CS130 - LAB - Introduction to OpenGL

Name: Patrick Dang SID: 862091714

Introduction

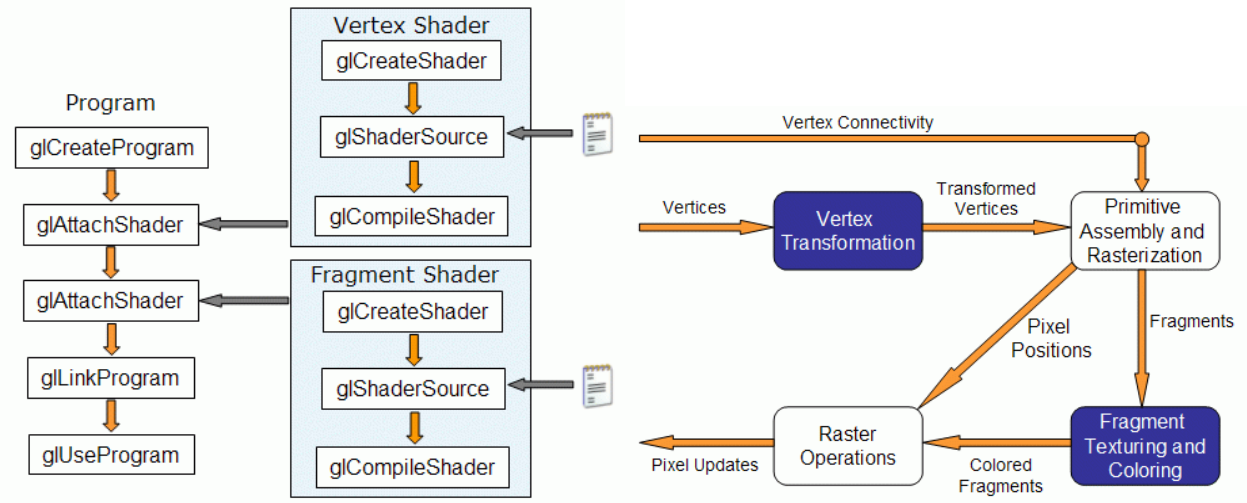
Open Graphics Library (OpenGL) is a cross-platform API for fast rendering of 2D and 3D graphics. OpenGL typically runs on a graphics processing unit (GPU) and it is optimized to render multiple images per second. For this reason, OpenGL is often used in game engines and other applications that require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

- An OpenGL program is written in C/C++ and consists of setting up the scene (camera position, objects, lights, among others).
- The OpenGL program is also responsible for reading a text file with shader code, compiling it and sending it to the GPU for execution.
- The language used in the shader program is very similar to C and is called OpenGL shader language (GLSL)
- The shader typically runs on the GPU and the shader determines the position and color of vertices. Vertices are the points that constitute geometry. For instance, a cube has 8 vertices.
- Here, we care about two types of shaders: vertex and fragment.
- The vertex shader receives vertices and applies transformations to these vertices (scale, translation, rotation, among others).
- 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 above. Describe briefly with your own words each one of the following functions. Look at the OpenGL documentation for reference. Link: https://www.khronos.org/registry/OpenGL-Refpages/gl4/

`glCreateShader`
input: Type of shader to be created
output: An empty shader object that can be referenced

`glShaderSource`
input 1: The handle of the shader object that will be replaced
input 2: The number of elements in inputs 3 and 4 (string and length arrays)
input 3: The array of pointers to strings will be loaded into the shader specified in input 1
input 4: The array of input 3's string lengths

`glCompileShader`
input: The shader object to be compiled

`glCreateProgram`
output: Creates an empty program object that can be referenced. A program object is an object that shaders can be attached to. This allows the shaders to interact with each other and the program itself.

`glAttachShader`
input 1: The program the shader is being attached to
input 2: The shader that is going to be attached

`glLinkProgram`

input: The handle of the program object that will be linked

`glUseProgram`
input: The handle of the program object that is being rendered currently

