ENSIM, 5A INFO IPS

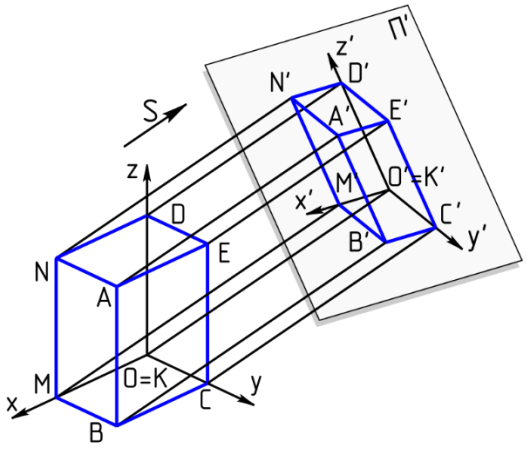
Image Synthesis, Lab3

****

Part I: Projections

3.1.1 Introduction

A 3D projection (or graphical projection) is a design technique used to display a three-dimensional (3D) object on a two-dimensional (2D) surface.
In practice, the projection operation consists of a mathematical transformation applied on the components of a 3D objects (e.g. the points of which it's composed) in order to map them on a 2D plane then eventually connect them together to create a visual element (the projection).

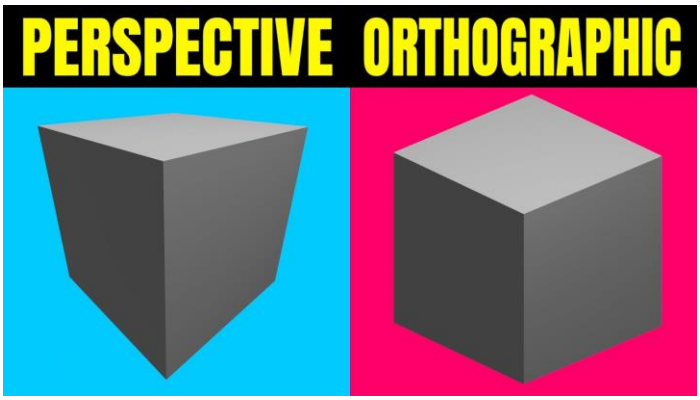


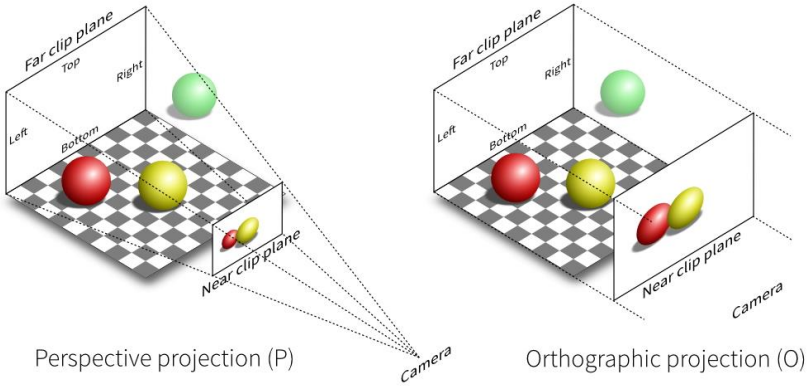
OpenGL offers two general classes of projection transformations: orthographic (parallel) and perspective.

Orthographic Projections in OpenGL

Orthographic, or parallel projections consist of those that involve **no perspective correction** (i.e. to correct converging verticals in pictures of buildings and sometimes to fix horizontal convergence too). This means that there is **no adjustment for distance** from the camera to be made in these projections, and thus, objects on the screen will appear the **same size** no matter how close or far away they are.
To setup this type of projection we use the OpenGL provided glOrtho() function.

glOrtho(left, right, bottom, top, near, far)





Where left and right specify the x-coordinate **clipping planes**, bottom and top specify the y-coordinate clipping planes, and near and far specify the distance between the camera or the viewer's eye and the z-coordinate clipping planes. Together these six coordinates provide a **box-shaped viewing volume**.

