TP4 - Lights

Defining lights in OpenGL.

Computer Graphics
2nd year, Multimedia track

Session 4

Version: v2025.1-rc3
(master @ 0320ef2 2025-03-13)

Contents

1 Objective **1**
1.1 OpenGL references . . . 1

2 Preamble **2**
2.1 The normal vectors . . . 2
2.2 Hidden Surface Removal . . . 2

3 Lighting **3**
3.1 Lighting model in OpenGL . . . 4
3.1.1 Defining Lights in OpenGL . . . 5
3.2 Materials . . . 6
3.2.1 Defining Materials in OpenGL . . . 6
3.3 Shading . . . 7
3.4 Exercise . . . 7

1 Objective

In the previous TP sessions, you saw how to model an OpenGL scene: how to draw and place 3D objects, and how to define and place the camera. So far we neglected the rendering aspects and all the 3D objects have been rendered as wire-frames using points and lines or as colored faces. In this TP we will see how to improve the rendering of 3D objects to achieve a more photo-realistic effect. In particular, we will see how we can define and place lights inside a scene.

1.1 OpenGL references

Some useful resources about OpenGL in general and the topics we will cover in this TP:

- Chapter 5 of the *OpenGL Programming Guide*, covers lighting. You can find the online version at this address http://www.glprogramming.com/red/chapter05.html

- Chapter 9 of the *OpenGL Programming Guide*, covers texture mapping. You can find the online version at this address http://www.glprogramming.com/red/chapter09.html
- *OpenGL et GLUT: Une Introduction*, by Edmond Boyer (INRIA Grenoble-Rhône–Alpes), in French. http://www.ann.jussieu.fr/hecht/ftp/DEA/OpenGL.pdf
- OpenGL function documentation http://www.opengl.org/sdk/docs/man4/
- GLUT function documentation http://www.opengl.org/resources/libraries/glut/spec3/spec3.html
- Google it!... The easiest way to find the documentation of a function (or help) is to google the name of the function.

2 Preamble

In the next subsection, we briefly introduce some aspects that we will need to better understand lights in OpenGL.

2.1 The normal vectors

The lighting model computes the color associated with each vertex of a 3D object according to the normal vector associated with the vertex. When we define an object as a sequence of vertex, we can also specify the normal to associate to each vertex using the function `glNormal3f()`:

```
glBegin(GL_TRIANGLES);
  // first face of the triangle
  glNormal3f(.0, .0, 1.);
  glVertex3f(1., 1., 1.);
  glVertex3f(1., 2., 1.);
  glVertex3f(2., 2., 1.);
  // second face of the triangle
  glNormal3f(.0, 1., .0);
  ...
glEnd();
```

The three vertices of the first face of the triangle share the same normal vector $[0, 0, 1]$. In general, all the vertices declared between two calls of the function `glNormal3f()` have the same normal (OpenGL is a state machine).

Most of the high-level functions to generate 3D objects that we have seen so far (cones, spheres, cubes, teapots *etc.*) create the vertices and the relevant normal vectors.

2.2 Hidden Surface Removal

When you draw a scene composed of 3D objects, some of them might obscure all or parts of others. Changing your viewpoint can change the occluding relationship. For example, if you view the scene from the opposite direction, any object that was previously in front of another is now behind it. To draw a realistic scene, these occluding relationships must be maintained.

When we draw the scene we project each object on the image plane and we set the color for the relevant pixels. Therefore the order with which we render the objects is important because an object may overwrite some or all the pixels we already draw for another object that maybe was actually in front of him (*e.g.* see Figure 1.(a)).

