Chapter 26.5 (N8). More Backgrounds and Overlays

In chapter N7, I described several ways of making detailed backgrounds which look different when viewed from different places in the scene. The techniques all utilize textures wrapped over 3D geometries connected to the scene graph.

This chaper considers a different approach: the background is represented by a standard BufferedImage, which is drawn separately from the rest of the scene graph. This means that a large variety of image processing techniques become available for modifying the graphic at run time.

The approach uses *mixed mode rendering*, where most of the scene is built using a scene graph, but the background is created outside the scene graph structure.

Mixed mode rendering also makes it very easy to create *overlays*: semi-transparent areas on screen where text or graphics can be drawn in the foreground without obscuring the gameplay underneath. I'll describe how to draw overlays composed from text and images.

There are three examples in this chapter:

1. A scene made up of three 'planets' (actually textured spheres) with a static background of stars. This shows the basic technique for background creation with mixed mode rendering.
2. The same three planets scene, but with a rotating background of stars. The background image is rotated by a separate thread, using an affine transformation. This illustrates how to manipulate the background at run time.
3. A variant of the 3D model application, ObjView3D, from Chapter N7. When the user turns left or right, the background is shifted by a small amount as well. Overlays are part of this example.

1. Retained, Immediate, and Mixed Modes

I've being using Java 3D's *retained mode* in all my 3D examples up until now. The scene is constructed by adding nodes to a scene graph, and the responsibility for rendering, and optimizing, the scene is left to Java 3D.

Immediate mode discards the scene graph, placing all the scene creation and rendering tasks into the hands of the programmer. To be precise, only the content branch part of the scene graph is jettisoned in immediate mode; the view branch is still present, to manage Java 3D's interface with the enclosing Java application or applet.

The middle path is to keep the scene graph for most 3D elements, but to build a few visual components outside the graph when necessary. This *mixed mode* approach is employed in this chapter to implement backgrounds and overlays.

 ©Andrew Davison 2006

The immediate and mixed modes are utilized in the same way, by subclassing Canvas3D. Canvas3D has four methods that are called at specific times during each rendering cycle:

- preRender(): called after the canvas has been cleared but before any rendering has been done for the current frame;
- renderField(): carries out scene rendering. For a desktop application, the screen is the 'field'. However, there may be multiple fields in more exotic configurations (e.g. two for stereoscopic displays).
- postRender(): called after all the rendering has been completed for the current frame, but before the buffer containing the frame has been made visible to the user;
- postSwap(): called after the frame has been made visible.

A background image is drawn into the frame by preRender(). The image is added to the frame *before* any retained mode rendering, and so appears at the back of the scene.

An overlay image is drawn into the frame by postRender(). Since the picture is added to the frame *after* the retained mode rendering, it appears at the front of the scene.

 ©Andrew Davison 2006

2. The Earth, Moon, and Mars

Figure 1 shows the Planets3D application, with a stars image as the background.

Figure 1. Three Planets and Stars.

Mouse and control key combinations permit the camera to be translated and rotated through the scene (by utilizing Java 3D's OrbitBehavior). Unfortunately, no matter how much the camera is moved or turned, the stars background never changes.

Class diagrams for Planet3D are given in Figure 2, with only the public methods listed.

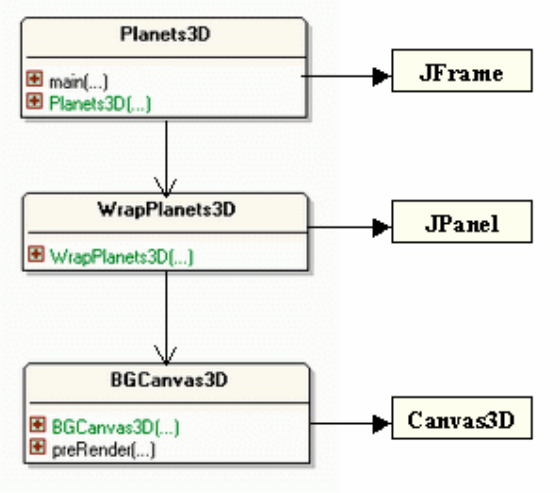

Figure 2. Class Diagrams for Planets3D.

The Planets3D class sets up the JFrame, and calls WrapPlanets3D to create the scene (the three planets, the lighting, and the OrbitBehavior). The background image is added to the scene by preRender() inside BGCanvas3D.

©Andrew Davison 2006

Figure 3 shows the content branch part of the scene graph constructed by WrapPlanets3D.

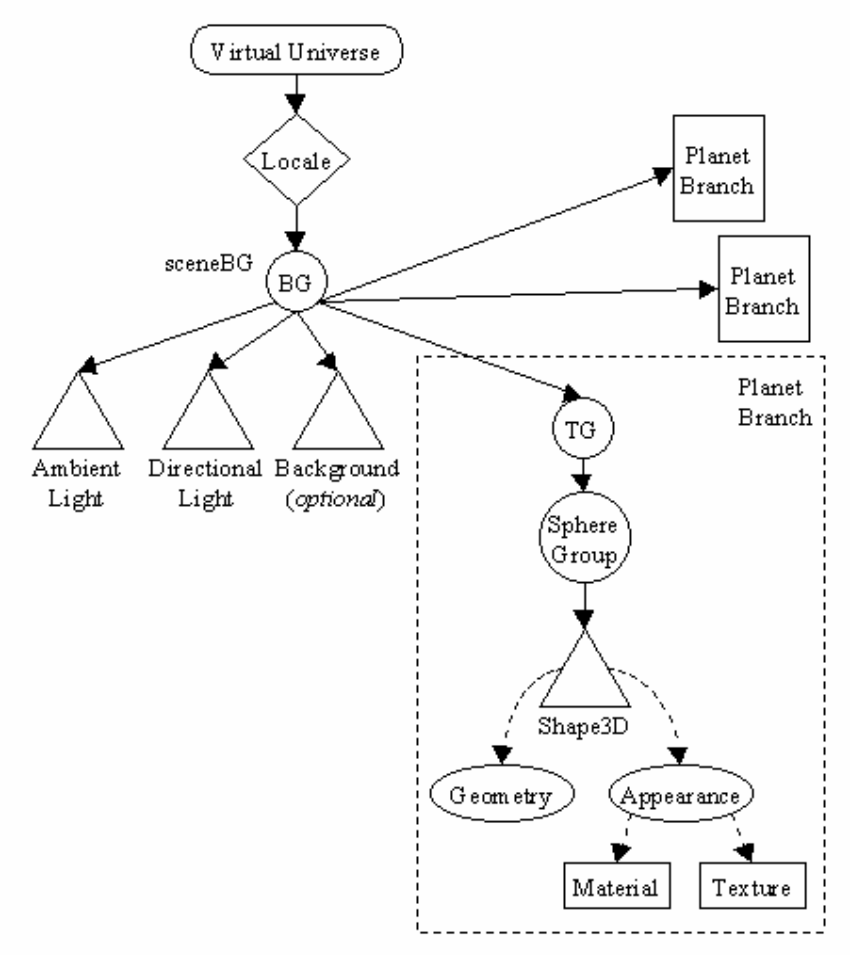

Figure 3. The Content Branch for Planets3D.

