Computer Graphics
Practical 6: Local illumination

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

INSA Fourth Year - 2023/2024
Maud Marchal, Glenn Kerbiriou

About this practical

In all previous practicals, the scene was not really pleasant to see. Indeed, the depth of objects was not perceptible and the appearance was neither realistic nor expressive. It is high time to change that! In this practical we will implement basic local illumination techniques: nothing fancy, just enough to bring back the fun in your scene.

1 About the provided code

The provided code enables to add different types of lights in the scene in order to lit objects and to reflect their material properties. Lights and material are based on the local illumination model of Phong.

For this practical, CPU and GPU codes are provided:

- CPU code is encapsulated into C++ classes that are gathered in a lighting subdirectory.
- GPU code is gathered in the shaders subdirectory.

The first group of classes is the core of the lighting system:

- The `DirectionalLight`, `PointLight`, `SpotLight` classes which are all declared inside the file `Light.hpp`. They define the attributes of the different kind of light in the Phong illumination model and provide functions to send these attributes to the GPU.
- The `Material` class. It defines the material of an object in the Phong illumination model.

The second group of classes corresponds to `Renderable objects` used for vizualisation:

- The `DirectionalLightRenderable`, `PointLightRenderable`, `SpotLightRenderable` classes. These classes allow to debug the parameters of your lights.
- The `LightedMeshRenderable`, `LightedCylinderRenderable` classes. They are similar to the renderable classes of previous practicals, but with the ability to be lit by lights. This is the kind of class you will need to adapt in your project to compute nice rendering for you scene.

The third group of code are the shaders. These are the files you will work on during this practical:

- The vertex shader `phongVertex.glsl`.
- The fragment shader `phongFragment.glsl`.

Comment: you do not need to call `make` to update your shaders. Shaders are compiled at run-time. The Viewer can reload, compile and link them on demand if you press F3. Edit your shader sources and press F3 to see the results, without relaunching the executable.

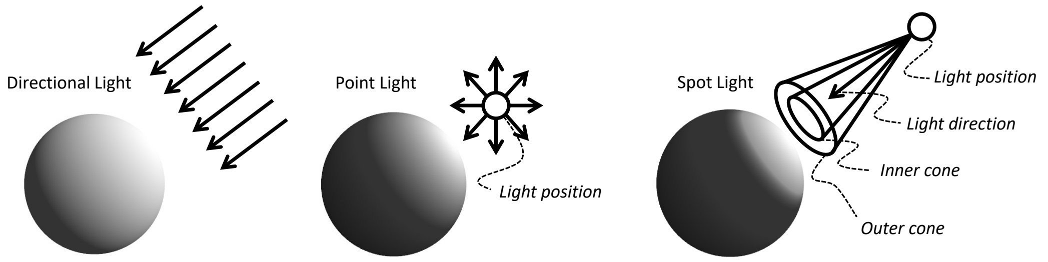
2 Light mechanism

You already know that the fragment shader processes fragments. A fragment is a linear (trilinear in case of triangle primitives) interpolation of vertex attributes, projected onto the screen. If the fragment is on top of all primitive fragments (its depth is smaller), it will become a pixel.

Here, vertices have position and normal attributes. Thus, fragments will have those two attributes too and will represent surfels: surface elements. While it may look like fragments and surfels are the same, keep in mind that they are in different spaces: fragments are on the 2D screen (screen coordinates) while surfel are in the 3D world (world coordinates).

A light is applied to a surfel. According to the Phong model, this light will react to the position and normal of the surfel, and also to the material applied on the surface. We will explain here how the light react to such properties to compute the appearance of a surfel. You may find different models on the internet: that's fine, people are adapting the same model to suit their need.

2.1 Light types

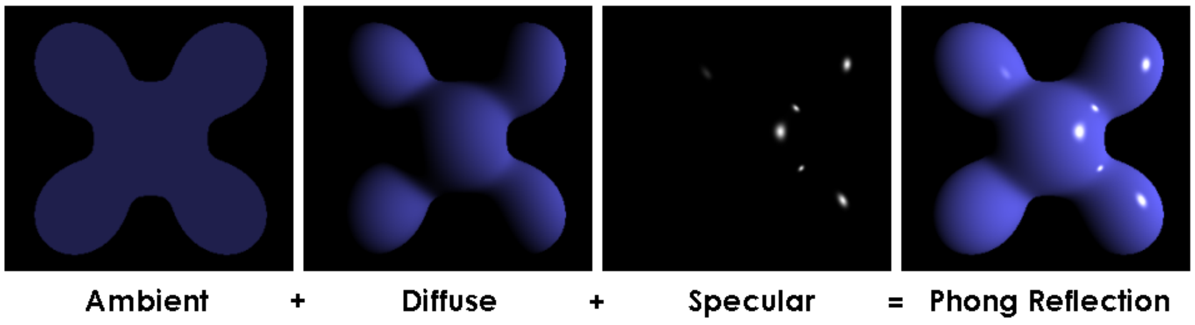


Three kind of (standard) lights are used in this tutorial:

- A directional light represents a light which source is so far away from the camera that the light rays can be considered parallel when they reach an object, regardless of the position of the objects or the camera. A good example is the sun. For such a light, we have the following direction: $\overrightarrow{surfel\_to\_light} = -\overrightarrow{light\_direction}$
- A point light is located at a given position, and illuminates in all directions. Good examples of point light are torches. Usually, the light intensity received by an objet decreases when its far from the source. In this case, we have this direction to light: $\overrightarrow{surfel\_to\_light}$=`normalize`$(light\_position - surfel\_position)$.

- A spot light is light which source is located somewhere in the environment, but rays are only shot in some directions instead of all around. An object receives light only if located in the cone defined by the spot. A good example of a spotlight would be a street lamp. The set of possible light ray directions is modeled by two cones with the same apex and directions. We will see their explanation in exercise 3 and exercise 4.

2.2 Phong's illumination



The Phong illumination model relies on the decomposition of **light** and **material** colors into three components: ambient, diffuse and specular (ADS). It is an empiric model which does not relies on the complex physical behavior of the light, but it has the advantage to be quite easy to understand and to implement while offering a wide range of shadings.

- The ambient component L_a (light) or K_a (material) specifies a minimum brightness. Even if there is no light ray directly hitting a surfel, the ambient component will lit a little this surfel, preventing it from being completely dark. The ambient component is constant for all surfels. It could be understood as what remains from a light's color after rays have infinitely bounced in every directions.
- The diffuse component L_d (light) or K_d (material) is the most important one to give a 3D appearance to a surfel. It models the light's color received by a ray directly hitting the surfel.
- The specularity L_s (light) or K_s (material) is the shiny component of the light, i.e. the color of a shiny spot on the object.

The color of a surfel is also divided into this three components. However, instead of being vertex attributes, those components are the same for the whole object. Combined, those components form the **material** of such object. The material defines how a surfel will react to the ADS components of a light. Sometimes, the diffuse component of the material can be replaced by a vertex attribute: it depends on what you want to render.

2.3 Appearance computation

The appearance of a surfel of material K lit by a particular light L is computed from:

- the color components of the light L_a, L_d, L_s,
- the material properties of the object: K_a, K_d, K_s, and `shininess`,
- the position and the normal of the surfel `surfel_position`, `surfel_normal`,
- and the other light properties (more about that in the exercises).

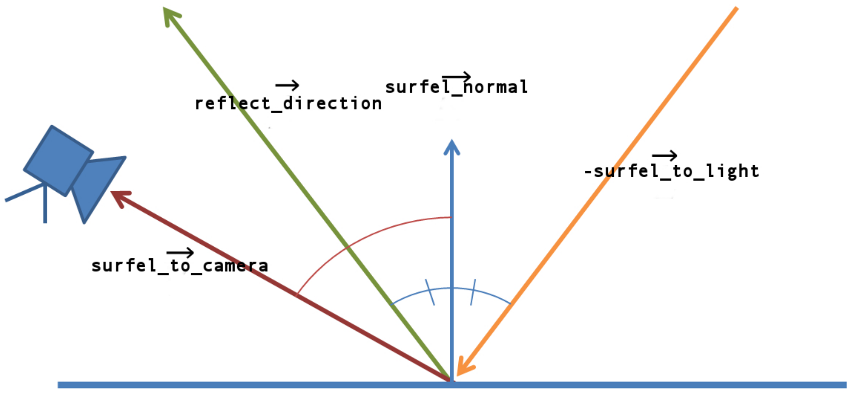
The appearance is computed component by component, then summed to obtain the contribution of a particular light to a surfel. When there are more than one light in the scene, the contributions of all lights are added. The result is the final appearance of a surfel, i.e. the color of its fragment.

Without taking into account distance attenuation and spot attenuation, the appearance computation for one light is given by:
`vec3 ambient` $= 1 \times L_a \times K_a$
`vec3 diffuse` $=$ diffuse_factor $\times L_d \times K_d$
`vec3 specular` $=$ specular_factor $\times L_s \times K_s$

The diffuse factor allows to decrease the diffuse component when the `surfel_to_light` and `surfel_normal` directions are too different. Indeed, when they are different, fewer photons will be reflected in the direction of the camera. The diffuse factor is then:
diffuse_factor=max($\overrightarrow{surfel\_normal} \cdot \overrightarrow{surfel\_to\_light}$,0);

The maximum function is here to keep the diffuse factor non negative. It is valid since when the dot product is negative, the ray arrives from under the surface. Thus, the ray will never hit the surface on the direction we are interested in (assuming the normals of surfels are pointing in such direction). Using minimum or maximum to keep the values in valid ranges is a common thing in Computer Graphics and is referred to as clamping.

