Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report III

[COMP 342]

Submitted by:

Abhijeet Poudel

CS(III/I)

Roll No: 44

Submitted to:

Mr. Dhiraj Shrestha

Department of Computer Science and Engineering

Submission Date: 18th December 2022

Table of Contents

Chapter 1: Introduction **4**

Chapter 2: The Two-Dimensional Object and Preliminaries **4**

Chapter 3: 2D Translation **5**

3.1. Translation Matrix 5

3.2. Source Code 6

3.3 Output 9

3.3.1. Original Triangle 9

3.3.2. Triangle after translation of 0.5 9

Chapter 4: 2D Rotation **10**

4.1. Rotation Matrix 10

4.2. Source Code 10

4.3. Output 13

4.3.1. Original Triangle 13

4.3.2. Triangle after Rotation by 45 degree 13

Chapter 5: 2D Scaling **14**

5.1. Scaling Matrix 14

5.2. Source Code 15

5.3. Output 17

5.3.1. Original Triangle 17

5.3.2. Scaled Triangle with scaling factor 1.5 18

5.3.3. Scaled Triangle with Scaling factor 1.5 along x-axis 19

5.3.4. Scaled Triangle with Scaling factor 1.5 along y-axis 19

Chapter 6: Reflection on 2D Objects **20**

6.1. Reflection Matrix 20

6.2. Source Code 20

6.3. Output 23

6.3.1. Original Triangle 23

6.3.2. Reflection about x-axis 23

6.3.2. Reflection about y-axis 23
Chapter 7: 2D Shearing 24
7.1. Shearing Matrix 24
7.2. Source Code 25
7.3. Output 28
7.3.1. Original Triangle 28
7.3.2. Triangle sheared on x-axis 28
7.3.3. Triangle sheared along y-axis 29
Chapter 8: Conclusion 30

Chapter 1: Introduction

For our lab work and project, the following programming language and tools are used:

Graphics Library: PyOpenGL 3.1.6

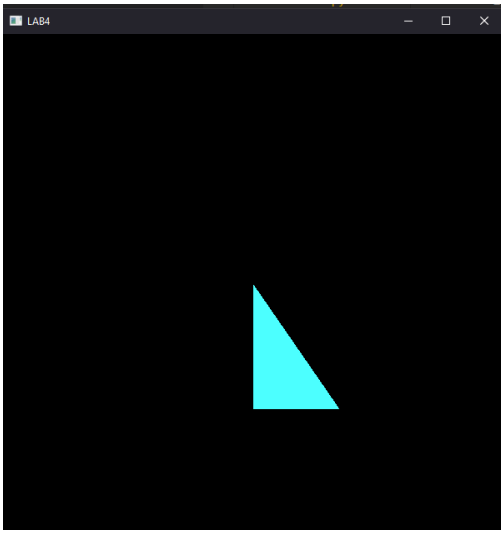
Programming Language: Python 3.10.5

Window Context: GLFW

Helpers: Numpy

Chapter 2: The Two-Dimensional Object and Preliminaries

The two-dimensional object that we will be using for this demonstration is a simple triangle that is drawn with the help of the primitive GL_TRIANGLES.



We will use a 4 by 4 homogenous matrix to translate the points of the two-dimensional object and hence, set the value of the z to 0 and proceed normally i.e., generate a 4x4 translation matrix and multiply the matrix with the vertices of the object, a triangle in this case. We will multiply the generated transformation matrix with the object vertices inside the vertex shader and fragment as follows:

```
  vertex_src = """
#version 330

layout(location=0) in vec3 aPos;
uniform mat4 transformation;

void main(){
    gl_Position = transformation * vec4(aPos,1.0);

    }
"""

  fragment_src = """
#version 330

out vec4 FragColor;

void main(){
    FragColor = vec4(1.0f,0.0f,0.3f,0.0f);

}
"""
```

Here, the transform matrix is multiplied with the 2D object which has also been made into a 4D matrix by setting the z axis to 0

Chapter 3: 2D Translation

3.1. Translation Matrix

In order to translate any 2D object the matrix to be used is:

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

Since, this is a 2D object the translation in z axis is set to 0. Furthermore, the vertex shader performs matrix multiplications in column major basis but the normal translation matrix is row major matrix so firstly the above matrix is to be transposed as:

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

Now, using this matrix any 2D object can be easily translated by the given factor.