Each 'planet' is a Java 3D Sphere, with a blended material and texture appearance, located at a fixed position in the scene. The planets don't rotate.

The OrbitBehavior is attached to the viewing platform in the scene graph's view branch, so is not shown in Figure 3.

The Background node is an optional component. It'll become useful when the background image is smaller than the canvas, a scenario I'll discuss later.

2.1. Building the Scene

The WrapPlanets3D() constructor creates the 3D canvas in the standard way, and attaches a SimpleUniverse to it:

```
// globals
private static final int PWIDTH = 512;   // size of panel
private static final int PHEIGHT = 512;
```

 ©Andrew Davison 2006

```
private SimpleUniverse su;


public WrapPlanets3D(String backFnm)
// A panel holding a 3D canvas
{
  setLayout( new BorderLayout() );
  setOpaque( false );
  setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

  GraphicsConfiguration config =
      SimpleUniverse.getPreferredConfiguration();

  // Canvas3D canvas3D = new Canvas3D(config);
  BGCanvas3D canvas3D = new BGCanvas3D(config, backFnm);

  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 );
} // end of WrapPlanets3D()
```

The usual Canvas3D object has been replaced by a BGCanvas3D instance, which takes the background image filename (backFnm) as an argument. Since BGCanvas3D is a subclass of Canvas3D, it can be used in the same way as Canvas3D.

The panel holding the canvas is set to be 512x512 pixels. The background image must have (at least) these dimensions in order to fill the canvas.

createSceneGraph() constructs the lights, the background node, and the planets.

```
//globals
private static final int BOUNDSIZE = 100;  // larger than world

private BranchGroup sceneBG;
private BoundingSphere bounds;   // for environment nodes


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

  lightScene();      // add the light
  addBackground();   // add the sky (optional)

  // add planets
  addPlanet("earth.jpg", 3.0f, new Vector3f(-2,0,0));
  addPlanet("moon.jpg", 0.8f, new Vector3f(4,-1,0));
  addPlanet("mars.jpg", 2.0f, new Vector3f(2,3,-15));

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

 ©Andrew Davison 2006

The lighting consists of a single ambient light and a single directional light.

The background colour is sky blue, and isn't really needed in this example since the background image fills the canvas. However, it'll become useful later:

```
private void addBackground()
{ Background back = new Background();
  back.setApplicationBounds( bounds );
  back.setColor(0.17f, 0.65f, 0.92f);    // sky colour
  sceneBG.addChild( back );
}  // end of addBackground()
```

addPlanet() builds the scene graph branch labeled as "Planet Branch" in Figure 3. A Sphere instance is created with a blended material and texture appearance, and positioned in the scene using a TransformGroup:

```
// globals
private static final Color3f BLACK = new Color3f(0.0f, 0.0f, 0.0f);
private static final Color3f GRAY = new Color3f(0.6f, 0.6f, 0.6f);
private static final Color3f WHITE = new Color3f(0.9f, 0.9f, 0.9f);


private void addPlanet(String texFnm, float radius, Vector3f posVec)
{
  Appearance app = new Appearance();

  // combine texture with material and lighting of underlying surface
  TextureAttributes ta = new TextureAttributes();
  ta.setTextureMode( TextureAttributes.MODULATE );
  app.setTextureAttributes( ta );

  // a gray material with lighting
  Material mat = new Material(GRAY, BLACK, GRAY, WHITE, 25.0f);
     // sets ambient, emissive, diffuse, specular, shininess
  mat.setLightingEnable(true);
  app.setMaterial(mat);

  // apply texture to shape
  Texture2D texture = loadTexture("images/" + texFnm);
  if (texture != null)
    app.setTexture(texture);

  // make the sphere with normals for lighting, and texture support
  Sphere globe = new Sphere(radius,
                        Sphere.GENERATE_NORMALS |
                        Sphere.GENERATE_TEXTURE_COORDS,
                        15, app);   // mesh division is 15

  // position the sphere
  Transform3D t3d = new Transform3D();
  t3d.set(posVec);
  TransformGroup tg = new TransformGroup(t3d);
  tg.addChild(globe);

  sceneBG.addChild(tg);
}  // end of addPlanet()
```

 ©Andrew Davison 2006

The gray material is enabled for lighting, so the ambient and directional light will affect the sphere's coloration.

Java 3D's Sphere class supports the automatic generation of normals and texture coordinates. Normals are required for the lighting effects, and texture coordinates are utilized to wrap the texture around the sphere. Similar support is offered by the other Java 3D geometry classes: Box, Cone, and Cylinder.

The divisions argument for Sphere() specify the sphere's tessellation: increasing the number of divisions makes the sphere smoother, but at the cost of increased rendering time.

The texture is loaded from the images/ subdirectory. The image used for the earth is shown in Figure 4.

Figure 4. The Earth Texture.

2.2. The Background

The BGCanvas class extends Canvas3D in order to redefine preRender() (which does nothing in Canvas3D).

The constructor loads the specified background image:

```
// globals
private BufferedImage backIm;    // the background image
private J3DGraphics2D render2D;  // for 2D rendering into 3D canvas


public BGCanvas3D(GraphicsConfiguration gc, String backFnm)
{ super(gc);
  backIm = loadImage(backFnm);
  render2D = this.getGraphics2D();
}
```

The canvas is configured by calling super() with the GraphicsConfiguration object.

render2D is a J3DGraphics2D object, which provides several methods for carrying out 2D rendering on a 3D canvas.

loadImage() attempts to load a picture from the images/ subdirectory:

```
private BufferedImage loadImage(String fnm)
{
  BufferedImage im = null;
  try {
    im = ImageIO.read( getClass().getResource("images/" + fnm));
    System.out.println("Loaded Background: images/" + fnm);
  }
```

 ©Andrew Davison 2006

```
  catch(Exception e)
  { System.out.println("Could not load background: images/" + fnm); }

  return im;
}  // end of loadImage()
```

preRender() is called in every cycle, prior to the scene graph's rendering. It draws the image using J3DGraphics2D.drawAndFlushImage().

