JOGL 2. Touring the World

In the last chapter, I described two programming frameworks for JOGL (callbacks and active rendering), with a simple rotating cube as the example. In this chapter, I utilize the active rendering framework to build a more substantial application, a small 3D world. A screenshot of the TourGL program is shown in Figure 1.

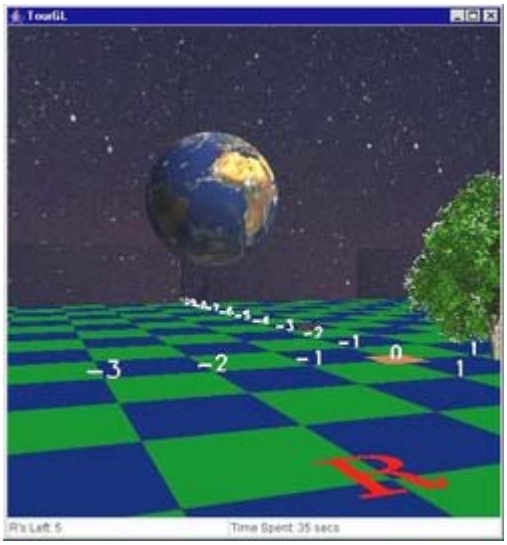

Figure 1. The TourGL Application.

The visual elements are:

- a green and blue checkerboard floor with a red square at its center, and numbers along its x- and z- axes;
- a skybox of stars;
- an orbiting 'earth' (a textured sphere);
- a billboard showing a tree, which rotates around its y-axis to always face the camera;
- five red 'R's placed at random on the ground.

The user can move around the world using the arrow keys to go forward, backwards, left, right, up, down, and turn left and right, but the player can't travel off the checkerboard or beyond the borders of the skybox. The game is terminated by pressing ESC, q, END, or ctrl-c, or by clicking the close box.

TourGL is a very basic game – the user must navigate over the red R's lying on the ground to make them disappear. The game ends when all the R's have been deleted, and a score is calculated based on how long it took to delete the R's.

Since TourGL utilizes the active rendering framework, the game automatically suspends when the window is iconified or deactivated, and can be resized. TourGL also offers two extensions to the framework:

- while the world is being initialized, a message is displayed in the canvas (see Figure 2);
- a game-over 2D image with text is displayed when all the R's have disappeared (see Figure 3). This stays on-screen until the application is terminated.

Figure 2. The Loading Message.

Figure 3. The Game-Over Message.

Various OpenGL (JOGL) techniques are illustrated in this example, including transparent images as textures, 2D overlays, and bitmap and stroke fonts.

The class diagrams for TourGL in Figure 4 highlight its use of active rendering.

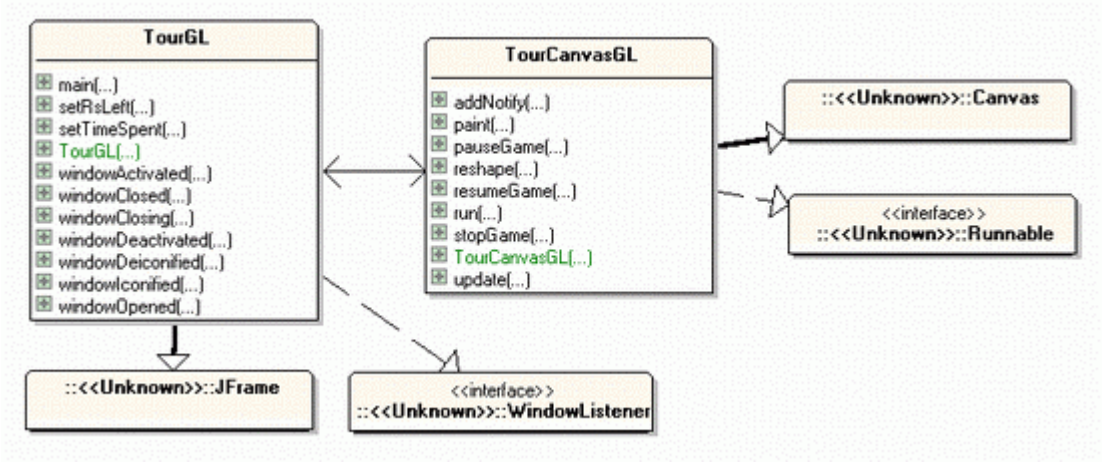

Figure 4. Class Diagrams for TourGL.

The TourGL class creates the GUI, which consists of the threaded canvas, TourCanvasGL, and two textfields at the bottom of the window.

TourGL captures window and component resizing events, and calls methods in TourCanvasGL to pause, resume, stop, and resize the 3D scene. It also has two set

methods for its textfields, setRsLeft() and setTimeSpent(). They're called by TourCanvasGL to report the number of remaining R's, and the game's running time.

1. Constructing the Canvas

The TourCanvasGL constructor obtains the OpenGL drawing surface and context, standard tasks for active rendering applications.

```
// globals
private TourGL tourTop;     // reference back to top-level JFrame
private long period;        // period between drawing in nanosecs

private int panelWidth, panelHeight;

private GLDrawable drawable;  // the rendering 'surface'
private GLContext context;    // rendering state info

// used by the 'loading' message
private Font font;
private FontMetrics metrics;
private BufferedImage waitIm;


public TourCanvasGL(TourGL top, long period, int width, int height,
                  GraphicsConfiguration config, GLCapabilities caps)
{
  super(config);

  tourTop = top;
  this.period = period;
  panelWidth = width;
  panelHeight = height;

  setBackground(Color.white);

  // get a rendering surface and a context for this canvas
  drawable = GLDrawableFactory.getFactory().getGLDrawable(this,
                                                    caps, null);
  context = drawable.createContext(null);

  generateRPosns();
  initViewerPosn();

  addKeyListener( new KeyAdapter() {
    public void keyPressed(KeyEvent e)
    { processKey(e);  }
  });

  // 'loading' message font and image
  font = new Font("SansSerif", Font.BOLD, 24);
  metrics = this.getFontMetrics(font);
  waitIm = loadImage("hourglass.jpg");

  // statistics initialization
     // not shown here...
} // end of TourCanvasGL()
```


TourCanvasGL performs four application-specific jobs: it generates the (x,z) coordinates of the R's (in generateRPosns()), initializes the camera coordinates (initViewerPosn()), prepares the font and image used in the loading message, and sets up a key listener (processKey()).

2. Generating the R Positions

TourGL's action-packed game requires the user to run over five red R's randomly placed on the floor. The final score is based on the time it takes the player to complete this daring feat.

```
// globals
private final static int FLOOR_LEN = 20;  // should be even
private final static int NUM_RS = 5;

private float[] xCoordsRs, zCoordsRs;    // placement of R's
private boolean[] visibleRs;     // if the R's are visible
private int numVisibleRs;


private void generateRPosns()
{
  Random rand = new Random();
  xCoordsRs = new float[NUM_RS];
  zCoordsRs = new float[NUM_RS];
  visibleRs = new boolean[NUM_RS];
  numVisibleRs = NUM_RS;

  for (int i=0; i < NUM_RS; i++) {
    xCoordsRs[i] = (rand.nextFloat()*(FLOOR_LEN-1)) - FLOOR_LEN/2;
    zCoordsRs[i] = (rand.nextFloat()*(FLOOR_LEN-1)) - FLOOR_LEN/2;
    visibleRs[i] = true;
  }
}  // end of generateRPosns()
```

The x- and z- axis coordinates are store in two arrays, xCoordsRs[] and zCoordsRs[], which are used later to specify the center of each R image. The values are randomly generated, but lie on the floor. (The floor is a square of sides FLOOR_LEN, centered on (0,0))

The rendering of the R's is controlled by the boolean array visibleRs[]. When a R is run over it becomes 'invisible', and will no longer be drawn.