3.2. Source Code

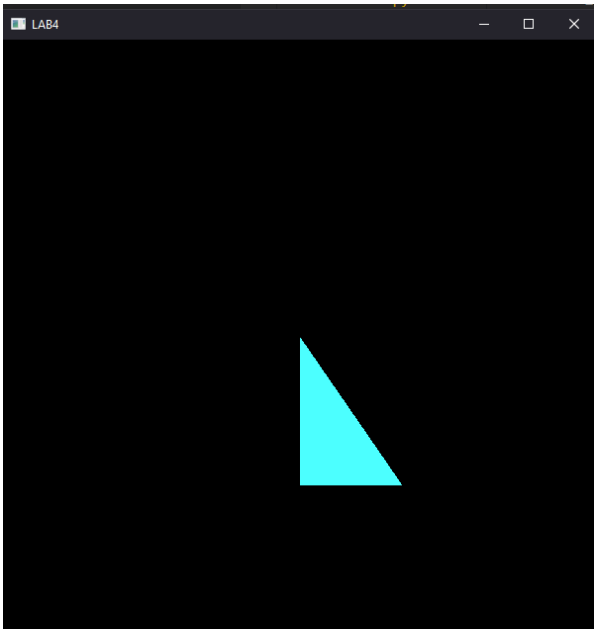
```
import glfw
import numpy as np
from OpenGL.GL import *
from OpenGL.GL.shaders import compileProgram, compileShader

from helpers import altList, toNVC


def window_resize(window, width, height):
    glViewport(0, 0, width, height)


RESOLUTION = 800

#translation matrix with tx=ty=0.5
translation = np.array(
    [
        1.0,0.0,0.0,0.0,
        0.0,1.0,0.0,0.0,
        0.0,0.0,1.0,0.0,
        0.5,0.5,0.0,1.0,
    ],
    dtype=np.float32,
)

def main(transformation):
```

```
  vertex_src = """
#version 330
layout(location=0) in vec3 aPos;
uniform mat4 transform;
void main(){
    gl_Position = transform * vec4(aPos,1.0);

    }
"""

  fragment_src = """
#version 330
out vec4 FragColor;
void main(){
    FragColor = vec4(1.0f,0.0f,0.3f,0.0f);

}
"""

  if not glfw.init():
      raise Exception("glfw can not be initialized!")

  window = glfw.create_window(RESOLUTION, RESOLUTION, "LAB4", None, None)

  if not window:
      glfw.terminate()
      raise Exception("glfw window can not be created!")

  glfw.set_window_pos(window, 100, 100)
  glfw.set_window_size_callback(window, window_resize)
  glfw.make_context_current(window)

  vertices = [
      -0.27390625,
      0.19890625000000003,
      0,
      -0.27390625,
      -0.4223437499999999,
      0,
      0.34421874999999985,
      -0.4223437499999999,
      0,
  ]
  vertices = np.array(vertices, dtype=np.float32)
  indices = [1, 2, 3]
  indices = np.array(indices, dtype=np.uint32)
```

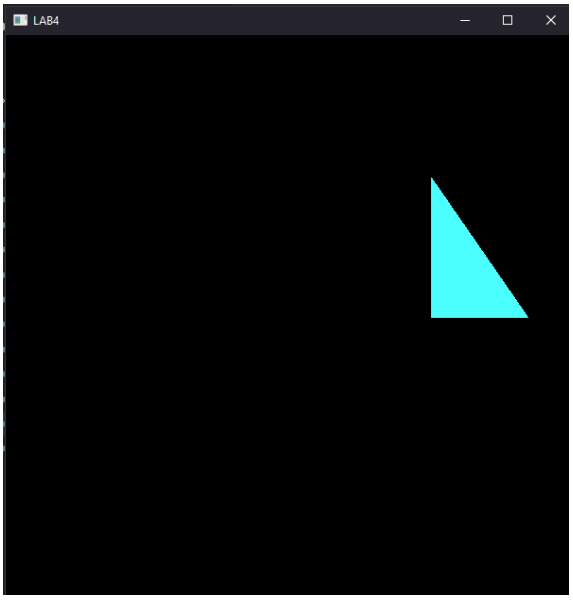
```
    shader = compileProgram(
        compileShader(vertex_src, GL_VERTEX_SHADER),
        compileShader(fragment_src, GL_FRAGMENT_SHADER),
    )

    vertex_buffer_object = glGenBuffers(1)
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_object)
    glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STREAM_DRAW)

    element_buffer_object = glGenBuffers(1)
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer_object)
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.nbytes, indices,
                 GL_STREAM_DRAW)

    glEnableVertexAttribArray(0)
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, ctypes.c_void_p(0))

    glUseProgram(shader)
    transformation_location = glGetUniformLocation(shader, "transformation")

    while not glfw.window_should_close(window):

        glfw.poll_events()

        glUniformMatrix4fv(transformation_location, 1, GL_FALSE,
                           transformation)
        glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)
        glfw.swap_buffers(window)

    glfw.terminate()

main(translation)
```

3.3 Output

3.3.1. Original Triangle



3.3.2. Triangle after translation of 0.5



Chapter 4: 2D Rotation

4.1. Rotation Matrix

The rotation matrix for 2D is:

cos(θ)	-sin(θ)	0	0
sin(θ)	cos(θ)	0	0
0	0	1	0
0	0	0	1

However, as this matrix is row major and vertex shader calculates matrix multiplication in column major basis, we have to transform the above matrix as:

cos(θ)	sin(θ)	0	0
-sin(θ)	cos(θ)	0	0
0	0	1	0
0	0	0	1

Now using this matrix, we can rotate any 2D object.

