KATHMANDU UNIVERSITY

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report

On

“Lab-1”

[Code No: COMP 342]

Submitted By

Rajaram Karki

Roll No: 23

Submitted To:

Dhiraj Shrestha

Department of Computer Science and Engineering

Submission Date: 18th March

Lab 1: Get Familiar with the coordinate system to draw a flag of Nepal using OpenTK and color the flag accordingly. Also display the resolution of the device.

Implementation

Programming Language Used: C#

Library Used: OpenTK

Graphics API used: OpenGL

```
using OpenTK.Graphics.OpenGL;
using OpenTK;
using OpenTK.Mathematics;
using OpenTK.Windowing.Common;
using OpenTK.Windowing.Desktop;
```

Source Code

For coding and implementation part, firstly the basic window output was written. A class called Game was created which inherits from a parent class called GameWindow which provides us with all necessary declarations to create a window. A constructor is created that passes the following arguments. The constructor then calls the CenterWindow method of the Game class and also passes the height and width of the window as Vector2.

```
1 reference
public Game()
    : base(GameWindowSettings.Default, NativeWindowSettings.Default)
{
    this.CenterWindow(new Vector2i(1270, 1200));
}
```

We also declare the required variables here

```
int circleSides = 250; // number of sides of the circle polygon

private int vertexbufferObject;
private int shaderProgramObject;
private int vertexArrayObject;
```

In order to set up OpenGL and draw anything we want; we need to override the following pre-defined functions of OpenGL so that we can manipulate the output.

OnUpdateFrame(FrameEventArgs args)
OnLoad()
OnUnload()
onResize(ResizeEventArgs e)
onRenderFrame(FrameEventArgs args)

```
protected override void OnUpdateFrame(FrameEventArgs args)
{
    base.OnUpdateFrame(args);
}
```

This code overrides the OnUpdateFrame method of the GameWindow class in OpenTK, which is called once per frame before rendering. This function is not used much in our code.

Next up is OnUnload method which is called when the window is closing where we clean up any resources allocated by the program like buffers and shaders.

```
0 references
protected override void OnUnload()
{
    GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
    GL.DeleteBuffer(this.vertexbufferObject);

    GL.UseProgram(0);
    GL.DeleteProgram(this.shaderProgramObject);

    base.OnUnload();
}
```

One of our major functions is OnLoad function that is associated with what we want the output to be when the window is loaded. The major part of the code is associated with this function. It includes the vertices for the triangles to be drawn, it handles the background color, binding and compiling the buffers and shaders. It also deletes all the shader objects when its used. It includes our vertexShaderCode and fragmentShaderCode.

First, a new buffer is generated. It is then bound to one of the array buffers whose data is copied to the buffer. Then finally, the buffer is unbound using GL.Bindbuffer()

Here it is also specified in the VertexAttribPointer that the vertex data consists of 3 floating-point balues (x, y, z) and the stride for ever iteration is 3 * the size of floating points as the given points are in float.

```
this.vertexbufferObject = GL.GenBuffer();
GL.BindBuffer(BufferTarget.ArrayBuffer, this.vertexbufferObject);
GL.BufferData(BufferTarget.ArrayBuffer, vertices.Length * sizeof(float), vertices, BufferUsageHint.StaticDraw);
GL.BindBuffer(BufferTarget.ArrayBuffer, 0);

this.vertexArrayObject = GL.GenVertexArray();
GL.BindVertexArray(this.vertexArrayObject);

GL.BindBuffer(BufferTarget.ArrayBuffer, this.vertexbufferObject);
GL.VertexAttribPointer(0, 3, VertexAttribPointerType.Float, false, 3 * sizeof(float), 0);
GL.EnableVertexAttribArray(0);

GL.BindVertexArray(0);
```

Vertex Shader Code and Fragment Shader Code

The vertex shader is responsible for transforming the vertices from 3D space to 2D screen coordinates and the fragment shader is responsible for calculating the final color of each pixel that will be drawn on the screen. It is in Vector4 form (red, green, blue, intensity).

```
string vertexShaderCode =
    @"
    #version 330 core

    layout (location=0) in vec3 aPosition;

    void main(){

       gl_Position = vec4(aPosition, 1.0f);

    }";

string pixelShaderCode =
    @"
    #version 330 core
    out vec4 fragColor;
    uniform vec4 vColor;

    void main(){
       fragColor = vColor;
    }
    ";
```

