Assignment 17 (Ambient Lighting)

Considering only direct light sources (such as directional, point or spotlight) can produce very dark images. Realistic rendering techniques, tries to consider also indirect lighting: illumination caused by lights that bounces from other objects.

Ambient lighting is the simplest approximation for indirect illumination.

Ambient Lighting

The **ambient light emission factor** (constant for the entire scene) accounts for the light reflected by all the objects in all the directions:

$$\boldsymbol{l}_A = (l_{AR}, l_{AG}, l_{AB})$$

The BRDF of the object, is then extended by adding another component $f_A(x, \omega_r)$ that specifically considers ambient lighting.

$$L\left(x, \omega_r\right) = \sum_l L(l, \overrightarrow{lx}) f_r\left(x, \overrightarrow{lx}, \omega_r\right) + l_A f_A\left(x, \omega_r\right)$$

Note that such component does not depend on the light direction.

Hemispheric Lighting

A slight extension of ambient lighting is the hemispheric lighting.

In this case, there are two ambient light colors (the "upper" or "sky" color, and the "lower" or "ground" color) and a direction vector.

The two colors, l_U and l_D, represent the values of the ambient light at the two extremes, and the direction vector d orients the blending of the two colors.

The rendering equation is then modified to have the ambient light term $l_A(x)$ that depends on the orientation of the surface of the object in the considered point x.

$$L\left(x, \omega_r\right) = \sum_l L(l, \overrightarrow{lx}) f_r\left(x, \overrightarrow{lx}, \omega_r\right) + l_A(x) f_A\left(x, \omega_r\right)$$

The orientation of the object is characterized by n_x which is the normal vector to the surface in point x.

- If the normal vector is aligned and in the same direction as d, ambient color l_U is used.
- If the normal vector is instead aligned, but in the opposite direction of d, ambient color l_D is used.
- For normal vectors oriented in other directions, the two colors are blended proportionally to the cosine of their angle with vector d:

$$\boldsymbol{l}_A(x) = \frac{n_x \cdot d + 1}{2} \boldsymbol{l}_U + \frac{1 - n_x \cdot d}{2} \boldsymbol{l}_D$$

Image Based Lighting

Spectral expansion (Spherical Harmonics)

Encoding function $l_A(x)$ in an efficient way is not a simple task. The simplest expansion using the Spherical Harmonics requires only four values: a basic color l_C, plus three "deviation terms" along the three main axis Δl_x, Δl_y and Δl_z.

If we call respectively $(n_x).x$, $(n_x).y$ and $(n_x).z$ the components of the (unitary) normal vector direction, we have:

$$\boldsymbol{l}_A(x) = \boldsymbol{l}_C + (n_x) \cdot x\boldsymbol{\Delta l}_x + (n_x) \cdot y\boldsymbol{\Delta l}_y + (n_x) \cdot z\boldsymbol{\Delta l}_z$$

Vertex attributes and Uniforms

Shaders are the pipeline components that compute the positions and the colors of the pixels on screen corresponding to points of the objects.

The Light and BRDF models seen in the previous lessons, which are the building blocks for computing approximations to the rendering equations, are the main algorithms with which the color of the pixel can be determined. Such algorithms, can be parametrized to produce results that depends on the surface of the objects. Object and environment parameters can be passed to the shaders in two ways:

- Vertex attributes
- Uniform variables

Vertex attributes

Recall

1. `in` and `out` variables are one of the way in which shaders interfaces with the other components (either fixed or programmable) of the pipeline.
2. graphics primitives are sent to the graphic pipeline as triangles lists or triangle strips, encoded with the values defining their vertices. Vertex coordinates are sent into in variables of the Vertex Shader.

Vertices Each vertex has an implicit integer value that represents its index. It is contained in global variable `gl_VertexIndex`:

```
layout(location = 0) out vec3 fragColor;
void main() {
    gl_Position = vec4(positions(gl_VertexIndex), 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

In addition, it can have other an arbitrary set of user defined attributes, each one characterized by one of the supported GLSL types. Vertices may also have no user defined attributes.

A bi-dimensional scene, can use for example, just a set of `vec2` normalized screen coordinates to denote the position of the elements to connect:

```
#version 450
layout(location = 0) in vec2 inPosition;
void main() {
    gl_Position = vec4(inPosition, 0.0, 1.0);
}
```

A classical 3D scene, such as the one in Assignment 09, uses a single `vec3` element to store the positions in the 3D local space:

```
#version 450
layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 viewMat;
    mat4 prjMat;
} ubo;

layout(location = 0) in vec3 inPosition;
layout(location = 0) out vec3 fragPos;

void main() {
    gl_Position = ubo.mvpMat * vec4(inPosition, 1.0);
    fragPos = (ubo.mMat * vec4(inPosition, 1.0)).xyz;
}
```

All the vertices of one mesh must have the same vertex format, i.e. the same attributes. The fixed functions of the pipeline pass such values to the Vertex Shaders.

Communication between Vertex and Fragment shaders The in and out variables used for the shaders communication, is implemented with a set of slots, each one identified by a `location` number. Location numbers starts from zero, and are limited by a hardware dependent constant (usually large enough to support standard applications).

Whenever an `in` or `out` variable is defined, the user provides the location id of the slot used for communication in a `layout` directive:

```
layout(location = 0) in vec3 inPosition;
layout(location = 0) out float real;
layout(location = 1) out float img;
```

The slots used by the Input Assembler, which will be available inside in variables of the **Vertex Shader**, are configured in the pipeline creation. The configuration of the pipeline, also defines the out variables that the **Fragment Shader** will write to set the final color of the considered pixel (fragment).

Communication between the Vertex and Fragment shader is controlled by their GLSL specification. As we will see, the fixed functions of the pipeline interpolate the values of the out variables emitted by the Vertex Shader, according to the position of the corresponding pixels on screen, before passing their values to the Fragment Shader.

Vertex Input Descriptor Vulkan allows to split vertex data into different arrays, each one containing some of the attributes. Each of these array is known as a binding, and it is characterized by a progressive `binding id`.

We create a C++ structure containing all the vertex attributes, using the GLM types that match the ones defined in the corresponding Vertex Shader:

```
struct Vertex {
    glm::vec3 pos;
    glm::vec3 normal;
    glm::vec4 tangent;
    glm::vec2 texCoord;
}
```

The binding, is defined inside a `VkVertexInputBindingDescription` structure. Its field contains the `binding id` and the size of the object in bytes.

```
VkVertexInputBindingDescription bindingDescription{};
bindingDescription.binding = 0;
bindingDescription.stride = sizeof(Vertex);
bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```

Single attributes are defined inside the element of an array of `VkVertexInputAttributeDescription` structures.

```
td::array<VkVertexInputAttributeDescription, 4> attributeDescriptions{};
```

Each attribute definition contain the specification of both its `binding` and its `location ids`:

```
attributeDescriptions[0].binding = 0;
attributeDescriptions[0].location = 0;
attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
// the offset in byte inside the data structure
// for the considered field must be provided
attributeDescriptions[0].offset = offsetof(Vertex, pos);

attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, normal);

attributeDescriptions[2].binding = 0;
```

```
attributeDescriptions[2].location = 2;
attributeDescriptions[2].format = VK_FORMAT_R32G32B32A32_SFLOAT;
attributeDescriptions[2].offset = offsetof(Vertex, tangent);

attributeDescriptions[3].binding = 0;
attributeDescriptions[3].location = 3;
attributeDescriptions[3].format = VK_FORMAT_R32G32_SFLOAT;
attributeDescriptions[3].offset = offsetof(Vertex, texCoord);
```

The most common formats are the following:

```
float VK_FORMAT_R32_SFLOAT
vec2 VK_FORMAT_R32G32_SFLOAT
vec3 VK_FORMAT_R32G32B32_SFLOAT
vec4 VK_FORMAT_R32G32B32A32_SFLOAT
```

Input assembler configuration Next, both binding and attributes descriptors are collected into a `VkPipelineVertexInputStateCreateInfo` structure:

```
VkPipelineVertexInputStateCreateInfo vertexInputInfo{};
vertexInputInfo.sType =
  VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;


// The structure requires the pointer to
// both the binding and attribute arrays,
// and the count of the corresponding elements
vertexInputInfo.vertexBindingDescriptionCount = 1;
vertexInputInfo.vertexAttributeDescriptionCount = 4;
vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;
vertexInputInfo.pVertexAttributeDescriptions =
  attributeDescriptions.data();

VkPipelineInputAssemblyStateCreateInfo inputAssembly{};
inputAssembly.sType =
  VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssembly.primitiveRestartEnable = VK_FALSE;
```

The configuration of the Input Assembler stage of the graphic pipeline is competed filling up a `VkPipelineInputAssemblyStateCreateInfo` structure with the type of primitives being drawn – i.e. triangle lists or triangle strips (with or without restart).

The assignment

We need to create different cases in which we'll use different lighting models. We also need to correctly specify the location.

Diffuse reflection models

Lambert

According to the reflection law by Lambert, each point of an object hit by a ray of light, reflects it with uniform probability in all the directions.

The reflection is thus independent of the viewing angle and it corresponds to a constant BRDF: $f_r(x, \omega_i, \omega_r) = \rho_x$.

The quantity of light received by an object is however not constant: it depends on the angle between the ray of light and reflecting surface. Suppose we have the following parameters:

- n_x: unitary normal vector to the surface;
- d: direction of ray of light;
- α: angle between the two vectors.

The incidence of the incoming light is maximized when angle α is $0°$, and null if it is greater or equal than $90°$. In particular, Lambert has shown that the amount of light reflected is proportional to $\cos\alpha$, which can be computed as $\cos\alpha = n_x \cdot d$. In the assignment this becomes: `dot(L, N)`.