4.2. Source Code

```
import glfw
import numpy as np
from OpenGL.GL import *
from OpenGL.GL.shaders import compileProgram, compileShader

from helpers import altList, toNVC


def window_resize(window, width, height):
    glViewport(0, 0, width, height)
```

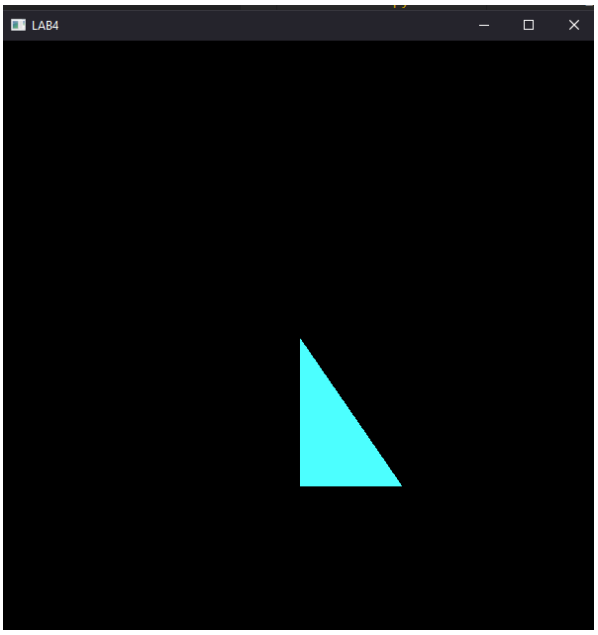
```
RESOLUTION = 800
rotation = np.array(
    [
        np.cos((np.pi / 180) * 45),  # D1
        np.sin((np.pi / 180) * 45),
        0.0,
        0.0,
        np.sin(-(np.pi / 180) * 45),
        np.cos((np.pi / 180) * 45),  # D2
        0.0,
        0.0,
        0.0,
        0.0,
        1.0,
        0.0,
        0.0,
        0.0,
        0.0,
        1.0,
    ],
    dtype=np.float32,
)

def main(transformation):

    vertex_src = """
  #version 330
  layout(location=0) in vec3 aPos;
  uniform mat4 transform;
  void main(){
      gl_Position = transform * vec4(aPos,1.0);

      }
  """

    fragment_src = """
  #version 330
  out vec4 FragColor;
  void main(){
      FragColor = vec4(1.0f,0.0f,0.3f,0.0f);

  }
  """

    if not glfw.init():
        raise Exception("glfw can not be initialized!")
```

```
window = glfw.create_window(RESOLUTION, RESOLUTION, "LAB4", None, None)

if not window:
    glfw.terminate()
    raise Exception("glfw window can not be created!")

glfw.set_window_pos(window, 100, 100)
glfw.set_window_size_callback(window, window_resize)
glfw.make_context_current(window)

vertices = [
    -0.27390625,
    0.19890625000000003,
    0,
    -0.27390625,
    -0.4223437499999999,
    0,
    0.34421874999999985,
    -0.4223437499999999,
    0,
]
vertices = np.array(vertices, dtype=np.float32)
indices = [1, 2, 3]
indices = np.array(indices, dtype=np.uint32)

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

vertex_buffer_object = glGenBuffers(1)
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_object)
glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STREAM_DRAW)

element_buffer_object = glGenBuffers(1)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer_object)
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.nbytes, indices,
             GL_STREAM_DRAW)

glEnableVertexAttribArray(0)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, ctypes.c_void_p(0))

glUseProgram(shader)
transformation_location = glGetUniformLocation(shader, "transformation")

while not glfw.window_should_close(window):
```

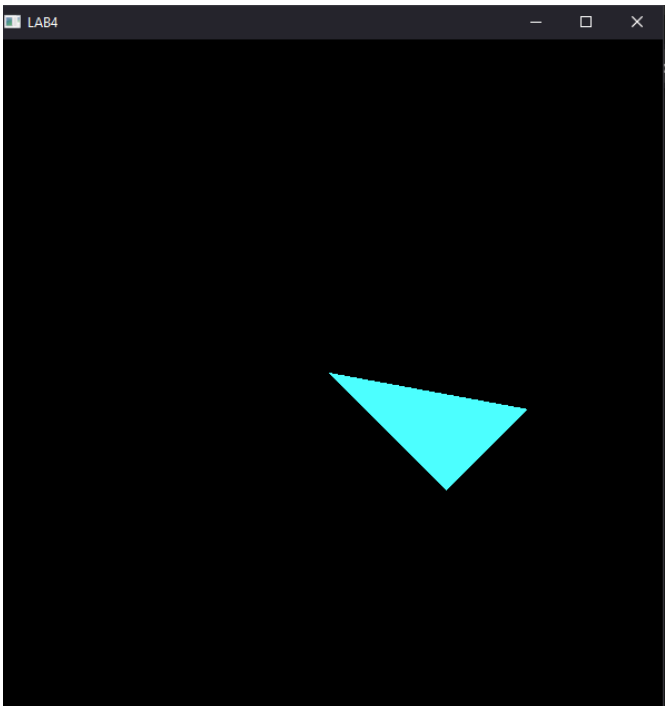
```
        glfw.poll_events()

        glUniformMatrix4fv(transformation_location, 1, GL_FALSE,
                           transformation)
        glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)
        glfw.swap_buffers(window)

    glfw.terminate()

main(rotation)
```

