# Chapter 8. A 3D Checkboard: Checkers3D

This chapter describes a Java 3D example called Checker3D: it creates a scene consisting of a dark green and blue tiled surface with labels along the X and Z axes, a blue background, and a floating sphere lit from two different directions. The user (viewer) can move through the scene by moving the mouse.

The screenshot in Figure 1 show the initial view; the picture in Figure 2 was taken after the user had moved.

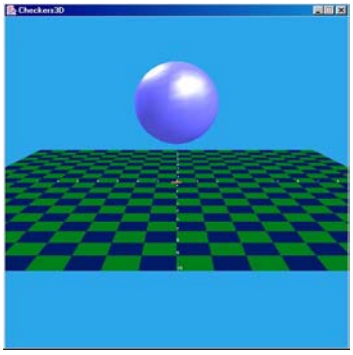

Figure 1. Initial View.

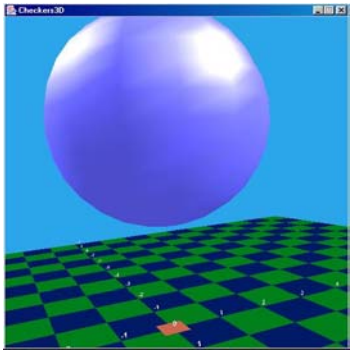

Figure 2. A Later View

This chapter will discuss the following Java 3D techniques:

- Canvas3D creation and its integration with Swing;
- scene graph creation;
- ambient and directional lighting;
- backgrounds using constant colour;
- the QuadArray, Sphere, and Text2D geometries;
- shape colouring and lighting;
- shape positioning;
- viewpoint positioning;
- viewpoint movement using OrbitBehavior;
- how to generate a scene graph diagram (like the one in Figure 4).

### UML Diagrams for Checkers3D

The UML class diagrams in Figure 3 were created with the help of ESS-Model and Reverse (see Appendix A5 for details).

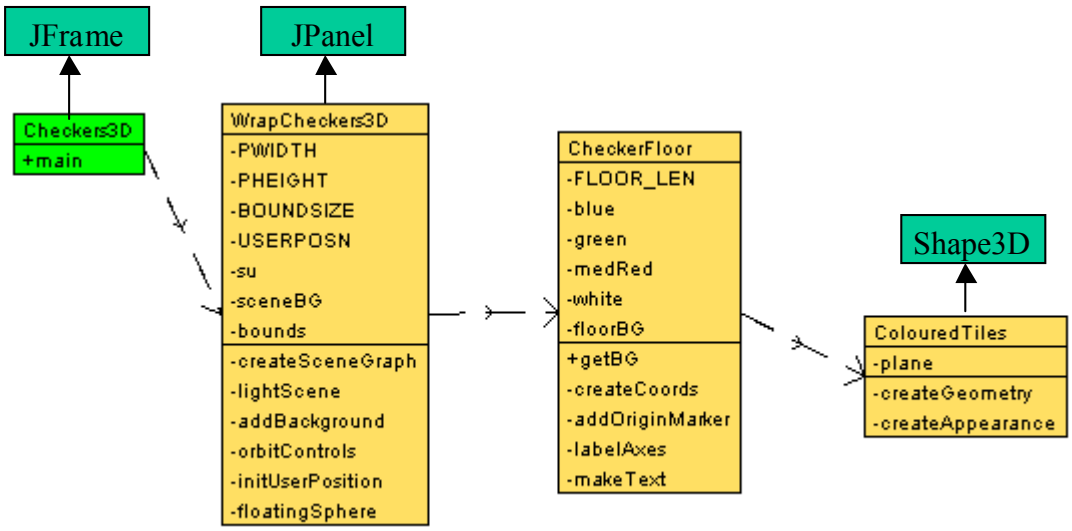


Figure 3. UML Class Diagrams for Checker3D

Checkers3D is the top-level JFrame for the application.

WrapChecker3D is a JPanel holding the scene graph, which is viewable via a Canvas3D object.

CheckerFloor creates the subgraph for the floor (tiles, axes, etc.), with all the same coloured tiles represented by a single ColouredTiles object.

The source code for Checkers3D is in Code/Checkers3D/.