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

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

F. glShaderSource(vertex_id, 1, &vertex_shader_file, NULL); // source vertex shader

G. 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, F, D, H, I, A, J, B, E, G

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

// fragment shader example
vec4 light_color = vec4(1, 1, 0, 1);
void main()
{
  gl_FragColor = light_color*gl_FrontColor;
}
```

This 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)`

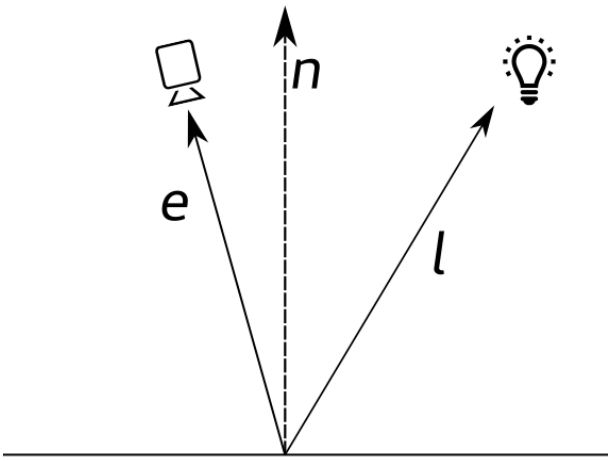
3.2. Consider an object with color green represented by the RGB color vector (0, 1, 0) and a blue light source with color (0, 0, 1). If we illuminate the object with the light, what is the output color? (Hint: Consider the fragment shader code above.)

`Output = (0, 0, 0, 1)`

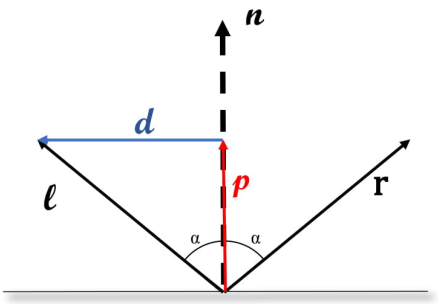
Part 2: Phong Lighting Model Review

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

Ambient: $L_a R_a$
Diffuse: $L_d R_d \max(n \cdot l, 0)$
Specular: $L_s R_s \max((2(n \cdot l) * n - l) \cdot e, 0)^{\alpha}$



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:

1. Formulate vector p, which is the projection of l on n, in terms of l and n. $p = (n \cdot l) * n$.
2. Formulate vector d, in terms of l and p. $d = l - p$.
3. Write vector r in terms of d, p and l (you do not have to use all of them). $r = p - d$.
4. Substitute p and d, with your results from steps 1 and 2, and write r in terms of l and n only. $r = 2(n \cdot l) * n - l$.

In order to write Phong's model in your shader, you can use (these structures and variables are already defined elsewhere in the program):

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

```
  float shininess; // this is the exponent of the specular component
};
uniform gl_MaterialParameters gl_FrontMaterial; // accessible variable
```

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).

Part 3: Notes on Assignment 1 - Checkpoint 2

If you implemented plane intersection, then you have test 04 working. The next steps are:

1. Phong shader
2. Shadows

Starting with the Phong shader. (Implement `Shade_Surface` in `phong_shader.cpp`). Recall Phong shader consists of 3 components: ambient, diffuse, specular. You will need to calculate each component and add them all to the color that is returned.

Ambient. Combination of three variables (you have access to all of them in `Shade_Surface`)

1. `world.ambient_color`
2. `world.ambient_intensity`
3. `color_ambient`

Diffuse. Is proportional to the cosine of the angle between the normal (n) and the vector from the intersection point to the light source (l). This term is the intensity of the diffuse component.

- The intersection point is calculated as the point in the ray with the earliest hit t. You can get any point on a ray using the function `ray.Point(t)`.
- You may need to calculate the intersection point in your `Cast_Ray` before passing it to the shader.
- Notice the normal should be pointing to outside of your object. If the nearest point is exiting the object, you may need to invert the normal so it is facing the right direction.
- Normalize the vectors when calculating the cosine using dot product.

- Check if the light source is behind the intersection point on the surface. In this case, the diffuse intensity is zero. You can check for this by taking $\max(l \cdot n, 0)$.
- You have access to `color_diffuse` in your Phong shader. This should be combined with the diffuse intensity.
- You will also need to compute the color of the light source and combine it in your diffuse component. In particular, the intensity of the light should decay proportional to the square distance between the intersection point and the light source.
- You can get the light color at the proper intensity by calling the function `Emitted_Light` passing the vector between the light source and the intersection point.

Specular. Proportional to the cosine of the angle between the reflected direction and the vector from the intersection point to the camera position (c).

- You can calculate the reflected direction using $r = (2 * (l \cdot n)n - l)$. Make sure l and r are normalized.
- The specular intensity is $\max(r \cdot c, 0)^{\alpha}$, where α is given to you as the `specular_power` variable.
- The final color is calculated similarly to the diffuse component by using the `light_color` with decay proportional to the square of the distance to the light source.

Shadows.

- In your Phong shader, check if shadows should be calculated by using the variable `world.enable_shadows`.
- If `world.enable_shadows` is true, then you should check if there is an object between your intersection point and the light source (You can use `Closest_Intersection` for this).
- If there is an object blocking all your light sources, then you should return only the ambient light component.