One possible solution to this problem is to sort all the objects according to their distance from the camera and render them starting from the farthest to the closest. This method is

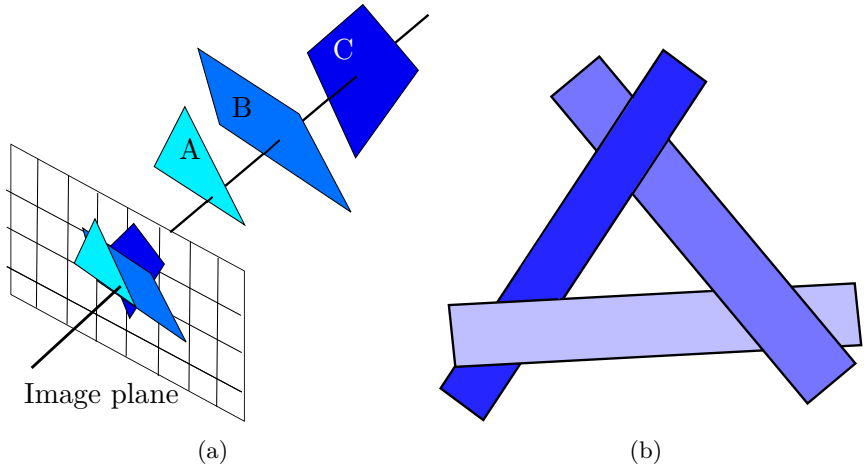


Figure 1: The hidden surface problem: in order to get the correct rendering of the scene, object C must be rendered before object B, which, in turn, must be rendered before object A. On the other hand, rendering the objects from the farthest to the closest cannot solve all the ambiguities (b).

computationally expensive and its complexity depends on the number of polygons. Moreover, it cannot deal with surfaces that are mutually hiding each other, such as the one in Figure 1.(b)

OpenGL provides two other methods for dealing with the hidden surface problem: the *back-face culling* and the *z-buffering*. Back-face culling is a method that eliminates all the faces not visible from the camera's point of view. The simplest way to verify whether a face (a triangle, a polygon *etc.*) has to be eliminated is to check the normal associated with the face: if the normal is directed "towards" the camera then the face could be visible (but, yet, it could be still hidden by another one), otherwise it can be safely removed from the rendering because it is not visible from the camera. We can activate this algorithm with `glEnable(GL_CULL_FACE)`.

The z-buffer is instead a matrix of the same size as the image to render: each element of the matrix contains information about the depth of the corresponding pixel in the image. Every time a point is projected into a pixel, the distance of the point from the camera is tested against the value stored in the z-buffer: if the distance of the point is lower then the point is drawn on the image, and the corresponding value in the z-buffer is updated with the distance of current point. Otherwise, the point is not drawn. The ambiguities of Figure 1.(b) can be resolved using this method. To activate the z-buffer:

- we need to properly set up the glut window `glutInitDisplayMode( GLUT_RGB | GLUT_DEPTH);`, where "`|`" means that we want both the RGB display and the z-buffer;
- `glEnable(GL_DEPTH_TEST)` to enable the test (it is enough to call it once);
- Every time we draw a new scene we need to clear the buffer with `glClear(GL_DEPTH_BUFFER_BIT)` (typically in the `display` function).

3 Lighting

When you look at a physical surface, your perception of the color depends on the distribution of photon energies reaching your eye. These photons come from a light source (or combination of light sources), some are absorbed and some are reflected by the surface. In addition, different

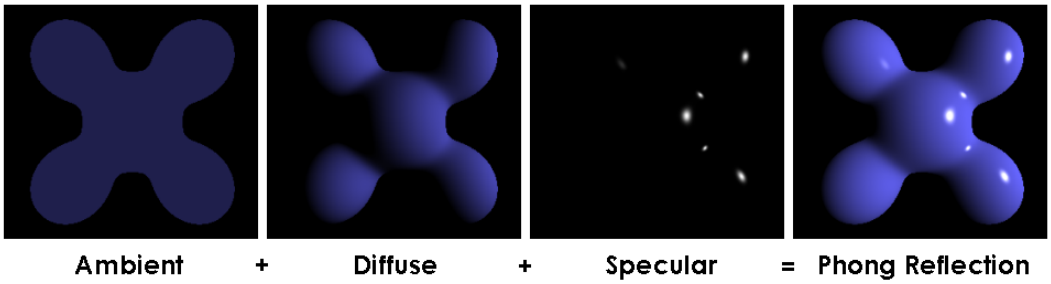


Figure 2: The three components of the Phong lighting model [image from Wikipedia].

surfaces may have very different properties - some are shiny and preferentially reflect light in certain directions, while others scatter incoming light equally in all directions.

