Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Three Dimensional Transformations

Lab Report Five

[COMP342]

(For partial fulfillment of 3rd Year/1st Semester in Computer Science)

Submitted by:

Yugesh Upadhyaya Luitel (38)

Submitted to:

Mr. Dhiraj Shrestha

Department of Computer Science and Engineering

Submission Date: December 15, 2022

Table of Contents

Chapter 1: Three Dimensional Transformation

1.1 Introduction

3D models are a mathematical representation of a physical entity that occupies space. Three Dimensional Transformations are the processes of manipulating the view of a three-dimensional object with respect to its original position by modifying its physical attributes. These processes when applied to a point on three-dimensional space (or 3D shape, constructed using 3D points) will map it into another point in the same three-dimensional space (or 3D shape, made out of the mapped 3D points). Essentially, Three Dimensional Transformations map a set of Original Points **O** of a **3D Space** into a set of Transformed Points **T** into the same **3D space**.

There are various transformations that can be obtained in three-dimensional space, and through this particular lab work we aim to explore these transformations and their effects to a **Three-Dimensional Cube** of our choice. Amongst the various transformations, we will be looking at:

- Translation
- Rotation
- Scaling

1.2 Additional Tools

The Programming Language, Graphics Library and Tools used for the Transformations are as follows:

Programming Language: Python 3.10

Graphics Library: PyOpenGL 3.1.6

Window Renderer: GLUT

Helper Library: ctypes, numpy

1.3 Setup and Helpers

For the purpose of demonstrating the effect of various transformations, we will be using a 3D Cube with 8 vertices at **(-0.5, 0.5, 0.5),(0.5, 0.5, 0.5),(-0.5, -0.5, 0.5),(0.5, -0.5, 0.5),(-0.5, 0.5, -0.5),(0.5, 0.5, -0.5),(-0.5, -0.5, -0.5),(0.5, -0.5, -0.5)** in a three dimensional space. Since OpenGL does not provide its user with default three-dimensional object, the cube is drawn manually using the OpenGL's gl.GL_TRIANGLES primitive along with **indicesData** to specify the vertices of 12 triangles to be drawn in order to create the various faces for a cube.

Moreover, for the purpose of implementing 3D Graphical elements, the **glut.glutDisplayMode** is set to **glut.GLUT_DEPTH** for depth buffer creation and updating. Also, to mitigate the overlapping effects , the Z Buffer test has been enabled using **gl.glEnable(gl.GL_DEPTH_TEST)** command.

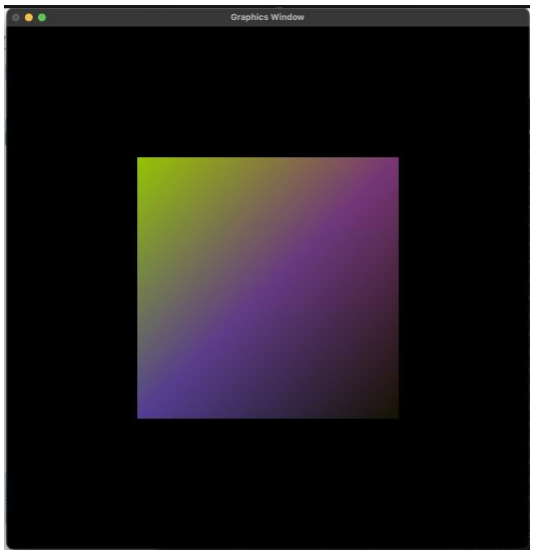
A helper function **generateCubeData()** is used for the generation of the vertices of Cube, the color value for each of these vertices, and the indicesData necessary for plotting the Cube.

Code Snippet for Build Data Helper

```
# -- Building Data --
def generateCubeData():
    data = np.zeros(8, [("position", np.float32, 3),
                        ("color",    np.float32, 4)])

    data["position"] = (
        (-0.5, 0.5, 0.5),
        (0.5, 0.5, 0.5),
        (-0.5, -0.5, 0.5),
        (0.5, -0.5, 0.5),
        (-0.5, 0.5, -0.5),
        (0.5, 0.5, -0.5),
        (-0.5, -0.5, -0.5),
        (0.5, -0.5, -0.5))

    data['color'] = (
                (0.114, 0.505, 0.345, 1.0),
                (0.483, 0.290, 0.734, 1.0),
                (0.097, 0.513, 0.064, 1.0),
                (0.245, 0.719, 0.592, 1.0),
                (0.583, 0.771, 0.014, 1.0),
                (0.473, 0.211, 0.457, 1.0),
                (0.322, 0.245, 0.574, 1.0),
                (0.083, 0.071, 0.014, 1.0))

    indicesData = np.array([0, 1, 2, 1, 2, 3,
                            4, 5, 6, 5, 6, 7,
                            4, 2, 5, 2, 5, 1,
                            5, 1, 3, 5, 3, 7,
                            0, 4, 2, 4, 2, 6,
                            2, 6, 3, 6, 3, 7], np.int32)

    return data, indicesData
```

