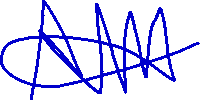
Anna Irene Margaritis S1905566  
Computer Games (Software Development), Graphics Programming coursework documentation

*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award*.

*Signature*.



Appendix

[1 Introduction 3](#_Toc103270526)

[2 Vertex Shader 3](#_Toc103270527)

[3 Fragment Shader 4](#_Toc103270528)

# Introduction

The finished project can be found here: <https://github.com/Shirehii/Graphics-Programming>

The shader used on the third model of this project is using the Phong lighting model. It is compromised of 3 main parts, those being the ambient, diffuse and specular lighting components. Additionally, the shader program uses the model’s texture in its calculations, so that the model will have a color based on the texture’s colors, instead of a preset one. The next few sections will explain the calculations that take place in the shaders to simulate lighting on the model.

# Vertex Shader

We begin by setting the positions and normal in the vertex shader. Next are the uniforms for the model matrix, the camera’s view and projection matrices. The vertex shader will forward the fragment’s position and the normal vectors to the fragment shader as calculations are done there instead.

   
*Figure 1: Code snippet showing the variables of the vertex shader*

For the texturing part, we set the texture coordinates, and also set them up to be forwarded to the fragment shader for later calculations.

To get the fragment’s position, we convert the vertex position to world space by multiplying the vertex position with the model matrix. Afterwards, we calculate the transpose of the inverse of the model matrix before multiplying it with the normal to ensure that scaling won’t distort the normal. Lastly, to convert the 3D point into a 2D point on the screen, the fragment position gets multiplied by the projection and the view matrices.

   
*Figure 2: Code snippet showing the main() function of the vertex shader*

As a last step, we pass the texture coordinates to the variable that will get forwarded to the fragment shader.

# Fragment Shader

The fragments shader receives input variables for the normal, fragment position and texture coordinates. It also contains uniforms for the light’s position, its color, the object’s color, and the view position, as well as a diffuse which will be used in texture calculations. The only output will be the fragment color.

   
*Figure 3: Code snippet showing the variables of the fragment shader*

The first step is to do the texture calculations before anything else. ADS lighting will then be applied on top of the previous calculations. For the ambient component, we simply set the ambient strength in a variable, then multiply the strength by the light color and store the result.

The diffuse requires the normal and the light direction. As we don’t care about the magnitude or the position of the vectors, it is better to simplify them into unit vectors. We do this by normalizing both of them before storing them in their respective variables, essentially saving only their directions. The light direction is calculated by subtracting the fragment position from the light position. To calculate the diffuse impact on the current fragment we need the dot product of the normal and light direction vectors. Lastly, the result is multiplied with the light color vector, and the diffuse is stored in its own variable.

The last component is the specular lighting. Like with the ambient component, we begin by defining the strength of the specular component. Next comes the calculation of the view direction and reflect direction vectors. The view direction is calculated by subtracting the fragment position from the view position, then normalizing the vector to get just the direction. Note that when calculating the reflect direction, the light direction vector is negated. This is because of the reflect function, which requires that the first vector is pointing from the light source to the fragment position. However, our light direction vector is the other way around, which is why we reverse it with the minus sign. To get the specular component we begin by getting the dot product of the view and reflect direction vectors. We also use max() to ensure that the result is always positive. Afterwards, we take the dot product and raise it to the power of 32. Different values result in different shininess levels, with lower values making the highlight seem bigger, and higher values making the highlight smaller. After taking that result and multiplying it with the specular strength defined at the beginning and the light color, we are ready to combine the components.

   
*Figure 4: Code snippet showing the main() function of the fragment shader*

To properly simulate Phong lighting, we need to combine the ambient, diffuse and specular components. We do this simply by adding them together. This result is then added to FragColor. Note that we’re using ‘+=’ since we’ve already assigned a value to FragColor when applying the texture, this is what allows the model to get a color based on the texture.