### Integrating Java3D and Swing

Many Java 3D examples use AWT and the nonstandard MainFrame class to link the scene graph with a Java GUI, the main reason being that Canvas3D is a heavyweight GUI element (a thin layer over an OS-generated window). Heavyweight components aren't easily combined with Swing controls, which are lightweight (the control is mostly generated by Java itself). There is a detailed discussion of this problem at j3d.org (http://www.j3d.org/tutorials/quick_fix/swing.html), and our approach is based on their suggestions.

Checkers3D is a JFrame where GUI controls, such as text fields and buttons, would be placed if necessary. In this example, it simply creates an instance of WrapCheckers3D (a JPanel) and places it in the center of a BorderLayout.

```
c.setLayout( new BorderLayout() );
WrapCheckers3D w3d = new WrapCheckers3D();   // panel for 3D canvas
c.add(w3d, BorderLayout.CENTER);
```

The Canvas3D view onto the scene is created inside WrapCheckers3D.

```
  public WrapCheckers3D()
  { setLayout( new BorderLayout() );
    :
    Canvas3D canvas3D = new Canvas3D(...);
    add("Center", canvas3D);
    :
```

Compared to applications in earlier chapters, there is no Timer object to trigger an update/redraw cycle. This is unnecessary because Java 3D contains it's own mechanism for monitoring change in the scene and initiating rendering. The algorithm is something like the following:

```
while(true) {
  process user input;
  if (exit request) break;
  perform behaviours;
  if (scene graph has changed)
    traverse scene graph and render;
}
```

The details are more complicated than this – for example, Java 3D uses multithreading to carry out parallel traversal and rendering, and behaviours can be scheduled.

### Scene Graph Creation

The scene graph is created within the constructor for WrapChecker3D:

```
  public WrapCheckers3D()
  {   :
    GraphicsConfiguration config =
      SimpleUniverse.getPreferredConfiguration();
    Canvas3D canvas3D = new Canvas3D(config);
    add("Center", canvas3D);
    canvas3D.setFocusable(true);     // give focus to the canvas
    canvas3D.requestFocus();

    su = new SimpleUniverse(canvas3D);

    createSceneGraph();
    initUserPosition();        // set user's viewpoint
    orbitControls(canvas3D);   // controls for moving the viewpoint

    su.addBranchGraph( sceneBG );
  }
```

The Canvas3D object is initialised with a configuration obtained from getPreferredConfiguration(): this method queries the hardware for rendering information. Some older Java 3D programs use a `null` configuration, which may lead to rendering errors later.

canvas3D is given focus so that keyboard events will be sent to behaviours in the scene graph. This is not required here, but is good coding practice.

The su SimpleUniverse object creates a standard view branch graph and the Virtual Universe and Locale nodes of the scene graph. createSceneGraph() sets up the lighting, the sky background, the floor, and floating sphere, while initUserPosition()

 

and orbitControls() handle viewer issues. The resulting BranchGroup is added to the scene graph at the end of the method.

createSceneGraph() details:

```
private void createSceneGraph()
{
  sceneBG = new BranchGroup();
  bounds = new BoundingSphere(new Point3d(0,0,0), BOUNDSIZE);

  lightScene();          // add the lights
  addBackground();       // add the sky
  sceneBG.addChild( new CheckerFloor().getBG() );  // add floor

  floatingSphere();      // add the floating sphere

  sceneBG.compile();   // fix the scene
} // end of createSceneGraph()
```

Various methods add subgraphs to sceneBG to build up the content branch graph. sceneBG is compiled once the graph has been finalised, to allow Java 3D to optimise it.

bounds is a global BoundingSphere used to specify the influence of environment nodes for lighting, background, and the OrbitBehavior object. The bounding sphere is placed at the center of the scene, and affects everything within a BOUNDSIZE units radius. Bounding boxes and polytopes are also available in Java 3D.

The scene graph by the end of WrapCheckers3D() is shown in Figure 4.