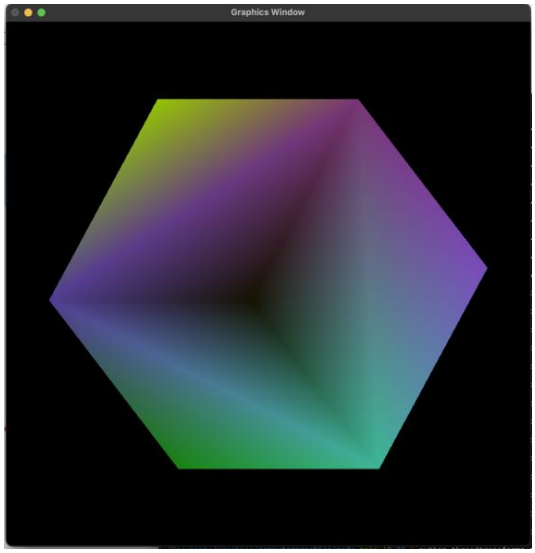
```
def display():
    gl.glClear(gl.GL_COLOR_BUFFER_BIT)
    gl.glDrawElements(gl.GL_TRIANGLES, len(indicesData),gl.GL_UNSIGNED_INT, None)
    gl.glFlush()
    glut.glutSwapBuffers()
```

Output Cube



At first glance, this **Output Cube** looks like a two-dimensional square. However, it is indeed a Cube, which looks like a square due to an orthogonal viewing onto a face of the cube, where the vertices of the faces at the back, top, bottom and sides overlap with the front face. And, since the front face has a lower z value it is displayed by the Graphical Library such that the front face blocks all other faces from appearing. Hence, for the purpose of this Lab Work we will be rotating this cube along both x and y axes by 40 degrees. This rotated cube will be considered the base cube for the future transformations.

40 Degrees Rotated Cube (Base Cube)



Moreover, we will be utilizing the **`generateTransforms()`** helper to build our transformation matrix based upon the **`transformationType`** and **`transformationData`** provided to it as parameters.

This helper functions looks at the **`transformationType`** parameter which is a string representing one of our 3 transformations. If it does not match with any of our defined transformations it returns an identity matrix of size **`(4, 4)`**, which represents a transformation that maps the given points to themselves. Upon **`transformationType`** match, the function uses the **`transformationData`** parameter, which is a list containing all the required data necessary for that transformation, to create a **`transformationMatrix`** and return it.

The data required for **`transformationData`** parameter depending upon the type of transformation are as follows:

a. Translation : **`transformationData=[translation in X, translation in Y, translation in Z]`**

b. Rotation : **`transformationData=[("pitch" or "yaw" or "roll"), degree of rotation]`**

c. Scaling : **`transformationData=[scaling in X, scaling in Y, scaling in Z]`**

Code Snippet for Transformation Generation Helper

```
def generateTransforms(transformationType = None, transformationData = None):

    transformationMatrix = np.identity(4, dtype = np.float32)
    transformationMatrix = transformationMatrix.flatten()

    if not transformationType:
        return transformationMatrix

    if transformationType == "translation":
        transformationMatrix = np.array([  1.0,0.0,0.0,transformationData[0],
                                           0.0,1.0,0.0,transformationData[1],
                                           0.0,0.0,1.0,transformationData[2],
                                           0.0,0.0,0.0,1.0], np.float32)

    elif transformationType == "rotation":

        cTheta = np.cos(transformationData[1]/180 * math.pi)
        sTheta = np.sin(transformationData[1]/180 * math.pi)

        # x - axis rotation
        if transformationData[0] == "pitch":
            transformationMatrix = np.array([1.0,0.0,0.0,0.0,
                                           0.0,cTheta,-sTheta,0.0,
                                           0.0,sTheta,cTheta,0.0,
                                           0.0,0.0,0.0,1.0], np.float32)

        # y - axis rotation
        elif transformationData[0] == "yaw":
            transformationMatrix = np.array([cTheta,0.0,sTheta,0.0,
```

```
                                        0.0,1.0,0.0,0.0,
                                        -sTheta,0.0,cTheta,0.0,
                                        0.0,0.0,0.0,1.0], np.float32)

        # z - axis rotation
        elif transformationData[0] == "roll":
            transformationMatrix = np.array([cTheta,-sTheta,0.0,0.0,
                                        sTheta,cTheta,0.0,0.0,
                                        0.0,0.0,1.0,0.0,
                                        0.0,0.0,0.0,1.0], np.float32)

    elif transformationType == "scaling":
        transformationMatrix = np.array([transformationData[0],0.0,0.0,0.0,
                                    0.0,transformationData[1],0.0,0.0,
                                    0.0,0.0,transformationData[2],0.0,
                                    0.0,0.0,0.0,1.0], np.float32)

    return transformationMatrix
```

Similarly, for a 3D Transformation we will be to using 4 x 4 Homogeneous Matrix to represent the Transformation Matrices. And, the data points given to the vertex shader is multiplied by these homogeneous transformation matrices to get the mapped data points, which is then plotted by the vertex shader itself.

