Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Mini Project Report

on

Orthographic and Perspective Projection in OpenGL

Computer Graphics

[Course Code: COMP 342]

[For the partial fulfillment of 3rd year/2nd Semester in Computer Engineering]

Submitted by:

Sabin Thapa

Roll no. 54

CE 3rd Year

Submitted to:

Mr. Dhiraj Shrestha

Assistant Professor

Department of Computer Science and Engineering

Submission date: May 20, 2022

Table of Contents

1. Introduction

1.1. Background

Projection refers to the mapping of three-dimensional points to a two-dimensional plane. There are basically two types of projection - parallel and perspective. Parallel projection preserves relative proportions of objects and accurate views of various sides of an object are obtained but doesn’t give realistic representation of a 3D object whereas perspective projection produces a realistic view but doesn’t preserve relative proportions. In perspective projection, equal sized objects appear at different sizes according to distance from the view plane.

In this mini project of Computer Graphics, I’ve tried implementing the orthographic (which is a type of parallel projection) and perspective projection using the OpenGL concepts in Python. These concepts are demonstrated with the help of a rotating cube and depending upon the type of projection, it either changes its size or it is clipped from the view plane. Rotating cubes are generated using the OpenGL functions and textures are applied to them by loading an image as texture to the cube surface. Predefined vertices and indices are provided to generate the cube and with the help of x and y axes rotation matrices, the cube is rotated in both the axes.

1.2. Software Requirements

The softwares required to to run the program are listed below:

i. Python3
ii. PyOpenGL

1.3. Tools and Libraries Used

The various tools and libraries used are listed below:

i. Pyrr
ii. Pillow
iii. Numpy
iv. PyOpenGL
v. GLFW

1.4. Mathematical Functions Used

To compute the mathematical functions and matrices, *pyrr* Math library is used. The Matrix44 class of *pyrr* library is used which represents a 4x4 matrix. This class provides a number of convenient functions and conversions. The 4x4 matrix supports rotation, translation, scale and skew. Matrices are laid out in a row-major format and can be loaded directly into OpenGL. To convert to column-major format, we transform the array using array.T method.

For example, to get the x-axis and y-axis rotation matrices, we do the following:

*rot_x = pyrr.Matrix44.from_x_rotation(0.5 * glfw.get_time())*

*rot_y = pyrr.Matrix44.from_y_rotation(0.8 * glfw.get_time())*

Now, to multiply these matrices, we use:

rotation = pyrr.matrix44.multiply(rot_x, rot_y)

Here, rotation is the combined rotation matrix which includes both the x and y axes rotation.

Similarly, to create a perspective and orthogonal projection matrices, the commands are:

projection = pyrr.matrix44.create_perspective_projection_matrix(45, 1280/720, 0.1, 100)

projection = pyrr.matrix44.create_orthogonal_projection_matrix(0, 1280, 0, 720, -1000, 1000)

These matrices are then used to compute the projection location.

Also, to get the translational and scaling model matrices, we do the following:

translation = pyrr.matrix44.create_from_translation(pyrr.Vector3([400, 200, -3]))

scale = pyrr.matrix44.create_from_scale(pyrr.Vector3([300, 300, 300]))

2. Working

Inorder to transform the coordinates from one space to the next coordinate space, several transformation matrices are used, of which the most important are the model, view and and projection matrix. Our vertex coordinates first start at local space or object space as local coordinates and are then further processed to world coordinates, view coordinates, clip coordinates and finally as screen coordinates. The process of each transformation is shown in the image below:

