large. After we create the light, we'll attach it to the scene graph at the top **`BranchGroup`**. Lights, behaviors, and textures are all added at the top of a scene graph.

Material properties

Material properties describe how an object reflects light. If your object (**`Primitive`** or **`Shape3D`**) does not have a **`Material`** object in its **`Appearance`** object, it will not be illuminated even though you have a light source specified. You must create a **`Material`** object, enable lighting in the **`Material`** object, and add it to the **`Appearance`** object. The **`Material`** object is one of several different attribute sets that are held in the **`Appearance`** object.

To better understand how material properties affect an object's appearance, think about a shiny ruby gemstone object and a red carpet. While they are both red, they will reflect light differently -- the ruby will have a bright highlight where the light bounces off it, and the carpet will appear to scatter the light. To specify this difference in appearance to Java 3D you'd give the ruby a high **`shininess`** value in the **`Material`** object and the carpet a very low **`shininess`** value.

The image below illustrates a number of spheres with different material properties.



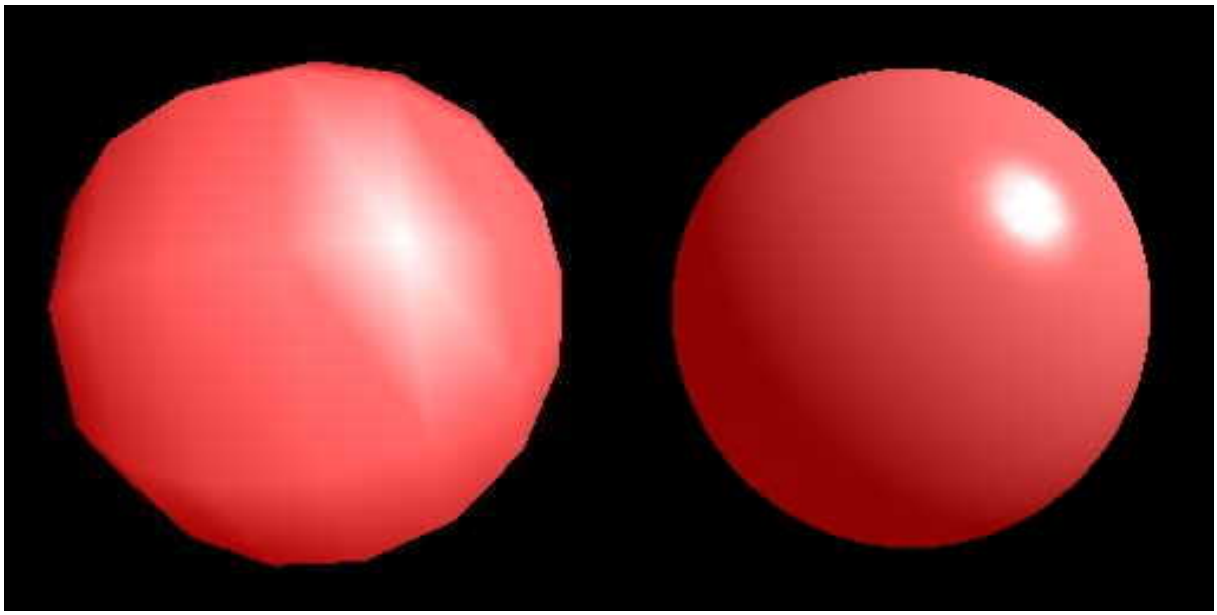
Surface normals

A *surface normal* is a vector that is perpendicular to the surface at the vertex, representing the orientation of the surface at that vertex. The surface normal affects how light is reflected from a surface when calculating lighting effects. The surface normals and position of the viewer determine where the shiny highlight, or *specular reflection*, is located on a sphere.

Fortunately, the **`Primitive`** classes will generate surface normals for you, so we won't discuss normals further in this tutorial.

Calculating lighting effects

Java 3D uses the surface normal to calculate lighting effects. Because the effect is calculated for each vertex, objects with many vertices will look more realistic.



The number of vertices in the **Sphere** object in the image above is controlled by the **divisions** parameter of the **Sphere** object. The coarse sphere uses a value of 9. The smooth sphere uses a value of 30 and thus looks smoother.

Lighting and material key points

* Light location and color, material properties, and surface normals all affect the final color of your lit objects.

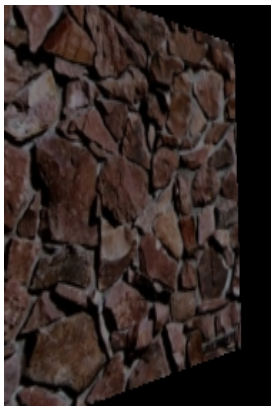
* The **Appearance** class contains a **Material** object, which holds the lighting state for your geometry. Remember to enable lighting in your **Material** object.

* The **Material** class defines how your geometry reflects light. It includes ambient, diffuse, and specular colors, and shininess.

* Ambient, point, and directional lights provide different lighting effects. Always throw in an ambient light just to make sure you have some light in your scene.

* Normal values determine a surface's plane orientation and are generated for you when using the **Primitive** classes.

Section 7. Texture mapping



Texture mapping

Texture mapping increases realism in your scene by adding additional visual detail to the surfaces of your objects.

Suppose you want to have a stone wall in your scene. You could define geometry for each stone, and geometry for each piece of mortar, and render all the objects to simulate the stone wall. That would be a lot of work. An alternative is to use texture mapping. In this case, you would use a single rectangle object to represent the whole wall, and paste an image of a stone wall onto the rectangle (such as the stone.jpg image that ships with Java 3D).

To use texture mapping, you need to specify the image, where to paste it on the object, and what to do when the image doesn't fit quite right, such as when applying a rectangular bitmap to a non-rectangular polygon. We'll work through these tasks in the remainder of this section.

