Lab 1 – JOGL setup and Basic Rendering Loop

Based on Tutorial at: https://sites.google.com/site/justinscsstuff/jogl-tutorial-1

JOGL (Java Bindings for openGL): JOGL provides access to the OpenGL graphics API, enabling hardware-accelerated graphics with Java. JOGL is currently maintained by JogAmp.

You can get the library from this address (not the most recent, but its ok):

http://download.java.net/media/jogl/builds/archive/jsr-231-2.0-beta10/

I have put up the windows libraries you need on moodle (this version is JOGL 2.0). There are a number of files you need but these can all be bundled in the one folder. It's not too important where you put these but put them in a sensible location. For example C:\jogl2.0\lib.

Now download the javadoc and expand this into a folder, again choose something along the lines of C:\jogl2.0\javadoc

Once you have the files you can open Eclipse, we need to make some configuration settings in here.

Part 1: Creating JOGL User Library (you should only need to do this once)

To use JOGL, it's necessary for your project to reference it. The JOGL library consists of a number of JAR files and native libraries, but it's preferable to refer to this collection as a whole rather than each file separately. To do this, create a "user library" that lists all of the required files.

- In the top Eclipse menu bar, click "Window > Preferences"
- The left side of the preferences frame should show a list of options. Make sure you have "Java > Build Path > User Libraries" selected.
- Click "New..." and type a name for your library, such as "jogl2.0". Leave the "System library (added to the boot class path)" checkbox unchecked and click "OK".
- Back in the "User Libraries" panel, you should see your library appear. Click "Add JARs..." and navigate to the location where you have JOGL library files.
- Select the following JARs: gluegen-rt.jar, jogl.all.jar, nativewindow.all.jar, newt.all.jar
- Click OK to add these four JARs to your user library. You should see them listed under "jogl2.0" now.
- Expand each JAR by clicking the button to its left. This should show you four properties for each JAR:
  - Source attachment is where you would link the JAR with the source code (.java files) corresponding to its .class files. This would let you view the source code of the library inside the IDE when looking through classes from the library.
  - Javadoc location refers to the HTML Javadoc files that provide programmers with information about the methods, classes, interfaces, and other elements of code inside the JAR. If you download the Javadoc for JOGL, you can see the documentation inside Eclipse.
  - Native library location indicates where platform specific libraries are located. Native libraries may be needed by the code inside a JAR files to work.
  - Access rules allow you to restrict usage of some code in a JAR file, which we don't care about here.
- We will configure the native library location for the JAR files. Click "Native library location" and then "Edit..." on the right.

- Select "External Folder..." and navigate to the directory where you have the JOGL library files. The path you enter should be the directory containing .so or .dll files. Click "OK".
- Next you want to have the Javadoc for the JOGL libraries available to you in Eclipse. This is a simple enough step. Expand each JAR again by clicking the button to its left which should give you the list of options again.
- Click the Javadoc option and Eclipse will ask you where the location of the Javadoc is. Navigate to the folder containing the Javadoc and select it. Eclipse performs a check to see if the Javadoc correlates with the library so it should report that the Javadoc is valid for each library.

Part 2: Linking JOGL with your Java Project

You're done setting up the JOGL user library, and you should only have to do this once unless you download a newer version of JOGL at some point. Now you simply have to tell your project to use your JOGL user library.

- Create a new Java project in Eclipse as you normally would. Call it some appropriate name, Graphics Lab 1 or something.
- In the package explorer, right click the project folder and select "Properties" at the bottom of the drop-down list.
- On the left side of the properties frame, select "Java Build Path" and click the "Libraries" tab at the top.
- Click "Add Library...", select "User Library" and click "Next", then put a check mark next to JOGL2.0 and click "Finish". You should see JOGL-2.0 in the "Libraries" list now, so click "OK" to exit properties.
- In the Package Explorer, you'll also see the library is now referenced by your project. Now you can use the code inside the JOGL library in any Java source code belonging to this project.

Part 3: Writing Java programs that use openGL

OpenGL is for making graphics. It is fast. Most of the time, it is hardware accelerated. It seems that OpenGL can do anything visually that you would want to do. OpenGL is written for the C programming language. One of the biggest drawbacks to OpenGL is that you can't make it do anything without a window to put your graphics in, but OpenGL doesn't provide a means for you to create windows. This makes OpenGL hard to learn for beginners. Luckily, GLUT (OpenGL Utility Toolkit) was introduced and made dealing with windows, buttons, and events generated by users easier. Still, learning OpenGL in C or even C++ can be painful for new programmers or programmers that want to use true object-oriented programming.

