Chapter 9: Loading and Manipulating External Models

It is possible to build complex geometries in Java 3D by using a subclass of GeometryArray (e.g. a QuadArray, TriangleStripArray), but there is little help available except for the “try it and see” approach. It makes much better sense to create the object using 3D modeling software, and then load it into your Java 3D application at run time.

This chapter describes two Java 3D programs which load models, placing them in the checkboard scene described in chapter 8.

LoaderInfo3D.java shows how a loaded object can be examined and its component shapes manipulated to change their colour, transparency, texture, and other attributes. Figure 1 shows a close-up of a robot that has been turned blue.

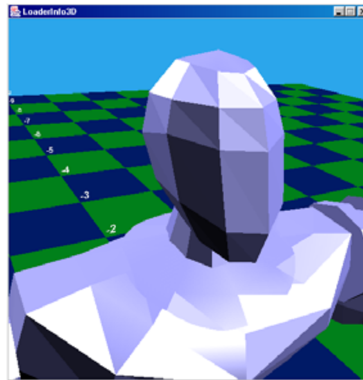

Figure 1. A Blue Robot.

Loader3D.java shows how a loaded model’s position, orientation, and size can be adjusted, and the details remembered when the model is next loaded. Figure 2 shows a castle that has been moved, rotated, and scaled.

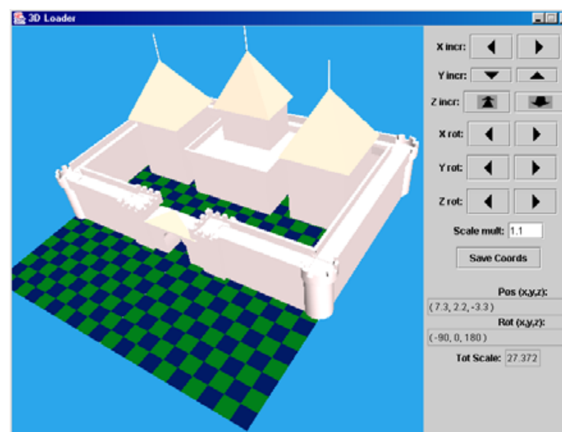

Figure 2. A Repositioned and Scaled Castle.

This chapter will discuss the following Java 3D techniques:

- external model loading (using NCSA Portfolio);

 © Andrew Davison. 2003

- scene graph traversal;
- shape modification (changing the shape's colour, rendering as a wireframe, setting a transparency level, adding a texture, modulating texture and colour);
- integrating GUI controls with the Java 3D canvas;
- shape positioning, scaling, rotation.

The first three points relate to LoaderInfo3D.java, the last two to Loader3D.java

UML Diagrams for LoaderInfo

The UML class diagrams in Figure 3 only show the visible methods (there is no public data in any of the classes).

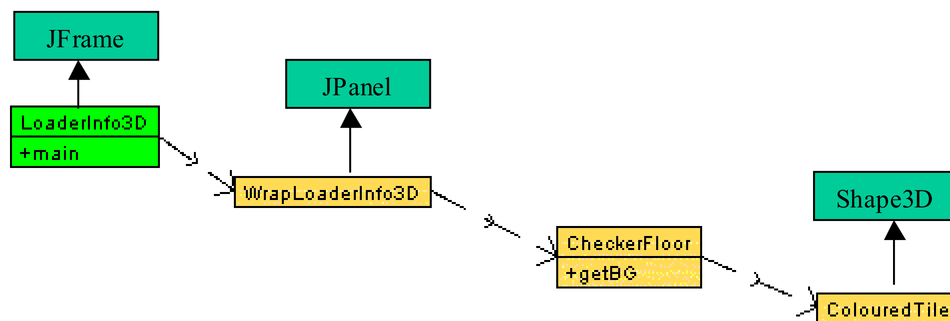

Figure 3. UML Class Diagrams for LoaderInfo3D.

An important point is the similarity between many of the classes in LoaderInfo and those in Checkers3D from chapter 8; in fact, we have reused the CheckerFloor and ColouredTile classes unchanged.

LoaderInfo3D is the top-level JFrame for the application, and is very similar to Checkers3D except that it takes two arguments from the command line (the name of the file to load and a 'adaption' number) and passes them to WrapLoaderInfo3D. An example command line:

```
java -cp %CLASSPATH%;ncsa\portfolio.jar LoaderInfo3D Coolrobo.3ds 0
```

This renders the robot model stored in Coolrobo.3ds in blue, as shown in Figure 1. The classpath argument is used to include the loaders package stored in portfolio.jar.

The code can be found in Code/LoaderInfo3D/.

 © Andrew Davison. 2003

Most of the new code appears in WrapLoaderInfo3D, and Figures 5 lists its methods.

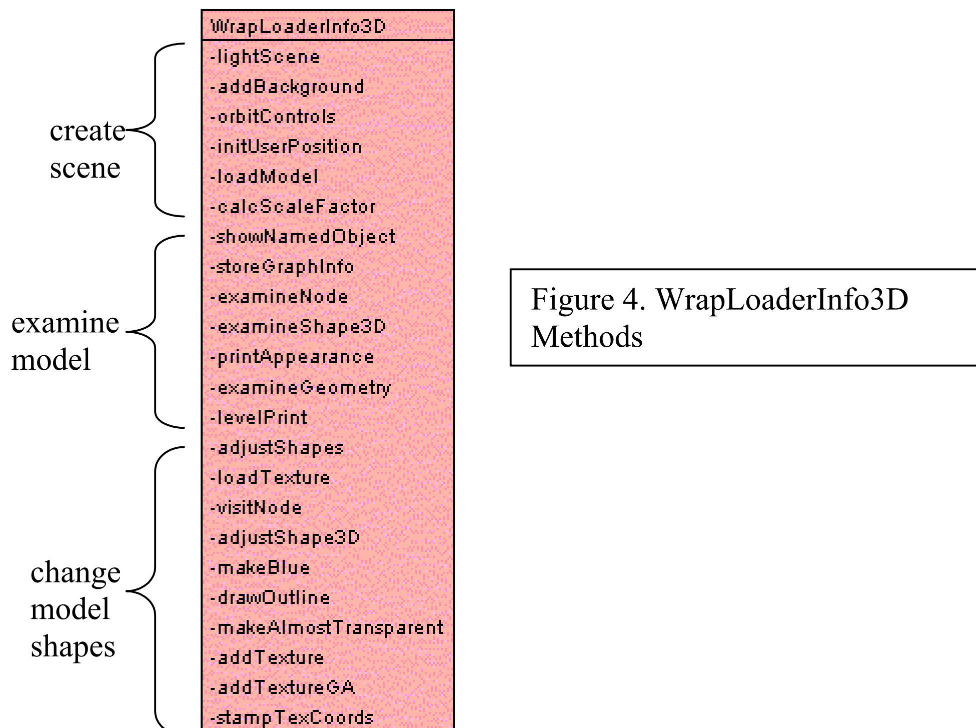

Figure 4. WrapLoaderInfo3D Methods

The methods fall into 3 groups: the 'create scene' methods build the scene by adding the checkboard floor, lights, the background, and the viewer's OrbitBehavior. In this respect, WrapLoaderInfo3D, is the same as WrapCheckers3D in chapter 8. However, there are additional methods for loading the model (loadModel() and calcScaleFactor ()).

The 'examine model' methods traverse the model's scene graph and save the collected information into a text file.

The 'change model shapes' methods traverse the scene graph looking for Shape3D nodes and modify them according to the 'adaption' number entered at the command line.

Loaders in Java 3D

Java 3D supports external model loading through its Loader interface and the Scene class.