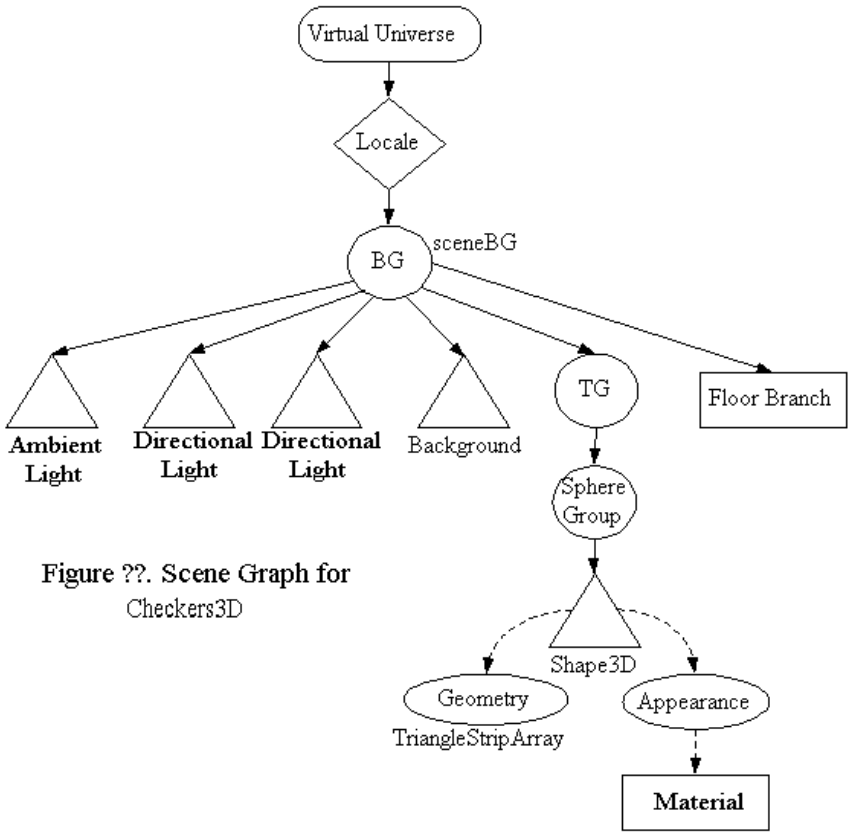


**Figure ??. Scene Graph for** Checkers3D

 

The "Floor Branch" node is my invention, to hide some details until later. Also missing from Figure 4 is the view branch part of the scene graph.

### Lighting

One ambient and two directional lights are added to the scene by lightScene(). An ambient light reaches every corner of the world, illuminating everything equally.

```
Color3f white = new Color3f(1.0f, 1.0f, 1.0f);
// Set up the ambient light
AmbientLight ambientLightNode = new AmbientLight(white);
ambientLightNode.setInfluencingBounds(bounds);
sceneBG.addChild(ambientLightNode);
```

The colour of the light is set, the ambient source is created along with bounds, and added to the scene. The Color3f() constructor takes Red/Green/Blue values between 0.0f and 1.0f (1.0f being 'full on').

A directional light mimics a light from a distant source, hitting the surfaces of objects from a specified direction. The main difference from an ambient light is the requirement for a direction vector.

```
Vector3f light1Direction  = new Vector3f(-1.0f, -1.0f, -1.0f);
   // left, down, backwards
DirectionalLight light1 =
        new DirectionalLight(white, light1Direction);
light1.setInfluencingBounds(bounds);
sceneBG.addChild(light1);
```

The direction can be visualized as the vector between (0,0,0) and (-1,-1,-1), but as multiple parallel lines originating at infinity.

The other forms of lighting are point and spot lights. Point lights position the light in space, emitting in all directions. Spot lights are focussed point lights, aimed in a particular direction. We use spot lights in the Maze3D?? example in chapter ??.

### Backgrounds

A background to a scene can be specified as a constant colour (as here), or a static image, or a texture-mapped geometry such as a sphere. These latter approaches are described in ??.

```
Background back = new Background();
back.setApplicationBounds( bounds );
back.setColor(0.17f, 0.65f, 0.92f);    // sky colour
sceneBG.addChild( back );
```

### Floating Sphere

Sphere is a utility class from the com.sun.j3d.utils.geometry package. It is a subclass of the Primitive class, which is essentially a Group node with a Shape3D child (see Figure 4). Its geometry is given by a TriangleStripArray, used to specify the sphere as

 

connected triangles. We do not have to adjust the geometry, but the sphere's appearance and position do require changes.

