
5

OpenGL Programming in Java: JOGL

Chapter Objectives:

- Set up Java, JOGL programming environments
- Understand simple JOGL programs

5.1 Introduction

JOGL implements Java bindings for OpenGL. It provides hardware-supported 3D graphics to applications written in Java. It is part of a suite of open-source technologies initiated by the Game Technology Group at Sun Microsystems. JOGL provides full access to OpenGL functions and integrates with the AWT and Swing widget sets.

First, let's spend some time to set up our working environment, compile `J1_0_Point.java`, and run the program. The following file contains links to all the example programs in this book and detailed information for setting up working environments on different platforms for the most recent version:

<http://cs.gmu.edu/~jchen/graphics/setup.html>

Since JOGL has been changed significantly over time, it is better to download and update the sample programs from the web instead of typing in examples from the book.

5.2 *Setting Up Working Environment*

JOGL provides full access to the APIs in the OpenGL specification as well as nearly all vendor extensions. To install and run JOGL, we need to install Java Development Kit. In addition, a Java IDE is also preferred to help coding. The following steps will guide you through installing Java, JOGL, and Eclipse or JBuilder IDE.

1. Installing Java Development Kit

Java Development Kit (JDK) contains a compiler, interpreter, and debugger. If you have not installed JDK, it is freely available from Sun Microsystems. You can download the latest version from the download section at <http://java.sun.com>. Make sure you download the JDK Java SE (Standard Edition) not the JRE (runtime environment) that matches the platform you use. After downloading the JDK, you can run the installation executable file. During the installation, you will be asked the directory "Install to:". You need to know where it is installed. For example, you can put it under: "C:\myJDK\". In default, it is put under "C:\Program Files\Java\jdkxxx\".

2. Installing JOGL

We need to obtain the JOGL binaries in order to compile and run applications from: <https://jogl.dev.java.net/>. Download the current release build binaries that match the platform you use. After that, you can extract and put all these files (jar and dll files) in the same directory with the Java (JOGL) examples and compile all them on the command line in the current directory with:

```
"C:\myJDK\bin\javac" -classpath jogl.jar *.java
```

After that, you can run the sample program with (the command in one line):

```
"C:\myJDK\bin\java" -classpath .;jogl.jar;gluegen-rt.jar;  
-Djava.library.path=. J1_0_Point
```

That is, you need to place the "*.jar" files in the CLASSPATH of your build environment in order to be able to compile an application with JOGL and run, and place "*.dll" files in the directory listed in the "java.library.path" environment variable during execution. Java loads the native libraries (such as the dll files for Windows) from the directories listed in the "java.library.path" environment variable. For Windows, placing the dll files under "C:\WINDOWS\system32\" directory works. This approach gets you up running quickly without worrying about the "java.library.path" setting.

3. Installing a Java IDE (Eclipse, jGRASP, or JBuilder)

Installing a Java IDE (Integrated Development Environment) is strongly recommended. Without an IDE, you can edit Java program files using any text editor, compile and run Java programs using the commands we introduced above after downloading JOGL, but that would be very difficult and slow. Java IDEs such as JBuilder, Eclipse, and jGRASP are development environments that make Java programming much faster and easier.

If you like to use jGRASP, you can download it from <http://www.jgrasp.org/>. In the project under "Settings->PATH/CLASSPATH->Workspace", you can add the directory of the *.dll files to the system PATH window, and add "*.jar" files with full path to the CLASSPATH window.

If you like to use Eclipse, you can download from <http://eclipse.org> the latest version of Eclipse that matches the platform you use. Expand it into the folder where you would like Eclipse to run from, (e.g., "C:\eclipse\"). There is no installation to run. You can put "*.jar" files under "Project->Properties->Libraries". To remove Eclipse you simply delete the directory, because Eclipse does not alter the system registry.

