JOGL Chapter 4. Stereo Viewing

Although JOGL can create wonderful 3D worlds, we normally only get to see them flattened onto a 2D computer screen.

True 3D (stereo) viewing is available in OpenGL (and JOGL) with the help of fancy quad-buffering hardware (e.g. the NVIDIA Quadro and ATI FireGL ranges), and drivers for LCD shutter glasses. This kind of technology isn't cheap, but stereoscopic viewing is also possible on ordinary PC hardware, provided the user can train their eyes to look at the screen in a certain way.

The StereoGL application shown in Figure 1 displays two views of a 3D scene, each offset slightly to match the different viewpoints of the user's left and right eyes.

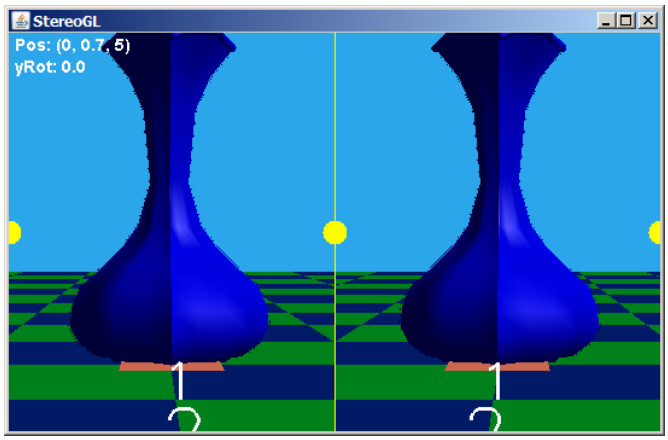

Figure 1. The StereoGL Application.

When the user looks at the images with his eyes aimed forward (almost in parallel), the two images blend into a single stereoscopic image. If 'parallel' viewing is too hard (it does take some practice to master), then StereoGL can be called in 'cross-eyed' mode: the left and right images are switched, and the stereo effect is achieved by the user crossing his eyes. Figure 2 illustrates the two viewing techniques.

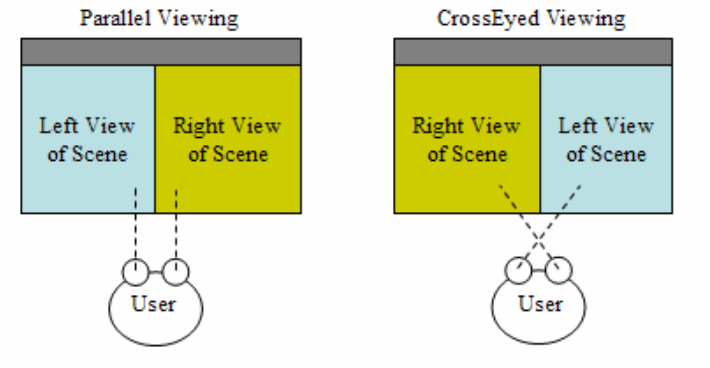

Figure 2. Two Ways of Looking at StereoGL.

This stereoscopic mechanism can be added to any 3D application with only minor changes to the code. StereoGL is based on the TourModelsGL example from the previous chapter, with the addition of a new KeyStereoCamera class to navigate through the scene, and EyeCamera to represent the view for each eye.

StereoGL also employs JOGL's Overlay utility class, added to JOGL 1.0, and modified slightly in version 1.1. It's used to draw the thin yellow vertical line between the views, the three yellow circles across the middle of the window, and the camera position and rotation information in the top-left corner.

How to do Parallel and Cross-eyed Viewing

StereoGL requires the user to learn parallel or cross-eyed viewing. The following technique for parallel viewing works for me:

1. Put your face close to the monitor.
2. Focus your eyes into the distance; don't try to focus on the screen.
3. Slowly move back from the screen, while still maintaining a distant focus.
4. Allow the combined stereo image to gradually come into focus.

It can take up to a minute for step 4 to happen.