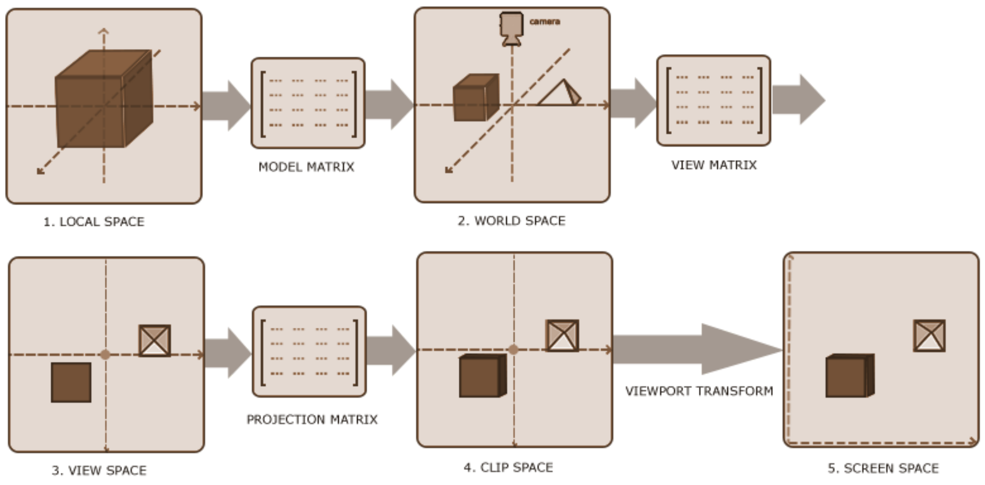


Fig: The global picture of transformation

1. Local coordinates are the coordinates of the object relative to its local origin i.e. the coordinates the object begins in.
2. Next, we transform the local coordinates to world-space coordinates (coordinates in respect of a larger world).
3. Then, we transform the world coordinates to view-space coordinates in such a way that each coordinate is as seen from the camera (or viewer's) point of view.
4. After obtaining the view space coordinates, they are projected to clip coordinates. Clip coordinates are processed to the -1.0 and 1.0 range and the vertices that will end up on the screen are determined. Projection to clip-space coordinates can add perspective if using perspective projection.
5. Lastly, clip coordinates are transformed to screen coordinates using viewport transform that transforms the coordinates from -1.0 and 1.0 to the coordinate range defined by glViewport. The resulting coordinates are then sent to the rasterizer which then turns them into fragments.

2.1. Orthographic Projection

The projection matrix for orthographic projection defines a cube-like frustum box that defines the clipping space where each vertex outside this box is clipped. The width, height and length of the visible frustum are specified when creating an orthographic projection.. All the coordinates inside this frustum will end up within the NDC range after being transformed by its matrix and thus won't be clipped. The frustum resembles a container as shown below:

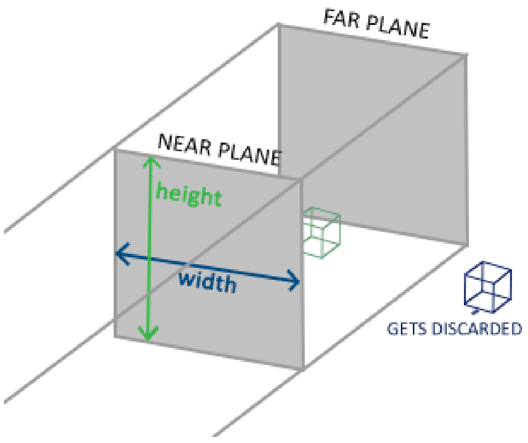


Fig: Orthographic Projection

Here, any coordinate behind the far plane and in front of the near plane is clipped. The orthographic frustum directly maps all coordinates inside the frustum to normalized device coordinates without any special side effects since it won't touch the w component of the transformed vector; if the w component remains equal to 1.0 perspective division won't change the coordinates.

To create an orthographic projection matrix, we make use of GLM's built-in function glm::ortho:

glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);

pyrr.matrix44.create_orthogonal_projection_matrix(0, 1280, 0, 720, -1000, 1000) (Python)

First parameter - left coordinate of the frustum

Second parameter - right coordinate of the frustum

Third parameter - bottom parameter of the frustum

Fourth parameter - top part of the frustum

Fifth and sixth parameters - distances between the near and far plane

This specific projection matrix transforms all coordinates between these x, y and z range values to normalized device coordinates. An orthographic projection matrix directly maps coordinates to the two-dimensional plane that is your screen, but in reality a direct projection produces unrealistic results since the projection doesn't take perspective into account.

