# Kathmandu University

## Department of Computer Science and Engineering

## Dhulikhel, Kavre


**Two Dimensional Transformations**

**Lab Report Four**

**[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: Two Dimensional Transformation

## 1.1 Introduction

Two Dimensional Transformations are functions which when applied to a two-dimensional point (or 2D shape, constructed using 2D points) will map it into another two-dimensional point (or 2D shape, made out of the mapped 2D points). Essentially, Two Dimensional Transformations map a set of Original Points **O** of a **2D Space** into a set of Transformed Points **T** into the same **2D space**.

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

- Translation
- Rotation
- Scaling
- Reflection
- Shearing

## 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

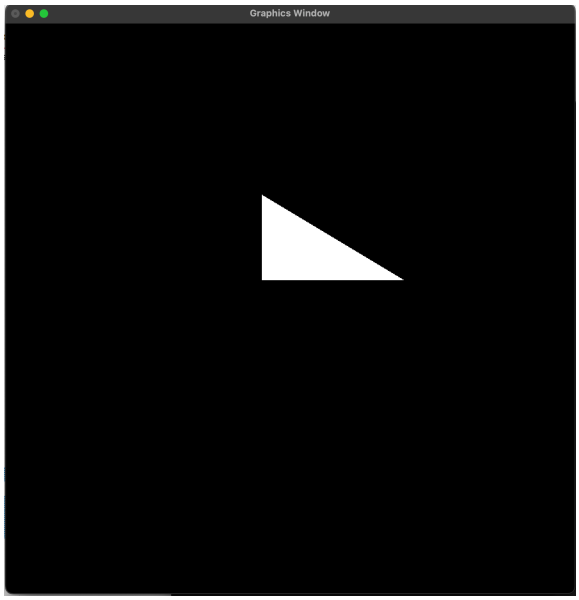
## 1.3 Helper Functions

For the purpose of demonstrating the effect of various transformations, we will be using a 2D Triangle with vertices at **(-75, -50), (75, -50), and (0, 50)** in a window **(-250, -250, 250, 250)**. The triangle is drawn using the OpenGL's gl.GL_TRIANGLES primitive, and the helper function **buildData()** is used for the generation of these data points.

### Code Snippet for Build Data Helper and Triangles Primitive

```
# -- Building Data --
def buildData():

    if len(sys.argv) >= 3:
        resolution = [int(sys.argv[1]), int(sys.argv[2])]
    else:
        resolution = [500,500]

    data = [[-25, 25, 0],
            [100, -25, 0],
            [-25, 100, 0]]

    return data, resolution

# -- Using GL_TRIANGLES Primitive --
def display():
    gl.glClear(gl.GL_COLOR_BUFFER_BIT)
    gl.glDrawArrays(gl.GL_TRIANGLES, 0, data.shape[0])
    glut.glutSwapBuffers()
```

### Output Triangle



**Moreover,** we will be utilizing the `transformationFunction()` 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 5 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]`

b. Rotation : `transformationData = [degree of rotation]`

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

d. Reflection : `transformationData = [reflection axis (x or y or origin)]`

e. Shearing : `transformationData = [shearing axis (x or y), shear amount]`

### Code Snippet for Transformation Function Helper

```
def transformationFunction(transformationType, transformationData):

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

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

    elif transformationType == "rotation":
        cTheta = np.cos(transformationData[0]/180 * math.pi)
        sTheta = np.sin(transformationData[0]/180 * math.pi)

        transformationMatrix = np.array([   cTheta, -sTheta ,0.0,0.0,
                                            sTheta, cTheta,0.0,0.0,
                                            0.0,0.0,0.0,0.0,
```

```
                                    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,
                                        0.0,0.0,0.0,0.0,
                                        0.0,0.0,0.0,1.0], np.float32)

elif transformationType == "reflection":

    if transformationData[0] == "y":
        transformationMatrix = np.array([   -1.0,0.0,0.0,0.0,
                                            0.0,1.0,0.0,0.0,
                                            0.0,0.0,0.0,0.0,
                                            0.0,0.0,0.0,1.0], np.float32)
    elif transformationData[0] == "x":
        transformationMatrix = np.array([   1.0,0.0,0.0,0.0,
                                            0.0,-1.0,0.0,0.0,
                                            0.0,0.0,0.0,0.0,
                                            0.0,0.0,0.0,1.0], np.float32)
    elif transformationData[0] == "xy":
        transformationMatrix = np.array([   0.0,1.0,0.0,0.0,
                                            1.0,0.0,0.0,0.0,
                                            0.0,0.0,0.0,0.0,
                                            0.0,0.0,0.0,1.0], np.float32)
    else:
        transformationMatrix = np.array([   -1.0,0.0,0.0,0.0,
                                            0.0,-1.0,0.0,0.0,
                                            0.0,0.0,0.0,0.0,
                                            0.0,0.0,0.0,1.0], np.float32)
elif transformationType == "shearing":
    if transformationData[0] == "y":
        transformationMatrix = np.array([   1.0,0.0,0.0,0.0,
                                            transformationData[1],1.0,0.0,0.0,
                                            0.0,0.0,0.0,0.0,
                                            0.0,0.0,0.0,1.0], np.float32)
    else:
        transformationMatrix = np.array([   1.0,transformationData[1],0.0,0.0,
                                            0.0,1.0,0.0,0.0,
                                            0.0,0.0,0.0,0.0,
                                            0.0,0.0,0.0,1.0], np.float32)

return transformationMatrix
```