The technology advances in texture mapping hardware in recent years have been fast and furious, resulting in a subject of considerable complexity, which is beyond the scope of this tutorial. If you're interested in the more advanced details of texture mapping, I recommend taking the Texturing tutorial on Sun's Java 3D web site (see Resources on page 26).

Loading a texture

Like lighting, texture mapping affects the entire geometry object. We will be using the **`Appearance`** class again to specify the texture mapping effect.

Java 3D has simplified the process of loading texture images. The **`TextureLoader`** class is in the Java 3D utility classes:

```
TextureLoader texLoader = new TextureLoader(url,imageobserver)
appearance.setTexture(texLoader.getTexture());
```

Always set the width and height of your texture image to a power of 2 for a realistic look. Other values will cause **`TextureLoader`** to squish the texture.

If the image file has only RGB values, and no alpha value, then use:

```
TextureLoader texLoader = new TextureLoader(url,new String("RGB"), imageobserver)
```

Pasting on the image

Now that we have a texture map loaded, we need to specify how to paste it on our object. We do this by using *texture coordinates* for each vertex. Texture coordinates define which chunk of the texture image is used for each polygon of your object. A useful analogy is to think about how you would place wrapping paper on a present.

Fortunately, the **`Primitive`** classes will generate texture coordinates for you -- just as they generate normals for calculating lighting.

If you are not using the **`Primitive`** classes, you need to define texture coordinates yourself. To do this, you specify values between **`0.0`** and **`1.0`** in both the **`x`** and **`y`** direction. Using our stone wall example, we would need to assign texture coordinates for each of the corners of the wall. Assuming we want to use the whole image of the stone on our wall, we would need to assign one for each of the four corners: **`(0,0)`**, **`(1,0)`**, **`(1,1)`** and **`(0,1)`**. This would stretch the whole stone image across the one rectangle that makes up the wall.

Shrinking and stretching a texture

Texture mapping gets complicated when you need to apply a rectangular texture to a non-rectangular region. Shrinking and stretching a texture as it is pasted on an object is called *filtering*.

Texture mapping will look more realistic if you provide several different sized representations (*minimaps*) of your texture for Java 3D to use when filtering.

There are several different ways to filter your texture. You choose the type you want based on your graphics card's performance and the look you want. You'll need to do some experimenting to determine what looks best.

Texture mapping example

Take a look at Wallpaper.java on page 32 , which uses texture-mapped**`Primitive`**s. Remember that the **`Primitive`** classes will generate the texture coordinates that define where the image is pasted on the object. Our example uses the **`TextureLoader`** utility to read in a jpeg image as well as to generate the filtered images used in mip-mapping. Note how the texture is squeezed into the poles of the sphere, and the tip of the cone. In each cone/sphere pair a different texture filter is used. You can notice filtering differences on the border between white and red stripes.

To have the texture loader generate prefiltered images for you, specify **`GENERATE_MIPMAP`** when constructing a **`TextureLoader`**. You specify your mipmap filtering function in your **`Texture`** object (stored in the **`Appearance`** object.)

Texture mapping key points

* Texture mapping is the process of gluing an image onto a geometry object. The image orientation and placement on the geometry object is controlled using texture coordinate data for each vertex.

* Use the **`TextureLoader`** utility to read in the image file, and specify the **`TextureLoader.GENERATE_MIPMAP`** flag in the constructor.

* Let the **`Primitive`** classes generate texture coordinates for you, instead of specifying them yourself. Use the **`Primitive.GENERATE_TEXTURE_COORDS`** when constructing a **`Primitive`**.

* Texture mapping is a complex subject. Try not to get buried in all the different options. Use the defaults when you're starting out.

Section 8. Behaviors and interpolators

The Behavior class

Behaviors allow you to animate scene elements and interact with the scene graph. The **`Behavior`** class provides a mechanism for your code to modify the scene graph. A **`Behavior`** is placed in the scene graph and is triggered by a *stimulus*, such as a mouse movement.

Like lights, behaviors have bounds and capability bits, and must be attached to the scene graph. So if your behavior doesn't work, make sure the bounds are large enough, that the behavior is enabled, and that you have attached it in the scene graph.

Interpolators provide an easy way to add some simple animations to your scene graph. Interpolators can be fun to play around with, and we will be using them in our examples. But first we'll look at some of the basics of **`Behavior`** handling.

Wakeup call

You specify the conditions that will trigger your behavior with a **`WakeupCriterion`**. Examples of wakeup criteria include a key press, mouse event, timer, and elapsed number of frames. You specify the criteria when you initially create the behavior and each time you handle a behavior message. **`OrbitBehavior`** is an example of a **`Behavior`** that wakes up on mouse events.

Using **`WakeupOnElapsedTime`**, you can change a **`TranformGroup`**'s **`Transform3D`** to animate a portion of your scene graph.