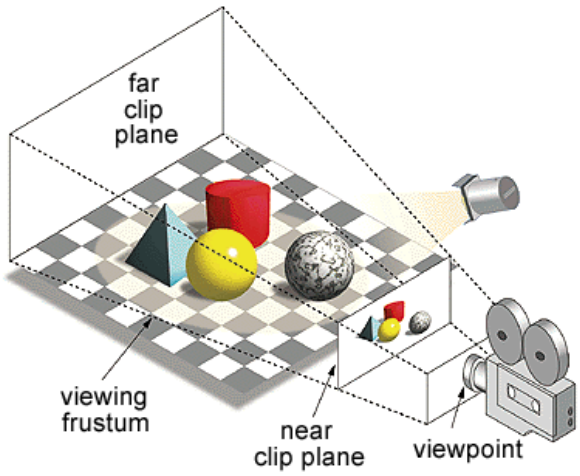
Perspective Projections in OpenGL

Perspective projections create **more realistic looking scenes**, so that's what we will most likely be using most often in our course.

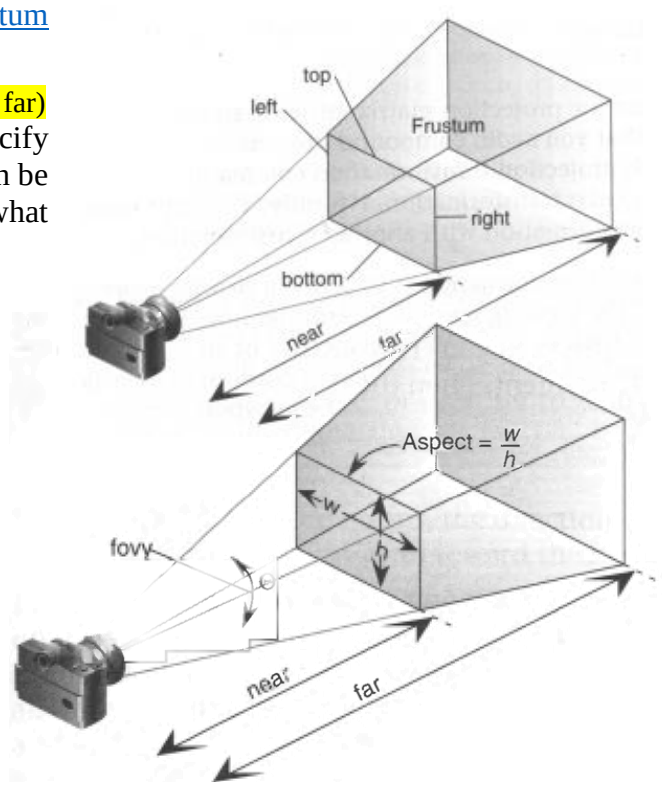
In perspective projections, as an object gets farther from the viewer/camera it will appear smaller on the screen- an effect often referred to as **foreshortening**.

The viewing volume for a perspective projection is a **frustum** (instead of the box-shaped volume of orthographic projections). A frustum looks like a pyramid with the top cut off, with the **narrow end toward the user** (camera).

To setup the view frustum, and thus the perspective projection, we have two



possible choices in OpenGL: glFrustum and gluPerspective.



glFrustum(left, right, bottom, top, near, far)
Using glFrustum enables us to specify an **asymmetrical** frustum, which can be useful in some instances, but isn't what we typically want to do.

The gluPerspective function specifies a **symmetrical** viewing frustum into the world coordinate system. fov (**field of view**) specifies, in degrees, the angle ***in the y direction*** that is visible to the user; aspect is the **aspect ratio** of the 2D scene, which is width divided by the height. This will determine the field of view ***in the x direction***. near and far specify the distance to the z-coordinate clipping planes (***always positive***).

gluPerspective(fov, aspect, near, far)

In order to view the perspective/orthographic projection from the beginning of the drawing process, it is preferable to put the call to the function gluPerspective/glOrtho in the **reshape callback** function (triggered when a window is reshaped or immediately before a window's first display callback after a window is created or whenever an overlay for the window is established). Recall that the reshape callback must be set/registered in the main function using the glutReshapeFunc function.

glutReshapeFunc(reshape)