2.2. Perspective Projection

In real life, the objects even if they are large, if they are far from us, they appear to be much smaller. This effect is called perspective. We can understand perspective by looking at a railway line as shown below:

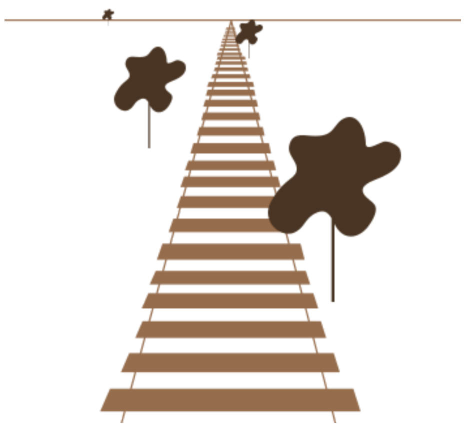


Fig: Perspective Projection

At a far enough distance, due to the perspective, the railway lines seem to coincide. Perspective projection tries to mimic this effect by using a perspective projection matrix. The projection matrix maps a given frustum range to clip space, but also manipulates the w value of each vertex coordinate in such a way that the further away a vertex coordinate is from the viewer, the higher this w component becomes. Once the coordinates are transformed to clip space they are in the range -w to w. Anything outside this range is clipped OpenGL requires that the visible coordinates fall between the range -1.0 and 1.0 as the final vertex shader output, thus once the coordinates are in clip space, perspective division is applied to the clip space coordinates:

out= $(x/w,\ y/w,\ x/w)$

A perspective projection matrix can be created in GLM as follows:

glm::mat4 proj = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);

pyrr.matrix44.create_perspective_projection_matrix(45, 1280/720, 0.1, 100) (Python)

The glm::perspective again creates a large frustum that defines the visible space. Anything outside the frustum will not end up in the clip space volume and will thus be clipped. A perspective frustum can be visualized as a non-uniformly shaped box from which each coordinate inside this box will be mapped to a point in clip space. An image of a perspective frustum is seen below:

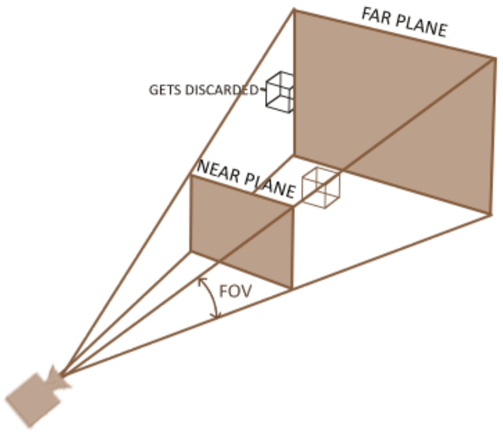


Fig: Perspective Frustum

First parameter - fov(field of view) value. For a realistic view, it is set to 45 degrees.

Second parameter - aspect ratio (viewport’s width / viewport’s height)

Third and fourth parameter - near and far plane of the frustum

Since the orthographic projection doesn't use perspective projection, objects farther away do not seem smaller, which produces a weird visual output. For this reason the orthographic projection is mainly used for 2D renderings. But to get a realistic 3D view of an object, we use perspective projection.

3. Demo Link

The links to the demo video are attached below.

Google Drive:

https://drive.google.com/file/d/1B_Cuo7OfHNAS2iuJocPoTxWCvoutLC5K/view?usp=sharing

YouTube:

https://youtu.be/pXl3qf2yYms

4. Output

Orthographic Projection

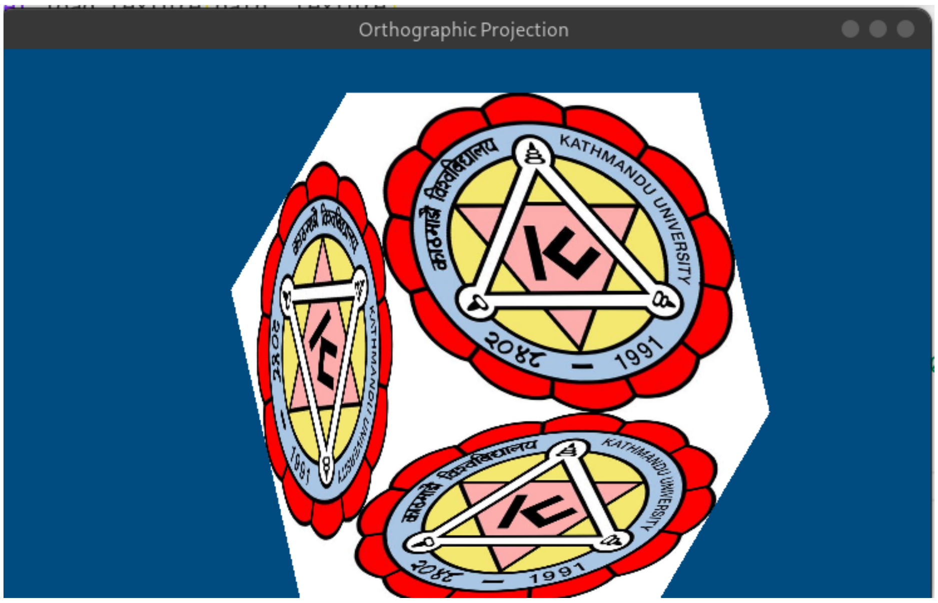


Fig: Orthographic Projection

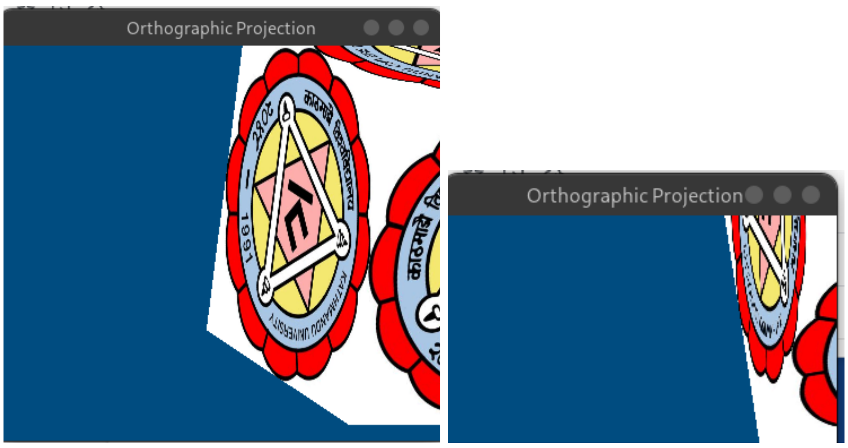


Fig: Orthographic Projection(reduced window sizes)

Perspective Projection

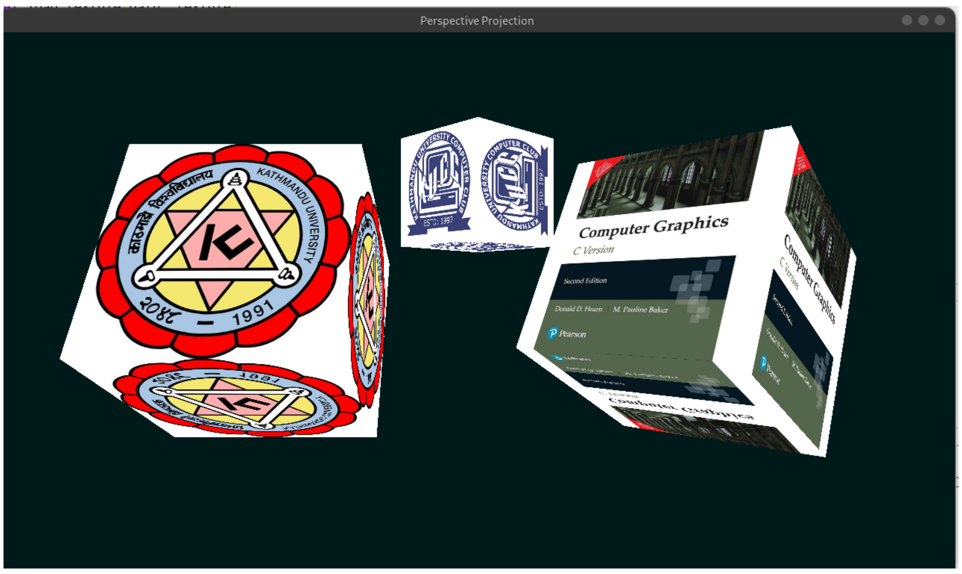
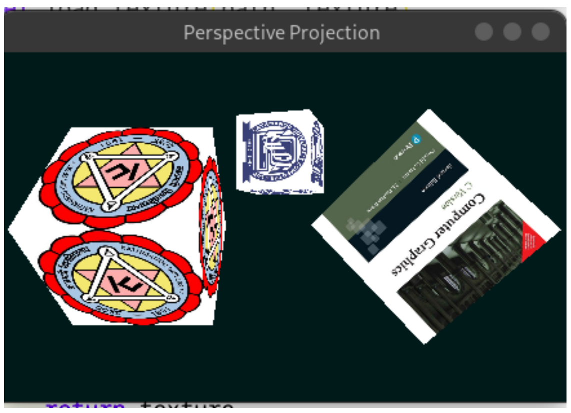