3. The Camera Position

initViewerPosn() stores the camera position in xPlayer, yPlayer, and zPlayer, and the point being looked at in xLookAt, yLookAt, and zLookAt.

The camera's orientation is recorded in viewAngle, and is used to calculate the x- and z- axis 'step' which moves the camera forward.


```
// camera related constants
private final static double LOOK_AT_DIST = 100.0;
private final static double Z_POS = 9.0;

// camera movement
private double xPlayer, yPlayer, zPlayer;
private double xLookAt, yLookAt, zLookAt;
private double xStep, zStep;
private double viewAngle;


private void initViewerPosn()
{
  xPlayer = 0; yPlayer = 1; zPlayer =  Z_POS;    // camera posn

  viewAngle = -90.0;   // along -z axis
  xStep = Math.cos( Math.toRadians(viewAngle));  // step distances
  zStep = Math.sin( Math.toRadians(viewAngle));

  xLookAt = xPlayer + (LOOK_AT_DIST * xStep);   // look-at posn
  yLookAt = 0;
  zLookAt = zPlayer + (LOOK_AT_DIST * zStep);
}  // end of initViewerPosn()
```

By setting the viewAngle to -90, the initial xStep and zStep values become (0, -1), which is a step 'forward' into the scene along the -z-axis.

The camera looks forward into the distance, represented by the coordinate (LOOK_AT_DIST*xStep, 0, LOOK_AT_DIST*zStep), which is (0,0,-100) initially.

These variables are used to adjust the camera in renderScene() (explained later), which calls GLU.gluLookAt() :

```
glu.gluLookAt(xPlayer, yPlayer, zPlayer,
                   xLookAt, yLookAt, zLookAt, 0,1,0);
```

The player and lookAt coordinates are changed whenever the user presses an arrow key to move the camera. Key processing is handled by processKey(), described next.

4. Responding to Key Presses

TourGL assigns focus to the canvas when it's created:

```
canvas.setFocusable(true);
canvas.requestFocus();
```

This means that key events will be passed to TourCanvasGL, where they're dealt with by processKey().

Two groups of keys are handled: those triggering the application's termination (ESC, q, END, ctrl-c) and those for moving the camera (the arrow keys, possibly combined with the ctrl key, and page-up and page-down).

The arrow keys cause the camera to move forwards, backwards, and rotate left and right. When the left or right arrow keys are pressed with the ctrl key, then the camera translates left or right. The page-up and page-down keys send the camera up and down.

The user can travek around the scene freely, but isn't allowed to go beyond the confines of the floor, which lie between -FLOOR_LEN/2 and FLOOR_LEN/2 in the x- and z- directions. Vertical movement is limited to be between 0 and FLOOR_LEN/2 (the height of the skybox).

The camera is moved by adding the xStep and zStep values to the camera's (x,z) position. xStep and zStep are calculated using Math.cos() and Math.sin() applied to the camera's current viewAngle, as illustrated by the 'forward' vector in Figure 5.

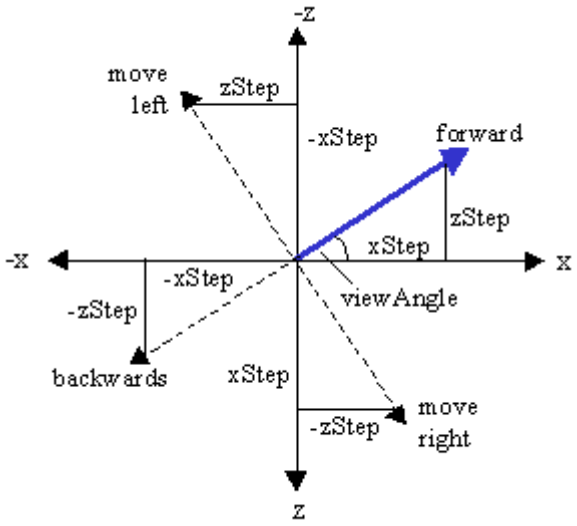

Figure 5. Camera Movements.

Figure 5 also shows how xStep and zStep can be employed to make left, right, and backwards vectors. One tricky aspect is that the zStep value for the forward vector is negative (since it represents a step along the -z axis), so its sign must be changed if a positive number is required (i.e. for translating right and backwards).

The processKey() method:

```
// camera related constants
private final static double SPEED = 0.4;   // for camera movement
private final static double LOOK_AT_DIST = 100.0;
private final static double ANGLE_INCR = 5.0;   // degrees
private final static double HEIGHT_STEP = 1.0;


private void processKey(KeyEvent e)
// handles termination, and the game-play keys
{
  int keyCode = e.getKeyCode();

  // termination keys
  // listen for esc, q, end, ctrl-c on the canvas
  if ((keyCode == KeyEvent.VK_ESCAPE) ||
      (keyCode == KeyEvent.VK_Q) ||
```



```
      (keyCode == KeyEvent.VK_END) ||
      ((keyCode == KeyEvent.VK_C) && e.isControlDown()) )
    isRunning = false;

  // game-play keys
  if (isRunning) {
    // move based on the arrow key pressed
    if (keyCode == KeyEvent.VK_LEFT) {    // left
      if (e.isControlDown()) {   // translate left
        xPlayer += zStep * SPEED;
        zPlayer -= xStep * SPEED;
      }
      else {  // turn left
        viewAngle -= ANGLE_INCR;
        xStep = Math.cos( Math.toRadians(viewAngle));
        zStep = Math.sin( Math.toRadians(viewAngle));
      }
    }
    else if (keyCode == KeyEvent.VK_RIGHT) {  // right
      if (e.isControlDown()) {   // translate right
        xPlayer -= zStep * SPEED;
        zPlayer += xStep * SPEED;
      }
      else {  // turn right
        viewAngle += ANGLE_INCR;
        xStep = Math.cos( Math.toRadians(viewAngle));
        zStep = Math.sin( Math.toRadians(viewAngle));
      }
    }
    else if (keyCode == KeyEvent.VK_UP) {   // move forward
      xPlayer += xStep * SPEED;
      zPlayer += zStep * SPEED;
    }
    else if (keyCode == KeyEvent.VK_DOWN) {  // move backwards
      xPlayer -= xStep * SPEED;
      zPlayer -= zStep * SPEED;
    }
    else if (keyCode == KeyEvent.VK_PAGE_UP) {  // move up
      if ((yPlayer + HEIGHT_STEP) < FLOOR_LEN/2) {
        // stay below stars ceiling
        yPlayer += HEIGHT_STEP;
        yLookAt += HEIGHT_STEP;
      }
    }
    else if (keyCode == KeyEvent.VK_PAGE_DOWN) {  // move down
      if ((yPlayer - HEIGHT_STEP) > 0) {   // stay above floor
        yPlayer -= HEIGHT_STEP;
        yLookAt -= HEIGHT_STEP;
      }
    }

    // don't allow player to walk off the edge of the world
    if (xPlayer < -FLOOR_LEN/2)
      xPlayer = -FLOOR_LEN/2;
    else if (xPlayer > FLOOR_LEN/2)
      xPlayer = FLOOR_LEN/2;

    if (zPlayer < -FLOOR_LEN/2)
      zPlayer = -FLOOR_LEN/2;
    else if (zPlayer > FLOOR_LEN/2)
      zPlayer = FLOOR_LEN/2;
```



```
    // new look-at point
    xLookAt = xPlayer + (xStep * LOOK_AT_DIST);
    zLookAt = zPlayer + (zStep * LOOK_AT_DIST);
  }
}  // end of processKey()
```

The 'termination' keys don't directly cause the application to exit, instead they set the isRunning boolean to false, which stops the game but keeps the application running. This permits a program to do other things after the end of the game.