3.1 Lighting model in OpenGL

In the OpenGL lighting model, the light in a scene comes from several light sources that can be individually turned on and off. Some light comes from a specific direction or position, and some light is generally scattered throughout the scene. For example, when you turn on a light bulb in a room, most of the light comes from the bulb, but some light comes after bouncing off one, two, three, or more walls. This bounced light (called ambient light) is assumed to be so scattered that there is no way to tell its original direction, but it disappears when a particular light source is turned off.

OpenGL adopts the Phong lighting model. The Phong reflection is an empirical model of local illumination. It describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. In particular, it assumes that the light is composed of three components: ambient light, diffuse reflection, and specular reflection. All three components are computed independently and then added together (see Figure 2).

- **ambient light**: it is a background light that accounts for the small amount of light scattered about the entire scene, *i.e.* that has been reflected multiple times in the scene.
- **specular reflection**: it describes the reflection of the light hitting a surface according to the classical reflection law. That is, the direction of incoming light and the direction of outgoing reflected light are at the same angle to the surface normal, and the incoming light, the reflected light, and the surface normal lie in the same plane. Thus, the contribution of specular reflection to the vertex color depends on the viewpoint.
- **diffuse reflection**: It is the light coming from one direction. Therefore, it is brighter when it hits a surface perpendicularly than when it hits the surface at a lower angle.
- Once it hits a surface, however, it is scattered equally in all directions, so it appears equally bright, no matter where the eye is located.

OpenGL approximates light and lighting as if light can be broken into red, green, and blue components. Thus, the color of light sources is characterized by the amount of red, green, and blue light they emit, and the material of surfaces is characterized by the percentage of the incoming red, green, and blue components reflected in various directions.

3.1.1 Defining Lights in OpenGL

Let's see how to set up the lights with this sample code:

```
1  GLfloat light_ambient[] = { .0, .0, .0, 1. };   // the ambient component
2  GLfloat light_diffuse[] = { 1., 1., 1., 1. };   // the diffuse component
3  GLfloat light_specular[] = { 1., 1., 1., 1. };  // the specular component
4  GLfloat light_position[] = { 1., 1., 1., .0 };  // the light position
5
6  // set the components to the first light
7  glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
8  glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
9  glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
10 glLightfv(GL_LIGHT0, GL_POSITION, light_position);
11
12 // activate lighting effects
13 glEnable(GL_LIGHTING);
14 // turn on the first light
15 glEnable(GL_LIGHT0);
```

- The first 3 vectors specify the color for each of the components of the light (ambient, diffuse, and specular). Each vector specifies the RGBA values for the color
- the forth vector `light_position` contains the position of the light expressed in (homogeneous) Cartesian coordinates. As you can see, in this case, the forth coordinate is set to zero: this means that the light is placed at the infinity and the other three coordinates indicate the light direction (*directional light*). Remember: lights are subject to the ModelView matrix, just like the other objects in the scene. So it's very important where you put the light position in the code!
- the next bunch of calls to `glLightfv` assigns the previously declared lights components to `GL_LIGHT0`: OpenGL allows to declare up to 8 different lights using `GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7`. `glLightfv` always takes as parameters the light to modify, the component we are setting, and the relevant data (here the doc). Once again, lights are part of the OpenGL's state machine, hence once the value is set it will hold until it is further changed.
- `glEnable(GL_LIGHTING)` enable the light management. To disable the lightings we can use `glDisable(GL_LIGHTING)`
- `glEnable(GL_LIGHT0)` turns the `GL_LIGHT0` on. `glDisable(GL_LIGHT0)` turns it off

Try it! You can try to add the lights to your `helloteapot2.cpp` (the one in which the viewpoint was changed and the teapot is drawn as solid):

- just copy-paste the above piece of code in the main just before the `glutMainLoop()`
- we need the z-buffer so remember to change the code accordingly (*c.f.* Section 2.2):
  - `glutInitDisplayMode( GLUT_RGB | GLUT_DEPTH);` in `main`
  - `glEnable(GL_DEPTH_TEST);` in `main`
  - `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );` in `display`

[git] • Branch: master @ 0320ef2 • Release: v2025.1-rc3 (2025-03-13)
Head tags: v2025.1-rc3

3.2 Materials

In the OpenGL model, the light sources have an effect only when there are surfaces that absorb and reflect light. Each surface is assumed to be composed of a material with various properties. A material can emit its light (like the headlights on a car), it can scatter some of the incoming light in all directions, and it can reflect some of the incoming light in a preferred direction, like a mirror or other shiny surface.

