Laboratory work 7

1 Objectives

The objective of this laboratory is to briefly describe how to add lighting to your OpenGL 4 scene. We will cover the basic mathematics behind OpenGL lighting and the structure of the vertex and fragment shader for per-vertex and per-fragment directional lighting.

2 Theoretical background

OpenGL fixed-pipeline lighting (the Gouraud model) represents the basic lighting model for graphics applications and is a decent attempt at modeling how real-world illumination works. However, the fixed functionality is constraining, lacks in realism and in performance-quality tradeoffs. Programmable shaders can provide far superior results, especially in realism. Nevertheless, it is quite important to thoroughly understand the basic OpenGL lighting model, since this lighting model still provides the fundamentals on which advanced models are based.

OpenGL fixed-pipeline lighting model sums-up a set of independently computed components to obtain the overall illumination effect of a spot on an object. These components can be visually observed in Figure 1 below.

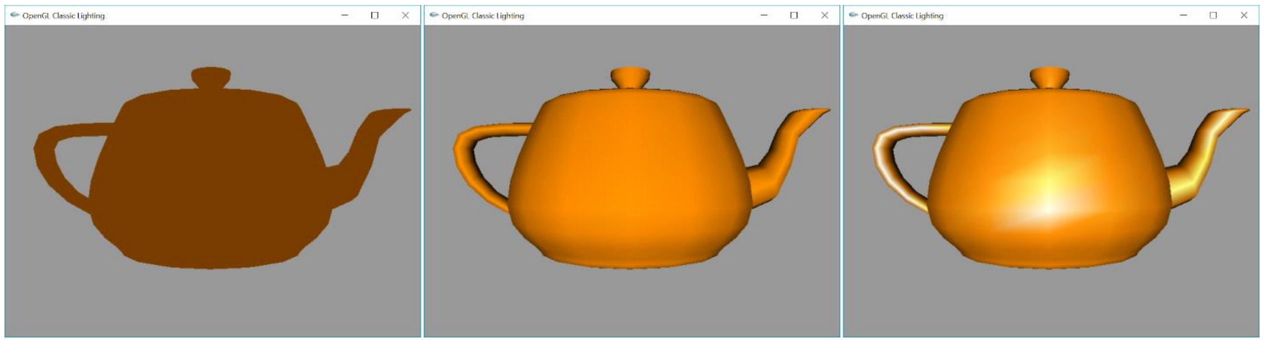

Figure 1 - OpenGL light components (from left to right: Ambient, Ambient + Diffuse, Ambient + Diffuse + Specular)

Ambient light does not come from any specific direction and is a decent approximation for light that exists scattered around in a scene. Its computation does not depend on the viewer's position or on the direction of the light and it can be either precomputed as a global effect or added on a per-light basis.

Diffuse light is scattered equally in all directions for a light source. The amount of diffuse light existent on a spot does not depend on the viewer's position, but it does depend on the direction of the light. It is brighter on the surfaces that are oriented towards the light and dimmer on the ones oriented away from it. Its computation depends on the surface **normal** and on the direction of the light source, but not on the direction of the eye.

Specular highlighting is light directly reflected by the surface and it refers to how much the material behaves similarly to a mirror. The strength of this effect is referred to as **Shininess**. Its computation requires knowing how close the surface's orientation is to the direction of direct reflection between the light direction and the eye, hence it depends on the surface **normal**, on the direction of the light and on the viewing direction.

3 OpenGL 4 lighting implementation

We will now incrementally build the shaders required for implementing the OpenGL classic lighting model (Gouraud lighting) using the GLSL shading language. Since the classical model is implemented on a per-vertex basis, the shader that we will incrementally change is the vertex shader. The fragment shader will remain the same throughout this process and it will be a simple pass through shader as presented below.

```
//pass-through fragment shader for OpenGL lighting
#version 410 core

uniform vec3 baseColor;

in vec3 color;

out vec4 fColor;

void main()
{
		fColor = vec4(color, 1.0f);
}
```

In theory, lighting computation can be carried out in any space (coordinate system) along the transformations pipeline. However, for correct (and thus more realistic) results, lighting should be performed in an affine space, namely in world or eye space. Both will offer identical lighting results. For this laboratory, we will be carrying out lighting computation in eye space.