Code Snippet for Vertex Shader

```
vertexShaderCode = """
    attribute vec3 position;
    attribute vec4 color;
    varying vec4 vColor;
    uniform mat4 rotationMatrix[2];
    uniform mat4 transformationMatrix;

    void main(){
       gl_Position = transformationMatrix * rotationMatrix[1] * rotationMatrix[0] *
       vec4(position, 1.0);
        vColor = color;
    }
    """
```

Chapter 2: General 3D Transformation

2.1 General Information

Since, the transformations using the Homogeneous Coordinate System works by cross multiplying the transformation function to the data points of our 3D Shape, we can write a general program for this multiplication. And, upon need we can vary the transformation matrix parameter to our desired one to achieve any of the described transformations, namely:

a. Translation

b. Rotation

c. Scaling

2.2 Source Code

```
import os
import sys
import ctypes
import numpy as np
import math
import OpenGL.GL as gl
import OpenGL.GLUT as glut
import OpenGL.GLU as glu

vertexShaderCode = """
    attribute vec3 position;
    attribute vec4 color;
    varying vec4 vColor;
    uniform mat4 rotationMatrix[2];
    uniform mat4 transformationMatrix;

    void main(){
        gl_Position = transformationMatrix * rotationMatrix[1] * rotationMatrix[0] *
vec4(position, 1.0);
        vColor = color;
    }
    """
fragmentShaderCode = """
    varying vec4 vColor;
    void main(){
        gl_FragColor = vColor;
    }
    """

# -- Building Data --
def generateCubeData():
    data = np.zeros(8, [("position", np.float32, 3),
                        ("color",    np.float32, 4)])

    data["position"] = (
        (-0.5, 0.5, 0.5),
        (0.5, 0.5, 0.5),
        (-0.5, -0.5, 0.5),
        (0.5, -0.5, 0.5),
```

