CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 155 – Advanced Computer Graphics
Fall 2018
Dr. Muyan

Assignment #1: OpenGL and JOGL

Due Date: Thursday, September 27th (11:59 PM)

Overview

The objective of this assignment is to insure that you are sufficiently familiar with the basic structure of applications written in JOGL, OpenGL, and GLSL (including vertex and fragment shaders), to build a system that includes both a simple GUI and graphical output. We will be using these techniques all semester, so it is important to master this framework.

The assignment is to write a program which uses JOGL and GLSL to draw a triangle that moves around the screen in various ways, with user input controls (keyboard, mouse, buttons, etc).

Program Structure

Your program will define a class that extends **`JFrame`.** It will attach a **`GLCanvas`** and some buttons to the **`JFrame`** (see the appendix for tips on adding buttons to **`JFrame`** in Java). A **`GLEventListener`** should be attached to **`GLCanvas`**.

This program will become the framework for future assignments in the class. Therefore, it is important to design it from the beginning with the idea of *flexibility* in mind. This means that your code should be organized into appropriate classes. For instance, you can use Model-View-Controller (MVC) architecture and command design pattern. Your "controller" which extends from **`JFrame`** would instantiate the model object, setup the GUI, instantiate and start the **`FPSAnimator`**, attach command objects to buttons and keys (see the appendix for tips on creating command objects, attaching them to buttons, and generating key bindings in Java), handle mouse wheel events (see the appendix for tips on handling mouse wheel events in Java), etc. Your "model" would hold your data (e.g., offset and color values) and related methods that manipulate this data (command objects, the mouse handling mechanism and other methods that need to update the data would invoke these methods). Your "view" would be a class that extends from **`GLCanvas`** and implements **`GLEventListener.`**

Shaders

You will need to build simple GLSL vertex and fragment shaders as described in the textbook. You do not need to write tesselation or geometry shaders. The shaders must specify a version of "430", and they must be read from files. See the book for an example of reading in shader code from files. The vertex shader should hardcode a simple triangle (as shown in the textbook) and receive some input. The fragment shader should simply output the colors it receives.

Your JOGL (Java) program will need to read in the shaders, compile them, attach them to a shader program, and then link the program. It will then use **`glUseProgram()`** and **`glDrawArrays()`** to invoke the GLSL shaders in the pipeline.

Animation and Graphical User Interface

In addition to launching your shaders, the JOGL (Java) program will need to incorporate the **`FPSAnimator`** that regularly invokes the **`display()`** method of your **`GLCanvas`** object. This in turn will invoke the **`display()`** method in your **`GLEventListener`**, in which the shaders are invoked. It is here that your Java program can send additional information to the vertex shader, in the form of uniform variables, that the shader can use to modify the location of the triangle's vertices, or its color(s).

Your GUI must include at least the following controls:

- a button that causes the triangle to move in a circle around the **`GLCanvas`**.
- a button that causes the triangle to move up and down, vertically.
- a key binding that toggles the triangle between a single solid color, and a gradient of three colors when user presses the 'c' key.
- mouse wheel control increases and decreases the size of the triangle.

You do not need to use transformation matrices to move the triangle. You can use the simple "offset" technique shown in Chapter 2 of the textbook. We will use transforms in Assignment #2.

Error Handling and Getting Version Information

Your program must implement error handling to catch GLSL compile and link errors. You can either use related built-in functions available in graphicslib3D or include those functions (as listed in CodedVersion of Program 2.3 in the book code samples) in your Java code, which is the recommended option.

It must also display the current OpenGL, JOGL, and Java versions on the console, when it first starts.

Getting version information on the OpenGL version at runtime, can be done in a manner very similar to that shown in the supplemental notes to Chapter 2 of the book. Instead of using **`glGetFloatv()`** you can use **`glGetString()`**. Send it one of the various built-in names: **`GL_VERSION, GL_VENDOR, GL_RENDERER`**, etc. Details can be found in the online OpenGL reference pages.

Since the JOGL classes are all defined in Java, you can use Java's "reflection" capabilities (which make it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time) to get the currently running version of JOGL. One way to do this is to use the **`Package.getPackage()`** method, passing it the name of the JOGL package (**`com.jogamp.opengl`**) as a string. This returns a reference to the JOGL package, and you can then call the **`getImplementationVersion()`** function on this reference to get the version.

You can get the version information on the Java with **`System.getProperty("java.version")`**.

All of the above can simply be output to the command window using **`System.out.println()`**.

Java Packages

Your program code must be contained in a Java package named "**`a1`**" (lower case). Each source file must contain the statement **`package a1;`** at the beginning. If you use an IDE that puts your code in a different package (or the "default package"), you must either override the IDE settings or, when the final version is finished, modify the package statements and recompile everything from a command line. If you do your development directly from a command line, you must create all your source code in a subdirectory named "`a1`" and compile/execute it from the parent directory. Also, it is a requirement that the "main" class in your program be named "`Starter`".