The Appearance node is a container for references to a multitude of different kinds of information, including colouring, line, point, polygon, rendering, transparency, and texture attributes.

ColouringAttributes fixes the colour of a shape, and is unaffected by scene lighting. For a shape requiring interaction between colour and light, the Material component is employed. For light to affect a shape's colour, three conditions must be met:

1. The shape's geometry must include normals;
2. The shape's Appearance node must have a Material component;
3. The Material component must have enabled lighting with setLightingEnable().

The utility Sphere class can automatically creates normals, so (1) is easily satisfied.

The Material component controls what colour the shape exhibits when lit by different kinds of lights.

```
Material mat = new Material(ambientColour, emissiveColour,
                      diffuseColour, specularColour, shininess);
```

The ambient colour argument specifies the shape's colour when lit by ambient light: this gives the object a uniform colour.

The emissive colour contributes the colour that the shape produces itself (as for a light bulb); frequently, this argument is set to black (off).

The diffuse colour is the colour of the object when lit, with its intensity depending on the angle the light beams make with the shape's surface. Often the diffuse and ambient colours are set to be the same.

The intensity of the specular colour parameter is related to how much the shape reflects from it's shiny areas. This is combined with the shininess argument which controls the size of the reflective highlights. Often the specular colour is set to be white.

In Checkers3D there are two directional lights, which create two shiny patches on the top of the floating sphere (see Figures 1 and 2). As explained later, the floor tiles are unlit, since their colour is set in the shape's geometry.

The code in floatingSphere() which handles the sphere's appearance:

```
Color3f black = new Color3f(0.0f, 0.0f, 0.0f);
Color3f blue = new Color3f(0.3f, 0.3f, 0.8f);
Color3f specular = new Color3f(0.9f, 0.9f, 0.9f); // near white

Material blueMat=
          new Material(blue, black, blue, specular, 25.0f);
blueMat.setLightingEnable(true);

Appearance blueApp = new Appearance();
blueApp.setMaterial(blueMat);
```

Positioning a shape is almost always done by placing it below a TransformGroup (see the Sphere Group in Figure 4). A TransformGroup can be used to position, rotate, and

 

scale the nodes which lie beneath it, with the transformations defined with Transform3D objects.

```
Transform3D t3d = new Transform3D();
t3d.set( new Vector3f(0,4,0));     // place at (0,4,0)
TransformGroup tg = new TransformGroup(t3d);
tg.addChild(new Sphere(2.0f, blueApp));
        // set the sphere's radius and appearance
        // and its normals by default
sceneBG.addChild(tg);
```

The set() method positions the sphere's center at (0,4,0), and resets any previous rotations or scalings. set() can also be used to scale and rotate, while reseting the other transformations. The methods setTranslation(), setScale() and setRotation() only affect the given transformation.

## Coordinates in Java 3D

Unlike some 3D drawing packages, the Y-axis in Java 3D is in the vertical direction, while the 'ground' is defined by the XZ plane, as shown in Figure 5.

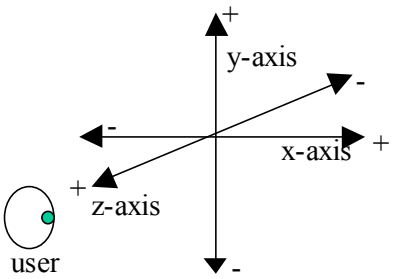


Figure 5. Axes in Java 3D

The position of the sphere is Checkers3D is set to be (0,4,0) which places it's center 4 units above the XZ plane.

## The Floor

The floor in Checkers3D is made up of tiles created with the ColouredTiles class, and axis labels made with the Text2D utility class. Figure 6 shows the 'floor branch' which was hidden in figure 4.

