CMSC427 Fall 2017
OpenGL Notes A: Starting with JOGL and OpenGL

Objectives of notes
- Show how to program full OpenGL in JOGL
- Start on shader programing

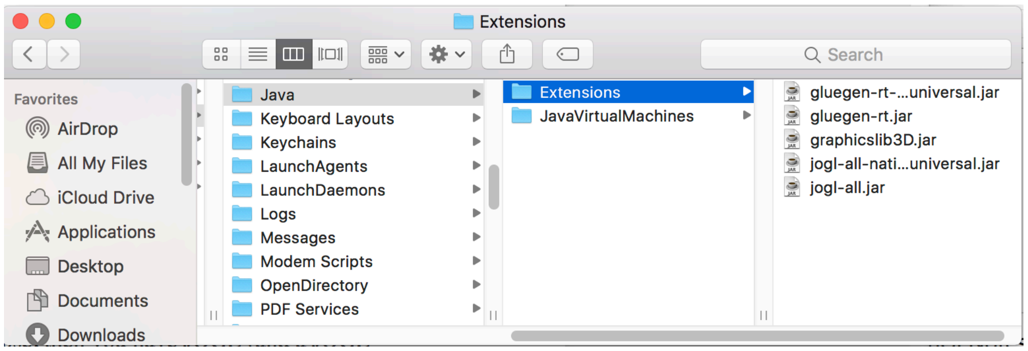
Readings
Computer Graphics Programming in OpenGL with Java, Gordon and Clevenger 2017
Chapters 2, 4
OpenGL SuperBible, Sellers et al, 2017
Chapters 2, 3, 5 and 6 (although 5 and 6 have more detail than needed now.)

Note: both books have chapters on math required for graphics (Chapter 3 in Gordon, Chapter 4 in the SuperBible up to page 89) which reviews much of the math we're done so far. These chapters may help you review this material.

Step 1: Configuring and compiling
On the class web page is a new file CMSC427OpenGLCode.zip . Download and unzip that. In the folder are OpenGL/JOGL programs from Chapter 2 and 4 of the Gordon textbook.

There's two pdfs in the zip file on how to install JOGL and graphicsLib3D. If you have already done the Microlab to get an OpenGL program running in Eclipse, then you have JOGL (and a JOGL project). Assuming you've done this, you can add these examples to Eclipse. You can also add the file graphics3Dlib.jar to the JOGL project you have in Eclipse, add that to the project libraries dependencies, and then have

At least on a Mac is also easy to add JOGL jar and graphics3Dlib.jar files to the /Library/Java/Extensions folder and set the CLASSPATH variable. Then you can compile and run these files with javac and java easily. My folder looks like this:



I've modified most of the examples to run on a Mac using these instructions:

http://athena.ecs.csus.edu/~gordonvs/errataMac.html

If this causes problems on a window machine you can reverse the changes by looking the comments in each file – the original "myCanvas=new GLCanvas()" line is still there, just commented out. Key is knowing your OpenGL version and making sure the code is set to that version.

Step 2: Creating a GLCanvas

The first example (Gordon 2.1) shows how to create a window in Java and add a GLCanvas to the window. A GLCanvas is a region of screen into which we can render OpenGL. The JFrame is the window and the GLCanvas is the drawable region inside of it. We'll use this code for future programs.

```
public class Code extends JFrame implements GLEventListener
{     private GLCanvas myCanvas;

      public Code()
      {     setTitle("Chapter2 - program1");
            setSize(600,400);
            setLocation(200,200);
            // Added for Mac
            GLProfile glp = GLProfile.getMaxProgrammableCore(true);
            GLCapabilities caps = new GLCapabilities(glp);
            myCanvas = new GLCanvas(caps);
            // Replacing this line
            //myCanvas = new GLCanvas();
            myCanvas.addGLEventListener(this);
            getContentPane().add(myCanvas);
            setVisible(true);
      }
```