Given that the program source code is in a folder named "`a1`", it must be possible to compile and run the program from a command prompt in `a1`'s parent directory by typing "**`javac a1/*.java`**" and "**`java a1.Starter`**", respectively.

Documentation

All code in assignments for this course must be *well-documented*, and you should follow standard Java coding conventions. Class names start with *uppercase* letters, package and variables start with *lowercase*, and names should use "CamelCase" ("**`MyClass`**", "**`myMethod`**", "**`myPackage`**").

Running under Microsoft Windows

Most Windows implementations try to take advantage of "Direct3D (d3d)" to accelerate OpenGL graphics. This can cause output which shows up *blank,* or sometimes flickers. The solution is to tell Java not to do this, by adding the following argument to the "java" command:

```
java –Dsun.java2d.d3d=false a1.Starter
```

Deliverables

Submit to Canvas **TWO files (zip file and txt file) SEPERATELY** (i.e., do NOT place the txt file inside the zip file). The *ZIP file* should be named as *YourLastName-YourFirstName-a#.zip* (e.g., Doe-Jane-a1.zip) and should contain: (1) Java source code files, (2) shader files, (3) compiled (.class) files, and (4) a screen capture of your program running. The submitted files must be organized in the proper hierarchy in the ZIP file; that is, the .java and .class files must be contained within a subdirectory named "**`a1`**". The *TEXT file* **(i.e., not a pdf, doc etc.)** should be named as *readme.txt* and should list: **the lab and the lab machine you have used to test your program** (it is a requirement that your program runs properly on at least one machine in the labs). You may also include additional information you want to share with the grader in this text file. You will receive the grader comments on your text file when grades are posted.

Appendix – Java Notes

Adding Buttons

You can create buttons and add it to north section of **`JFrame`** that has **`BorderLayout`** as follows:

```
import javax.swing.JPanel;
import javax.swing.JButton;
…
JPanel topPanel = new JPanel();
this.add(topPanel,BorderLayout.NORTH);
JButton myButton = new JButton ("Button Label");
topPanel.add(myButton);
```

In Codename One (CN1), instead of **`Swing,`** we were using **`UI`** framework to build our GUI. **`JFrame`**, **`JPanel,`** and **`JButton`** correspond to **`Form`**, **`Container,`** and **`Button`**, respectively, in UI.

Creating Commands and Attaching them to Buttons

You can create a command class (e.g., **`CustomCommand`**) by extending from **`AbstractAction.`**

`AbstractAction` corresponds to **`Command`** in CN1. However, the code that goes into the command class is the same in both CN1 and Java.

You can create and attach the command objects as follows:

```
CustomCommand myCommand = new CustomCommand();
myButton.setAction(myCommand);
```

`setAction()` corresponds to **`setCommand()`** in CN1.

Generating Key Bindings

Every **`JComponent`** has a set of Key-Action maps ("bindings"). There are two kinds of maps: input maps (hold **`KeyStroke`** objects), action maps (hold command objects). Utilizing these concepts, you can attach command objects to the keys by adding the following code to your **`JFrame`**:

```
// get the content pane of the JFrame (this)
JComponent contentPane = (JComponent) this.getContentPane();
// get the "focus is in the window" input map for the content pane
int mapName = JComponent.WHEN_IN_FOCUSED_WINDOW;
InputMap imap = contentPane.getInputMap(mapName);
// create a keystroke object to represent the "c" key
KeyStroke cKey = KeyStroke.getKeyStroke('c');
// put the "cKey" keystroke object into the content pane’s "when focus is
// in the window" input map under the identifier name "color“
imap.put(cKey, "color");
// get the action map for the content pane
ActionMap amap = contentPane.getActionMap();
// put the "myCommand" command object into the content pane's action map
amap.put("color", myCommand);
//have the JFrame request keyboard focus
this.requestFocus();
```

The above code corresponds to **`addKeyListener('c', myCommand)`** in CN1.

Handling Mouse Wheel Events

Mouse wheel events are generated when wheel is rotated in a GUI component. We can handle these events by implementing **`MouseWheelListener`** interface:

```
public interface MouseWheelListener {
      public void mouseWheelMoved (MouseWheelEvent e);
}
```

You can make your **`JFrame`** a "self-listener" by making **`JFrame`** implement this interface and add itself as a listener for mouse wheel events generated on itself as follows:

```
this.addMouseWheelListener(this);
```

In **`mouseWheelMoved()`** method, you can determine the amount of wheel movement by calling **`getWheelRotation()`** on the **`MouseWheelEvent`** object which is passed as a parameter to the method.