 

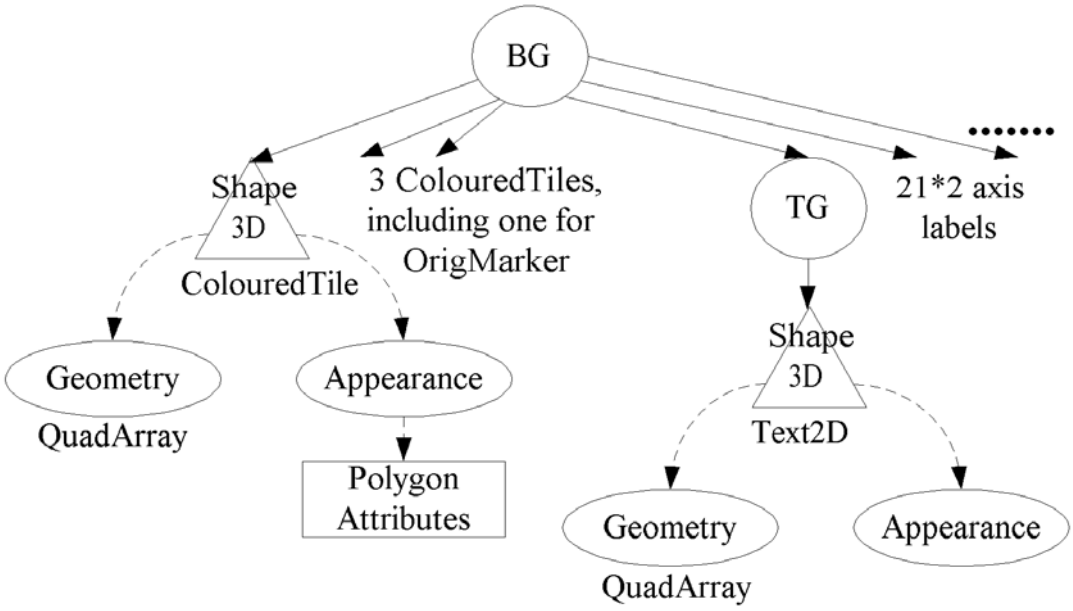


Figure 6. Floor Branch of the Scene Graph for Checkers3D

The floor subgraph is constructed with an instance of the CheckerFloor class, and made available via the getBG() method.

```
sceneBG.addChild( new CheckerFloor().getBG() );  // add the floor
```

The CheckerFloor() constructor uses nested for-loops to initialise two ArrayLists: the blueCoords list contains all the coordinates for the blue tiles, and greenCoords holds the coordinates for the green tiles. Once the ArrayLists are completed, they are passed to ColouredTiles objects, along with the colour that should be used to render the tiles. A ColouredTiles object is a subclass of Shape3D, so can be added directly to the floor's graph:

```
floorBG.addChild( new ColouredTiles(blueCoords, blue) );
floorBG.addChild( new ColouredTiles(greenCoords, green) );
```

The red square at the origin (most clearly seen in Figure 2) is made in a similar way:

```
Point3f p1 = new Point3f(-0.25f, 0.01f, 0.25f);
Point3f p2 = new Point3f(0.25f, 0.01f, 0.25f);
Point3f p3 = new Point3f(0.25f, 0.01f, -0.25f);
Point3f p4 = new Point3f(-0.25f, 0.01f, -0.25f);

ArrayList oCoords = new ArrayList();
oCoords.add(p1); oCoords.add(p2);
oCoords.add(p3); oCoords.add(p4);

floorBG.addChild( new ColouredTiles(oCoords, medRed) );
```

The red square is centered at (0,0) on the XZ plane, and raised a little above the Y-axis (0.01 units) so that it is visible above the tiles. Each side of the square is of length 0.5 units. Note that the four Point3f points in the ArrayList are stored in a counter-clockwise order. This is also true for each group of four points in blueCoords and greenCoords.

 

It helps to consider Figure 7 to see the ordering of the square's points.

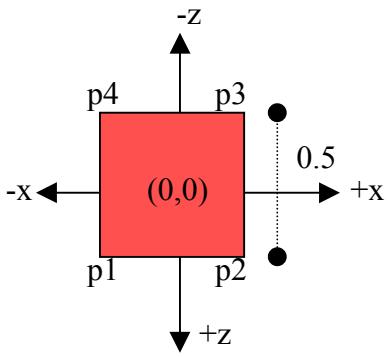


Figure 7. The OrigMarker, viewed from above.