For cross-eyed viewing, the yellow circles drawn in the window can help:

1. Stare at the middle circle between the two viewports.
2. Slowly cross your eyes until the two copies of the middle circle move out to join the circles at the edges.
3. Allow the combined stereo image to come into focus.

It may help to reduce the window's size when first attempting cross-eyed viewing.

1. StereoGL Overview

Figure 3 shows the class diagrams for the StereoGL application. Only public methods are shown.

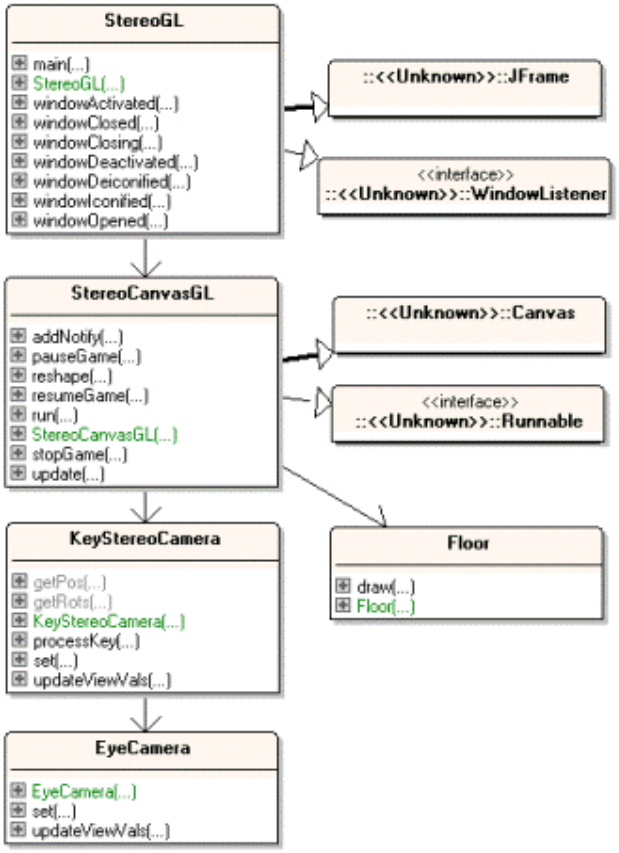

Figure 3. Class Diagrams for StereoGL.

StereoGL creates the JFrame and JPanel around the heavy-weight StereoCanvasGL canvas, and handles window events such as resizing and iconification. It positions the window in the middle of the screen, and keeps it there if the window is resized, which helps the user to keep their eyes focused correctly.

StereoCanvasGL implements the active rendering framework: it spawns a single thread which initializes rendering, then loops, carrying out an update/render/sleep cycle with a fixed period. I won't be explaining the framework again, since it's the same as in previous examples.

StereoCanvasGL is somewhat simpler than earlier 'canvas' classes, since I've deleted the code that gathers frame rate statistics, and the floor-generation code has been moved to a Floor class. Aside from being in a separate class, the floor code is unchanged from earlier examples, so won't be explained again either.

StereoCanvasGL displays three models (a penguin, rose and vase, and couch), loaded with the OBJLoader package from the last chapter, but this time the canvas doesn't support picking or penguin singing.

The camera navigation code, last seen in TourModelsGL, has been moved into the KeyStereoCamera class. It converts key presses into camera movements forwards, backwards, left, and right. It also manages the two views of the scene, via two instances of the EyeCamera class.

EyeCamera is where stereo viewing is implemented. It creates a viewport, an asymmetric viewing frustum, and moves the camera to the left or right depending on whether the EyeCamera instance is for the left or right eye.

In the rest of this chapter, I explain how stereoscopic viewing relates to the camera's viewing frustum and position. I show how these ideas are implemented in EyeCamera. KeyStereoCamera, which manages the two EyeCamera instances, is examined, and I finish by considering the novel parts of StereoCanvasGL (including the Overlay code).

2. Stereoscopic Viewing

In most JOGL applications, the camera's viewing frustum for a perspective projection is like the one in Figure 4.