```
public void preRender()
{  render2D.drawAndFlushImage(backIm, 0, 0, this);  }
```

The second and third arguments are the (x, y) coordinate in the canvas where the top-left corner of the image is positioned.

2.3. Some Variations on a Theme

If Planets3D is called with sun.gif as the background image, then the scene is displayed as shown in Figure 5.

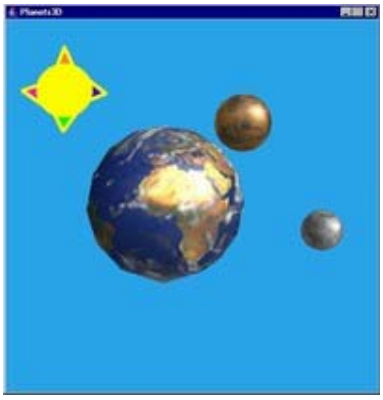

Figure 5. Planets with a Sunny Background.

The sun is a transparent GIF, shown in Figure 6.

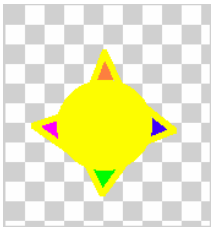

Figure 6. The Sun.

The gray and white squares indicate that the background of the image is transparent.

The blue background in Figure 5 is generated by the Background node, which highlights a rather surprising rendering order. Although the call to preRender()

 ©Andrew Davison 2006

precedes scene graph rendering, the scene graph's Background node is placed behind the preRender() image.

This ordering offers the possibility of optimizing background generation, since it's no longer necessary for the background picture to be the same size as the 3D canvas; any 'gaps' can be filled with the colour supplied by a Background node.

Figure 7 shows the contents of galaxy.jpg.

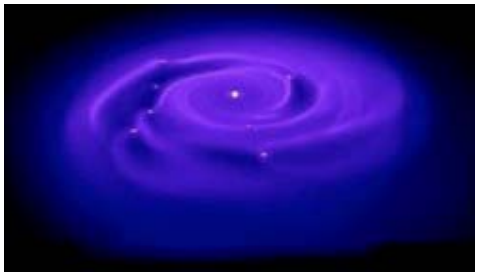

Figure 7. The Galaxy.

This image is part of the scene in Figure 8.

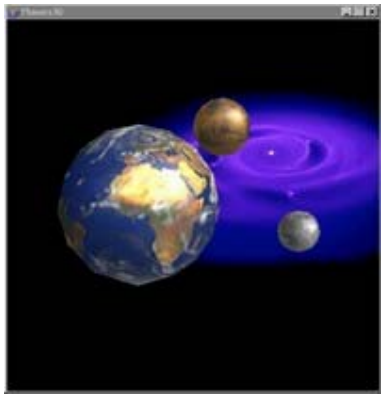

Figure 8. Planets in a Galaxy.

The easiest way of creating a black background is not to add a Background node to the scene at all; black is the default background colour. The call to addBackground() can be commented out in WrapPlanets3D.

Figure 8 also shows that I've repositioned the image. I changed the (x, y) coordinates supplied to drawAndFlushImage() in preRender():

```
public void preRender()
{  render2D.drawAndFlushImage(backIm, 150, 100, this);  }
```

3. Spinning the Background

A major benefit of rendering the background using a subclass of Canvas3D is that the image is a BufferedImage, rather than a Java 3D Background node. This makes it easier to apply Java 2D effects to the image at run time. I discussed a lot of these

 ©Andrew Davison 2006

kinds of effects in Chapter 6, including rotation, fading, scaling, and various colour adjustments.

In this section, I'll describe how to rotate the background image. Figure 9 shows two screenshots of the SpinPlanets3D application taken a few seconds apart.

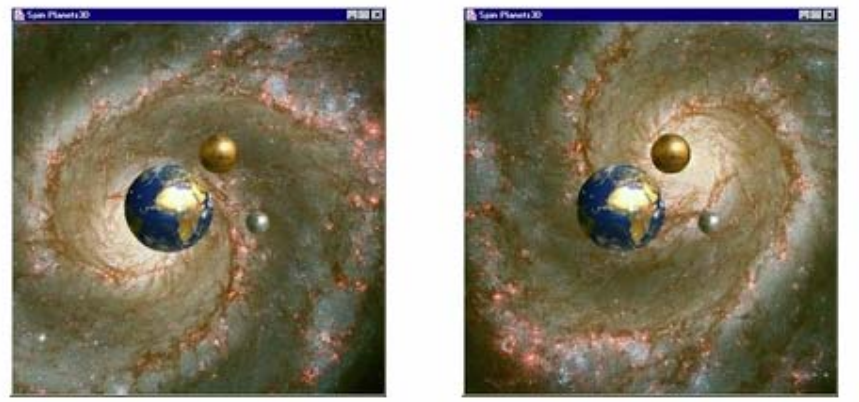

Figure 9. The Whirlpool of Space.

The background image is rotated around its center by 2 degrees in a clockwise direction every 200 ms. The image is drawn onto the canvas so its center is located at the canvas's center.

The class diagrams for SpinPlanets3D are shown in Figure 10; only the public methods are listed.

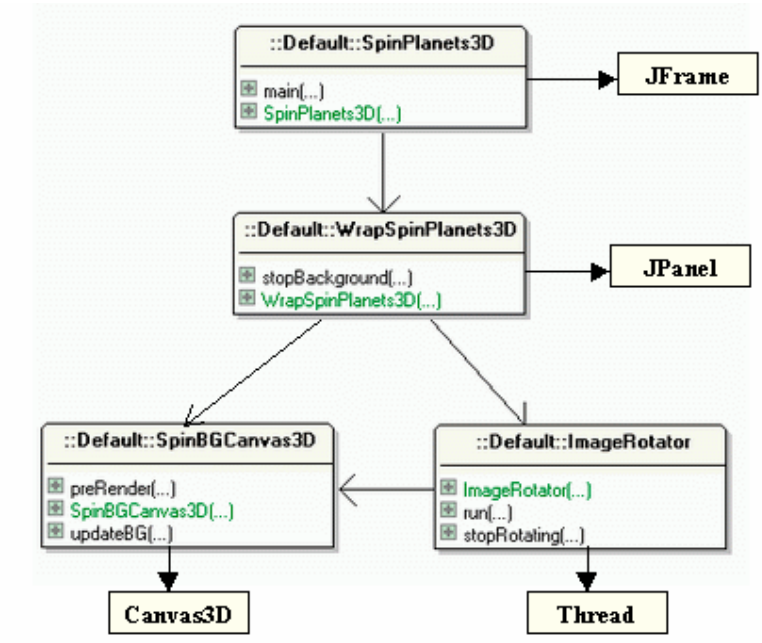

Figure 10. Class Diagrams for SpinPlanets3D.

SpinPlanets3D creates the top-level JFrame, and uses WrapSpinPlanets3D to create the scene graph. The ImageRotator thread loads the background image, and applies

 ©Andrew Davison 2006

the rotation operation at 200 ms intervals. ImageRotator updates SpinBGCanvas3D's image by calling updateBG().

SpinBGCanvas3D's preRender() is called by Java3D's rendering thread to draw the background.

The scene graph built by WrapSpinPlanets3D is virtually identical to the one made by WrapPlanets3D in Figure 3. It's a little simpler since no Background node is added to the graph. Black is a fine substitute when the background image has transparent elements, or is too small to fill the canvas.

3.1. Building the Scene, and Terminating

WrapSpinPlanets3D's constructor is similar to the constructor in WrapPlanets3D.