### The ColouredTiles Class

The ColouredTiles class extends Shape3D, and defines the geometry and appearance of tiles with the same colour. The geometry uses a QuadArray to represent the tiles as a series of quadlilaterals. The constructor is:

```
QuadArray(int vertexCount, int vertexFormat);
```

The vertex format is an OR'd collection of static integers which specify the different aspects of the quad to be initialised later, such as its coordinates, colour, and normals. In ColouredTiles, the QuadArray plane is created using:

```
plane = new QuadArray(coords.size(),
        GeometryArray.COORDINATES | GeometryArray.COLOR_3 );
```

The size() method returns the number of coordinates in the supplied ArrayList. The coordinate and colour data is supplied in createGeometry():

```
int numPoints = coords.size();
Point3f[] points = new Point3f[numPoints];
coords.toArray( points );   // ArrayList-->array
plane.setCoordinates(0, points);

Color3f cols[] = new Color3f[numPoints];
for(int i=0; i < numPoints; i++)
  cols[i] = col;
plane.setColors(0, cols);
```

The order in which a quad's coordinates are specified is significant: the front of a polygon is the face where the vertices form a counter-clockwise loop. Knowing front from back is important for lighting and hidden face culling and, by default, only the front face of a polygon will be visible in a scene. Our tiles are oriented so their fronts are facing upwards along the y-axis.

It is also necessary to make sure that the points of each quad from a convex, planar polygon, or rendering may be compromised. However, each quad in the coordinates array does not need to be connected or adjacent to others, which is the case for our tiles.

Since the geometry has been given no information about its normals, a Material node component cannot be used to specify its colour when lit. Instead we could use a

 

ColoringAttributes, but a third alternative is to set the colour in the geometry, as done here. This colour will be constant, unaffected by the scene lighting.

Once finalised, the Shape3D's geometry is set with:

```
setGeometry(plane);
```

The shape's appearance is handled by createAppearance() which uses a PolygonAttribute component to switch off the culling of the back face. PolygonAttribute can also be employed to render polygons in point or line form (i.e. as wire frames), and to flip normals of back facing shapes.

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

Once the appearance has been fully specified, it is fixed in the shape with:

```
setAppearance(app);
```

**The Axes**

The floor's axis labels are generated with the labelAxes() and makeText() methods in CheckerFloor(). labelAxes() uses two loops to create labels along the x- and z- axes. Each label is constructed by makeText(), and then added to the floor's BranchGroup (see Figure 6):

```
floorBG.addChild( makeText(pt,""+i) );
```

makeText() uses the Text2D utility class to create a 2D string of a specified colour, font, point size, and font style:

```
Text2D message = new Text2D(text, white, "SansSerif",
        36, Font.BOLD );    // 36 point bold Sans Serif
```

A Text2D object is a Shape3D object with a quad geometry (a rectangle), and appearance given by a texture map (image) of the string, placed on the front face. By default, the back face is culled, so if the user moves behind an axis label it becomes invisible.

The point size is converted to virtual world units by dividing by 256. Generally, it is not a good idea to use too large a point size in the Text2D() constructor since the text may be rendered incorrectly. Instead, a TransformGroup should be placed above the shape and used to scale it to the necessary size.

The positioning of each label is done by a TransformGroup above the shape:

```
TransformGroup tg = new TransformGroup();
Transform3D t3d = new Transform3D();
t3d.setTranslation(vertex);   // the position for the label
tg.setTransform(t3d);
tg.addChild(message);
```

setTranslation() only affects the position of the shape. The tg TransformGroup is added to the floor scene graph.

 

### Viewer Positioning

The scene graph in Figure 4 does not include the view branch graph, which appears in Figure 8 below.

