Implementing The Phong Reflection Model With OpenGL

Lorcan Purcell

September 7, 2025

1 Introduction

1.1 OpenGL

OpenGL is a graphics API used for rendering 2D and 3D vector based graphics. OpenGL offers very bare-bone assistance to the user and is utilized completely through function calls which are then implemented through graphics drivers to render graphics. This is both a strength and a weakness, as direct communication to the GPU allows for far more granular and low-level control, however, it also significantly complicates tasks compared to established game engines such as Unreal or Unity if one wishes to render something.

1.2 Phong Reflection Model

The Phong reflection model is a standard mathematical model that is used to show how light interacts with surfaces. It is a combination of three lighting techniques, which are ambient lighting, diffuse lighting, and specular lighting.

1.2.1 Ambient Lighting

Ambient light is constant lighting that gives an object some color (or otherwise it would not be visible). An example would be the moon, where even during the night it is still possible to make rough outlines and colors of objects. Furthermore, in real life objects are almost always illuminated simultaneously by multiple light sources, and thus will still have all faces illuminated to some degree.

Figure 1 shows an orange cube without ambient lighting (not visible) and a white cube with ambient lighting.

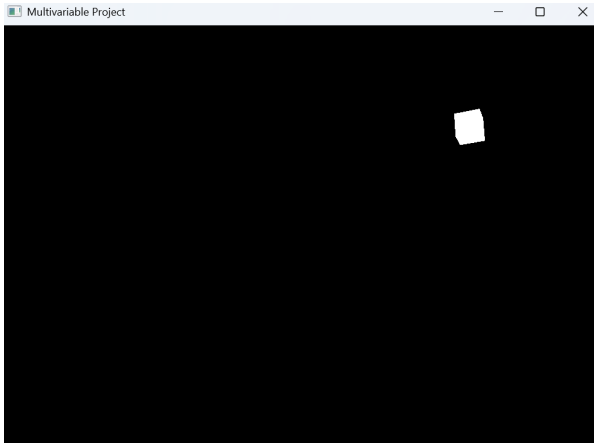


Figure 1: Cube without ambient lighting

Figure 2 shows what a surface looks like with ambient lighting but without directional lighting.

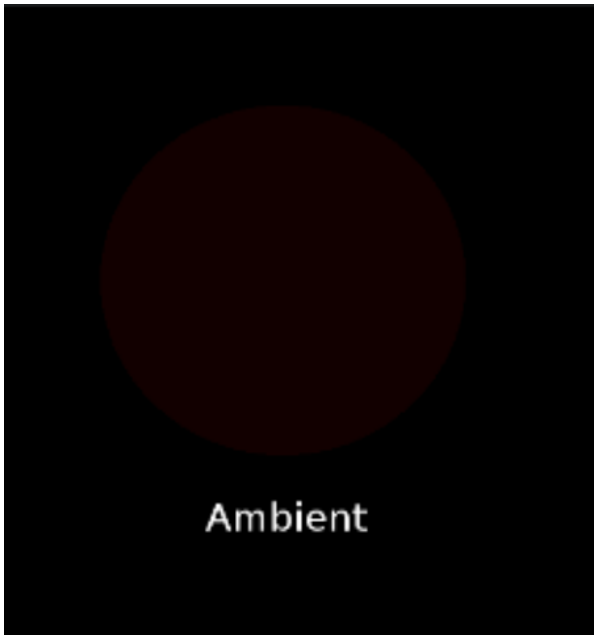


Figure 2: Sphere with ambient lighting (Source)

1.2.2 Diffuse Lighting

Diffuse lighting takes into account the directional impact a light source has on an object; when a light is shined on something, the parts most illuminated are brighter in color than the parts not illuminated.

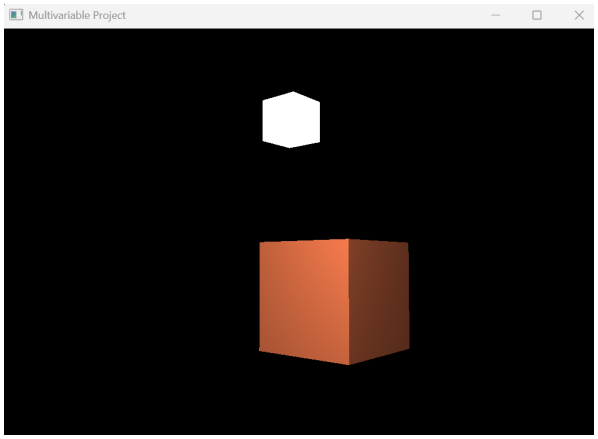


Figure 3: Orange cube with diffuse and ambient lighting

1.2.3 Specular Lighting

Lastly is specular, which deals with the bright spot that forms when objects are shiny. Interestingly, the 'hotspot' of light created by specular lighting will be the color of the light source and not the color of the object. When these three techniques are combined, one is left with Phong lighting which simulates real world lighting decently well.