Then came JOGL

Java is possibly the most popular true object-oriented programming language. There have been many attempts to marry OpenGL with Java, but the first one that made everyone stand up and take notice was Java Bindings for OpenGL, or JOGL. The reason for this is that this effort is supported by Sun Microsystems (the creators of Java) and SGI (the creators of OpenGL).

So what are the main parts to your JOGL project? There are some initialisations that need to be performed in JOGL to set things up. These should be done for all the JOGL applications you create.

Basic JOGL components

GLProfile

The field of computer graphics changes rapidly. Unlike many programming APIs, graphics APIs must often break backwards compatibility with each revision. New versions of OpenGL typically incorporate new functionality while deprecating or entirely removing older functionality. Programming OpenGL applications is not as simple as using the most recent version of OpenGL, because many users will not be able to run these applications. Even if users update their graphics drivers, their hardware itself must support the core features of the OpenGL version a program was written against.

It is important that programmers have control over which version of OpenGL is used by the application, and the GLProfile class allows just that. If a user has hardware drivers that support OpenGL 4.0, for example, they can still run OpenGL 2.1 programs. By using a GLProfile, the programmer can write their program in a manner that supports all users while still providing newer, more advanced functionality if is available. An instance of the GLProfile object can specify which versions are supported by the client and allow the program to conditionally use different pieces of code.

To select OpenGL 2.1, for example, a GLProfile would be created as follows:

```
GLProfile glp = GLProfile.get(GLProfile.GL2);
```

The system's default OpenGL profile can be selected as well, which represents the profile which is "best for the running platform":

```
GLProfile glp = GLProfile.getDefault();
```

NOTE:

The developers of JOGL highly recommend calling the static method initSingleton before doing anything else in a JOGL program. This appears to be particularly relevant if you are using Linux, intend to perform any GUI work, or you're writing a multi-threaded application. The method allows JOGL to prepare some "Linux specific locking optimization" and what may otherwise be some unrecoverable crashes. It is suggested you add the following code to your main method before doing anything else:

```
GLProfile.initSingleton(true);
```

GLContext (OpenGL Rendering Context)

OpenGL is essentially a state machine, which might cause some confusion if you've only experienced pure object-oriented programming. States in OpenGL are things like current color, active drawing mode, if lighting is enabled, transformation details, and so on. An OpenGL rendering context is what keeps track of all these states. In JOGL, this context is abstracted into the GLContext class:

"In order to perform OpenGL rendering, a context must be 'made current' on the current thread. OpenGL rendering semantics specify that only one context may be current on the current thread at any given time, and also that a given context may be current on only one thread at any given time. Because components can be added to and removed from the component hierarchy at any time, it is possible that the underlying OpenGL context may need to be destroyed and recreated multiple times over the lifetime of a given component. This process is handled by the implementation, and the GLContext abstraction provides a stable object which clients can use to refer to a given context."

GLCapabilities

This object describes some specific capabilities a rendering context should support. The default settings are usually fine, but you can play with some of the capabilities for achieving certain effects such as full-scene anti-aliasing, stereo rendering, and so forth. This object takes an instance of GLProfile as a parameter.

```
GLCapabilities caps = new GLCapabilities(glp);
```

GLDrawable / GLAutoDrawable

These are interfaces that describes a target for OpenGL rendering, and it has the purpose of creating and maintaining the rendering context. Implementations of GLAutoDrawable automatically create that rendering context, while GLDrawable requires the programmer to do so. I will focus on GLAutoDrawable, as the details of creating a context aren't necessary for basic graphics programming.

The rendering mechanism of GLAutoDrawable is event-based, so the programmer provides an event listener (GLEventListener) to any implementation of GLAutoDrawable to perform the actual rendering. This class and the rendering process will be described in more detail later.