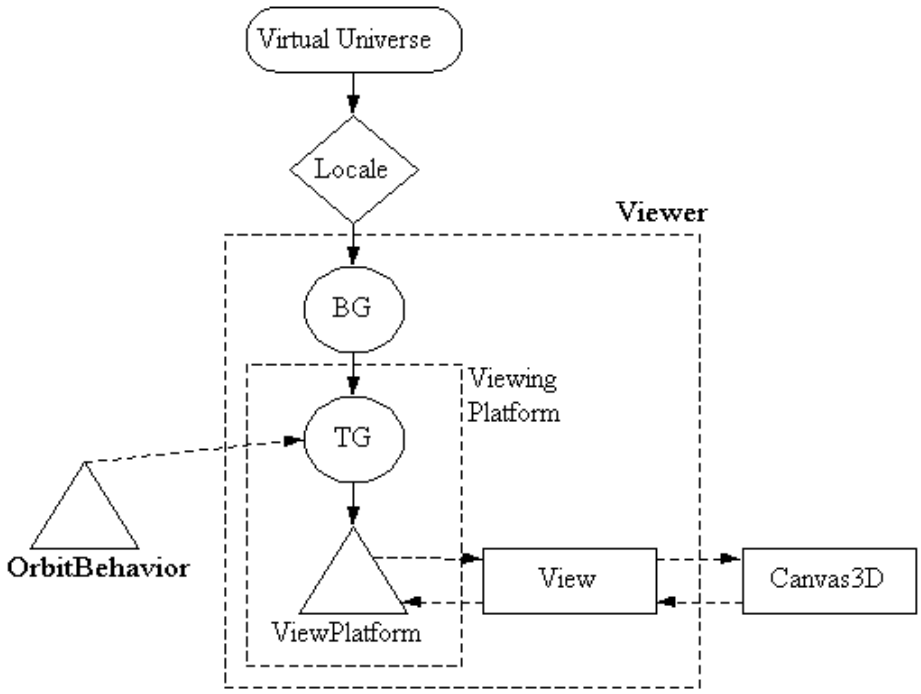


Figure ??. The View Branch Graph for Checkers3D

It is created by a call to the SimpleUniverse constructor in the WrapChecker3D() constructor:

```
su = new SimpleUniverse(canvas3D);
```

SimpleUniverse offers 'simplified' access to the view branch graph via the ViewingPlatform and Viewer classes, which are mapped to the graph as shown in Figure 8.

ViewingPlatform is used in initUserPosition() to access the TransformGroup above the ViewPlatform node:

```
ViewingPlatform vp = su.getViewingPlatform();
TransformGroup steerTG = vp.getViewPlatformTransform();
```

steerTG corresponds to TG in Figure 8. Then its Transform3D component is extracted and changed with the lookAt() and invert() methods:

```
Transform3D t3d = new Transform3D();
steerTG.getTransform(t3d);

t3d.lookAt( USERPOSN, new Point3d(0,0,0), new Vector3d(0,1,0));
t3d.invert();

steerTG.setTransform(t3d);
```

 

lookAt() is a convenient way to set the viewer's position in the virtual world. The method requires the viewer's intended position, the point which he is looking at, and a vector specifying the upward direction. In our code, the viewer's position is USERPOSN (the (0,5,20) coordinate), he is looking towards the origin (0,0,0), and 'up' is along the positive y-axis. This is shown in Figure 9.

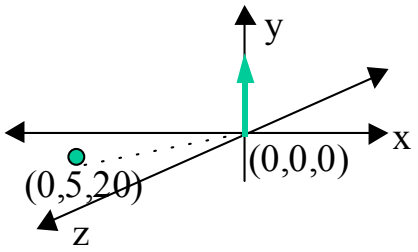


Figure 9. lookAt() Graphically.

invert() is required since the position is relative to the viewer rather than an object in the scene.

### Viewer Movement

The user is able to move through the scene by utilising the utility OrbitBehavior behaviour class sin the view graph. A combination of control keys and mouse button presses move and rotates ('orbits') the viewer's position.

The behaviour is set up in orbitControls() in WrapCheckers3D:

```
OrbitBehavior orbit =
        new OrbitBehavior(c, OrbitBehavior.REVERSE_ALL);
orbit.setSchedulingBounds(bounds);
ViewingPlatform vp = su.getViewingPlatform();
vp.setViewPlatformBehavior(orbit);
```

The REVERSE_ALL flag makes the viewpoint move in the same direction as the mouse move. There are numerous other flags and methods for affecting the rotation, translation, and zooming characteristics.

MouseRotate, MouseTranslate, and MouseZoom are similar behaviour classes which appear in many Java 3D examples; their principal difference from OrbitBehavior is that they affect the objects in the scene rather than the viewer.