Loader takes as input the model's filename and flags for enabling/disabling the loading of certain elements of the model, such as light nodes, sound nodes, and view graphs.

Java 3D's utilities package includes two subclasses of Loader aimed at particular file formats: Lw3dLoader handles Lightwave 3D scene files, and ObjectFile processes Wavefront .obj files. A third subclass, LoaderBase, implements the Loader interface in a generic way to encourage the building of loaders for other 3D formats through subclassing.

 © **Andrew Davison. 2003**

The Scene class uses a Loader object to extract details about a model, the most significant being its BranchGroup (usually for the purpose of adding it to the application scene). Information about other aspects of the model is also available, including the model's light nodes, object names, view points, and behaviour nodes. However, not all loaders supply this information -- the relevant get methods may return nothing.

There are a wide range of Java 3D loaders for different file formats, written by third party developers. A very good list is maintained at http://www.j3d.org/utilities/loaders.html.

In this chapter, and others, we employ the loaders in the NCSA Portfolio package (available from http://www.ncsa.uiuc.edu/~srp/Java3D/portfolio/). Using a single ModelLoader interface, the package supports a wide range of formats, including 3D Studio Max (3ds files), AutoCAD (dxf), Digital Elevation Maps (dem), TrueSpace (cob), and VRML 97 (wrl). The downsides of Portfolio are its relatively advanced age (the current version is 1.3, from 1998), and its relatively simple support of the formats: often only the geometry and shape colours are loaded, no textures, behaviours, or lights. Portfolio offers more than just loaders, it also has interfaces for several kinds of input devices, and makes it easy to take snapshots of the 3D canvas and convert them into video clips.

Inspector3ds is a more recent 3ds loader, developed by John Wright at Starfire Research (http://www.starfireresearch.com). The loader handles geometry, materials, textures, and normals.

The popular modeling package ac3d (http://www.ac3d.org) has a loader written by Jeremy Booth, available at http://www.newdawn.software.com. Also at the same site are loaders for different versions of Quake.

Programmers wishing to utilise a modern VRML loader should consider the Xj3D loader (http://www.web3d.org), which is actively being developed, and covers most of the VRML 2.0 standard. The actual aim is to load X3D files which extend VRML with XML functionality.

For the artistically-impared (e.g. yours truly), there are a profusion of Web sites that offer 3D models. A good starting point is the Google directory on 3D models http://directory.google.com/Top/Computers/Software/Graphics/3D/Models/. One site with many free models is 3D Model World (http://3DmodelWorld.com/).

Using NCSA Portfolio Loaders

The ModelLoader interface is used by WrapLoaderInfo3D in loadModel():

```
import ncsa.j3d.loaders.*;      // Portfolio loaders
import com.sun.j3d.loaders.Scene;
   :
private Scene loadedScene = null;   // globals
private BranchGroup loadedBG = null;

    :
public void loadModel(String fn)
{   :
  try {
    ModelLoader loader = new ModelLoader();
```

 © Andrew Davison. 2003

```
      loadedScene = loader.load(fn);    // the model's scene
      if(loadedScene != null ) {
        loadedBG = loadedScene.getSceneGroup();  // model's BG
          :
      }
    }
    catch( IOException ioe )
    { System.err.println("Could not find object file: " + fn); }
  }
```

The code shows the usual loading sequence: first a Loader object is obtained, which is used to load a Scene object. If this is successful, then a call to getSceneGroup() extracts the model's BranchGroup (into loadedBG).

The compilation of the LoaderInfo3D classes must refer to portfolio.jar which contains the Portfolio packages:

```
javac -classpath %CLASSPATH%;ncsa\portfolio.jar *.java
```

Using the Model's BranchGroup

The model's BranchGroup usually requires some manipulation before it is suitable for the application. In loadModel(), the BranchGroup is rotated clockwise around the x-axis by 90 degrees (PI/2 radians) and scaled to be no bigger than 10 world units across. The resulting graph is added to the scene:

```
loadedBG = loadedScene.getSceneGroup();    // model's BG

Transform3D t3d = new Transform3D();
t3d.rotX( -Math.PI/2.0 );        // rotate
Vector3d scaleVec = calcScaleFactor(loadedBG, fn);
t3d.setScale( scaleVec );        // scale
TransformGroup tg = new TransformGroup(t3d);

tg.addChild(loadedBG);
sceneBG.addChild(tg);   // add (tg->loadedBG) to scene
```

Figure 5 shows a loaded model containing three dolphins.

Figure 5. Loaded Dolphins Model.

 © Andrew Davison. 2003

The scene graph for the application is shown in Figure 6.

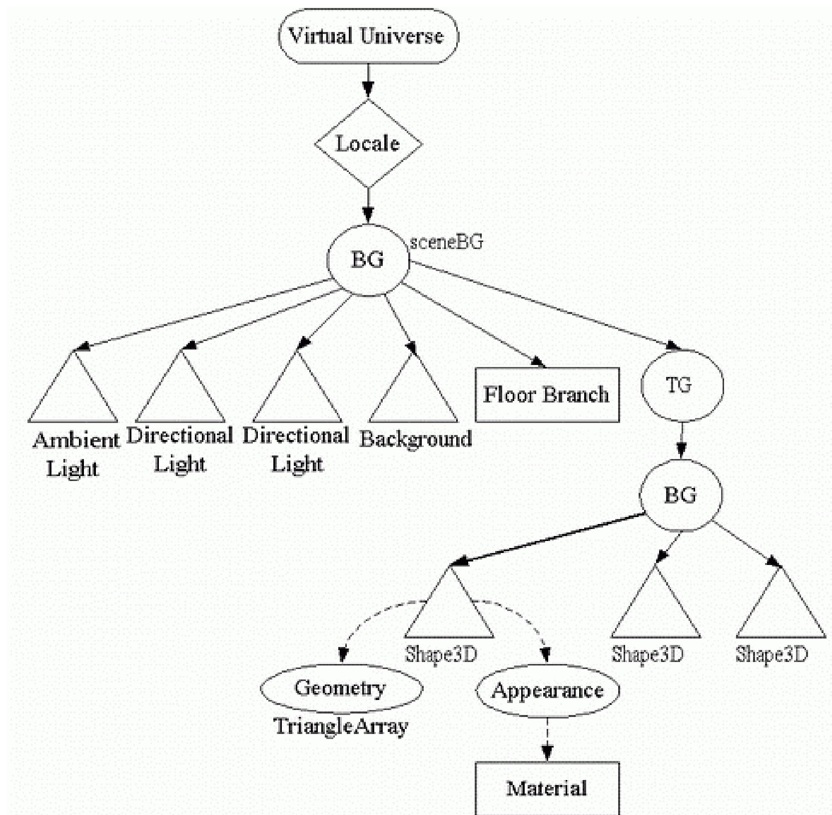

Figure 6. Scene Graph for the Loaded Dolphins.

Each dolphin is represented by a Shape3D node, as a child of the same BranchGroup. In the code fragment above, loadedBG is the BG of figure 6, and tg is the TG.

 © **Andrew Davison. 2003**

Rotating the Model

Without the rotation operation, many models, including most 3ds files, are loaded in a face down position (as shown in Figure 7).

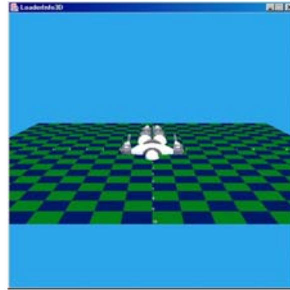

Figure 7. A Loaded Robot with no Rotation.