4.3. Output

4.3.1. Original Triangle



4.3.2. Triangle after Rotation by 45 degree



Chapter 5: 2D Scaling

5.1. Scaling Matrix

The scaling matrix is given as:

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

As it is 2D scaling, the scaling factor across z-aix is set to 0. As discussed earlier, the above row major matrix should be converted in column major matrix by transposing, but as the original

matrix and the transposed matrix are identical, above rotation matrix can be used to get required scale.

5.2. Source Code

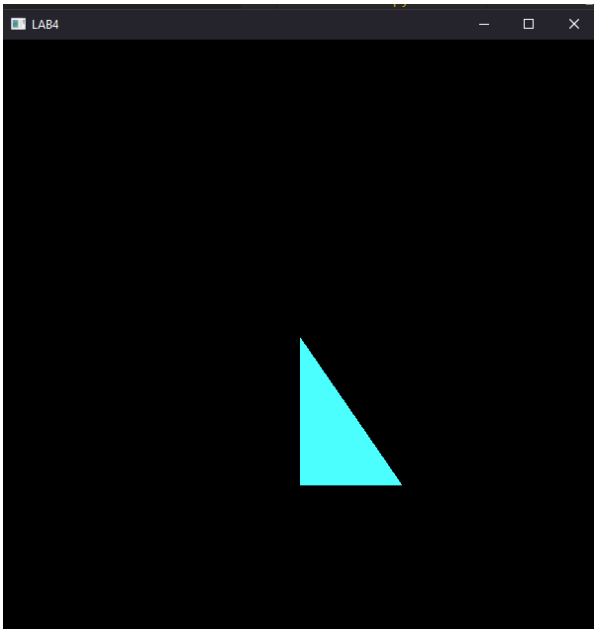
```
import glfw
import numpy as np
from OpenGL.GL import *
from OpenGL.GL.shaders import compileProgram, compileShader

from helpers import altList, toNVC


def window_resize(window, width, height):
    glViewport(0, 0, width, height)


RESOLUTION = 800

scaling = np.array(
    [
        2.0,  # D1
        0.0,
        0.0,
        0.0,
        0.0,
        2.0,  # D2
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,  # D3
        0.0,
        0.0,
        0.0,
        0.0,
        1.0,
    ],
    dtype=np.float32,
)

def main(transformation):

    vertex_src = """
  #version 330
```

```
layout(location=0) in vec3 aPos;
uniform mat4 transform;
void main(){
    gl_Position = transform * vec4(aPos,1.0);

    }
"""

  fragment_src = """
#version 330
out vec4 FragColor;
void main(){
    FragColor = vec4(1.0f,0.0f,0.3f,0.0f);

}
"""

  if not glfw.init():
      raise Exception("glfw can not be initialized!")

  window = glfw.create_window(RESOLUTION, RESOLUTION, "LAB4", None, None)

  if not window:
      glfw.terminate()
      raise Exception("glfw window can not be created!")

  glfw.set_window_pos(window, 100, 100)
  glfw.set_window_size_callback(window, window_resize)
  glfw.make_context_current(window)

  vertices = [
      -0.27390625,
      0.19890625000000003,
      0,
      -0.27390625,
      -0.4223437499999999,
      0,
      0.34421874999999985,
      -0.4223437499999999,
      0,
  ]
  vertices = np.array(vertices, dtype=np.float32)
  indices = [1, 2, 3]
  indices = np.array(indices, dtype=np.uint32)

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

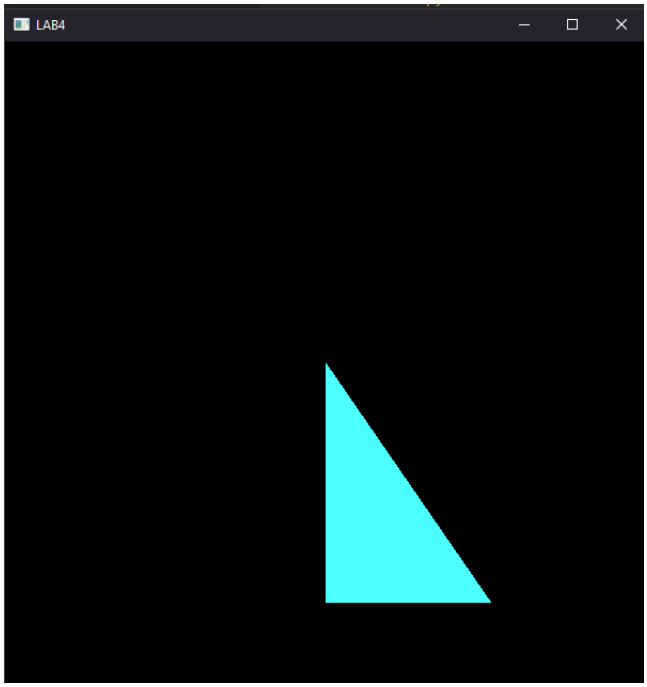
```
    )