The xStep and zStep values are added to the camera's current position with an additional SPEED multiple; for example:

```
xPlayer += xStep * SPEED;
zPlayer += zStep * SPEED;
```

This allows the distance moved to be changed by adjusting the SPEED constant.

Rotation is handled by altering viewAngle: when viewAngle is decremented the camera rotates to the left, an increment causes a right rotation.

Aside from calculating a new camera location, it's also necessary to update the 'look at' location, which is in front of the camera, off into the distance. It's obtained by multiplying the xStep and zStep values by a suitably large constant, and adding them to the camera position:

```
xLookAt = xPlayer + (xStep * LOOK_AT_DIST);
zLookAt = zPlayer + (zStep * LOOK_AT_DIST);
```

The y- components of the camera and 'look at' locations is usually 0 since the camera moves along the ground, but the 'page-up' and 'page-down' keys increment and decrement the y- axis values.

5. Let the Rendering Commence

The top-level parts of TourCanvasGL's active rendering thread are the same as in the rotating cube example from the previous chapter.

```
public void run()
// initialize rendering and start frame generation
{
  initRender();
  renderLoop();

  // discard the rendering context and exit
  context.destroy();
  System.exit(0);
} // end of run()
```

The initialization phase in initRender() may take a few seconds, especially if many, large textures are being loaded. During this time the canvas will remain blank, which

may worry users. A simple alternative, employed in TourCanvasGL, is to draw something onto the canvas, by overriding it's paint() method:

```
// globals
private volatile boolean isRunning = false;
private volatile boolean gameOver = false;


public void paint(Graphics g)
// display a loading message while the canvas is being initialized
{
  if (!isRunning && !gameOver) {
    String msg = "Loading. Please wait...";
    int x = (panelWidth - metrics.stringWidth(msg))/2;
    int y = (panelHeight - metrics.getHeight())/3;
    g.setColor(Color.blue);
    g.setFont(font);
    g.drawString(msg, x, y);

    // draw image underneath text
    int xIm = (panelWidth - waitIm.getWidth())/2;
    int yIm = y + 20;
    g.drawImage(waitIm, xIm, yIm, this);
  }
} // end of paint()
```

paint() is called automatically when the canvas first becomes visible on-screen; it's output is shown in Figure 2.

The if-test in paint() ensures that the text and image will only be drawn before the game starts being rendered.

6. Rendering Initialization

initRender() in TourCanvasGL is an extended version of initRender() in CubeCanvasGL from the last chapter.

```
// OpenGL globals
private GL gl;
private GLU glu;
private GLUT glut;

private int starsDList;       // display lists for stars
private GLUquadric quadric;   // for the sphere

private void initRender()
{
  makeContentCurrent();

  gl = context.getGL();
  glu = new GLU();
  glut = new GLUT();

  resizeView();
```



```
  gl.glClearColor(0.0f, 0.0f, 0.0f, 0.5f);      // black background

  // z- (depth) buffer initialization for hidden surface removal
  gl.glEnable(GL.GL_DEPTH_TEST);

  gl.glShadeModel(GL.GL_SMOOTH);    // use smooth shading

  // create a textured quadric
  quadric = glu.gluNewQuadric();
  glu.gluQuadricTexture(quadric, true);   // creates texture coords

  loadTextures();
  addLight();

  // create a display list for drawing the stars
  starsDList = gl.glGenLists(1);
  gl.glNewList(starsDList, GL.GL_COMPILE);
    drawStars();
  gl.glEndList();

  /* release the context, otherwise the AWT lock on X11
     will not be released */
  context.release();
}  // end of initRender()
```

References to the GL, GLU, *and* GLUT classes are obtained during the initialization stage. A GLUT instance is needed in order to utilize bitmap and stroke fonts.

Smooth shading is switched on to improve the appearance of the sphere when it's drawn later.

The sphere is created with the help of a textured quadric, a GLU feature that also supports cylinders and disks. Incidentally, GLUT also offers complex shapes, including the sphere, cone, torus, several regular polyhedra (e.g. the dodecahedron), and the Utah teapot.

6.1. Loading Textures

The JOGL reference implementation for JSR-231 has several utility classes for texture manipulation.

loadTextures() calls loadTexture() to load each of the texture graphics, and that method utilizes JOGL's TextureIO utility class.

```
// globals
private Texture earthTex, starsTex, treeTex, rTex, robotTex;


private void loadTextures()
{
  earthTex = loadTexture("earth.jpg");  // for the sphere
  starsTex = loadTexture("stars.jpg");  // for the sky box
  treeTex = loadTexture("tree.gif");    // for the billboard
  rTex = loadTexture("r.gif");          // the red R
  robotTex = loadTexture("robot.gif");  // the game-over image

  // repeat the sky box texture in every direction
  starsTex.setTexParameteri(GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
```



```
  starsTex.setTexParameteri(GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT);
}  // end of loadTextures()


private Texture loadTexture(String fnm)
{
  String fileName = "images/" + fnm;
  Texture tex = null;
  try {
    tex = TextureIO.newTexture( new File(fileName), false);
    tex.setTexParameteri(GL.GL_TEXTURE_MAG_FILTER, GL.GL_NEAREST);
    tex.setTexParameteri(GL.GL_TEXTURE_MIN_FILTER, GL.GL_NEAREST);
  }
  catch(Exception e)
  { System.out.println("Error loading texture " + fileName);  }

  return tex;
}  // end of loadTexture()
```

TextureIO supports a wide variety of image types, including GIF, JPEG, and PNG.

TextureIO.newTexture() returns a Texture object, an instance of another JOGL utility class. Texture offers a convenient way of setting/getting texture parameters, binding the texture, and computing texture coordinates.

All the textures use the fast resizing option, GL.GL_NEAREST, when they're magnified or minified. In addition, the stars texture used in the skybox is set to repeat when pasted onto a large surface.

Several of the images have transparent backgrounds, but no special settings are needed to load them.

Texture mapping isn't switched on in these methods, instead it's enabled as needed inside the methods for rendering the skybox, billboard, sphere, the R's, and game-over message.

6.2. Lighting the Scene

OpenGL's lighting model supports multiple light sources, which may have ambient, diffuse, specular, or emissive components, in much the same way as Java 3D.

Lighting interacts with the material settings for objects, which specify the colour reflected when the object is exposed to ambient, diffuse, or specular light. Materials can also emit light, and have a shininess value.

addLight() only deals with light properties; material properties are set when an object is being rendered.

```
private void addLight()
{
  // enable a single light source
  gl.glEnable(GL.GL_LIGHTING);
  gl.glEnable(GL.GL_LIGHT0);

  float[] grayLight = {0.1f, 0.1f, 0.1f, 1.0f};  // weak gray ambient
  gl.glLightfv(GL.GL_LIGHT0, GL.GL_AMBIENT, grayLight, 0);

  float[] whiteLight = {1.0f, 1.0f, 1.0f, 1.0f};
```



```
                    // bright white diffuse & specular
  gl.glLightfv(GL.GL_LIGHT0, GL.GL_DIFFUSE, whiteLight, 0);
  gl.glLightfv(GL.GL_LIGHT0, GL.GL_SPECULAR, whiteLight, 0);

  float lightPos[] = {1.0f, 1.0f, 1.0f, 0.0f};
                    // top right front direction
  gl.glLightfv(GL.GL_LIGHT0, GL.GL_POSITION, lightPos, 0);
}  // end of addLight()
```

The scene uses a single source, GL.GL_LIGHT0, producing a grayish ambient light, with white diffuse and specular components. White is the default colour for LIGHT0's diffuse and specular elements, so the second and third GL.glLightfv() calls aren't necessary, but I've left them in as examples.

The final GL.glLightfv() call sets the light's position; the '0' argument specifies that lightPos defines a *vector* passing through the origin. The light becomes *directional*, corresponding to rays hitting all parts of the scene with the same vector (e.g. somewhat like sun rays hitting the earth). The (1,1,1) vector creates parallel light coming from the front, top right of the scene.