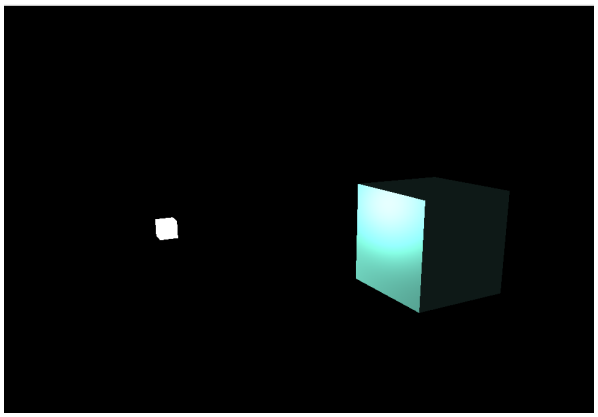


Figure 4: Blue cube with phong lighting (specular, ambient and diffuse lighting)

Here is a link to a live demo of my final project showcasing the Phong lighting model in real time

2 Math

2.1 Diffuse Lighting

To implement diffuse lighting, we have to find the normal vector at each fragment and then take the dot product between the normal vector and the light ray. The light ray is the vector found by subtracting the coordinates of the fragment from the coordinates of the light source. The greater the value of the dot product, the brighter the fragment will be as the light source will cast light more directly onto the fragment.

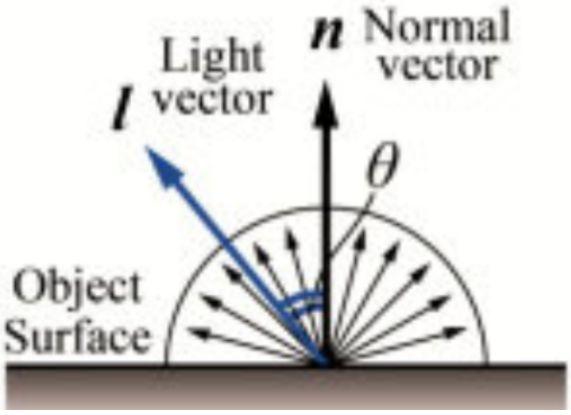


Figure 5: Diffuse lighting

As we only wish for the theta, the vectors will be normalized before taking the dot product. Furthermore, if the dot product is less than 0, that means the light ray is perpendicular with the fragment and will not illuminate it, thus negative results are ignored. Here is code section of my code used to calculate the diffused lighting from the fragment shader:

```
uniform vec3 objectColor;
uniform vec3 lightColor;
uniform vec3 lightPos // coordinate of light cube
in vec3 FragPos; //coordinate of fragment
in vec3 Normal; //normal vector of fragment
    // 1) calculate light ray vector, NB this is direction from surface to
        light; then normalize (vec3 = x,y,z)
```

```
    vec3 lightDir = normalize(lightPos - FragPos);
    // 2) Uses dot product to calculate how 'direct' the light ray is to the fragment.
    // dot(norm, lightDir) takes the dot product between normal vector and light ray,
    // the closer the value is to 1, the more direct the lighting
    // max(dot(norm, lightDir), 0.0); prevents negative values (no light from behind)
    float diff = max(dot(norm, lightDir), 0.0);
    // 3) Scale by the lights color (white here, so it just carries the intensity)
    vec3 diffuse = diff * lightColor;
    // 4) Combine constant ambient light with diffuse and then tine by the objects base // colour then tint by the light sourcess base color
    vec3 result = (ambient + diffuse) * objectColor;
    // 5) Output the final color (RGB) plus alpha = 1 for full opacity
    FragColor = vec4(result, 1.0);
```

The equation for diffuse lighting is as follows:

$$I_{\text{diffuse}} = k_d \, I_l \, \max\big(0, \; \hat{N} \cdot \hat{L}\big).$$

N represents the normal vector for the fragment
L represents the light ray vector

2.2 Specular

Specular lighting is calculated in a similar manner to diffuse, taking the normal vector of the fragment along with the light ray; however, this time the light ray is reflected across the normal vector, and then the dot product is taken between the reflected ray and the view direction.