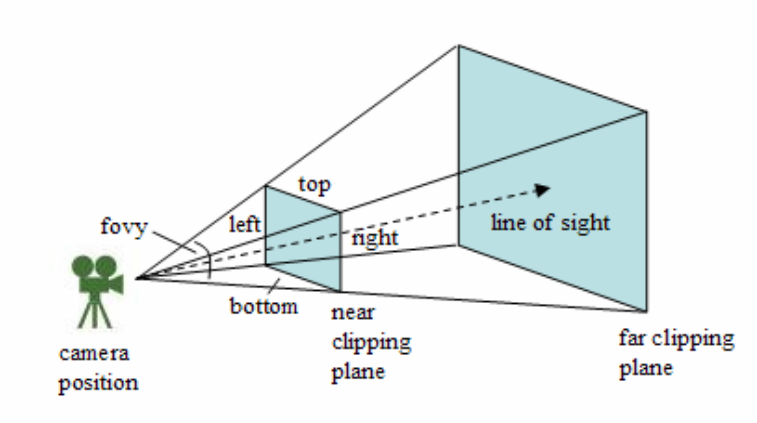

Figure 4. A Viewing Frustum.

The frustum can be set up with the GL.glFrustum() method:

```
GL.glFrustum(double left, double right, double bottom,
             double top, double near, double far)
```

It specifies the near and far clipping planes, as measured from the camera position. The left, right, top, and bottom values are relative to the line of sight which passes through the center of the near clipping plane.

The top and bottom distances can be obtained from the near distance and the field-of-view (FOV) angle around the y-axis with some simple trigonometry (see Figure 5).

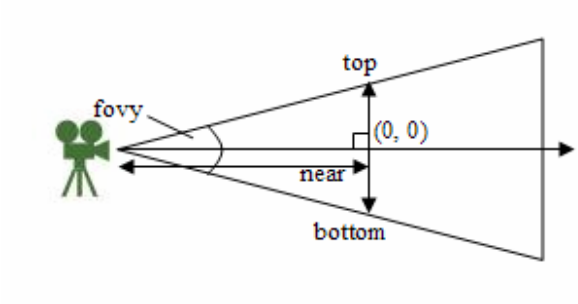

Figure 5. The Viewing Frustum From the Side.

The equations are:

top = tan(fovy/2)*near and bottom = –top

The left and right values for the frustum are calculated from the viewport's aspect ratio:

aspectRatio = viewport width / viewport height

left = bottom * aspectRatio

right = top * aspectRatio

The left and right values need to be recalculated if the viewport dimensions change, which occurs when the window is resized.

Figure 6 shows the frustum from above, and includes the position of the screen (also known as the projection plane, focal length, or fusion distance).

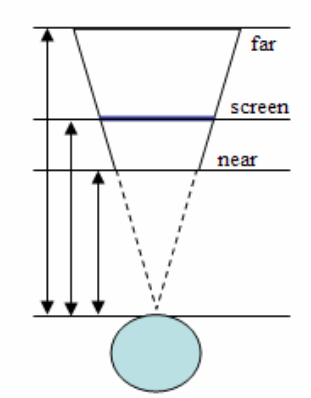

Figure 6. The Viewing Frustum From Above.

The screen may be anywhere in front of the camera, but for first-person viewing will often be further from the camera than the near clipping plane. This reduces the likelihood that scenery will be visibly clipped when the user moves close to it.

For stereo viewing, the view of the screen from each eye will be slightly different, as shown by the dotted triangles in Figure 7.

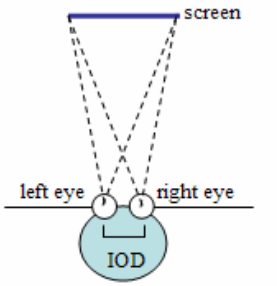

Figure 7. Viewing the Screen with Two Eyes.

IOD is the intraocular distance, the distance between the eyes.

A comparison of the viewing volumes in Figures 6 and 7 (the dotted triangles), shows that the left and right eye's volumes are slanted to the right and left respectively. This asymmetry needs to be implemented in the application.

Figure 8 shows more details about how the viewing frustum of Figure 6 can be slanted to the left to make it suitable for the right eye.