Creating a Window with AWT

To work with AWT, JOGL provides a class GLCanvas which extends the AWT Component class, so it can be added to an AWT/Swing GUI. There are a few steps involved in creating a window that is prepared for OpenGL rendering:

1. Choose GLProfile
2. Configure GLCapabilities
3. Create GLCanvas (implementation of GLAutoDrawable), which automatically creates our GLContext
4. Add GLCanvas to Frame (the AWT Frame is our window)

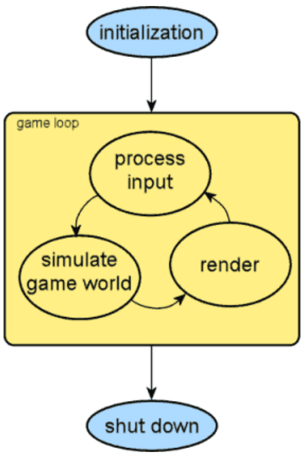
Here's an example that will create a window using AWT. This is kept as simple as possible to show the minimum code to get a window up and running with AWT. The actual OpenGL rendering, animation loop, and GLEventListener details will be covered later.

```
import java.awt.Frame;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.media.opengl.*;
import javax.media.opengl.awt.GLCanvas;

public class SimpleScene {
     public static void main(String[] args) {
          GLProfile glp = GLProfile.getDefault();
          GLCapabilities caps = new GLCapabilities(glp);
          GLCanvas canvas = new GLCanvas(caps);

          Frame frame = new Frame("AWT Window Test");
          frame.setSize(300, 300);
          frame.add(canvas);
          frame.setVisible(true);
          // by default, an AWT Frame doesn't do anything when you
click
          // the close button; this bit of code will terminate the
program when
          // the window is asked to close
          frame.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
          System.exit(0);
          }
          });
     }
}
```

The result of running this program will actually vary depending on your windowing system and graphics drivers, it may be filled with black or white, contain garbled graphics.

Part 4: The Rendering Loop

In computer graphics applications you have a rendering loop where you might alter the state of things, move things, receive events and interactions etc and then update the display. This happens in one big loop called the rendering loop. If the application were a game, the loop might be illustrated according to the following diagram:



So what happens at each stage in terms of a JOGL application:

Initialization

- Choosing a GLProfile and configuring GLCapabilities for a rendering context
- Creating a window and GLContext through the GLAutoDrawable
- Making an animator thread
- Loading resources needed by program

Process Input

- Listen for mouse and keyboard events
- Update user's view (often called a camera)

Update (Simulate Game World)

- Calculate geometry
- Rearrange data
- Perform computations

Render

- Draw scene geometry from a particular view

Shut Down

- Save persistent data
- Clean up resources on graphics card

Using this type of design is useful for almost any type of application, because it's flexible and probably won't cause problems later on. It is especially important to separate the update and render stages from each other. If not to adhere to better software engineering practices, consider the following advantages of separating these two stages:

- You may want to render a scene multiple times from different views or with different effects. You do not want to repeat the update logic for each rendering. In other words, rendering should be able to reproduce the same results until the next update.
- Keeping a consistent loop, especially when the program is receiving data in real-time, is not as simple as it might initially seem. It's also nice to have a smoother, more consistent framerate (even if it's lower) as opposed to a jittery one.
- You may want to allow users to scale performance by setting a target framerate without affecting the rate of computations in the update part. If the user has a poor graphics card,

the rendering itself may dramatically slow down the simulation running on the CPU. It may be preferable to render after every other update, for example.

Example - Drawing a Triangle

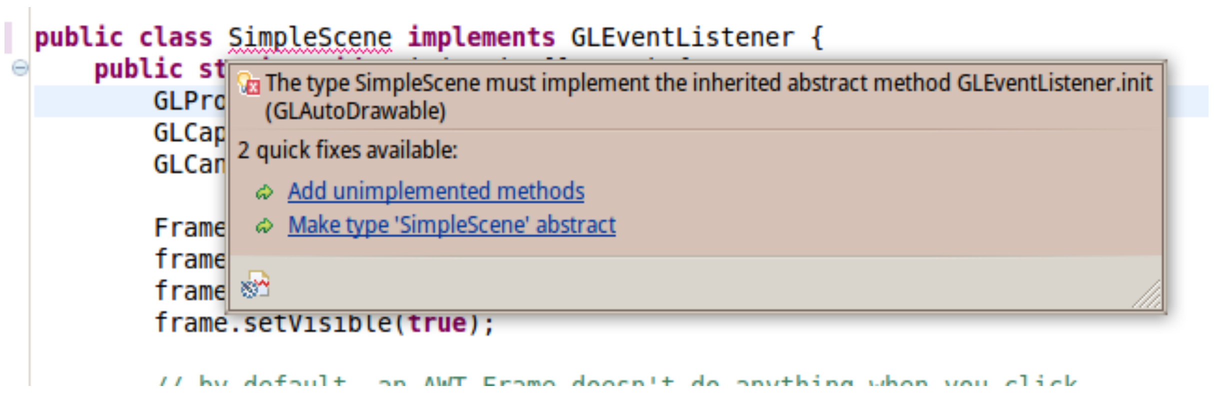
To illustrate how to implement a render loop, we'll make a program that draws a triangle. We're going to build on top of the SimpleScene.java (AWT version) described in the previous tutorial; instead of having a blank window, the result will be a large multi-colored triangle inside the window.