**Similarly,** for a 2D Transformation we were accustomed to using 3 x 3 Homogeneous Matrix to represent the Transformation Matrix. However, since we are using a 3D Graphics Library with z value set to 0 and no depth buffer enabled, we need to use modify our 3 x 3 Homogeneous Matrix to function with our Graphics Library setup. In order to do so, will be constructing using a 4 x 4 Homogeneous Matrix using the 3 x 3 Homogeneous Matrix by adding **all 0 third-row and third-column** as the values for z-direction.

For example, our 3 x 3 Homogeneous Translation Matrix for 2D Space would be converted to a 4 x 4 Homogeneous Translation Matrix as:

| | | |
|---|---|---|
| 1 | 0 | Tx |
| 0 | 1 | Ty |
| 0 | 0 | 1 |

⇨

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | Tx |
| 0 | 1 | 0 | Ty |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

**In addition,** the vertices generated for our triangle are discrete integer values. However, the Modern OpenGL approach requires the coordinates to be in Normalized form from `(-1,-1) to (1, 1)`. So, for conversion of the generated vertices to normalized form, I have created a helper function named `tonormalized`. It takes in the generated vertices and the screen resolution for which the data has been generated as inputs and returns the normalized coordinates needed for our Graphics Library as its output.

### Code Snippet for `tonormalized`

```
def tonormalized(coordinates, resolution):
    for coordinate in (coordinates):
        coordinate[0] = coordinate[0] * 2 / (resolution[0])
        coordinate[1] = coordinate[1] * 2 / (resolution[1])

    return np.array(coordinates, dtype = np.float32)
```

# Chapter 2: General 2D 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 2D 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

d. Reflection

e. Shearing

## 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

vertexShaderCode = """
    attribute vec3 position;
    uniform mat4 transformMatrix;
    uniform mat4 translate;

    mat4 retranslate;

    void main(){

        retranslate = translate;
        retranslate[0][3] *= -1.0;
        retranslate[1][3] *= -1.0;

        gl_Position = retranslate * transformMatrix * translate * vec4(position, 1.0);
    }
    """

fragmentShaderCode = """
    uniform vec4 vColor;
    void main(){
        gl_FragColor = vColor;
    }
    """
# -- Building Data --
def buildData():

    if len(sys.argv) >= 3:
        resolution = [int(sys.argv[1]), int(sys.argv[2])]
    else:
```

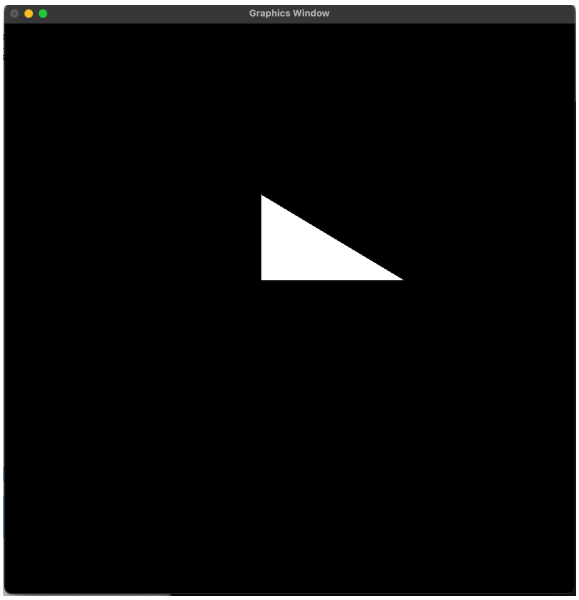
```
        resolution = [500,500]

    data = [[-25, 25, 0],
            [100, 25, 0],
            [-25, 100, 0]]

    return data, resolution


# normalization function
def tonormalized(coordinates, resolution):
    for coordinate in (coordinates):
        coordinate[0] = coordinate[0] * 2 / (resolution[0])
        coordinate[1] = coordinate[1] * 2 / (resolution[1])

    return np.array(coordinates, dtype = np.float32)

# helper function to generate our required transformation matrix
def transformationFunction(transformationType, transformationData):