As another alternative, you can download a free version of JBuilder from <http://www.borland.com/jbuilder/>. JBuilder comes with its own JDK. If you use JBuilder as the IDE and want to use your downloaded JDK, you need to start JBuilder, go to "Tools->Configure JDKs", and click "Change" to change the "JDK home path:" to where you install your JDK. For example, "C:\myJDK\". Also, under "Tools->Configure JDKs", you can click "Add" to add "*.jar" files from wherever you save it to the JBuilder environment.

4. Creating a Sample Program in Eclipse

As an example, here we introduce using Eclipse. After downloading it, you can run it to start programming. Now in Eclipse you click on "File->New->Project" to create a new *Java Project* at a name you prefer. Then, you click on "File->New->Class" to create a new class with name: "J1_0_Point". After that, you can copy the following code into the space, and click on "Run->Run As->Java Application" to start compiling and running. You should see a window with a very tiny red pixel at the center. In the future, you can continue creating new classes, as we introduce each example as a new class. Alternatively, you can download all the examples from the web.

/* draw a point */

```

/* Java's supplied classes are "imported". Here the awt
(Abstract Windowing Toolkit) is imported to provide "Frame"
class, which includes windowing functions */
import java.awt.*;

// JOGL: OpenGL functions
import javax.media.opengl.*;

/* Java class definition: "extends" means "inherits". So
Jl_0_Point is a subclass of Frame, and it inherits Frame's
variables and methods. "implements" means GLEventListener is
an interface, which only defines methods (init(), reshape(),
display(), and displaychanged()) without implementation. These
methods are actually callback functions handling events.
Jl_0_Point will implement GLEventListener's methods and use
them for different events. */

```

```

public class Jl_0_Point extends Frame implements
    GLEventListener {

    static int HEIGHT = 600, WIDTH = 600;
    static GL gl; //interface to OpenGL
    static GLCanvas canvas; // drawable in a frame
    static GLCapabilities capabilities;

    public Jl_0_Point() {

        //1. specify a drawable: canvas
        capabilities = new GLCapabilities();
        canvas = new GLCanvas();

        //2. listen to the events related to canvas: reshape
        canvas.addGLEventListener(this);

        //3. add the canvas to fill the Frame container
        add(canvas, BorderLayout.CENTER);
        /* In Java, a method belongs to a class object.
        Here the method "add" belongs to Jl_0_Point's
        instantiation, which is frame in "main" function.
        It is equivalent to use "this.add(canvas, ...)" */

        //4. interface to OpenGL functions
        gl = canvas.getGL();
    }

    public static void main(String[] args) {

```

```
Jl_0_Point frame = new Jl_0_Point();

//5. set the size of the frame and make it visible
frame.setSize(WIDTH, HEIGHT);
frame.setVisible(true);
}

// called once for OpenGL initialization
public void init(GLAutoDrawable drawable) {

    //6. specify a drawing color: red
    gl.glColor3f(1.0f, 0.0f, 0.0f);
}

// called for handling reshaped drawing area
public void reshape(
    GLAutoDrawable drawable,
    int x,
    int y,
    int width,
    int height) {

    WIDTH = width; // new width and height saved
    HEIGHT = height;

    //7. specify the drawing area (frame) coordinates
    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrtho(0, width, 0, height, -1.0, 1.0);
}

// called for OpenGL rendering every reshape
public void display(GLAutoDrawable drawable) {

    //8. specify to draw a point
    //gl.glPointSize(10);
    gl.glBegin(GL.GL_POINTS);
    gl.glVertex2i(WIDTH/2, HEIGHT/2);
    gl.glEnd();
}

// called if display mode or device are changed
public void displayChanged(
    GLAutoDrawable drawable,
    boolean modeChanged,
```

```
        boolean deviceChanged) {  
    }  
}
```

5.3 Drawing a Point

The above *Jl_0_Point.java* is a Java application that draws a red point using JOGL. If you are a C/C++ programmer, you should read all the comments in the sample program carefully, because they include explanations about Java-specific terminologies and coding. Our future examples are built on top of this one. Here we explain in detail. The program is complex to us at this point of time. We only need to understand the following:

1. Class `GLCanvas` is an Abstract Window Toolkit (AWT) component that provides OpenGL rendering support. Therefore, the `GLCanvas` object, `canvas`, corresponds to the drawing area that will appear in the `Frame` object `frame`, which corresponds to the display window. Here *object* means an instance of a class in object-oriented programming, not a 3D object. In the future, we omit using a class name and underline its object name in our discussion. In many cases, object names are lowercases of the corresponding class names to facilitate understanding.
2. An *event* is a user input or a system state change, which is queued with other events to be handled. Event handling is to register an object to act as a listener for a particular type of event on a particular component. Here `frame` is a listener for the GL events on `canvas`. When a specific event happens, it sends `canvas` to the corresponding event handling method and invokes the method. `GEventListener` has four event-handling methods:
 - *init()* is called immediately after the OpenGL context is initialized for the first time, which is a system event. It can be used to perform one-time OpenGL initialization;
 - *reshape()* is called if `canvas` has been resized, which happens when the user changes the size of the window. The listener also passes the drawable `canvas` and the display area's lower-left corner (x, y) and size (*width*, *height*) to the method. At this time, (x, y) is always $(0, 0)$, and the `canvas`' size is the same as the display window's `frame`. The client can update the coordinates of the display

corresponding to the resized window appropriately. *reshape()* is called at least once when program starts. Whenever *reshape()* is called, *display()* is called as well;

- *display()* is called to initiate OpenGL rendering when program starts. It is called afterwards when reshape event happens;
 - *displayChanged()* is called when the display mode or the display device has been changed. Currently we do not use this event handler.
3. `canvas` is added to `frame` to cover the whole display area. `canvas` will reshape with `frame`.
 4. `gl` is an interface handle to OpenGL methods. All OpenGL commands are prefixed with “gl” as well, so you will see OpenGL method like *gl.glColor()*. When we explain the OpenGL command, we often omit the interface handle.
 5. Here we set the physical size of `frame` and make its contents visible. Here the physical size corresponds to the number of pixels in *x* and *y* direction. The actual physical size also depends on the *resolution* of the display, which is measured in number of pixels per inch. At this point, the window frame appears. Depending on the JOGL version, the physical size may include the boarders, which is a little larger than the visible area that is returned as *w* and *h* in *reshape()*.
 6. The foreground drawing color is specified as a vector (red, green, blue). Here (1, 0, 0) represents a red color.
 7. These methods specify the logical coordinates. For example, if we use the command *glOrtho(0, width, 0, height, -1.0, 1.0)*, then the coordinates in `frame` (or `canvas`) will be $0 \leq x \leq \text{width}$ from the left side to the right side of the window, $0 \leq y \leq \text{height}$ from the bottom side to the top side of the window, and $-1 \leq z \leq 1$ in the direction perpendicular to the window. The *z* direction is ignored in 2D applications. It is a coincidence that the logical coordinates correspond to the physical (pixel) coordinates, because *width* and *height* are initially from `frame`’s WIDTH and HEIGHT. We can specify *glOrtho(0, 100*WIDTH, 0, 100*HEIGHT, -1.0, 1.0)* as well, then point (*WIDTH/2, HEIGHT/2*) will appear at the lower-left corner of the `frame` instead of the center of the `frame`.

8. These methods draw a point at ($WIDTH/2$, $HEIGHT/2$). The coordinates are logical coordinates not directly related to the canvas' size. The *width* and *height* in *glOrtho()* are actual window size. It is the same as $WIDTH$ and $HEIGHT$ at the beginning, but if you reshape the window, they will be different, respectively. Therefore, if we reshape the window, the red point moves.

In summary, when *Frame* is instantiated, constructor *Jl_0_Point()* will create a drawable canvas, add event listener to it, attach the display to it, and get a handle to *gl* methods from it. *reshape()* will set up the display's logical coordinates in the window frame. *display()* will draw a point in the logical coordinates. When program starts, *main()* will be called, then *frame* instantiation, *Jl_0_Point()*, *setSize()*, *setVisible()*, *init()*, *reshape()*, and *display()*. *reshape()* and *display()* will be called again and again if the user changes the display area. You may not find it, but a red point appears in the window.

