



Two JOGL Programming Frameworks

This chapter introduces JOGL (<https://jogl.dev.java.net/>), a Java wrapper for the 3D (and 2D) graphics library OpenGL (<http://www.opengl.org/>). I'll implement a simple example, a rotating multicolored cube, using two programming frameworks, one employing *callbacks*, the other utilizing *active rendering*. One way that I compare them is by seeing how well they handle different frame rates for the cube's animation.

The next chapter explores JOGL features in more detail when I develop an application containing many of the elements you've already seen coded in Java 3D, including a checkerboard floor, a rotating textured sphere, a skybox, a billboard, overlays, and keyboard navigation. Chapter 17 examines how to load OBJ models, implement collision detection, and play 3D sound.

What Is JOGL?

JOGL is one of the open-source technologies initiated by the Game Technology Group at Sun Microsystems back in 2003 (the others are JInput and JOAL, which I cover in Chapters 11 through 14). JOGL provides full access to the APIs in the OpenGL 2.0 specification, as well as vendor extensions, and can be combined with AWT and Swing components. It supports both of the main shader languages, GLSL and Nvidia's Cg.

JOGL has the same focus as OpenGL on 2D and 3D rendering. It doesn't include support for gaming elements such as sound or input devices, which are nicely dealt with by JOAL and JInput.

Most features of the popular OpenGL GLU and GLUT libraries are present in JOGL. GLU (the OpenGL Utility Library) includes support for rendering spheres, cylinders, disks, camera positioning, tessellation, and texture mipmaps. The JOGL version of GLUT (OpenGL Utility Toolkit) doesn't include its windowing functionality, which is handled by Java, but does offer geometric primitives (both in solid and wireframe mode). JOGL's utility classes include frame-based animation, texture loading, file IO, and screenshot capabilities.

JOGL has evolved into the reference implementation for the JSR-231 specification for binding OpenGL to Java (<http://jcp.org/en/jsr/detail?id=231>). JOGL 1.1.1 was superseded by JSR-231 in October 2005, and the current JSR-231 release candidate, 1.1.0-rc2, came out in January 2007. I'll be using that version in the following chapters, but will keep using the name JOGL.

To become JSR-231 compliant, many JOGL classes, methods, and packages have been modified, mostly in minor ways. This means that older examples need some tweaking to get them to compile and run. Details about the changes can be found in the JOGL forum thread <http://www.javagaming.org/forums/index.php?topic=11189.0>.

The new GLDrawable and GLContext classes are the most important for this chapter since they allow direct access to OpenGL's drawing surface and state information. These new classes support a new style of coding, called *active rendering*, which I use as the basis of the second programming framework.

The OpenGL API is accessed via Java Native Interface (JNI) calls, leading to a very direct mapping between the API's C functions and JOGL's Java methods. As a consequence, it's extremely easy to translate most OpenGL examples into JOGL. The drawback is that the OpenGL programming style is based around affecting a global graphics state, which makes it difficult to structure Java code into meaningful classes and objects. JOGL does provide class structuring for the OpenGL API, but the vast majority of its methods are in the very large GL and GLU classes.

OpenGL is a vast, complex, and powerful API, with entire books dedicated to its explanation. In the next three chapters, I'll only explain the OpenGL features I need for my examples. For an all-around knowledge, you'll need other sources, and I point you toward some at the end of this chapter.

Installing JOGL

JOGL will work with J2SE 1.4.2 or later; I used Java 1.6.0 for my tests and downloaded the JSR-231 1.1.0 release candidate 2 of JOGL from <https://jogl.dev.java.net/>. I chose the Windows build from January 23, 2007, `jogl-1.1.0-rc2-windows-i586.zip`, which contains a `lib\` subdirectory holding two JAR files (`jogl.jar` and `gluegen-rt.jar`) and four DLLs (`jogl.dll`, `gluegen-rt.dll`, `jogl_awt.dll`, and `jogl_cg.dll`).

The JOGL user guide (which is part of the ZIP file) recommends that the JARs and DLLs should be installed in their own directory rather than inside the JRE directories. Consequently, I extracted the `lib\` directory, renamed it to `jogl\`, and stored it on my test machine's d: drive (`d:\jogl\`).

The JARs and DLLs can be utilized at compile time and runtime by supplying suitable `classpath` and `java.library.path` parameters on the command line. For example, when I compile the JOGL demo `PrintExt.java`, I type the following:

```
javac -classpath "d:\jogl\jogl.jar;d:\jogl\gluegen-rt.jar;." PrintExt.java
```

Its execution requires the following:

```
java -cp "d:\jogl\jogl.jar;d:\jogl\gluegen-rt.jar;."
    -Djava.library.path="d:\jogl"
    -Dsun.java2d.noddraw=true PrintExt
```

The `java.exe` command is a single line, which I've reformatted so it's easier to read.

The `sun.java2d.noddraw` property disables Java 2D's use of `DirectDraw` on Windows. This avoids any nasty interactions between `DirectDraw` and OpenGL, which can cause application crashes, poor performance, and flickering. The property is only needed if you're working on a Windows platform.

Another useful command-line option is `-Dsun.java2d.opengl=true`, which switches on the Java2D OpenGL pipeline. The pipeline provides hardware acceleration for many Java 2D rendering operations (e.g., text, images, lines, fills, complex transforms, composites, clips). It's essential when JOGL's `GLJPanel` class is employed as a drawing surface (as explained in the "Rotating a `GLJPanel` Cube with Callbacks" section). Unfortunately, `-Dsun.java2d.opengl=true` may cause crashes on older graphics hardware and drivers. If you don't like lengthy command-line arguments, another approach is to modify the `CLASSPATH` environment variable and `PATH` (Windows), `LD_LIBRARY_PATH` (Solaris and Linux), or `DYLD_LIBRARY_PATH` (Mac OS X). More details can be found in the JOGL user guide.

I packaged up the compilation command line in `compileGL.bat`:

```
@echo off
echo Compiling %1 with JOGL...
javac -classpath "d:\jogl\jogl.jar;d:\jogl\gluegen-rt.jar;." %1
```

echo Finished.

The call to java.exe is in runGL.bat:

```
@echo off
echo Executing %1 with JOGL...
java -cp "d:\jogl\jogl.jar;d:\jogl\gluegen-rt.jar;."
      -Djava.library.path="d:\jogl"
      -Dsun.java2d.noddraw=true %1 %2
echo Finished.
```

The batch variables (%1 and %2) allow up to two arguments to be passed to runGL.bat.

The Callback Framework

The two main JOGL GUI classes are GLCanvas and GLJPanel, which implement the GLAutoDrawable interface, allowing them to be utilized as *drawing surfaces* for OpenGL commands.

GLCanvas is employed in a similar way to AWT's Canvas class. It's a heavyweight component, so care must be taken when combining it with Swing. However, it executes OpenGL operations very quickly due to hardware acceleration.

GLJPanel is a lightweight widget that works seamlessly with Swing. In the past, it's gained a reputation for being slow since it copies the OpenGL frame buffer into a BufferedImage before displaying it. However, its speed has improved significantly in Java SE 6, as I show with some timing tests later in the "Timing the GLJPanel" section of this chapter.

A key advantage of GLJPanel over GLCanvas is that it allows 3D graphics (courtesy of OpenGL) and 2D elements in Swing to be combined in new, exciting ways.

Using GLCanvas

A GLCanvas object is paired with a GLEventListener listener, which responds to changes in the canvas and to drawing requests.

When the canvas is first created, GLEventListener's `init()` method is called; this method can be used to initialize the OpenGL state.

Whenever the canvas is resized, including when it's first drawn, GLEventListener's `reshape()` is executed. It can be overridden to initialize the OpenGL viewport and projection matrix (i.e., how the 3D scene is viewed). `reshape()` is also invoked if the canvas is moved relative to its parent component.

Whenever the canvas' `display()` method is called, the `display()` method in GLEventListener is executed. Code for rendering the 3D scene should be placed in that method.

Aside from the canvas and listener, most games will need a mechanism for triggering regular updates to the canvas. This functionality is available through JOGL's FPSAnimator utility class, which can schedule a call to the canvas' `display()` method with a frequency set by the user. All these elements are shown in Figure 15-1.