```
// globals
private static final int PWIDTH = 512;   // size of panel
private static final int PHEIGHT = 512;

private ImageRotator imRotator; // for rotating the background image


public WrapSpinPlanets3D(String backFnm)
{
  setLayout( new BorderLayout() );
  setOpaque( false );
  setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

  GraphicsConfiguration config =
         SimpleUniverse.getPreferredConfiguration();
  SpinBGCanvas3D canvas3D =
         new SpinBGCanvas3D(config, PWIDTH, PHEIGHT);
  add("Center", canvas3D);
  canvas3D.setFocusable(true);     // give focus to the canvas
  canvas3D.requestFocus();

  imRotator = new ImageRotator(backFnm, canvas3D);

  su = new SimpleUniverse(canvas3D);
  createSceneGraph();
  initUserPosition();        // set user's viewpoint
  orbitControls(canvas3D);   // controls for moving the viewpoint
  su.addBranchGraph( sceneBG );

  imRotator.start();   // start the background rotating
} // end of WrapSpinPlanets3D()
```

The ImageRotator thread, imRotator, rotates the background visual and passes the result to SpinBGCanvas3D for rendering. Therefore, it's constructor requires the image's filename and a SpinBGCanvas3D reference as arguments. The thread is started once the scene has been constructed.

The thread is terminated when the application is about to finish. A WindowListener in SpinPlanets3D is triggered by a window closing event, and calls WrapSpinPlanets3D's stopBackground():

 ©Andrew Davison 2006

```
public void stopBackground()
{  imRotator.stopRotating();  }
```

3.2. Rotating the Image

The ImageRotator constructor loads the image, and stores the graphics configuration to speed up the rotation operations applied later.

```
// globals
private static final int ROT_AMT =  2;   // rotation, in degrees

private SpinBGCanvas3D canvas3D;
private BufferedImage backIm;
private GraphicsConfiguration gc;
private int currAngle;


public ImageRotator(String backFnm, SpinBGCanvas3D c3d)
{
  canvas3D = c3d;

  backIm = loadBGImage(backFnm);
  canvas3D.updateBG(backIm);
     // initially use the unrotated image as the background
  currAngle = ROT_AMT;

  /* get GraphicsConfiguration so images can be copied
     easily and efficiently */
  GraphicsEnvironment ge =
       GraphicsEnvironment.getLocalGraphicsEnvironment();
  gc = ge.getDefaultScreenDevice().getDefaultConfiguration();
} // end of ImageRotator()
```

The original image, stored in backIm, is passed to the canvas by calling SpinBGCanvas3D.updateBG(). This gives the canvas something to draw at start-up.

A while loop in run() rotates the picture, passes the result to the canvas, and sleeps for a short time before repeating.

```
// globals
private static final int DELAY = 200;    // ms (update interval)

private boolean isRunning;

public void run()
{
  isRunning = true;
  BufferedImage rotIm;

  while(isRunning) {
    rotIm = getRotatedImage(backIm, currAngle);
    canvas3D.updateBG(rotIm);  // pass image to canvas
```

 ©Andrew Davison 2006

```
    currAngle += ROT_AMT;
    if (currAngle > 360)   // reset after one complete rotation
      currAngle -= 360;

    try {
      Thread.sleep(DELAY);      // wait a while
    }
    catch (Exception ex) {}
  }
} // end of run()
```

The rotation angle in currAngle is incremented in ROT_AMT steps, modulo 360.

3.3. Manipulating the Image

getRotatedImage() is virtually identical to the same-named method from Chapter 6. Given a BufferedImage and an integer angle, it creates a new BufferedImage rotated by that number of degrees in a clockwise direction. The image's center is used as the center of rotation.

```
private BufferedImage getRotatedImage(BufferedImage src, int angle)
{
  if (src == null)
    return null;

  int transparency = src.getColorModel().getTransparency();
  BufferedImage dest =  gc.createCompatibleImage(
                    src.getWidth(), src.getHeight(), transparency);
  Graphics2D g2d = dest.createGraphics();

  AffineTransform origAT = g2d.getTransform();
                                    // save original transform

  // rotate the coord. system of the dest. image around its center
  AffineTransform rot = new AffineTransform();
  rot.rotate( Math.toRadians(angle), src.getWidth()/2,
                                    src.getHeight()/2);
  g2d.transform(rot);

  g2d.drawImage(src, 0, 0, null);   // copy in the image

  g2d.setTransform(origAT);    // restore original transform
  g2d.dispose();