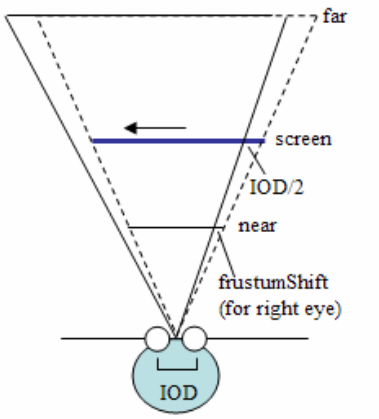

Figure 8. Slanting the Viewing Frustum for the Right Eye.

The original viewing frustum is shown as a dotted triangle, the slanted frustum as a solid triangle.

The viewing volume is shifted IOD/2 units to the left along the screen plane since this is the offset of the right eye from the camera's center point. The screen shift causes a correspondingly smaller shift along the near clipping plane, by a frustumShift amount. The value for frustumShift can be obtained by noting that the shifts along the two planes define similar triangles, as illustrated by Figure 9.

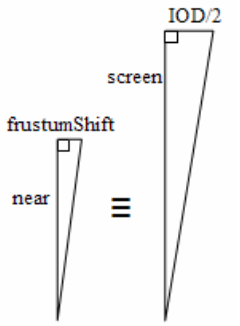

Figure 9. Similar Plane-Shift Triangles.

The similar triangles allow frustumShift to be calculated:

frustumShift = IOD/2 * near/screen

The shift can be utilized in the call to GL.glFrustum() for the right eye:

```
gl.glFrustum(left-frustumShift, right-frustumShift,
             bottom, top, near, far);
```

This changes the viewing volume into an asymmetric frustum for the right eye by *subtracting* frustumShift from the near plane's left and right values. A frustum for the left eye is generated by *adding* frustumShift to its left and right values.

The resulting asymmetric frustums are still centered between the eyes, so must be translated so they coincide with the eye positions (a distance of IOD/2). For example the right eye frustum is moved to the right as shown in Figure 10.

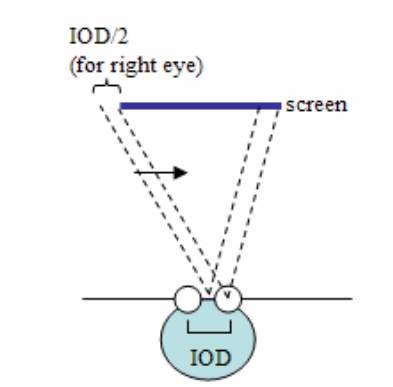

Figure 10. Moving the Right Eye Frustum to the Right.

The IOD/2 translation means that the right-eye frustum will pass through the screen plane at the same coordinates as the original frustum.

The left-eye frustum is shifted by IOD/2 to the left, which means that it will also cut the screen plane at the same coordinates as the original.

These translations can be achieved by supplying a modified x-axis coordinate to GLU.gluLookAt() which sets the camera position. For example, the position for the right eye's camera will be

```
glu.gluLookAt(xPlayer+IOD/2.0, yPlayer, zPlayer,
              xLookAt, yLookAt, zLookAt, 0,1,0);
```

(xPlayer, yPlayer, zPlayer) is the camera position between the eyes, (xLookAt, yLookAt, zLookAt() is the point being looked at, and (0, 1, 0) is the up-direction (the positive y-axis).

3. Building an Eye Camera

The EyeCamera class has two instances, one for the left eye, one for the right. Each EyeCamera is in charge of three things: the viewport for that eye, the asymmetric viewing frustum for the camera, and the camera's offset from the midpoint between the eyes.

Both cameras move in unison relative to the midpoint position when the user navigates around the scene. The left EyeCamera is offset to the left by IOD/2, and the right EyeCamera by IOD/2 to the right.

EyeCamera also deals with updates to its viewport (triggered by the user resizing the StereoGL window). A viewport change requires a recalculation of the near plane's left and right values.

