Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Environment Setup and Nepal Flag

Lab Report One

[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: Programming Language and Library

1.1 Introduction

For the completion of Computer Graphics Lab work and Project, I have decided on using the following Programming Language, Graphics Library and Tools:

Programming Language: Python 3.10

Graphics Library: PyOpenGL 3.1.6

Window Renderer: GLUT

Helper Library: ctypes, numpy

Chapter 2: Graphics Environment Setup

2.1 Code Snippet

```
import os
import OpenGL.GL as gl
import OpenGL.GLUT as glut

def display():
    # Clearing the window buffer for display
    gl.glClear(gl.GL_COLOR_BUFFER_BIT)
    gl.glClearColor(0.0, 0.0, 0.0, 0.0)
    glut.glutSwapBuffers()

def reshape(width, height):
    gl.glViewport(0, 0, width, height)
    print(f"Resolution of the Window : {width} x {height}")

# function to exit the Graphics Window on escape
def keyboard( key, x, y ):
    if key == b'\x1b':
        os._exit(1)

#Initializing GLUT for Window Rendering
glut.glutInit()
glut.glutInitDisplayMode(glut.GLUT_DOUBLE | glut.GLUT_RGBA)
glut.glutCreateWindow("Graphics Window")

#Window Reshaping
glut.glutReshapeWindow(1920, 1080)

#Resolution of the Display System
print(f"Resolution of the Display System :
{glut.glutGet(glut.GLUT_SCREEN_WIDTH)} x
{glut.glutGet(glut.GLUT_SCREEN_HEIGHT)}")

glut.glutReshapeFunc(reshape)
glut.glutDisplayFunc(display)
glut.glutPostRedisplay()
glut.glutKeyboardFunc(keyboard)
glut.glutMainLoop()
```

2.2 Description

The environmental setup to start utilizing the function of OpenGL starts with window rendering. My setup uses GLUT for window creation where, the window context is initialized with glut.glutInit() command and the glut.glutMainLoop() enters the program into the GLUT event processing loop. Within the event processing loop, upon encountering external events that occurs within the perimeter of the rendered window, we can catch the event and handle them using their respective function blocks. One such event is the glut.glutDisplayFunc(), which is

initially triggered when the window is created. This function calls upon the display function to clear the Color Buffer and clear the screen to display a fully black window. Another event is the glut.glutReshapeWindow() which is also executed once during the initial context creation where it reshapes the rendered window to the specified width and height. Moreover, externally triggered asynchronous events such as a key press can also be caught through glu.glutKeyboardFunc() which invokes our keyboard function with the key pressed, and the position of the mouse relative to the window when the key was pressed. This keyboard pressing action is utilized by our program to exit the window context. Similarly, OpenGL facilitates us with other asynchronous event functions, but for the context of our environmental setup we will not be pursuing it.

2.3 Outputs

The rendered window using the initial environment setup and the console output stating the resolution of the Display System and the Created Window are shown below.