Fig: Perspective Projection



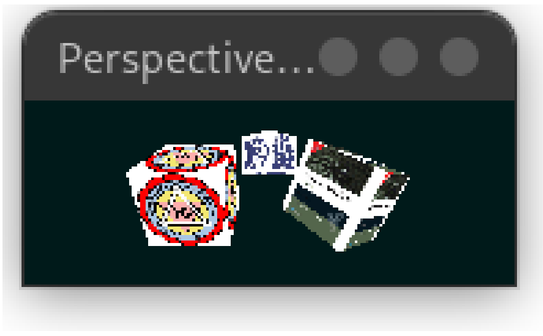


Fig: Perspective Projection(reduced window sizes)

5. Conclusion

In this way, orthogonal and perspective projection were implemented as the mini project for the course Computer Graphics, using OpenGL. This project helped me enrich my knowledge on how projections work and how computer graphics uses different types of projections based on the requirements. Various OpenGL functions to create a rotating cube, create the texture of the rotating cube and to display the results were used.

Appendix I (Orthographic Projection)

```
import glfw

from OpenGL.GL import *

from OpenGL.GL.shaders import compileProgram, compileShader

import numpy as np

import pyrr

from PIL import Image

vertex_src = """

# version 330

layout(location = 0) in vec3 a_position;

layout(location = 1) in vec2 a_texture;


uniform mat4 model; // combined translation and rotation

uniform mat4 projection;


out vec3 v_color;

out vec2 v_texture;


void main()

{

   gl_Position = projection * model * vec4(a_position, 1.0);

   v_texture = a_texture;

}

"""

fragment_src = """

# version 330


in vec2 v_texture;
```

```
out vec4 out_color;

uniform sampler2D s_texture;


void main()

{

   out_color = texture(s_texture, v_texture);

}

"""

# glfw callback functions

def window_resize(window, width, height):

   glViewport(0, 0, width, height)

    projection = pyrr.matrix44.create_orthogonal_projection_matrix(0, width,
0, height, -1000, 1000)

   glUniformMatrix4fv(proj_loc, 1, GL_FALSE, projection)

# initializing glfw library

if not glfw.init():

   raise Exception("glfw can not be initialized!")

# creating the window

window  =  glfw.create_window(1280,  720,  "Orthographic  Projection",  None,
None)

# check if window was created

if not window:

   glfw.terminate()

   raise Exception("glfw window can not be created!")

# set window's position

glfw.set_window_pos(window, 400, 200)

# set the callback function for window resize

glfw.set_window_size_callback(window, window_resize)

# make the context current

glfw.make_context_current(window)

vertices = [
                -0.5, -0.5,  0.5, 0.0, 0.0,
                 0.5, -0.5,  0.5, 1.0, 0.0,
                 0.5,  0.5,  0.5, 1.0, 1.0,
                -0.5,  0.5,  0.5, 0.0, 1.0,

                -0.5, -0.5, -0.5, 0.0, 0.0,
                 0.5, -0.5, -0.5, 1.0, 0.0,
                 0.5,  0.5, -0.5, 1.0, 1.0,
                -0.5,  0.5, -0.5, 0.0, 1.0,

                 0.5, -0.5, -0.5, 0.0, 0.0,
                 0.5,  0.5, -0.5, 1.0, 0.0,
                 0.5,  0.5,  0.5, 1.0, 1.0,
                 0.5, -0.5,  0.5, 0.0, 1.0,

                -0.5,  0.5, -0.5, 0.0, 0.0,
                -0.5, -0.5, -0.5, 1.0, 0.0,
                -0.5, -0.5,  0.5, 1.0, 1.0,
                -0.5,  0.5,  0.5, 0.0, 1.0,

                -0.5, -0.5, -0.5, 0.0, 0.0,
                 0.5, -0.5, -0.5, 1.0, 0.0,
                 0.5, -0.5,  0.5, 1.0, 1.0,
                -0.5, -0.5,  0.5, 0.0, 1.0,

                 0.5, 0.5, -0.5, 0.0, 0.0,
                -0.5, 0.5, -0.5, 1.0, 0.0,
                -0.5, 0.5,  0.5, 1.0, 1.0,
                 0.5, 0.5,  0.5, 0.0, 1.0]
    indices = [
                #Top
                0,  1,  2,  2,  3,  0,
                #Bottom
                4,  5,  6,  6,  7,  4,
                #Front
                8,  9, 10, 10, 11,  8,
                #Back
               12, 13, 14, 14, 15, 12,
                #Left
               16, 17, 18, 18, 19, 16,
                #Right
               20, 21, 22, 22, 23, 20]

     vertices = np.array(vertices, dtype=np.float32)
     indices = np.array(indices, dtype=np.uint32)

     shader    =    compileProgram(compileShader(vertex_src,    GL_VERTEX_SHADER),
     compileShader(fragment_src, GL_FRAGMENT_SHADER))

     # Vertex Buffer Object
     VBO = glGenBuffers(1)
     glBindBuffer(GL_ARRAY_BUFFER, VBO)
     glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STATIC_DRAW)

     # Element Buffer Object
     EBO = glGenBuffers(1)
     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO)
     glBufferData(GL_ELEMENT_ARRAY_BUFFER,        indices.nbytes,        indices,
     GL_STATIC_DRAW)

     glEnableVertexAttribArray(0)
```