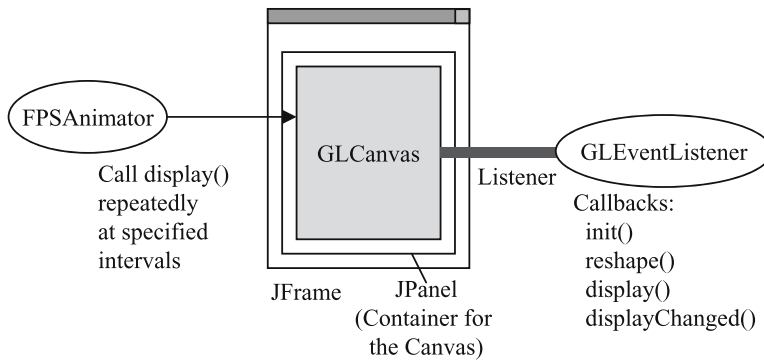


Figure 15-1. A callback application with `GLCanvas`

The `GLCanvas` can be placed directly inside the `JFrame`, but by wrapping it in a `JPanel` the `JFrame` can contain other (lightweight) GUI components as well.

The `GLEventListener` callbacks include `displayChanged()`, which should be called when the display mode or device has changed. This might occur when the monitor's display settings are changed or when the application is dragged to another monitor in a multidisplay configuration. `displayChanged()` is not currently implemented in JOGL.

Missing from Figure 15-1 is how user interactions, such as mouse and keyboard activity, affect the canvas. The basic technique is to set up mouse and keyboard listeners in the usual Java manner and have them change global variables in `GLEventListener`. When its `display()` method is called, it can check these globals to decide how to act. The next chapter has an extended example that employs this approach.

A common source of coding errors with JOGL is to have a mouse or keyboard listener call OpenGL functions directly, which usually results in the application crashing. The OpenGL state can only be safely manipulated via the `GLAutoDrawable` interface, which is exposed in `GLEventListener`'s callback methods. Many vendors' OpenGL drivers aren't that reliable when faced with multithreading so should not be accessed from listener threads.

Using `GLJPanel`

Since the `GLJPanel` is a lightweight Swing component, it can be added directly to the enclosing `JFrame`, as shown in Figure 15-2.

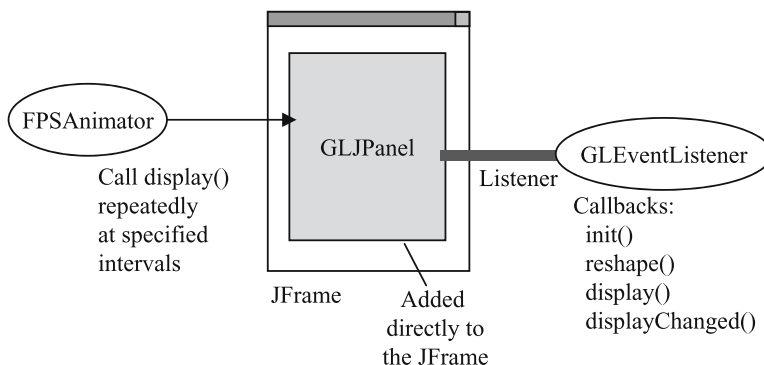


Figure 15-2. A callback application with `GLJPanel`

The rest of the callback framework is identical to Figure 15-1: FPSAnimator drives the animation, and GLEventListener catches changes to the drawing area. This means that it's just a matter of changing a few lines of code to switch between GLCanvas and GLJPanel, as I show in the rotating cube example in the following sections.

A commonly used variant of Figure 15-2 is to place GLJPanel inside a JPanel, which renders a “background” image such as a gradient fill or picture. For the background to be visible, GLJPanel's own background must be made transparent. I'll explain how to do this for the rotating cube application.

Rotating a GLCanvas Cube with Callbacks

The GLCanvas and callback technique outlined in the last section is used in the CubeGL application to rotate a colored cube around the x-, y-, and z- axes. Figure 15-3 shows a screenshot of the cube in action.

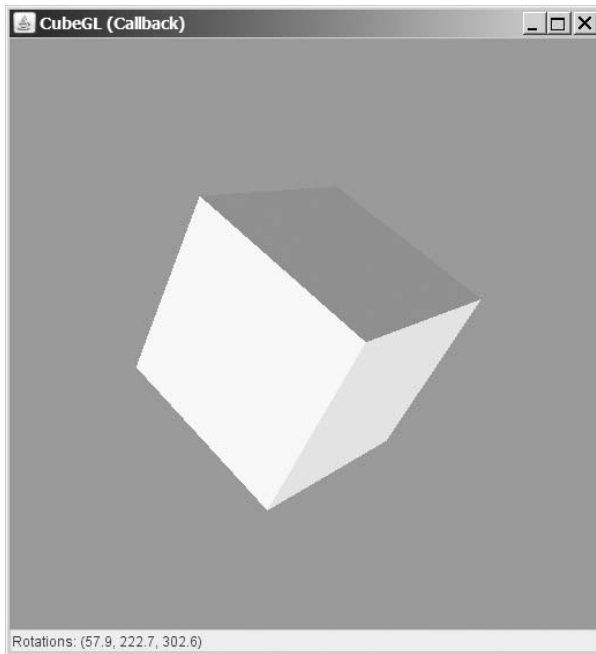


Figure 15-3. *CubeGL with GLCanvas and callbacks*

The window consists of a JPanel in the center holding the GLCanvas and a text field at the bottom that reports the current x-, y-, and z-axis rotations of the cube.

Class diagrams for the application are given in Figure 15-4, showing only public methods.

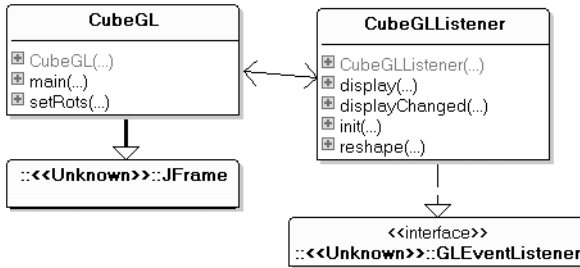


Figure 15-4. Class diagrams for *CubeGL* with *GLCanvas* and callbacks

CubeGL is the top-level *JFrame* that creates the *GLCanvas* and *FPSAnimator* objects. *CubeGLListener* is the canvas' listener, a subclass of *GLEventListener* that implements the callbacks *init()*, *reshape()*, and *display()*. (*displayChanged()* is empty since JOGL doesn't support it.)

CubeGL is an example of the *GLCanvas* callback coding style illustrated by Figure 15-1.

Building the Top-Level Window

CubeGL's constructor builds the GUI and sets up a window listener for responding to a window closing event:

```
// globals
private FPSAnimator animator;
private JTextField rotsTF; // displays cube rotations

public CubeGL(int fps)
{
    super("CubeGL (Callback)");

    Container c = getContentPane();
    c.setLayout( new BorderLayout() );
    c.add( makeRenderPanel(fps), BorderLayout.CENTER);

    rotsTF = new JTextField("Rotations: ");
    rotsTF.setEditable(false);
    c.add(rotsTF, BorderLayout.SOUTH);

    addWindowListener( new WindowAdapter() {
        public void windowClosing(WindowEvent e)
        { new Thread( new Runnable() {
            public void run() {
                animator.stop();
                System.exit(0);
            }
        }).start();
    } // end of windowClosing()
    });

    pack();
    setVisible(true);
}
```

```
    animator.start();
} // end of CubeGL()
```

The frame/second (FPS) input argument comes from the command line, or a default value of 80 is used. The aim is to update the rotating cube at the specified rate.

The windowClosing() terminates the FPSAnimator object (*animator*) and makes the application exit. The code is carried out in its own thread instead of the one associated with the window listener to ensure that the animator stops before System.exit() is called.