The reason is that the axes in 3D Studio Max use the XY plane as the floor with the z-axis vertical (see Figure 8 on the left). This means that a vector, such as (0,0,1) which is straight up in 3D Studio Max will be loaded in a prone position in Java 3D (see Figure 8 on the right).

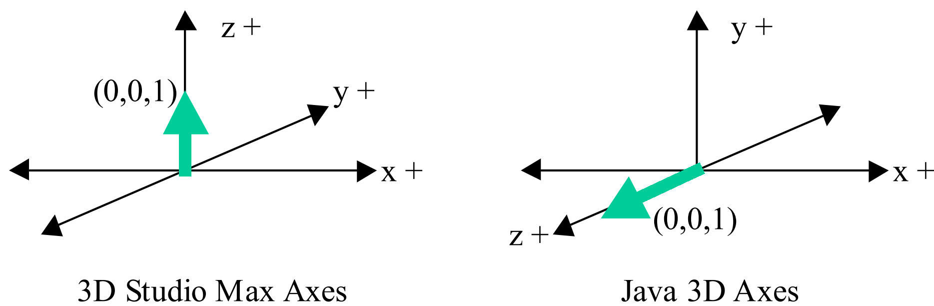

Figure 8. Coordinate Axes in 3D Studio Max and Java 3D.

 © Andrew Davison. 2003

The rotation operation in loadModel() is:

```
t3d.rotX(-Math.PI/2.0);
```

It specifies a negative, clockwise rotation of 90 degrees about the x-axis according to Java 3D's right-hand rule for rotations: place your closed right hand with its thumb pointing in the direction of the positive axis of interest, and your fingers will be bent in the direction of a positive rotation (see Figure 9).

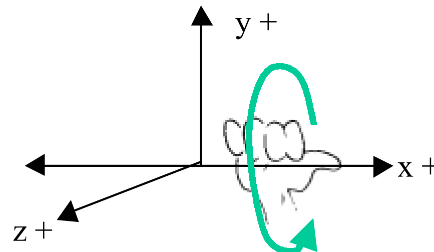

Figure 9. Right Hand, Positive Rotation for the X-Axis.

Figure 10 shows the robot from Figure 7 after being rotated.

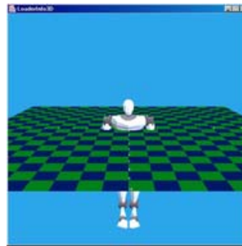

Figure 10. A Loaded Robot with a Negative Rotation.

Scaling the Model

A model may become very large when loaded into Java 3D's coordinate space. This can be corrected by using the object's bounding box to calculate a suitable scaling factor. This approach is employed in calcScaleFactor():

```
private Vector3d calcScaleFactor(BranchGroup loadedBG, String fn)
{
   BoundingBox boundbox = new BoundingBox( loadedBG.getBounds() );
   // obtain the upper and lower coordinates of the box
   Point3d lower = new Point3d();
   boundbox.getLower( lower );
   Point3d upper = new Point3d();
   boundbox.getUpper( upper );

   // calculate the maximum dimension, e.g.
   ... double max = (upper.x - lower.x );
       :  // the actual code is a bit more complex
   double scaleFactor = 10.0/max;
   return new Vector3d(scaleFactor, scaleFactor, scaleFactor);
}  // end of calcScaleFactor()
```

 © Andrew Davison. 2003

Accessing Other Scene Information

The Scene class offers numerous methods for accessing details of the model related to lighting, sound, behaviours, and so on. For example, many file formats allow the components of the model to be labeled with names; this is done with the DEF construct in VRML 2.0, as in:

```
DEF Box Transform {
  children Shape {
    appearance Appearance {
            material Material {...} }
    geometry Box {...}
} }
```

The names of these objects can be obtained from the Scene object with getNamedObjects() which returns a Hashtable of objects, where the keys are the names and the values are the objects. It is simple to iterate through the keys and print them out; see showNamedObject() for details.

```
Hashtable namedObjects = loadedScene.getNamedObjects();
Enumeration e = namedObjects.keys();
while(e.hasMoreElements())
  System.out.println( (String)e.nextElement() );
```

Such code depends on the capabilities of the underlying loader to supply the basic information. Unfortunately, the Portfolio loader for VRML 2.0 does not retrieve it.

Traversing the Scene Graph

Moving over the model's scene graph is relatively easy due to the parent-child relationship between the nodes, and the fact that all the nodes are subclasses of a single superclass, SceneGraphObject. A simplified inheritance hierarchy is shown in Figure 11.

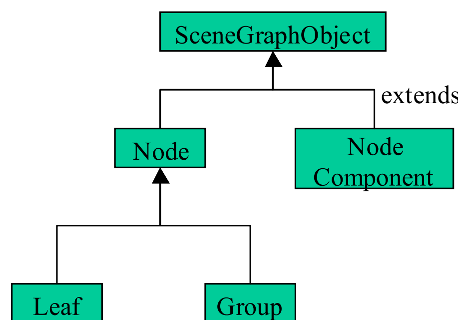

Figure 11. Some Subclasses of SceneGraphObject.

As explained in chapter 8, Leaf nodes are subclassed in various ways to obtain Shape3D and environment nodes for lighting, backgrounds, sound, and so on. The subclasses of Group include BranchGroup andTransformGroup, which may have their own children (Group and/or Leaf nodes). NodeComponent objects are used to store information in nodes, such as Geometry and Appearance attributes, and may be shared between nodes.

A simple algorithm for traversing a scene graph:

© Andrew Davison. 2003

```
examineNode(node)
  if the node is a Group {
    print Group info;
    for each child of the node
      examineNode(child);  // recursive call
  }
  else if the node is a Leaf
    if the node is a Shape3D {
      examine its appearance;
      examine its geometry;
    }
    else print Leaf info
  }
  else print general node info;
```

The algorithm is simplified by concentrating only on a few node types, principally Shape3D, and by considering the graph as a tree. Shape3D details are often the most important since they store the model's geometry, and there is little point looking for environmental data since it is frequently not converted to Java 3D by the loader.

The above pseudo code is the heart of the examineNode() and examineShape3D() methods in WrapLoaderInfo3D.

examineShape3D() calls printAppearance() to "examine its appearance", which is confined to reporting ColouringAttributes and/or Material details. Many other Appearance components could be considered.

examineShape3D() calls examineGeometry() to "examine its geometry", which checks out the possible subclasses of the Geometry object. Loaded models almost always use a subclass of GeometryArray (e.g. TriangleArray, QuadArray), and we report the number of verticies in the array.

examineShape3D() is made a little more complicated by dealing with the (slight) possibility that there may be several geometries assigned to a single shape.

Two useful methods for this kind of traversal code are: getClass() which returns the class name of the object, and the infix operation `instanceof` that tests for membership in a class (or superclass).