The processStimulus() method

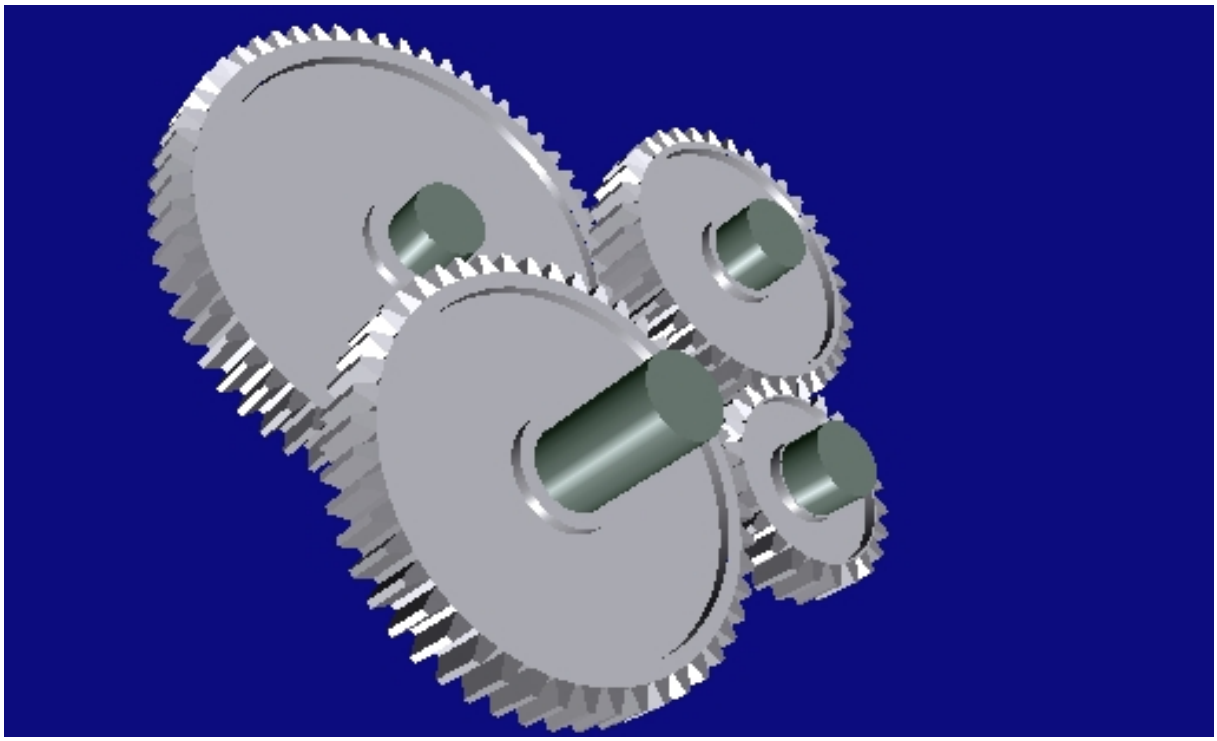
When the wakeup event occurs, the behavior's **`processStimulus()`** method is called. You can do just about anything in the **`processStimulus()`** method -- you can add or delete objects from the scene, change transformations, change an object's appearance, detect when two objects collide with each other, or whatever else you need to do.

Keep in mind that anything you want to change in your scene graph will need to have its related capability bit set. For example, to change a **`TransformGroup`** node in the **`processStimulus()`** method, you will need to set the **`TransformGroup.ALLOW_TRANSFORM_WRITE`** bit.

Interpolators

Interpolators are built-in goodies that can help you add simple animations to your scene graph. They smoothly transition, or interpolate, between a range of values that you define.

The GearBox sample that ships with the Java 3D SDK uses several rotation interpolators to animate these gears shown in the image below.



Each gear is hooked up to a **RotationInterpolator** that gets triggered repeatedly after a certain amount of elapsed time. The **RotationInterpolator** applies a rotation to the **TransformGroup** that contains the gear geometry object, causing the gear to rotate.

It's all in the timing

The **Alpha** object generates a value between 0.0 and 1.0 as a function of time. (Note that the **Alpha** object is not related to a vertex's alpha value, which specifies its transparency.)

Interpolators use the **Alpha** object to drive their changes to the scene graph. The value of an **Alpha** object can change over time, depending on how you program it. Our gears spin continuously at a constant speed. Each time the **RotationInterpolator** gets triggered by the **Alpha** object, it uses the generated alpha value to compute the new rotation (that is, to make the gears rotate.) You create a simple **Alpha** that will indefinitely loop at a constant speed using the following code:

```
Alpha(int loopCount, long speed)
```

There are plenty of options for changing the behavior of the **Alpha** object. You can have it accelerate, stop for a period of time, and then slowly go back down.

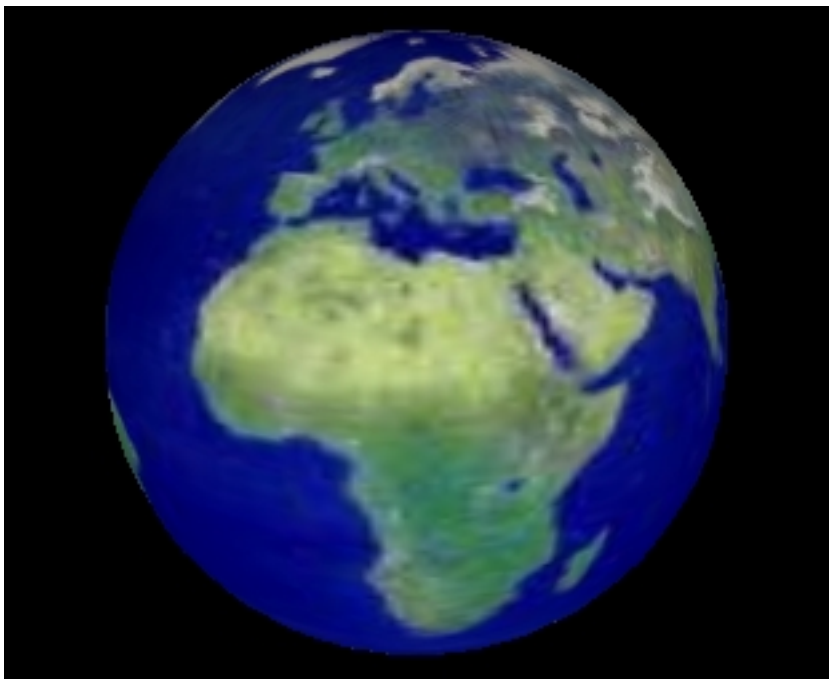
Rotation example

SupermanInterp.java on page 34 creates a**Sphere** and then applies a map texture to it. (The earth looks a little fuzzy because the map used for the texture did not have a lot of detail.) I set up a **TransformGroup** in a **RotationInterpolator**, and then attach the textured **Sphere** below the **TransformGroup**. As the **Interpolator** changes the **TransformGroup**, the earth spins. Because **Interpolator**s change values in the scene

graph after the rendering loop starts, we must set one of the capability bits:

```
spinGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE)
```

In this example I've configured the **Alpha** object to accelerate the spin clockwise, and then reverse it to spin counterclockwise.



