OpenGL

OpenGL is an API designed to allow the computer to interface with the graphics hardware in the computer. It’s not directly available for Java, so instead, LWJGL is used, which also provides GLFW, a library for creating a window and processing events.

There are several parts of OpenGL this program uses:

Buffer Objects

Vertex Array Objects

Shaders, Uniforms, and Vertex Attributes

Buffer Objects

The primary way to hold data in OpenGL is the buffer object. They’re basically the equivalent of the array, but stored in graphics memory. They are created with the method GL15.glGenBuffers() which returns an integer id, which is used throughout the rest of OpenGL to refer to it. The next step in using buffer objects is the method GL15.glBindBuffer(target, buffer). buffer is the id provided by GL15.glGenBuffers(), and target is a symbolic enum representing the type of buffer storage. There are many types of buffers, but only two are used in this application: GL\_ARRAY\_BUFFER, and GL\_ELEMENT\_ARRAY\_BUFFER.

GL\_ARRAY\_BUFFER is used for storage of generic vertex data, depending on the use, i.e., vertex positions, vertex normals, colors, and texture coordinates. GL\_ELEMENT\_ARRAY\_BUFFER is used to store indexes into a vertex buffer, allowing for the reuse of vertexes. To upload data, call GL15.glBufferData(target, buffer, usage) where target is the target used in glBindBuffer, usage is an enum telling OpenGL how often you plan to update the data, (this allows OpenGL to optimize how the data is stored properly), and buffer is one the \*Buffer classes, where \* is one of the primitive types, i.e., FloatBuffer or ByteBuffer. These classes are like an array, but are stored directly in native memory, instead of being managed by Java’s memory management system. This makes the process of passing it to the graphics hardware faster. Now that the data has been uploaded to the buffers, the next step is telling OpenGL how to use it, with Vertex Array Objects.

Vertex Array Objects

Vertex Array Objects store a format for the data in the GL\_ARRAY\_BUFFER. This format is usually specified with GL20.glVertexAttribPointer(index, size, type, normalized, stride, offset). index refers to an attribute index, which aren’t defined in code, but defined in shaders, which will be explained later. size allows you to specify things like vectors, which have 3 or 4 components per attribute, by setting size to the number of components. type is an enum that specifies the type of the specified data. The type used in this application is GL\_FLOAT. normalized is a boolean that tells OpenGL whether or not to convert the data to a range of 0…1. stride is the number of bytes in between instances of the same attribute, and offset is the number of bytes from the beginning of the data until the first instance of the attribute. glVertexAttribPointer has to be called for each attribute, like position, normal, colors, and texture coordinates. This can result in a lot of function calls during setup, so the Vertex Array Object saves the calls you make to glVertexAttribPointer and replays them when you activate it. Vertex Array Objects are created and used in the same way as Buffer Objects, using GL30.glGenVertexArrays() and GL30.glBindVertexArray(). Bind a Vertex Array Object, and call glVertexAttribPointer as needed. When the Vertex Array Object is rebound, the earlier calls will be automatically called again. Now that OpenGL has the data, and knows its format, it’s time to actually get that on the screen.

Shaders, Uniforms, and Vertex Attributes

The actual calculation work of rendering is done by shaders. Shaders are small programs written in GLSL, a language made specifically for OpenGL programming. There are two shader stages: (there are actually 4-ish, but only two are used for this program) vertex shading, and fragment shading. Vertex shaders are executed first, followed by the fragment shaders. Vertex shaders get the extension .vert, and fragment shaders get the extension .frag. Shaders are loaded from text files and compiled at runtime. There are two ways to get data into a shader, and one way to get it out. The two input methods are input variables and uniforms. Input variables are defined like this: layout(location = X) in \*type\* \*name\*, where type is similar to Java primitive types, like float, int, with some mathematical types, like mat4 and vec3, and name is a user defined name. X refers to an attribute index, and the type and index must match the ones used earlier in glVertexAttribPointer. Uniforms are declared with uniform \*type\* \*name\*, where type and names mean the same thing. However, they must be uploaded more specifically, using glGetUniformLocation with the uniform’s name first, and passing that value into glUniformXXX where XXX is the type of uniform to upload. The one way to get data out of a shader is the output variable, described similar to an input variable, but with out instead of in. Another important thing to note with shaders is that output variables in one stage get fed into input variables with the same name in type in the next stage. The vertex shader’s job is to take in the vertex position, normal, color, and texture coordinate, and transform them if needed, and pass them into the fragment shader. It also calculates the proper per vertex positions, and writes into the built in variable gl\_Position, which is later read by OpenGL. The fragment shader calculates lighting using the passed through variables, and writes the final color of the fragment into a single out variable representing a RGBA color. The colors are then taken by OpenGL and written to the screen, creating a frame.