Implementing GLEventListener Interface

The first step is to make the SimpleScene class implement GLEventListener so it can listen for rendering events. Modify the class declaration as follows:

```
public class SimpleScene implements GLEventListener {
```

If you're using Eclipse, you'll notice SimpleScene is underlined in red, meaning there is an error. This is because the class doesn't implement the methods in the GLEventListener interface yet. If you hover your mouse over the underlined SimpleScene, a popup will give an option to "Add unimplemented methods", which will automatically add stubs in your class such that it conforms to the interface:



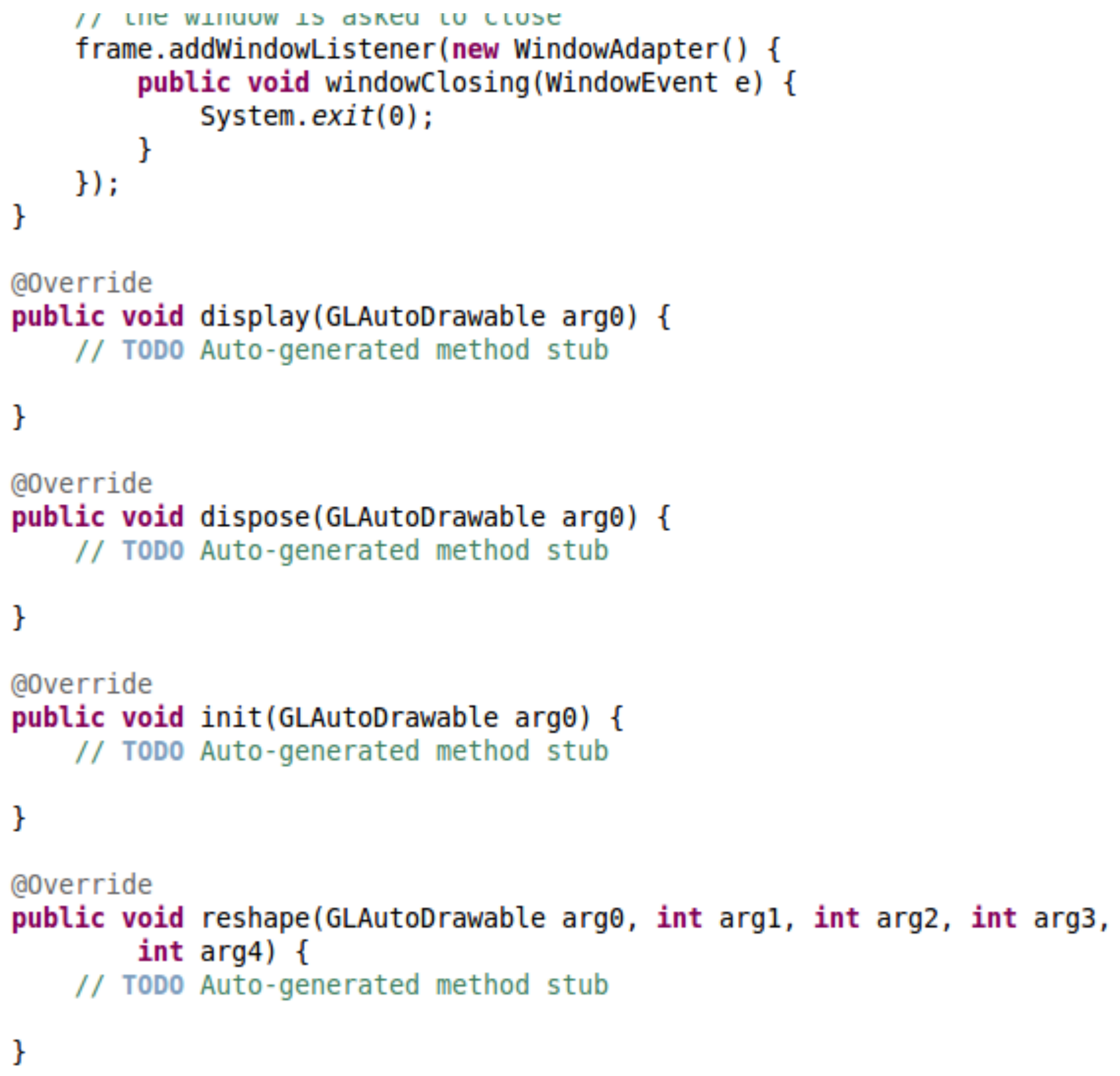
```
public class SimpleScene implements GLEventListener {
    public st
        GLPro
        GLCap
        GLCan

        Frame
        frame
        frame
        frame.setVisible(true);
```