Connecting the Canvas

The GLCanvas is embedded inside a JPanel by makeRenderPanel() and connected to its animator and listener:

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

private CubeGLListener listener;

private JPanel makeRenderPanel(int fps)
{
    JPanel renderPane = new JPanel();
    renderPane.setLayout( new BorderLayout() );
    renderPane.setOpaque(false);
    renderPane.setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

    GLCanvas canvas = new GLCanvas();      // the canvas
    listener = new CubeGLListener(this, fps); // the listener
    canvas.addGLEventListener(listener);

    animator = new FPSAnimator(canvas, fps, true);
                // the animator uses fixed rate scheduling

    renderPane.add(canvas, BorderLayout.CENTER);
    return renderPane;
} // end of makeRenderPanel()
```

The canvas' enclosing JPanel is given an initial size (512 by 512 pixels), but the window can be resized later, affecting the canvas.

The FPSAnimator constructor takes a reference to the GLAutoDrawable instance (i.e., the *canvas*). Its display() method will be called with a frequency set by the fps argument. FPSAnimator's third argument (set to true) indicates that *fixed-rate scheduling* will be used. Each task is scheduled relative to the scheduled execution time of the initial task. If a task is delayed for any reason (such as garbage collection), two or more tasks will occur in rapid succession to catch up.

Building the Listener

The rotating colored cube is implemented with OpenGL function calls inside the GLEventListener callback methods init(), reshape(), and display().

The listener also includes statistics-gathering code to report how well the application meets the requested frame rate.

The CubeGLListener constructor creates various statistics data structures. It then waits for the canvas to be displayed, which triggers a call to init().

Initializing OpenGL

The OpenGL initialization code in `init()` typically includes the setup of the z- (depth) buffer, the creation of lights, texture loading, and display-list building. This example doesn't use lights or textures:

```
// globals
private static final float INCR_MAX = 10.0f;    // rotation increments

private GLU glu;

private int cubeDList;    // display list for displaying the cube

// rotation variables
private float rotX, rotY, rotZ;    // total rotations in x,y,z axes
private float incrX, incrY, incrZ;    // increments for x,y,z rotations

public void init(GLAutoDrawable drawable)
{
    GL gl = drawable.getGL();    // don't make this gl a global!
    glu = new GLU();    /* this is okay as a global, but
                           only use it in callbacks */

    // gl.setSwapInterval(0);
    // switch off vertical synchronization, for extra speed (maybe)

    // initialize the rotation variables
    rotX = 0; rotY = 0; rotZ = 0;
    Random random = new Random();
    incrX = random.nextFloat()*INCR_MAX;    // 0 - INCR_MAX degrees
    incrY = random.nextFloat()*INCR_MAX;
    incrZ = random.nextFloat()*INCR_MAX;

    gl.glClearColor(0.17f, 0.65f, 0.92f, 0.0f); //sky color background

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

    // create a display list for drawing the cube
    cubeDList = gl.glGenLists(1);
    gl.glNewList(cubeDList, GL.GL_COMPILE);
    drawColourCube(gl);
    gl.glEndList();
} // end of init()
```

`init()`'s `GLAutoDrawable` input argument is the programmer's entry point into OpenGL. The `GLAutoDrawable.getGL()` call returns a `GL` object that can be employed to call OpenGL routines.

The JOGL documentation advises against making the `GL` instance global, since it might tempt programmers into calling OpenGL functions from mouse and keyboard listeners or other threads. This would almost certainly cause the application to crash, since the OpenGL context (its internal state) is tied to the `GLEventListener`. However, it is OK to make the `GLU` instance a global, but it should only be utilized in the callback methods.

The `GL.setSwapInterval()` call switches off vertical synchronization, which may increase the frame rate, depending on the display card and its settings. It makes no discernable difference on my three test machines, so is commented out here.

The cube's current x-, y-, and z- rotations are stored in the globals rotX, rotY, and rotZ. The rotation increments are randomly generated but have values somewhere between 0 and 10 degrees.

An OpenGL display list acts as a storage space for OpenGL rendering and state commands. The commands are compiled into an optimized form, which allows them to be executed more quickly. The benefit of a display list is that it can be called multiple times without OpenGL having to recompile the commands, thereby saving processing time. The cubeDList display list created in init() groups the commands that draw the cube.

Drawing the Colored Cube

The colored cube is made from six differently colored squares—an unchanging rendering task that's a good choice for a display list.

Figure 15-5 shows the cube's vertices, which are positioned so the box is centered on the origin, and has sides of length 2. Each vertex is assigned a number, which is used in the code that follows.

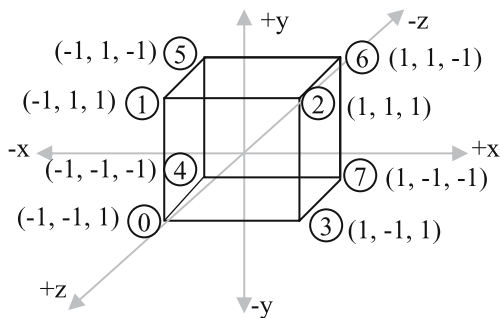


Figure 15-5. The colored cube's numbered vertices

The vertices are stored in a global array:

```
private float[][] verts = {
    {-1.0f, -1.0f, 1.0f}, // vertex 0
    {-1.0f, 1.0f, 1.0f}, // 1
    { 1.0f, 1.0f, 1.0f}, // 2
    { 1.0f, -1.0f, 1.0f}, // 3
    {-1.0f, -1.0f, -1.0f}, // 4
    {-1.0f, 1.0f, -1.0f}, // 5
    { 1.0f, 1.0f, -1.0f}, // 6
    { 1.0f, -1.0f, -1.0f}, // 7
};
```

The array positions of the vertices are used by drawPolygon() to draw a cube face. drawPolygon() is called six times from drawColourCube():

```
private void drawColourCube(GL gl)
// six-sided cube, with a different color on each face
{
    gl.glColor3f(1.0f, 0.0f, 0.0f); // red
    drawPolygon(gl, 0, 3, 2, 1); // front face

    gl.glColor3f(0.0f, 1.0f, 0.0f); // green
    drawPolygon(gl, 2, 3, 7, 6); // right
```

```

gl.glColor3f(0.0f, 0.0f, 1.0f);    // blue
drawPolygon(gl, 3, 0, 4, 7);       // bottom

gl.glColor3f(1.0f, 1.0f, 0.0f);    // yellow
drawPolygon(gl, 1, 2, 6, 5);       // top

gl.glColor3f(0.0f, 1.0f, 1.0f);    // light blue
drawPolygon(gl, 4, 5, 6, 7);       // back

gl.glColor3f(1.0f, 0.0f, 1.0f);    // purple
drawPolygon(gl, 5, 4, 0, 1);       // left
} // end of drawColourCube()

private void drawPolygon(GL gl, int vIdx0, int vIdx1,
                        int vIdx2, int vIdx3)
// the polygon vertices come from the verts[] array
{
    gl.glBegin(GL.GL_POLYGON);
    gl.glVertex3f(verts[vIdx0][0],verts[vIdx0][1], verts[vIdx0][2] );
    gl.glVertex3f(verts[vIdx1][0],verts[vIdx1][1], verts[vIdx1][2] );
    gl.glVertex3f(verts[vIdx2][0],verts[vIdx2][1], verts[vIdx2][2] );
    gl.glVertex3f(verts[vIdx3][0],verts[vIdx3][1], verts[vIdx3][2] );
    gl.glEnd();
} // end of drawPolygon()

```

`GL.glBegin()` and `GL.glEnd()` bracket a sequence of vertex definitions, and `glBegin()`'s argument specifies the vertices' collective shape. Other modes include `GL.GL_POINTS` (a collection of points) and `GL.GL_LINES` (a set of lines).

Reshaping the Canvas

`reshape()` is called when the canvas is moved or resized, which includes when it's first drawn onscreen. That makes `reshape()` the natural place to hold OpenGL commands for setting the viewport and projection matrix:

```

public void reshape(GLAutoDrawable drawable, int x, int y,
                  int width, int height)
{
    GL gl = drawable.getGL();

    if (height == 0)
        height = 1;    // to avoid division by 0 in aspect ratio below

    gl.glViewport(x, y, width, height); // size of drawing area

    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    glu.gluPerspective(45.0, (float)width/(float)height, 1, 100);
        // FOV, aspect ratio, near & far clipping planes

    gl.glMatrixMode(GL.GL_MODELVIEW);
    gl.glLoadIdentity();
} // end of reshape()

```

`reshape()`'s (x, y) input arguments specify the canvas' position relative to its enclosing container. In this example, they're always (0, 0).

A GL object is freshly created from the GLAutoDrawable input argument.

The GL.glViewport() call defines the size of 3D drawing window (viewport) in terms of a lower-left corner (x, y), width, and height.

The matrix mode is switched to PROJECTION (OpenGL's projection matrix) so the mapping from the 3D scene to the 2D screen can be specified. GL.glLoadIdentity() resets the matrix, and GLU.gluPerspective() creates a mapping with perspective effects (which mirrors what happens in a real-world camera). FOV is the camera's angle of view.

The matrix mode is switched to MODELVIEW at the end of reshape() so OpenGL's model-view matrix can be utilized from then on. It defines the scene's coordinate system, used when positioning, or moving, 3D objects. It's set up at the end of reshape() since display(), which draws the scene, will be called next.

Scene Rendering

As FPSAnimator ticks, it calls display() in the canvas, triggering a call to display() in CubeGLListener. Its display() method holds code that updates and redraws the scene:

```
// global
private static final double Z_DIST = 7.0;    // for camera position

public void display(GLAutoDrawable drawable)
{
    // update the rotations
    rotX = (rotX + incrX) % 360.0f;
    rotY = (rotY + incrY) % 360.0f;
    rotZ = (rotZ + incrZ) % 360.0f;
    top.setRots(rotX, rotY, rotZ); // report at top-level

    GL gl = drawable.getGL();

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

    glu.gluLookAt(0,0,Z_DIST, 0,0,0, 0,1,0); // position camera

    // apply rotations to the x,y,z axes
    gl.glRotatef(rotX, 1.0f, 0.0f, 0.0f);
    gl.glRotatef(rotY, 0.0f, 1.0f, 0.0f);
    gl.glRotatef(rotZ, 0.0f, 0.0f, 1.0f);
    gl.glCallList(cubeDList); //execute display list for drawing cube
    // drawColourCube(gl);

    reportStats();
} // end of display
```

The cube's x-, y-, and z- rotations in rotX, rotY, and rotZ are updated. The new values are reported onscreen by writing them to the text field in the top-level JFrame (see Figure 15-3).

After the new rotations have been applied to the world coordinates, the cube is drawn via its display list. Alternatively, display() could call drawColourCube() directly.

Measuring FPS Accuracy

reportStats() is called at the end of display(). It prints an average frame rate value roughly every second, as shown in the following CubeGL execution:

```
> runGL CubeGL
Executing CubeGL with JOGL...
fps: 80
period: 12 ms
54.4
64.18
68.42
70.93
72.63
73.84
74.78
75.53
76.13
Finished.
```

The average is calculated from the previous ten FPS values (or less, if ten numbers haven't been calculated yet). This weighted approach discounts earlier slow frame rate data.

In the previous example, CubeGL is started with a requested frame rate of 80, which is converted into a millisecond time period using integer division:

```
int period = (int) 1000.0/fps;    // in ms
```

This is later converted back to a frame rate of $1000/12$, which is 83.333. This means an optimally running application should report an average frame rate of around 83 FPS. The example is slowly approaching that and reaches 83 after about 30 seconds.

The implementation of reportStats() doesn't have anything to do with JOGL or OpenGL, so I'll skip its explanation.

Table 15-1 shows the reported average FPS on different versions of Windows when the requested FPS are 20, 50, 80, and 100. Windows XP appears twice since I ran the tests on two different machines using XP.

Table 15-1. *Average FPS for GLCanvas CubeGL with FPSAnimator (Fixed Rate Scheduling)*

Requested FPS	20	50	80	100
Windows 2000	20	50	79	80
Windows XP (1)	20	50	83	100
Windows XP (2)	20	50	81	99

Each test was run three times on a lightly loaded machine running for a few minutes.

The average frame rates are excellent for 80 FPS, although the average hides the fact that it takes a minute or so for the frame rate to rise toward the average. Also, JVM garbage collection reduces the FPS for a few seconds every time it occurs.

The Windows 2000 machine is not capable of achieving 100 FPS, due to its slow hardware.

The FPSAnimator constructor can also be instructed to use a fixed *period* scheduler rather than fixed-rate scheduling. This only requires the change of the boolean argument in FPSAnimator's constructor in makeRenderPanel():

```
// makeRenderPanel() in CubeGL
animator = new FPSAnimator(canvas, fps, false);
// the animator uses fixed period scheduling
```

The timing tests were run again on the same machines under the same load conditions. The results are shown in Table 15-2.

Table 15-2. Average FPS for GLCanvas CubeGL and FPSAnimator (Fixed Period Scheduling)

Requested FPS	20	50	80	100
Windows 2000	19	49	49	98
Windows XP (1)	16	32	63	62
Windows XP (2)	16	31	62	62

The results show a wide variation in FPS accuracy, but the results for the 80 FPS request (the refresh rate on my test machines) are quite poor.

The fixed period scheduler in FPSAnimator uses `java.util.Timer.schedule()` to repeatedly trigger actions. Unfortunately, the timer's frequency can drift because of extra delays introduced by the garbage collector or long-running game updates and rendering.

Best results are obtained by using FPSAnimator's fixed rate scheduler, as Table 15-1 shows.

Rotating a GLJPanel Cube with Callbacks

An alternative to GLCanvas is GLJPanel, a lightweight widget. Its interface is almost the same as GLCanvas, so the two components can be interchanged easily. This is illustrated by the callback frameworks for GLCanvas and GLJPanel in Figures 15-1 and 15-2.

GLJPanel has historically been much slower than GLCanvas, but its speed has significantly improved in J2SE 5 and Java SE 6. Its key advantage over GLCanvas is that it allows Java 2D and JOGL to be combined in new ways.

Figure 15-6 shows one such combination, a GLJPanel with a background supplied by an enclosing JPanel.

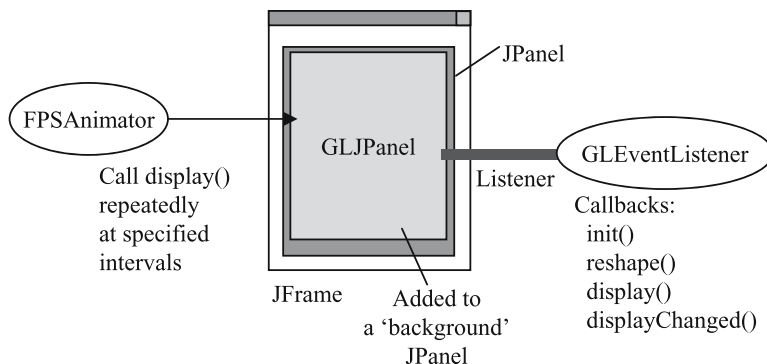


Figure 15-6. A callback application with GLJPanel and JPanel background

The callback framework is the same as in Figure 15-2; only the background JPanel is new.

Figure 15-7 shows the rotating cube example again, implemented using the Figure 15-6 approach. The 3D parts are rendered in a GLJPanel with a transparent background. The gradient fill and “Hello World” text are drawn by Java 2D in the JPanel enclosing the GLJPanel.



Figure 15-7. Rotating cube inside a GLJPanel and JPanel background

The required code changes to convert from a GLCanvas to the GLJPanel are quite small, as I outline in the next three subsections.

A very important command-line change is to include `-Dsun.java2d.opengl=true` to switch on the Java2D OpenGL pipeline and so increase Java 2D's rendering speed. The application will still run without `-Dsun.java2d.opengl=true` but much slower. My `runGL.bat` batch file becomes the following:

```
@echo off
echo Executing %1 with JOGL...
java -cp "d:\jogl\jogl.jar;d:\jogl\gluegen-rt.jar;."
      -Djava.library.path="d:\jogl"
      -Dsun.java2d.noddraw=true
      -Dsun.java2d.opengl=true %1 %2
echo Finished.
```

There have been reports of problems with `-Dsun.java2d.opengl=true` at the JOGL forum at javagaming.org. For example, it appears to affect rendering speeds when the window is maximized and sometimes crashes the application. On my oldest test machine, which uses a Radeon 9000 PRO AGP graphics chip, the `-Dsun.java2d.opengl=true` argument causes the gradient fill to disappear, and the application crashes when the window is closed.

The most common response from forum readers is to suggest that people update their graphics cards and drivers. This lack of backward-compatibility is an important concern when using GLJPanel in games aimed at older machines.

Building the Panels

`makeRenderPanel()` in `CubeGL` now creates a `GLJPanel` object rather than a `GLCanvas` instance and embeds it inside a background panel:

```
private JPanel makeRenderPanel(int fps)
{
    // JPanel renderPane = new JPanel();
    JPanel renderPane = createBackPanel();    // for the GLJPanel

    renderPane.setLayout( new BorderLayout() );
    renderPane.setOpaque(false);
    renderPane.setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

    // GLCanvas canvas = new GLCanvas();
    // create the GLJPanel
    GLCapabilities caps = new GLCapabilities();
    caps.setAlphaBits(8);
    GLJPanel canvas = new GLJPanel(caps);
    canvas.setOpaque(false);

    listener = new CubeGLListener(this, fps);
    canvas.addGLEventListener(listener);

    animator = new FPSAnimator(canvas, fps, true);

    renderPane.add(canvas, BorderLayout.CENTER);
    return renderPane;
} // end of makeRenderPanel()
```

The old code for creating the `GLCanvas` object and its `JPanel` have been commented out.

A transparent `GLJPanel` requires a nonzero alpha depth, set using a `GLCapabilities` object and a call to `GLJPanel.setOpaque()`.

The Background Panel

The `JPanel` acting as the background draws a gradient fill and text:

```
// global
private Font font;

private JPanel createBackPanel()
{
    font = new Font("SansSerif", Font.BOLD, 48);

    JPanel p = new JPanel() {
        public void paintComponent(Graphics g)
        {
            Graphics2D g2d = (Graphics2D) g;
            int width = getWidth();
            int height = getHeight();
            g2d.setPaint( new GradientPaint(0, 0, Color.YELLOW,
                                           width, height, Color.BLUE));
            g2d.fillRect(0, 0, width, height);

            g2d.setPaint(Color.BLACK);
            g2d.setFont(font);
            g2d.drawString("Hello World", width/4, height/4);
        }
    };
    return p;
}
```

```

    } // end of paintComponent()
};
return p;
} // end of createBackPanel()

```

The gradient fill and text change position when the application is resized since they utilize the panel's current width and height values.

Making the 3D Background Transparent

The OpenGL background drawn into the GLJPanel must be transparent (or at least translucent) so the background JPanel's gradient fill and text will be visible.

The rotating cube's background (a light-blue color) is set up inside `init()` inside `CubeGLListener`. It is changed to be transparent (or translucent).

The effect shown in Figure 15-7 is achieved with the following:

```
gl.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);    // no OpenGL background
```

The important argument is the fourth, which sets the alpha value for the RGB color preceding it. *0.0f* means fully transparent; *1.0f* is opaque. The *0.0f* value in the example means that all the background color comes from the background panel.

A translucent effect (a mix of the background panel and OpenGL's background colors) is obtained with the following:

```
gl.glClearColor(0.17f, 0.65f, 0.92f, 0.3f);
                        // translucent OpenGL sky
```

The *0.3f* alpha value makes the OpenGL sky translucent.

The result is shown in Figure 15-8.



Figure 15-8. Rotating cube inside a GLJPanel with a bluish JPanel background

The effect is hard to see (especially when rendered in shades of gray), but the yellow parts of the JPanel's gradient fill have turned green due to the blue OpenGL background.

Timing the GLJPanel

Timing tests were run using the same Windows XP machines under the same load conditions as the GLCanvas callback code with fixed rate scheduling. The results are shown in Table 15-3.

Table 15-3. Average FPS for GLJPanel CubeGL and FPSAnimator (Fixed Rate Scheduling)

Requested FPS	20	50	80	100
Windows XP (1)	20	50	71	87
Windows XP (2)	20	50	75	90

The results are very good but slower at higher frame rates than the GLCanvas code.

The speeds are substantially less when the OpenGL pipeline is not enabled (i.e., when `Dsun.java2d.opengl=true` isn't part of the command line). For instance, the application only manages about 25 FPS when 80 FPS are requested.

No results are shown for my antiquated Windows 2000 machine since the background rendering didn't work with its old ATI graphics card; the background was always drawn in black.

More Visual Effects with GLJPanel

Chris Campbell's blog entry, "Easy 2D/3D Mixing in Swing" (http://weblogs.java.net/blog/campbell/archive/2006/10/easy_2d3d_mixin.html), is a good starting point for more examples of how to integrate 2D and 3D effects in a GUI.

His PhotoCube application includes a `CompositeGLJPanel` class that offers methods for common types of 2D/3D mixing (e.g., `render2DBackground()`, `render3DScene()`, and `render2DForeground()`). There are also pointers to other articles and online code.

Callback Summary

The callback technique (for GLCanvas and GLJPanel) delivers great frame rates, as long as fixed rate scheduling is utilized and the hardware is fast enough. GLJPanel's successful operation is particularly sensitive to the underlying hardware and graphics driver.

An important advantage of the JOGL callback coding style is its similarity to the callback mechanism used in OpenGL's GLUT. This allows numerous OpenGL examples to be ported over to JOGL with minimal changes.

One drawback of the callback approach is the way that the application life cycle (initialization, resizing, frame-based animation, and termination) is divided across multiple disjoint methods. Also, the use of a timer (inside the animator class) makes it difficult to vary the application's timing behavior at runtime and to separate the frame rate (FPS) from the application's update rate (UPS). The active rendering framework described in the next section addresses these concerns.

The Active Rendering Framework

The active rendering framework utilizes the new features in JSR-231 for directly accessing the drawing surface and context (OpenGL's internal state). This means that there's no longer any need to utilize GUI components that implement the `GLAutoDrawable` interface, such as GLCanvas. An

application can employ a subclass of AWT's Canvas, with its own rendering thread, as illustrated by Figure 15-9.

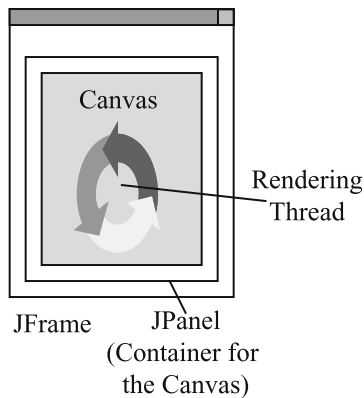


Figure 15-9. *An active rendering application*

The rendering thread can be summarized using the following pseudocode:

```
make the context current for this thread;
initialize rendering;
while game isRunning {
    update game state;
    render scene;
    put the scene onto the canvas;