 © Andrew Davison. 2003

examineNode() is called from StoreGraphInfo() which first sets up a FileWriter object linked to the text file examObj.txt. The output of the traversal is redirected into the file, as illustrated in Figure 12 when the dolphins model was examined. The rendering of the dolphins is shown in Figure 5.

```
Group: class javax.media.j3d.BranchGroup
3 children
  Leaf: class javax.media.j3d.Shape3D
  Material Object:AmbientColor=(0.7, 0.7, 0.7) EmissiveColor=(0.0, 0.0, 0.0)
  DiffuseColor=(0.3, 0.3, 0.3) SpecularColor=(1.0, 1.0, 1.0) Shininess=0.6
  LightingEnable=true ColorTarget=2
    Geometry: class javax.media.j3d.TriangleArray
    Vertex count: 1692

  Leaf: class javax.media.j3d.Shape3D
  Material Object:AmbientColor=(0.7, 0.7, 0.7) EmissiveColor=(0.0, 0.0, 0.0)
  DiffuseColor=(0.3, 0.3, 0.3) SpecularColor=(1.0, 1.0, 1.0) Shininess=0.6
  LightingEnable=true ColorTarget=2
    Geometry: class javax.media.j3d.TriangleArray
    Vertex count: 1692

  Leaf: class javax.media.j3d.Shape3D
  Material Object:AmbientColor=(0.7, 0.7, 0.7) EmissiveColor=(0.0, 0.0, 0.0)
  DiffuseColor=(0.3, 0.3, 0.3) SpecularColor=(1.0, 1.0, 1.0) Shininess=0.6
  LightingEnable=true ColorTarget=2
    Geometry: class javax.media.j3d.TriangleArray
    Vertex count: 1692
```

Figure 12. examObj.txt after Examining the Dolphins Model

The three dolphins are represented by a BranchGroup with three Shape3D children. These store TriangleArrays for each dolphin’s geometry, and have the same Material colours. This information corresponds to the model’s scene graph in Figure 6.

Adjusting a Model’s Shape Attributes

Many aspects of a model can be easily changed once its individual Shape3D nodes are accessible. This can be done with a variant of the examineNode() pseudo code, concentrating only on Leaf nodes which are Shape3Ds:

```
visitNode(node)
  if the node is a Group {
    for each child of the node
      visitNode(child);  // recursive call
  }
  else if the node is a Shape3D
     adjust the node’s attributes;
```

This pseudo code is the basis of visitNode() in WrapLoaderInfo3D.

The manipulation of the shape’s attributes is initiated in adjustShape3D() which uses the ‘adaption’ number entered by the user to choose between six possibilities:

- 0: Make the shape blue with makeBlue();
- 1: Draw the shape in outline with drawOutline();
- 2: Render the shape almost transparent with makeAlmostTransparent();
- 3: Lay a texture over the shape with addTexture();

- 4: Make the shape blue and add a texture by calling makeBlue() and addTexture();
- Anything else: Make no changes at all.

Turning the Shape Blue

Figure 11 shows the rendering of the dolphins model after being turned blue.

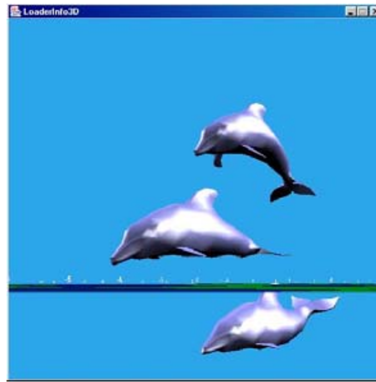

Figure 11. Blue Dolphins.

The Material node used in makeBlue() is:

```
Material blueMat = new Material(black, black, blue, white, 20.0f);
```

The use of a black (Color3f(0.0f, 0.0f, 0.0f)) ambient colour means that unlit parts of the shape are rendered in black, which looks like shadow on the model. However, the model does not cast any shadows onto other surfaces, such as the floor. A simple technique for shadow casting is described in chapter ??.

```
Appearance app = shape.getAppearance();
Material blueMat = new Material(black,black,blue,white,20.0f);
blueMat.setLightingEnable(true);
app.setMaterial( blueMat );
shape.setAppearance(app);
```

Note how the appearance is obtained from the shape, its material attribute changed, and then the appearance component assigned back to the shape -- only the attribute of interest is modified.

© Andrew Davison. 2003

Drawing a Shape in Outline

Figure 12 shows a VRML model of a box, cone and sphere rendered in outline:

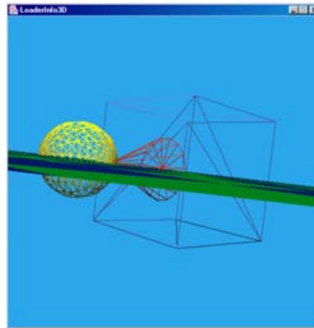

Figure 12. Shapes in Outline.

The original colours of the three objects (yellow, red, blue) are still visible in their line colours.

The effect is achieved by setting the POLYGON_LINE mode in PolygonAttribute in drawOutline():

```
Appearance app = shape.getAppearance();
PolygonAttributes pa = new PolygonAttributes();
pa.setCullFace( PolygonAttributes.CULL_NONE );
pa.setPolygonMode( PolygonAttributes.POLYGON_LINE );
app.setPolygonAttributes( pa );
shape.setAppearance(app);
```

We also disable culling so that the lines are visible from every direction.

Making a Shape Almost Transparent

Figure 13 show a model of gun rendered almost transparent.

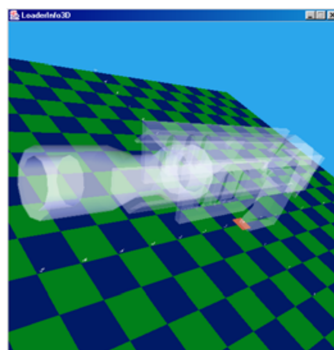

Figure 13. Semi-visible Gun.

This is done in makeAlmostTransparent() by setting the TransparencyAttributes of the shape's Appearance.

```
Appearance app = shape.getAppearance();
TransparencyAttributes ta = new TransparencyAttributes();
ta.setTransparencyMode( TransparencyAttributes.BLENDED );
ta.setTransparency(0.8f);     // 1.0f is totally transparent
app.setTransparencyAttributes( ta );
shape.setAppearance(app);
```

 © Andrew Davison. 2003

There are various transparency mode settings which affect how the original colour of the shape is mixed with the background pixels.

A comparison of the last three examples shows the general strategy for manipulating a shape: create an attribute setting, then add it to the existing Appearance component of the shape.

Adding a Texture to a Shape

Figure 14 shows a castle with a rock texture wrapped over it.

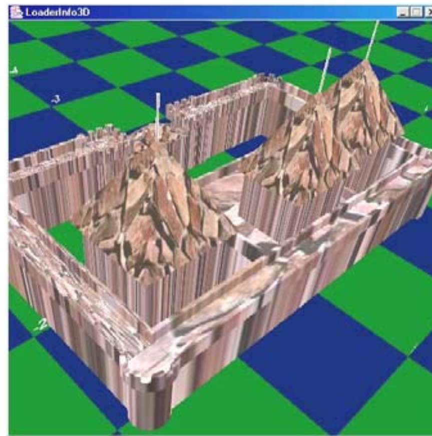

Figure 14. Castle Rock.

A quick look at Figure 14 shows that some of the texturing is quite unrealistic: there are very clear stripes of textures running down the walls. These stripes are actually running along the z-axis of the model, but the model was rotated about the x-axis at load time. The reason for this z-striping is explained below.

Loading a Texture

A texture is created in two stages: first a TextureLoader object is created for an associated graphics file, then used to get the texture:

```
TextureLoader texLoader = new TextureLoader("stone.jpg", null);
Texture2D texture = (Texture2D) texLoader.getTexture();
 if (texture != null)
   texture.setEnable(true);
```

If the texture is successfully loaded, then it should be enabled, so it can be mapped to a geometry later.

TextureLoader can handle JPEGs and GIFs (which are useful if transparency is required), and it can be employed in conjunction with JAI (Java Advanced Imaging) to load other formats, such as BMP, PNG, and TIFF files. The loader can include various flags, such as one for creating textures at various levels of resolution for rendering onto small areas. Aside from Textures, the loader can return ImageComponent2D objects which is the Java 3D format for images used in backgrounds and rasters.

 © Andrew Davison. 2003

Textures can be 2D (as here) or 3D; Texture3D objects are employed for volumetric textures, typically in scientific applications.

