6 Lighting

The perception of objects of the real world by humans relies mainly on vision. This form of perception is based on the projection of light on the human retina as the interation of the light and the scene objects. OpenGL simulates this via some simple but effective techniques that try to balance the quality of the rendered scene and the realtime requirements of interactive computer graphics. We call *Lighting* to use of techniques applied to increase the realism of these scenes. Lighting indeed simulates the interaction of light and the objects, mainly considering how objects reflect light given:

- the object material composition
- the light colour and source localization
- other global lighting parameters

Typically three different lighting components are computer: ambient, diffuse, and specular.

6.1 Ambient

Ambient light does not come from any specific direction. It can be seen as the remaining light that affects the non-illuminated faces of objects. It can be accumulated per light source. For a particular fragment of an object the resulting colour is computed from the interaction of the object colour and the light (colour and intensity). As both colours and light are defined by RGB components between 0.0 and 1.0, if an object has a material for which the ambient component is $K_a = (k_r, k_g, k_b, k_\alpha)$, then each of the components will be multiplied by the corresponding components of the light I, as

$$C_{amb} = IK_a.$$

Note here that light can be coloured and that will affect the resulting colour of the objects. For instance, if an object is white and light is green the object will be viewed with a green tone. Note that GLSL provides vector operations that enables the above operation to be written as

```
fcolour=vec4(light.rgb*objectambientcolour.rgb, objectambientcolout.a);
```

note that the light does not change the transparency parameter of the objects in the above.

6.2 Diffuse

The diffuse component accounts for the light component which is scattered in all directions (from an object) for a particular light source. This component is responsible for making a surface visible even when it does not reflect the light directly to your eyes (or camera). The computation of this component depends on the surface normal, and the direction of the light source, but not on the viewing direction. By consequence in addition to the vertex coordinates, vertex colours and eventually texture coordinates, we will need to pass also the vertex normals, as well as the light parameters.

In this case we should consider different cases of lights: directional, point lights and spotlights.

Directional light

Directional light is typically used to simulate far light sources and in most cases sunlight effects. Instead of a location of the light source what matters in this case is the vector that represents the lighting direction. This means that object surface will be lit non-uniformly, as in the previous case, producing shading effects that will also contribute to the perception of its three dimensional structure. The visible object zones will have brighter or darker colours depending on their orientation with respect to the light direction. This can be obtained by using the inner product between the normal of the surface at the fragment (using interpolation) and direction of light as in the following equation

$$C_{dif} = IM_d\left(\vec{N}.\vec{L}\right) \tag{6.1}$$

where C_f represents the resulting RGB components of the fragment colour, I the light RGB components, M_d the RGB diffuse components of the object "material", $\vec{N}$ the surface normal and $\vec{L}$ the light direction vector.

Point light sources

Point light sources mimic lights near or within the scene: lamps, ceiling lights, street lights, etc. In this case light direction changes along the surface of the objects. To provide a better approximation of these types of lights an attenuation factor is used to reflect the decrease in the amount of light received with the increase in the distance to the source. Given this, the differences in respect to the previous case is that the direction of light is replaced by the position of the light source and the attenuation factor is added. Defining $D = ||P_f - P_l||$, light direction is computed as

$$\vec{L} = \frac{P_f - P_l}{D}$$

where P_f if the fragment position and P_l the light position. Equation (6.1) is modified to take into account the attenuation factor given by

$$K_A = \frac{1}{A_c + A_f D + A_q D^2}$$

it becomes

$$C_{dif} = I M_d K_A \left(\vec{N}.\vec{L}\right) \tag{6.2}$$

Spotlights

Spotlights project strong beams of light that illuminate limited areas. They can be modelled as point lights limited by cones, or using some decay function that depends on the distance to the light direction axis. (to be completed)