3.1 No light

When no lighting is used, OpenGL renders each object uniformly using its base color. All lighting effects will modulate this base color as a basis of following lighting calculations. The vertex shader for drawing an object without lighting effects is presented below. For this laboratory, the base color is passed from the application, as a uniform value. In real-world applications, the base color is usually passed as a vertex attribute.

```
//vertex shader for no lighting
#version 410 core

layout(location=0) in vec3 vPosition;
layout(location=1) in vec3 vNormal;

//matrices
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

//lighting
uniform vec3 lightColor;
uniform vec3 baseColor;

out vec3 color;

void main()
{
		//compute final vertex color
		color = baseColor;

		//transform vertex
		gl_Position = projection * view * model * vec4(vPosition, 1.0f);
}
```

3.2 Ambient light only

Ambient light is uniform and thus doesn't change across vertices. We can pass the Ambient component from the application as a uniform variable. The interaction result between the ambient light and the color of the surface it hits is modeled through multiplication. During the various multiplications required to compute the effects of various light components on a surface, it is ok for components' values to go above 1.0. However, the final color needs to be saturated at white (i.e. no light component must be above 1.0), thus the usage of the **min** function to achieve this saturation. The additions and alterations to the vertex shader for incorporating ambient light are highlighted below.

```
//vertex shader for ambient lighting
#version 410 core

layout(location=0) in vec3 vPosition;
layout(location=1) in vec3 vNormal;

//matrices
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

//lighting
uniform vec3 lightColor;
uniform vec3 baseColor;

out vec3 color;

vec3 ambient;
float ambientStrength = 0.2f;

void main()
{
		//compute ambient light
		ambient = ambientStrength * lightColor;

		//compute final vertex color
		color = min(ambient * baseColor, 1.0f);

		//transform vertex
		gl_Position = projection * view * model * vec4(vPosition, 1.0f);
}
```

3.3 Ambient and diffuse lighting

Diffuse lighting depends on the direction of the light that hits the spot we are currently computing. Since our light is a **directional** one (located at infinity, as opposed to point light sources which have a defined 3D position), all light rays are parallel and have the same direction. This means that the direction of the light is constant across all vertices and can be computed in the application and passed as a **uniform** variable to the shader. The amount of scattered diffuse light depends on the surface's orientation (i.e. the angle between the direction of the light and the surface's normal). The maximum amount of scattered light is obtained when the light hits the surface perpendicularly (i.e. the angle between the normal and the direction of the light is 0.0, thus the cosine is 1.0). As this angle increases (the surface turns away from the light), the cosine decreases towards 0.0, thus the amount of scattered diffuse light is lower. The cosine of the angle between two normalized vectors can be computed as their dot product.

The vertex normal are passed in from the application (they are pre-computed in the GLM model) as a per-vertex attribute (similarly to vertex positions). As the model is affected by transformations (each vertex is multiplied with the Model, View and Projection matrices) the normal must be subjected to the same transformations as the object, otherwise the illumination will not work properly (i.e. the normal must move together with the object). In OpenGL, normals are not transformed using the Model-View-Projection matrix, but using the so called Normal Matrix. The Normal Matrix is the upper left 3x3 matrix of the transposed inverse of the Model matrix (if carrying out lighting computation in world space) or Model-View matrix (if carrying out lighting computation in eye space) and can be easily computed in the application using the OpenGL mathematics (GLM) library and passed to the shader as a **uniform** variable. The additions to the vertex shader to incorporate diffuse light are highlighted below.

```
//vertex shader for ambient and diffuse lighting
#version 410 core

layout(location=0) in vec3 vPosition;
layout(location=1) in vec3 vNormal;

//matrices
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat3 normalMatrix;

//lighting
uniform vec3 lightDir;
uniform vec3 lightColor;
uniform vec3 baseColor;

out vec3 color;

vec3 ambient;
float ambientStrength = 0.2f;
vec3 diffuse;
```

```
void main()
{
        //compute ambient light
        ambient = ambientStrength * lightColor;

        //normalize the light's direction
        vec3 lightDirN = normalize(lightDir);

        //compute eye coordinates for normals (transform normals)
        vec3 normalEye = normalize(normalMatrix * vNormal);

        //compute diffuse light
        diffuse = max(dot(normalEye, lightDirN), 0.0f) * lightColor;

        //compute final vertex color
        color = min((ambient + diffuse) * baseColor, 1.0f);

        //transform vertex
        gl_Position = projection * view * model * vec4(vPosition, 1.0f);
}
```

3.4 Ambient, diffuse and specular lighting

Specular highlights depend both on the direction of the light and the direction of viewing. The specular component depends on the cosine of the angle (α) between the viewing direction (Eye) and the reflection coming from the light (R) (Figure 2). Since we carry out lighting computation in eye coordinates, the camera is situated at the origin, thus the Eye position is (0, 0, 0).