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, vertices.itemsize * 5,
ctypes.c_void_p(0))


glEnableVertexAttribArray(1)

glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, vertices.itemsize * 5,
ctypes.c_void_p(12))


texture = glGenTextures(1)

glBindTexture(GL_TEXTURE_2D, texture)


# Set the texture wrapping parameters

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)

# Set texture filtering parameters

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)


# load image

image = Image.open("textures/ku.jpg")

image = image.transpose(Image.FLIP_TOP_BOTTOM)

img_data = image.convert("RGBA").tobytes()

# img_data = np.array(image.getdata(), np.uint8) # second way of getting the
raw image data

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, image.width, image.height, 0,
GL_RGBA, GL_UNSIGNED_BYTE, img_data)


glUseProgram(shader)

# glClearColor(0, 0.1, 0.1, 1)

glClearColor(0, 0.3, 0.5, 1)


glEnable(GL_DEPTH_TEST)

glEnable(GL_BLEND)
```

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)

#   projection   =   pyrr.matrix44.create_perspective_projection_matrix(45,
1280/720, 0.1, 100)

projection  = pyrr.matrix44.create_orthogonal_projection_matrix(0,  1280,  0,
720, -1000, 1000)

translation  = pyrr.matrix44.create_from_translation(pyrr.Vector3([400,  200,
-3]))

scale = pyrr.matrix44.create_from_scale(pyrr.Vector3([300, 300, 300]))

model_loc = glGetUniformLocation(shader, "model")

proj_loc = glGetUniformLocation(shader, "projection")

glUniformMatrix4fv(proj_loc, 1, GL_FALSE, projection)


# the main application loop

while not glfw.window_should_close(window):

   glfw.poll_events()

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)


   rot_x = pyrr.Matrix44.from_x_rotation(0.5 * glfw.get_time())

   rot_y = pyrr.Matrix44.from_y_rotation(0.8 * glfw.get_time())


   rotation = pyrr.matrix44.multiply(rot_x, rot_y)

   model = pyrr.matrix44.multiply(scale, rotation)

   model = pyrr.matrix44.multiply(model, translation)

   glUniformMatrix4fv(model_loc, 1, GL_FALSE, model)

   glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)

   glfw.swap_buffers(window)


# terminate glfw, free up allocated resources

glfw.terminate()
```