The shaders should be compiled before it can be used. We must specify the buffer about the vertex shader code. For debugging purposes, the errors in vertexShaderCode are checked (if any comes up)

```
int vertexShaderObject = GL.CreateShader(ShaderType.VertexShader);
GL.ShaderSource(vertexbufferObject, vertexShaderCode);
GL.CompileShader(vertexShaderObject);

string vertexShaderInfo = GL.GetShaderInfoLog(vertexShaderObject);
if(vertexShaderInfo != String.Empty)
{
    Console.WriteLine(vertexShaderInfo);
}
```

Similar is done in case of the fragment shader

```
int pixelShaderObject = GL.CreateShader(ShaderType.FragmentShader);
GL.ShaderSource(pixelShaderObject, pixelShaderCode);
GL.CompileShader(pixelShaderObject);

string pixelShaderInfo = GL.GetShaderInfoLog(pixelShaderObject);
if (pixelShaderInfo != String.Empty)
{
    Console.WriteLine(pixelShaderInfo);
}
```

We must also attach create an object to attach vertex shader and pixel (fragment) shader. This way we can link them to the program. Then finally the shaders are detached and deleted after its work is done.

```
this.shaderProgramObject = GL.CreateProgram();

GL.AttachShader(this.shaderProgramObject, vertexShaderObject);
GL.AttachShader(this.shaderProgramObject, pixelShaderObject);

GL.LinkProgram(this.shaderProgramObject);

GL.DetachShader(this.shaderProgramObject, vertexShaderObject);
GL.DetachShader(this.shaderProgramObject, pixelShaderObject);

GL.DeleteShader(vertexShaderObject);
GL.DeleteShader(pixelShaderObject);

base.OnLoad();
```

Here the base.OnLoad basically completes the loading of OpenGL context. The function OnLoad completes.

The basic vertices for the triangles are also added in this function.

```
float[] vertices = new float[]
{
    // Triangle 1 - bottom
    -0.5f, 0.35f, 0.0f, // top
    -0.5f, -0.5f, 0.0f, // bottom left
    0.5f, -0.5f, 0.0f, // bottom right

    // Triangle 2 - top
    -0.5f, 0.6f, 0.0f, // top
    -0.5f, 0.0f, 0.0f, // bottom right
    0.5f, 0.0f, 0.0f, // bottom left

    //Inner Triangle Bot
    -0.47f, -0.47f, 0.0f,   //bottom left
    0.42f, -0.47f, 0.0f,    //bottom right
    -0.47f, 0.28f, 0.0f,       //top

    //Inner triangle top
    -0.47f, 0.03f, 0.0f,   //bottom left
    -0.47f, 0.54f, 0.0f,     //top
    0.38f, 0.03f, 0.0f       //bottom right

};
```

The code when invoked draws four triangles. The idea is such that there will be two inner crimson triangles and two outer navy-blue triangles so that the blue border can be achieved.

This is done in the onRenderFrame function which is the main rendering function of the program. It is basically called continuously to render the scene. The color and depth buffer are cleared and selective colors are set using Uniform. That tells our pixelShaderCode what the color is supposed to be.

The drawing of the first four triangles is done by first changing the color to navy blue and drawing the two outer triangles. Then the color is changed to crimson and the outer two triangles are drawn to give the required border.

```
GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);

GL.UseProgram(this.shaderProgramObject);
int uniformLocations = GL.GetUniformLocation(this.shaderProgramObject, "vColor");

GL.BindVertexArray(this.vertexArrayObject);


///for outer two triangles that are blue
GL.Uniform4(uniformLocations, 0.078f, 0.114f, 0.420f, 1.0f);
GL.DrawArrays(PrimitiveType.Triangles, 0, 3); // left triangle
GL.DrawArrays(PrimitiveType.Triangles, 3, 3); // right triangle

///for inner two triangles that are crimson
GL.Uniform4(uniformLocations, 0.8627f, 0.0784f, 0.2353f, 1.0f);     //crimson color
GL.DrawArrays(PrimitiveType.Triangles, 6, 3);
GL.DrawArrays(PrimitiveType.Triangles, 9, 3);
```

The output for drawing the triangles is as follows.