Color interpolator

For ColorInterp.java on page 37 , I've added code to attach a color interpolator to the spinning globe. Instead of affecting a **TransformGroup**, it affects the diffuse color in a **Material** object. Because the **Material** object will be changing after the render loop starts, I set its capability bit.

```
material.setCapability(Material.ALLOW_COMPONENT_WRITE);
Alpha colorAlpha = new Alpha(-1, 2000);
Color3f endColor = new Color3f(1.0f,1.0f,1.0f);
Color3f startColor = new Color3f(0.0f,0.0f,0.0f);
ColorInterpolator colorInterp =
   new ColorInterpolator(colorAlpha, mat,startColor,endColor);
colorInterp.setSchedulingBounds(bounds);
```

Section 9. Java 3D wrapup

Summary

That completes our introduction to Java 3D. We covered a lot of ground in a very short time, so let's review the high points:

* Java 3D allows you to develop 3D graphics applications that have a high degree of visual realism. You build up a scene graph that describes everything you want to render, including geometry objects and visual effects.

* The built-in set of **`Primitive`** classes allow you to quickly build up a scene graph full of geometry objects without worrying about some of the more complicated details like texture coordinates and surface normals.

* You describe lighting of your scene with the **`Light`** classes and the **`Material`**.

* You can provide additional realism using texture mapping.

* Interpolators are a type of **`Behavior`** that allow you to add animations to your scene.

The fun of graphics programming in general and Java 3D in particular is hard to convey with words. It's the hands-on experimentation that will hook you and keep you up late at night. I recommend that you download the example code in Resources on page 26 and try out different techniques to see what works and how. Use different lighting and play around with textures. Just have fun!

Resources

* Download the source code samples used in this tutorial (java3dsource.zip).

* Visit Sun's *Java 3D Web site*, which includes lots of good information, including some excellent *tutorials*.

* The *Java 3D community site* offers comprehensive resources and includes a great FAQ that addresses common problems you may encounter as you get up to speed with this complex API.

* *NCSA Portfolio* is a collection of utility objects to use with your Java 3D programs. It comes with full documentation for the objects in the library and example source code that shows how to use the objects.

* Visit the *Web3D Consortium* Web site for details on VRML.

* *Java 3D for AIX*, now included in the *IBM Java Developer Kit for AIX*, is a scene graph API for 3D graphics that extends the core Java API and allows you to easily incorporate high-quality, scalable, platform-independent, three-dimensional graphics into Java-based applications and applets.

* IBM research has several ongoing and very interesting projects in the world of *graphics and visualization*.

* If you're interested in expanding your graphics programming universe beyond Java 3D, visit the *OpenGL home page* for information on this robust library.

Feedback

We welcome your feedback on this tutorial, and look forward to hearing from you.

Section 10. Appendix: The source

UglyCube.java

```
/*
 *
 * Copyright (c) 1996-2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
 * modify and redistribute this software in source and binary code form,
 * provided that i) this copyright notice and license appear on all copies of
 * the software; and ii) Licensee does not utilize the software in a manner
 * which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind. ALL
 * EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY
 * IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR
 * NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE
 * LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
 * OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS
 * LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER
 * CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF
 * OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control of
 * aircraft, air traffic, aircraft navigation or aircraft communications; or in
 * the design, construction, operation or maintenance of any nuclear
 * facility. Licensee represents and warrants that it will not use or
 * redistribute the Software for such purposes.
 */
import java.applet.Applet;
import java.awt.BorderLayout;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;
//Bare minimum really boring cube.
// Shows bare bones Universe creation
public class UglyCube extends Applet {
    private SimpleUniverse universe ;
    public UglyCube() {
        }
    public void init() {
      //canvas to draw on, ask SimpleUniverse what config to use
      Canvas3D canvas = new Canvas3D(
          SimpleUniverse.getPreferredConfiguration());
      setLayout(new BorderLayout());
      add("Center", canvas);
      //create top of our scene graph
      BranchGroup scene = new BranchGroup();
      //attach the cube to it
      scene.addChild(new ColorCube(0.4));
      //create universe, and attach our geometry to it.
      SimpleUniverse u = new SimpleUniverse(canvas);
      u.getViewingPlatform().setNominalViewingTransform();
        //rendering starts after BranchGroup is attached.
      u.addBranchGraph(scene);
    }
    // The following allows UglyCube to be run as an application
    // as well as an applet
```

```
    public static void main(String[] args) {
        new MainFrame(new UglyCube(), 256, 256);
    }
}
```