On the other hand, **before** calling the gluPerspective function, we must set the current matrix mode to GL_PROJECTION so that OpenGL switches to the projection matrix. This can be done using the glMatrixMode function.

glMatrixMode(GL_PROJECTION)

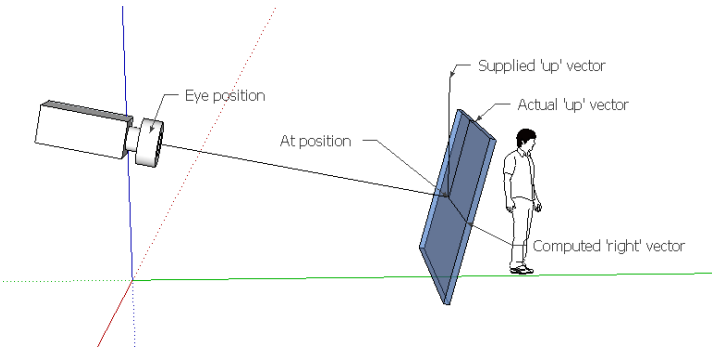
After glMatrixMode, we must call the function glLoadIdentity() in order to set the matrix at the top of the stack in the matrix stack as the identity matrix, so that any previous transformation will not affect the subsequent changes.

glLoadIdentity()

The **position of the virtual camera** i.e. the "HOW" and from "WHERE" to view the scene are configured using the function gluLookAt that sets an **eye point**, a **reference point** indicating the center of the scene, and an **up vector** to precise the up direction of the scene.

gluLookAt(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)

The default values of the arguments of the function gluLookAt are (0, 0, 0, 0, 0, -1, 0, 1, 0). These values may not be suitable for a given scenario so a call to the function gluLookAt must be made with the appropriate arguments.



Finally, we must specify the area of the window where the drawing region should be put into (i.e. the **visible area of the scene**). This is done calling the glViewport function that fixes the bottom left corner of the drawing region (coordinates argument (x, y)), and the number of pixels the drawing region should go in each direction (dimensions arguments (width, height)).

glViewport(x, y, width, height)

Mathematically speaking, glViewport specifies the affine transformation of x and y **from normalized device coordinates to window coordinates**. Let x_{nd} and y_{nd} be the normalized device coordinates. Then the window coordinates x_w and y_w are computed as follows:

$$x_w = x_{nd} + 1\frac{width}{2} + x$$

$$y_w = y_{nd} + 1\frac{height}{2} + y$$

For example, glViewport(0,0,width,height) call tells OpenGL to map the lower left corner of the projected image to the coordinate (0,0) of the OpenGL window; It also specifies that the upper right corner of the projected image is mapped to the coordinate (width, height) in window space; this is the upper right corner of the OpenGL window.

An example of drawing a 3D wireframe teapot with orthographic projection is the following:

```
def display():
    # Reset background
    glClear(GL_COLOR_BUFFER_BIT)

    # Render scene
    render_Scene()

    # Swap buffers
    glutSwapBuffers()

def render_Scene():      # Scene render function
    # Draw a wireframe red tea pot
    glColor3f(1.0,0.0,0.0);
    glutWireTeapot(0.5)

def reshape(x,y):
    glLoadIdentity()
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.5, 1.5)
    gluLookAt(0.0, 0.0, 0.0, 0.0, 1.0, -1.0, 0.0, 1.0, 0.0)
    glViewport(0, 0, x, y)
```

N.B. Don't forget to add the following line to precise the name of reshaping callback function:

glutReshapeFunc(reshape)

Practice:

1. Try to modify the parameters of the function glOrtho and note the effect of each of them on the rendered scene.
2. Try to modify separately each of the three triplets parameters of the function gluLookAt and note the effect of each of them on the rendered scene.
3. Try to modify the parameters of the function glViewport and note the effect of each of them on the rendered scene.