    sleep a while;
    maybe do game updates without rendering them;
    gather statistics;
}
discard the rendering context;
print the statistics;
exit;
```

The tricky aspect of this code is remembering that OpenGL should be manipulated from the rendering thread only. Any mouse, key, or window events must be processed there, rather than in separate listeners.

The OpenGL callback code, located inside `GLEventListener`'s `init()`, `reshape()`, and `display()` methods, can be moved without many changes into the active rendering thread. The `init()` code is carried out in the “initialize rendering” stage, while `reshape()` and `display()` are handled inside “render scene.”

The principal advantage of the active rendering approach is that it allows the programmer to more directly control the application's execution. For example, it's straightforward to add code that suspends updates when the application is iconified or deactivated (i.e., when it's not the topmost window). Also, access to the timing code inside the animation loop permits a separation of frame rate processing from application updates. I'll illustrate these points by implementing the rotating cube application once again.

Rotating a Cube with Active Rendering

The active rendering CubeGL looks the same as the GLCanvas callback version, as shown in Figure 15-10.

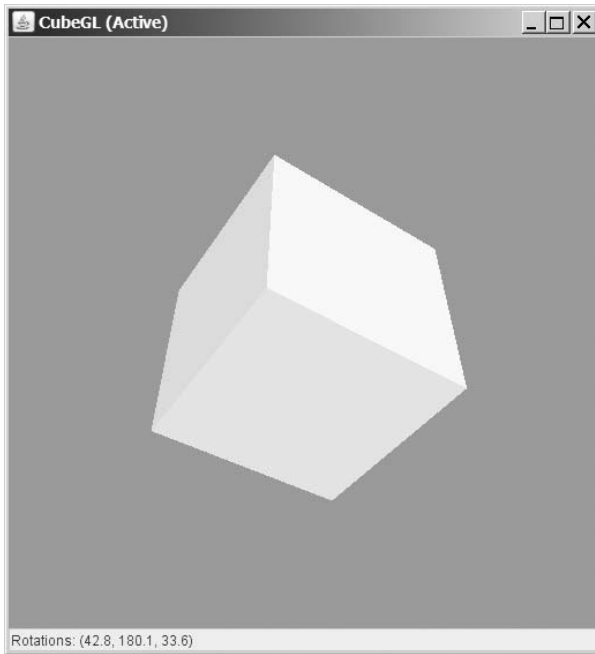


Figure 15-10. *CubeGL with active rendering*

The application has new functionality, courtesy of active rendering: when the window is iconified or deactivated, the cube stops rotating until the window is deiconified or activated again.

The class diagrams for this version of CubeGL are given in Figure 15-11.

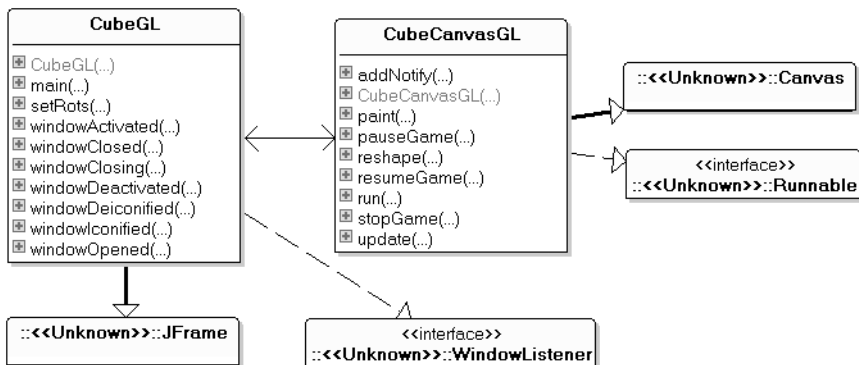


Figure 15-11. *Class diagrams for CubeGL with active rendering*

CubeGL creates the GUI, embedding the threaded canvas, `CubeCanvasGL`, inside a `JPanel`. It also captures window events and component resizes and calls methods in `CubeCanvasGL` to deal with them.

Building the Application

CubeGL creates the threaded canvas inside `makeRenderPanel()`:

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

private CubeCanvasGL canvas;

private JPanel makeRenderPanel(long period)
// construct the canvas inside a JPanel
{
    JPanel renderPane = new JPanel();
    renderPane.setLayout( new BorderLayout() );
    renderPane.setOpaque(false);
    renderPane.setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

    canvas = makeCanvas(period);
    renderPane.add(canvas, BorderLayout.CENTER);

    canvas.setFocusable(true);
    canvas.requestFocus();    //canvas has focus, so receives key events

    // detect window resizes, and reshape the canvas accordingly
    renderPane.addComponentListener( new ComponentAdapter() {
        public void componentResized(ComponentEvent evt)
        { Dimension d = evt.getComponent().getSize();
          canvas.reshape(d.width, d.height);
        }
    });

    return renderPane;
} // end of makeRenderPanel()
```

The panel has two roles: it surrounds the canvas, protecting lightweight GUI widgets from the heavyweight AWT Canvas and is a convenient place to connect a component listener to detect window resizes. A resize generates a call to `CubeCanvasGL.reshape()`, which triggers a recalculation of the OpenGL viewport and perspective.

The period input to `makeRenderPanel()` comes from the frame rate supplied on the command line. It's calculated as the following:

```
long period = (long) 1000.0/fps;
```

`makeCanvas()` obtains an optimal graphics configuration for the canvas. It passes this information to an instance of the threaded canvas, `CubeCanvasGL`:

```
private CubeCanvasGL makeCanvas(long period)
{
    // get a configuration suitable for an AWT Canvas
    GLCapabilities caps = new GLCapabilities();
```

```

AWTGraphicsDevice dev = new AWTGraphicsDevice(null);
AWTGraphicsConfiguration awtConfig =
    (AWTGraphicsConfiguration)GLDrawableFactory.getFactory().
        chooseGraphicsConfiguration(caps, null, dev);

GraphicsConfiguration config = null;
if (awtConfig != null)
    config = awtConfig.getGraphicsConfiguration();

return new CubeCanvasGL(this, period, PWIDTH, PHEIGHT,
                        config, caps);
} // end of makeCanvas()

```

Dealing with Window Events

CubeGL is a window listener:

```

public void windowActivated(WindowEvent e)
{ canvas.resumeGame(); }

public void windowDeactivated(WindowEvent e)
{ canvas.pauseGame(); }

public void windowDeiconified(WindowEvent e)
{ canvas.resumeGame(); }

public void windowIconified(WindowEvent e)
{ canvas.pauseGame(); }

public void windowClosing(WindowEvent e)
{ canvas.stopGame(); }

public void windowClosed(WindowEvent e) {}
public void windowOpened(WindowEvent e) {}

```

pauseGame(), resumeGame(), and stopGame() trigger extra processing inside CubeCanvasGL's rendering loop to pause, resume, or terminate the application.

Preparing the Canvas

Before the rendering thread can start inside CubeCanvasGL, the rendering surface and context for the canvas need to be accessed. This is done in CubeCanvasGL's constructor:

```

// globals
private GLDrawable drawable; // the rendering 'surface'
private GLContext context;
    // the rendering context (holds rendering state info)

```

```

// in the CubeCanvasGL constructor:
drawable =
    GLDrawableFactory.getFactory().getGLDrawable(this, caps, null);
context = drawable.createContext(null);

```

The GLCapabilities instance, caps, comes from CubeGL's makeCanvas(), which creates the canvas object.

Rendering should be delayed until the canvas is visible onscreen, which occurs once the canvas calls its `addNotify()` method. This behavior can be implemented by starting the thread from `addNotify()` in `CubeCanvasGL`:

```
// global
private Thread animator;    // thread that performs the animation

public void addNotify()
// wait for the canvas to be added to the JPanel before starting
{
    super.addNotify();        // make the component displayable
    drawable.setRealized(true); // canvas can now be used for rendering

    // initialize and start the animation thread
    if (animator == null || !isRunning) {
        animator = new Thread(this);
        animator.start();
    }
} // end of addNotify()
```

Thread Rendering

The `run()` method in `CubeCanvasGL` follows the pseudocode given earlier. This is the first version; I describe a slightly modified version later in this section:

```
public void run()
// initialize rendering and start frame generation; first version
{
    makeContentCurrent();

    initRender();
    renderLoop();

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

```
private void makeContentCurrent()
// make the rendering context current for this thread
{
    try {
        while (context.makeCurrent() == GLContext.CONTEXT_NOT_CURRENT) {
            System.out.println("Context not yet current...");
            Thread.sleep(100);
        }
    }
    catch (InterruptedException e)
    { e.printStackTrace(); }
} // end of makeContentCurrent()
```

`makeContentCurrent()` calls `GLContext.makeCurrent()`, which should immediately succeed since no other thread is using the context. The while-loop around the `GLContext.makeCurrent()` call

is extra protection since the application will crash if OpenGL commands are called without the thread holding the current context.

When execution returns from the rendering loop inside `renderLoop()`, the context is released and destroyed and the application exits.

This coding approach means that the context is current for the entire duration of the thread's execution. This causes no problems on most platforms (e.g., it's fine on Windows), but unfortunately there's an issue when using X11. On X11 platforms, a AWT lock is created between the `GLContext.makeCurrent()` and `GLContext.release()` calls, stopping mouse and keyboard input from being processed.

The only solution is to periodically release the context, giving the JRE under X11 time to act on mouse and keyboard events.

This means that `run()` must have its calls to `makeCurrentContext()` and `GLContext.release()` commented out. This leads to a second version of the code:

```
public void run()
// initialize rendering and start frame generation; 2nd version
{
    // makeContentCurrent(); // commented out due to X11

    initRender();
    renderLoop();

    // discard the rendering context and exit
    // context.release(); // commented out due to X11
    context.destroy();
    System.exit(0);
} // end of run()
```

Instead, the context will be made current and released inside `initRender()` and `renderLoop()`.

Rendering Initialization

The `initRender()` method in `CubeCanvasGL` corresponds to the `init()` callback in `GLEventListener` with one important OpenGL-related change:

```
// globals
private GL gl;
private GLU glu;

private void initRender()
{
    makeContentCurrent();

    gl = context.getGL(); // gl is now global
    glu = new GLU();

    resizeView();

    gl.glClearColor(0.17f, 0.65f, 0.92f, 0.0f); // sky color backgrnd

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

    // create a display list for drawing the cube
    cubeDList = gl.glGenLists(1);
```

```

gl.glNewList(cubeDList, GL.GL_COMPILE);
    drawColourCube(gl);
gl.glEndList();

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

```

The recommended coding style in `GLEventListener` callbacks, such as `init()`, is to obtain a fresh GL reference inside each method via the `GLAutoDrawable` input argument. This is unnecessary in the active rendering approach since there's only a single thread executing inside the canvas. Therefore, the GL instance is made global and initialized once at the start of `initRender()`.

The initialization of the rotation variables has been moved to `CubeCanvasGL`'s constructor so only OpenGL code is left in `initRender()`.

The color cube drawing code in `drawColourCube()` (and its helper method `drawPolygon()`) are unchanged from the callback version of `CubeGL`, so I'll skip them here.

`resizeView()` sets the viewpoint and perspective and corresponds to the initial call to the `reshape()` callback in `GLEventListener`:

```

// globals
private int panelWidth, panelHeight;

private void resizeView()
{
    gl.glViewport(0, 0, panelWidth, panelHeight); // drawing area

    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    glu.gluPerspective(45.0, (float)panelWidth/(float)panelHeight, 1, 100);
    // fov, aspect ratio, near & far clipping planes
} // end of resizeView()

```

`panelWidth` and `panelHeight` are assigned their initial values in `CubeCanvasGL`'s constructor.

I explain how `resizeView()` is called in the “Rendering the Scene” section when I describe how the application's window is resized.

The Rendering Loop

`renderLoop()` implements the while-loop in the active rendering pseudocode:

```

while game isRunning {
    update game state;
    render scene;
    put the scene onto the canvas;

    sleep a while;
    maybe do game updates without rendering them;

    gather statistics;
}

```

The loop is complicated by having to calculate the amount of time it takes to do the update-render pair. The sleep time that follows must be adjusted so the time to complete the iteration is as close to the desired frame rate as possible.

If an update-render takes too long, it may be necessary to carry out some game updates without rendering their changes. The result is a game that runs close to the requested frame rate by skipping the time-consuming rendering of the updates.

The timing code distinguishes between two rates: the actual frame rate that measures the number of renders/second (FPS), and the update rate that measures the number of updates/second (UPS).

FPS and UPS aren't the same. It's quite possible for a slow platform to limit the FPS value, but the program performs additional updates (without rendering) so that its UPS number is close to the requested frame rate.

This separation of FPS and UPS makes the animation loop more complicated, but it's one of the standard ways to create reliable animations. It's especially good for games where the hardware is unable to render at the requested frame rate.

The following is the code for `renderLoop()`:

```
// constants
private static final int NO_DELAYS_PER_YIELD = 16;
/* Number of iterations with a sleep delay of 0 ms before the
   animation thread yields to other running threads. */

private static int MAX_RENDER_SKIPS = 5;
/* no. of renders that can be skipped in any one animation loop;
   i.e. the games state is updated but not rendered. */

// globals
private long prevStatsTime;
private long gameStartTime;
private long rendersSkipped = 0L;

private long period;          // period between drawing in nanosecs
private volatile boolean isRunning = false;
                             // used to stop the animation thread

private void renderLoop()
{
    // timing-related variables
    long beforeTime, afterTime, timeDiff, sleepTime;
    long overSleepTime = 0L;
    int noDelays = 0;
    long excess = 0L;

    gameStartTime = System.nanoTime();
    prevStatsTime = gameStartTime;
    beforeTime = gameStartTime;

    isRunning = true;

    while(isRunning) {
        makeContentCurrent();

        gameUpdate();
        renderScene(); // rendering
        drawable.swapBuffers(); // put the scene onto the canvas
        // swap front and back buffers, making the rendering visible

        afterTime = System.nanoTime();
```

```

timeDiff = afterTime - beforeTime;
sleepTime = (period - timeDiff) - overSleepTime;

if (sleepTime > 0) { // some time left in this cycle
    try {
        Thread.sleep(sleepTime/1000000L); // nano -> ms
    }
    catch(InterruptedException ex){}
    overSleepTime = (System.nanoTime() - afterTime) - sleepTime;
}
else { // sleepTime <= 0; this cycle took longer than the period
    excess -= sleepTime; // store excess time value
    overSleepTime = 0L;

    if (++noDelays >= NO_DELAYS_PER_YIELD) {
        Thread.yield(); // give another thread a chance to run
        noDelays = 0;
    }
}

beforeTime = System.nanoTime();

/* If the rendering is taking too long,
   then update the game state without rendering it, to
   get the UPS nearer to the required frame rate. */
int skips = 0;
while((excess > period) && (skips < MAX_RENDER_SKIPS)) {
    excess -= period;
    gameUpdate(); // update state but don't render
    skips++;
}
rendersSkipped += skips;

/* release the context, otherwise the AWT lock in X11
   will not be released */
context.release();

storeStats();
}

printStats();
} // end of renderLoop()

```

The “sleep a while” code in the loop is complicated by dealing with inaccuracies in `Thread.sleep()`. `sleep()`’s execution time is measured and the error (stored in `overSleepTime`) adjusts the sleeping period in the next iteration.

The if-test involves `Thread.yield()`:

```

if (++noDelays >= NO_DELAYS_PER_YIELD) {
    Thread.yield();
    noDelays = 0;
}

```

It ensures that other threads get a chance to execute if the animation loop hasn’t slept for a while.

`renderLoop` calls `makeContentCurrent()` and `GLContext.release()` at the start and end of each rendering iteration. This allows the JRE under X11 some time to process AWT events.

Updating the Game

gameUpdate() should contain any calculations that affect gameplay, which for CubeGL are only the x-, y-, and z- rotations used by the cube:

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

private CubeGL top; // reference back to the top-level JFrame

// rotation variables
private float rotX, rotY, rotZ; // total rotations in x,y,z axes
private float incrX, incrY, incrZ; // increments for x,y,z rotations

private void gameUpdate()
{ if (!isPaused && !gameOver) {
    // update the rotations
    rotX = (rotX + incrX) % 360.0f;
    rotY = (rotY + incrY) % 360.0f;
    rotZ = (rotZ + incrZ) % 360.0f;
    top.setRots(rotX, rotY, rotZ);
  }
} // end of gameUpdate()
```

The isPaused and gameOver booleans allow the updates to be skipped when the game is paused or has finished.

Rendering the Scene

The scene generation carried out by renderScene() is similar to what display() does in the callback version of CubeGL:

```
// global
private boolean isResized = false; // for window resizing

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

    if (isResized) { // resize the drawable if necessary
        resizeView();
        isResized = false;
    }

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

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

    glu.gluLookAt(0,0,Z_DIST, 0,0,0, 0,1,0); // position camera
```

```

// apply rotations to the x,y,z axes
gl.glRotatef(rotX, 1.0f, 0.0f, 0.0f);
gl.glRotatef(rotY, 0.0f, 1.0f, 0.0f);
gl.glRotatef(rotZ, 0.0f, 0.0f, 1.0f);
gl.glCallList(cubeDList); // execute display list for drawing cube
// drawColourCube(gl);

if (gameOver)
    System.out.println("Game Over"); //report that the game is over
} // end of renderScene()

```

One of the new things that `renderScene()` does is to check that the thread still has the current context; if it doesn't, the application exits. A more robust response would be to try to regain the context by calling `GLContext.makeCurrent()` again, reinitializing the scene, and restarting the animation loop.