The specularity determines the spread of the reflected light with respect to the shininess property:
specular_factor=pow(clamp($\overrightarrow{surfel\_to\_camera} \cdot \overrightarrow{reflect\_direction}$,0,1),shininess)
with
$\overrightarrow{reflect\_direction} = -\overrightarrow{surfel\_to\_light} - 2 \cdot \overrightarrow{surfel\_to\_light} \cdot \overrightarrow{surfel\_normal}$



$\overrightarrow{\text{reflect\_direction}}$
$\overrightarrow{\text{surfel\_normal}}$
$-\overrightarrow{\text{surfel\_to\_light}}$
$\overrightarrow{\text{surfel\_to\_camera}}$

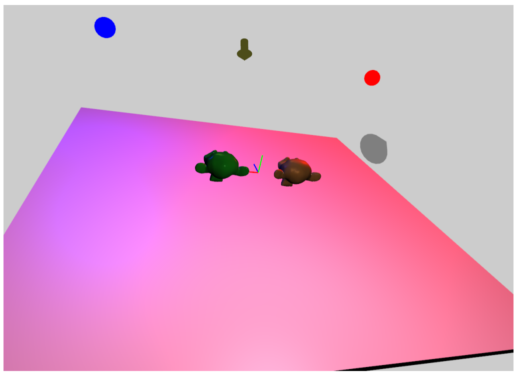
Again, we *clamp* the dot product value to avoid strange values (real power of a negative value) while remaining correct (if the dot product is negative, this means the camera is not in the cone of highly dense photons reflected by the surfel giving birth to a shiny spot). Also, numerical roundings can let a the result of a dot product slightly greater than 1 which is bad especially in the *pow* function.

Warning: In this practical we give the light the ADS components because historically, OpenGL implemented Phong illumination in this way in a `fixed pipeline`, before the arrival of shaders.

However, it is also common to give the light a single color because we almost always set light's ADS component to the "light color". This is also, relatively speaking, more realistic.

3 Exercise 1: Phong shading

- Start the main program and check that you get the expected result:



- In `phongFragment.glsl`, read carefully the `computeDirectionalLight()` function and make the parallel with the the equations of Phong.

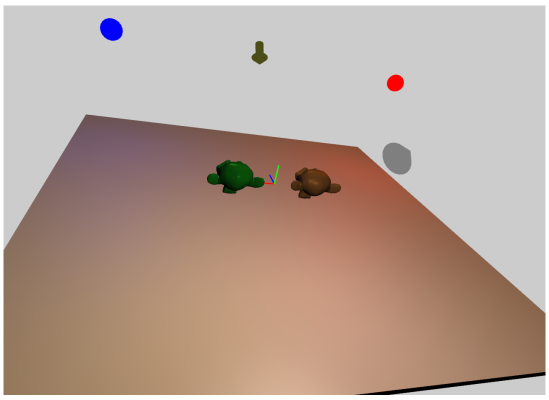
4 Exercise 2: Attenuation

To get a more realistic approximation of the light, one cheap improvement is to simulate its intensity decay through a medium. Commonly, we model this attenuation a thanks to three coefficients and the distance from the light to the surface of the lit object d:

$$a = \frac{1}{k_{constant} + k_{linear} \times d + k_{quadratic} \times d^2} \tag{1}$$

The three factors $k_{constant}$, k_{linear} and $k_{quadratic}$ are properties of a light, and are applied to the three illumination components computed of a surfel (ambient, diffuse and specular).

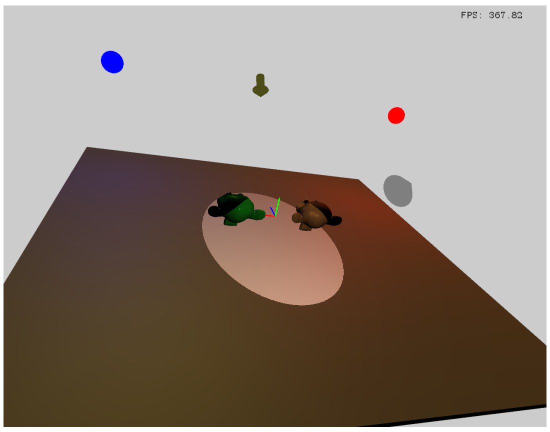
- In `phongFragment.glsl`, in the function `computePointLight()`, implement the aformentionned attenuation model and apply it to the light attributes.
- Do the same for the spot light.
- Check that you get a result similar to the one below.