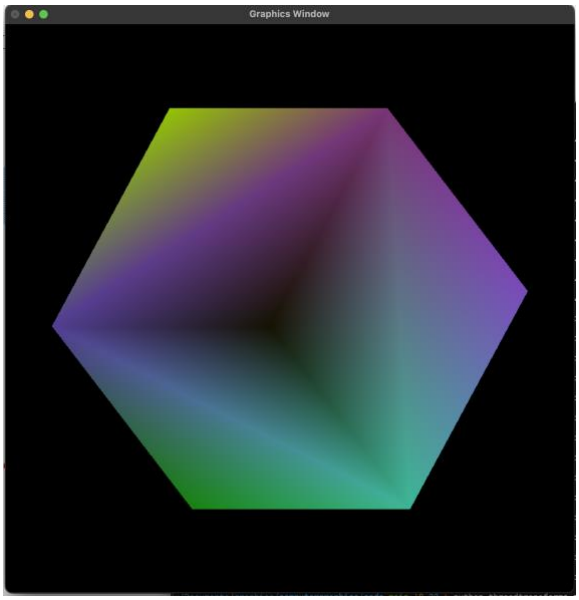
```
        (-0.5, 0.5, -0.5),
        (0.5, 0.5, -0.5),
        (-0.5, -0.5, -0.5),
        (0.5, -0.5, -0.5),
    )

    data['color'] = (
                (0.114, 0.505, 0.345, 1.0),
                (0.483, 0.290, 0.734, 1.0),
                (0.097, 0.513, 0.064, 1.0),
                (0.245, 0.719, 0.592, 1.0),
                (0.583, 0.771, 0.014, 1.0),
                (0.473, 0.211, 0.457, 1.0),
                (0.322, 0.245, 0.574, 1.0),
                (0.083, 0.071, 0.014, 1.0),
                )

    indicesData = np.array([0, 1, 2, 1, 2, 3,
                            4, 5, 6, 5, 6, 7,
                            4, 2, 5, 2, 5, 1,
                            5, 1, 3, 5, 3, 7,
                            0, 4, 2, 4, 2, 6,
                            2, 6, 3, 6, 3, 7], np.int32)
    return data, indicesData

def generateTransforms(transformationType = None, transformationData = None):

    transformationMatrix = np.identity(4, dtype = np.float32)
    transformationMatrix = transformationMatrix.flatten()

    if not transformationType:
        return transformationMatrix

    if transformationType == "translation":
        transformationMatrix = np.array([  1.0,0.0,0.0,transformationData[0],
                                           0.0,1.0,0.0,transformationData[1],
                                           0.0,0.0,1.0,transformationData[2],
                                           0.0,0.0,0.0,1.0], np.float32)

    elif transformationType == "rotation":

        cTheta = np.cos(transformationData[1]/180 * math.pi)
        sTheta = np.sin(transformationData[1]/180 * math.pi)

        # x - axis rotation
        if transformationData[0] == "pitch":
            transformationMatrix = np.array([1.0,0.0,0.0,0.0,
                                             0.0,cTheta,-sTheta,0.0,
                                             0.0,sTheta,cTheta,0.0,
                                             0.0,0.0,0.0,1.0], np.float32)

        # y - axis rotation
        elif transformationData[0] == "yaw":
            transformationMatrix = np.array([cTheta,0.0,sTheta,0.0,
                                             0.0,1.0,0.0,0.0,
                                             -sTheta,0.0,cTheta,0.0,
                                             0.0,0.0,0.0,1.0], np.float32)

        # z - axis rotation
        elif transformationData[0] == "roll":
            transformationMatrix = np.array([cTheta,-sTheta,0.0,0.0,
                                             sTheta,cTheta,0.0,0.0,
                                             0.0,0.0,1.0,0.0,
                                             0.0,0.0,0.0,1.0], np.float32)

    elif transformationType == "scaling":
```

```
        transformationMatrix = np.array([transformationData[0],0.0,0.0,0.0,
                                         0.0,transformationData[1],0.0,0.0,
                                         0.0,0.0,transformationData[2],0.0,
                                         0.0,0.0,0.0,1.0], np.float32)

    return transformationMatrix

# function to request and compiler shader slots from GPU
def createShader(source, type):
    # request shader
    shader = gl.glCreateShader(type)

    # set shader source using the code
    gl.glShaderSource(shader, source)

    gl.glCompileShader(shader)
    if not gl.glGetShaderiv(shader, gl.GL_COMPILE_STATUS):
        error = gl.glGetShaderInfoLog(shader).decode()
        print(error)
        raise RuntimeError(f"{source} shader compilation error")
    return shader

# func to build and activate program
def createProgram(vertex, fragment):
    program = gl.glCreateProgram()

    # attach shader objects to the program
    gl.glAttachShader(program, vertex)
    gl.glAttachShader(program, fragment)

    gl.glLinkProgram(program)
    if not gl.glGetProgramiv(program, gl.GL_LINK_STATUS):
        print(gl.glGetProgramInfoLog(program))
        raise RuntimeError('Linking error')

    # Get rid of shaders (no more needed)
    gl.glDetachShader(program, vertex)
    gl.glDetachShader(program, fragment)

    return program

# initialization function
def initialize(transformationMatrix):
    global program
    global data

    gl.glEnable(gl.GL_DEPTH_TEST)
    gl.glClear(gl.GL_COLOR_BUFFER_BIT | gl.GL_DEPTH_BUFFER_BIT)
    gl.glClearColor(0.0, 0.0, 0.0, 0.0)
    gl.glLoadIdentity()

    program = createProgram(
        createShader(vertexShaderCode, gl.GL_VERTEX_SHADER),
        createShader(fragmentShaderCode, gl.GL_FRAGMENT_SHADER),
    )

    degrees = 40
    cTheta = np.cos(degrees/180 * math.pi)
    sTheta = np.sin(degrees/180 * math.pi)
    initialRotation = np.array([1.0,0.0,0.0,0.0,
                                0.0,cTheta,-sTheta,0.0,
                                0.0,sTheta,cTheta,0.0,
                                0.0,0.0,0.0,1.0], np.float32)
    secondRotation = np.array([cTheta,0.0,sTheta,0.0,
                                0.0,1.0,0.0,0.0,
                                -sTheta,0.0,cTheta,0.0,
```

```
                                0.0,0.0,0.0,1.0], np.float32)
    rotation = np.array([initialRotation, secondRotation])
    # make program the default program
    gl.glUseProgram(program)
    buffer = gl.glGenBuffers(1)
    indicesBuffer = gl.glGenBuffers(1)

    # make these buffer the default one
    gl.glBindBuffer(gl.GL_ARRAY_BUFFER, buffer)
    gl.glBindBuffer(gl.GL_ELEMENT_ARRAY_BUFFER, indicesBuffer)

    # bind the position attribute
    stride = data.strides[0]
    offset = ctypes.c_void_p(0)
    loc = gl.glGetAttribLocation(program, "position")
    gl.glEnableVertexAttribArray(loc)
    gl.glBindBuffer(gl.GL_ARRAY_BUFFER, buffer)
    gl.glVertexAttribPointer(loc, 3, gl.GL_FLOAT, False, stride, offset)

    offset = ctypes.c_void_p(data.dtype["position"].itemsize)
    loc = gl.glGetAttribLocation(program, "color")
    gl.glEnableVertexAttribArray(loc)
    gl.glBindBuffer(gl.GL_ARRAY_BUFFER, buffer)
    gl.glVertexAttribPointer(loc, 4, gl.GL_FLOAT, False, stride, offset)

    loc = gl.glGetUniformLocation(program, "rotationMatrix")
    gl.glUniformMatrix4fv(loc, 2, gl.GL_TRUE, rotation)

    loc = gl.glGetUniformLocation(program, "transformationMatrix")
    gl.glUniformMatrix4fv(loc, 1, gl.GL_TRUE, transformationMatrix)

    # Upload data
    gl.glBufferData(gl.GL_ARRAY_BUFFER, data.nbytes, data, gl.GL_DYNAMIC_DRAW)
    gl.glBufferData(gl.GL_ELEMENT_ARRAY_BUFFER, indicesData.nbytes, indicesData,
gl.GL_STATIC_DRAW)

def display():
    gl.glClear(gl.GL_COLOR_BUFFER_BIT)
    gl.glDrawElements(gl.GL_TRIANGLES, len(indicesData),gl.GL_UNSIGNED_INT, None)
    gl.glFlush()
    glut.glutSwapBuffers()

def reshape(width,height):
    gl.glViewport(0, 0, width, height)

def keyboard( key, x, y):
    if key == b'\x1b':
        os._exit(1)

# GLUT init
glut.glutInit()
glut.glutInitDisplayMode(glut.GLUT_DOUBLE | glut.GLUT_RGBA | glut.GLUT_DEPTH)
glut.glutCreateWindow('Graphics Window')
glut.glutReshapeWindow(800,800)
glut.glutReshapeFunc(reshape)
data, indicesData = generateCubeData()

# generate transformation matrix through paramter provision
transformationMatrix = generateTransforms()
initialize(transformationMatrix = transformationMatrix)

glut.glutDisplayFunc(display)
glut.glutPostRedisplay()
glut.glutKeyboardFunc(keyboard)
# enter the mainloop
glut.glutMainLoop()
```