5.4 Drawing Randomly Generated Points

Jl_1_Point extends *Jl_0_Point*, so it inherits all the methods from *Jl_0_Point* that are not private. We can reuse the constructor and some of the methods.

/* draw randomly generated points */

```
import javax.media.opengl.*;
import com.sun.opengl.util.Animator;
import java.awt.event.*;

//built on Jl_0_Point class
public class Jl_1_Point extends Jl_0_Point {
    static Animator animator; // drive display() in a loop

    public Jl_1_Point() {

        // use super's constructor to initialize drawing

        //1. specify using only a single buffer
        capabilities.setDoubleBuffered(false);

        //2. add a listener for window closing
```

```
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        animator.stop(); // stop animation
        System.exit(0);
    }
});

// called one-time for OpenGL initialization
public void init(GLAutoDrawable drawable) {

    // specify a drawing color: red
    gl.glColor3f(1.0f, 0.0f, 0.0f);

    //3. clear the background to black
    gl.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    gl.glClear(GL.GL_COLOR_BUFFER_BIT);

    //4. drive the display() in a loop
    animator = new Animator(canvas);
    animator.start(); // start animator thread
}

// called for OpenGL rendering every reshape
public void display(GLAutoDrawable drawable) {
    //5. generate a random point
    double x = Math.random()*WIDTH;
    double y = Math.random()*HEIGHT;

    // specify to draw a point
    gl.glBegin(GL.GL_POINTS);
    gl.glVertex2d(x, y);
    gl.glEnd();
}

public static void main(String[] args) {
    J1_1_Point f = new J1_1_Point();

    //6. add a title on the frame
    f.setTitle("JOGL J1_1_Point");

    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}
```

1. *Jl_1_Point* is built on (extends) the super (previous) class, so we can reuse its methods. The super class's constructor is automatically called to initialize drawing and event handling. Here we specify using a single frame buffer. Frame buffer corresponds to the display, which will be discussed in the next section.
2. The drawing area corresponding to the display is the frame buffer. JOGL in default is using double-buffering for animation. Here we just need a single buffer that corresponds to the frame buffer.
3. In order to avoid window hanging, we add a listener for window closing and stop animation before exit. Animation (`animator`) will be discussed later.
4. `glClearColor()` specifies the background color. OpenGL is a state machine, which means that if we specify the color, unless we change it, it will always be the same. Therefore, whenever we call `glClearColor()`, the background will be black unless we call `glClearColor()` to set it differently.
5. Object `animator` is attached to `canvas` to drive its `display()` method in a loop. When `animator` is started, it will generate a thread to call `display` repetitively. A thread is a process or task that runs with current program concurrently. Java is a multi-threaded programming language that allows starting multiple threads. `animator` is stopped before window closing.
6. A random point is generated. Because `animator` will run `display()` again and again in its thread, randomly generated points are displayed.

In summary, the super class' constructor, which is called implicitly, will create a drawable `canvas`, add event listener to it, and attach the display to it. `reshape()` will set up the display's logical coordinates in the window frame. `animator.start()` will call `display()` multiple times in a thread. `display()` will draw a point in logical coordinates. When program starts, `main()` will be called, then red points appear in the window.

5.5 Building an Executable JAR File

To facilitate sharing and deployment, we can generate an executable jar file for use with our JOGL applications as follows.