5 Exercise 3: Spot light

A spot light cast light inside a cone of direction $\overrightarrow{spot\_direction}$ with an aperture θ. Whenever a surfel is outside this cone, it is not lit by the spot (i.e. the spot intensity is set to 0 for this surfel). This case occurs when:
$cos(\varphi) = \overrightarrow{surfel\_to\_light} \cdot (-\overrightarrow{spot\_direction}) < cos(\theta)$

- In `phongFragment.glsl`, modify the function `computeSpotLight()` so that it takes into account the cone aperture and computes correctly the spot intensity factor.
- Check that you get a result similar to the image below.

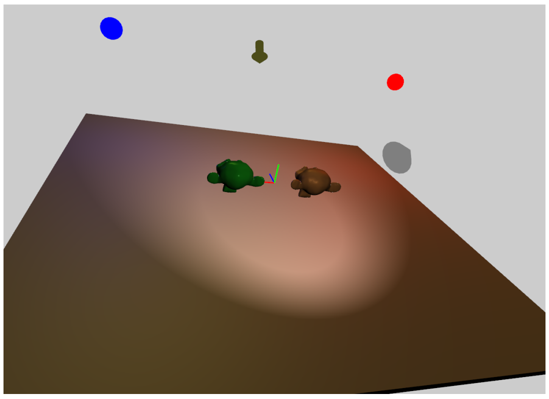


6 Exercise 4: Smooth spot edges

You can see that the spot light has hard edges that are not likely to appear in the real world. To increase the realism of a spot light, we will use two cones of light. The first cone, named the inner cone, is the same as before, with an aperture of θ_{inner}. The second cone is larger and referred to as the outer cone, with an aperture of θ_{outer}. When a surfel is between the inner and the outer cone, its intensity value will be between 0 and 1. If the surfel is inside the inner cone, its intensity is still 1, while when outside both cones, its intensity is set to 0. To compute the spot intensity, we have the following nice formula:

$$intensity = clamp(\frac{cos(\varphi) - cos(\theta_{outer})}{cos(\theta_{inner}) - cos(\theta_{outer})}, 0, 1) \tag{2}$$

- Implement this other version of the spot light intensity.
- Check that you get a result similar to the expected one.



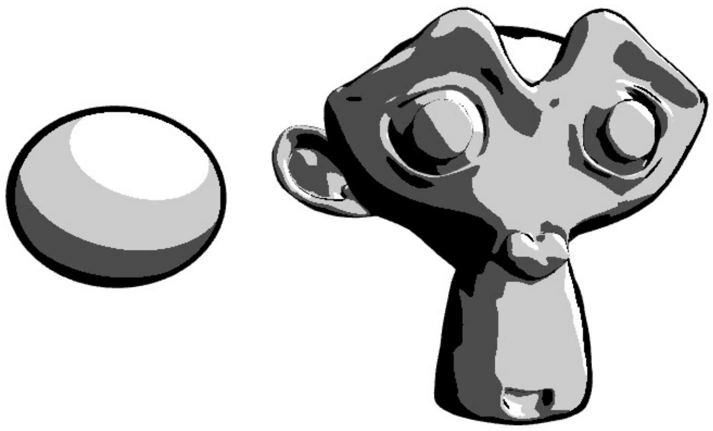
7 Exercise 5: Moving lights

You may have noticed that `Light` class inherits from `KeyframedHierarchicalRenderable`.

- What type of transformation matrix do you need to animate each light type ?
- Read the functions `lookAtModel()` and `lookAtUpModel()` from `Utils`. These functions might be useful for setting transforms of lights with a direction (direction light and spot light).
- Add keyframes to the lights of the scene (you can get inspiration from `demo_moving_lights.cpp`).
- `glm::lookAt()` is a useful function to create the view matrix of a camera standing at a position, looking at a particular point while having an "up" direction. Read `lookAtUp()` and `lookAtUpModel()` from `Utils`. What is the relationship between the view matrix and the camera's model matrix ?

8 Exercise 6: Project related

You can now play around with basic lighting in your North Pole environment. Lighting and materials are 90% responsible for the appearance of the virtual scene. Keep in mind that we implemented Phong illumination because it simulates a wide range of materials with few parameters, but there are a number of other shading models that you can implement. In particular, one can think of non-realistic shading such as cartoon shading. Shaders are frequently designed to match a specific object (*e.g.* shader of water surface).



Simple cartoon shading with diffuse factor quantization and edge darkening.