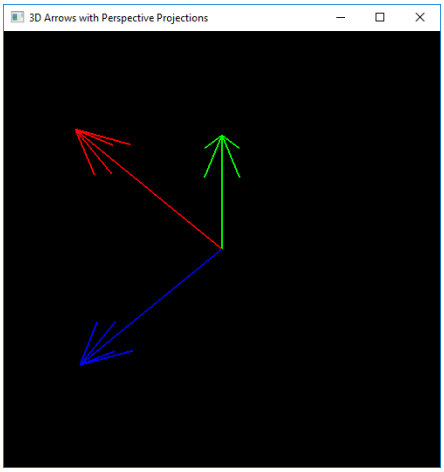
If we want to change, in the previous example, the projection type from orthographic to perspective we must follow the steps mentioned above. This yields the reshape function given below:

```
def reshape(x,y):
    # Set a new projection matrix
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(20.0,x/y,4.5,20.0);
    gluLookAt(2,-4,-2,0,0,0,0,1,0);
    glViewport(0,0,x,y);  # Use the whole window for rendering
```

1. Try to tune the parameters of the function gluPerspective and note the effect of each of them on the final view of the scene.
2. Test the previous code with a solid version of the rendered object. What is missing to have a realistic view of the scene (object)?

Practice:

- Draw three arrows (red, green and blue) in the directions of x, y and z –axis respectively (the arrows start at the origin (0,0,0) and end at (length,0,0), (0,length,0) and (0,0,length) respectively where length is a parameter to define at the beginning of the code (as global variable). For each arrow add two arrowheads (each composed of two small segments), each in the planes in which the arrow is located. For example, the x-axis arrow should have its arrowheads in the XY and XZ planes. The dimension of the arrow should be set through a parameter arrow_dim set at the beginning of the code.



- Make a perspective projection of the scene (the arrows) and tune the parameters of the functions gluPerspective and gluLookAt so that you can see a clear view of the whole scene with an acceptable size of the three arrows (like the one in the following figure).

- Add some animation to the previous code by performing 3D rotations about the 3 coordinate axes (x, y and z) and add the subsequent modifications that allow to have a clear view of the whole scene with an acceptable size of the three arrows.
- Modify the previous code to replace the arrows shape with a shape of your choice.

- What will happen if we combine the rotation process with a translation without any control on it? Propose a solution.

Practice:

The code given below allows to draw a cube. Write a program around this code that allows to rotate the cube about x-axis (left/right) and y-axis (up/down) with a step of 5 degrees via the directional keys (↑, ↓, →, ←) using the corresponding callback function. Make sure objects in front cover objects in back by enabling the depth test (glEnable (GL_DEPTH_TEST)).