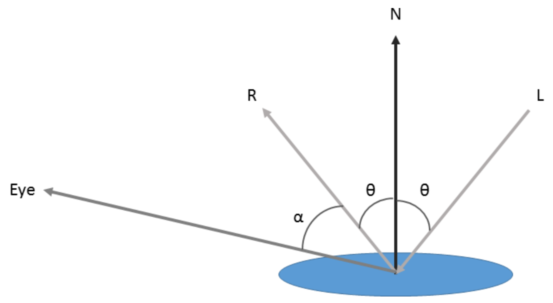

Figure 2 – Specular highlight computation

Shininess is used as an exponent to obtain a measure of the sharpness of the angular fall from a direct reflection. The grater the exponent, the more concentrated the specular highlight, since raising a number smaller than 1.0 to an exponential power decreases its value. Reflection angles close to 0.0 will result in values close to 1.0 for the specular highlight, while for other angles it quickly decays to 0.0. Because specular highlights are reflected light, the specular component is not modulated with the object's color. The additions to the vertex shader to incorporate specular highlights are highlighted below.

```
//vertex shader for ambient, diffuse and specular lighting
#version 410 core

layout(location=0) in vec3 vPosition;
layout(location=1) in vec3 vNormal;
```

```
//matrices
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat3 normalMatrix;

//lighting
uniform vec3 lightDir;
uniform vec3 lightColor;
uniform vec3 baseColor;

out vec3 color;

vec3 ambient;
float ambientStrength = 0.2f;
vec3 diffuse;
vec3 specular;
float specularStrength = 0.5f;
float shininess = 32.0f;

void main()
{
	//compute ambient light
	ambient = ambientStrength * lightColor;

	//normalize the light's direction
	vec3 lightDirN = normalize(lightDir);

	//compute eye coordinates for normals
	vec3 normalEye = normalize(normalMatrix * vNormal);

	//compute diffuse light
	diffuse = max(dot(normalEye, lightDirN), 0.0f) * lightColor;

	//compute the vertex position in eye coordinates
	vec4 vertPosEye = view * model * vec4(vPosition, 1.0f);

	//compute the view (Eye) direction (in eye coordinates, the camera is at the origin)
	vec3 viewDir = normalize(- vertPosEye.xyz);

	//compute the light's reflection (the reflect function requires a direction pointing towards the vertex, not away
from it)
	vec3 reflectDir = normalize(reflect(-lightDir, normalEye));

	//compute specular light
	float specCoeff = pow(max(dot(viewDir, reflectDir), 0.0f), shininess);
	specular = specularStrength * specCoeff * lightColor;

	//compute final vertex color
	color = min((ambient + diffuse) * baseColor + specular, 1.0f);

	//transform vertex
	gl_Position = projection * view * model * vec4(vPosition, 1.0f);
}
```

4 Tutorial

Download the resources available on the laboratory web page. The first archive (Laboratory 7 resources - part 1.zip) contains an application with the vertex and fragment shaders required for rendering a teapot without lighting (see subsection 3.1)

Extend the vertex shader as to incorporate ambient lighting into your application (see section 3.2). Your application should render the teapot as in Figure 3 below.