A common reason for the failure of getTexture() is that the image must have equal width and height dimensions, and the dimensions must be a power of 2. For instance, our stone.jpg image is 256 by 256 pixels.

For a texture to be applied to a shape, three conditions are necessary:

1. The shape must have texture coordinates, either set through its geometry (considered later) or using a TexCoordGeneration object (as here);
2. The shape's appearance must have been assigned a Texture2D;
3. The texture must be enabled (which we have already done).

Texture Coordinates

Texture2D coordinates are measured with (s,t) values which range between 0 and 1. Texture mapping is the art of mapping (s,t) values (sometimes called texels) onto geometry coordinates (x,y,z) to create a realistic looking effect.

The main question is how the texture will be reused to cover geometry coordinates outside of the texture's 0 to 1 range.

One solution is to repeat (tile) the texture in 1-by-1 patches over the geometry's surface. However, this mapping is carried out upon the original geometry of the shape, which is commonly very large. This means that tiling may create excessive repetition of the pattern and, after the geometry has been scaled down for the Java 3D world, the texture's details may be too small to see.

Another answer is to 'magnify' the texture so that it maps onto a large range of pixels. It is also possible to 'shrink' the texture so it maps onto a smaller range of pixels.

A TexCoordGeneration object lets the programmer create custom equations that specify how geometry coordinates (x,y,z) are converted into texels (s,t). The simplest equations are linear, of the form:

s = (x*planeS.xc) + (y*planeS.yc) + (z*planeS.zc) + (planeS.w)

t = (x*planeT.xc) + (y*planeT.yc) + (z*planeT.zc) + (planeT.w)

planeS and planeT are vectors that contain the xc, yc, zc, and w constants which define the equations. The constants can be varied to produce some interesting looking effects.

We will explain a simple technique for calculating a set of constants for 'stretching' a single texture over an entire geometry. An advantage of our approach is that the constants for *any* shape are rapidly calculated with the help of bounding boxes.

© **Andrew Davison. 2003**

Figure 15 shows a bounding box for a shape, with its upper and lower points highlighted. The upper point contains the maximum x, y, and z values, while the lower point has the minima. The problem is to map all the coordinates inside this box to a 0-by-1 texture.

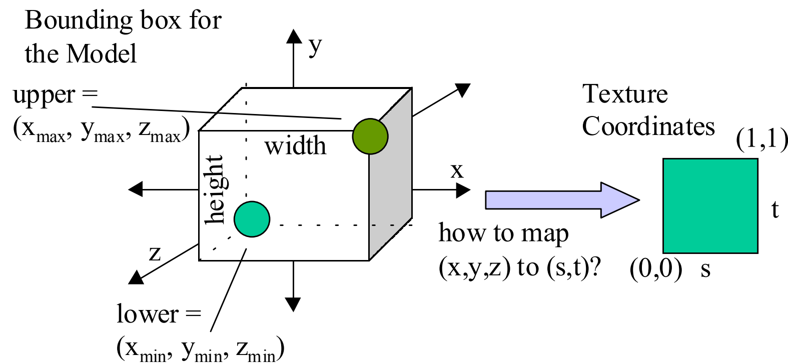

Figure 15. From Bounding Box to Texture.

The height and width of the bounding box is easily calculated:

$$height = y_{max} - y_{min}$$
$$width = x_{max} - x_{min}$$

Two simple equations for s and t are then given by:

$$s = x/width - x_{min}/width$$
$$t = y/height - y_{min}/height$$

This is expressed in vector form as:

$$planeS = [1/width, 0, 0, -x_{min}/width]$$
$$planeT = [0, 1/height, 0, -y_{min}/height]$$

The drawback is that we have ignored the z-axis, which means that (x,y,z) coordinates with the same (x,y) value but different z values will all map to the same (s,t) texel. This is why z-striping is visible on the castle in Figure 14.

The code for all of this can be found in stampTexCoords() which takes the shape as input and returns a TexCoordGeneration object.

```
BoundingBox boundBox = new BoundingBox( shape.getBounds() );
Point3d lower = new Point3d();
Point3d upper = new Point3d();
boundBox.getLower(lower);  boundBox.getUpper(upper);
double width = upper.x - lower.x;
double height = upper.y - lower.y;
Vector4f planeS = new Vector4f( (float)(1.0/width), 0.0f,
                    0.0f, (float)(-lower.x/width));
Vector4f planeT = new Vector4f( 0.0f, (float)(1.0/height),
                    0.0f, (float)(-lower.y/height));
```

The TexCoordGeneration object is initialised with the two vectors:

 © Andrew Davison. 2003

```
TexCoordGeneration texGen = new TexCoordGeneration();
texGen.setPlaneS(planeS);
texGen.setPlaneT(planeT);
```

texGen is passed back to the main texture method, addTextureGA().

addTextureGA() has four main duties:

- to switch off face culling so the texture appears on all sides of the shape;
- to generate a TexCoordGeneration object, as detailed above;
- to modulate the texture mode so that the underlying colour and texture are combined;
- to assign the texture to the shape, which is done using setTexture().

Texture Attributes

The modulation task in addTextureGA() utilises a TextureAttributes object to control how the texture is combined with the surface colours of the shape. TextureAttributes can be employed for other things, such as transforming the texture (e.g. rotating, scaling), but we will not use those features. There are several texture modes:

- REPLACE: the texture replaces any shape colour;
- MODULATE: the texture and colour are combined (as here);
- DECAL: the transparent areas of the texture are not drawn onto the shape;
- BLEND: allows a varying mix of texture and colour.

The MODULATE mode is often used to combine an underlying Material with a texture, which allows lighting and shading effects to be seen alongside the texture. Figure 16 shows the dolphins models turned blue and with a texture. The effects are clearly visible, and should be compared with the blue dolphins of Figure 11.

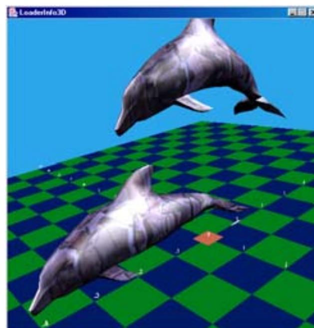

Figure 16. Textured, Blue Dolphins.

The code which sets the texture attribute is:

```
TextureAttributes ta = new TextureAttributes();
ta.setTextureMode( TextureAttributes.MODULATE );
app.setTextureAttributes( ta );
```

 © Andrew Davison. 2003

Loader3D Overview

Like LoaderInfo3D, Loader3D also loads an external model with a Porfolio loader, but is concerned with how it can be subsequently moved, rotated and scaled.

The model is displayed in a 3D canvas on the left hand side of the application, while a series of buttons (and a textfield) on the right allow the model to be manipulated (see Figure 18 for an example). Details of the model's new configuration can be saved to a text file, which can be loaded with the model next time, so that the model begins with the given location, orientation, and size.

The UML diagrams for the Loader3D application are shown in Figure 17; only the visible methods are shown.