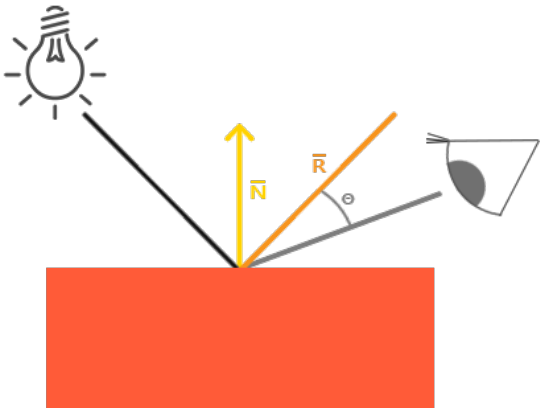


Figure 6: Specular lighting

Here is the code from the fragment shader to calculate specular lighting:

```
uniform vec3 objectColor;
uniform vec3 lightColor;
uniform vec3 lightPos;
// Coordinates of camera
uniform vec3 viewPos;
in vec3 Normal;
in vec3 FragPos;
    //Intensity of specular component. The closer to 1 the shinier the object
    float specularStrength = 0.5;
    vec3 lightDir = normalize(lightPos - FragPos);
```

```
    // 1) Find vector from camera to fragment
    vec3 viewDir = normalize(viewPos - FragPos);
    // 2) reflects the light ray across the normal vector. As the reflect
        function expects the vector to be pointing away from the light source,
         lightDir is negated, as before it was going from the fragment to the
        light source.
    vec3 reflectDir = reflect(-lightDir, norm);
    // 3) This is the actual specular calculation, mathematical representation
         can be seen below. First the dot product is taken between the view
        direction vector (which comes from the camera position) and the
        reflected light ray. Next, the greater value is chosen between the
        result of the dot product and 0.0f, as negative values would indicate
        the light source can not reach the fragment. Finally, the resulting
        quantity is raised to an n factor that indicates shininess. The
        greater the value, the more concentrated the specular highlight will
        be.
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    // 4) finally, the result of the specular highlight is combined with the
        diffuse and ambient values for the specific fragment and then
        multiplied by the objects colour, the final result is then outputted
        in a Vec4 (x,y,z,a), where a represents opacity.
    vec3 result = (ambient + diffuse + specular) * objectColor;
    FragColor = vec4(result, 1.0);
```

Here is the equation for specular lighting

$$k_{\text{spec}} = \|\mathbf{R}\| \, \|\mathbf{V}\| \, \cos^n \beta = \max(0, (\hat{A} \cdot \hat{B}))^n.$$

To elaborate on the previous equation:
R represents the reflected light ray across the fragment's normal vector
V represents the viewing vector
Beta represents the angle between the two vectors
max takes the greater value between 0 and the result of the dot product.

2.3 Combining Into Phong

Combining the equation for specular, diffuse and ambient we get the following equation representing the Phong reflection model

$$I_{\text{p}} = k_{\text{a}} i_{\text{a}} + \sum_{m \in \text{ lights}} (k_{\text{d}} (\hat{L}_m \cdot \hat{N}) i_{m,\text{d}} + k_{\text{s}} (\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,\text{s}}).$$

Here is a diagram showing each vector:

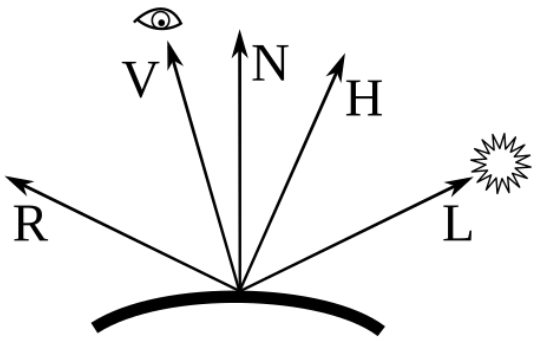


Figure 7: Vectors for calculating phong shading

All of the vectors in the figure have been discussed before with the exception of H, which is used in a variant of Phong lighting called Blinn-Phong.