2.3 Output

Upon execution of the above source code from the command line using **`python threedtransforms.py`** command produces the following output:



This output cube is the same one that we initially assumed to be the base cube with no further transformations applied to it. This is because in the general program we have not provided any input parameters to the **`generateTransforms()`** function.

```
# generate transformation matrix through paramter provision
transformationMatrix = generateTransforms()
```

Looking at the definition of **`generateTransforms()`** function, we can see that the transformation matrices are set to identity matrix by default meaning that upon no **`transformationType`** our **`transformationMatrix`** will return and perform identity transformation to our shape.

```
def generateTransforms(transformationType = None, transformationData = None):

    transformationMatrix = np.identity(4, dtype = np.float32)
    transformationMatrix = transformationMatrix.flatten()

    if not transformationType:
        return transformationMatrix

initialize(transformationMatrix = transformationMatrix)
```

Now, in the upcoming sections, we will be varying these transformation matrices using our **`generateTransforms()`** helper function and pass it whilst calling the initialize function to achieve our desired transformations.

Chapter 3: Translation

3.1 Matrix

To translate our Cube 0.15 in x-direction, 0.2 in y-direction and 0.2 in z-direction we generate the following 4 x 4 Homogeneous Translation Matrix:

1	0	0	Tx
0	1	0	Ty
0	0	1	Tz
0	0	0	1

where, Tx = 0.15, Ty = 0.2, Tz = 0.2

3.2 Code Snippet for Translation

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = generateTransforms("translation", [0.15, 0.2, 0.2])

# Pass the Transformation Matrix and Translation Matrix
# as Parameters to initialize if need be
initialize(transformationMatrix = transformationMatrix)
```

Description

Our code will produce the following translation numpy matrix:

```
transformationMatrix = np.array([  1.0,0.0,0.0,0.15,
                                   0.0,1.0,0.0,0.2,
                                   0.0,0.0,1.0,0.2,
                                   0.0,0.0,0.0,1.0], np.float32)
```

And, pass it to **`transformationMatrix`** parameter of the **`initialize()`** function during the call, which then feeds the matrix to the vertex shader code. Eventually, the vertex shader handles the translation through matrix multiplication.

3.3 Output

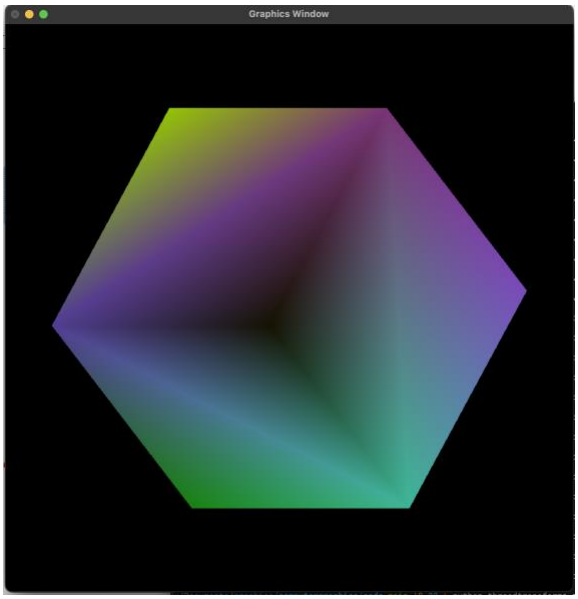


fig 3.3.1 : Before Translation

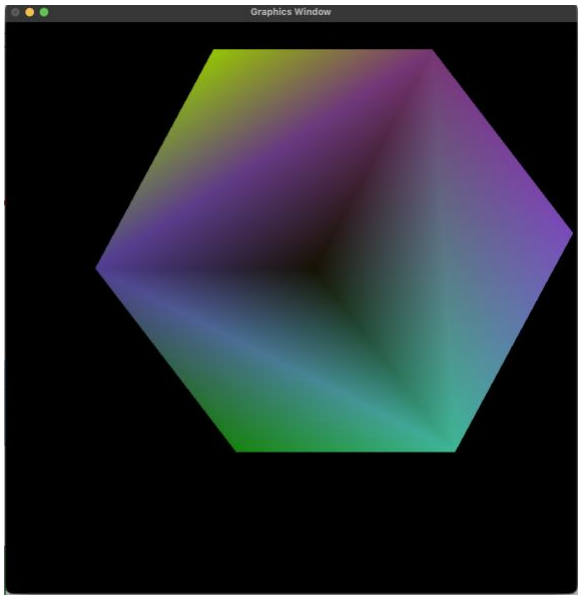


fig 3.3.2 : After Translation [0.15, 0.2, 0.2]

We can see that the Cube has been translated into x and y direction by the mentioned amount. But, we see no difference due to the z direction translation. This is because the view we are looking the Cube from is using parallel projection so the size of the object remains the same despite the change in the z-direction values for the vertices. Since, all the vertices data are translated in z direction by same amount, the actual values at z buffer does change. However, they change by the same amount, meaning that there appears to be no changes in the cube faces and z direction based parameters.

Chapter 4: Rotation

4.1 Matrix

We can rotate the Cube about 3 primary axes, namely: x-axis (pitch), y-axis (yaw), and z-axis (roll). To rotate our Cube 30 degrees in each axis we generate the following 4 x 4 Homogeneous Rotation Matrices:

Rotation about x-axis

1	0	0	0
0	cθ	-sθ	0
0	sθ	cθ	0
0	0	0	1

Rotation about y-axis

cθ	0	sθ	0
0	1	0	0
-sθ	0	cθ	0
0	0	0	1

Rotation about z-axis

cθ	-sθ	0	0
sθ	cθ	0	0
0	0	1	0
0	0	0	1

where, cθ = cos(θ) and sθ = sin(θ) and θ = 30 degrees

4.2 Code Snippet for Rotation

Rotation about x-axis (Pitch)

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = generateTransforms("rotation", [“pitch”, 30])

# Pass the Transformation Matrix and Translation Matrix
# as Parameters to initialize if need be
initialize(transformationMatrix= transformationMatrix)
```