TransformOrder.java

```
/*
 *
 * Copyright (c) 1996-2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
 * modify and redistribute this software in source and binary code form,
 * provided that i) this copyright notice and license appear on all copies of
 * the software; and ii) Licensee does not utilize the software in a manner
 * which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind. ALL
 * EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY
 * IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR
 * NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE
 * LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
 * OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS
 * LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER
 * CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF
 * OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control of
 * aircraft, air traffic, aircraft navigation or aircraft communications; or in
 * the design, construction, operation or maintenance of any nuclear
 * facility. Licensee represents and warrants that it will not use or
 * redistribute the Software for such purposes.
 */
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.awt.*;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.behaviors.vp.*;
/**
   Generates a scene graph with the cylinders that
   make up the axis, and 3 cones.
   Add one Cone with rotate before translate, and
   another that has translate before rotate.
   For reference, throw in a cone without translate
   or rotate
*/
public class TransformOrder extends Applet {
    public static final int X =1;
    public static final int Y =2;
    public static final int Z =3;
    public static final int ROTATE_TOP    =4;
    public static final int TRANSLATE_TOP =5;
    public static final int NO_TRANSFORM  =6;
    private SimpleUniverse universe ;
```

```
private BranchGroup scene;
private Canvas3D canvas;
private BoundingSphere bounds =
        new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 1000.0);
private Appearance red = new Appearance();
private Appearance yellow = new Appearance();
private Appearance purple = new Appearance();
Transform3D rotate = new Transform3D();
Transform3D translate = new Transform3D();
public void setupView() {
    /** Add some view related things to view branch side
    of scene graph */
    // add mouse interaction to the ViewingPlatform
    OrbitBehavior orbit = new OrbitBehavior(canvas,
            OrbitBehavior.REVERSE_ALL|OrbitBehavior.STOP_ZOOM);
    orbit.setSchedulingBounds(bounds);
    ViewingPlatform viewingPlatform = universe.getViewingPlatform();
    // This will move the ViewPlatform back a bit so the
    // objects in the scene can be viewed.
    viewingPlatform.setNominalViewingTransform();
    viewingPlatform.setViewPlatformBehavior(orbit);
    }
//construct each branch of the graph, changing the order children added
// since  Group node can only have one parent, have to construct
// new translate and rotate group nodes for each branch.
Group rotateOnTop(){
      Group root=new Group();
      TransformGroup objRotate = new TransformGroup(rotate);
      TransformGroup objTranslate = new TransformGroup(translate);
      Cone redCone=
          new Cone(.3f, 0.7f, Primitive.GENERATE_NORMALS, red);
      root.addChild(objRotate);
      objRotate.addChild(objTranslate);
      objTranslate.addChild(redCone); //tack on red cone
      return root;
}
Group translateOnTop(){
      Group root=new Group();
      TransformGroup objRotate = new TransformGroup(rotate);
      TransformGroup objTranslate = new TransformGroup(translate);
      Cone yellowCone=
          new Cone(.3f, 0.7f, Primitive.GENERATE_NORMALS, yellow);
      root.addChild(objTranslate);
      objTranslate.addChild(objRotate);
      objRotate.addChild(yellowCone); //tack on yellow cone
      return root;
}
Group noTransform(){
      Cone purpleCone=
          new Cone(.3f, 0.7f, Primitive.GENERATE_NORMALS, purple);
      return purpleCone;
}
/** Represent an axis using cylinder Primitive. Cylinder is
    aligned with Y axis, so we have to rotate it when
    creating X and Z axis
*/
public TransformGroup createAxis(int type) {
    //appearance and lightingProps are used in
    //lighting. Each axis a different color
    Appearance appearance = new Appearance();
    Material lightingProps = new Material();
    Transform3D t = new Transform3D();
    switch (type) {
       case Z:
          t.rotX(Math.toRadians(90.0));
```

```
            lightingProps.setAmbientColor(1.0f,0.0f,0.0f);
         break;
         case Y:
            // no rotation needed, cylinder aligned with Y already
            lightingProps.setAmbientColor(0.0f,1.0f,0.0f);
         break;
         case X:
            t.rotZ(Math.toRadians(90.0));
            lightingProps.setAmbientColor(0.0f,0.0f,1.0f);
         break;
         default:
         break;
      }
      appearance.setMaterial(lightingProps);
      TransformGroup objTrans = new TransformGroup(t);
      objTrans.addChild( new Cylinder(.03f,2.5f,Primitive.GENERATE_NORMALS,appearance
      return objTrans;
   }
   /** Create X, Y , and Z axis, and 3 cones. Throws in
      some quick lighting to help viewing the scene
   */
   public BranchGroup createSceneGraph() {
      // Create the root of the branch graph
      BranchGroup objRoot = new BranchGroup();
      //45 degree rotation around the X axis
      rotate.rotX(Math.toRadians(45.0));
      //translation up the Y axis
      translate.setTranslation(new Vector3f(0.0f,2.0f,1.0f)); //SCD 0.0f));
      //Material objects are related to lighting, we'll cover
      //that later
      Material redProps = new Material();
      redProps.setAmbientColor(1.0f,0.0f,0.0f); //red cone
      red.setMaterial(redProps);
      Material yellowProps = new Material();
      yellowProps.setAmbientColor(1.0f,1.0f,0.0f); //yellow cone
      yellow.setMaterial(yellowProps);
      Material purpleProps = new Material();
      purpleProps.setAmbientColor(0.8f,0.0f,0.8f); //purple cone
      purple.setMaterial(purpleProps);
      // Create a x,y,z axis, and then 3 cone branches
      objRoot.addChild(createAxis(X));
      objRoot.addChild(createAxis(Y));
      objRoot.addChild(createAxis(Z));
      objRoot.addChild(noTransform());    //purple cone
      objRoot.addChild(rotateOnTop());      //red cone
      objRoot.addChild(translateOnTop());   //yellow cone
      //throw in some light so we aren't stumbling
      //around in the dark
      Color3f lightColor = new Color3f(.3f,.3f,.3f);
      AmbientLight ambientLight= new AmbientLight(lightColor);
      ambientLight.setInfluencingBounds(bounds);
      objRoot.addChild(ambientLight);
      DirectionalLight directionalLight = new DirectionalLight();
      directionalLight.setColor(lightColor);
      directionalLight.setInfluencingBounds(bounds);
      objRoot.addChild(directionalLight);
      return objRoot;
   }
   public TransformOrder() {
   }
   public void init() {
      BranchGroup scene = createSceneGraph();
      setLayout(new BorderLayout());
      GraphicsConfiguration config =
         SimpleUniverse.getPreferredConfiguration();
```

```
        canvas = new Canvas3D(config);
        add("Center", canvas);
        // Create a simple scene and attach it to the virtual universe
        universe = new SimpleUniverse(canvas);
        setupView();
        universe.addBranchGraph(scene);
    }
    public void destroy() {
        universe.removeAllLocales();
    }
    //
    // The following allows TransformOrder to be run as an application
    // as well as an applet
    //
    public static void main(String[] args) {
        new MainFrame(new TransformOrder(), 256, 256);
    }
}
```