    vertex_buffer_object = glGenBuffers(1)
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_object)
    glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STREAM_DRAW)

    element_buffer_object = glGenBuffers(1)
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer_object)
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.nbytes, indices,
                 GL_STREAM_DRAW)

    glEnableVertexAttribArray(0)
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, ctypes.c_void_p(0))

    glUseProgram(shader)
    transformation_location = glGetUniformLocation(shader, "transformation")

    while not glfw.window_should_close(window):

        glfw.poll_events()

        glUniformMatrix4fv(transformation_location, 1, GL_FALSE,
                           transformation)
        glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)
        glfw.swap_buffers(window)

    glfw.terminate()

main(scaling)
```

5.3. Output

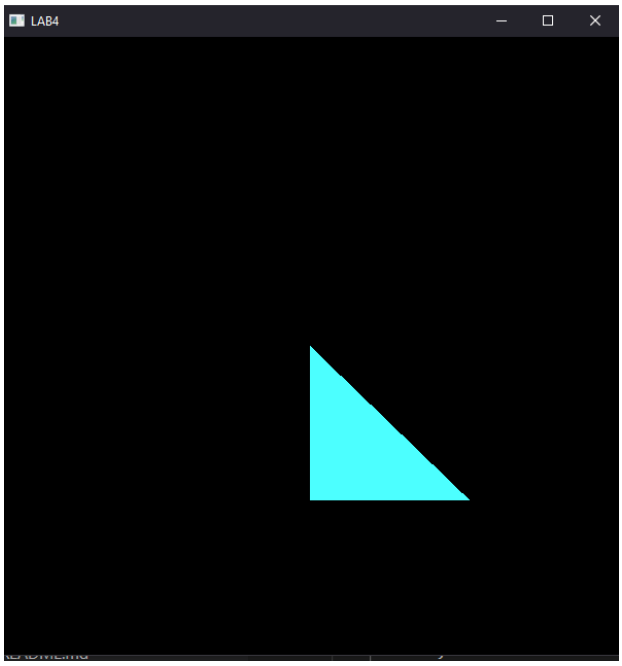
5.3.1. Original Triangle



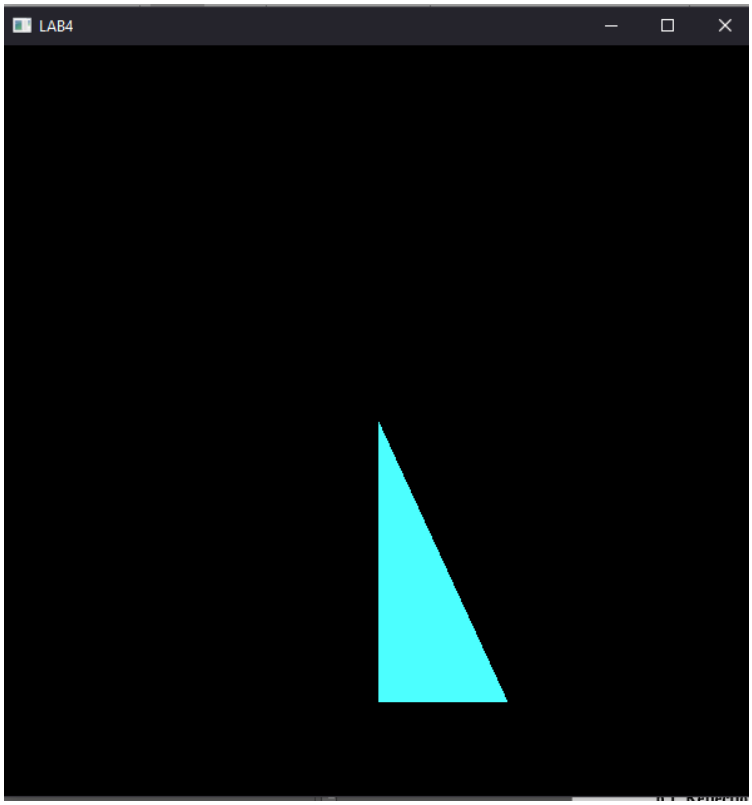
5.3.2. Scaled Triangle with scaling factor 1.5



5.3.3. Scaled Triangle with Scaling factor 1.5 along x-axis



5.3.4. Scaled Triangle with Scaling factor 1.5 along y-axis



Chapter 6: Reflection on 2D Objects

6.1. Reflection Matrix

The reflection matris on x-axis is given by:

1	0	0	0
0	-1	0	0
0	0	0	0
0	0	0	1

And, the reflection matrix on y-axis is given by:

-1	0	0	0
0	1	0	0
0	0	0	0
0	0	0	1

The reflection in z axis is set to 0 as it is not required for our two-dimensional triangle. As discussed earlier, the above row major matrix should be converted in column major matrix by transposing, but as the original matrix and the transposed matrix are identical, above rotation matrix can be used to get required scale.

6.2. Source Code

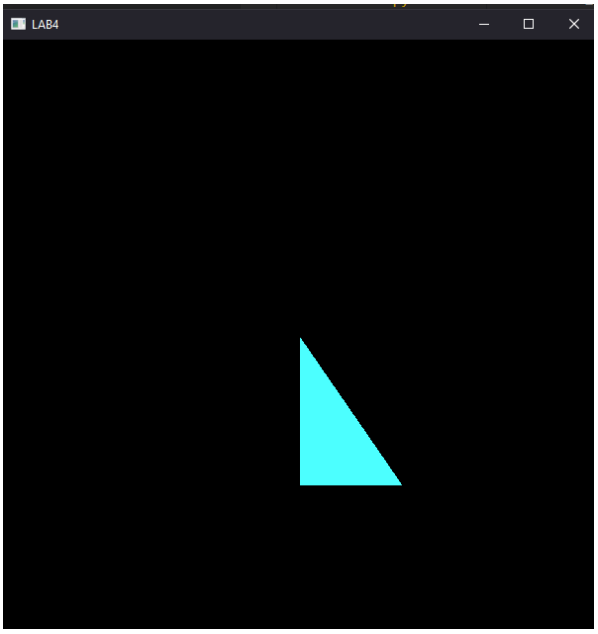
```
import glfw
import numpy as np
from OpenGL.GL import *
from OpenGL.GL.shaders import compileProgram, compileShader