If the final argument of GL.glLightfv() was a '1', then the light would become *positional*, more like a light bulb, emitting light in all directions from the (1,1,1) coordinate.

OpenGL supports up to 8 light sources, called LIGHT0 to LIGHT7, and aside from the features in addLight(), offers spotlights, and various forms of attenuation (decreasing light intensity based on the distance from the source).

7. The Rendering Loop

The rendering cycle in renderLoop() is almost unchanged from the one explained in the last chapter. It can be summarized using a mix of code and pseudo-code:

```
isRunning = true;
while (isRunning) {
  makeContentCurrent();
  gameUpdate();
  renderScene();
  drawable.swapBuffers();

  // sleep a while;
  // maybe do extra game updates;
  // gather statistics;
}
// print statistics;
glu.gluDeleteQuadric(quadric);
```

The new element is the call to GLU.gluDeleteQuadric(), which deletes the textured sphere quadric at the end of rendering.

8. Updating the Game

The changing parts of the scene are the rotating sphere, and which R's are visible on the floor.

```
// globals
private volatile boolean isRunning = false;
private volatile boolean isPaused = false;

// sphere movement
private float orbitAngle = 0.0f;
private float spinAngle = 0.0f;


private void gameUpdate()
{
  if (!isPaused && !gameOver) {
    // update the earth's orbit and the R's
    orbitAngle = (orbitAngle + 2.0f) % 360.0f;
    spinAngle = (spinAngle + 1.0f) % 360.0f;
    checkRs();
  }
}
```

The if-test in gameUpdate() stops the game updating if it's been paused (i.e. minimized or deactivated) or if the game is over (i.e. all the R's are invisible).

The sphere's position is dictated by two rotation variables: orbitAngle is the sphere's current angle around a central point (similar to the way the earth orbits the sun), and spinAngle is the sphere's rotation around its own y-axis.

checkRs() checks whether the current camera position is near a visible R, and then makes it invisible. If this means that all the R's are now invisible, then the gameOver boolean is set to true.

```
// globals
private final static int NUM_RS = 5;
private final static double R_CLOSE_DIST = 1.0;

private float[] xCoordsRs, zCoordsRs;    // placement of R's
private boolean[] visibleRs;     // if the R's are visible
private int numVisibleRs;


private void checkRs()
{
  if (numVisibleRs == 0)    // no R's left already
    return;

  int nearRIdx = isOverR();    // is the camera near a R?
  if (nearRIdx != -1) {
    visibleRs[nearRIdx] = false;    // make that R invisible
    numVisibleRs--;
    tourTop.setRsLeft(numVisibleRs);
  }
  if (numVisibleRs == 0) {   // all the R's are gone, so game is over
    gameOver = true;
    score =  25 - timeSpentInGame;   // hack together a score
  }
```



```
}  // end of checkRs()


private int isOverR()
// returns index of R that the player (camera) is 'near', or -1
{
  for(int i=0; i < NUM_RS; i++) {
    if (visibleRs[i]) {
      double xDiff = xPlayer - (double) xCoordsRs[i];
      double zDiff = zPlayer - (double) zCoordsRs[i];
      if (((xDiff*xDiff) + (zDiff*zDiff)) < R_CLOSE_DIST)
        return i;
    }
  }
  return -1;
}  // end of isOverR()
```

isOverR() checks the camera position against all the visible R's, with 'nearness' based on the squared distance between each R and the camera. Only the x- and z- axis values are considered, so it doesn't matter how high the camera is in the air.

9. Rendering the Scene

renderScene() draws the scene, and also checks that the context is current, resizes the OpenGL viewport if necessary, and responds to the gameOver boolean being true.

```
private void renderScene()
{
  if (context.getCurrent() == null) {
    System.out.println("Current context is null");
    System.exit(0);
  }

  if (isResized) {
    resizeView();
    isResized = false;
  }

  // clear colour and depth buffers
  gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);

  gl.glMatrixMode(GL.GL_MODELVIEW);
  gl.glLoadIdentity();

  glu.gluLookAt(xPlayer, yPlayer, zPlayer,
            xLookAt, yLookAt, zLookAt, 0,1,0); // position camera

  drawTree();
  drawRs();
  drawSphere();
  drawFloor();

  // execute display list for drawing the stars
  gl.glCallList(starsDList);  // it calls drawStars();

  if (gameOver)
```



```
    gameOverMessage();
} // end of renderScene()
```

The game-specific parts of renderScene() are drawing the tree billboard, the R's, the sphere, the floor, and the skybox, which are carried out by the highlighted draw methods and the display list.

10. Drawing the Tree Billboard

drawTree() wraps a tree image over a quadrilateral (a quad), without drawing its transparent background. It also rotates the quad to face the current camera position. The tree billboard is shown in Figure 6.

Figure 6. The Tree Billboard.

drawTree() specifies the quad's coordinates so it rests on the XZ plane, initially facing along the +z axis, with its base centered at (1,0,0); Figure 7 shows the billboard's coordinates.

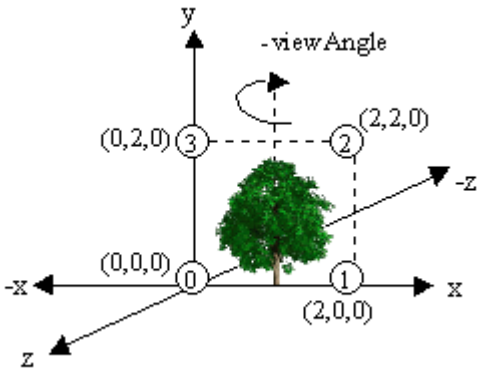

Figure 7. The Tree Billboard's Coordinates.

The numbered circles in Figure 7 indicate the order that the vertices are specified in the quad, starting with the lower-left and progressing counter-clockwise. The order is used when mapping the texture onto the quad's surface.

drawTree() lets drawScreen() do most of the work of creating the quad:

```
private void drawTree()
{
  float[] verts = {0,0,0, 2,0,0, 2,2,0, 0,2,0};  // posn of tree
  gl.glPushMatrix();
  gl.glRotatef(-1*((float)viewAngle+90.0f), 0, 1, 0);
       // rotate in the opposite direction to the camera
  drawScreen(verts, treeTex);
  gl.glPopMatrix();
}  // end of drawTree()
```

The billboard's rotation is derived from the camera's rotation stored in viewAngle. viewAngle has an initial value of -90, which needs to be subtracted away. Also, the billboard must rotate in the opposite direction to the camera to keep facing it, so viewAngle is multiplied by -1.

10.1 Drawing a Textured Quad

drawScreen() is used in several places in the code, to render the billboard, the R's, and the game-over image. It's supplied with four vertices (in an array) which it uses to draw a quad wrapped with the supplied texture. Transparent parts of the texture aren't rendered.

```
private void drawScreen(float[] verts, Texture tex)
{
  boolean enableLightsAtEnd = false;
  if (gl.glIsEnabled(GL.GL_LIGHTING)) {
             // switch lights off if currently on
    gl.glDisable(GL.GL_LIGHTING);
    enableLightsAtEnd = true;
  }

  // do not draw the transparent parts of the texture
  gl.glEnable(GL.GL_BLEND);
  gl.glBlendFunc(GL.GL_SRC_ALPHA, GL.GL_ONE_MINUS_SRC_ALPHA);
            // don't show source alpha parts in the destination

  // determine which areas of the polygon are to be rendered
  gl.glEnable(GL.GL_ALPHA_TEST);
  gl.glAlphaFunc(GL.GL_GREATER, 0);  // only render if alpha > 0

  // enable texturing
  gl.glEnable(GL.GL_TEXTURE_2D);
  tex.bind();

  // replace the quad colours with the texture
  gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_TEXTURE_ENV_MODE,
                                                GL.GL_REPLACE);

  gl.glBegin(GL.GL_QUADS);
  gl.glTexCoord2f(0.0f, 0.0f);
  gl.glVertex3f(verts[0], verts[1], verts[2]);

  gl.glTexCoord2f(1.0f, 0.0f);
  gl.glVertex3f(verts[3], verts[4], verts[5]);
```



```
  gl.glTexCoord2f(1.0f, 1.0f);
  gl.glVertex3f(verts[6], verts[7], verts[8]);

  gl.glTexCoord2f(0.0f, 1.0f);
  gl.glVertex3f(verts[9], verts[10], verts[11]);
  gl.glEnd();

  gl.glDisable(GL.GL_TEXTURE_2D);

  // switch back to modulation of quad colours and texture
  gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_TEXTURE_ENV_MODE,
                                        GL.GL_MODULATE);
  gl.glDisable(GL.GL_ALPHA);  // switch off transparency
  gl.glDisable(GL.GL_BLEND);

  if (enableLightsAtEnd)
    gl.glEnable(GL.GL_LIGHTING);
}  // end of drawScreen()
```

