CS130 LAB: 4 – Part 1: Introduction to OpenGL

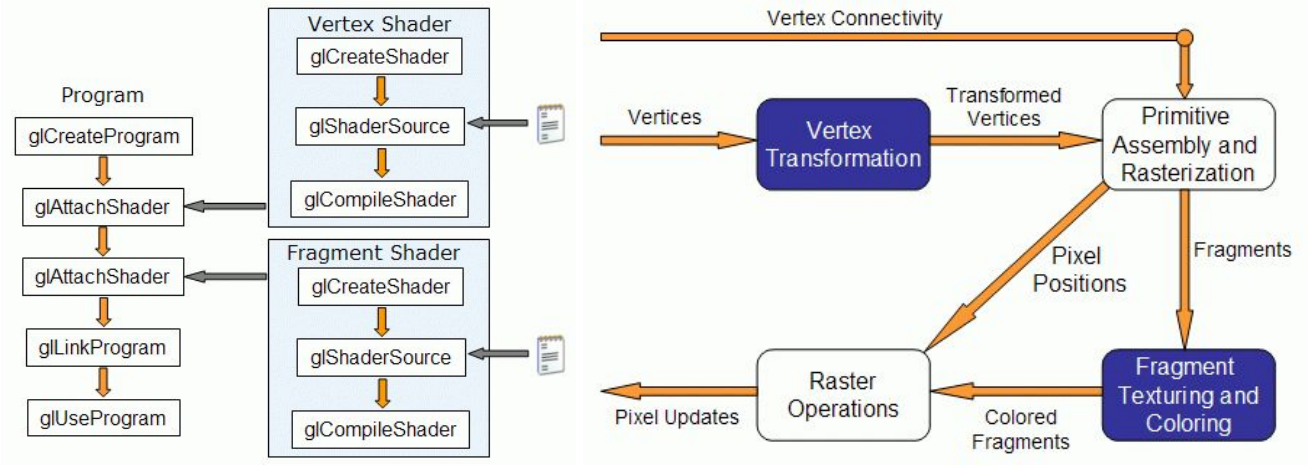
Open Graphics Library (OpenGL) is a cross-platform API for fast rendering of 2D and 3D graphics. OpenGL typically runs on a graphics processing unit (GPU) and it is optimized to render multiple images per second. For this reason, OpenGL is often used in game engines and other applications that require interactivity with the user.

The goal of this lab is to get you started with OpenGL by implementing Phong's illumination model into special OpenGL programs called *shaders*.

The process is summarized as follows:

1. **OpenGL program is written in C/C++** and consists of setting up the scene (camera position, objects, lights, among others).
2. The OpenGL program is also responsible for **reading a text file** with shader code, **compile** it and **send it to the GPU** for execution.
3. The language used in the shader program is very similar to C and is called ***OpenGL shader language* (GLSL)**
4. The **shader typically runs on the GPU** and the shader **determines the position and color of vertices**. Vertices are the points that constitute a geometry. For instance, a cube has 8 vertices.
5. There are two types of shaders: **vertex** and **fragment**.
    a. The **vertex shader receives vertices and applies transformations** to these vertices (scale, translation, rotation, among others).
    b. **The fragment shader receives *fragments* and determines the color** of that fragment. Fragments are transformed vertices outputted by the vertex shader after rasterization.

The left diagram below depicts the process of loading the vertex and fragment shaders in the OpenGL C/C++ code. The right diagram depicts the vertex and fragment shaders.



Taken from http://www.lighthouse3d.com/tutorials/glsl-12-tutorial/pipeline-overview/.

1. Consider the OpenGL code diagram depicted in the last page. Describe briefly with your own words each one of the following functions. Look at the OpenGL documentation for reference.

Llink: https://www.khronos.org/registry/OpenGL-Refpages/gl4/
Google: “opengl 4 references”

`glCreateShader`

`input:` An enum of shader type

`output:` A GLuint which references the shader just created

`glShaderSource`

`input:` The shader created by glCreateShader
The number of strings in the array
The shader program in c style string
Length of the string, default is null terminated

`glCompileShader`

`input:` The shader created by glCreateShader

`glCreateProgram:`

`output:` A reference to the program it just created

`glAttachShader`

`input:` The shader created by glCreateShader
The program created by glCreateProgram