While the equation may appear complicated, it is simply the compilation of the previous work. The right side inside the summation is the specular lighting dot product equation, raised to a value n.

$$k_s \tag{1}$$

is simply the SpecularStrength component from the code above that indicates the reflectiveness of the material.

$$I_{m,s} \tag{2}$$

and

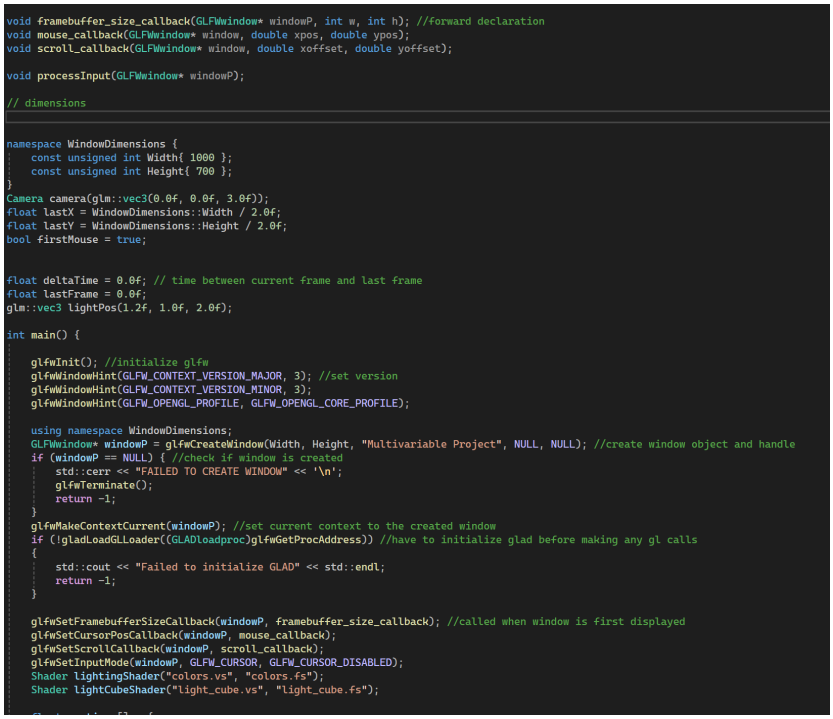
$$I_{m,d} \tag{3}$$

are constants which represent the color of the light source, in the code it is lightColor. The left side is the equation for diffuse lighting, where the dot product is taken between the light ray and the normal vector. The result is multiplied by the same constants as the specular result. The purpose of the summation sign is to account for the existence of multiple light sources, though due to time constraints the code contains only one light source, so m = 1. However, adding more to the code would not be difficult and all the same math would apply. Lastly, the left most side of the equation is simply the ambient contribution to the overall lighting.

3 High-level Code Review

The total code spanned over 600 lines across 7 different files and would take too long to describe in detail, however, the following sections will showcase some of the important parts and briefly describe their function. The code shown so far has been from fragment shaders only.

3.1 Initialization

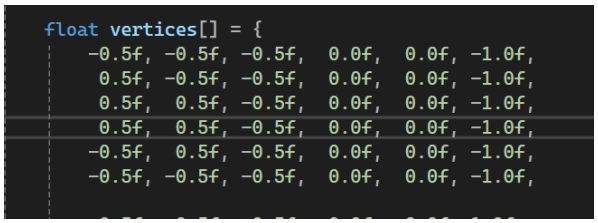


```
void framebuffer_size_callback(GLFWwindow* windowP, int w, int h); //forward declaration
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);

void processInput(GLFWwindow* windowP);

// dimensions


namespace WindowDimensions {
    const unsigned int Width{ 1000 };
    const unsigned int Height{ 700 };
}
Camera camera(glm::vec3(0.0f, 0.0f, 3.0f));
float lastX = WindowDimensions::Width / 2.0f;
float lastY = WindowDimensions::Height / 2.0f;
bool firstMouse = true;


float deltaTime = 0.0f; // time between current frame and last frame
float lastFrame = 0.0f;
glm::vec3 lightPos(1.2f, 1.0f, 2.0f);

int main() {

    glfwInit(); //initialize glfw
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); //set version
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    using namespace WindowDimensions;
    GLFWwindow* windowP = glfwCreateWindow(Width, Height, "Multivariable Project", NULL, NULL); //create window object and handle
    if (windowP == NULL) { //check if window is created
        std::cerr << "FAILED TO CREATE WINDOW" << '\n';
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(windowP); //set current context to the created window
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) //have to initialize glad before making any gl calls
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }

    glfwSetFramebufferSizeCallback(windowP, framebuffer_size_callback); //called when window is first displayed
    glfwSetCursorPosCallback(windowP, mouse_callback);
    glfwSetScrollCallback(windowP, scroll_callback);
    glfwSetInputMode(windowP, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
    Shader lightingShader("colors.vs", "colors.fs");
    Shader lightCubeShader("light_cube.vs", "light_cube.fs");
```

Figure 8: OpenGL set up code

This first part initialized the GLFW library, GLAD and sets OpenGL version to 3.3. which allows us to call modern OpenGL functions. It also sets the window size and defines a first-person-perspective camera.