Lighting is disabled since it affects the texture rendering.

The non-drawing of the transparent parts of the image involve the use of blending, alpha testing, and the replacement of the quad's colours. Blending and alpha testing are switched off at the end of the method, and quad colour replacement is switched back to modulation (the default behaviour).

The texture coordinates are mapped to the quad coordinates by assuming they're stored in counter-clockwise order, with the first vertex at the bottom left of the quad. This ordering is represented in Figure 7 by the numbered circles at the four corners of the tree quad.

11. Drawing the R's

Figure 8 shows two R's, although the one in the background on the right is quite hard to see.

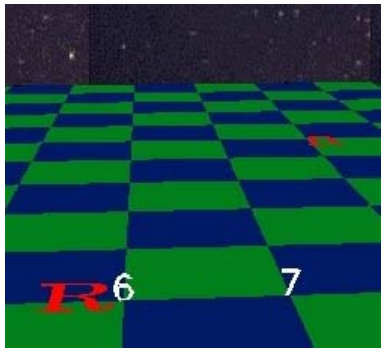

Figure 8. Red R's.

The coordinates for a R are derived from the (x,z) value produced by generateRPosns (). The (x,z) becomes the R's center point on the ground, with the quad orientated so its base is further down the +z axis than its top; the idea is illustrated by Figure 9.

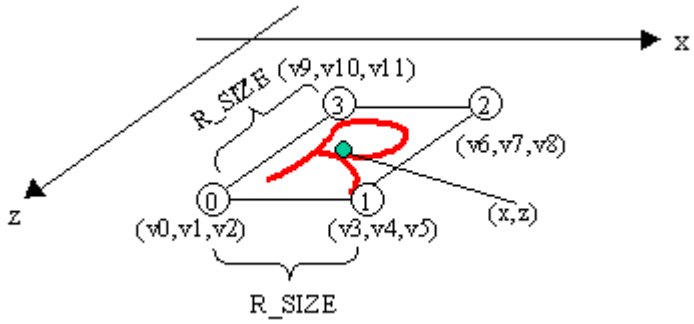

Figure 9. The Coordinates for a R.

drawRs() generates four vertices for each R quad, and renders them by calling drawScreen():

```
// global
private final static float R_SIZE = 0.5f;   // size of R quad


private void drawRs()
{
  if (numVisibleRs == 0)    // nothing to draw
    return;

  float[] verts = new float[12];    // to hold 4 (x,y,z) vertices
  float xc, zc;
  for (int i=0; i < NUM_RS; i++) {
    if (visibleRs[i]) {
      xc = xCoordsRs[i];
      zc = zCoordsRs[i];
      /* make a quad with center (xc,zc) with sides R_SIZE,
         lying just above the floor */
      verts[0] = xc-R_SIZE/2; verts[1] = 0.1f;
                                    verts[2] = zc+R_SIZE/2;
      verts[3] = xc+R_SIZE/2; verts[4] = 0.1f;
                                    verts[5] = zc+R_SIZE/2;
      verts[6] = xc+R_SIZE/2; verts[7] = 0.1f;
                                    verts[8] = zc-R_SIZE/2;
      verts[9] = xc-R_SIZE/2; verts[10] = 0.1f;
                                    verts[11] = zc-R_SIZE/2;
      drawScreen(verts, rTex);
    }
  }
}  // end of drawRs()
```

The (x,z) point in Figure 9 is represented in drawRs() by (xc,zc), and the diagram's vertices, v0, v1, v2, etc., correspond to the cells of the verts[] array.

Each R image is lifted a little bit off the floor by having its vertices utilize a y-axis value of 0.1.

12. The Planet Earth

The rotating and spinning earth is shown in Figure 10.

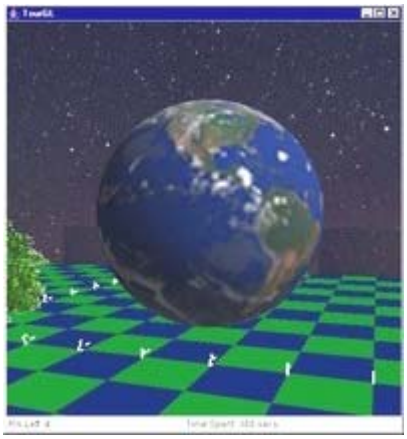

Figure 10. The Earth.

The sphere is built with the textured quadric created in initRender():

```
quadric = glu.gluNewQuadric();
glu.gluQuadricTexture(quadric, true);
```

It's necessary to define material properties for the sphere's surface to specify how light is reflected. Also, since the sphere is orbiting a point and spinning around its y-axis, the sphere will need to be translated and rotated.

The code for drawSphere():

```
private void drawSphere()
{
  // enable texturing and choose the 'earth' texture
  gl.glEnable(GL.GL_TEXTURE_2D);
  earthTex.bind();

  // set how the sphere's surface responds to the light
  gl.glPushMatrix();
  float[] grayCol = {0.8f, 0.8f, 0.8f, 1.0f};
  gl.glMaterialfv(GL.GL_FRONT, GL.GL_AMBIENT_AND_DIFFUSE,grayCol,0);

  float[] whiteCol = {1.0f, 1.0f, 1.0f, 1.0f};
  gl.glMaterialfv(GL.GL_FRONT, GL.GL_SPECULAR, whiteCol, 0);
  gl.glMateriali(GL.GL_FRONT, GL.GL_SHININESS, 100);

  gl.glTranslatef(0.0f, 2.0f, -5.0f);  // position the sphere (1)
  gl.glRotatef(orbitAngle, 0.0f, 1.0f, 0.0f);
                                       // orbit around the point (2)
  gl.glTranslatef(2.0f, 0.0f, 0.0f);   // the orbit's radius (3)

  gl.glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
                /* rotate sphere upwards around the x-axis so
                   the texture is correctly orientated (4) */
  gl.glRotatef(spinAngle, 0.0f, 0.0f, 1.0f);
```



```
        // spin around the z-axis (which looks like the y-axis) (5)

  glu.gluSphere(quadric, 1.0f, 32, 32);  // generate textured sphere
                      // radius, slices, stacks
  gl.glPopMatrix();

  gl.glDisable(GL.GL_TEXTURE_2D);
} // end of drawSphere()
```

The sphere reflects gray for ambient and diffuse light, and white for specular light, with a large shininess value which makes the specular region bigger.

The sphere's translations and rotations are illustrated in Figure 11.

