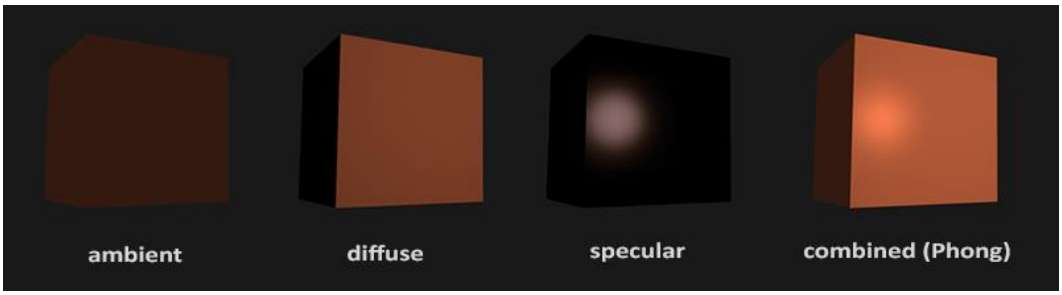
Lab5. Adding lights

Introduction

Lighting in OpenGL is based on approximations of reality using **simplified models** that are much easier to process and look relatively similar. These lighting models are based on the physics of light as we understand it. One of those models is called the **Phong lighting model**. The major building blocks of the Phong lighting model consist of **three components**: ambient, diffuse and specular lighting. Below you can see what these lighting components look like on their own and combined:



- Ambient lighting: even when it is dark there is usually still some light somewhere in the world (the moon, a distant light) so objects are almost never completely dark. To simulate this, we use an ambient lighting constant that always gives the object some color.
- Diffuse lighting: simulates the **directional impact** of a light object on an object. This is the most visually significant component of the lighting model. The more a part of an object faces the light source, the brighter it becomes.
- Specular lighting: simulates the **bright spot of a light** that appears on **shiny objects**. Specular highlights are more inclined to the color of the light than the color of the object.

To create visually interesting scenes, we want at least to simulate these 3 lighting components.

N.B. There is a fourth component of the lighting model called emissive lighting that simulates light originating from an object (but we will not take it into account).

In OpenGL, a **material** is defined a set of coefficients that define how the lighting model interacts with a surface. In particular, ambient, diffuse, and specular coefficients for each color component (R,G,B) are defined and applied to a surface and effectively multiplied by the amount of light of each kind/color that strikes the surface.

Note that we can have **several lights in the same scene**, each of them having a name predefined by OpenGL (for example `GL_LIGHT0`, `GL_LIGHT1`, `GL_LIGHT2`, etc.)

OpenGL commands related to lightning

Commands to set light intensities:

We can set the light intensities for a given light (e.g. `GL_LIGHT0`) using the `glLightfv` function that takes as arguments (1) the name of the light (light), (2) the parameter name to set (pname) and (3) its corresponding value(s) (params).

Prototype: `glLightfv(light, pname, * params)`

Some of the parameters that we will use are:

- GL_AMBIENT: params contains four floating-point values that specify the ambient RGBA intensity of the light. The initial ambient light intensity is (0, 0, 0, 1).
- GL_DIFFUSE: params contains four floating-point values that specify the diffuse RGBA intensity of the light. The initial value for GL_LIGHT0 is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 1)
- GL_SPECULAR: params contains four floating-point values that specify the specular RGBA intensity of the light. The initial value for GL_LIGHT0 is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 1).
- GL_SHININESS: params is a single value that specifies the RGBA specular exponent of the material. Only values in the range [0, 128] are accepted. The initial specular exponent is 0.
- GL_POSITION: params contains four values that specify the position of the light in homogeneous object coordinates.
- GL_SPOT_DIRECTION: params contains three values that specify the direction of the light in homogeneous object coordinates.

N.B.1. After setting the light parameters using `glLightfv`, we must tell OpenGL to use the them to compute the vertex color (`glEnable(GL_LIGHTING)`), then by including the light no. i (`GL_LIGHT0`) in the evaluation of the lighting equation (`glEnable(GL_LIGHT0)`).