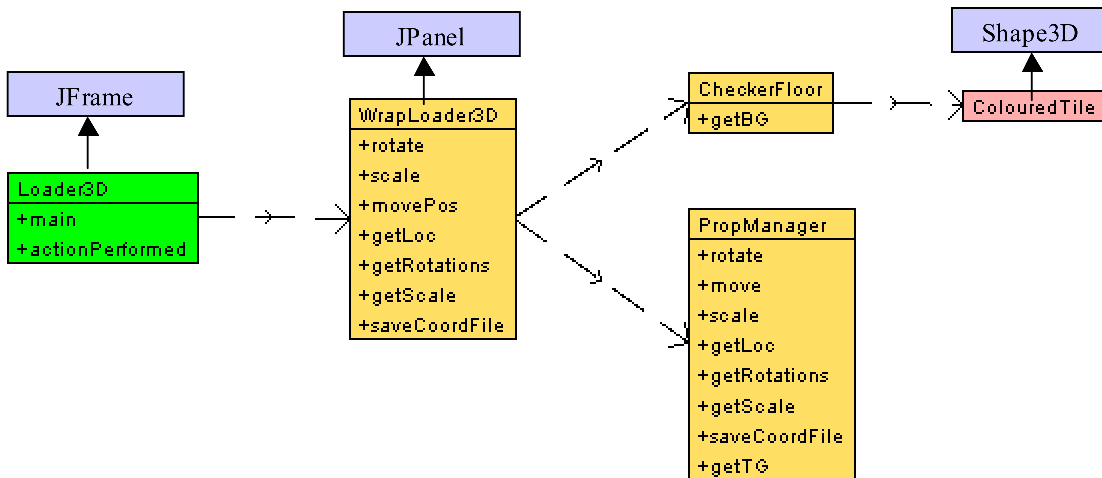

Figure 17. UML Class Diagrams for Loader3D.

The CheckerFloor and ColouredTile classes are as in previous examples.

Loader3D is complicated by the need for GUI controls, and to process user interactions. These are sent to WrapLoader3D, which passes them onto PropManager. PropManager is in charge of altering the model's position, orientation, and scale.

The code is located in Code/Loader3D/.

Using Loader3D

Loader3D can be called in two ways:

```
java -cp %CLASSPATH%;ncsa\portfolio.jar Loader3D <filename>
```

or

```
java -cp %CLASSPATH%;ncsa\portfolio.jar Loader3D –c <filename>
```

The application searches the /models subdirectory for the filename, and loads the file. If the –c option is included, it will also attempt to load the text file `<filename>Coords.txt`, which contains translation, rotation, and scaling values (called "coords" data) that should be applied to the model.

 © Andrew Davison. 2003

Figure 18 shows the Coolrobo.3ds model initially loaded into the application.

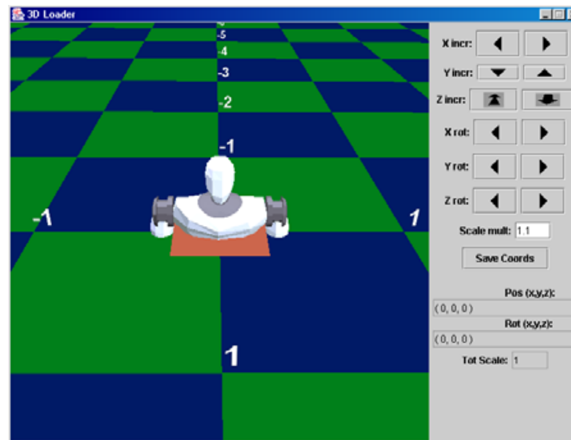

Figure 18. Coolrobo.3ds first Loaded.

Figure 19 shows the model after it has been moved, rotated, and scaled in various ways.

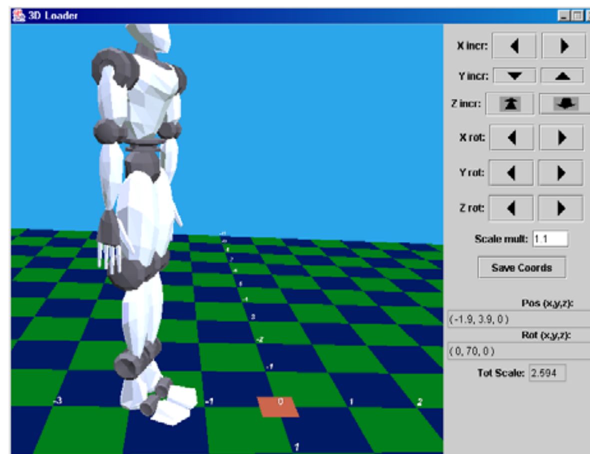

Figure 19. Coolrobo.3ds after Manipulation.

The user's viewpoint has been changed in figures 18 and 19 in order to make the images bigger on screen. However, the axis labels and red central square give an indication of how the model's configuration has changed.

The bottom half of the GUI pane in Figure 19 shows the current configuration: the (x,y,z) position is (-1.9, 3.9, 0) which is the distance of the model's centre from its starting point. The rotation values are (0, 70, 0) which means a 70 degree positive rotation aroud the y-axis. The model has been scaled by a factor of 2.594.

When the "Save Coords" button is presssed, the current "coords" data is saved to a text file in the subdirectory /models. The contents of the file generated for Coolrobo.3ds (CoolroboCoords.txt) is shown below:

```
Coolrobo.3ds
-p -1.9 3.9 0
-r 3333333
-s 2.594
```

 © Andrew Davison. 2003

The –p line gives the (x,y,z) translation, the –r line contains a series of rotation numbers, explained below, and the –s value is for scaling.

The Loader3D Class

The methods defined in Loader3D are given in Figure 20.

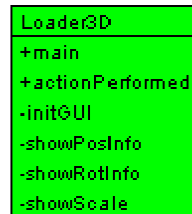

Figure 20. Loader3D Methods.

Loader3D creates its GUI control panel with initGUI(). actionPerformed() handles the various GUI events triggered by pressing buttons and typing in the textfield.

Depending on the user request, actionPerformed() calls movePosn(), rotate(), scale(), or saveCoordsFile() in the WrapLoader3D class to request changes to the model's position, rotation, scaling, or to save its "coords" data.

At the end of actionPerformed(), showPosInfo(), showRotInfo() and showScale() communicate with WrapLoader3D to obtain the current "coord" data, to update the GUI display.

 © Andrew Davison. 2003

The WrapLoader3D Class

Figure 21 shows the methods defined in WrapLoader3D.

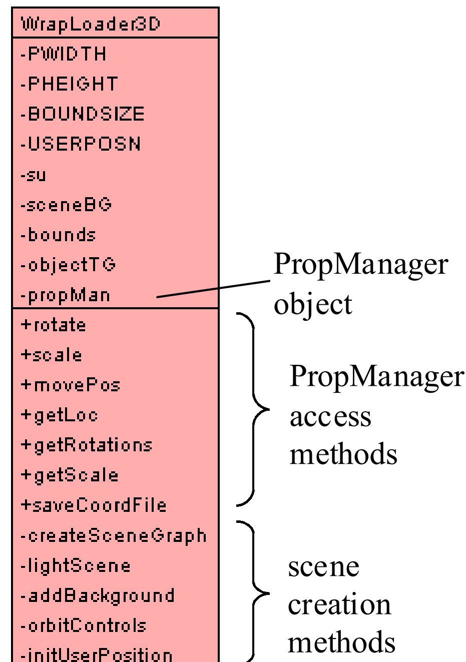

Figure 21. WrapLoader3D Methods.

WrapLoader3D does the usual job of creating the application's scene graph, but it also has a collection of visible methods for accessing its PropManager object, propMan. These methods do little more than pass the requests sent from the GUI interface into propMan.

The PropManager object is created in WrapLoader3D's constructor:

```
propMan = new PropManager(filename, hasCoordsInfo);
```

The call includes the model's filename and a boolean indicating the need for a "coords" data file.

The top-level TransformGroup for the model is accessed in createSceneGraph():

```
sceneBG.addChild( propMan.getTG() );
```

 © Andrew Davison. 2003

The PropManager Class

Figure 22 shows PropManager's methods. The numbered tasks are explained below. There are also methods which help with the translation and rotation operations, and methods used during debugging.