6.3 Specular

This component accounts for the amount of light that is directly reflected towards the viewer. It accounts for how much the surface behaves like a mirror. Smooth surfaces are very reflective whereas rough surfaces are not. The computation of this component uses the surface normal, the direction of the light source and the direction of the viewer. The specular component can be computed for either directed, point, or spot lights. Figure 6.1 shows the vectors involved in this computation for a given point in the object surface. They are: $\vec{L}$ pointing towards the light, the $\vec{N}$ the local surface normal, $\vec{V}$ the viewer direction - pointing towards the camera, $\vec{R}$ the reflected light ray direction. Using this figure it is easy to understand the

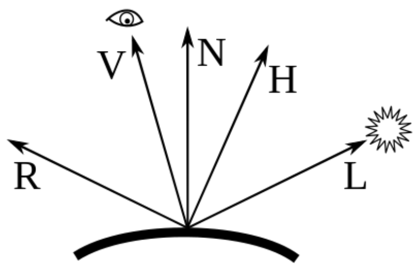

Figure 6.1: Vectors used in the definition of the specular illumination components

computation that of the reflected vector as

$$\vec{R} = 2\left(\vec{L}.\vec{N}\right)\vec{N}\ \vec{L}$$

which can be implemented at the fragment shader level.

The specular component is then computed using

$$C_{spec} = IK_s\left(\vec{R}.\vec{V}\right)^n$$

where I represents the light components and K_s the specular coefficients, finally the exponent n will define how sharp the specular effects are.

6.4 Blinn-Phong simplification

From the above we have shown that the computation of the lighting effects for each fragment is the sum of the 3 components of light: ambient, diffuse and specular. In figure 6.2 you can see the results of each component for a particular case and the composition of the 3 on one example taken from wikipedia.

In particular computing the specular component represents a considerable amount of work for each fragment of the surface. The Blinn-Phong simplification proposes to replace the computation of the reflection vector $\vec{R}.\vec{V}$ by $\vec{N}.\vec{H}$, where the "half-vector" is defined as

$$\vec{H} = \frac{\vec{L}+\vec{V}}{||\vec{L}+\vec{V}||}.$$

Its computation can be performed at the vertex shader level what reduces considerably the necessary computations.

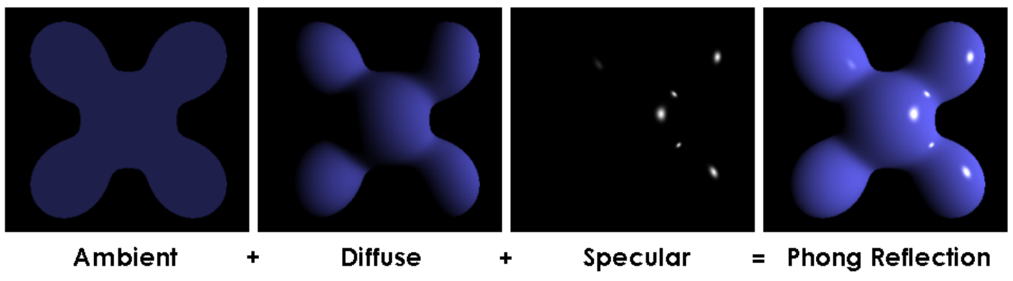

Figure 6.2: Phong lighting model results for ambient, diffuse, specular components, and their composition. Example taken from Wikipedia

6.5 Exercises

Start with one or more objects, e.g. cube or sphere, that you have created by yourself, thus reusing your own code.

Exercise 6.1

The following example code is a recall of a vertex and fragment shaders that do not use any lighting effects, and is similar to the one used previously. Develop (reusing previously build code) and application with moving objects, using the shaders below. Make sure that the objects have solid colours, i.e. a single colour to every vertex.

```
——————————————————— Vertex Shader ———————————————————
// Vertex shader with no lighting
#version 330 core
uniform mat4 MVPMatrix; // model−view−projection transform
in vec4 VertexColor; // sent from the application , includes alpha
in vec4 VertexPosition;// pre−transformed position
out vec4 Color; // sent to the rasterizer for interpolation
void main() {
      Color = VertexColor;
      gl_Position = MVPMatrix * VertexPosition;
 }
```

```
——————————————————— Fragment Shader ———————————————————
// Fragment shader with no lighting
#version 330 core

in vec4 Color; // interpolated between vertices
out vec4 FragColor; // color result for this fragment
void main() {
      FragColor = Color;
 }
```