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

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

    elif transformationType == "rotation":
        cTheta = np.cos(transformationData[0]/180 * math.pi)
        sTheta = np.sin(transformationData[0]/180 * math.pi)

        transformationMatrix = np.array([   cTheta, -sTheta ,0.0,0.0,
                                            sTheta, cTheta,0.0,0.0,
                                            0.0,0.0,0.0,0.0,
                                            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,
                                            0.0,0.0,0.0,0.0,
                                            0.0,0.0,0.0,1.0], np.float32)

    elif transformationType == "reflection":

        if transformationData[0] == "y":
            transformationMatrix = np.array([   -1.0,0.0,0.0,0.0,
                                                0.0,1.0,0.0,0.0,
                                                0.0,0.0,0.0,0.0,
                                                0.0,0.0,0.0,1.0], np.float32)
        elif transformationData[0] == "x":
            transformationMatrix = np.array([   1.0,0.0,0.0,0.0,
                                                0.0,-1.0,0.0,0.0,
                                                0.0,0.0,0.0,0.0,
                                                0.0,0.0,0.0,1.0], np.float32)
        elif transformationData[0] == "xy":
            transformationMatrix = np.array([   0.0,1.0,0.0,0.0,
                                                1.0,0.0,0.0,0.0,
                                                0.0,0.0,0.0,0.0,
                                                0.0,0.0,0.0,1.0], np.float32)
        else:
            transformationMatrix = np.array([   -1.0,0.0,0.0,0.0,
                                                0.0,-1.0,0.0,0.0,
                                                0.0,0.0,0.0,0.0,
                                                0.0,0.0,0.0,1.0], np.float32)
    elif transformationType == "shearing":
        if transformationData[0] == "y":
            transformationMatrix = np.array([   1.0,0.0,0.0,0.0,
                                                transformationData[1],1.0,0.0,0.0,
                                                0.0,0.0,0.0,0.0,
                                                0.0,0.0,0.0,1.0], np.float32)
        else:
            transformationMatrix = np.array([   1.0,transformationData[1],0.0,0.0,
```

```
                                                0.0,1.0,0.0,0.0,
                                                0.0,0.0,0.0,0.0,
                                                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 = np.identity(4, np.float32), translationMatrix =
np.identity(4, np.float32)):
    global program
    global data

    gl.glClear(gl.GL_COLOR_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),
    )

    # make program the default program
    gl.glUseProgram(program)

    buffer = gl.glGenBuffers(1)

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

    # 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)

    loc = gl.glGetUniformLocation(program, "vColor")
```

```
    gl.glUniform4fv(loc, 1, [1.0,1.0,1.0,1.0])

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

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


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


def display():
    gl.glClear(gl.GL_COLOR_BUFFER_BIT)
    gl.glDrawArrays(gl.GL_TRIANGLES, 0, data.shape[0])
    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.glutCreateWindow('Graphics Window')
glut.glutReshapeWindow(800,800)
glut.glutReshapeFunc(reshape)

data, resolution = buildData()
data = tonormalized(data, resolution)

# Calculate the Transformation Matrix Here as follows :

# transformationMatrix = transformationFunction("rotation", [30])
# translationMatrix = np.array([   1.0,0.0,0.0, -data[0][0],
#                                  0.0,1.0,0.0, -data[0][1],
#                                  0.0,0.0,1.0,0.0,
#                                  0.0,0.0,0.0,1.0], np.float32)

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

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 twodtransforms.py` command produces the following output:




This output triangle is the same one that we initially created with no transformations applied to it. This is because in the general program we have not provided any input parameters to the **`initialize()`** function.

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

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

```
def initialize(transformationMatrix = np.identity(4, np.float32), translationMatrix =
np.identity(4, np.float32))
```

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

# Chapter 3: Translation

## 3.1 Matrix

To translate our Triangle 0.2 in x-direction and 0.5 in y-direction we generate the following 4 x 4 Homogeneous Translation Matrix:

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | Tx |
| 0 | 1 | 0 | Ty |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

where, Tx = 0.2 and Ty = 0.5

## 3.2 Code Snippet for Translation

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = transformationFunction("translation", [0.2, 0.5])

# 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([  1.0,0.0,0.0,0.2,
                                   0.0,1.0,0.0,0.5,
                                   0.0,0.0,0.0,0.0,
                                   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 translation through matrix multiplication.