Now for drawing the sun and the moon, a separate function for each is made and called. Also, the color is set to white.

For drawing a circle, there is no pre-defined function. So TraingleFan is used that creates a fan like structure using triangles. If our sides are big enough then a perfect circle can be achieved. The center is made such that it fits the flag.

Similar is done for the crescent where half of the circle is drawn and mirrored horizontally to create the crescent shape.

Following are the codes used

For drawing the circle

```
1 reference
void DrawCircle(float centerX, float centerY,float radius, int circleSides, ref float[] vertices)
{
    float angleIncrement = (float)(2.0f * Math.PI / circleSides);
    float currentAngle = 0.0f;

    int originalLength = vertices.Length;
    Array.Resize(ref vertices, originalLength + circleSides * 3);

    for (int i = 0; i < circleSides; i++)
    {
        float x = centerX + (float)(radius * Math.Cos(currentAngle));
        float y = centerY + (float)(radius * Math.Sin(currentAngle));
        vertices[originalLength + i * 3] = x;
        vertices[originalLength + i * 3 + 1] = y;
        vertices[originalLength + i * 3 + 2] = 0.0f;
        currentAngle += angleIncrement;
    }
}
```

For drawing the crescent

```
1 reference
void DrawCrescent(float centerX, float centerY, float radius, int circleSides, ref float[] vertices)
{
    float angleIncrement = (float)(Math.PI / circleSides);
    float currentAngle = 0.0f;

    int originalLength = vertices.Length;
    Array.Resize(ref vertices, originalLength + circleSides * 2*3);

    for (int i = 0; i < circleSides; i++)
    {
        float x = centerX + (float)(radius * Math.Cos(currentAngle));
        float y = centerY + (float)(radius * Math.Sin(currentAngle));
        vertices[originalLength + i * 3] = x;
        vertices[originalLength + i * 3 + 1] = y;
        vertices[originalLength + i * 3 + 2] = 0.0f;
        currentAngle += angleIncrement;
    }

    // Mirror the circle horizontally to create the crescent shape
    for (int i = 0; i < circleSides - 1; i++)
    {
        float x = centerX - (float)(radius * Math.Cos(currentAngle));
        float y = centerY + (float)(radius * Math.Sin(currentAngle));
        vertices[originalLength + circleSides * 3 + (circleSides - i - 1) * 3] = x;
        vertices[originalLength + circleSides * 3 + (circleSides - i - 1) * 3 + 1] = y;
        vertices[originalLength + circleSides * 3 + (circleSides - i - 1) * 3 + 2] = 0.0f;
        currentAngle -= angleIncrement;
    }
}
```

All the calculated points are being stored in the vertices array itself by resizing it everytime so that it becomes easy for us to work with only one buffer data. The functions are called and the vertices will be stored in the array.

```
DrawCircle(-0.2f, -0.25f, 0.07f, 250, ref vertices);
DrawCrescent(-0.22f, 0.23f, 0.07f, 250, ref vertices);
```

Then the color is set to white and the given vertices are drawn to get the final output.

```
///for the sun and moon
GL.Uniform4(uniformLocations, 1.0f, 1.0f, 1.0f, 1.0f); // white
GL.DrawArrays(PrimitiveType.TriangleFan, 12, this.circleSides);
GL.DrawArrays(PrimitiveType.TriangleFan, 12 + (int)(circleSides/2.0f), circleSides );
```

At the end we must swap the front and back buffers of the window so that it display what is being rendered. And finally the base call implementation of the OnRenderFrame is called

```
this.Context.SwapBuffers();
base.OnRenderFrame(args);
```