  return dest;
} // end of getRotatedImage()
```

 ©Andrew Davison 2006

3.4. Rotation and Clipping

A tricky aspect of getRotatedImage() is that the affine transformation employs the original image as a clipping rectangle. If the rotation takes the picture outside those bounds then it'll be clipped, as illustrated by Figure 11.

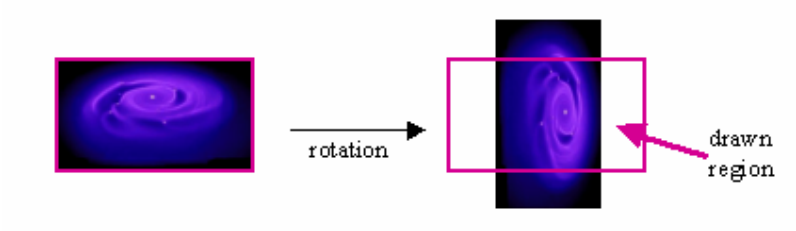

Figure 11. A Rotation that Clips the Image.

In Figure 11, the top and bottom parts of the rotated image are clipped. Also, the two vertical white gaps in the drawn region will be rendered transparent if the original picture has an alpha channel, or in black if the image is opaque.

The simplest way to avoid clipping important parts of the image is to give the image a large enough border so that the significant visuals reside within the drawn region. If the picture is square, then all its important details should lie within a 'rotation' circle defined by the dimensions of the square. This idea is shown in Figure 12.

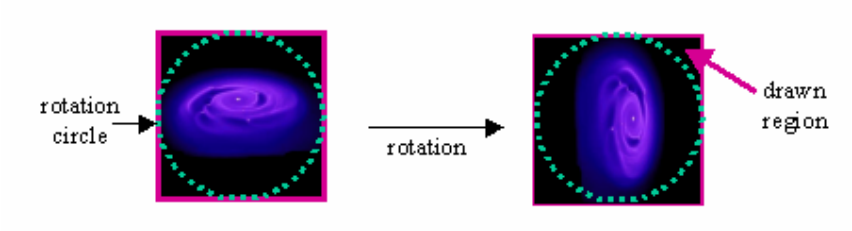

Figure 12. Details Inside the Rotation Circle will not be Clipped.

Visuals within the rotation circle won't be clipped when a rotation is applied around the image's center.

3.5. Avoiding Gaps in the Rotated Image

Another issue with the rotation operation is how large the picture should be so there are no gaps in the rotated version (e.g. how do we avoid the gaps in Figure 11)? A square is a good shape for a centrally rotated image, but how big should the square be to always fill the canvas no matter what the rotation?

©Andrew Davison 2006

When a square image of size len is rotated around its center, it'll mark out a rotation circle defined by the length of its diagonal, as shown on the left side of Figure 13.

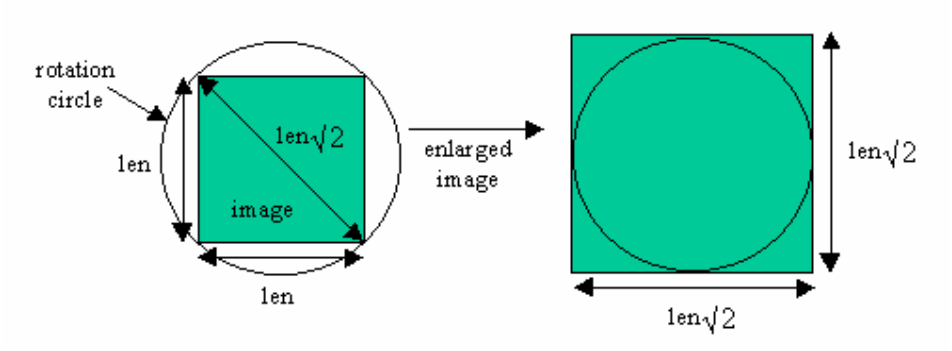

Figure 13. Enlarging an Image to fit a Rotation Circle.

When the image is rotated inside a canvas with the same len-by-len dimensions then gaps will appear. The only way to avoid this problem is to enlarge the picture so its length is equal to the diameter of the rotation circle, which is $\sqrt{(len^2 + len^2)}$, or $len\sqrt{2}$.

For instance, an image the same size as a 512x512 canvas will need to be enlarged to at least $512\sqrt{2}$ by $512\sqrt{2}$ (724x724 pixels) to avoid any gaps when rotated around the center of that canvas.

The drawback of this enlargement is the large increase in image size (it doubles), which makes the rotation calculation more time-consuming.

This is the case for the whirlpool image shown in Figure 9. It's dimensions are 725x726 pixels, and is 175 KB in size. SpinPlanets3D responds very sluggishly to mouse movements due to the processing required to rotate the large image every 200 ms.

The speed problem can be resolved by utilizing a smaller background image, and filling the gaps with a Background node colour. In fact, for SpinPlanets3D, no Background node is needed at all, since the default black background is suitable as a stand-in for deep space.

Figure 14 shows the scene with galaxy.jpg as the background picture (439x440 pixels; 8.2 KB), and there's only a very slight delay when responding to mouse movements. The small file size is due to the image containing large areas of blue and black, which can be readily compressed.

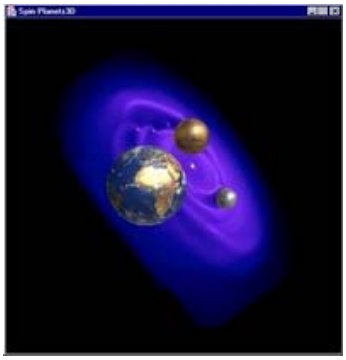

Figure 14. Planets in a Spinning Galaxy.

 ©Andrew Davison 2006

Figure 15 shows the scene with sun.gif (167x180 pixels; 1.5 KB); there's no discernable delay in the application when the user moves the mouse.

Figure 15. Planets with a Spinning Sun.

3.6. Terminating the Thread

The loop inside run() in ImageRotator continues while isRunning is true. The thread is stopped by setting isRunning to false with stopRotating().

```
public void stopRotating()
{  isRunning = false; }
```

stopRotating() is called from stopBackground() in WrapSpinPlanets3D, which is called from SpinPlanets3D when the frame's close box is clicked.

3.7. Drawing the Background Image

SpinBGCanvas3D draws the background image using preRender() in a similar way to previously. However, the details are a little more complicated due to the issue of shared data between multiple threads.

preRender() is called by Java 3D's rendering thread when it needs to update the scene (for example, when the camera position has moved). Complications arise because there's also an ImageRotator thread ticking away, updating the background image every 200 ms.

Since the picture is manipulated by two threads, SpinBGCanvas3D must contain synchronization code to enforce mutual exclusion when both threads want to access the image concurrently.

The SpinBGCanvas3D constructor saves the canvas's dimensions and creates a J3DGraphics2D object for 2D drawing.

```
// globals
private BufferedImage backIm = null;    // the background image
private J3DGraphics2D render2D;
                          // for 2D rendering into the 3D canvas
private int panelWidth, panelHeight;   // of drawing area
```

 ©Andrew Davison 2006

```
public SpinBGCanvas3D(GraphicsConfiguration gc,
                            int pWidth, int pHeight)
{ super(gc);
  panelWidth = pWidth;
  panelHeight = pHeight;
  render2D = this.getGraphics2D();
} // end of SpinBGCanvas3D()
```

The image reference, backIm, which is initially null, is updated with updateBG(), called from ImageRotator.

```
public void updateBG(BufferedImage im)
{  setBGImage(im);
   drawBackground();
}

private synchronized void setBGImage(BufferedImage im)
{  backIm = im;  }
```

After backIm has been updated, drawBackground() draws the image:

```
private synchronized void drawBackground()
{
  if (backIm == null)   // no image to draw
    return;
  int xc = (panelWidth - backIm.getWidth())/2;
  int yc = (panelHeight - backIm.getHeight())/2;
  render2D.drawAndFlushImage(backIm, xc, yc, this);
}
```

drawBackground() positions the center of the image at the center of the canvas.

The synchronization keywords used in setBGImage() and drawBackground() seem unnecessary, until we consider the Java 3D thread. It calls preRender() whenever it needs to redraw the scene. preRender() delegates this task to drawBackground():