## 3.3 Output

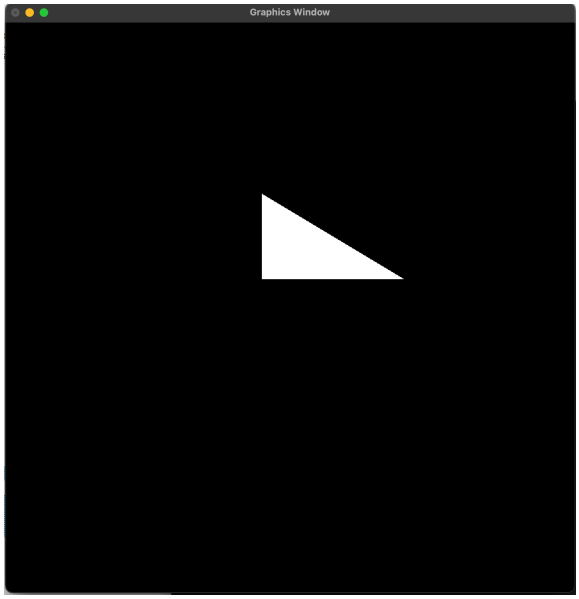



*fig 3.3.1 : Before Translation*

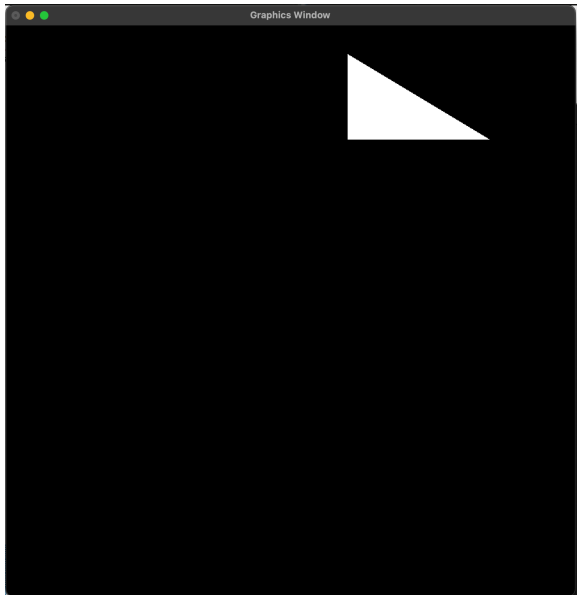



*fig 3.3.2 : After Translation [0.2, 0.5]*

# Chapter 4: Rotation

## 4.1 Matrix

To rotate our Triangle 45 degrees about the origin in anti-clockwise direction we generate the following 4 x 4 Homogeneous Rotation Matrix:

| | | | |
|---|---|---|---|
| cθ | -sθ | 0 | 0 |
| sθ | cθ | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

where, cθ = cos(θ) and sθ = sin(θ) and θ = 45 degrees

## 4.2 Code Snippet for Rotation

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = transformationFunction("rotation", [45])

# 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 matrix:

```
transformationMatrix = np.array([  0.707,-0.707,0.0,0.0,
                                   0.707,0.707,0.0,0.0,
                                   0.0,0.0,0.0,0.0,
                                   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 through matrix multiplication.

## 4.3 Output

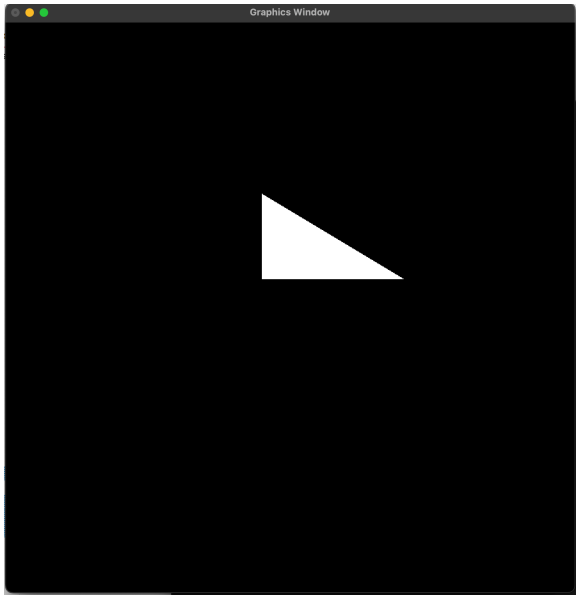



*fig 4.3.1 : Before Rotation*

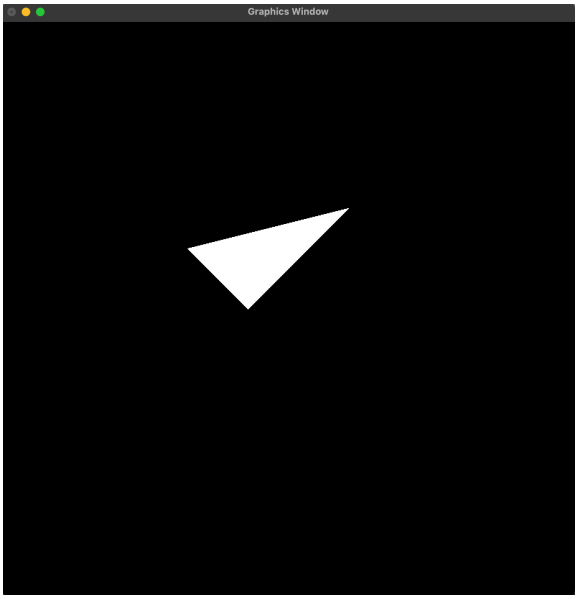



*fig 4.3.2 : After Rotation of 45 degrees*

# Chapter 5: Scaling

## 5.1 Matrix

To scale our Triangle 2 times in x-direction and 1.4 times in y-direction about its vertex `(-75, -50)` we generate following 4 x 4 Homogeneous Translation Matrix and Scaling Matrix:

**Translation Matrix**

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | -Tx |
| 0 | 1 | 0 | -Ty |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

where, (Tx, Ty) = normalized (-75, -50)

**Scaling Matrix**

| | | | |
|---|---|---|---|
| Sx | 0 | 0 | 0 |
| 0 | Sy | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

where, Sx = 2 and Sy = 1.2

## 5.2 Code Snippet for Scaling

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = transformationFunction("scaling", [2, 1.2])
translationMatrix = np.array([  1.0,0.0,0.0, -data[0][0],
                                0.0,1.0,0.0, -data[0][1],
                                0.0,0.0,1.0,0.0,
                                0.0,0.0,0.0,1.0], np.float32)

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

### Description

Our code will produce the following Scaling numpy matrix:

```
transformationMatrix = np.array([  2.0,0.0,0.0,0.0,
                                   0.0,1.2,0.0,0.0,
                                   0.0,0.0,0.0,0.0,
                                   0.0,0.0,0.0,1.0], np.float32)
```

And, the following Translation numpy matrix:

```
translationMatrix = np.array([  1.0,0.0,0.0,0.15,
                                0.0,1.0,0.0,0.10,
                                0.0,0.0,0.0,0.0,
                                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 the translation of the triangle to the origin, scaling of the triangle and the retranslation of the triangle by bringing the translated vertex to its original position through matrix multiplication.

## 5.3 Output

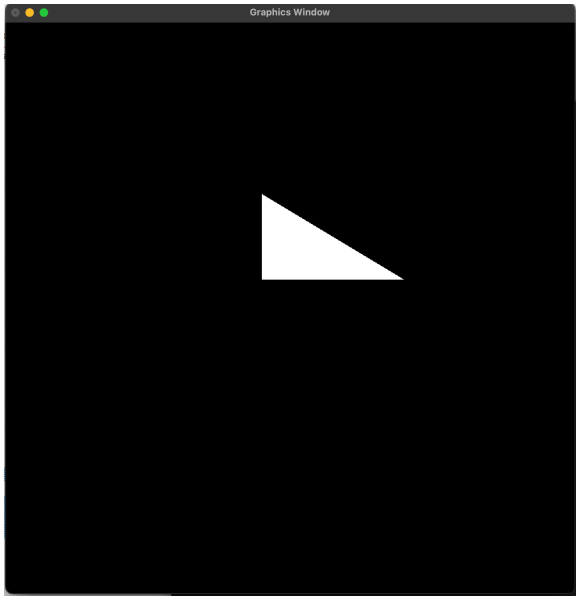



*fig 5.3.1 : Before Scaling*

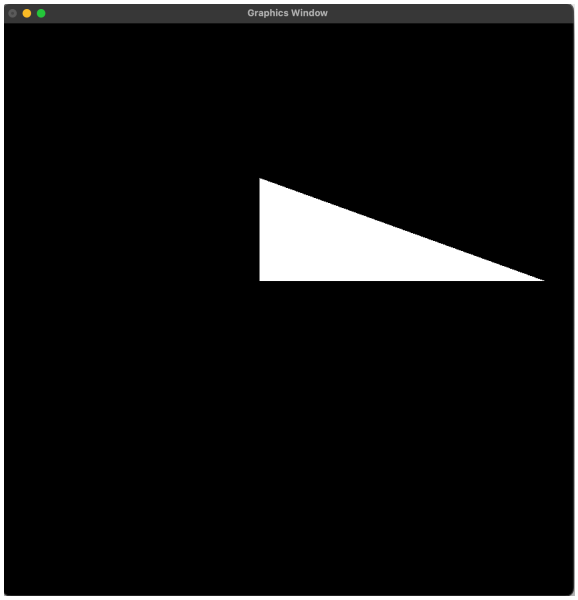



*fig 5.3.2 : After Scaling [2, 1.2] about a vertex*

# Chapter 6: Reflection

## 6.1 Matrix

We can reflect our triangle about 4 axes, namely: x-axis, y-axis, x = y line and origin. To reflect our Triangle using one of the 4 axes we generate following 4 x 4 Homogeneous Reflection Matrices depending upon our axis of reflection:

**Reflection X axis:**

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | -1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

**Reflection Y axis:**

| | | | |
|---|---|---|---|
| -1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

**Reflection X=Y Line:**

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

**Reflection Origin:**

| | | | |
|---|---|---|---|
| -1 | 0 | 0 | 0 |
| 0 | -1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

## 6.2 Code Snippet for Reflection

### Reflection about x-axis

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = transformationFunction("reflection", [“x”])

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

### Reflection about y-axis

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = transformationFunction("reflection", [“y”])

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

### Reflection about x = y line

```
# Calculate the Transformation Matrix as follows:
```

```
transformationMatrix = transformationFunction("reflection", [“xy”])

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

### Reflection about origin

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = transformationFunction("reflection", [“origin”])

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

## Description

Our code will produce the following Reflection numpy matrix depending on our axis choice:

### Reflection about x-axis

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

### Reflection about y-axis

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

### Reflection about x = y line

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

### Reflection about origin

```
transformationMatrix = np.array([  -1.0,0.0,0.0,0.0,
                                   0.0,-1.0,0.0,0.0,
                                   0.0,0.0,0.0,0.0,
                                   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 these matrices to the vertex shader code. Eventually, the vertex shader handles the reflection of the triangle about the specified axis through matrix multiplication.