The type SimpleScene must implement the inherited abstract method GLEventListener.init (GLAutoDrawable)

2 quick fixes available:

- Add unimplemented methods
- Make type 'SimpleScene' abstract

If you click the link to add the unimplemented methods, Eclipse will create the methods for you (you can delete the @Override annotations and TODO comments if they bother you):



```
        // the window is asked to close
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

    @Override
    public void display(GLAutoDrawable arg0) {
        // TODO Auto-generated method stub

    }

    @Override
    public void dispose(GLAutoDrawable arg0) {
        // TODO Auto-generated method stub

    }

    @Override
    public void init(GLAutoDrawable arg0) {
        // TODO Auto-generated method stub

    }

    @Override
    public void reshape(GLAutoDrawable arg0, int arg1, int arg2, int arg3,
            int arg4) {
        // TODO Auto-generated method stub

    }
```

Otherwise, you must manually type the four methods of GLEventListener inside the SimpleScene class.

You can read the documentation on GLEventListener for more details on what these methods are used for. Having SimpleScene implement this interface allows it to render to a GLAutoDrawble; since this is an AWT example, that means the GLCanvas.

Next, we have to let the GLAutoDrawable (GLCanvas) know we want a SimpleScene object to listen for rendering events. This is easy, and takes one line of code at the end of the main method:

```
canvas.addGLEventListener(new SimpleScene());
```

Making the Triangle

At this point, your program is ready to draw something. First, split the display method into an update and render stage:

```
public void display(GLAutoDrawable drawable) {
update();
render(drawable);
}
```

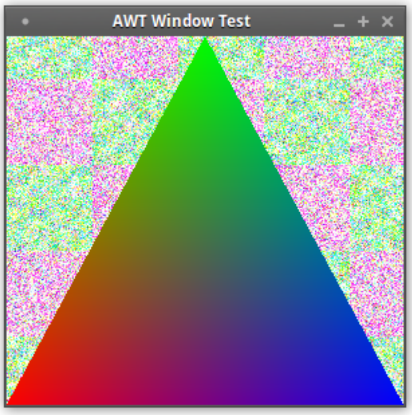
Create the update method, which for now has nothing in it:

```
private void update() {
// nothing to update yet
}
```

Create the render method, which will draw the triangle:

```
private void render(GLAutoDrawable drawable) {
      GL2 gl = drawable.getGL().getGL2();
      // draw a triangle filling the window
      gl.glBegin(GL.GL_TRIANGLES);
      gl.glColor3f(1, 0, 0);
      gl.glVertex2f(-1, -1);
      gl.glColor3f(0, 1, 0);
      gl.glVertex2f(0, 1);
      gl.glColor3f(0, 0, 1);
      gl.glVertex2f(1, -1);
      gl.glEnd();

}
```

In an OpenGL program written in C or C++, much of this render method would look the same. What's different is that we have this GL object floating around which exposes the OpenGL API. Notice that to call the glBegin, glEnd, glVertex, and glColor OpenGL functions we have to prefix them with "gl.". The "gl" object is a type of GL2, which in turn is a type of GL. Recall that we have a GLProfile specifying the OpenGL version. This program assumes that the user's computer can handle the GL2 profile, and we have to use GL2 to access the above functions (they are no longer available in GL3, for example). This GL2 object can be retrieved from the GLCanvas (our GLAutoDrawable), which is why it's a parameter for this method. If you run your program now, you should get something like this:

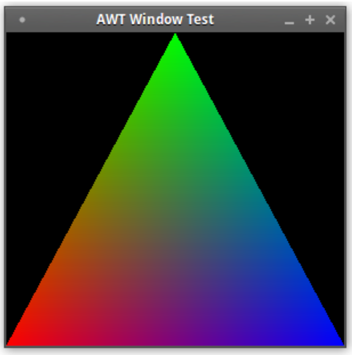


There are two "problems" with the result here. The first may be obvious (or may not be, depending on your platform): the background is still garbage. It would be much better if we could set it to a solid color to admire the beautiful triangle. The second problem you probably didn't notice is that the display method is not being called repeatedly; we don't actually have a rendering loop yet.