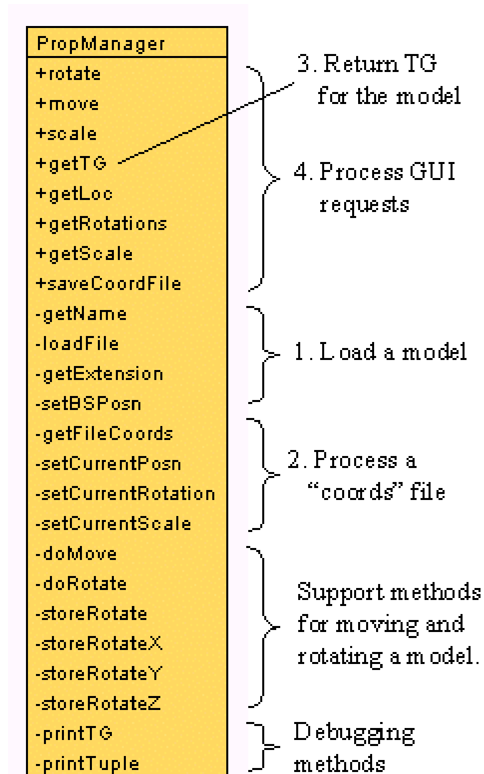

Figure 22. PropManager Methods.

There are four main tasks carried out by a PropManager object:

1. It loads the specified model, scales it to a standard size, and possibly rotates it .
2. It loads a "coords" data file if requested, and applies the translations, rotations, and scaling values to the model.
3. It makes the top-level TransformGroup for the model available. In this program, the subgraph is added to the scene by WrapLoader3D.
4. At run time, the PropManager object accepts commands to modify the model's position, orientation, and size, causing alterations to the model's scene graph. In this application, these commands come from the GUI, via WrapLoader3D.

The Scene Graph for the Loaded Model

Figure 23 shows the scene graph after the dolphins.3ds file has been loaded. The PropManager object creates the long branch shown on the right of the figure, consisting of a chain of four TransformGroup nodes and a BranchGroup with three Shape3D children. The loaded model is contained in the BranchGroup (the dolphins.3ds file contains three dolphins, each represented by a Shape3D node).

 © Andrew Davison. 2003

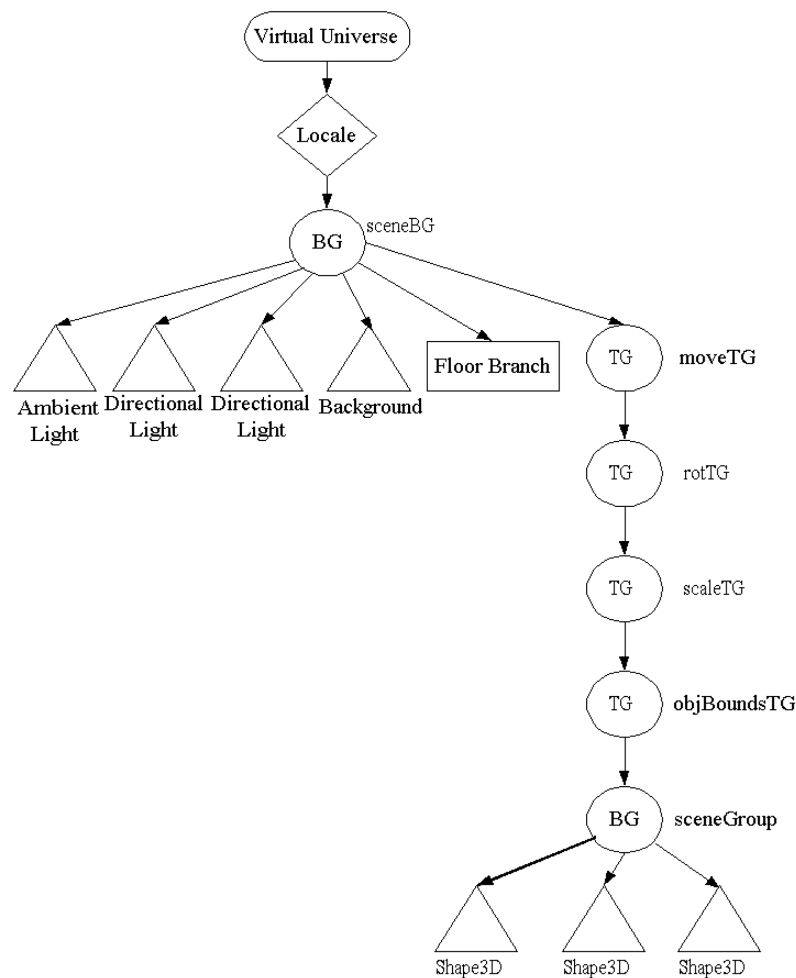

Figure 23. Scene Graph for Loaded Dolphins.

PropManager utilises four TransformGroups to deal with different aspects of the model's configuration:

- moveTG handles the translations;
- rotTG for rotations;
- scaleTG for scaling;
- objBoundsTG carries out the scaling and possible rotation of the model when it is first loaded.

The reason for this separation is to process distinct operations in different nodes in the graph. This reduces the overall complexity of the coding, because we can take advantage of *the hierarchy of local coordinate systems* used by the TransformGroup nodes.

A TransformGroup's local coordinate system means that it always starts at (0,0,0), with no rotation or scaling. However, when Java 3D renders the node into the virtual world, it must obtain its global coordinates (i.e. its virtual world position). It does this by calculating the combined effects of all the ancestor TransformGroup nodes operations upon the node.

 © **Andrew Davison. 2003**

For example, if the moveTG node is moved to coordinate (1,5,0) from its starting point of (0,0,0), then all the TransformGroup nodes below it are also repositioned as well. Java 3D generates this effect at render time but, as far as the child nodes are concerned, they are still at (0,0,0) in their local coordinate systems.

This mechanism greatly simplifies the programmer's task of writing TransformGroup transformations. For instance, a rotation of 70 degrees around the y-axis for rotTG is applied in its local coordinate system, so it is a straightforward rotation around the center. If the transformations of its parent (grandparent, great-grandparent, etc.) had to be taken into account, then the rotation operation would be much more complicated (we would need to undo all the transformations, rotate around the center, then apply the transformations again).

An advantage of spliting the translation and rotation effects so that the translation component (in moveTG) is above the rotation (in rotTG) is that rotational changes only apply to rotTG and its children.

For instance, a positive rotation of 90 degrees around the y-axis turns the XZ plane so the x and z axes are pointing in new directions. Subsequently, if a child of rotTG moves 2 units in the positive x direction, it will appear on screen as a 2 unit move in the negative z direction!

Fortunately, since moveTG is above rotTG the axes changes made by rotTG do not trouble moveTG: an x direction move applied to moveTG is always carried out along the x-axis, as expected by the user.

Task 1. Loading the Model

The scene graph is created in PropManager's loadFile(): first the model is loaded with ModelLoader, and it's BranchGroup is extracted into the sceneGroup variable. The chain of four TransformGroups are then created. The code snippet below shows the creation of objBoundsTG and scaleTG, and how scaleTG is linked to objBoundsTG:

```
// create a transform group for the object's bounding sphere
TransformGroup objBoundsTG = new TransformGroup();
objBoundsTG.addChild( sceneGroup );

// resize loaded object's bounding sphere (and maybe rotate)
String ext = getExtension(fnm);
BoundingSphere objBounds =
             (BoundingSphere) sceneGroup.getBounds();
setBSPosn(objBoundsTG, objBounds.getRadius(), ext);

// create a transform group for scaling the object
scaleTG = new TransformGroup();
scaleTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
scaleTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
scaleTG.addChild( objBoundsTG );  // link TGs
```

Note that the capability bits of scaleTG (and rotTG and moveTG) must be set to allow these nodes to be adjusted after the scene has been made live.