```
public void preRender()
{  drawBackground();  }
```

I don't want the rendering thread to call drawBackground() when the image is being updated in setBGImage(), or when the image is already being drawn by ImageRotator calling drawBackground(). These mutual exclusion cases are handled by adding the synchronization keyword to setBGImage() and drawBackground().

 ©Andrew Davison 2006

4. The Model Viewer with a Shifting Background and Overlays

SpinPlanets3D is an example of an application where the background changes independently of other scene activity: the background rotates no matter what the user does inside the 3D space.

A more common type of application is one where the background changes depending on the user's movements in the scene. For instance, as the user rotates *left*, the background rotates to the *right*, in a manner similar to how we see the real world.

The ObjView3D example in this section implements this kind of background manipulation. It's a (slightly) modified version of the model viewer described in Chapter N7.

Figure 16 shows two screenshots of ObjVIew3D, the second taken after the user has moved to a different part of the scene.

Figure 16. Moving in ObjView3D.

As the user's moves left or right, or turns left or right, the background shifts in the opposite direction (i.e. right or left). If the user pirouettes on the spot, the background will disappear off one side of the canvas and eventually reappear at the other side.

This version of ObjView3D also includes the two overlays at the bottom of the screen, as shown in Figure 17.

Figure 17. The Overlays in ObjView3D.

On the left is the red text "Moves: 0", and on the right a translucent image containing the title of the application. The number in the "Moves" message is incremented each time the background is shifted.

 ©Andrew Davison 2006

The class diagrams for this version of ObjView3D are given in Figure 18. Only the public methods are shown.

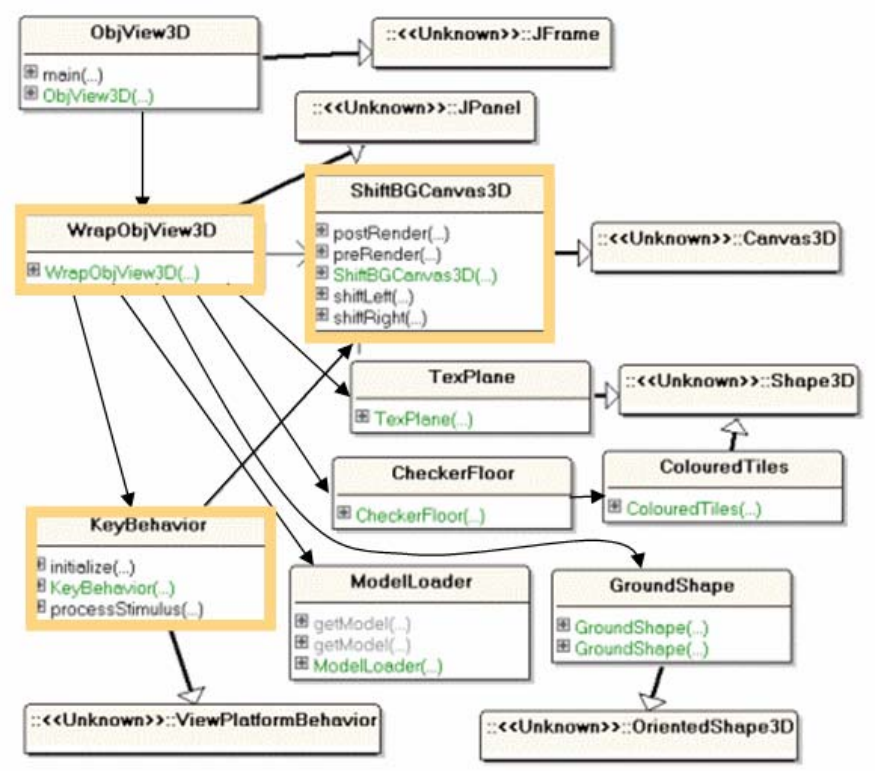

Figure 18. Class Diagrams for ObjView3D.

The application is complicated, but most of it is the same as in Chapter N7, with all the changes localized in the three classes with the thick yellow borders. The main differences are:

- ShiftBGCanvas3D is new – it shifts the background and manages the overlays;
- WrapObjView3D, which sets up the 3D world, now utilizes ShiftBGCanvas3D rather than Canvas3D. It passes a ShiftBGCanvas3D reference to KeyBehavior. Only a single 3D model is loaded (the 'humanoid') to simplify the scene.
- KeyBehavior contain extra code to communicate with ShiftBGCanvas3D. It asks it to move the background left or right.

I won't talk about the unchanged parts of ObjView3D; if you're not familiar with those classes, please look back at Chapter N7.

4.1. Setting up the Canvas

The WrapObjView3D constructor employs ShiftBGCanvas3D in almost the same way as the standard Canvas3D.

 ©Andrew Davison 2006

```
// globals
private static final int PWIDTH = 512;   // size of panel
private static final int PHEIGHT = 512;
  // other constants...

private ShiftBGCanvas3D canvas3D;
  // other variables...


public WrapObjView3D()
// construct the 3D canvas
{
  setLayout( new BorderLayout() );
  setOpaque( false );
  setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

  GraphicsConfiguration config =
        SimpleUniverse.getPreferredConfiguration();
  canvas3D = new ShiftBGCanvas3D(config, PWIDTH, PHEIGHT);
  add("Center", canvas3D);

  canvas3D.setFocusable(true);
  canvas3D.requestFocus();
  su = new SimpleUniverse(canvas3D);

  // depth-sort transparent objects on a per-geometry basis
  View view = su.getViewer().getView();
  view.setTransparencySortingPolicy(View.TRANSPARENCY_SORT_GEOMETRY);

  createSceneGraph();
  createUserControls();

  su.addBranchGraph( sceneBG );
} // end of WrapObjView3D()
```

The dimensions of the panel are passed to ShiftBGCanvas3D to help it with the positioning of the background image.

The ShiftBGCanvas3D variable, canvas3D, is global so it can be easily passed to the KeyBehavior instance created in createUserControls():

```
KeyBehavior keyBeh = new KeyBehavior(canvas3D);
```

4.2. Modifying the Key Behavior

KeyBehavior calls two public methods in ShiftBGCanvas3D: shiftLeft() and shiftRight().

When the camera moves or turns left/right, the background is moved right/left with shiftRight()/shiftLeft().

Rotation is handled by KeyBehavior's standardMove() method:

```
// globals
private AmmoManager ammoMan;
private ShiftBGCanvas3D canvas3D;
```

 ©Andrew Davison 2006

```
private void standardMove(int keycode)
/* Make viewer moves forward or backward;
   rotate left or right. */
{
  if(keycode == forwardKey)
    doMove(FWD);
  else if(keycode == backKey)
    doMove(BACK);
  else if(keycode == leftKey) {
    rotateY(ROT_AMT);
    canvas3D.shiftRight(10);
  }
  else if(keycode == rightKey) {
    rotateY(-ROT_AMT);
    canvas3D.shiftLeft(10);
  }
} // end of standardMove()
```

The new code is highlighted in bold. I experimented with various shift values until the background seemed to move realistically.

KeyBehavior deals with camera translation in altMove():

```
private void altMove(int keycode)
// moves viewer up or down, left or right
{
  if(keycode == forwardKey) {
    upMoves++;
    doMove(UP);
  }
  else if(keycode == backKey) {
    if (upMoves > 0) {  // don't drop below start height
      upMoves--;
      doMove(DOWN);
    }
  }
  else if(keycode == leftKey) {
    doMove(LEFT);
    canvas3D.shiftRight(1);
  }
  else if(keycode == rightKey) {
    doMove(RIGHT);
    canvas3D.shiftLeft(1);
  }
}  // end of altMove()
```

Both turning and translation are handled by shifting the background. The only difference is that a rotation left or right triggers a larger shift (by 10 units) than a translation (1 unit). This difference reflects the fact that when a person turns, the view in front of them changes more drastically than if they only move laterally.

4.3. A Canvas with a Background and Overlays

ShiftBGCanvas3D overrides Canvas3D's preRender() *and* postRender() methods.

 ©Andrew Davison 2006

The preRender() method draws a background which is moved left or right depending on the camera position. postRender() draws a "Moves: <no>" text message and a translucent title image in the foreground.

The ShiftBGCanvas3D constructor loads the background and title images, and creates a drawing surface for the background and foreground visuals.