N.B.2. For details about homogeneous coordinates see https://yasenh.github.io/post/homogeneous-coordinates/

A complete example of setting light intensities is:

```
# Setup light 0 and enable lighting
glLightfv(GL_LIGHT0, GL_AMBIENT, GLfloat_4(0.0, 1.0, 0.0, 1.0))
glLightfv(GL_LIGHT0, GL_DIFFUSE, GLfloat_4(1.0, 1.0, 1.0, 1.0))
glLightfv(GL_LIGHT0, GL_SPECULAR, GLfloat_4(1.0, 1.0, 1.0, 1.0))
glLightfv(GL_LIGHT0, GL_POSITION, GLfloat_4(1.0, 1.0, 1.0, 0.0));
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, GLfloat_4(0.2, 0.2, 0.2, 1.0))
glEnable(GL_LIGHTING)
glEnable(GL_LIGHT0)
```

Setting material colors:

The glMaterialfv function specifies material parameters for the lighting model. This function takes as arguments:

(1) The face or faces that are being updated (must be one of the following: GL_FRONT, GL_BACK, or GL_FRONT and GL_BACK),
(2) The material parameter of the face or faces being updated (pname). The main parameters that can be specified using glMaterialfv, are: GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION and GL_SHININESS.
(3) its corresponding value(s) (params).

Prototype: `glMaterialfv(face, GLenum  pname,*params)`

A complete example of setting light intensities is:

```
# Setup material for the shape to draw
glMaterialfv(GL_FRONT, GL_AMBIENT, GLfloat_4(0.2, 0.2, 0.2, 1.0))
glMaterialfv(GL_FRONT, GL_DIFFUSE, GLfloat_4(0.8, 0.8, 0.8, 1.0))
glMaterialfv(GL_FRONT, GL_SPECULAR, GLfloat_4(1.0, 0.0, 1.0, 1.0))
glMaterialfv(GL_FRONT, GL_SHININESS, GLfloat(50.0))
```

Depth buffers and depth testing

A depth buffer, also known as a z-buffer, is a type of data buffer used in computer graphics to **represent depth information** of objects in 3D space from a particular perspective. Depth buffers are an aid to rendering a scene to **ensure that the correct shapes/objects properly occlude other shapes/objects**. When depth testing is enabled, OpenGL tests the depth value of a fragment against the content of the depth buffer. If this test passes, the fragment is rendered and the depth buffer is updated with the new depth value. If the depth test fails, the fragment is discarded.

To enable depth testing in OpenGL we must use the function call `glEnable(GL_DEPTH_TEST)`, then we must precise the function used to compare each incoming pixel depth value with the depth value present in the depth buffer using the `glDepthFunc` function. This function takes as argument the condition under which the pixel will be drawn. For example, the condition GL_LESS passes if the incoming depth value is less than the stored depth value.

Details about all the possible conditions can be found at https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDepthFunc.xhtml.

A complete example of setting depth testing is:

```
# Setup depth testing
glEnable(GL_DEPTH_TEST)
glDepthFunc(GL_LESS)
```

N.B.1. In order to be able to apply the depth buffer in the current window we must initialize OpenGl rendering context with depth buffer by setting the corresponding Bit mask (GLUT_DEPTH) in the glutInitDisplayMode function call.

Example: `glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);`

N.B.2. Remember that setting all the parameters/transformations/conditions/etc. to apply on a scene should be performed before drawing the shapes/objects of which the scene is composed.