Also included in the code fragment is the call to setBSPosn(). It scales the model so that it's bounding sphere has a unit radius. This is a variant of the bounding box technique used in LoaderInfo3D. Also, if the file extension is ".3ds" then the model is

 © Andrew Davison. 2003

rotated -90 degrees around the x-axis to compensate for the axes differences between 3D Studio Max and Java 3D, as outlined earlier.

Task 2. Loading the "coords" Data

The "coords" data file requires parsing to extract its translation, rotation, and scaling values. getFileCoords() opens the file and reads in lines of text. These are passed to setCurrentPosn(), setCurrentRotation(), or setCurrentScale() depending on the character following the '-' at the start of a line.

setCurrentPosn() extracts the (x,y,z) values and calls doMove() with the values packaged as a Vector3d object. doMove() 'adds' the translation to the current value:

```
private void doMove(Vector3d theMove)
{ moveTG.getTransform(t3d);      // get current posn from TG
   chgT3d.setIdentity();             // reset change TG
   chgT3d.setTranslation(theMove);  // setup move
   t3d.mul(chgT3d);              // 'add' move to current posn
   moveTG.setTransform(t3d);         // update TG
}
```

chgT3d is a global Transform3D, so is reintialised before use by setting it to be an identity matrix.

The addition of the new translation is done using multiplication since we are dealing with matricies inside the Transform3D objects.

setCurrentScale() is similar: it extracts single value, then calls scale() to apply it to the scene graph:

```
public void scale(double d)
{ scaleTG.getTransform(t3d);     // get current scale from TG
  chgT3d.setIdentity();          // reset change Trans
  chgT3d.setScale(d);            // set up new scale
  t3d.mul(chgT3d);              // multiply new scale to current one
  scaleTG.setTransform(t3d);     // update the TG
  scale *= d;    // update scale variable
}
```

The coding style of scale() is the same as doMove().

© Andrew Davison. 2003

Handling Rotation

Dealing with rotation is more complicated due to the mathematical property that rotations about different axes are *non-commutative*. For example, a rotation of 80 degrees around the x-axis followed by 80 degrees about the z-axis (see Figure 24) produces a different result if carried out in the opposite order (se Figure 25).

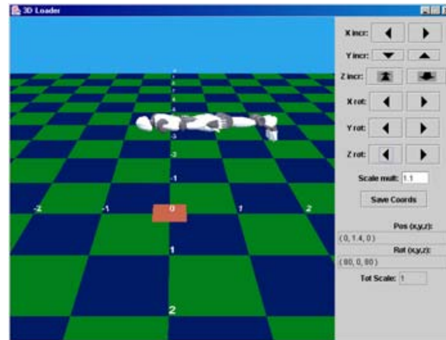

Figure 24. Rotation Order: x-axis rotation then z-axis.

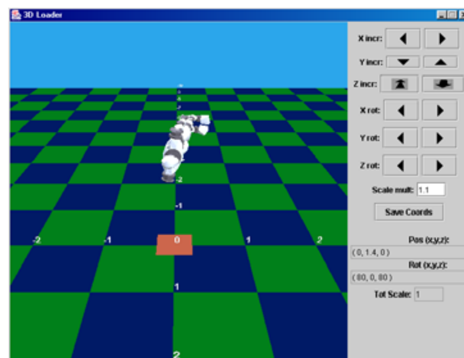

Figure 25. Rotation Order: z-axis rotation then x-axis.

Although the GUI displays are too small to read, they both show the rotation values to be (80,0,80).

This rotation property means that the “coord” data file cannot simply store the rotation information as three total rotations; it is also necessary to store the *order* in which the rotations are carried out.

The solution used here relies on a simplication of the user interface: a click of a rotation button always results in a rotation of 10 degrees (negative or positive, around the x-, y-, or z- axes). Then the user’s rotation commands can be represented by a sequence of rotation “numbers”, which must be executed in sequence order to duplicate the desired final orientation. The rotation numbers range between 1 and 6, and mean:

- 1 = positive ROT_INCR around x-axis
- 2 = negative ROT_INCR around x-axis
- 3 = positive ROT_INCR around y-axis
- 4 = negative ROT_INCR around y-axis
- 5 = positive ROT_INCR around z-axis
- 6 = negative ROT_INCR around z-axis

 © Andrew Davison. 2003

ROT_INCR is a constant defined in PropManager (10 degrees).

This approach means that the rotation information for Figure 24 is encoded as:

-r 1111111155555555

while Figure 25 is represented by:

-r 5555555511111111

The eight '1's mean 80 degrees around the x-axis, and the eight '5's mean 80 degrees around the z-axis.

This representation has the drawback that it may lead to very long strings, but this is unlikely considering the application. Usually, a model only needs turning through 90 or 180 degrees along one or perhaps two axes. However, if the user makes lots of adjustments to the rotation, they are all stored: in that case, it is probably better to exit the application and start again.

An advantage of the representation is the simple way that the sequence can be modified manually, through editing the "coords" data file in a text editor. This is also true for the position and scaling data, which can be changed to any value.

The sequence of rotation numbers is extracted form the "coords" data file in PropManager's setCurrentRotation(). It calls rotate() to carry out a rotation for each rotation number.

rotate() calls doRotate() to change the scene graph, and one of storeRotateX(), storeRotateY(), or storeRotateZ() to record the rotation in an ArrayList of rotation numbers and to update the total rotations for the x-, y-, or z- axes.

The doRotate() method:

```
private void doRotate(int axis, int change)
{
  double radians = (change == INCR) ? ROT_AMT : -ROT_AMT;
  rotTG.getTransform(t3d);      // get current rotation from TG
  chgT3d.setIdentity();         // reset change Trans
  switch (axis) {               // setup new rotation
    case X_AXIS: chgT3d.rotX(radians); break;
    case Y_AXIS: chgT3d.rotY(radians); break;
    case Z_AXIS: chgT3d.rotZ(radians); break;
    default: System.out.println("Unknown axis of rotation"); break;
  }
  t3d.mul(chgT3d);      // 'add' new rotation to current one
  rotTG.setTransform(t3d);      // update the TG
}
```

The coding style is quite similar to that in doMove() and scale(): the existing Tranform3D value is extracted from the TransforGroup node, updated to reflect the change, and then stored back in the node.

Task 3. Making the Model Available

As Figure 23 shows, the top-level of the model's scene graph is the moveTG TransformGroup. This can be accessed by calling getTG():

 © Andrew Davison. 2003

```
  public TransformGroup getTG()
  {  return moveTG; }
```

The one subtlety here is that the moveTG, rotTG, and scaleTG nodes will almost certainly be modified after the model's graph has been added to the scene. This means that their capability bits must be set to permit runtime access and changing.

Task 4. Modifying the Model's Configuration at Runtime

User requests to move, rotate, scale, or save the "coords" data, are passed from the GUI in Loader3D, through WrapLoader3D to the PropManager object. The relevant methods are move(), rotate(), scale(), and saveCoordFile().

We have already described rotate() and scale() : they are also employed when the "coords" data is being applied to the model. move()'s main purpose is to translate the data supplied by the GUI (an axis and direction) into a vector, which is passed to doMove().

saveCoordFile() is quite straightforward, but relies on global variables holding the current configuration information.

Another aspect of Loader3D's GUI is that it displays the current configuration. This is achieved by calling getLoc(), getRotations(), and getScale() in PropManager via WrapLoader3D.

© Andrew Davison. 2003