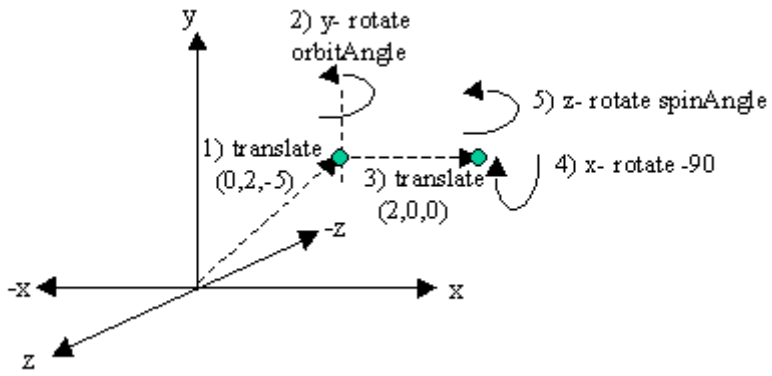

Figure 11. Translating and Rotating the Earth.

The five transformations are numbered in the figure and in the code.

The earth orbits around (0, 2, -5), with a y-axis rotation using orbitAngle, at a distance of 2 units.

Unfortunately, the automatic texturing for a quadric sphere wraps the texture around the z-axis so the top of the texture (the Arctic region) is facing out of the scene along the positive z-axis. This orientation is shown in Figure 12, which employs a version of drawSphere() without the x-axis rotation labelled as operation 4 in Figure 11.

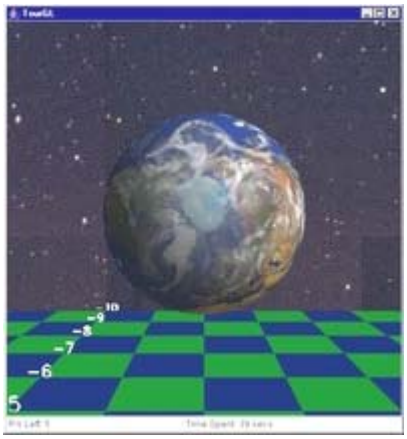

Figure 12. The Earth on its Side.

The simplest solution is to rotate the sphere 90 degrees counter-clockwise around its x-axis so the texture is facing upwards. This also means that what appears to be the earth's y-axis (straight up) is really the z-axis. Consequently, the earth's y-axis spin must be coded as a spin around its z-axis, with the rotation amount supplied by spinAngle.

13. The SkyBox

The skybox is a series of quads acting as four walls and a ceiling. Each wall is covered with two copies of a stars texture, and the ceiling with four copies. The walls and ceiling, with texture divisions, are shown in Figure 13.

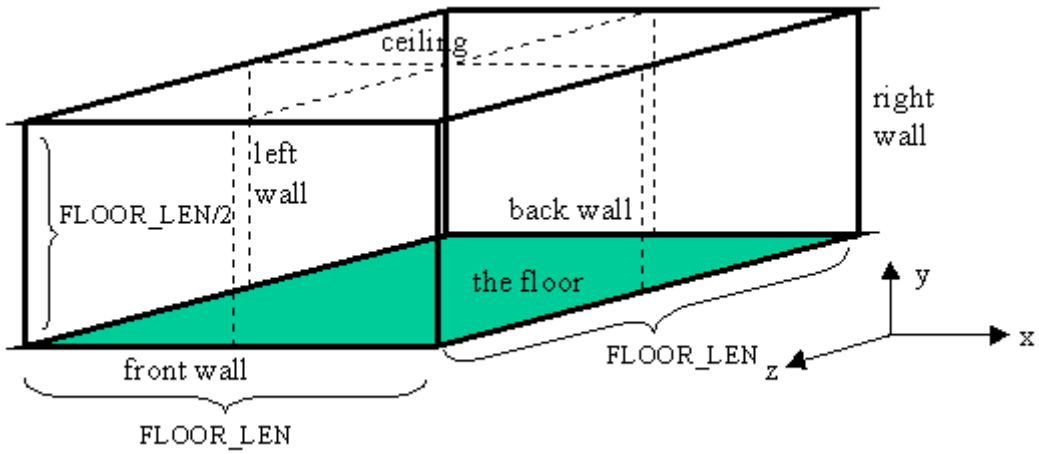

Figure 13. The Skybox with Texture Divisions.

The walls and ceiling are edged in thick lines in Figure 13, while the texture divisions are dotted lines. The walls and ceiling are divided so that each texture will occupy a square of sides FLOOR_LEN/2.

The texture divisions can also be seen in Figure 14, where the stars texture has been replaced by the R image. I've also disabled the constraints on camera movement so the camera can be moved off the floor to get an overview of the entire scene.

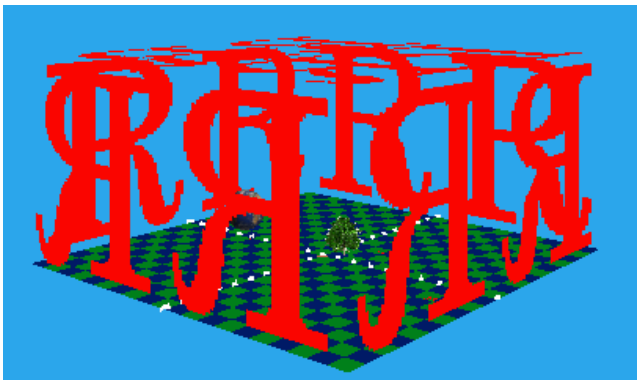

Figure 14. A SkyBox of Red 'R's.

The background colour was altered to blue using GL.glClearColour() in initRender():

```
gl.glClearColor(0.17f, 0.65f, 0.92f, 0.0f);
```

The R's are orientated so they face inwards, since the user will be looking at them from inside the scene. This means that the texture coordinates must be supplied in a clockwise order. Figure 15 shows the ordering of the coordinates for the front face of the skybox.

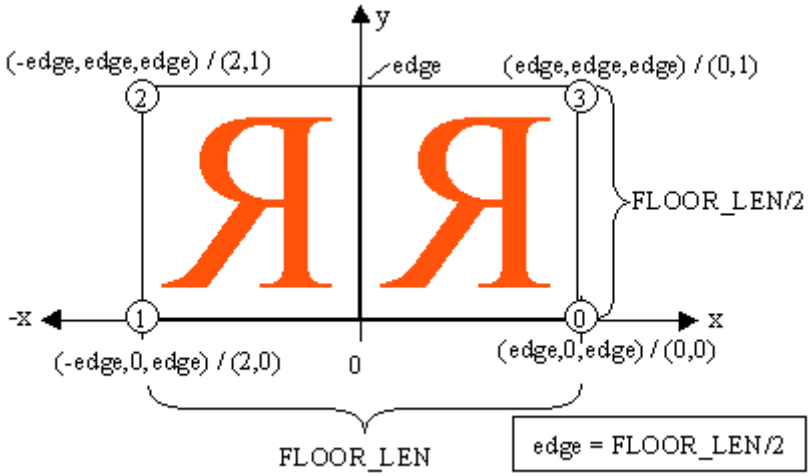

Figure 15. Coordinates for the Front Face of the Skybox.

Multiple copies of the texture are drawn by mapping the physical coordinates to texture coordinates larger than 1 in the s- and/or t- directions. The (s, t) texture coordinates for the front wall are shown after the '/'s in Figure 15.