```
// globals
private static final String BACK_IM = "night.gif";
private static final String TITLE_IM = "title.png";

private BufferedImage drawIm, backIm, titleIm;
private Graphics2D drawg2d;   // for drawing into drawIm
private J3DGraphics2D render2D;   // for 2D rendering into 3D canvas

private int panelWidth, panelHeight;  // size of drawing area
private int imWidth;        // width of background image
private int xc;  // current x-axis position for drawing the BG image

// for displaying messages
private Font msgsFont;
private int moveNum = 0;


public ShiftBGCanvas3D(GraphicsConfiguration gc,
                            int pWidth, int pHeight)
{ super(gc);
  panelWidth = pWidth;
  panelHeight = pHeight;

  /* create a drawing surface with support for an alpha channel,
     and a graphic context for drawing into it */
  drawIm = new BufferedImage(panelWidth, panelHeight,
                                BufferedImage.TYPE_4BYTE_ABGR);
  drawg2d = drawIm.createGraphics();

  render2D = this.getGraphics2D();

  backIm = loadImage(BACK_IM);   // load background image
  xc = 0;  // x-axis drawing position for BG image
  if (backIm != null) {
    imWidth = backIm.getWidth();
    if (imWidth < panelWidth)
      System.out.println("WARNING: background image is too narrow");
  }

  titleIm = loadImage(TITLE_IM);   // load title image

  // create message font
  msgsFont = new Font("SansSerif", Font.BOLD, 24);
} // end of ShiftBGCanvas3D()
```

The drawing surface is a BufferedImage called drawIm, and its associated graphics context, drawg2d.

The background will often be composed from *two* copies of the background picture, and these images need to be combined into a single visual before being rendered by

 ©Andrew Davison 2006

drawAndFlushImage() in preRender(). The drawing surface is used for this composition.

The foreground consists of two overlays: the "Moves" text and the translucent title. They're combined into a single image in postRender() by writing them onto the drawing surface, and calling drawAndFlushImage().

These composition tasks could be assigned to two separate drawing surfaces, one for the background, one for the foreground, but it's straightforward to reuse the same surface.

The drawIm BufferedImage is of type BufferedImage.TYPE_4BYTE_ABGR, which includes an alpha channel. This means that transparent or translucent images will be rendered correctly.

Efficiency is another reason for using BufferedImage.TYPE_4BYTE_ABGR. If OpenGL is the underlying graphics API then drawIm can be rendered without drawAndFlushImage() having to make a copy of it.

backIm contains the basic background image, a sunset, shown in Figure 19.

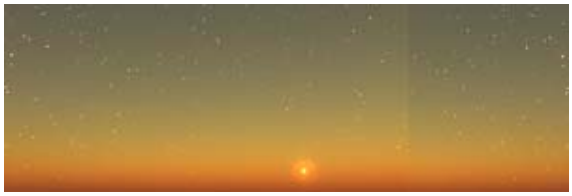

Figure 19. The ObjView3D Background.

I've widened the image so it's longer than the canvas width. The code in preRender() assumes that the image is at least the width of the canvas, and the constructor prints a warning if this is not so. This assumption means that at most two copies of the image will be needed to completely span the width of the canvas.

The translucent image in titleIm is loaded from a PNG file, and is shown in Figure 20.

Figure 20. The Translucent Title.

The white and gray checkboard effect is the 50% alpha setting which makes the image translucent.

4.4. Drawing the Background

The background is moved left and right by KeyBehavior calling ShiftBGCanvas3D's shiftLeft() and shiftRight() methods:

```
// globals
private int xc = 0;
```

 ©Andrew Davison 2006

```
private int moveNum = 0;


public void shiftLeft(int amt)
// shift the background left by amt
{
  xc -= amt;
  moveNum++;
}

public void shiftRight(int amt)
// shift the background right by amt
{
  xc += amt;
  moveNum++;
}
```

The global xc variable stores the x-coordinate of the left edge of the backIm image. It starts at 0, and is decremented and incremented by shiftLeft() and shiftRight(). The moveNum counter records the number of times that the image has been shifted.

preRender() uses the current value of xc to decide where to position backIm in the canvas, and whether another copy needs to be drawn in order to fully span the canvas.

```
public void preRender()
// draw the background
{
  if (backIm == null)   // no background image to draw
    return;

  clearSurface();

  // adjust x coord to be within the bounds of the image width
  if (xc >= imWidth)
    xc -= imWidth;
  else if (xc <= -imWidth)
    xc += imWidth;

  if (xc > 0) {    // background starts on the right
    drawg2d.drawImage(backIm, xc-imWidth, 0, null);   // draw lhs
    if (xc < panelWidth)      // start of BG is on-screen
      drawg2d.drawImage(backIm, xc, 0, null);   // draw rhs
  }
  else {  // xc <= 0, background starts on the left
    drawg2d.drawImage(backIm, xc, 0, null);   // draw lhs
    int endBG = xc + imWidth;
    if (endBG < panelWidth)   // end of BG is on-screen
      drawg2d.drawImage(backIm, endBG, 0, null);   // draw rhs
  }

  render2D.drawAndFlushImage(drawIm, 0, 0, this);
}  // end of preRender()
```

clearSurface() clears the drawing surface prior to the creation of a new background. The technique was suggested by 'mokopa' in the Java 3D forum at javagaming.org (http://www.javagaming.org/forums/index.php?topic=7320.0).