Rotation about y-axis (Yaw)

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = generateTransforms("rotation", [“yaw”. 30])

# Pass the Transformation Matrix and Translation Matrix
# as Parameters to initialize if need be
initialize(transformationMatrix= transformationMatrix)
```

Rotation about z-axis (Roll)

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = generateTransforms("rotation", [“roll”, 30])

# Pass the Transformation Matrix and Translation Matrix
# as Parameters to initialize if need be
initialize(transformationMatrix= transformationMatrix)
```

Description

Our code will produce the following rotation numpy matrices:

Rotation about x-axis (Pitch)

```
transformationMatrix = np.array([  1.0,0.0,0.0,0.0,
                                   0.0, 0.866,-0.5,0.0,
                                   0.0,0.5,0.866,0.0,
                                   0.0,0.0,0.0,1.0], np.float32)
```

Rotation about y-axis (Yaw)

```
transformationMatrix = np.array([  0.866,0.0,0.5,0.0,
                                   0.0,1.0,0.0,0.0,
                                   -0.5,0.0,0.866,0.0,
                                   0.0,0.0,0.0,1.0], np.float32)
```

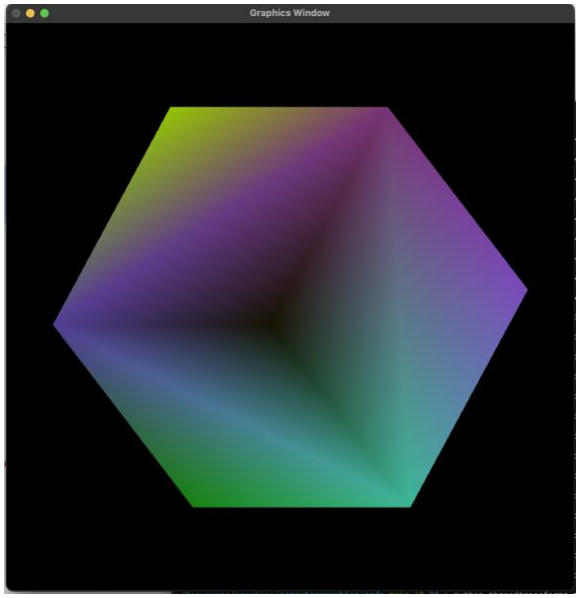
Rotation about z-axis (Roll)

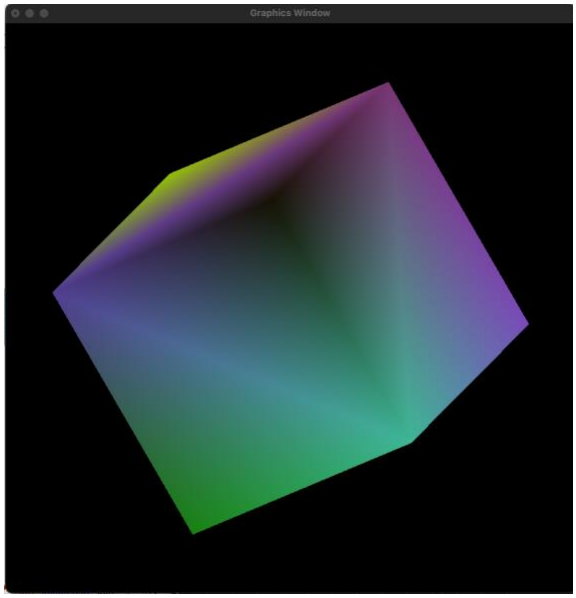
```
transformationMatrix = np.array([  0.866,-0.5,0.0,0.0,
                                   0.5,0.866,0.0,0.0,
                                   0.0,0.0,1.0,0.0,
                                   0.0,0.0,0.0,1.0], np.float32)
```

And, pass it to **`transformationMatrix`** parameter of the **`initialize()`** function during the call, which then feeds the matrix to the vertex shader code. Eventually, the vertex shader handles the rotation about the respective axes through matrix multiplications.