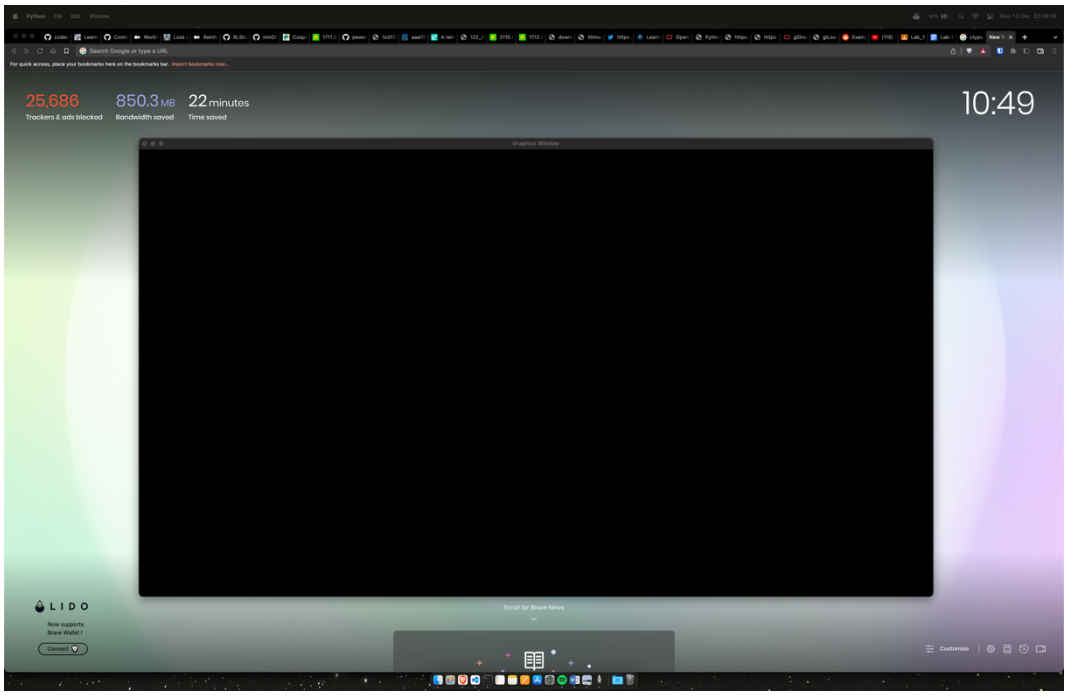

fig 2.2.1 : Rendered Graphics Window using GLUT

```
~/Documents/graphics/computergraphics main ?12 ❯ python settingup.py
Resolution of the Display System : 2560 x 1655
Resolution of the Window : 1920 x 1080
```

fig 2.2.2 : Console Output

Chapter 3: Flag Creation

3.1 Code Snippet

```
import os
import ctypes
import numpy as np

import OpenGL.GL as gl
import OpenGL.GLUT as glut

vertexShaderCode = """
    attribute vec3 position;
    attribute vec4 color;
    varying vec4 vColor;
    void main(){
        gl_Position = vec4(position, 1.0);
        vColor = color;
    }
    """

fragmentShaderCode = """
    varying vec4 vColor;
    void main(){
        gl_FragColor = vColor;
    }
    """

# 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

# -- Building Data --
data = np.zeros(12, [("position", np.float32, 3),
                     ("color",    np.float32, 4)])
data['position'] = [[-0.30, +0.45, +1.00],
                    [-0.30, +0.01, +1.00],
                    [+0.45, +0.01, +1.00],
                    [-0.30, +0.25, +1.00],
                    [-0.30, -0.45, +1.00],
                    [+0.41, -0.45, +1.00],
                    [-0.27, +0.40, +1.00],
                    [-0.27, +0.04, +1.00],
                    [+0.35, +0.04, +1.00],
                    [-0.27, +0.18, +1.00],
                    [-0.27, -0.42, +1.00],
                    [+0.35, -0.42, +1.00]]

data['color'] =((0, 0.00, 0.60, 1),
                (0, 0.00, 0.60, 1),
                (0, 0.00, 0.60, 1),
                (0, 0.00, 0.60, 1),
                (0, 0.00, 0.60, 1),
                (0, 0.00, 0.60, 1),
                (0.9, 0.0, 0.2, 1),
                (0.9, 0.0, 0.2, 1),
                (0.9, 0.0, 0.2, 1),
                (0.9, 0.0, 0.2, 1),
                (0.9, 0.0, 0.2, 1),
                (0.9, 0.0, 0.2, 1))

indicesData = np.array([0,1,2,3,4,5,6,7,8,9,10,11], dtype=np.int32)

# initialization function
def initialize():
    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)
    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)

    # 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)
    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(1080,1080)
glut.glutReshapeFunc(reshape)

initialize()

glut.glutDisplayFunc(display)
glut.glutPostRedisplay()
glut.glutKeyboardFunc(keyboard)

# enter the mainloop
glut.glutMainLoop()
```