1. Set up a working directory to build your jar file.

2. Move all the necessary java class files into this directory, including all inheritance class files. For example, if you are to run program J1_1_Point, you should have your directory as follows:

```

2008-01-07 08:31 <DIR>      .
2008-01-07 08:31 <DIR>      ..
2008-01-07 09:19          1,766 J1_1_Point.class
2008-01-06 18:10          2,190 J1_0_Point.class
2008-01-07 09:19          736 J1_1_Point$1.class
          3 File(s)      4,692 bytes

```

3. Create a text file “manifest-info.txt” in the same directory that contains the following information with a carriage return at the last line:

```

Class-Path: gluegen-rt.jar jogl.jar
Main-Class: J1_1_Point

```

The Class-Path entry should include any jar files needed to run this program (jogl.jar and gluegen-rt.jar). When you run your program, you must make sure that these jar files are in the same directory. The Main-Class entry tells Java system which file contains your main method.

4. Execute the following in the same directory from the command line:

```
> "C:\myJDK\bin\jar" -cfm myexe.jar manifest-info.txt *.class
```

This will create your jar file with the specified manifest information and all of the *.class files in this directory.

5. Run your executable jar file:

You should now have your executable jar file in this directory (myexe.jar). To run the file, you need to put the library jar files (jogl.jar and gluegen-rt.jar) in the same directory. You may want to put all the dll files in the same directory as well if they are not installed in the system. Your directory will contain the following files as in our example:

```

2008-01-07 08:31 <DIR>      ..
2008-01-07 09:19          1,766 J1_1_Point.class
2008-01-06 18:10          2,190 J1_0_Point.class
2008-01-07 09:19          736 J1_1_Point$1.class
2008-01-07 09:46           61 manifest-info.txt
2008-01-07 09:46          3,419 myexe.jar
2007-04-22 02:00      1,065,888 jogl.jar

```

```
2007-04-22 02:00      17,829 gluegen-rt.jar
2007-04-22 02:00      20,480 gluegen-rt.dll
2007-04-22 02:00      315,392 jogl.dll
2007-04-22 02:00      20,480 jogl_awt.dll
      10 File(s)      1,448,241 bytes
```

Now you can either double-click on the jar file in Windows interface environment or execute it on a command line with:

```
> "C:\myJDK\bin\java" -jar myexe.jar
```

To get additional help or learn more on this topic you may visit the following place:

```
http://java.sun.com/docs/books/tutorial/deployment/jar/index.html
```

5.6 Review Questions

1. What is provided by the Animator class in JOGL?

- a. calling reshape()
- b. implementing interface functions
- c. calling display() repetitively
- d. transforming the objects

2. What are provided by the JOGL's GLUT class?

- a. bitmap and stroke font methods
- b. antialiasing
- c. calling reshape() or display()
- d. handling display area

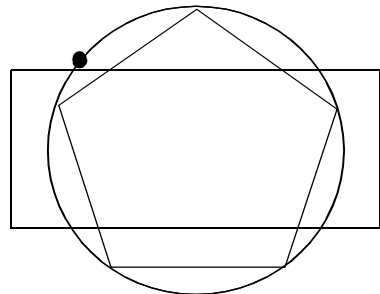
5.7 Programming Assignments

1. Draw a point that moves slowly along a circle. You may want to draw a circle first, and a point that moves on the circle with a different color.

2. Draw a point that bounces slowly in a square or circle.

3. Draw a star in a circle that rotates, as shown on the right. You can only use `glBegin(GL_POINTS)` to draw the star.

4. Write down "Bitmap" using Glut bitmap font function and "Stroke" using Glut stroke font function in the center of the display.



5. With the star rotating in the circle, implement the clipping of a window as shown on the right.

6. Implement an antialiasing line algorithm that works with the background that has a texture. The method is to blend the background color with the foreground color. You can get the current pixel color in the frame buffer using `glGet()` with `GL_CURRENT_RASTER_COLOR`.

7. Implement a triangle filling algorithm for `J1_3_Triangle` class that draws a randomly generated triangle. Here you can only use `glBegin(GL_POINTS)` to draw the triangle.

8. Draw (and animate) the star with antialiasing and clipping. Add a filled circle inside the star using the subdivision method discussed in this chapter. You should use your own triangle filling algorithm. Also, clipping can be tricky done by checking the point to be drawn against the clipping window.