When an EyeCamera instance is created, the frustumShift, top, and bottom values are generated.

```
// globals
// type of camera (public)
public final static int LEFT_EYE = 0;
public final static int RIGHT_EYE = 1;

// camera frustum constants
private static final double IOD = 0.2;
       // intraocular distance (eye separation distance)
private static final double FOVY = 45.0;
private static final double NEAR_DIST = 1.0;
private static final double SCREEN_DIST = 5.0;
private static final double FAR_DIST = 100.0;


private GL gl;
private GLU glu;

private int eyeType;    // eye info

// frustum info
private double top, bottom;
private double frustumShift;
```



```
public EyeCamera(GL gl, GLU glu, int eType)
{
  this.gl = gl;
  this.glu = glu;
  eyeType = eType;

  frustumShift = (IOD/2.0) * NEAR_DIST/SCREEN_DIST;
  top = NEAR_DIST * Math.tan((FOVY/2.0*Math.PI)/360.0);
  bottom = -top;
} // end of EyeCamera()
```

The frustum shift, top, and bottom calculations use the equations described in the previous section.

The values for the IOD and SCREEN_DIST constants were arrived at by experimentation. The aim is to create a frustum shift which makes the stereo effect between the near and far objects in the scene look realistic, which is rather subjective.

The eyeType value can be LEFT_EYE or RIGHT_EYE, and is used in later methods to determine the direction of the frustum slanting and the camera offset.

3.1. Resizing the Viewport

When the window is first drawn, and whenever it is resized, the viewport dimensions are passed to EyeCamera via updateViewVals():

```
// globals
// viewport dimensions
private int xVP, yVP, widthVP, heightVP;

// frustum info
private double top, bottom, left, right;
private double frustumShift;

public void updateViewVals(int x, int y, int w, int h)
{
  // update viewport coordinates
  xVP = x; yVP = y;
  widthVP = w;
  if (h == 0)
    h = 1;
  heightVP = h;

  double aspectRatio = (double)widthVP/(double)heightVP;

  // calculate left and right near clipping plane dimensions
  left = bottom * aspectRatio;
  right = top * aspectRatio;

  // modify left and right to specify an asymmetric frustum
  if (eyeType == LEFT_EYE) {
    left += frustumShift;    // shift left eye frustum to the right
    right += frustumShift;
  }
  else {  // right eye
    left -= frustumShift;    // shift right eye frustum to the left
```



```
    right -= frustumShift;
  }
}  // end of updateViewVals()
```

The new viewport numbers are saved, and the frustum's left and right values are recalculated based on the viewport's new aspect ratio. The frustum shift is also added, or subtracted, depending on the eye type.

3.2. Setting up the Camera

The camera is defined by its viewport, frustum, and position. These are set at render-time by set():

```
public void set(double xPlayer, double yPlayer, double zPlayer,
                double xLookAt, double yLookAt, double zLookAt)
/* Set the viewport, perspective projection frustum for this
   camera, and its position. */
{
  gl.glViewport(xVP, yVP, widthVP, heightVP);

  gl.glMatrixMode(GL.GL_PROJECTION);  // set the view attributes
  gl.glLoadIdentity();

  gl.glFrustum(left, right, bottom, top, NEAR_DIST, FAR_DIST);
           // specifies an asymmetric frustum view volume

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

  // adjust for x-axis eye offset
  double xEye;
  if (eyeType == LEFT_EYE)
    xEye = xPlayer - IOD/2.0;  // shift left eye to the left
  else  // right eye
    xEye = xPlayer + IOD/2.0;  // shift right eye to the right

  glu.gluLookAt(xEye, yPlayer, zPlayer,
                xLookAt, yLookAt, zLookAt, 0,1,0); // posn camera
} // end of set()
```

set() uses GL.glViewport(), GL.glFrustum() and GLU.gluLookAt() to define the viewport, frustum, and camera position.

4. Managing the Camera

KeyStereoCamera manages the two EyeCamera instances, passing them information about their viewports, and the position of the midpoint between the eyes.