Code that allows to draw a cube

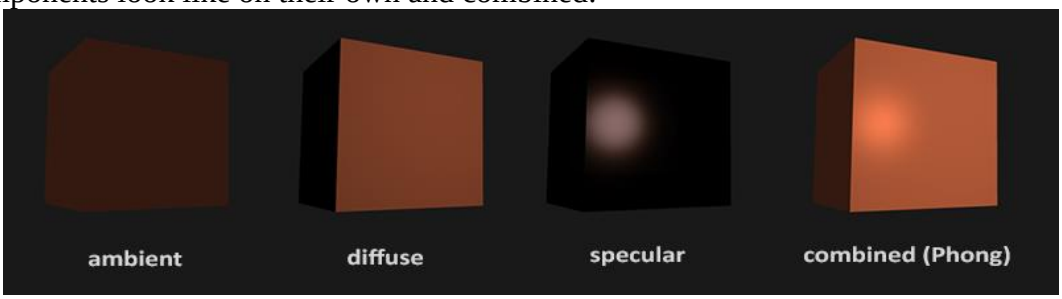
```
def render_Scene():
    # Multi-colored side - FRONT
    glBegin(GL_POLYGON);
    glColor3f( 1.0, 0.0, 0.0 );
    glVertex3f(  0.5, -0.5, -0.5 );      # Point P1 is red
    glColor3f( 0.0, 1.0, 0.0 );
    glVertex3f(  0.5,  0.5, -0.5 );      # Point P2 is green
    glColor3f( 0.0, 0.0, 1.0 );
    glVertex3f( -0.5,  0.5, -0.5 );      # Point P3 is blue
    glColor3f( 1.0, 0.0, 1.0 );
    glVertex3f( -0.5, -0.5, -0.5 );      # Point P4 is purple
    glEnd();

    # White side - BACK
    glBegin(GL_POLYGON);
    glColor3f(   1.0,  1.0, 1.0 );
    glVertex3f(  0.5, -0.5, 0.5 );
    glVertex3f(  0.5,  0.5, 0.5 );
    glVertex3f( -0.5,  0.5, 0.5 );
    glVertex3f( -0.5, -0.5, 0.5 );
    glEnd();

    # Purple side - RIGHT
    glBegin(GL_POLYGON);
    glColor3f(  1.0,  0.0,  1.0 );
    glVertex3f( 0.5, -0.5, -0.5 );
    glVertex3f( 0.5,  0.5, -0.5 );
    glVertex3f( 0.5,  0.5,  0.5 );
    glVertex3f( 0.5, -0.5,  0.5 );
    glEnd();

    # Green side - LEFT
    glBegin(GL_POLYGON);
    glColor3f(   0.0,  1.0,  0.0 );
    glVertex3f( -0.5, -0.5,  0.5 );
    glVertex3f( -0.5,  0.5,  0.5 );
    glVertex3f( -0.5,  0.5, -0.5 );
    glVertex3f( -0.5, -0.5, -0.5 );
    glEnd();

    # Blue side - TOP
    glBegin(GL_POLYGON);
```

```
glColor3f(  0.0,  0.0,  1.0 );
glVertex3f(  0.5,  0.5,  0.5 );
glVertex3f(  0.5,  0.5, -0.5 );
glVertex3f( -0.5,  0.5, -0.5 );
glVertex3f( -0.5,  0.5,  0.5 );
glEnd();

# Red side - BOTTOM
glBegin(GL_POLYGON);
glColor3f(  1.0,  0.0,  0.0 );
glVertex3f(  0.5, -0.5, -0.5 );
glVertex3f(  0.5, -0.5,  0.5 );
glVertex3f( -0.5, -0.5,  0.5 );
glVertex3f( -0.5, -0.5, -0.5 );
glEnd();
```

Part II : 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 as 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.

The function glLightfv will be called as many times as necessary to set all the intensities and related configuration parameters.

N.B.1. After setting the light parameters using glLightfv, we must tell OpenGL to use them to **compute** the vertex color (glEnable(GL_LIGHTING)), then by **including** the light we want (e.g. 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)
```

N.B.
The function glLightModelfv sets the lighting model parameters. For example, GL_LIGHT_MODEL_AMBIENT specifies the ambient RGBA intensity of the entire scene.

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 completely 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 (intensities and others) 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 tasks, we will write the code corresponding to each step as a function, then we call all the functions from within the display callback function.

For instance, 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 choosing the aspect ratio of gluPerspective
    _,_,width,height = glGetDoublev(GL_VIEWPORT)
    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()
```