The OpenGL light model makes the approximation that the color of a material is a function of the percentages of incoming red, green, and blue light that it reflects. For example, a perfectly red ball reflects all the incoming red light and absorbs all the green and blue light that strikes it. If you view such a ball in white light (composed of equal amounts of red, green, and blue light), all the red is reflected, and you see a red ball. If the ball is viewed in pure red light, it also appears as red. If, however, the red ball is viewed in pure green light, it appears black (all the green is absorbed, and there's no incoming red, so no light is reflected).

Like lights, materials have different ambient, diffuse, and specular colors. These colors determine the ambient, diffuse, and specular reflectance of the material. A material's ambient reflectance is combined with the ambient component of each incoming light source, the diffuse reflectance with the light's diffuse component, and similarly for the specular reflectance and component. Ambient and diffuse reflectances define the color of the material and are typically similar if not identical. Specular reflectance is usually white or gray so that specular highlights end up being the color of the light source's specular intensity. If you think of a white light shining on a shiny red plastic sphere, most of the sphere appears red, but the shiny highlight is white. [1]

3.2.1 Defining Materials in OpenGL

The materials are defined in a similar way as the lights with the function `glMaterialfv` (here the doc). Let's see a simple example:

```
1 GLfloat mat_specular[] = { 1., 1., 1., 1. };      // define the specular component
2 GLfloat mat_shininess[] = { 50. };                  // define the shininess
3 // assign the material property
4 glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
5 glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
6 // draw the teapot
7 glutSolidTeapot(.5);
```

`glMaterialfv` takes the same parameters as `glLightfv`, except for the first one that indicates which face of the object the material property is to apply. It can be set to the front face (`GL_FRONT`), the back face (`GL_BACK`), or both of them(`GL_FRONT_AND_BACK`). Once more again, the material is part of the OpenGL's state machine, once the values are set they hold until they are further changed. This means that if we add a second teapot to the code above it will be drawn with the same property of the current one, unless we first call again `glLightfv` to set new properties.

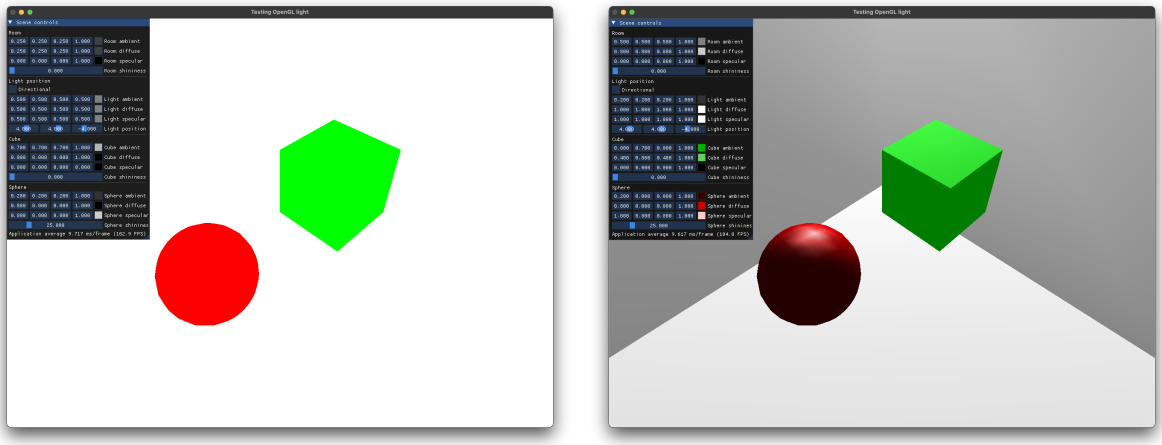
[1] In addition to ambient, diffuse, and specular colors, materials have an emissive color, which simulates light originating from an object. In the OpenGL lighting model, the emissive color of a surface adds intensity to the object but is unaffected by any light sources. Also, the emissive color does not introduce any additional light into the overall scene.