KeyStereoCamera also handles user navigation: it converts the arrow keys, CTRL key, and 'r' key into translations forwards, backwards, left, and right, and turns to the left and right around the y-axis. The code is taken from the processKey() method in TourGL (JOGL Chapter 2), so won't be explained again.

The KeyStereoCamera constructor creates the two EyeCameras, and records whether the user is utilizing parallel or cross-eyed viewing.

```
// globals
private boolean isCrossed;
private EyeCamera leftCamera, rightCamera;


public KeyStereoCamera(GL gl, GLU glu, boolean isCrossed)
{
  this.isCrossed = isCrossed;
  if (isCrossed)
    System.out.println("Using cross-eyed viewing");
  else
    System.out.println("Using parallel viewing");

  // create the eye cameras
  leftCamera = new EyeCamera(gl, glu, EyeCamera.LEFT_EYE);
  rightCamera = new EyeCamera(gl, glu, EyeCamera.RIGHT_EYE);

  reset();
}  // end of KeyStereoCamera()
```

reset() sets the player's starting position and orientation:

```
// globals
// initial player position
private static final double X_START = 0.0;
private static final double Y_START = 0.7;
private static final double Z_START = 5.0;

private final static double LOOK_AT_DIST = 100.0;

// player positioning
private double xPlayer, yPlayer, zPlayer;
private double xLookAt, yLookAt, zLookAt;
private double xStep, zStep;
private double viewAngle;

private void reset()
// set player starting position and orientation
{
  xPlayer = X_START;
  yPlayer = Y_START;
  zPlayer = Z_START;

  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 reset()
```

The coding approach is the same as in JOGL Chapter 2, where the player is defined using its position (xPlayer, yPlayer, zPlayer), the point being looked at (xLookAt,

yLookAt, zLookAt), the step along the x- and z- axes, and the angle the player is rotated around the y-axis relative to the positive x-axis.

4.1. Updating the Viewport Dimensions

Window resizing triggers a call to KeyStereoCamera's updateViewVals(), which passes on the new window size (its width and height) to the left and right EyeCameras.

```
public void updateViewVals(int width, int height)
{
  if (isCrossed) {  // cross-eyed viewing  (right | left position)
    rightCamera.updateViewVals(0, 0, width/2, height);
    leftCamera.updateViewVals(width/2, 0, width/2, height);
  }
  else {  // parallel viewing  (left | right position)
    leftCamera.updateViewVals(0, 0, width/2, height);
    rightCamera.updateViewVals(width/2, 0, width/2, height);
  }
}  // end of updateViewVals()
```

The isCrossed boolean determines where the left and right eye camera viewports should be located in the window. Parallel viewing will cause the left eye camera to be placed in the left half of the window, and the right eye camera in the right hand half. Cross-eyed viewing will cause the cameras to be switched. Figure 11 shows the camera positions for the two viewing styles.

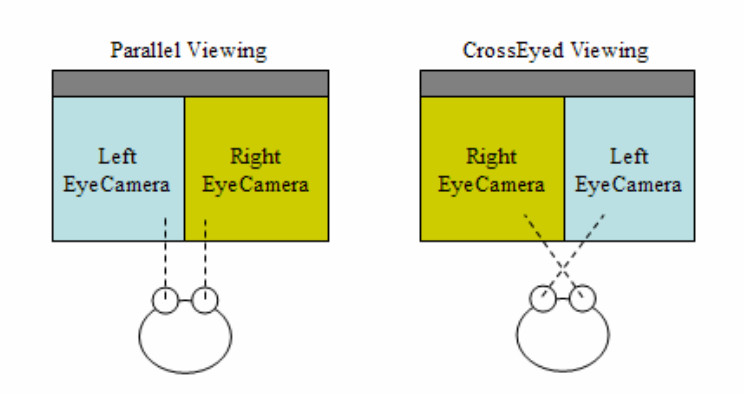

Figure 11. The Viewports for the Two Viewing Styles.