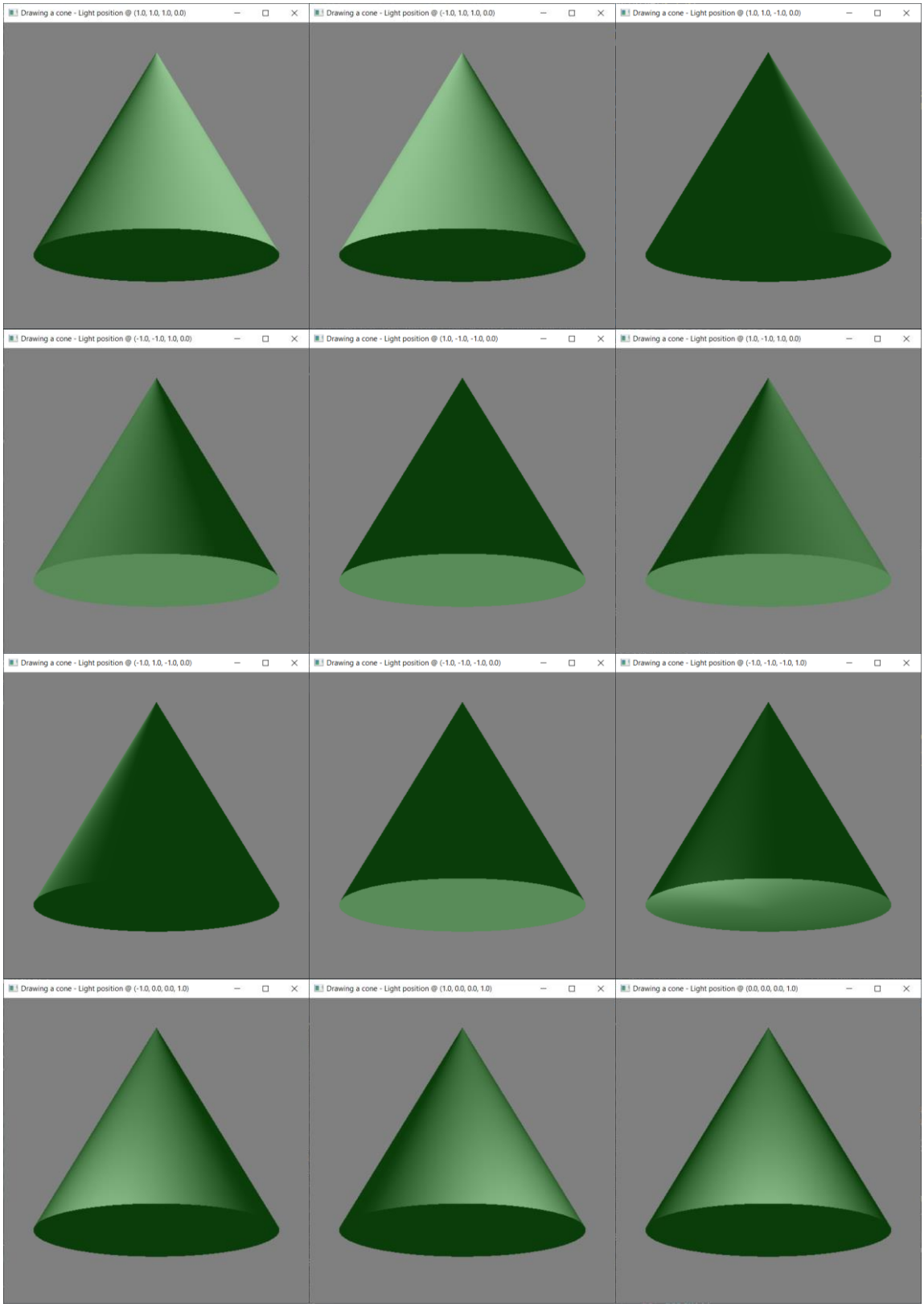
```
def display():
    background()
    perspective()
    lookat()
    light()
    depth()
    coneMaterial()
    transformations()
    drawCone(1,2,50,10)
    glutSwapBuffers()

# 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

Practice

- Modify the code that draws the cone with lighting effect (File S5_00_cone_light_template.py) to allow to see a face view of it (like in the previous set of images).
- Change the light intensities in function light()
- Change the setup of cone materials in function coneMaterial()
- Change the glLookAt parameters in function lookat()

Add another light GL_Light1 (first as a copy of GL_Light0 then change its parameters)


Drawing a cone - Light position @ (1.0, 1.0, 1.0, 0.0)
Drawing a cone - Light position @ (-1.0, 1.0, 1.0, 0.0)
Drawing a cone - Light position @ (1.0, 1.0, -1.0, 0.0)
Drawing a cone - Light position @ (-1.0, -1.0, 1.0, 0.0)
Drawing a cone - Light position @ (1.0, -1.0, -1.0, 0.0)
Drawing a cone - Light position @ (1.0, -1.0, 1.0, 0.0)
Drawing a cone - Light position @ (-1.0, 1.0, -1.0, 0.0)
Drawing a cone - Light position @ (-1.0, -1.0, -1.0, 0.0)
Drawing a cone - Light position @ (-1.0, -1.0, -1.0, 1.0)
Drawing a cone - Light position @ (-1.0, 0.0, 0.0, 1.0)
Drawing a cone - Light position @ (1.0, 0.0, 0.0, 1.0)
Drawing a cone - Light position @ (0.0, 0.0, 0.0, 1.0)

Practice:

Draw a cone and a sphere with 3D perspective projection and lighting effect (at least two lights) in order to have a shape as similar as it can be to the one shown below.