from helpers import altList, toNVC


def window_resize(window, width, height):
    glViewport(0, 0, width, height)
```

```
RESOLUTION = 800

reflection = np.array(
    [
        1.0, 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,
)


def main(transformation):

    vertex_src = """
  #version 330
  layout(location=0) in vec3 aPos;
  uniform mat4 transform;
  void main(){
      gl_Position = transform * vec4(aPos,1.0);

      }
  """

    fragment_src = """
  #version 330
  out vec4 FragColor;
  void main(){
      FragColor = vec4(1.0f,0.0f,0.3f,0.0f);

  }
  """

    if not glfw.init():
        raise Exception("glfw can not be initialized!")

    window = glfw.create_window(RESOLUTION, RESOLUTION, "LAB4", None, None)

    if not window:
        glfw.terminate()
        raise Exception("glfw window can not be created!")

    glfw.set_window_pos(window, 100, 100)
    glfw.set_window_size_callback(window, window_resize)
    glfw.make_context_current(window)

    vertices = [
```

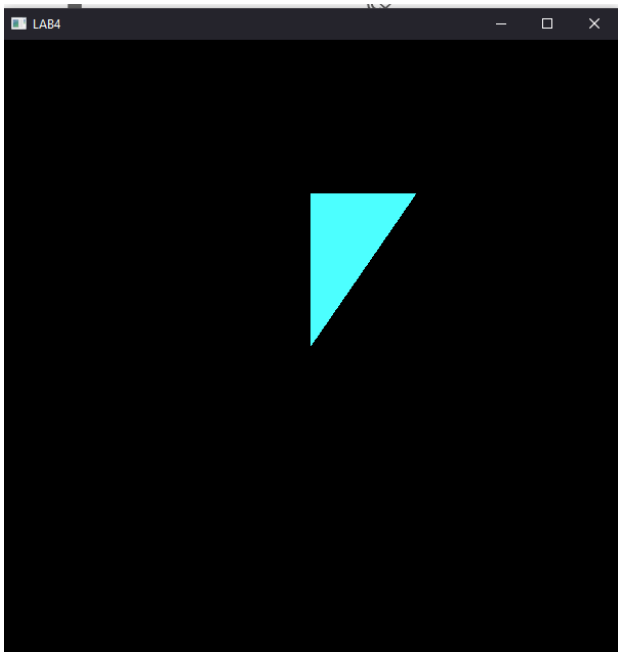
```
        -0.27390625,
        0.19890625000000003,
        0,
        -0.27390625,
        -0.4223437499999999,
        0,
        0.34421874999999985,
        -0.4223437499999999,
        0,
    ]
    vertices = np.array(vertices, dtype=np.float32)
    indices = [1, 2, 3]
    indices = np.array(indices, dtype=np.uint32)