Summary

The following "to-do" list summarizes the steps needed to display a scene with lighting effect (e.g. the tasks to execute in the display callback function):

1. Set the background of the window
   `glClearColor(0.5, 0.5, 0.5, 0)`
2. Set the Bit Masks
   `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`
3. Establish the projection matrix (perspective)
   `glMatrixMode(GL_PROJECTION)`
4. Reset the matrix back to its default state to ensure that each time when we enter the projection mode, the matrix will be reset to identity matrix, so that the new viewing parameters are not combined with the previous one.
   `glLoadIdentity()`
5. Set up a perspective projection matrix. We may need to get automatically the values of the window's width and height to set the aspect ratio of the perspective projection. This can be done by reading the viewport parameters (GL_VIEWPORT).
   ```
   _,_,width,height = glGetDoublev(GL_VIEWPORT)
   gluPerspective(45,width/height,0.25,200)
   ```
6. Create the viewing matrix
   `gluLookAt(0,1,5,0,0,0,0,1,0)`
7. Set up the lights parameters and enable lightning
8. Set up depth testing (if enabled)
9. Set up materials for the shape(s) to draw
10. Perform transformations (if any)
11. Draw shapes

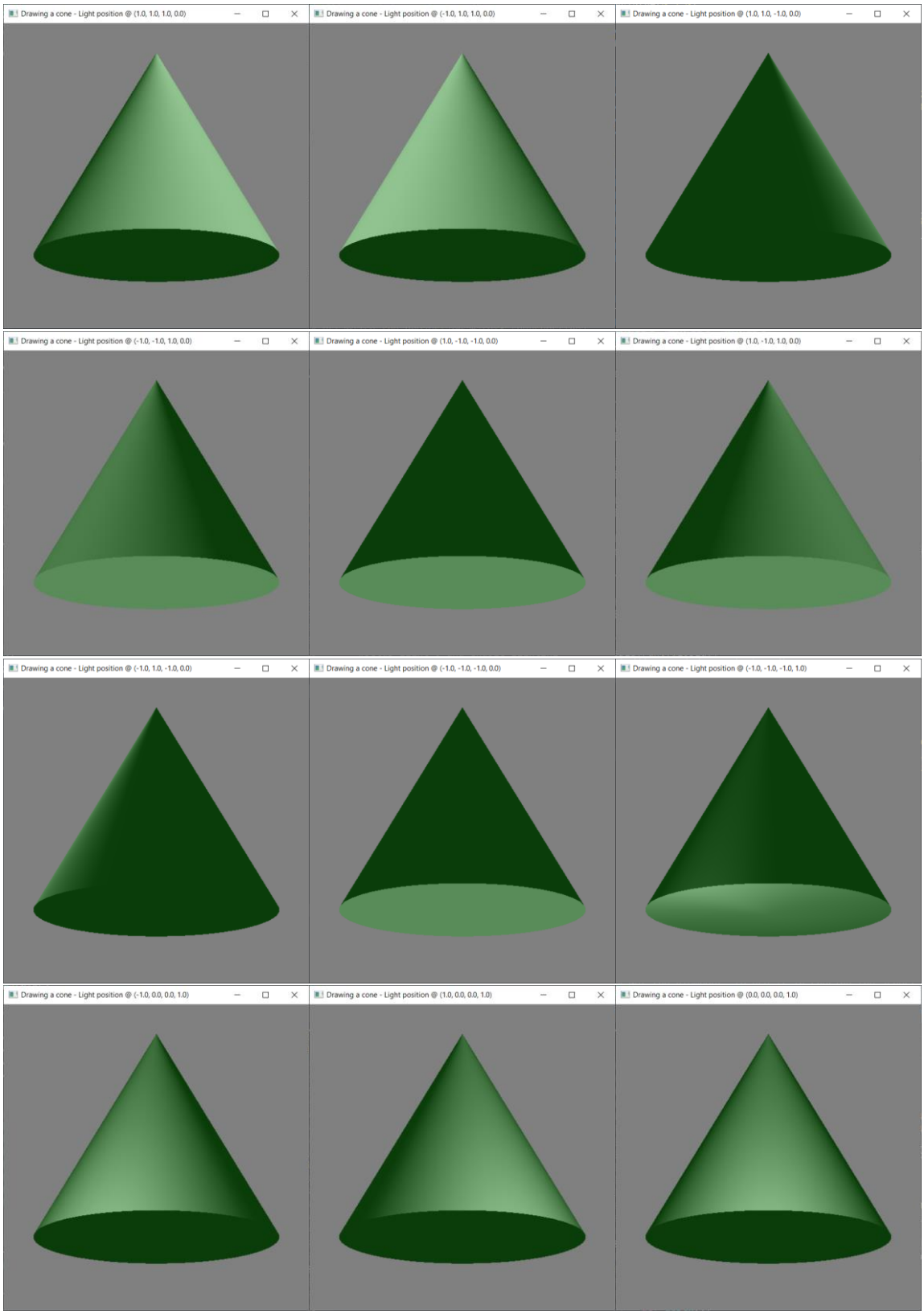
Of course, we can add all the previous steps in the display callback function but, for teaching purpose and for better modular organization of the processing, we will write the code corresponding to each step as a function, then we call all the functions from within the display callback function.