This example doesn't do much, only clearing the window to a fixed color. But it show the use of a GL4 object for making OpenGL calls. If your system doesn't support GL4, you can use GL3 instead.

```
      public void display(GLAutoDrawable drawable)
      {     GL4 gl = (GL4) GLContext.getCurrentGL();

            float bkg[] = { 1.0f, 1.0f, 0.0f, 1.0f };
            FloatBuffer bkgBuffer = Buffers.newDirectFloatBuffer(bkg);
            gl.glClearBufferfv(GL_COLOR, 0, bkgBuffer);
            System.out.println("display");
      }
```

The OpenGL command glClearBufferfv() illustrates two things. First, OpenGL commands want Java NIO buffers, not basic arrays, which is why we promote bkg to a buffer. And OpenGL commands come in different flavors: glClearBufferfv() has fv at the end to show it takes a float vector (or buffer). You can have glClearBufferiv() for int buffer. OpenGL is based on C so functions are not properly overloaded.

If you want a proper Java application with buttons and other controls you can use a JPanel instead of a JFrame to host the GLCanvas.

Step 3: Compiling shaders

The second example (Gordon 2.2) uses shaders. A full OpenGL program uses these to determine the processing of vertices and fragments. You must have a vertex and a fragment shader. There's two ways to include them: as strings in your program:

```
String vshaderSource[] =
{     "#version 410    \n",
      "void main(void) \n",
      "{ gl_Position = vec4(0.0, 0.0, 0.0, 1.0); } \n"
};

String fshaderSource[] =
{     "#version 410    \n",
      "out vec4 color; \n",
      "void main(void) \n",
      "{ color = vec4(0.0, 0.0, 1.0, 1.0); } \n"
};
```

Or as external files you read into your program (here with the help of a utility function from graphcis3Dlib.jar.)

```
String vshaderSource[] = util.readShaderSource("code/vert.shader");
String fshaderSource[] = util.readShaderSource("code/frag.shader");
int lengths[];
```

Either way you read in the shaders you have to compile and link them as follows:

```
int vShader = gl.glCreateShader(GL_VERTEX_SHADER);
int fShader = gl.glCreateShader(GL_FRAGMENT_SHADER);

gl.glShaderSource(vShader, vshaderSource.length, vshaderSource, null, 0);
gl.glCompileShader(vShader);

gl.glShaderSource(fShader, fshaderSource.length, fshaderSource, null, 0);
gl.glCompileShader(fShader);

int vfprogram = gl.glCreateProgram();
gl.glAttachShader(vfprogram, vShader);
gl.glAttachShader(vfprogram, fShader);
gl.glLinkProgram(vfprogram);

gl.glDeleteShader(vShader);
gl.glDeleteShader(fShader);
return vfprogram;
```

Compiled shaders should return a small positive integer – if you get -1 there has been an error. The final result is the shader program which you use to render your scenes.

The examples in Gordon 2.3 show how to more thoroughly check for shader compilation errors.

Step 4: Reading shaders
We're still on Gordon example 2.2. Shaders for OpenGL are typically written in GLSL (GL shader language) which is based on C. We'll over more through the semester. What we need to know now is the following:

• We need two, the vertex shader which processes each vertex in the pipeline, and the fragment shader, which processes each pixel belonging to a fragment.
• Each shader starts with a version number. Mac OS only runs up to version 4.1 at the moment, so if a shader has 430 (as Gordon does) it won't run on a Mac. The version number is important.
• Each shader has a main routine and can have other functions.
• GLSL has new types, such as vec2, vec3 and vec4 for small arrays.
• There are a few fixed variables, including gl_position, which are built-in.

Vert.shader: This shader sets a fixed location for every vertex.
```
#version 410
void main(void)
{
      gl_Position = vec4(0.0, 0.0, 0.5, 1.0);
}
```

Frag.shader: This shader sets a fixed color for every pixel.
```
#version 410
out vec4 color;
void main(void)
{
      color = vec4(0.0, 0.0, 1.0, 1.0);
}
```

These shader programs are ultimately invoked by the following two lines in display():

```
            gl.glUseProgram(rendering_program);
            gl.glDrawArrays(GL_POINTS,0,1);
```

This says to use the compiled and linked shader programs, and to draw one vertex starting at vertex 0.