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

    vertex_buffer_object = glGenBuffers(1)
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_object)
    glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STREAM_DRAW)

    element_buffer_object = glGenBuffers(1)
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer_object)
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.nbytes, indices,
                 GL_STREAM_DRAW)

    glEnableVertexAttribArray(0)
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, ctypes.c_void_p(0))

    glUseProgram(shader)
    transformation_location = glGetUniformLocation(shader, "transformation")

    while not glfw.window_should_close(window):

        glfw.poll_events()

        glUniformMatrix4fv(transformation_location, 1, GL_FALSE,
                           transformation)
        glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)
        glfw.swap_buffers(window)

    glfw.terminate()

main(reflection)
```

6.3. Output

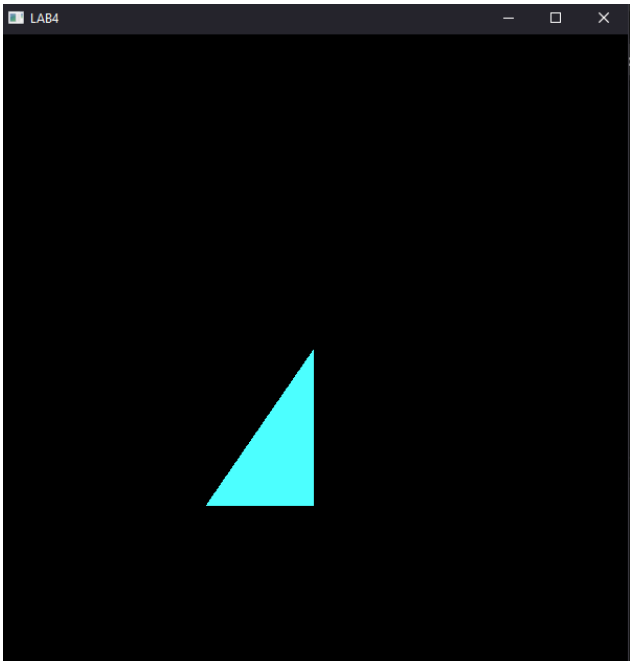
6.3.1. Original Triangle



6.3.2. Reflection about x-axis



6.3.2. Reflection about y-axis



Chapter 7: 2D Shearing

7.1. Shearing Matrix

The shearing matrix on x-axis is given as:

1	0	0	0
shx	1	0	0
0	0	0	0
0	0	0	1

And the shearing matrix in y-axis is given as:

1	shy	0	0
0	1	0	0
0	0	0	0
0	0	0	1

However, there is one more problem as the vertex shader calculates matrix multiplications in a column major basis but the normal matrix as shown above is a row major matrix. Hence the matrix sent to the GPU is actually transposed to get the following:

1	shx	0	0
0	1	0	0
0	0	0	0
0	0	0	1

And in y-axis is given as:

1	0	0	0
shy	1	0	0
0	0	0	0
0	0	0	1

7.2. Source Code

```
import glfw
import numpy as np
from OpenGL.GL import *
from OpenGL.GL.shaders import compileProgram, compileShader

from helpers import altList, toNVC


def window_resize(window, width, height):
    glViewport(0, 0, width, height)


RESOLUTION = 800

shearing = np.array(
```

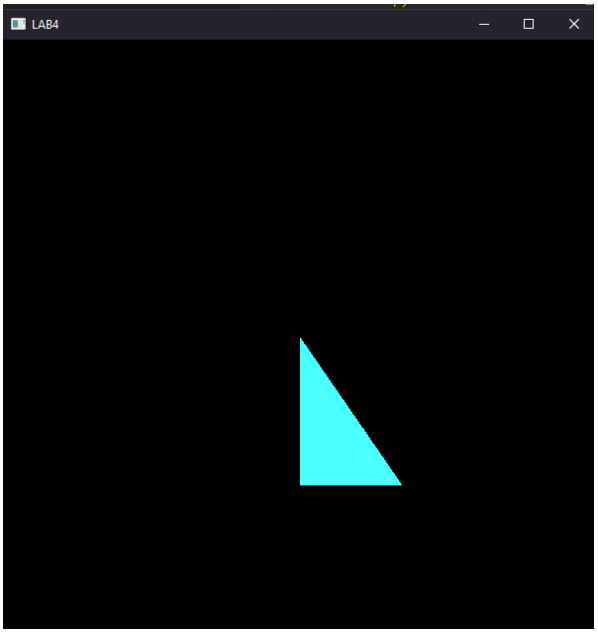
```
    [
        1.0,  # D1
        0.8,  # ShearX
        0.0,
        0.0,
        0.0,  # ShearY
        1.0,  # D2
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        1.0,
    ],
    np.float32,
)

def main(transformation):

    vertex_src = """
  #version 330
  layout(location=0) in vec3 aPos;
  uniform mat4 transform;
  void main(){
      gl_Position = transform * vec4(aPos,1.0);

      }
  """

    fragment_src = """
  #version 330
  out vec4 FragColor;
  void main(){
      FragColor = vec4(1.0f,0.0f,0.3f,0.0f);

  }
  """

    if not glfw.init():
        raise Exception("glfw can not be initialized!")