4.3 Output

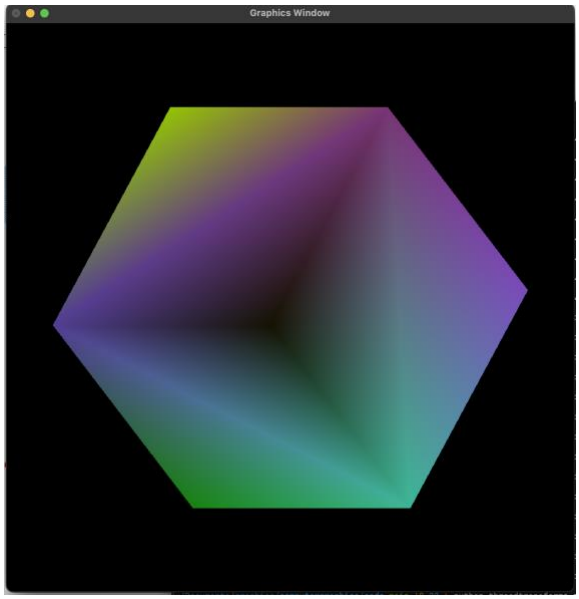
Rotation about x-axis (Pitch)

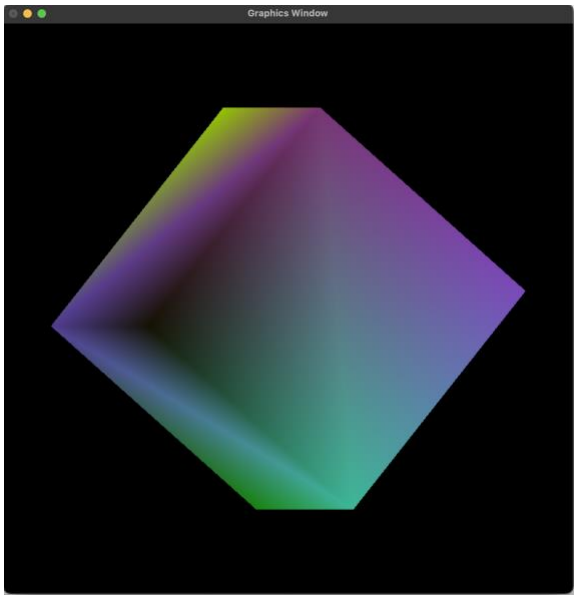




.. *fig 4.3.1 : Before Rotation* . *fig 4.3.2 : After 30 degrees Rotation about x-axis*

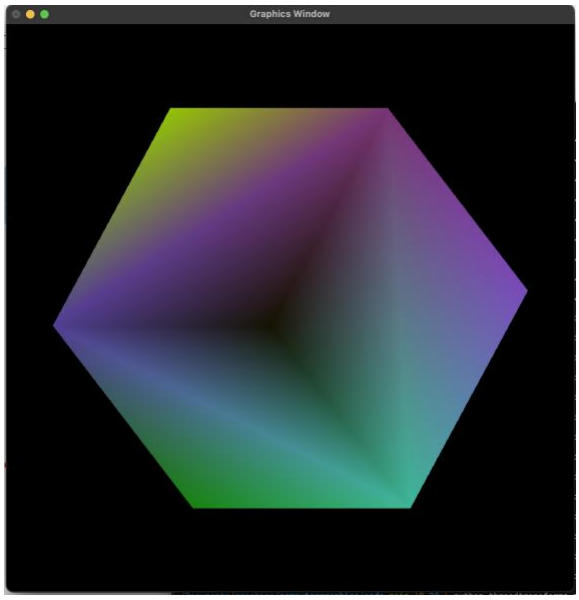
Rotation about y-axis (yaw)

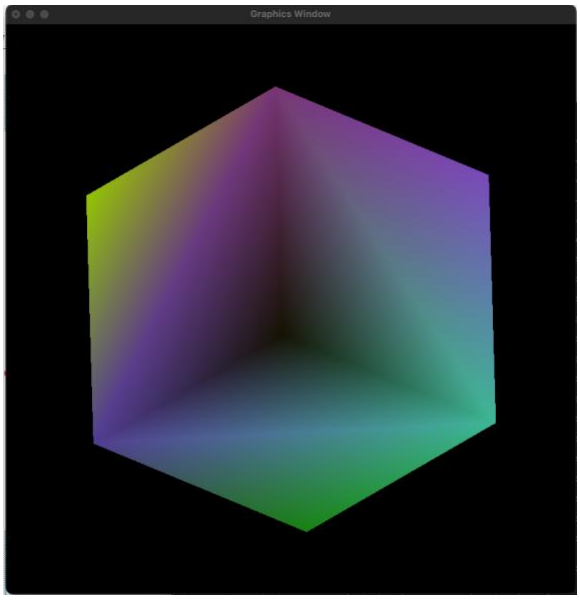




.. *fig 4.3.3 : Before Rotation* . *fig 4.3.4 : After 30 degrees Rotation about y-axis*