Wallpaper.java

```
/*
 *
 * Copyright (c) 1996-2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
 * modify and redistribute this software in source and binary code form,
 * provided that i) this copyright notice and license appear on all copies of
 * the software; and ii) Licensee does not utilize the software in a manner
 * which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind. ALL
 * EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY
 * IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR
 * NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE
 * LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
 * OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS
 * LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER
 * CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF
 * OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control of
 * aircraft, air traffic, aircraft navigation or aircraft communications; or in
 * the design, construction, operation or maintenance of any nuclear
 * facility. Licensee represents and warrants that it will not use or
 * redistribute the Software for such purposes.
 */
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.awt.*;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.image.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.behaviors.vp.*;
/**
```

```
   Simple Texture Mapping example
*/
public class Wallpaper extends Applet {
    private SimpleUniverse universe ;
    private BranchGroup scene;
    private Canvas3D canvas;
    private BoundingSphere bounds =
            new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 1000.0);
    private static java.net.URL texImage = null;
    public Group  createGeometry(int filter, float y, java.net.URL texImage) {
        /**
        Create some Texture mapped objects
        */
        Appearance appearance = new Appearance();
        TextureLoader tex = new TextureLoader(texImage, TextureLoader.GENERATE_MIPMAP ,
        Texture texture = tex.getTexture();
        texture.setMinFilter(filter) ;
        appearance.setTexture(texture);
        TextureAttributes texAttr = new TextureAttributes();
        texAttr.setTextureMode(TextureAttributes.MODULATE);
        appearance.setTextureAttributes(texAttr);
        Color3f black = new Color3f(0.0f, 0.0f, 0.0f);
        Color3f white = new Color3f(1.0f, 1.0f, 1.0f);
        // Set up the material properties
        appearance.setMaterial(new Material(white, black, white, black, 1.0f));
        //use to build tree hierarchy
        Group topNode = new Group();
        Transform3D translate = new Transform3D();
        translate.setTranslation(new Vector3f(.5f,y,-0.5f));
        TransformGroup gimmeSpace = new TransformGroup(translate);
        Cone cone =  new Cone(.4f,0.8f,Primitive.GENERATE_NORMALS|Primitive.GENERATE_TE
        gimmeSpace.addChild(cone);
        topNode.addChild(gimmeSpace); //cone at bottom
        translate = new Transform3D();
        translate.setTranslation(new Vector3f(-0.5f,y,-0.5f));
        gimmeSpace = new TransformGroup(translate);
        Sphere sphere = new Sphere(.4f,Primitive.GENERATE_NORMALS|Primitive.GENERATE_TE
        gimmeSpace.addChild(sphere);
        topNode.addChild(gimmeSpace); //cone at bottom
        return topNode;
    }
    public void setupView() {
        /** Add some view related things to view branch side
        of scene graph */
        // add mouse interaction to the ViewingPlatform
        OrbitBehavior orbit = new OrbitBehavior(canvas,
                OrbitBehavior.REVERSE_ALL|OrbitBehavior.STOP_ZOOM);
        orbit.setSchedulingBounds(bounds);
        ViewingPlatform viewingPlatform = universe.getViewingPlatform();
        // This will move the ViewPlatform back a bit so the
        // objects in the scene can be viewed.
        viewingPlatform.setNominalViewingTransform();
        viewingPlatform.setViewPlatformBehavior(orbit);
        }
    public BranchGroup createSceneGraph() {
        // Create the root of the branch graph
        BranchGroup objRoot = new BranchGroup();
        // Create a simple Shape3D node; add it to the scene graph.
        // Set up the texture map
        // the path to the image
        objRoot.addChild(createGeometry( Texture.BASE_LEVEL_POINT,1.0f,texImage));
        objRoot.addChild(createGeometry( Texture.MULTI_LEVEL_POINT,0.0f,texImage));
        objRoot.addChild(createGeometry( Texture.MULTI_LEVEL_LINEAR,-1.0f,texImage));
        //throw in some light so we aren't stumbling
        //around in the dark
```

```
        Color3f lightColor = new Color3f(.5f,.5f,.5f);
        AmbientLight ambientLight= new AmbientLight(lightColor);
        ambientLight.setInfluencingBounds(bounds);
        objRoot.addChild(ambientLight);
        DirectionalLight directionalLight = new DirectionalLight();
        directionalLight.setColor(lightColor);
        directionalLight.setInfluencingBounds(bounds);
        objRoot.addChild(directionalLight);
        return objRoot;
    }
    public Wallpaper() {
    }
    public void init() {
        BranchGroup scene = createSceneGraph();
        setLayout(new BorderLayout());
        GraphicsConfiguration config =
           SimpleUniverse.getPreferredConfiguration();
        canvas = new Canvas3D(config);
        add("Center", canvas);
        // Create a simple scene and attach it to the virtual universe
        universe = new SimpleUniverse(canvas);
        setupView();
        universe.addBranchGraph(scene);
    }
    public void destroy() {
        universe.removeAllLocales();
    }
    //
    // The following allows Wallpaper to be run as an application
    // as well as an applet
    //
    public static void main(String[] args) {
        try{
           if (args.length == 0) {
             texImage = new java.net.URL("file:./images/speedchase.jpg");
           } else {
             texImage = new java.net.URL(args[0]);
            }
        }
        catch (java.net.MalformedURLException ex) {
            System.out.println(ex.getMessage());
            System.exit(1);
        }
        new MainFrame(new Wallpaper(), 256, 256);
    }
}
```