Exercise 6.2

Now use a shader that creates interaction with ambient light by replacing the above by the following shaders. Explain every detail of this code.

```
Vertex Shader
// Vertex shader for ambient light
#version 330 core
uniform mat4 MVPMatrix;
in vec4 VertexColor;
in vec4 VertexPosition;
out vec4 Color;
void main() {
      Color = VertexColor;
      gl_Position = MVPMatrix * VertexPosition;
 }
```

```
Fragment Shader
 // Fragment shader for global ambient lighting
#version 330 core
uniform vec4 Ambient; // sets lighting level, same across many vertices
in vec4 Color;
out vec4 FragColor;
void main() {
    vec4 scatteredLight = Ambient; // this is the only light
    // modulate surface color with light, but saturate at white
    FragColor = min(Color * scatteredLight, vec4(1.0));
}
```

Note: It is important to make sure that the light computations do not change the transparency of surfaces. For this the following code snippet shows how to deal with this.

```
vec3 scatteredLight = vec3(Ambient);// this is the only light
vec3 rgb = min(Color.rgb * scatteredLight, vec3(1.0));
FragColor = vec4(rgb, Color.a);
```

Exercise 6.3

Place two similar squares side by side rotating along one axis of your choice, where one uses the "no lighting" shader, and the other uses the "ambient light" shader.

Exercise 6.4

In this exercise you will modify your code to support directional light (sunlight), making sure that the light direction is such that as the squares rotate the incidence angle on their surface changes. For this you have to

1. Add normals to the vertices, one normal per vertex.

2. Each of these normals must suffer the same afine transformation as the solid itself. For this add to the vertex shader the code to multiply the normal by the transformation matrix and normalize the result, using the function $y = normalize(x)$.

3. User the shaders but first you must understand which are the parameters that must be passed

```
Vertex Shader
#version 330 core
uniform mat4 MVPMatrix;
uniform mat3 NormalMatrix; // to transform normals, pre-perspective
in vec4 VertexColor;
in vec3 VertexNormal;
in vec4 VertexPosition;
out vec4 Color;
out vec3 Normal;
void main() {
    Color = VertexColor;
    // we now need a surface normal
    // interpolate the normalized surface normal
    // transform the normal, without perspective, and normalize it
    Normal = normalize(NormalMatrix * VertexNormal);
    gl_Position = MVPMatrix * VertexPosition;
}
```

```
Fragment Shader
#version 330 core
uniform vec3 Ambient;
uniform vec3 LightColor;
uniform vec3 LightDirection; // direction toward the light
uniform vec3 HalfVector; // surface orientation for shiniest spots
uniform float Shininess; // exponent for sharping highlights
uniform float Strength; // extra factor to adjust shininess
in vec4 Color;
in vec3 Normal; // surface normal, interpolated between vertices
out vec4 FragColor;
void main() {
    // compute cosine of the directions, using dot products,
    // to see how much light would be reflected
    float diffuse = max(0.0, dot(Normal, LightDirection));
    float specular = max(0.0, dot(Normal, HalfVector));

    // surfaces facing away from the light
    // won't be lit by the directional light
    if (diffuse == 0.0)
        specular = 0.0;
    else
        specular = pow(specular, Shininess); // sharpen the highlight
    vec3 scatteredLight = Ambient + LightColor * diffuse;
    vec3 reflectedLight = LightColor * specular * Strength;
    // don't modulate the underlying color with reflected light,
    // only with scattered light
    vec3 rgb = min(Color.rgb * scatteredLight + reflectedLight, vec3(1.0));
    FragColor = vec4(rgb, Color.a);
}
```

Exercise 6.5

Repeat the above with another shape, for example a cube.