3.2 Vertices



```
float vertices[] = {
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
     0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
     0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
     0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
    -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
```

Figure 9: Vertex and normal data for one cube face (repeated 5 more times)

Here is an array with all the vertices for the blue cube from the program. The first three floating point numbers are for the location of the vertices in 3D and the second 3 points show the normal vector for each vertices. Normals could have been calculated with a cross product formula, however in this case listing them explicitly was more convenient.

3.3 Render Loop

```
while (!glfwWindowShouldClose(windowP)) {
    using namespace WindowDimensions;
    processInput(windowP);
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    lightingShader.setVec3("lightPos", lightPos);
    float currentFrame = static_cast<float>(glfwGetTime());
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    // inside while(!glfwWindowShouldClose)
    float t = static_cast<float>(glfwGetTime());

    // 1) spiral in X?Y
    lightPos.x = sin(t) * radius;
    lightPos.y = cos(t) * radius;
    // 2) bob up/down in Z
    lightPos.z = baseHeight + sin(t * verticalSpeed) * heightAmplitude;

    lightingShader.use();

    lightingShader.setVec3("objectColor", 0.53f, 1.0f, 0.9f);
    lightingShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);
    lightingShader.setVec3("lightPos", lightPos);

    glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)Width / (float)Height, 0.1f, 100.0f);
    glm::mat4 view = camera.GetViewMatrix();
    lightingShader.setMat4("projection", projection);
    lightingShader.setMat4("view", view);
    lightingShader.setVec3("viewPos", camera.Position);


    // world transformation
    glm::mat4 model = glm::mat4(1.0f);
    lightingShader.setMat4("model", model);

    // render the cube
    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);


    // also draw the lamp object
    lightCubeShader.use();
    lightCubeShader.setMat4("projection", projection);
    lightCubeShader.setMat4("view", view);
    model = glm::mat4(1.0f);
    model = glm::translate(model, lightPos);
    model = glm::scale(model, glm::vec3(0.2f)); // a smaller cube
    lightCubeShader.setMat4("model", model);

    glBindVertexArray(lightCubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);




    glfwSwapBuffers(windowP);
    glfwPollEvents();
```

Figure 10: Main render loop

This is the render loop which is called for every frame. During each frame, the position of the light source is updated, camera movement is handled from keyboard and mouse input and the lighting is calculated. Because the light source is moving, the lighting of the cube has to be recalculated at each frame. This is done by passing the value of several variables (uniforms in previous code) into the appropriate shaders which then process the proper calculations and color the scene appropriately. Modern graphics both a front and back buffer, each frame, the GPU draws the next image into the back buffer while the front buffer is being sent to the display. Once rendering finishes, the buffers swap. This ensures the user always sees a complete, tear-free image rather than partial updates.

3.4 Cube Fragment Shader

```
#version 330 core
out vec4 FragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;
uniform vec3 lightPos;
uniform vec3 viewPos;
in vec3 Normal;
in vec3 FragPos;


void main()
{


    float ambientS = 0.1;
    vec3 ambient = ambientS * lightColor;

    float specularStrength = 1.0;
    vec3 viewDir = normalize(viewPos - FragPos);




    vec3 norm = normalize(Normal);

    vec3 lightDir = normalize(lightPos - FragPos);


    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    vec3 specular = specularStrength * spec * lightColor;

    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    vec3 result = (ambient + diffuse + specular) * objectColor;

    FragColor = vec4(result, 1.0);
}
```

Figure 11: Enter caption

This is a picture of one of the four shaders used in this project. In a nutshell, the fragment shader takes in a variety of inputs like position and lighting and implements the correct color for each pixel. It is called every frame within the render loop above and its inputs are updated to match changed positions and/or textures. You might notice the code here is the same as the samples provided during the explanation of each lighting technique, that is because the fragment shader is where all those calculations are done!

4 Conclusion

This project translated otherwise abstract vector-math concepts such as the dot-product into a concrete OpenGL application through use of the Phong lighting model. It took around 20 hours in total as I discovered the hard way that there is a steep learning curve to graphics programming. Nevertheless, this project was very fun and proved to be particularly edifying. Thank you for reading.

Bibliography

References

Online Resources

- OpenGL Programming Guide ("The Red Book")
- Wikipedia—Phong Reflection Model
- Wikipedia—Specular Highlight
- LearnOpenGL (background and images)

- YouTube—Video on Phong Lighting