SupermanInterp.java

```
/*
 *
 * Copyright (c) 1996-2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
 * modify and redistribute this software in source and binary code form,
 * provided that i) this copyright notice and license appear on all copies of
 * the software; and ii) Licensee does not utilize the software in a manner
 * which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind. ALL
 * EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY
 * IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR
```

```
 * NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE
 * LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
 * OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS
 * LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER
 * CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF
 * OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control of
 * aircraft, air traffic, aircraft navigation or aircraft communications; or in
 * the design, construction, operation or maintenance of any nuclear
 * facility. Licensee represents and warrants that it will not use or
 * redistribute the Software for such purposes.
 */
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.awt.*;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.image.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.behaviors.vp.*;
/**
   Rotation Interpolator example
   Very similar to texture mapping example, but
   attaches an Interpolator above geometry that
   keeps the world spinning. Play with Alpha timing,
   you can have it slowly ease to a halt by using
   some of the other parameters that aren't in this
   simple example. It will also reverse direction.
*/
public class SupermanInterp extends Applet {
    private SimpleUniverse universe ;
    private BranchGroup scene;
    private Canvas3D canvas;
    private BoundingSphere bounds =
            new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 1000.0);
    public Primitive  createGeometry(int filter,  java.net.URL texImage, Appearance app
        /**
        Create Sphere and texture it
        */
        TextureLoader tex =
            new TextureLoader(texImage, TextureLoader.GENERATE_MIPMAP , this);
        Texture texture = tex.getTexture();
        texture.setMinFilter(filter) ;
        appearance.setTexture(texture);
        TextureAttributes texAttr = new TextureAttributes();
        texAttr.setTextureMode(TextureAttributes.MODULATE);
        appearance.setTextureAttributes(texAttr);
        Color3f black = new Color3f(0.0f, 0.0f, 0.0f);
        Color3f white = new Color3f(1.0f, 1.0f, 1.0f);
        // Set up the material properties
        appearance.setMaterial(new Material(white, black, white, black, 1.0f));
        Sphere sphere =
            new Sphere(.4f,Primitive.GENERATE_NORMALS|
                           Primitive.GENERATE_TEXTURE_COORDS,appearance);
        return sphere;
    }
    public void setupView() {
        /** Add some view related things to view branch side
```

```
        of scene graph */
        // add mouse interaction to the ViewingPlatform
        OrbitBehavior orbit = new OrbitBehavior(canvas,
                OrbitBehavior.REVERSE_ALL|OrbitBehavior.STOP_ZOOM);
        orbit.setSchedulingBounds(bounds);
        ViewingPlatform viewingPlatform = universe.getViewingPlatform();
        // This will move the ViewPlatform back a bit so the
        // objects in the scene can be viewed.
        viewingPlatform.setNominalViewingTransform();
        viewingPlatform.setViewPlatformBehavior(orbit);
        }
    public BranchGroup createSceneGraph() {
        // Create the root of the branch graph
        BranchGroup objRoot = new BranchGroup();
        // Create a simple Shape3D node; add it to the scene graph.
        // Set up the texture map
        java.net.URL texImage = null;
        // the path to the image
        try {
            texImage = new java.net.URL("file:../images/earth.jpg");
        }
        catch (java.net.MalformedURLException ex) {
            System.out.println(ex.getMessage());
            System.exit(1);
        }
        Appearance app= new Appearance();
        Primitive geo = createGeometry( Texture.MULTI_LEVEL_LINEAR,texImage,app);
        //spinGroup will be hooked into the interpolator
        TransformGroup spinGroup = new TransformGroup();
        spinGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        spinGroup.addChild(geo);
        // Create a new Behavior object that will perform the
        // desired operation on the specified transform and add
        // it into the scene graph.
        //OLD:  straight constant spin
        //      Alpha rotationAlpha = new Alpha(-1, 4000);
        //NEW:  accelerate one direction, stop, rotate opposite direction
        Alpha rotationAlpha = new Alpha(-1, Alpha.INCREASING_ENABLE |
                                        Alpha.DECREASING_ENABLE,
                                        0, 0,
                                        5000, 2500, 200,
                                        5000, 2500, 200);
        RotationInterpolator rotator =
            new RotationInterpolator(rotationAlpha, spinGroup);
        rotator.setSchedulingBounds(bounds);
        //throw in some light so we aren't stumbling
        //around in the dark
        Color3f lightColor = new Color3f(.5f,.5f,.5f);
        AmbientLight ambientLight= new AmbientLight(lightColor);
        ambientLight.setInfluencingBounds(bounds);
        DirectionalLight directionalLight = new DirectionalLight();
        directionalLight.setColor(lightColor);
        directionalLight.setInfluencingBounds(bounds);
        objRoot.addChild(rotator); //behavior gets attached at the top
        objRoot.addChild(spinGroup); //TransformGroup and sphere
        objRoot.addChild(directionalLight);
        objRoot.addChild(ambientLight);
        return objRoot;
    }
    public SupermanInterp() {
    }
    public void init() {
        BranchGroup scene = createSceneGraph();
        setLayout(new BorderLayout());
        GraphicsConfiguration config =
```

```
            SimpleUniverse.getPreferredConfiguration();
        canvas = new Canvas3D(config);
        add("Center", canvas);
        // Create a simple scene and attach it to the virtual universe
        universe = new SimpleUniverse(canvas);
        setupView();
        universe.addBranchGraph(scene);
    }
    public void destroy() {
        universe.removeAllLocales();
    }
    //
    // The following allows SupermanInterp to be run as an application
    // as well as an applet
    //
    public static void main(String[] args) {
        new MainFrame(new SupermanInterp(), 256, 256);
    }
}
```