Step 5: Reading shaders, second example

In Gordon example 2.5 draws a triangle. There's two significant changes. First, the vertex shader now changes the point depending on the vertex id. The vertex id is a built-in predefined variable passed to the shader.

```
#version 410


void main(void)
{ if (gl_VertexID == 0)
      gl_Position = vec4( 0.25,-0.25, 0.0, 1.0);
  else if (gl_VertexID == 1)
      gl_Position = vec4(-0.25,-0.25, 0.0, 1.0);
  else
      gl_Position = vec4( 0.25, 0.25, 0.0, 1.0);
}
```

Second, the command to actually draw, glDrawArrays(), is updated to indicate three vertices are to be drawn as a triangle:

```
      public void display(GLAutoDrawable drawable)
      {     GL4 gl = (GL4) GLContext.getCurrentGL();
            gl.glUseProgram(rendering_program);
            gl.glDrawArrays(GL_TRIANGLES,0,3);
            System.out.println("display");
      }
```

Step 6: Animation
The animation example (Gordon 2.6) introduces Uniform shader variables. These are values that remain the same for all vertices and are passed in as discrete items. We use these for transforms (camera matrix, etc.)

To animate a JOGL program we include an animator object and start it – they can also be stopped and paused, and vary the framerate:

```
            FPSAnimator animator = new FPSAnimator(myCanvas, 30);
            animator.start();
```

Now the vertex shader has a new uniform variable "inc". Since it is declared uniform, it is expected not to vary from vertex to vertex.

The built-in variable "gl_VertexID" does varying, indexing into the vertices 0 to n, depending on how many are expected (that's given the the glDrawArrays() call).

```
#version 410

uniform float inc;

void main(void)
{
  if (gl_VertexID == 0)
      gl_Position = vec4( 0.25+inc,-0.25, 0.0, 1.0);
  else if (gl_VertexID == 1)
      gl_Position = vec4(-0.25+inc,-0.25, 0.0, 1.0);
  else
      gl_Position = vec4( 0.25+inc, 0.25, 0.0, 1.0);
}
```

To get the value of a uniform variable into a shader, one way is to query the shader for an integer offset, and them use that offset to pass in a value to the uniform variable (x is a float here):

```
      x += inc;
      int offset_loc = gl.glGetUniformLocation(rendering_program, "inc");
      gl.glProgramUniform1f(rendering_program, offset_loc, x);
```

Step 7: Passing in vertices
In this final example (Gordon 4.1a) we will have a relatively complete program with camera, perspective, color and a polygonal mesh (a box made of triangles.) We've covered all the concepts but not the details of OpenGL.

There's three key sections of the program that do specific computation for the scene.

First are lines 80-81 in display() that set the camera and cube locations:

```
            cameraX = 0.0f; cameraY = 0.0f; cameraZ = 8.0f;
            cubeLocX = 0.0f; cubeLocY = -2.0f; cubeLocZ = 0.0f;
```

No rotation so the camera matrix calculation is simplified.

Second are lines 86-98 in setUpVertices that record the vertex positions. Each triangle is given separately so vertices are repeated:

```
private void setupVertices()
      {     GL4 gl = (GL4) GLContext.getCurrentGL();
            float[] vertex_positions =
            {     -1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
                  1.0f, -1.0f, -1.0f, 1.0f,  1.0f, -1.0f, -1.0f,  1.0f, -1.0f,
                  1.0f, -1.0f, -1.0f, 1.0f, -1.0f,  1.0f, 1.0f,  1.0f, -1.0f,
```

And third is line 50 which sets the perspective transformation:

```
Matrix3D pMat = perspective(60.0f, aspect, 0.1f, 1000.0f);
```

This last example is long and complex – we'll review it in class.