Most games, such as first person shooters, require greater control over the viewer's movements than these utility behaviours can offer, so we will be implementing our own behaviours in later chapters.

### Viewing the Scene Graph

This chapter used scene graphs to illustrate the coding techniques, and scene graphs are a generally useful way of understanding (and checking) code.

 

I received help with my drawings by using Daniel Selman's Java3dTree package. It creates a JFrame holding a textual tree representation of the scene graph (see Figure 10).

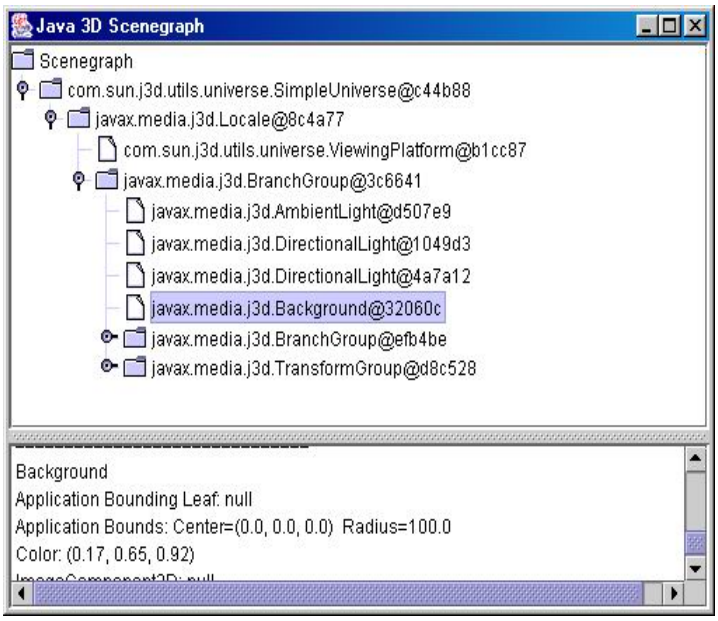


Figure 10. Java3dTree Representation of the Checkers3D Scene Graph

The tree (a JTree object) is initially minimized, and branches can be examined by clicking on the subfolder icons. Information about the currently selected node appears in the bottom window.

The package is available in j3dtree.jar, part of the source code downloadable from http://www.manning.com/selman/ for Selman's *Java 3D Programming* text.

Augmenting code to generate the JTree is quite simple. WrapCheckers3D must import the j3dtree package, and declare a global variable for the JFrame tree display:

```
import com.sun.j3d.utils.behaviors.vp.*;
  :
private Java3dTree j3dTree;
```

The WrapCheckers3D() constructor must create the j3dTree variable:

```
j3dTree = new Java3dTree();
```

After the scene graph has been completed (i.e. at the end of the constructor), the tree display can be built with:

```
j3dTree.updateNodes( su );
```

However, prior to building, the capabilities of the scene graph nodes must be adjusted with:

```
j3dTree.recursiveApplyCapability( sceneBG );
```

This must be done after the content branch group (sceneBG) has been completed, but before it is compiled (or made live).

Unfortunately, we cannot write:

```
j3dTree.recursiveApplyCapability( su );
```

 

without generating errors, because the SimpleUniverse() constructor makes the ViewingPlatform live, which prevents further changes to its capabilities.

Since only the capabilities in the content branch have been adjusted, the call to updateNodes() will generate some warning messages when the view branch below the Locale node is encountered.

Compilation and execution must include j3dtree.jar in the classpath. My preferred approach is to do this via command line arguments:

```
javac -classpath %CLASSPATH%;j3dtree.jar *.java
java -cp %CLASSPATH%;j3dtree.jar Checkers3D
```

If repetitive typing isn't to your taste, command lines like these can be hidden inside DOS batch files or shell scripts.

The Java3dTree object is a textual representation of the scene, which means that I had to draw the scene graph myself. But the plus side is that tree generation has negligible impact on the rest of the program.

Another solution is to use the Java 3D Scene Graph Editor (http://java3d.netbeans.org/j3deditor_intro.html). This displays a graphical version of the scene graph, but has the downsides that its installation and usage are rather complicated, and the memory requirements may be overly severe on some machines.

 