4.2. Setting the Eye Cameras

The rendering loop calls KeyStereoCamera.set() to set the left or right eye cameras.

```
public void set(int eyeType)
/* Set the viewport and perspective projection frustum for
   a camera, and its position. */
{
  if (eyeType == EyeCamera.LEFT_EYE)   // left eye
    leftCamera.set(xPlayer, yPlayer, zPlayer,
                           xLookAt, yLookAt, zLookAt);
  else  // right eye
```



```
    rightCamera.set(xPlayer, yPlayer, zPlayer,
                            xLookAt, yLookAt, zLookAt);
} // end of set()
```

KeyStereoCamera passes the current player position (and look-at point) to the relevant eye camera which then offsets the position to the left or right depending on the camera type.

5. Rendering the Scene

The scene created by StereoCanvasGL is very similar to the one in earlier chapters, so I'll limit myself to a description of the new elements only: how KeyStereoCamera is utilized, and the use of JOGL's Overlay utility class.

5.1. Initializing the Camera

Initialization tasks, such as defining the lighting and loading the OBJ models, are performed in initRender() in StereoCanvasGL. KeyStereoCamera is also instantiated there, and supplied with the current window dimensions:

```
// globals in StereoCanvasGL
private KeyStereoCamera keyCamera;
private boolean isCrossed;
private int panelWidth, panelHeight;

// in initRender()
keyCamera = new KeyStereoCamera(gl, glu, isCrossed);
keyCamera.updateViewVals(panelWidth, panelHeight);
```

The isCrossed value is true or false depending on whether the user has included a "-x" on the command line. The boolean is passed through to the KeyStereoCamera instance to determine where the cameras' viewports should be placed in the window.

5.2. Dealing with User Input

The StereoCanvasGL constructor sets up a KeyAdapter, which passes key presses to processKey().

```
// in the StereoCanvasGL constructor
addKeyListener( new KeyAdapter() {
  public void keyPressed(KeyEvent e)
  { processKey(e); }
});
```

processKey() deals with the termination keys (ESC, 'q', END, CTRL-c) by stopping the rendering loop, but the other keys are sent to the KeyStereoCamera instance for processing as navigation commands.

```
private void processKey(KeyEvent e)
```



```
{
  int keyCode = e.getKeyCode();

  /* termination keys: 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;
  else {  // navigation keys
    if (keyCamera == null)
      System.out.println("Ignoring key press");
    else
      keyCamera.processKey(e);
  }
}  // end of processKey()
```

The test to check if keyCamera is null is included since the object is created at rendering initialization time (in initRender()), and there's a slim chance that the user presses a key before that method has finished.

5.3. Rendering the Scene

StereoCanvasGL's rendering loop calls renderScene(). In previous examples, this method drew the scene (as the name suggests). However, the StereoGL window has two viewports, so StereoCanvasGL's version of the method must render the scene *twice*, once in each viewport. It must also draw the overlay information (the yellow line, circles, and text) across the window.

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

  if (isResized) {
    // Update the viewport and frustum dimensions in the camera
    keyCamera.updateViewVals(panelWidth, panelHeight);
    isResized = false;
  }

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

  renderEyeScene(EyeCamera.LEFT_EYE);  // left eye view of scene
  renderEyeScene(EyeCamera.RIGHT_EYE); // right eye

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

renderEyeScene() draws the floor and models after setting up the relevant eye camera.

```
private void renderEyeScene(int eyeType)
// draw the scene for the given eye camera
{
  keyCamera.set(eyeType);
```



```
      // set the camera's viewport, frustum, and position

  // set light direction
  gl.glLightfv(GL.GL_LIGHT0, GL.GL_POSITION, LIGHT_DIR, 0);

  // draw the scenery
  gl.glPushMatrix();
    floor.draw();
    drawModels();
  gl.glPopMatrix();
}  // end of renderEyeScene()
```

5.4. Adding an Overlay

I've coded with overlays previously – I used one in JOGL Chapter 2 to place a game-over message on the screen. However, I wrote it from scratch, which is no longer necessary since the release of JOGL v1.0 with its Overlay utility class. The class was slightly modified in release v1.1 (April 2007) when one of its method names was changed (sync() became markDirty()).