It's also necessary to switch on texture repetition in loadTextures():

```
starsTex.setTexParameteri(GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
starsTex.setTexParameteri(GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT);
```

drawStars() creates the skybox:

```
private void drawStars()
{
  gl.glDisable(GL.GL_LIGHTING);

  // enable texturing and choose the 'stars' texture
  gl.glEnable(GL.GL_TEXTURE_2D);
  starsTex.bind();

  // replace the quad colours with the texture
  gl.glTexEnvf(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_ENV_MODE,
                                                GL.GL_REPLACE);

  gl.glBegin(GL.GL_QUADS);
    // back wall
    int edge = FLOOR_LEN/2;
    gl.glTexCoord2i(0, 0); gl.glVertex3i(-edge, 0, -edge);
```



```
    gl.glTexCoord2i(2, 0); gl.glVertex3i(edge, 0, -edge);
    gl.glTexCoord2i(2, 1); gl.glVertex3i(edge, edge, -edge);
    gl.glTexCoord2i(0, 1); gl.glVertex3i(-edge, edge, -edge);

    // right wall
    gl.glTexCoord2i(0, 0); gl.glVertex3i(edge, 0, -edge);
    gl.glTexCoord2i(2, 0); gl.glVertex3i(edge, 0, edge);
    gl.glTexCoord2i(2, 1); gl.glVertex3i(edge, edge, edge);
    gl.glTexCoord2i(0, 1); gl.glVertex3i(edge, edge, -edge);

    // front wall
    gl.glTexCoord2i(0, 0); gl.glVertex3i(edge, 0, edge);
    gl.glTexCoord2i(2, 0); gl.glVertex3i(-edge, 0, edge);
    gl.glTexCoord2i(2, 1); gl.glVertex3i(-edge, edge, edge);
    gl.glTexCoord2i(0, 1); gl.glVertex3i(edge, edge, edge);

    // left wall
    gl.glTexCoord2i(0, 0); gl.glVertex3i(-edge, 0, edge);
    gl.glTexCoord2i(2, 0); gl.glVertex3i(-edge, 0, -edge);
    gl.glTexCoord2i(2, 1); gl.glVertex3i(-edge, edge, -edge);
    gl.glTexCoord2i(0, 1); gl.glVertex3i(-edge, edge, edge);

    // ceiling
    gl.glTexCoord2i(0, 0); gl.glVertex3i(edge, edge, edge);
    gl.glTexCoord2i(2, 0); gl.glVertex3i(-edge, edge, edge);
    gl.glTexCoord2i(2, 2); gl.glVertex3i(-edge, edge, -edge);
    gl.glTexCoord2i(0, 2); gl.glVertex3i(edge, edge, -edge);
  gl.glEnd();

  // switch back to modulation of quad colours and texture
  gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_TEXTURE_ENV_MODE,
                                              GL.GL_MODULATE);

  gl.glDisable(GL.GL_TEXTURE_2D);
  gl.glEnable(GL.GL_LIGHTING);
} // end of drawStars()
```

14. Finishing the Game

The gameOver boolean is checked at the end of renderScene():

```
if (gameOver)
  gameOverMessage();
```

The game-over message consists of a robot image with a transparent background, and the player's score inside a red rectangle (see Figure 3). The message stays at the front of the game, even if the camera is moved about.

2D viewing is turned on while the message is being drawn, then switching back to 3D afterwards; a technique I've borrowed from an example posted by ozak at http://www.javagaming.org/forums/index.php?topic=8110.0.

```
private void gameOverMessage()
{
  gl.glDisable(GL.GL_LIGHTING);
```



```
  String msg = "Game Over. Your Score: " + score;
  int msgWidth =
       glut.glutBitmapLength(GLUT.BITMAP_TIMES_ROMAN_24, msg);
       // use a bitmap font (since no scaling required)

  // get (x,y) for centering the text on screen
  int x = (panelWidth - msgWidth)/2;
  int y = panelHeight/2;

  begin2D();  // switch to 2D viewing

  drawRobotImage(x+msgWidth/2, y-12, msgWidth*1.5f);

  // draw a medium red rectangle, centered on the screen
  gl.glColor3f(0.8f, 0.4f, 0.3f);   // medium red
  gl.glBegin(GL.GL_QUADS);
    gl.glVertex3i(x-10, y+10, 0);
    gl.glVertex3i(x+msgWidth+10, y+10, 0);
    gl.glVertex3i(x+msgWidth+10, y-24, 0);
    gl.glVertex3i(x-10, y-24, 0);
  gl.glEnd();

  // write the message in the center of the screen
  gl.glColor3f(1.0f, 1.0f, 1.0f);   // white text
  gl.glRasterPos2i(x, y);
  glut.glutBitmapString(GLUT.BITMAP_TIMES_ROMAN_24, msg);

  end2D();  // switch back to 3D viewing

  gl.glEnable(GL.GL_LIGHTING);
}  // end of gameOverMessage()
```

The message employs GLUT library methods for bitmap strings. Bitmap fonts can be rendered quickly, but don't have 3D depth, and can't be transformed (e.g. rotated and scaled). These drawbacks don't matter when the text is being used as part of a 2D overlay, as here.

The begin2D() method switches the view to use an orthographic projection, which is suitable for 2D-style rendering. Orthographic projection doesn't perform perspective correction, so objects close or far from the camera appear the same size. This is useful for 2D or isometric gaming.

```
private void begin2D()
{
  gl.glMatrixMode(GL.GL_PROJECTION);
  gl.glPushMatrix();   // save projection settings
  gl.glLoadIdentity();
  gl.glOrtho(0.0f, panelWidth, panelHeight, 0.0f, -1.0f, 1.0f);
          // left, right, bottom, top, near, far

  /* In an orthographic projection, the y-axis runs from
     the bottom-left, upwards. This is reversed to the
     more familiar top-left, downwards, by switching the
     the top and bottom values in glOrtho().
  */
  gl.glMatrixMode(GL.GL_MODELVIEW);
  gl.glPushMatrix();   // save model view settings
```



```
  gl.glLoadIdentity();
  gl.glDisable(GL.GL_DEPTH_TEST);
} // end of begin2D()
```

One concern is to save the 3D projection and model view settings so they can be restored at the end of gameOverMessage(). They're pushed onto their respective matrix stacks with GL.glPushMatrix() calls, and retrieved with GL.glPopMatrix() in end2D().

The orthographic projection can be thought of as a 2D drawing surface with its y-axis starting at the *bottom* of the screen and increasing upwards. This is the opposite of the y-axis used by Java when doing 2D painting into a JPanel or Canvas.

The orthographic y-axis can be reversed with GL.glOrtho(). The call in begin2D() is:

```
gl.glOrtho(0.0f, panelWidth, panelHeight, 0.0f, -1.0f, 1.0f);
```

GL.glOrtho() defines a *viewing volume* for the projection – a rectangular box within which things are drawn. The x-axis of the volume is defined by the first two arguments (its left and right edges), the y-axis by the third and fourth arguments (bottom and top), and the z-axis by the last two arguments (front and back of the box).

The GL.glOrtho() call specifies that the y-axis 'bottom' is panelHeight, and its 'top' is 0. This means that the 'bottom' is actually the top of the orthographic volume, and extends downwards to y == 0 (the 'top'). This reverses the y-axis in the projection, making it correspond to the one used by Java components.

Subsequent (x,y) calculations for positioning the image, rectangle, and text, can now use the familiar Java 2D view of things: the x-axis runs left-to-right and the y-axis top-to-bottom.

The drawing of the robot image utilizes drawScreen() again, so the robot's transparent background won't be rendered.

```
private void drawRobotImage(float xc, float yc, float size)
/* The screen is centered on (xc,yc), with sides of length size,
   and standing upright. */
{
  float[] verts = { xc-size/2, yc+size/2, 0,
                    xc+size/2, yc+size/2, 0,
                    xc+size/2, yc-size/2, 0,
                    xc-size/2, yc-size/2, 0};
  drawScreen(verts, robotTex);
}
```

end2D() pops the stored projection and model view matrices from their stacks, and restores them.

```
private void end2D()
// switch back to 3D viewing
{
  gl.glEnable(GL.GL_DEPTH_TEST);
  gl.glMatrixMode(GL.GL_PROJECTION);
```



```
  gl.glPopMatrix();   // restore previous projection settings
  gl.glMatrixMode(GL.GL_MODELVIEW);
  gl.glPopMatrix();   // restore previous model view settings
} // end of end2D()
```

15. Drawing the Floor

The floor consists of a series of green and blue tiles in a checkerboard pattern, a single red square over the origin, and x- and z- axes labels.

```
// globals
private final static int BLUE_TILE = 0;   // floor tile colour types
private final static int GREEN_TILE = 1;

private void drawFloor()
{
  gl.glDisable(GL.GL_LIGHTING);

  drawTiles(BLUE_TILE);
  drawTiles(GREEN_TILE);
  addOriginMarker();
  labelAxes();

  gl.glEnable(GL.GL_LIGHTING);
}  // end of CheckerFloor()
```

15.1. Drawing the Tiles

The trick to tiles creation is the calculation of their coordinates. Figure 16 shows a simplified view of the floor from above (B is for blue tile, G for green).