3.2 Output

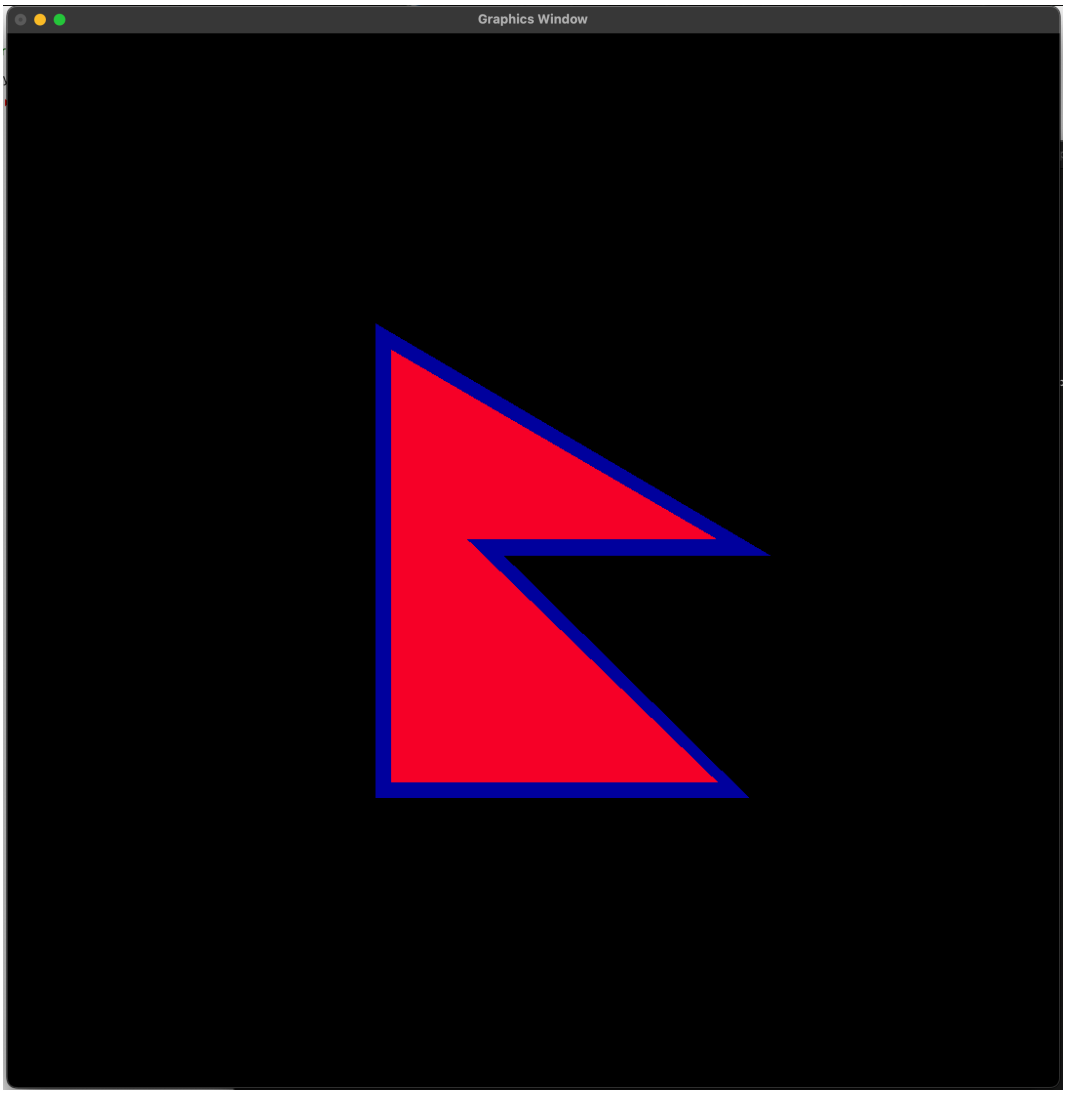

fig 3.2.1 : Colored Nepal's Flag

3.3 Description

The flag drawing program uses a modern approach currently in rise within the OpenGL community that utilizes a dynamic pipeline through the introduction of shaders, namely Vertex and Fragment Shaders. This allows the users to have direct access to the GPU. The use of these shaders grants the user of OpenGL the full control over the complete pipeline of the rendering process instead of using previously popular however, limited and fixed pipeline based commands.