## 6.3 Output

### Reflection about x-axis

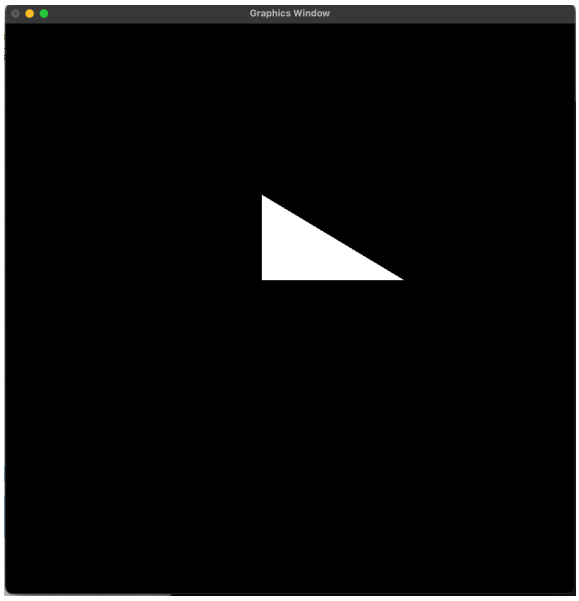



*fig 6.3.1 : Before Reflection*

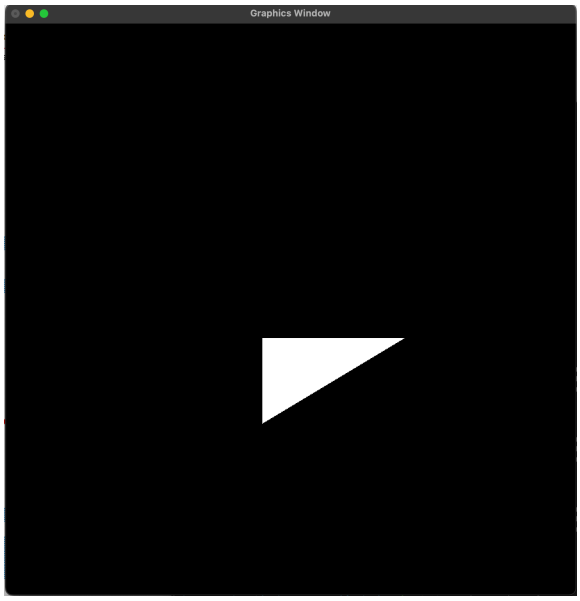



*fig 6.3.2 : After Reflection about x-axis*

### Reflection about y-axis

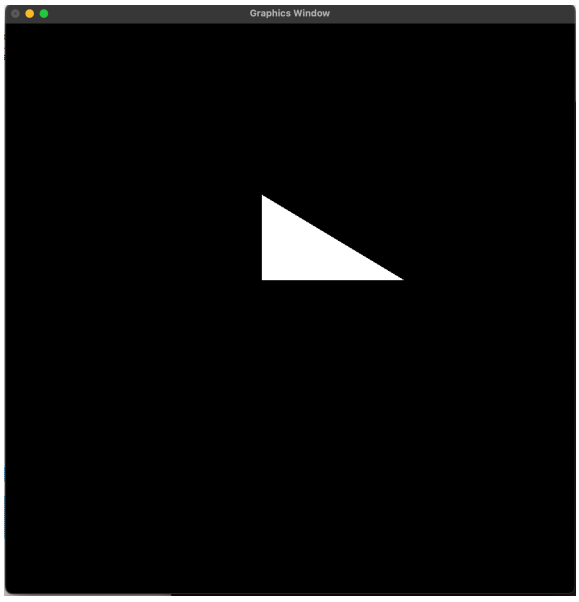



*fig 6.3.4 : Before Reflection*

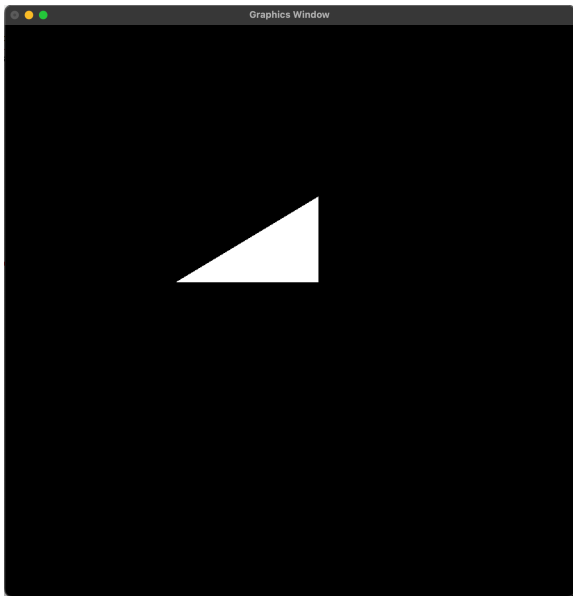



*fig 6.3.5 : After Reflection about y-axis*

**Reflection about x = y line**

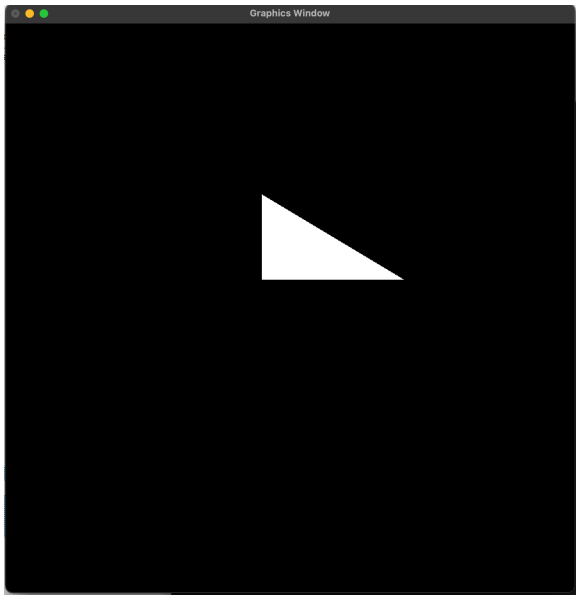



*fig 6.3.5 : Before Reflection*