Rotation about z-axis (Roll)





.. fig 4.3.5 : Before Rotation . *fig 4.3.6 : After 30 degrees Rotation about z-axis*

Chapter 5: Scaling

5.1 Matrix

To Scale our Cube by 0.8 times in x-direction, 0.6 times in y-direction and 0.6 times in z-direction we generate following 4 x 4 Homogeneous Translation Matrix and Scaling Matrix:

Scaling Matrix

Sx	0	0	0
0	Sy	0	0
0	0	0	0
0	0	0	1

where, Sx = 0.8, Sy = 0.6 and Sz = 0.6

5.2 Code Snippet for Scaling

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = generateTransforms("scaling", [0.8, 0.6, 0.6])

# Pass the Transformation Matrix and Translation Matrix
# as Parameters to initialize if need be
initialize(transformationMatrix = transformationMatrix)
```

Description

Our code will produce the following translation numpy matrix:

```
transformationMatrix = np.array([  0.8,0.0,0.0,0.0,
                                   0.0,0.6,0.0,0.0,
                                   0.0,0.0,0.6,0.0,
                                   0.0,0.0,0.0,1.0], np.float32)
```

And, pass it to **`transformationMatrix`** and **`translationMatrix`** parameter of the **`initialize()`** function during the call, which then feeds these matrices to the vertex shader code. Eventually, the vertex shader handles scaling of the cube through matrix multiplication.

5.3 Output

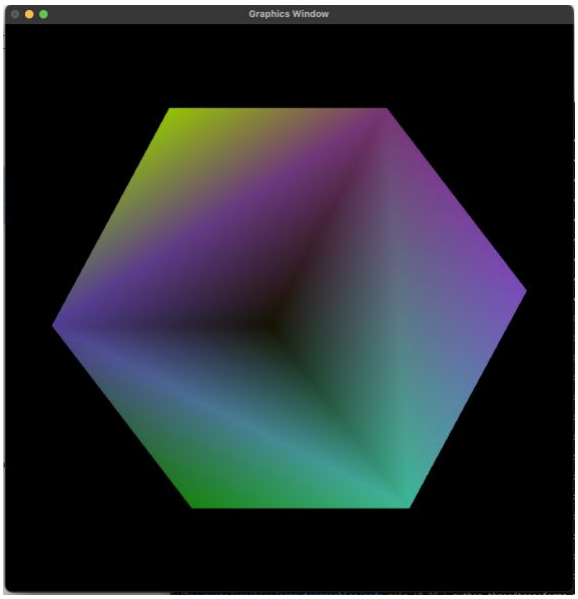


fig 5.3.1 : Before Scaling

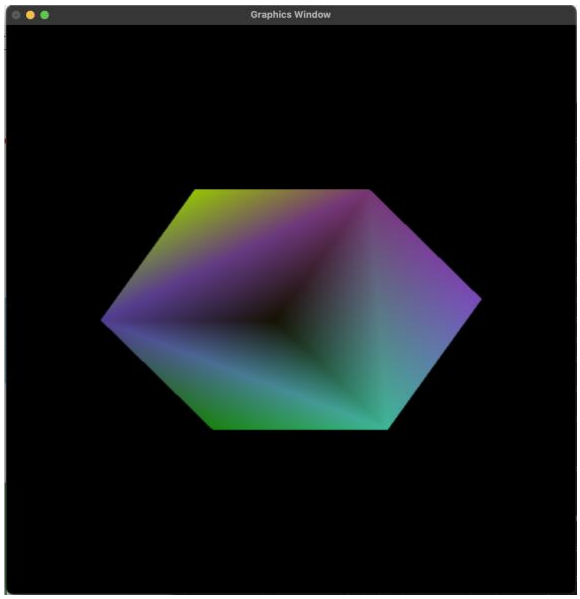


fig 5.3.2 : After Scaling [0.8, 0.6, 0.6]

Chapter 6: Conclusion

Through this Lab Work, I was able to realize the ease of using homogeneous coordinate and matrix system for implementing various transformations for a 3D Shape (Cube) in three-dimensional space. Moreover, I was also able to picturize the unrealistic appearance of a 3D Shape portrayed using parallel projection method, and the need to have an implementation for perspective projections for all 3D Graphical Systems.

Also, GLSL (OpenGL Shading Language) is written for the purpose of graphical manipulation, it automatically handles the matrix multiplications meaning transformations of a shape only needed the correct creation of transformation matrix and identification of correct order for matrix multiplications. Even though, achieving various transformations were simple, this comprehensive lab work will be extremely crucial for the implementation of our selected topic for the mini project.