    window = glfw.create_window(RESOLUTION, RESOLUTION, "LAB4", None, None)

    if not window:
```

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

    glfw.set_window_pos(window, 100, 100)
    glfw.set_window_size_callback(window, window_resize)
    glfw.make_context_current(window)

    vertices = [
        -0.27390625,
        0.19890625000000003,
        0,
        -0.27390625,
        -0.4223437499999999,
        0,
        0.34421874999999985,
        -0.4223437499999999,
        0,
    ]
    vertices = np.array(vertices, dtype=np.float32)
    indices = [1, 2, 3]
    indices = np.array(indices, dtype=np.uint32)

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

    vertex_buffer_object = glGenBuffers(1)
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_object)
    glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STREAM_DRAW)

    element_buffer_object = glGenBuffers(1)
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer_object)
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.nbytes, indices,
                 GL_STREAM_DRAW)

    glEnableVertexAttribArray(0)
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, ctypes.c_void_p(0))

    glUseProgram(shader)
    transformation_location = glGetUniformLocation(shader, "transformation")

    while not glfw.window_should_close(window):

        glfw.poll_events()

        glUniformMatrix4fv(transformation_location, 1, GL_FALSE,
                           transformation)
```

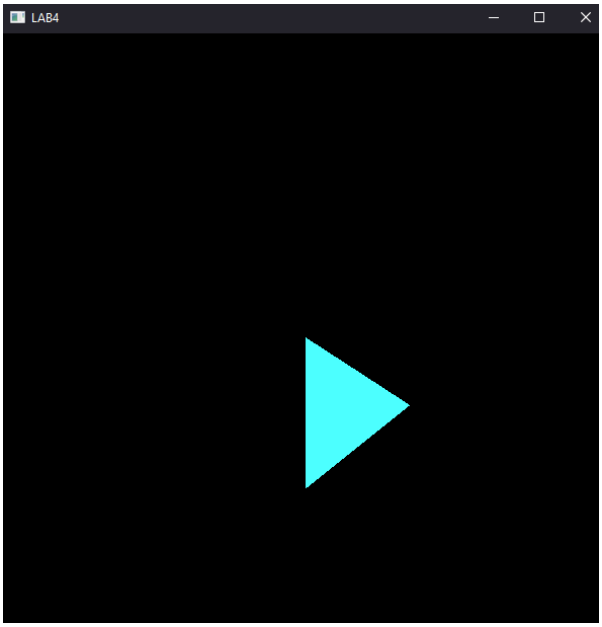
```
        glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)
        glfw.swap_buffers(window)

    glfw.terminate()

main(shearing)
```

7.3. Output

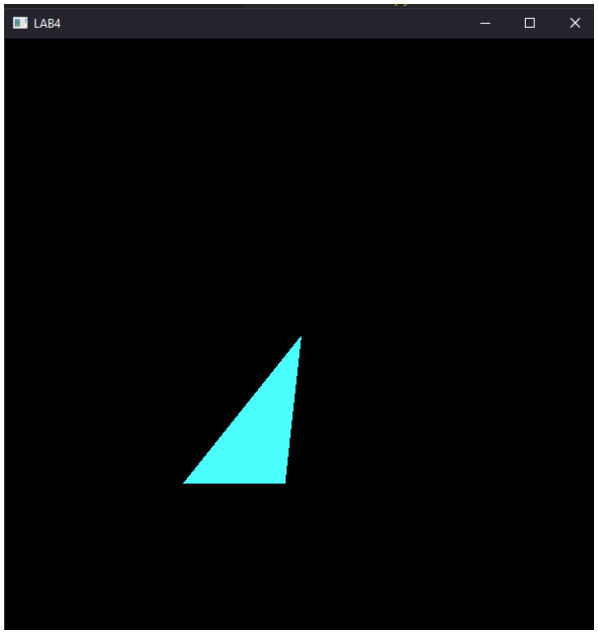
7.3.1. Original Triangle



7.3.2. Triangle sheared on x-axis



7.3.3. Triangle sheared along y-axis



Chapter 8: Conclusion

In this Lab work, various transformation matrix was used to achieve various transformations on 2D objects. The transformations that we implemented are: translations, reflection, rotation, scaling and shearing. The transformation matrix that we normally use cannot be directly sent to the program as GPU uses column major format of matrices, so the transposed matrix has been sent to the program.

The desired transformation operation is provided to the program as parameter like:

```
main(shearing)
```

And the program is provided with 4 x 4 transposed matrix using is stored using NumPy array.

```
shearing = np.array(
    [
        1.0,  # D1
        0.8,  # ShearX
        0.0,
        0.0,
        0.0,  # ShearY
        1.0,  # D2
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        1.0,
    ],
    np.float32,
)
```

Also, the vertices of our original triangle was also defined as:

```
vertices = [
    -0.27390625,
    0.19890625000000003,
    0,
    -0.27390625,
    -0.4223437499999999,
    0,
    0.34421874999999985,
    -0.4223437499999999,
    0,
]
```

Furthermore, a uniform matrix transformation shader had been made, and the location of which was stored under transformation_location as:

```
transformation_location = glGetUniformLocation(shader, "transformation")
```

Finally, using the glUniformMatrix4fv matrix was transformed and GL_TRIANGLES was made by using glDrawElements.

```
glUniformMatrix4fv(transformation_location, 1, GL_FALSE,
                           transformation)
```