Overlay is built on top of the TextureRenderer utility class which provides a convenient way to dynamically render into a texture using Java 2D calls. A Graphics2D reference is obtained from the Overlay object, and text and graphics are drawn into it to build up the overlay image.

I instantiate the Overlay object in StereoCanvasGL's constructor, along with a font for the overlay's text:

```
// globals
private Overlay infoOverlay;
private Font font;

// in the StereoCanvasGL constructor
infoOverlay = new Overlay(drawable);
font = new Font("SansSerif", Font.BOLD, 14);
```

The overlay image is created by showOverlayInfo(), which draws a thin vertical line dividing the two camera views, three circles across the middle of the window to help the user align his eyes, and writes camera details in the top left corner (see Figure 1).

```
private void showOverlayInfo()
{
  // draw over the entire window
  gl.glViewport(0, 0, panelWidth, panelHeight);
  gl.glMatrixMode(GL.GL_PROJECTION);
  gl.glLoadIdentity();

  Graphics2D g2d = infoOverlay.createGraphics();

  // clear the overlay
  g2d.setComposite(AlphaComposite.Src);
  g2d.setColor(TRANSPARENT_BLACK);
  g2d.fillRect(0, 0, panelWidth, panelHeight);

  g2d.setColor(Color.YELLOW);
```



```
  g2d.drawLine(panelWidth/2, 0, panelWidth/2, panelHeight);
                                          // draw a vertical line
  // draw three eye alignment circles
  int len = CIRCLE_RADIUS*2;
  int yPosn = panelHeight/2-CIRCLE_RADIUS;

  g2d.fillOval(-CIRCLE_RADIUS, yPosn, len, len); // middle left edge
  g2d.fillOval(panelWidth/2-CIRCLE_RADIUS, yPosn, len, len);
                                                  // middle of window
  g2d.fillOval(panelWidth-CIRCLE_RADIUS, yPosn, len, len);
                                                 // middle right edge
  // draw camera info (in white)
  g2d.setColor(Color.WHITE);
  g2d.setFont(font);
  g2d.drawString( keyCamera.getPos(), 5, 15);
  g2d.drawString( keyCamera.getRots(), 5, 33);

  // render all the overlay to the screen
  infoOverlay.markDirty(0, 0, panelWidth, panelHeight);
  infoOverlay.drawAll();

  g2d.dispose();
}  // end of showOverlayInfo()
```

At the start of the method, the viewport is changed to span the entire window.

The overlay is drawn to the screen with Overlay.drawAll(). Just to be on the safe side, I also call Overlay.markDirty() to signal that the entire overlay has been updated.

6. More Information on Stereo Rendering

Paul Bourke's excellent information on stereo rendering with OpenGL and GLUT can be found at http://local.wasp.uwa.edu.au/~pbourke/projection/stereorender/. He describes how to use quad-buffering and the asymmetric frustum approach, and also explains how to generate stereoscopic images. A similar approach is used for medical stereoscopy at http://www.orthostereo.com/geometryopengl.html.

Steve Cunningham utilizes the same left and right viewports approach as my code in his OpenGL example at http://www.cs.csustan.edu/~rsc/NSF/conicstereo.c. He also employs clipping to show various conic sections.

Lesson 42 at the NeHe site (http://nehe.gamedev.net/) describes how to create four viewports in a window, but doesn't use them for stereo viewing. The JOGL version of the lesson is at http://pepijn.fab4.be/?page_id=34

The *anaglyph* is a popular and inexpensive form of stereographic image. The general idea is to render the scene twice, slightly offset from each other for the two eyes. One version is drawn using a red color mask, the other with a blue (or cyan) mask. The user wears two-color glasses (one lens is red, the other blue (or cyan)). This fools the brain into combining the two images in a single stereoscopic view. Paul Bourke describes this approach, with OpenGL code by Daniel van Vugt, at http://local.wasp.uwa.edu.au/~pbourke/texture_colour/anaglyph/.

My own experiments with anaglyphs indicate that they're most effective when the original scene doesn't use color (e.g. is drawn in grays), and is relatively simple (e.g. consists of just a few objects).