Notes on a JOGL Active Rendering Framework

Andrew Davison
Department of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla 90112, Thailand
ad@fivedots.coe.psu.ac.th

ABSTRACT

These notes describe an active rendering framework for JOGL which updates and renders a game (or any animated application) at a reliable, near constant, framerate. It also allows greater control over the application's execution behavior, such as how it pauses, resumes, and deals with resizing.

These notes form part of a tutorial held at *CyberGames 2007*. The main aim is to introduce JOGL and OpenGL to an audience unfamiliar with 3D graphics through the means of a simple 3D game. The tutorial includes pointers to numerous sources of further information.

Categories and Subject Descriptors

I.3.6 [**Computer Graphics**]: Methodology and Techniques – *graphics data structures and data type, interaction techniques, languages.*

General Terms

Algorithms, Performance, Design, Languages.

Keywords

JOGL, OpenGL, active rendering, animation, framework.

1. INTRODUCTION

JOGL is an open-source technology initiated by the Game Technology Group at Sun Microsystems in 2003 (https://jogl.dev.java.net/). JOGL provides full access to the APIs in the OpenGL 2.0 specification (http://www.opengl.org/), as well as vendor extensions, and can be combined with Java's AWT and Swing GUI components. It supports both 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 dealt with by other Java APIs.

CyberGames 2007, September 10–11, 2007, Manchester, UK.

The active rendering framework described here utilizes JOGL features for accessing the application's drawing surface and context (OpenGL's internal state) [2]. The surface is typically a subclass of AWT's Canvas, and is manipulated with a dedicated rendering thread, as illustrated by Figure 1.

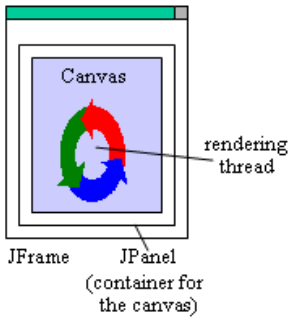

Figure 1. An active rendering application.

The rendering thread's pseudocode:

```
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;
}
discard the rendering context;
exit;
```

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

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.

2. AN ACTIVE RENDERING EXAMPLE

A simple application using the active rendering framework is shown in Figure 2. The program, called CubeGL, rotates a multi-colored cube around its x-, y-, and z- axes.

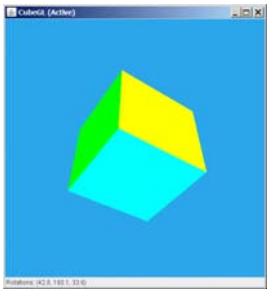

Figure 2. The CubeGL application.

The class diagrams for CubeGL are given in Figure 3.

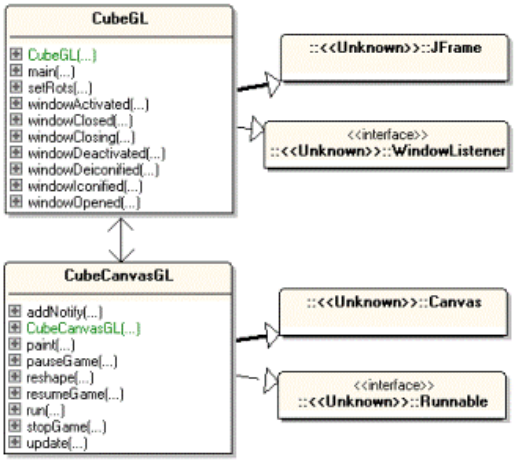

Figure 3. Class diagrams for CubeGL with active rendering.

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.

2.1 Thread Rendering

The run() method in CubeCanvasGL is based on the pseudocode given earlier:

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

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

2.2 Rendering Initialization

The initRender() method initializes the OpenGL link, which includes its context, access to the OpenGL APIs, and setting up the viewport.

```
// globals
private GL gl;    // to access OpenGL and
private GLU glu;  // OpenGL's utility libraries

private void initRender()
{
  makeContentCurrent();

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

  resizeView();

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

  // z- (depth) buffer 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();

  context.release();
}  // end of initRender()
```

An OpenGL display list acts as a storage space for 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 initRender() groups the commands that draw the cube. This part of initRender() will vary from application to application.

makeContentCurrent() connects OpenGL's graphic context to the thread:

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

makeCurrentContext() calls GLContext.makeCurrent(), which should immediately succeed, since no other thread is using the context. The while-loop around the call is extra protection, since the application will crash if OpenGL commands are called without the thread holding the current context.

resizeView() sets the OpenGL camera viewport dimensions, and specifies a perspective view into the scene:

```
// globals
private int panelWidth, panelHeight;
            // dimensions of the JPanel

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

  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()
```

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 viewing angle.

2.3 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;
}
```

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 for short), 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();
    drawable.swapBuffers();
                  // put 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) { // time left in cycle
     try {
       Thread.sleep(sleepTime/1000000L);//nano->ms
     }
     catch(InterruptedException ex){}
     overSleepTime = (System.nanoTime()-afterTime)
                  - sleepTime;
   }
   else {  // sleepTime <= 0;
     // this cycle took longer than 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;

    context.release();
  }
} // 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 some time to process AWT events.

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

2.4 Rendering the Scene

Scene generation is carried out by renderScene():

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

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

  if (isResized) { // resize 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

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

renderScene() checks 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 the window has been resized (i.e. when isResized is true).

The matrix mode is switched to MODELVIEW so OpenGL's model-view matrix can be utilized. It defines the scene's coordinate system, used when positioning or moving 3D objects.

After the new rotations have been applied to the world coordinates, the cube is drawn via its display list. This part of renderScene() will vary from application to application.

The method finishes by checking the gameOver boolean, and printing a simple message. In a real game, the output would be more complicated.

3. MORE INFORMATION

The principal source for JOGL help is its forum site at http://www.javagaming.org/forums/index.php?board=25.0.

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.

There are a growing number of textbooks on OpenGL (e.g. [1, 3, 4], with a comprehensive list available at http://www.opengl.org/documentation/books.html.

4. REFERENCES

[1] Angel, E. *OpenGL: A Primer*, Pearson, 2005, 2nd ed., `http://www.cs.unm.edu/~angel/`

[2] Davison, A. *Pro Java 6 3D Game Developmen*t, Apress, 2007, `http://fivedots.coe.psu.ac.th/~ad/jg2/`

[3] Hawkins, K., and Astle, D. *Beginning OpenGL Game Programming*, Course Technology, 2004, `http://glbook.gamedev.net/`

[4] The OpenGL Architecture Review Board, *OpenGL Programming Guide: The Official Guide to Learning OpenGL Version 2*, Addison-Wesley, 2005, `http://www.opengl.org/documentation/red_book/`