The Dynamic Pipeline utilized for the creation of the Nepal Flag is as follows:

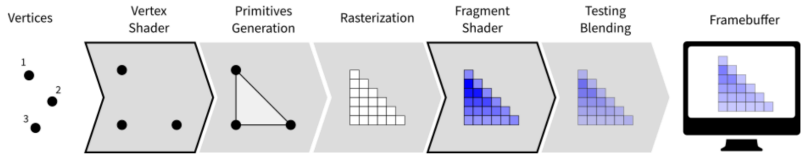

fig 3.3.1 : Dynamic Shader Pipeline

The vertex shader acts on vertices and outputs the vertex position described using the gl_Position on to the viewport. And, the fragment shader acts at the fragment level and controls the output color using the gl_FragColor of the fragment.

In the **Code Snippet** above, we describe the vertices of the object we aim to draw. Then, through the use of attribute type of the vertex shader we pass these vertices onto gl_Position, which maps these vertices onto the viewport. These vertices plotted onto the viewport can be connected using one of the various primitives provided by the OpenGL. For our purpose of rendering a triangular flag, we utilize the gl.GL_TRIANGLES primitive and pass the indices array that specifies the coordinates that connect to each other to form the triangles. The created object is then rasterized by the library. After which, the fragment shader is given a color for each of the vertices. These colors are loaded onto gl_FragColor using the varying data type, which helps graphics library to use interpolation to determine the color contained within these vertices. And, finally the frame buffer that is created is uploaded to the rendered window and the result is displayed onto the screen.

Within the **Code**, the createShader() function requests, sets and compiles the vertex and fragment shader slots from GPU. The createProgram() function links all the compiled shader onto a program object which is continuously utilized throughout the program to set position / color values and render the object. The initialize() function request buffers from the GPU and loads the data into the buffer, which is later used to initialize the vertex and color values within the GPU. And, finally the render function clears the output screen and draws the objects using the gl_TRIANGLES primitive whilst using the indicesData stored in the element array buffer.

If we look at the following data, we can clearly visualize the functioning of the program:

```
data['position'] = [[-0.30, +0.45, +1.00],
                    [-0.30, +0.01, +1.00],
                    [+0.45, +0.01, +1.00],
                    [-0.30, +0.25, +1.00],
                    [-0.30, -0.45, +1.00],
                    [+0.41, -0.45, +1.00],
                    [-0.27, +0.40, +1.00],
                    [-0.27, +0.04, +1.00],
                    [+0.35, +0.04, +1.00],
                    [-0.27, +0.18, +1.00],
                    [-0.27, -0.42, +1.00],
                    [+0.35, -0.42, +1.00]]
and,

indicesData = np.array([0,1,2,3,4,5,6,7,8,9,10,11], dtype=np.int32)
```

The data stores 12 Vertices for four triangles that we aim to draw to complete the flag of Nepal. As stated above, these vertices are loaded onto the gl_Position of vertex shader using the position attribute. But, since we have decided on using the gl_TRIANGLES primitive, we use the indicesData to specify the triplet of points that need to be connected to create a triangle; that is, the values of indicesData are interpreted in a group of three.

Hence, vertex 0, 1 and 2 are used to create the first triangle. Vertex 3, 4 and 5 are used to create the second triangle and so on. These indicesData are stored on an Element Buffer which is used by the gl.glDrawElements() function during the render.

Moreover, if we glance at the color data, we can see that we have specified the same color values for vertices of a single triangle. This means that even when interpolated amongst the vertices the color values remain the same, creating triangle that is of a single color.

```
data['color'] =
((0, 0.00, 0.60, 1), (0, 0.00, 0.60, 1), (0, 0.00, 0.60, 1),
 (0, 0.00, 0.60, 1), (0, 0.00, 0.60, 1), (0, 0.00, 0.60, 1),
 (0.9, 0.0, 0.2, 1), (0.9, 0.0, 0.2, 1), (0.9, 0.0, 0.2, 1),
 (0.9, 0.0, 0.2, 1), (0.9, 0.0, 0.2, 1), (0.9, 0.0, 0.2, 1))
```

Chapter 4: Conclusion

Through this lab work, I was able to setup a working environment for Window Rendering using PyOpenGL and understand the various data types, output primitives of the OpenGL Shading Language (GLSL), whilst gaining an insight on the workings of vertex shader and fragment shader. However, there still remains a huge part of OpenGL that needs exploration. Specifically, the understanding of complex mechanisms needed to implement the Sun and the Moon of the Flag using the modern approach. This will be explored and discussed while working on future lab works.