A complete example of the modular code that renders a 3D solid cone with lightning effects is given below:

```
# Program that draws a solid Cone with lights

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *

def background():
    # Set the background color of the window to Gray
    glClearColor(0.5, 0.5, 0.5, 0)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

def perspective():
    # establish the projection matrix (perspective)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    # Get the viewport  to use it in chosing the aspect ratio of
gluPerspective
    _,_,width,height = glGetDoublev(GL_VIEWPORT) # we don't need x and y
    gluPerspective(45,width/height,0.25,200)

def lookat():
    # and then the model view matrix
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    gluLookAt(0,0,4,0,0,0,0,1,0)

def light():
    #Setup light 0 and enable lighting
    glLightfv(GL_LIGHT0, GL_AMBIENT, GLfloat_4(0.0, 1.0, 0.0, 1.0))
    glLightfv(GL_LIGHT0, GL_DIFFUSE, GLfloat_4(1.0, 1.0, 1.0, 1.0))
    glLightfv(GL_LIGHT0, GL_SPECULAR, GLfloat_4(1.0, 1.0, 1.0, 1.0))
    glLightfv(GL_LIGHT0, GL_POSITION, GLfloat_4(1.0, 1.0, 1.0, 0.0));
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, GLfloat_4(0.2, 0.2, 0.2, 1.0))
    glEnable(GL_LIGHTING)
    glEnable(GL_LIGHT0)

def depth():
    #Setup depth testing
    glEnable(GL_DEPTH_TEST)
    glDepthFunc(GL_LESS)

def coneMaterial():
    #Setup material for cone
    glMaterialfv(GL_FRONT, GL_AMBIENT, GLfloat_4(0.2, 0.2, 0.2, 1.0))
    glMaterialfv(GL_FRONT, GL_DIFFUSE, GLfloat_4(0.8, 0.8, 0.8, 1.0))
    glMaterialfv(GL_FRONT, GL_SPECULAR, GLfloat_4(1.0, 0.0, 1.0, 1.0))
    glMaterialfv(GL_FRONT, GL_SHININESS, GLfloat(50.0))

def transformations():
    pass

def drawCone(radius, height, slices,stacks):
    glPushMatrix()
    glutSolidCone(radius, height, slices, stacks )
    glPopMatrix()
```

```
def display():
    background()
    perspective()
    lookat()
    light()
    depth()
    coneMaterial()
    transformations()
    drawCone(1,2,50,10)
    glutSwapBuffers()

# Initialize GLUT
glutInit()

# Initialize the window with double buffering and RGB colors
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)

# Set the window size to 500x500 pixels
glutInitWindowSize(500, 500)

# Create the window and give it a title
glutCreateWindow("Drawing a 3D cone with lights")

glClearColor(0.0,0.0,0.0,0.0)

# Set the initial window position to (50, 50)
glutInitWindowPosition(50, 50)

# Define display callback
glutDisplayFunc(display)

# Begin event loop
glutMainLoop()
```

Effects of changing the position of the lights



Exercise:

- Modify the code that draws the cone with lighting effect (File cone_template.py) to allow to see a face view of it (like in the previous set of images).
- Add some animations (e.g. rotation).
- Add more lights to the scene (at least one).
- The control of the animation **and** the lights should be done using keyboard.
- The parameters of the animation (transformations) should be set at the beginning of the code (translation step, rotation angle, scaling factor, light position, shininess level).