Appendix II (Perspective Projection)

```
import glfw

from OpenGL.GL import *

from OpenGL.GL.shaders import compileProgram, compileShader

import numpy as np

import pyrr

from TextureLoader import load_texture




vertex_src = """

# version 330


layout(location = 0) in vec3 a_position;

layout(location = 1) in vec2 a_texture;


uniform mat4 model;

uniform mat4 projection;

uniform mat4 view;


out vec3 v_color;

out vec2 v_texture;


void main()

{

   gl_Position = projection * view * model * vec4(a_position, 1.0);

   v_texture = a_texture;

}

"""
```

```
fragment_src = """

# version 330


in vec2 v_texture;


out vec4 out_color;


uniform sampler2D s_texture;


void main()

{

   out_color = texture(s_texture, v_texture);

}

"""


# glfw callback functions

def window_resize(window, width, height):

   glViewport(0, 0, width, height)

   projection = pyrr.matrix44.create_perspective_projection_matrix(45, width
/ height, 0.1, 100)

   glUniformMatrix4fv(proj_loc, 1, GL_FALSE, projection)


# initializing glfw library

if not glfw.init():

   raise Exception("glfw can not be initialized!")

# creating the window

window = glfw.create_window(1280, 720, "Perspective Projection", None, None)


# check if window was created

if not window:
   glfw.terminate()
   raise Exception("glfw window can not be created!")
# set window's position
glfw.set_window_pos(window, 400, 200)

# set the callback function for window resize
glfw.set_window_size_callback(window, window_resize)

# make the context current
glfw.make_context_current(window)
vertices = [-0.5, -0.5,  0.5, 0.0, 0.0,
             0.5, -0.5,  0.5, 1.0, 0.0,
             0.5,  0.5,  0.5, 1.0, 1.0,
            -0.5,  0.5,  0.5, 0.0, 1.0,

            -0.5, -0.5, -0.5, 0.0, 0.0,
             0.5, -0.5, -0.5, 1.0, 0.0,
             0.5,  0.5, -0.5, 1.0, 1.0,
            -0.5,  0.5, -0.5, 0.0, 1.0,

             0.5, -0.5, -0.5, 0.0, 0.0,
             0.5,  0.5, -0.5, 1.0, 0.0,
             0.5,  0.5,  0.5, 1.0, 1.0,
             0.5, -0.5,  0.5, 0.0, 1.0,

            -0.5,  0.5, -0.5, 0.0, 0.0,
            -0.5, -0.5, -0.5, 1.0, 0.0,
            -0.5, -0.5,  0.5, 1.0, 1.0,
            -0.5,  0.5,  0.5, 0.0, 1.0,
           -0.5, -0.5, -0.5, 0.0, 0.0,

            0.5, -0.5, -0.5, 1.0, 0.0,

            0.5, -0.5,  0.5, 1.0, 1.0,

           -0.5, -0.5,  0.5, 0.0, 1.0,


            0.5, 0.5, -0.5, 0.0, 0.0,

           -0.5, 0.5, -0.5, 1.0, 0.0,

           -0.5, 0.5,  0.5, 1.0, 1.0,

            0.5, 0.5,  0.5, 0.0, 1.0]


indices = [ 0,  1,  2,  2,  3,  0,

           4,  5,  6,  6,  7,  4,

           8,  9, 10, 10, 11,  8,

          12, 13, 14, 14, 15, 12,

          16, 17, 18, 18, 19, 16,

          20, 21, 22, 22, 23, 20]


vertices = np.array(vertices, dtype=np.float32)

indices = np.array(indices, dtype=np.uint32)


shader    =    compileProgram(compileShader(vertex_src,    GL_VERTEX_SHADER),
compileShader(fragment_src, GL_FRAGMENT_SHADER))

# Vertex Buffer Object

VBO = glGenBuffers(1)

glBindBuffer(GL_ARRAY_BUFFER, VBO)

glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STATIC_DRAW)

# Element Buffer Object

EBO = glGenBuffers(1)

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO)

glBufferData(GL_ELEMENT_ARRAY_BUFFER,        indices.nbytes,        indices,
GL_STATIC_DRAW)
```

```