`renderScene()` calls `resizeView()` to update the OpenGL view if `isResized` is true. The boolean is set to true by `CubeGL` calling `reshape()` in `CubeCanvasGL` when the window is resized:

```

public void reshape(int w, int h)
/* Called by the JFrame's ComponentListener when the window
   is resized. */
{
    isResized = true;
    if (h == 0)
        h = 1; // to avoid div by 0 in aspect ratio in resizeView()
    panelWidth = w; panelHeight = h;
} // end of reshape()

```

This illustrates the single-threaded coding style needed for OpenGL. `reshape()` does *not* call OpenGL routines itself since it's being executed by a component listener in `CubeGL`. Instead, it sets `isResized` and lets the rendering thread handle the resizing.

`renderScene()` finishes by checking the `gameOver` boolean and printing a simple message. In a real game, the output would be more complicated.

The Game Life Cycle Methods

Window events detected in `CubeGL` are processed by calling `CubeCanvasGL` methods:

```

public void resumeGame()
// called when the JFrame is activated / deiconified
{ isPaused = false; }

public void pauseGame()
// called when the JFrame is deactivated / iconified
{ isPaused = true; }

public void stopGame()
// called when the JFrame is closing
{ isRunning = false; }

```

In the same way as `reshape()`, these methods do not call OpenGL functions since they're being executed by the window listener in `CubeGL`. Instead, they set global booleans checked by the rendering thread.

Statistics Reporting

CubeCanvasGL utilizes two statistics methods: `storeStats()` and `printStats()`. `storeStats()` collects a range of data, and `printStats()` prints a summary just before the application exits. Neither method utilizes JOGL features, so I won't explain their implementation here. Typical output from `printStats()` is shown here:

```
> runGL CubeGL
Executing CubeGL with JOGL...
fps: 80; period: 12 ms
Average FPS: 82.47
Average UPS: 83.28
Time Spent: 33 secs
Finished.
```

The averages are calculated from the last ten recorded FPS and UPS values. If the FPS and UPS numbers are the same, the game was able to match the requested frame rate without skipping the rendering of any updates.

Table 15-4 shows the average FPS and UPS figures for different requested FPS on different versions of Windows.

Table 15-4. *Average FPS/UPS for CubeGL with Active Rendering*

Requested FPS	20	50	80	100
Windows 2000	20/20	43/50	73/83	79/100
Windows XP (1)	20/20	50/50	80/83	95/100
Windows XP (2)	20/20	50/50	81/83	97/100

Each test was run three times on a lightly loaded machine, executing for a few minutes.

The numbers are very good for the machines hosting Windows XP, but the frame rates on the Windows 2000 machine plateau at about 80. This behavior is due to the age of the machine.

The Windows 2000 figures show that active rendering can deal with slow hardware. The processing power of the machine isn't able to deliver the requested frame rate, but the application doesn't seem slow since the UPS stays near to the request FPS. When 80 FPS are requested, about 12% of the updates aren't rendered ((83-73)/83). This isn't apparent when the cube is rotating, which shows the benefit of decoupling updates from rendering.

Java 3D and JOGL

Most of the examples in this book utilize Java 3D, so it's natural to wonder whether Java 3D and JOGL can be used together. The news as of March 2007 was disappointing, but matters may improve in the future.

A posting to the Java Desktop 3D forum in 2004 (<http://forums.java.net/jive/thread.jspa?threadID=5465>) describes the use of JOGL's `GLCanvas` to create a HUD (heads-up display) within a Java 3D application. The canvas was manipulated in the pre- and postrendering phases of Java 3D's immediate mode (or *mixed mode*) to allow JOGL-generated objects to appear in the background and foreground of the scene.

When I tried to duplicate this approach, the objects had a tendency to disappear when the camera position was moved, and sometimes the Java 3D parts of the scene didn't appear.

(For readers unfamiliar with Java 3D's immediate and mixed modes, Chapter 8 explains how to use mixed mode to draw purely Java 3D backgrounds and overlays.)

On a brighter note, Java 3D 1.6 is scheduled for release early in 2008. One of its stated aims is to allow Java 3D and JOGL code to be utilized together. The first steps have already been taken in version 1.5, which offers three versions of Java 3D implemented on top of OpenGL, DirectX, and JOGL.

One of the reasons for using JOGL is its access to shading languages for special effects such as fish eyes, shadow textures, and spherization. However, both GLSL and Cg are already supported in Java 3D.

A user who needs scene graph functionality and OpenGL functions today may want to look at Xith3D (<http://xith.org/>).

More Information on JOGL and OpenGL

The JOGL web site (<https://jogl.dev.java.net/>) hosts the latest software releases together with demos, presentation slides, and a user guide.

The principal source for JOGL help is its forum site at <http://www.javagaming.org/forums/index.php?board=25.0>. A good (but old) JOGL introduction by Gregory Pierce is at <http://www.javagaming.org/forums/index.php?topic=1474.0>. Another introductory article, "Jumping into JOGL" by Chris Adamson in 2003, is at <http://today.java.net/pub/a/today/2003/09/11/jogl2d.html>.

A minor drawback of the forum is that search results will include information for the out-of-date JOGL 1.1. However, it's possible to date-limit the searches to exclude older threads. JSR-231 implementations started appearing in October 2005.

The first stop for information on OpenGL is <http://www.opengl.org/>, which offers a lot of documentation, coding resources, and links to applications, games, and code samples.

The NeHe site (<http://nehe.gamedev.net/>) is an excellent place to start learning OpenGL. It contains an extensive collection of tutorials, articles, examples, and other programming materials. The tutorial, starting from first principles, consists of 48 lessons and has been ported to a variety of languages, including JOGL/JSR-231. The tutorial examples were ported to JOGL by Kevin Duling, Pepijn Van Eeckhoudt, Abdul Bezrati, and Nicholas Cambel. Van Eeckhoudt placed them in a common framework and ported them to JSR-231. The examples are available at http://pepijn.fab4.be/?page_id=34.

There are a growing number of textbooks on OpenGL (e.g., see <http://www.opengl.org/documentation/books/>). For a quick overview that covers the basics without a great deal of computer graphics theory you could try *OpenGL: A Primer* (Second Edition) by Edward Angel (Pearson, 2005). The book's code is available at http://www.cs.unm.edu/~angel/BOOK/PRIMER/SECOND_EDITION/PROGRAMS/.

If you don't have a background in computer graphics you should probably switch to Angel's more technical book, *Interactive Computer Graphics: A Top-Down Approach Using OpenGL* (Fourth Edition) (Addison Wesley, 2005).

A good OpenGL text with a gaming slant is *OpenGL Game Programming* by Kevin Hawkins and David Astle (Premier Press, 2001). The examples use the complex Microsoft windowing library, wgl, but they're still fun and informative. The online support page is <http://glbook.gamedev.net/oglgp.asp>.

Astle and Hawkins have released two more recent books: *Beginning OpenGL Game Programming* (Course Technology, 2004) and *More OpenGL Game Programming* (Course Technology, 2005), which cover similar ground and more advanced topics, such as programmable shaders. Details are available at <http://glbook.gamedev.net/>.

The ultimate OpenGL programming text, *OpenGL Programming Guide: The Official Guide to Learning OpenGL Version 2* (Fifth Edition) by the OpenGL Architecture Review Board (Addison-Wesley, 2005), is known as the “red book” in OpenGL circles. An early version, for OpenGL 1.0, is online at http://www.opengl.org/documentation/red_book/ in PDF and HTML formats and at <http://www.gamedev.net/download/redbook.pdf>.

Summary

I introduced JOGL in this chapter by coding a simple rotating multicolored cube using several approaches.

The *callback framework* utilizes an animator, an event listener, and a drawing surface. I coded two variants of it—one with the fast GLCanvas class, the other using GLJPanel. Their speeds were roughly equivalent, but GLJPanel’s performance depends on the suitability of the hardware, graphics driver, and Java version. Its main advantage is its ability to closely integrate with other Swing components to create novel, fun 2D/3D GUIs. An important issue is that the `-Dsun.java2d.opengl=true` argument needed for GLJPanel’s speed may cause crashes on older graphics hardware and drivers.

The cube application was also implemented using *active rendering*, a technique only made possible with the recent extensions to JOGL to make it JSR-231 compliant. Active rendering is as fast as the callback approach and allows finer control over the application’s timing behavior. For example, a poor frame rate on a slow machine can be compensated for by performing updates without the slow rendering. The downside is the increased complexity of the rendering loop and the fact that most JOGL/OpenGL examples use the callback approach, offering more help to novice programmers.