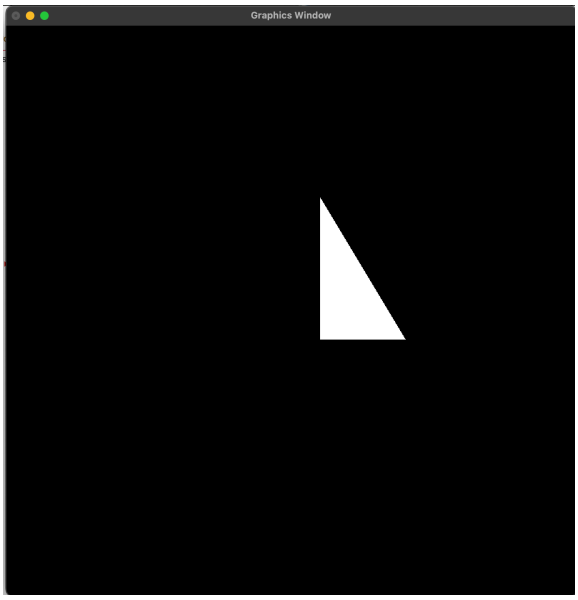



*fig 6.3.6 : After Reflection about x = y*

**Reflection about origin**

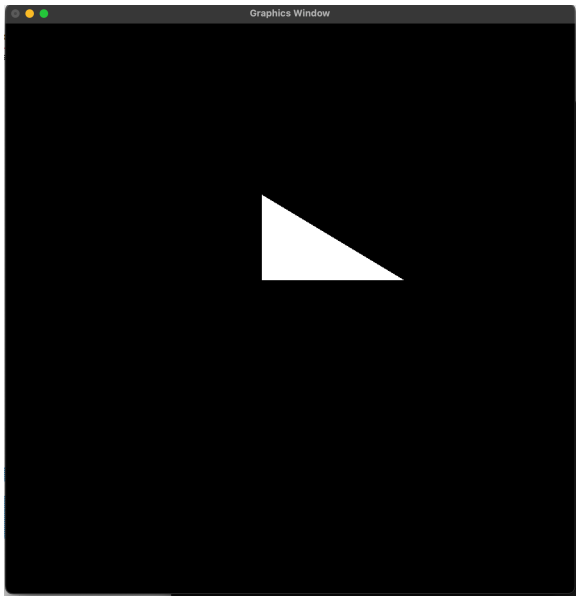



*fig 6.3.7 : Before Reflection*

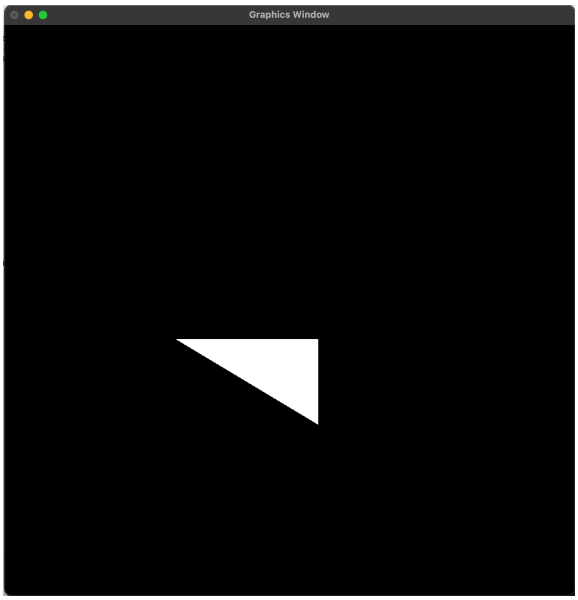



*fig 6.3.8 : After Reflection about origin*

# Chapter 7: Shearing

## 7.1 Matrix

We can shear our triangle in two directions, namely: x-direction, and y-direction. To shear our Triangle using one of the two directions we generate following 4 x 4 Homogeneous Shearing Matrices:

**Shearing in X Direction:**

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| Shx | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

where, Shx = 2 : Shearing factor in x direction and,

**Shearing in Y direction:**

| | | | |
|---|---|---|---|
| 1 | Shy | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

Shy = 2 : Shearing factor in y direction

## 7.2 Code Snippet for Shearing

### Shearing in x-direction

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = transformationFunction("shearing", [“x”, 2])

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

### Shearing in y-direction

```
# Calculate the Transformation Matrix as follows:
transformationMatrix = transformationFunction("shearing", [“y”, 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 Shearing numpy matrix depending on our axis choice:

### Shearing in x-direction

```
transformationMatrix = np.array([  1.0,0.0,0.0,0.0,
                                   2.0,1.0,0.0,0.0,
                                   0.0,0.0,0.0,0.0,
                                   0.0,0.0,0.0,1.0], np.float32)
```

### Shearing in y-direction

```
transformationMatrix = np.array([  1.0,2.0,0.0,0.0,
                                   0.0,1.0,0.0,0.0,
                                   0.0,0.0,0.0,0.0,
                                   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 these matrices to the vertex shader code. Eventually, the vertex shader handles the shearing of the triangle in the specified direction through matrix multiplication.

## 7.3 Output

### Shearing in x-direction

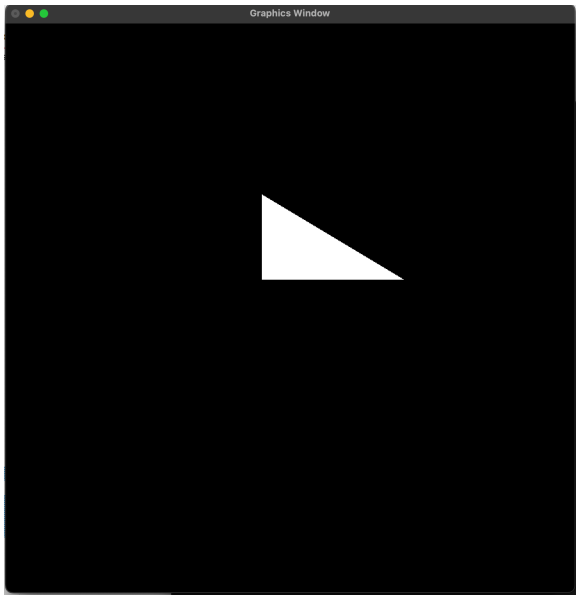



*fig 7.3.1 : Before Shearing*

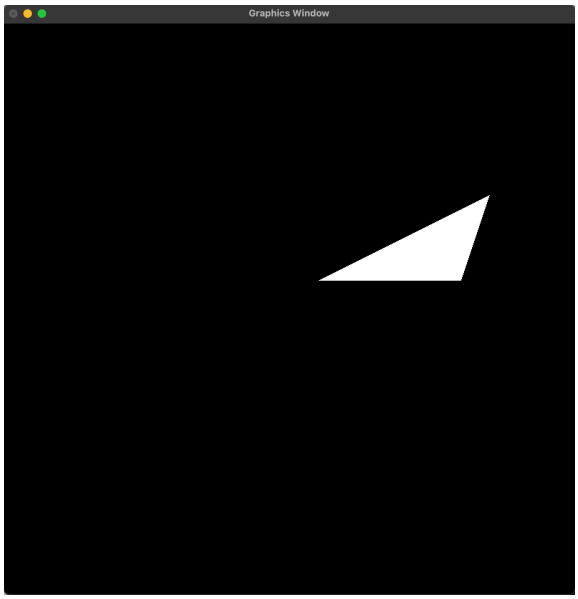



*fig 7.3.2 : After Shearing [2] in x-direction*

**Shearing in y-direction**

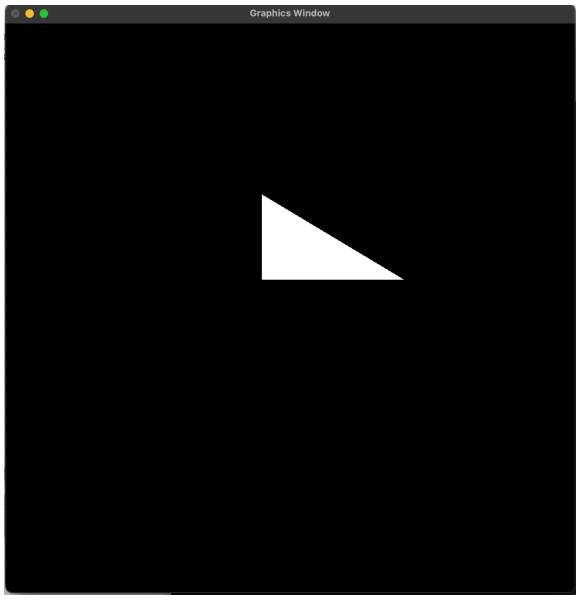



*fig 7.3.3 : Before Shearing*


*fig 7.3.4 : After Shearing [2] in y-direction*

# Chapter 8: 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 2D Shape (Triangle) in two-dimensional space. Since, 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.