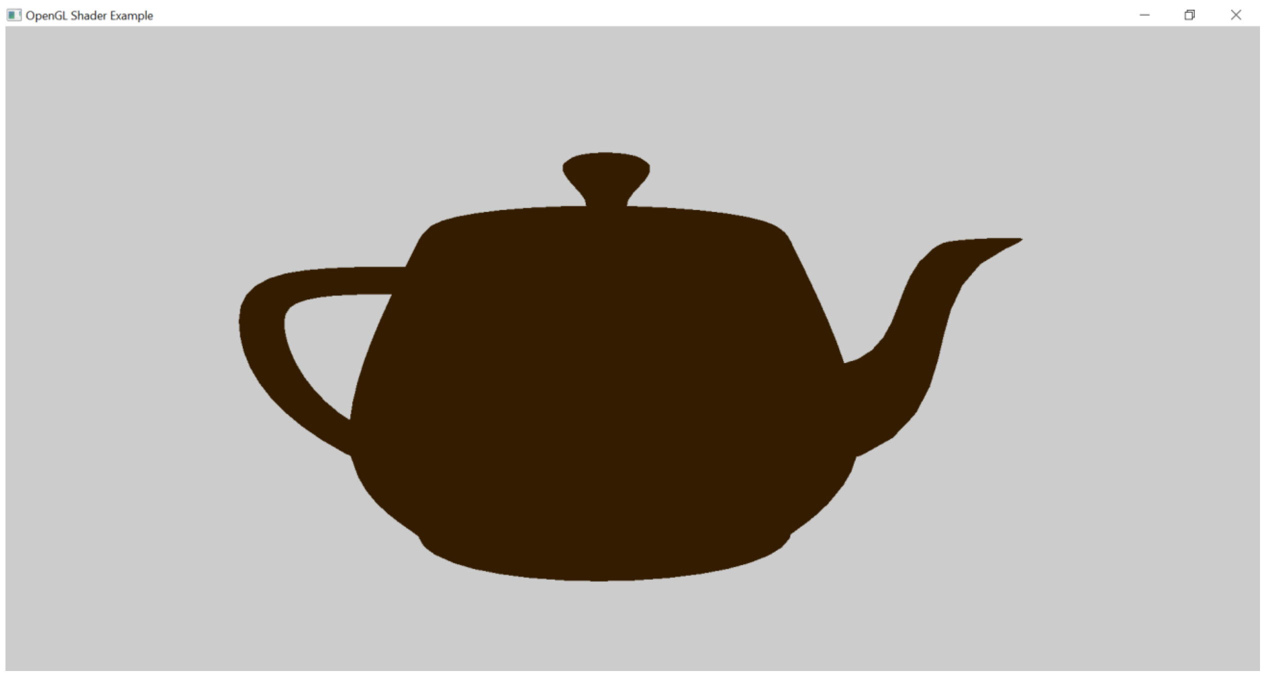

Figure 3 – Rendering with ambient lighting

Extend the vertex shader as to incorporate diffuse lighting into your application (see section 3.3).

Your application should render the teapot as in Figure 4 below.

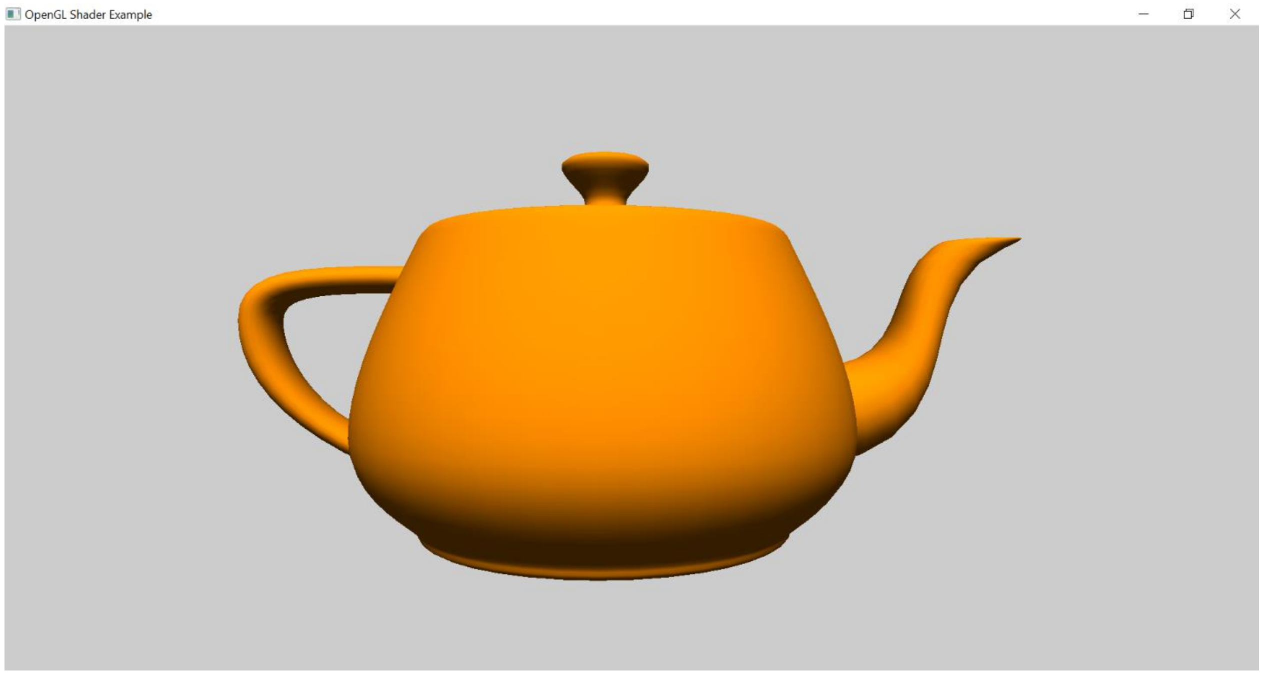

Figure 4 – Rendering with ambient and diffuse lighting

Extend the vertex shader as to incorporate specular lighting into your application (see section 3.4). Your application should render the teapot as in Figure 5 below.