`glLinkProgram`

`input:` The program created by glCreateProgram

`glUseProgram`

`input:` The program created by glCreateProgram

2. Read the comments and order the lines of code in correct order for loading shaders.
Fill in the blanks afterwards.

```
A.glCompileShader(_____fragment_id_____); // compile fragment shader
B.glAttachShader(____program____,____vertex_id____); //attach vertex shader to program
C.GLuint vertex_id = glCreateShader(__GL_VERTEX_SHADER__); // create vertex shader
D.glCompileShader(_____vertex_id_____); //compile vertex shader
E.glAttachShader(____program____,fragment_id); //attach program shader to program
G.glShaderSource(_____vertex_id_____,1,&vertex_shader_file,NULL); //source vertex shader
F.glLinkProgram(____program____); //link program
H.GLuint fragment_id=glCreateShader(_GL_FRAGMENT_SHADER_); // create fragment shader
I.glShaderSource(____fragment_id____,1,&fragment_shader_file,NULL); //source fragment shader
J.GLuint program = glCreateProgram();
```

Ordering: C G D H I A J B E F
(The answer may vary.)

3. Consider the following GLSL vertex (left) and fragment (right) GLSL codes.

```
void main() {
  gl_Position = gl_ProjectionMatrix
      * gl_ModelViewMatrix
      * gl_Vertex;
  gl_FrontColor =
      vec4(0, 1, 0, 1);
}
```

```
vec4 light_color = vec4(1, 0, 0, 1);

void main() {
  gl_FragColor = gl_FrontColor;
}
```

The vertex shader receives a `vec4` **`gl_Vertex`** and returns a `vec4` **`gl_Position`**. `gl_ProjectionMatrix` and `gl_ModelViewMatrix` are transformation matrices given by OpenGL.

The fragment shader receives **`gl_FrontColor`** from the vertex shader and returns the color of the fragment as **`gl_FragColor`**.

3.1. What is the output color of the fragment shader?

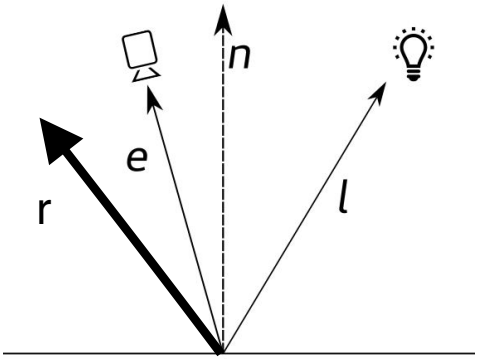
`gl_FragColor = (__0__, __1__, __0__, __1__)`

3.2. Consider an object with color green represented by the RGB color vector (0, 1, 0) and a blue light source with color (0, 0, 1). If we illuminate the object with the light, what is the output color?

(0, 1, 1)

CS130 LAB: 4 – Part 2: Phong model

Write the equations for the the Phong model components. Draw any missing vectors in the figure below.



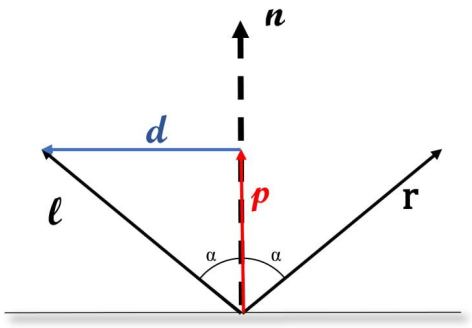
Ambient: RaLa

Diffuse: RdLd*max(0, dot(n, l.normalized()))

Specular:

RsLs*pow(max(dot(r, e), 0), specular_power)

In the figure below, vector r is the reflection of vector l from the surface, and n vector is the unit-length normal of the surface.



Write the reflection vector r in terms of n and l, following the steps below:

Step 1: Formulate vector p, which is the projection of l on n, in terms of l and n.

p = (l.n).n

Step 2: Formulate vector d, in terms of l and p

d = l - p

Step 3: Write vector r in terms of d, p and l (you do not have to use all of them)

r = p - d

Step 4: Substitute p and d, with your results from steps 1 and 2, and write r in terms of l and n only.

r = (l.n).n - l + (l.n).n = -l + 2(l.n).n

In order to write Phong's model in your shader, you can use:

```
struct gl_LightSourceParameters {
     vec4 ambient;
     vec4 diffuse;
     vec4 specular;
     vec4 position;
};
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
struct gl_LightModelParameters {
     vec4 ambient;
};
uniform gl_LightModelParameters gl_LightModel;

struct gl_MaterialParameters {
     vec4 ambient;
     vec4 diffuse;
     vec4 specular;
     float shininess; // this is the exponent of the specular
component
};
uniform gl_MaterialParameters gl_FrontMaterial;
```

You may also use the following functions: `max(_,_); dot(_,_); normalize(_);`
You can assume the camera position is at the origin, i.e., at coordinates (0, 0, 0).