ColorInterp.java

```
/*
 *
 * Copyright (c) 1996-2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
 * modify and redistribute this software in source and binary code form,
 * provided that i) this copyright notice and license appear on all copies of
 * the software; and ii) Licensee does not utilize the software in a manner
 * which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind. ALL
 * EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY
 * IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR
 * NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE
 * LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
 * OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS
 * LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER
 * CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF
 * OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control of
 * aircraft, air traffic, aircraft navigation or aircraft communications; or in
 * the design, construction, operation or maintenance of any nuclear
 * facility. Licensee represents and warrants that it will not use or
 * redistribute the Software for such purposes.
 */
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.awt.*;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.image.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.behaviors.vp.*;
```

```
/**
   Color Interpolator example
   Very similar to texture mapping example,
   will interpolate the color values of the
   earth.
*/
public class ColorInterp extends Applet {
    private SimpleUniverse universe ;
    private BranchGroup scene;
    private Canvas3D canvas;
    private BoundingSphere bounds =
            new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 1000.0);
    public Primitive  createGeometry(int filter,  java.net.URL texImage, Appearance app
        /**
        Create Sphere and texture it
        */
        TextureLoader tex =
            new TextureLoader(texImage, TextureLoader.GENERATE_MIPMAP , this);
        Texture texture = tex.getTexture();
        texture.setMinFilter(filter) ;
        appearance.setTexture(texture);
        TextureAttributes texAttr = new TextureAttributes();
        texAttr.setTextureMode(TextureAttributes.MODULATE);
        appearance.setTextureAttributes(texAttr);
        Color3f black = new Color3f(0.0f, 0.0f, 0.0f);
        Color3f white = new Color3f(1.0f, 1.0f, 1.0f);
        Color3f gray = new Color3f(0.3f, 0.3f, 0.3f);
        Color3f ltgray = new Color3f(0.6f, 0.6f, 0.6f);
        // Set up the material properties
        appearance.setMaterial(new Material(white, black, ltgray, ltgray, 32.0f));
        Sphere sphere =
            new Sphere(.4f,Primitive.GENERATE_NORMALS|
                           Primitive.GENERATE_TEXTURE_COORDS,appearance);
        return sphere;
    }
    public void setupView() {
        /** Add some view related things to view branch side
        of scene graph */
        // add mouse interaction to the ViewingPlatform
        OrbitBehavior orbit = new OrbitBehavior(canvas,
                OrbitBehavior.REVERSE_ALL|OrbitBehavior.STOP_ZOOM);
        orbit.setSchedulingBounds(bounds);
        ViewingPlatform viewingPlatform = universe.getViewingPlatform();
        // This will move the ViewPlatform back a bit so the
        // objects in the scene can be viewed.
        viewingPlatform.setNominalViewingTransform();
        viewingPlatform.setViewPlatformBehavior(orbit);
        }
    public BranchGroup createSceneGraph() {
        // Create the root of the branch graph
        BranchGroup objRoot = new BranchGroup();
        // Create a simple Shape3D node; add it to the scene graph.
        // Set up the texture map
        java.net.URL texImage = null;
        // the path to the image
        try {
            texImage = new java.net.URL("file:../images/earth.jpg");
        }
        catch (java.net.MalformedURLException ex) {
            System.out.println(ex.getMessage());
            System.exit(1);
        }
        Appearance app= new Appearance();
        Primitive geo = createGeometry( Texture.MULTI_LEVEL_LINEAR,texImage,app);
        //spinGroup will be hooked into the interpolator
```

```
        TransformGroup spinGroup = new TransformGroup();
        spinGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        spinGroup.addChild(geo);
        // Create a new Behavior object that will perform the
        // desired operation on the specified transform and add
        // it into the scene graph.
        Alpha rotationAlpha = new Alpha(-1, 4000);
        RotationInterpolator rotator =
            new RotationInterpolator(rotationAlpha, spinGroup);
        rotator.setSchedulingBounds(bounds);
        //we'll need the Material to Interpolate the diffuse color
        //set capability bit to allow interpolator to change at render time
        Material mat = app.getMaterial();
        mat.setCapability(Material.ALLOW_COMPONENT_WRITE);
        Alpha colorAlpha = new Alpha(-1, 2000);
        //We interpolate from black to white, looping indefinitely
        Color3f endColor = new Color3f(1.0f,1.0f,1.0f);
        Color3f startColor = new Color3f(0.0f,0.0f,0.0f);
        ColorInterpolator colorInterp =
            new ColorInterpolator(colorAlpha, mat,startColor,endColor);
        colorInterp.setSchedulingBounds(bounds);
        //throw in some light so we aren't stumbling
        //around in the dark
        Color3f lightColor = new Color3f(.5f,.5f,.5f);
        AmbientLight ambientLight= new AmbientLight(lightColor);
        ambientLight.setInfluencingBounds(bounds);
        DirectionalLight directionalLight = new DirectionalLight();
        directionalLight.setColor(lightColor);
        directionalLight.setInfluencingBounds(bounds);
        objRoot.addChild(rotator); //behavior gets attached at the top
        objRoot.addChild(colorInterp); //behavior gets attached at the top
        objRoot.addChild(spinGroup); //TransformGroup and sphere
        objRoot.addChild(directionalLight);
        objRoot.addChild(ambientLight);
        return objRoot;
    }
    public ColorInterp() {
    }
    public void init() {
        BranchGroup scene = createSceneGraph();
        setLayout(new BorderLayout());
        GraphicsConfiguration config =
           SimpleUniverse.getPreferredConfiguration();
        canvas = new Canvas3D(config);
        add("Center", canvas);
        // Create a simple scene and attach it to the virtual universe
        universe = new SimpleUniverse(canvas);
        setupView();
        universe.addBranchGraph(scene);
    }
    public void destroy() {
        universe.removeAllLocales();
    }
    //
    // The following allows ColorInterp to be run as an application
    // as well as an applet
    //
    public static void main(String[] args) {
        new MainFrame(new ColorInterp(), 256, 256);
    }
}
```

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.