Clearing the Color Buffer

The first problem is a result of having "garbage" in the graphics card's memory, the color buffer. A graphics card stores color values for the pixels in this buffer; when you perform drawing using OpenGL, these values are modified. That's why the area covered by the triangle looks fine - we've manually drawn to those pixels. We haven't asked OpenGL to do anything with the other pixels, however, so whatever values are stored there is anyone's guess. There is an OpenGL function glClear that will efficiently set all the pixels in this buffer to a desired color (default is black). The function can actually clear other buffers too (such as the depth buffer, which you will find out about later), but we only care about clearing colors for now. Add the following line to your render method before drawing the triangle:

```
gl.glClear(GL.GL_COLOR_BUFFER_BIT);
```



Starting the Animation Loop

JOGL provides some utility classes for animating our program. An Animator object can be created to ensure the display method of a GLAutoDrawable is repeatedly called. An FPSAnimator allows us make the framerate relatively consistent and can reduce the resource consumption of the program (if it is simple enough to run faster than the target framerate). Add the following lines to the end of the main method:

```
Animator animator = new FPSAnimator(canvas, 60);
animator.add(canvas);
animator.start();
```

The animator is attached to the GLCanvas (again, our GLAutoDrawable) and asked to render at roughly 60 frames per second. In other words, the display method will be called approximately every 17 ms (1000 / 60). The animator classes need to be imported, so add the following statement to the top of your program:

```
import com.sun.opengl.util.*;
```

Now our program is fulfills the rendering loop design above. Because the program is so simple, we don't have any resources to clean up in the dispose method; the window resources are automatically cleaned up for us when the program exits. However, it is important to remember that resources on the graphics card, such as textures, shaders, or vertex buffer objects, should be released. Just because we're using Java doesn't mean we can be sloppy and let garbage collection take care of everything! This isn't part of the current example, however, so the dispose method can be ignored for now.

NOTE:
If you use a regular Animator object instead of the FPSAnimator, your program will render as fast as possible. You can, however, limit the framerate of a regular Animator by asking the graphics driver to synchronize with the refresh rate of the display (v-sync). Because the target framerate is often the same as the refresh rate, which is often 60-75 or so, this method is a great choice as it lets the driver do the work of limiting frame changes. However, some Intel GPUs may ignore this setting. In the init method, active v-sync as follows:

```
public void init(GLAutoDrawable drawable) {
 drawable.getGL().setSwapInterval(1);
}
```

Then, in the main method, replace the FPSAnimator with a regular Animator:

```
Animator animator = new Animator(canvas);
```

Adding Some Animation

Finally, we'll demonstrate the rendering is now animated by making the triangle bounce around inside the window. Create three fields inside the SimpleScene class:

```
private double theta = 0;
private double s = 0;
private double c = 0;
```

We'll change theta over time, and the variables s and c refer to the sine and cosine of theta. The proper place to put these calculations is in the update method:

```
private void update() {
    theta += 0.01;
    s = Math.sin(theta);
    c = Math.cos(theta);
}
```

Now, instead of using hard-coded values for the vertices of the triangle we'll use the s and c variables which are changing over time. Replace the triangle draw code as follows:

```
gl.glBegin(GL.GL_TRIANGLES);
gl.glColor3f(1, 0, 0);
gl.glVertex2d(-c, -c);
gl.glColor3f(0, 1, 0);
gl.glVertex2d(0, c);
gl.glColor3f(0, 0, 1);
gl.glVertex2d(s, -s);
gl.glEnd();
```

Lab work

- Read though the above and try and understand the structure of a JOGL program.
- The complete version of this Program is on moodle, have a look and get it working.
- In addition to this you should implement Bresenham's line and circle drawing algorithms that were discussed in class.
- To do this you should create another project like the one above and remove the code to draw/update the triangle and add in the code to display a point (this is currently commented out in the program). Then modify this code to display the points for the following,
- Line endpoints - (0,0) -> (200,170)
- Circle with centre in the middle of the window (150,150) and radius 100. Hint: get it working for one quadrant first with circle centre = (0,0)