We can express the BRDF of the Lambert reflection for scan-line rendering with the following expression:

$$f_r\left(x, \overrightarrow{lx}, \omega_r\right) = f_{\text{diffuse}}\left(x, \overrightarrow{lx}\right) = \boldsymbol{m}_D \cdot \max\left(\overrightarrow{lx} \cdot \boldsymbol{n}_x, 0\right)$$

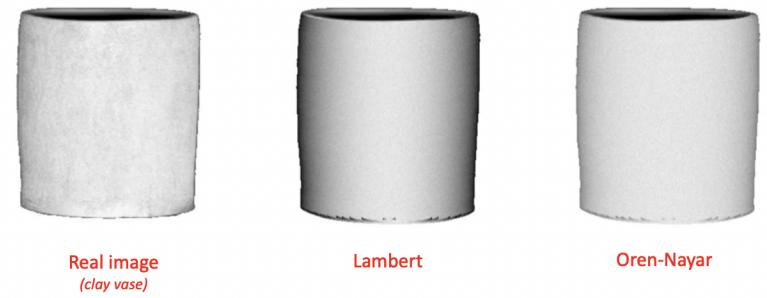
Which in shaders code becomes:

```
// vec3 C: color of the surface (RGB vector)
f = C * max(dot(L, N), 0.0);
```

Oren-Nayar

Idea: some materials are characterized by a phenomenon called retroreflection: they tend to reflect back in the direction of the light source. They are characterized by very rough surfaces, and they cannot be accurately simulated with the Lambert diffuse reflection model. The Oren-Nayar diffuse reflection model has been devised to more appropriately model such materials.

Typical real life materials that require this special technique are clay, dirt and some types of cloths:



It requires three vectors:

1. d: direction of the light;
2. n: normal vector;
3. ω_r: direction of the viewer.

The model is characterized by two parameters:

1. $m_D = (m_R, m_G, m_B)$: main color of the material;
2. $\sigma \epsilon \left[0, \frac{\pi}{2}\right]$: roughness of the material, higher values of σ produces rougher surfaces. If $\sigma = 0$ it converges to the Lambert diffusion.

The model, can be computed with the following formulas:

$$\theta_i = \cos^{-1}\left(\overrightarrow{lx} \cdot \boldsymbol{n}_x\right)$$
$$\theta_{\boldsymbol{r}} = \cos^{-1}\left(\omega_r \cdot \boldsymbol{n}_x\right)$$
$$\alpha = \max\left(\theta_{\boldsymbol{i}}, \theta_{\boldsymbol{r}}\right)$$
$$\beta = \min\left(\theta_{\boldsymbol{i}}, \theta_{\boldsymbol{r}}\right)$$
$$A = 1 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33}$$
$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09}$$
$$\boldsymbol{v_i} = \text{ normalize } \left(\overrightarrow{lx} - \left(\overrightarrow{lx} \cdot \boldsymbol{n}_x\right)\boldsymbol{n}_x\right)$$
$$\boldsymbol{v_r} = \text{ normalize } \left(\omega_r - \left(\omega_r \cdot \boldsymbol{n}_x\right)\boldsymbol{n}_x\right)$$
$$G = \max\left(0, \boldsymbol{v_i} \cdot \boldsymbol{v_r}\right)$$
$$L = m_D \cdot \text{clamp}\left(\overrightarrow{lx} \cdot \boldsymbol{n}_x\right)$$
$$f_{diffuse} = \left(x, \overrightarrow{lx}, \omega_r\right) = L(A + BG \sin\alpha \tan\beta)$$

Which in code become:

```
float thi = acos(dot(L, N));
float thr = acos(dot(V, N));
float alpha = max(thi, thr);
float beta = min(thi, thr);
```

```
float A = 1 - 0.5 * pow(sigma, 2) / (pow(sigma, 2) + 0.33);
float B = 0.45 * pow(sigma, 2) / (pow(sigma, 2) + 0.09);
vec3 vi = normalize(L - dot(L, N) * N);
vec3 vr = normalize(V - dot(V, N) * N);
float G = max(dot(vi, vr), 0.0);
vec3 L1 = C * clamp(dot(L, N), 0.0, 1.0);
return L1 * (A + B * G * sin(alpha) * tan(beta));
```

Blinn

The Blinn reflection model is an alternative to the Phong shading model that uses the half vector h: a vector that is in the middle between ω_r and d.

$$\boldsymbol{h}_{l,x} = \frac{\overrightarrow{lx} + \omega_r}{\left|\overrightarrow{lx} + \omega_r\right|} = \text{ normalize } \left(\overrightarrow{lx} + \omega_r\right)$$

The formula when considering Blinn specular reflection becomes:

$$f_{\text{specular}}\ (x, \vec{x}, \omega_r) = \boldsymbol{m}_S \cdot \text{clamp}\left(\boldsymbol{n}_x \cdot \boldsymbol{h}_{l,x}\right)^{\gamma}$$