To get the resolution of the device

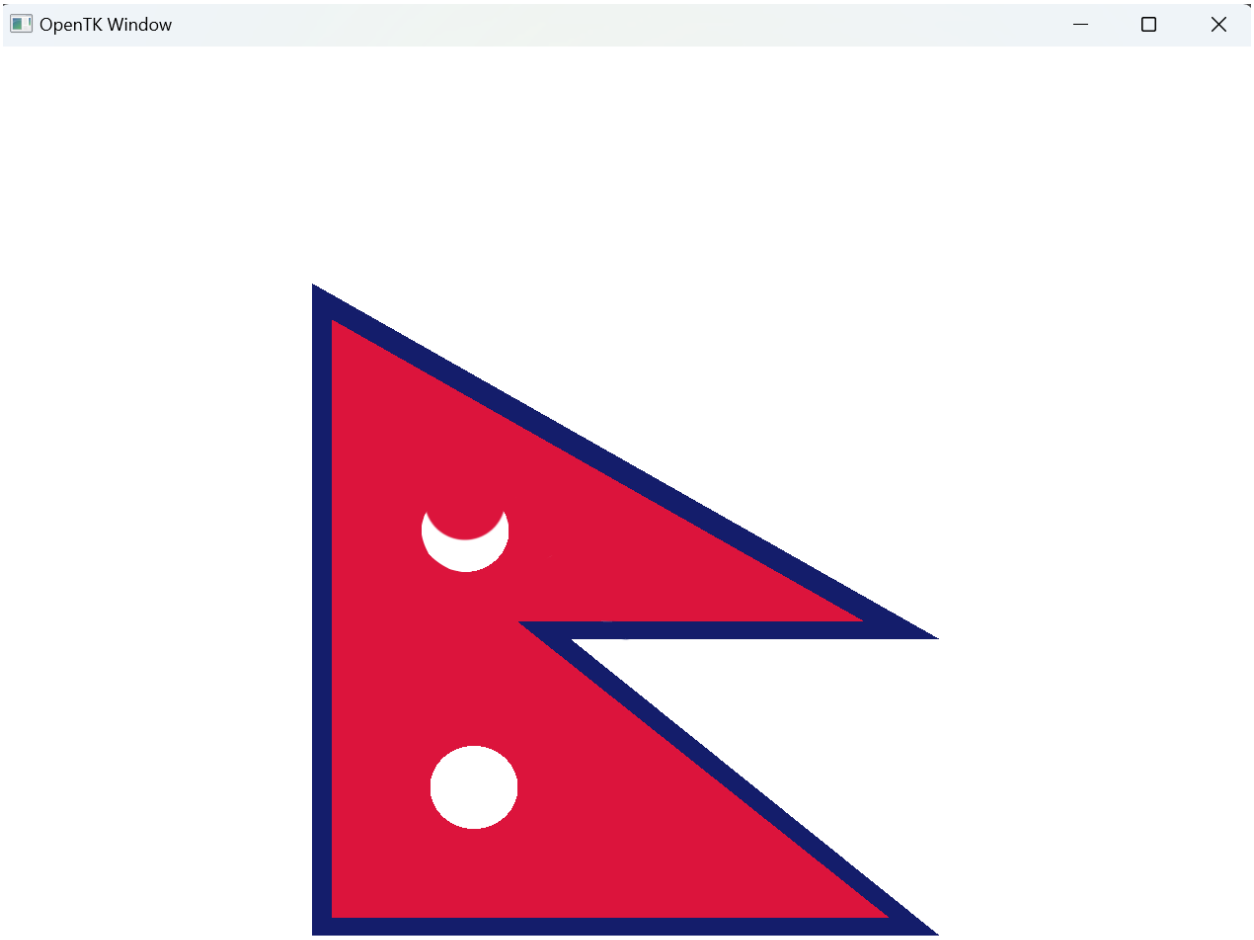
```
2 references
public class DisplayResolution
{
    1 reference
    public int Height { get; }
    1 reference
    public int Width { get; }

}
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        DisplayResolution resolution = new DisplayResolution();
        Console.WriteLine($"Screen resolution: {resolution.Width}x{resolution.Height}");

        using (Game game = new Game())
        {
            game.Run();
        }
    }
}
```

The full source code along with the output is available at

https://github.com/Rajaramkarki/Lab_1_Graphics

Output

The final output of the project is received as expected and is shown below.




```
Screen resolution: 0x0
Width: 2560 Height: 1570
```

Conclusion

In conclusion, the project for computer graphics was successfully completed by using C# programming language and the OpenTK graphics library. I was able to set up the graphics library with ease by using Microsoft Visual Studio. Through this lab, I also gained familiarity with the coordinate system and the geometrical functions and classes provided by the library to draw the flag of Nepal and color it appropriately. The set-up process was challenging but after its set up, the drawing is a smooth run.