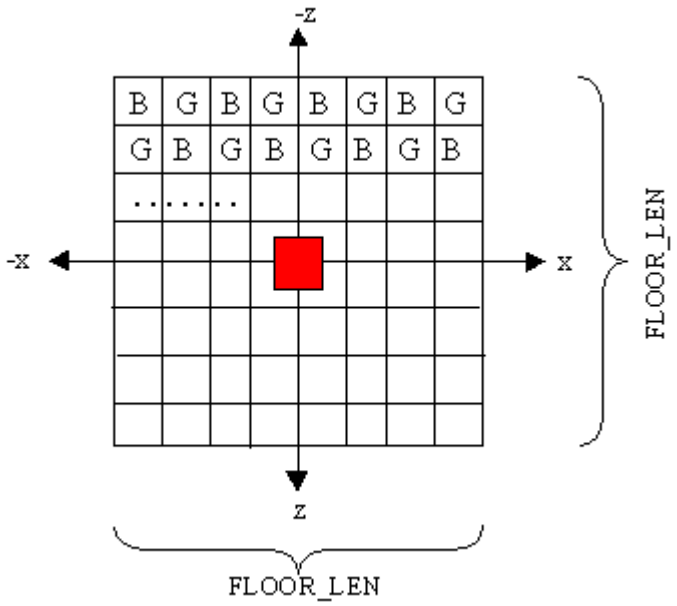

Figure 16. The Floor from Above.

drawTiles() calculates the top-left (x,z) coordinate for all the tiles of the given colour type. Each coordinate is passed to drawTile() to generate the other three coordinates for a tile.

```
private void drawTiles(int drawType)
{
  if (drawType == BLUE_TILE)
    gl.glColor3f(0.0f, 0.1f, 0.4f);
  else  // green
    gl.glColor3f(0.0f, 0.5f, 0.1f);

  gl.glBegin(GL.GL_QUADS);
  boolean aBlueTile;
  for(int z=-FLOOR_LEN/2; z <= (FLOOR_LEN/2)-1; z++) {
    aBlueTile = (z%2 == 0)? true : false;   // set colour for new row
    for(int x=-FLOOR_LEN/2; x <= (FLOOR_LEN/2)-1; x++) {
      if (aBlueTile && (drawType == BLUE_TILE))
        drawTile(x, z);             // blue tile and drawing blue
      else if (!aBlueTile && (drawType == GREEN_TILE))   // green
        drawTile(x, z);
      aBlueTile = !aBlueTile;
    }
  }
  gl.glEnd();
}  // end of drawTiles()
```

The (x,z) coordinate passed to drawTile() is the top-left corner of the desired quad, and the sides of the quad will be unit length. drawTile() creates a quad with the coordinates shown in Figure 17.

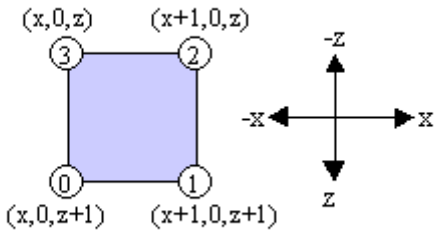

Figure 17. Coordinates for a Single Tile.

The numbered circles in Figure 17 indicate the order the vertices are passed to OpenGL. They're supplied in counter-clockwise order so the resulting quad faces upwards.

The drawTile() code:

```
private void drawTile(int x, int z)
{
  // points created in counter-clockwise order
  gl.glVertex3f(x, 0.0f, z+1.0f);   // bottom left point
  gl.glVertex3f(x+1.0f, 0.0f, z+1.0f);
  gl.glVertex3f(x+1.0f, 0.0f, z);
  gl.glVertex3f(x, 0.0f, z);
```



```
}  // end of drawTile()
```

The red origin square, centered at (0, 0.01, 0) with sides 0.5, is created with code very similar to drawTile():

```
private void addOriginMarker()
{
  gl.glColor3f(0.8f, 0.4f, 0.3f);   // medium red
  gl.glBegin(GL.GL_QUADS);

  // points created counter-clockwise, a bit above the floor
  gl.glVertex3f(-0.25f, 0.01f, 0.25f);  // bottom left point
  gl.glVertex3f(0.25f, 0.01f, 0.25f);
  gl.glVertex3f(0.25f, 0.01f, -0.25f);
  gl.glVertex3f(-0.25f, 0.01f, -0.25f);

  gl.glEnd();
} // end of addOriginMarker();
```

The red square is raised slightly above the XZ plane (0.01 up the y-axis) so it's visible above the tiles.

15.2. Drawing the Axes

labelAxes() places numbers along the x- and z-axes at the integer positions. The axes labels are drawn using a stroke font, so they can be scaled, and the thickness of the line increased (to make the labels bolder). Figure 18 shows some of the numbers.

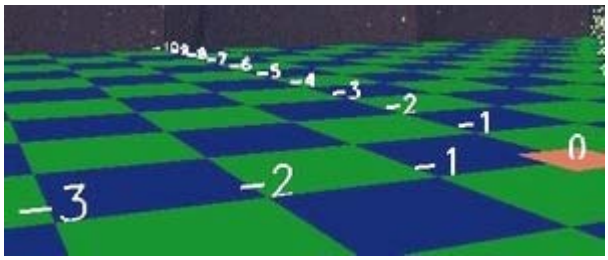

Figure 18. Some of the Axes Labels.

```
private void labelAxes()
{
  gl.glColor3f(1.0f, 1.0f, 1.0f);   // white
  gl.glLineWidth(3.0f);  // thicken the line

  for (int i=-FLOOR_LEN/2; i <= FLOOR_LEN/2; i++)
    drawAxisText(""+i, (float)i, 0.0f, 0.0f);  // along x-axis

  for (int i=-FLOOR_LEN/2; i <= FLOOR_LEN/2; i++)
    drawAxisText(""+i, 0.0f, 0.0f, (float)i);  // along z-axis

  gl.glLineWidth(1.0f);  // reset line width
}
```

drawAxisText() draws the specified text string at a given coordinate, with the text centered in the x-direction, and facing along the +z axis.


```
private void drawAxisText(String txt, float x, float y, float z)
{
  gl.glPushMatrix();
  gl.glTranslatef(x, y, z);    // position the text
  gl.glScalef(0.0015f, 0.0015f, 0.0015f);    // reduce rendering size

  // center text on the x-axis
  float width = glut.glutStrokeLength(GLUT.STROKE_MONO_ROMAN, txt);
  gl.glTranslatef(-width/2.0f, 0, 0);

  // render the text using a stroke font
  for (int i = 0; i < txt.length(); i++) {
    char ch = txt.charAt(i);
    glut.glutStrokeCharacter(GLUT.STROKE_MONO_ROMAN, ch);
  }

  gl.glPopMatrix();   // restore model view
} // end of drawAxisText()
```

Since the text needs to be translated and scaled, a GLUT *stroke* font is used (you may recall I used a GLUT *bitmap* font for the game-over message).

The scaling factor (0.0015) was arrived at by experimentation. A full size character is almost too big to be seen.