```
private void clearSurface()
```

 ©Andrew Davison 2006

```
{
  drawg2d.setComposite(
          AlphaComposite.getInstance(AlphaComposite.CLEAR, 0.0f));
  drawg2d.fillRect(0,0, panelWidth, panelHeight);
  drawg2d.setComposite(
          AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 1.0f));
  drawg2d.setColor(Color.BLACK);
}  // end of clearSurface()
```

The alpha composition rule is set to AlphaComposite.CLEAR with full transparency. As a consequence, the filled rectangle will set the alpha and RGB channels in the image to 0 for all the pixels. This deletes whatever was in the drawing surface, making it fully transparent.

Then the alpha composition is changed to fully opaque AlphaComposite.SRC_OVER. Any subsequent drawings will be written onto the surface without being influenced by the surface's transparency.

Back in preRender(), xc is adjusted to be within the range -imWidth to imWidth. These extremes are the x-axis values where the image is completely off the canvas on the left and right sides, and so can be redrawn at the xc=0 position.

The if-test breaks the drawing task into two parts.

If xc is greater than 0, then the left edge of backIm may be in the middle of the canvas, as illustrated by Figure 21.

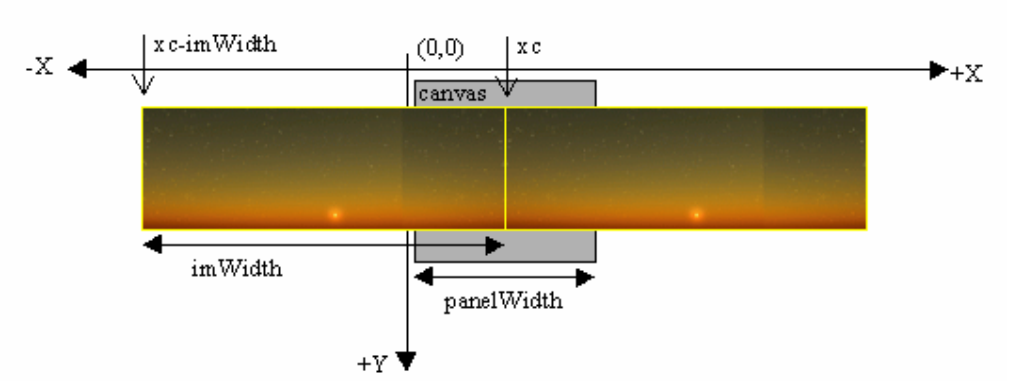

Figure 21. The Background when xc > 0.

This requires a second copy of backIm be drawn on the left, so the full width of the canvas is covered. Figure 21 shows gray areas above and below the background pictures in order to clarify the position of the images; none of the canvas is visible in the application.

The Figure 21 situation is implemented using the code fragment:

```
if (xc > 0) {    // background starts on the right
  drawg2d.drawImage(backIm, xc-imWidth, 0, null);   // draw lhs
  if (xc < panelWidth)      // start of BG is on-screen
    drawg2d.drawImage(backIm, xc, 0, null);   // draw rhs
```

 ©Andrew Davison 2006

```
}
```

Two copies of backIm are drawn into the drawIm BufferedImage via the drawg2d graphics context. However, the right-hand backIm is only drawn if xc is on the canvas (i.e. less than the panel width).

This code could be optimized by drawing subregions of the images into drawIm. There's really no need to draw the parts of the picture located off the sides of the canvas.

The other drawing case is when xc is less than or equal to 0, which means that backIm's left edge starts off the left side of the canvas. This situation is shown graphically in Figure 22.

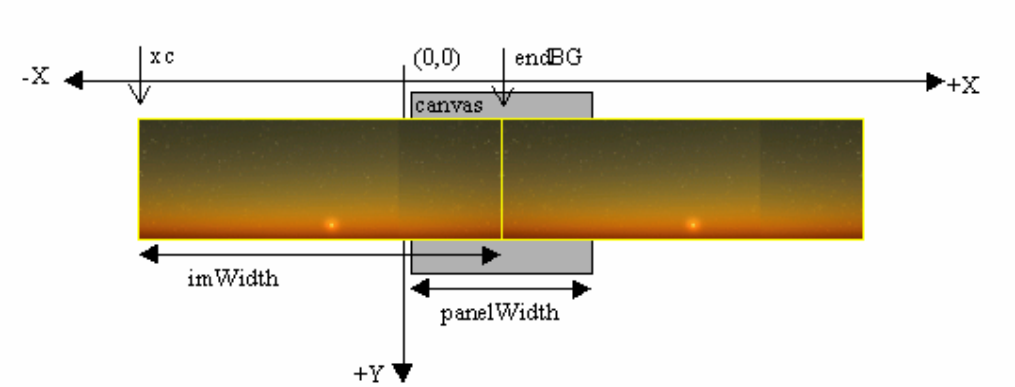

Figure 22. The Background when xc <= 0.

This requires a second copy of backIm be drawn on the right, so the canvas is fully covered. Figure 22 is implemented using the code fragment:

```
// xc <= 0, background starts on the left
drawg2d.drawImage(backIm, xc, 0, null);   // draw lhs
int endBG = xc + imWidth;
if (endBG < panelWidth)   // end of BG is on-screen
  drawg2d.drawImage(backIm, endBG, 0, null);   // draw rhs
```

Once again two copies of backIm are drawn via the drawg2d graphics context into the drawIm BufferedImage. But the right-hand backIm is only drawn if endBG (xc+imWidth) is on the canvas (i.e. less than the panel width).

4.5. The Deep Blue Sea

The scene graph for ObjView3D includes a Background node displaying dark blue. The relevant method in WrapObjView3D is:

```
private void addBackground()
// the deep blue sea
{ Background back = new Background();
  back.setApplicationBounds( bounds );
  back.setColor(0, 0, 0.639f);    // dark blue
  sceneBG.addChild( back );
```

 ©Andrew Davison 2006

```
} // end of addBackground()
```

This means that the background image doesn't need to extend the full height of the canvas, as shown by Figure 23.

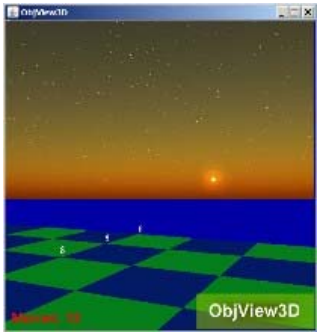

Figure 23. The View from the Checkboard's Edge.

Where the image ends, the dark blue Background node takes over, giving the impression of ocean extending to the horizon. It means that the background image can be shorter than the height of the canvas, making the picture smaller, and therefore less time-consuming to manipulate.

4.6. Adding Overlays

Overlays can be implemented as part of the scene graph, by pasting a texture representing the overlay onto a quad positioned just in front of the user's viewpoint.

The drawback with this approach is that the overlay quad may overlap or obscure objects in the scene when the user moves very close to them. This problem can't occur with overlays generated in postRender() since they're drawn onto the canvas after scene graph rendering has been completed.

```
public void postRender()
// draw the text message and the title image in the foreground
{
  clearSurface();

  // draw the title image, with hardwired positioning
  drawg2d.drawImage(titleIm, panelWidth-195, panelHeight-60, null);

   // draw the firing info.
  drawg2d.setFont(msgsFont);
  drawg2d.setColor(Color.red);
  drawg2d.drawString("Moves: " + moveNum, 10, panelHeight-10);

  render2D.drawAndFlushImage(drawIm, 0, 0, this);
}  // end of postRender()
```

postRender() uses the same drawing surface (drawIm and drawg2d) as preRender(), so starts by wiping it clean. The text message and the translucent title are drawn onto the surface, and the resulting drawIm image is placed onto the canvas.

 ©Andrew Davison 2006