[git] • Branch: master @ 0320ef2 • Release: v2025.1-rc3 (2025-03-13)
Head tags: v2025.1-rc3

3.3 Shading

Once the color it's determined for each vertex of a face (a triangle, a polygon *etc.*), the color of the face is computed by linear interpolation of the color of the relevant vertex. This allows to have a smooth shading which is called **Gouraud shading**. In OpenGL we can activate the smooth shading calling `glShadeModel(GL_SMOOTH)`. A simpler shading model can be used by passing `GL_FLAT`. This will draw the face with a single color chosen from one of the vertices that compose the face.

3.4 Exercise



(a) (b)

Figure 3: The same scene rendered without light (a) and with light and materials defined for each object (b).

For this exercise, we will work with the program in `lumiere.cpp`. If you compile the program and run it, you should see something like in Figure 3.(a): a white room with a red sphere and a green cube. The objects are drawn using the `glColor` function. This is a simplified version of the Cornell box, a famous scene used to test the rendering algorithms.

The goal of the exercise is to introduce the light in the scene and the materials for the objects so that we can have a more photo-realistic rendering such as the one depicted in Figure 3.(b). The controls on the menu will help you select the values for each component of the light and material once you have activated the light in the scene and defined the material for each object.

Here are the steps to follow:

1. You can move around the scene using the arrow keys and zoom in/out using the page up/down keys.

2. To better understand how to define the normal vector for each vertex look at the code of the function `glRoom()`.

3. Look at the beginning of the file, some global variables define the light and the material for the objects. These are meant to group the properties of each object. You can see the definition of the relevant data structure in `lumiere.hpp`:

[git] • Branch: master @ 0320ef2 • Release: v2025.1-rc3 (2025-03-13)
Head tags: v2025.1-rc3

- `Material` groups all the properties of the material of an object: the ambient, diffuse, and specular colors, and the shininess.
- `Light` groups all the properties of the light: the ambient, diffuse, and specular colors, and the position.

Both of them rely on the `vec4f` structure that is simply an array of 4 floats.

4. Activate the lighting and place a light in one of the corners of the room: complete the function `place_light(Light& light)` that is used to define an OpenGL light based on the properties of `light`.
   - To visualize the position of the light (just for debugging) in the scene you can draw a yellow point in the position of the light.
   - What do you need to do if you want to use `glColor` to draw the yellow point? Think about the state machine of OpenGL ...
5. Where are the colors??? Since the light has been enabled we need to define the colors of the objects with the materials. Take a look at the function `define_material(const Material& mat)`.
   Replace each call to `glColor` before the objects with a proper call to `define_material()` using the associated global material variable.
   - complete the `init` function to enable the Gouraud shading, the back-face culling, and the depth test.
6. Everything is gray now. With the help of the controls in the menu, play with the different values of the light and the materials to understand how they work. You can have a look here to see some examples of real materials and their values for the different components.
7. Once you find the proper values for each object, **hardcode them in the definition of the global variables**.
8. Remember how can we define a directional light? You can use the boolean field `directional` of the `Light` structure to properly set up the light. The checkbox in the menu sets its value. You can also set the key D (see the `keyboard() function`) to change the light property so that it becomes a directional light.

Advanced The location of the viewpoint affects the calculations for highlights produced by specular reflectance. The intensity of the highlight at a particular vertex depends on the normal at that vertex, the direction from the vertex to the light source, and the direction from the vertex to the viewpoint. A local viewpoint tends to give more realistic results, but since the direction from the vertex to the viewpoint has to be calculated for each vertex, the overall performance is degraded with a local viewpoint. By default, OpenGL assumes an infinite viewpoint, so for each vertex, the direction viewpoint-vertex is constant and the same for all vertices. You can change this settings with `glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, [GL_TRUE|GL_FALSE]);`. Use a key of the keyboard to switch from true to false and vice-versa, and compare the effect.

[git] • Branch: master @ 0320ef2 • Release: v2025.1-rc3 (2025-03-13)
Head tags: v2025.1-rc3