glEnableVertexAttribArray(0)

glVertexAttribPointer(0,  3,  GL_FLOAT,  GL_FALSE,  vertices.itemsize  *  5,
ctypes.c_void_p(0))


glEnableVertexAttribArray(1)

glVertexAttribPointer(1,  2,  GL_FLOAT,  GL_FALSE,  vertices.itemsize  *  5,
ctypes.c_void_p(12))


texture = glGenTextures(3)


cube1_texture = load_texture("textures/graphics.jpg", texture[0])

cube2_texture = load_texture("textures/ku.jpg", texture[1])

cube3_texture = load_texture("textures/kucc.jpg", texture[2])


glUseProgram(shader)

glClearColor(0, 0.1, 0.1, 1)

glEnable(GL_DEPTH_TEST)

glEnable(GL_BLEND)

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)


projection      =      pyrr.matrix44.create_perspective_projection_matrix(45,
1280/720, 0.1, 100)

cube1 = pyrr.matrix44.create_from_translation(pyrr.Vector3([1, 0, 0]))

cube2 = pyrr.matrix44.create_from_translation(pyrr.Vector3([-1, 0, 0]))

cube3 = pyrr.matrix44.create_from_translation(pyrr.Vector3([0, 1, -3]))

# eye, target, up

view     =     pyrr.matrix44.create_look_at(pyrr.Vector3([0,     0,     3]),
pyrr.Vector3([0, 0, 0]), pyrr.Vector3([0, 1, 0]))


model_loc = glGetUniformLocation(shader, "model")

proj_loc = glGetUniformLocation(shader, "projection")

view_loc = glGetUniformLocation(shader, "view")
```

```
glUniformMatrix4fv(proj_loc, 1, GL_FALSE, projection)

glUniformMatrix4fv(view_loc, 1, GL_FALSE, view)


# the main application loop

while not glfw.window_should_close(window):

   glfw.poll_events()

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)


   rot_x = pyrr.Matrix44.from_x_rotation(0.5 * glfw.get_time())

   rot_y = pyrr.Matrix44.from_y_rotation(0.8 * glfw.get_time())


   rotation = pyrr.matrix44.multiply(rot_x, rot_y)

   model = pyrr.matrix44.multiply(rotation, cube1)

   glBindTexture(GL_TEXTURE_2D, texture[0])

   glUniformMatrix4fv(model_loc, 1, GL_FALSE, model)

   glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)

   model = pyrr.matrix44.multiply(rot_x, cube2)

   glBindTexture(GL_TEXTURE_2D, texture[1])

   glUniformMatrix4fv(model_loc, 1, GL_FALSE, model)

   glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)

   model = pyrr.matrix44.multiply(rot_y, cube3)

   glBindTexture(GL_TEXTURE_2D, texture[2])

   glUniformMatrix4fv(model_loc, 1, GL_FALSE, model)

   glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)

   glfw.swap_buffers(window)

# terminate glfw, free up allocated resources

glfw.terminate()
```

Appendix III (Texture Loader)

```
from   OpenGL.GL   import   glBindTexture,   glTexParameteri,   GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, \

                GL_TEXTURE_WRAP_T,     GL_REPEAT,     GL_TEXTURE_MIN_FILTER,
GL_TEXTURE_MAG_FILTER, GL_LINEAR,\

   glTexImage2D, GL_RGBA, GL_UNSIGNED_BYTE

from PIL import Image

# for use with GLFW

def load_texture(path, texture):

   glBindTexture(GL_TEXTURE_2D, texture)

   # Set the texture wrapping parameters

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)

   # Set texture filtering parameters

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)

   # load image

   image = Image.open(path)

   image = image.transpose(Image.FLIP_TOP_BOTTOM)

   img_data = image.convert("RGBA").tobytes()

     glTexImage2D(GL_TEXTURE_2D,  0,  GL_RGBA,  image.width,  image.height,  0,
GL_RGBA, GL_UNSIGNED_BYTE, img_data)

   return texture
```