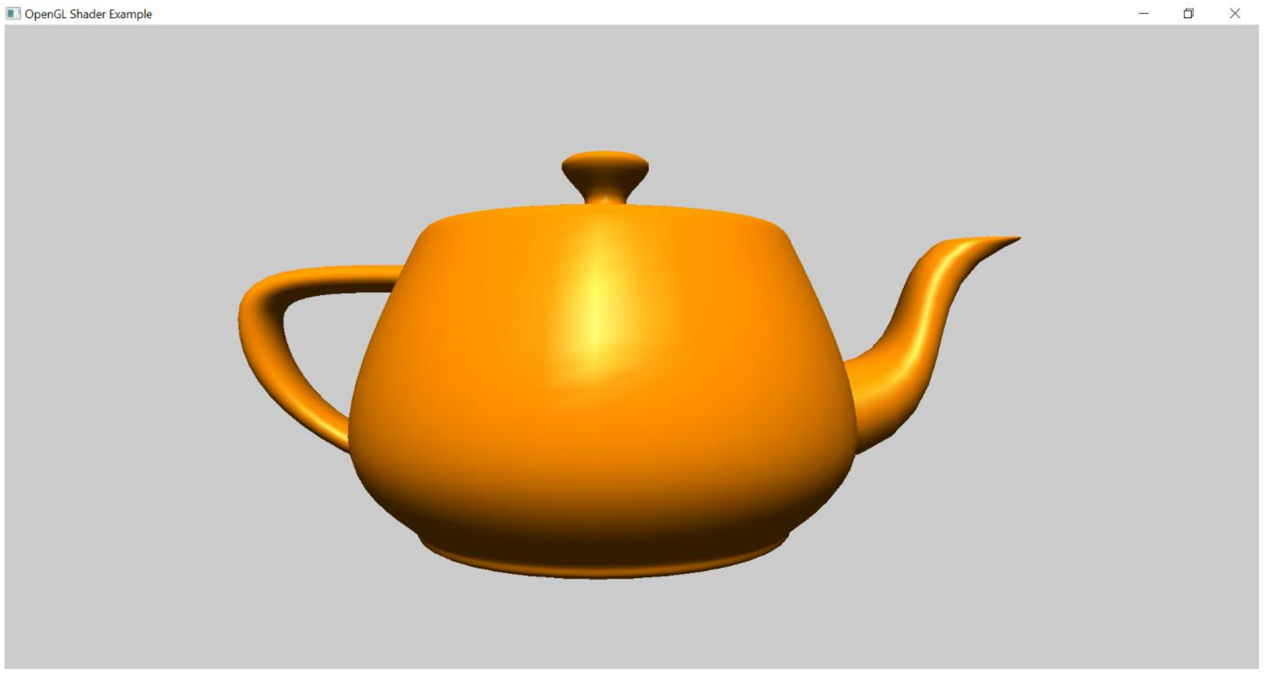

Figure 5 – Rendering with ambient, diffuse and specular lighting

5 Further reading

- OpenGL Programming Guide 8th Edition – Chapter 7

6 Assignment

1. Implement OpenGL per-vertex lighting (Gouraud) for your application following the steps described in section 4
2. Implement per-pixel lighting (Phong). Per-pixel lighting computes the final colors in the fragment shader, using the **interpolated normals** instead of using the vertex normals in the vertex shader. Compare the overall lighting quality with the per-vertex model obtained at step 1, for all teapot resolutions (there are several teapots available, generated using 4, 10, 20 and 50 segments). For this task, use the resources provided in the second archive (Laboratory 7 resources - part 2.zip) and modify the set of shaders shaderPPL.(vert/frag).
3. Extend the per-pixel lighting implementation to incorporate material properties for objects. Material properties can be passed from the application as uniform variables and the final color will be modulated (multiplied) with the corresponding material component (Ambient, Diffuse and Specular) instead of the object's base color.
4. Extend the per-pixel lighting implementation to incorporate light component intensities. Instead of having a single light color and using hard-coded coefficients to decide the contribution of each lighting component, describe a light source using three different components (ambient, diffuse and specular) and use them to compute the final color intensities.