Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report - I

[COMP 342]

Submitted by:

Abhijeet Poudel (44)

Submitted to:

Mr. Dhiraj Shrestha

Department of Computer Science and Engineering

Submission Date: 18/12/2022

TABLE OF CONTENTS

Chapter 1: Introduction 3
1.1 Language Primitives and Graphics Library 3
Chapter 2: Helper Functions 3
2.1 toNVC() Function 3
2.2 toNVC2() Function 4
2.3 altList() Function 4
Chapter 3: Mid- Point Circle Drawing Algorithm 4
3.1. Algorithm for Circle Drawing 4
3.2. Source Code 5
3.3. Output 8
3.3.1. Circle at center (0,0) 8
3.3.2. Circle at center (50, 50) 9
Chapter 4: Mid-point Ellipse Drawing Algorithm 9
4.1. Algorithm 9
4.2. Source Code 11
4.3. Output 15
4.3.1. Ellipse at center (0,0) 15
4.3.2. Ellipse at center (50,50) 15
Chapter 5: Conclusion 16

Chapter 1: Introduction

1.1 Language Primitives and Graphics Library

Graphics Library: PyOpenGL 3.1.6

Programming Language: Python 3.10.5

Window Context: GLFW

Helpers: Numpy

Chapter 2: Helper Functions

Helper functions can help to make code more concise and easier to read by breaking up complex tasks into smaller tasks. This can also help to improve code maintainability and reduce the risk of errors by eliminating the need to repeat code. Additionally, helper functions can help to abstract away complex logic, making the code easier to understand and debug.

In this report we have made use of 3 different helper funtions namely 1 toNVC(), altList() anf toNVC2()

2.1 toNVC() Function

```
def toNVC(xList, yList, resolution):
  for i in range(len(xList)):
    xList[i] = (xList[i]) / (resolution)
    yList[i] = (yList[i]) / (resolution)
  coordinateList = altList(xList,yList)
  return coordinateList
```

This code is used to convert a given list of coordinates from their original values to normalized values. It takes in two lists of x and y coordinates, as well as a resolution value. It then divides each coordinate by the resolution value, creating a normalized coordinate list. Finally, it returns the normalized coordinate list.

2.2 toNVC2() Function

```
def toNVC2(lst,resolution):
  for i in range(len(lst)):
    lst[i] = (lst[i]) / (resolution)

  return lst
```

This code takes a list of numbers (lst) and a resolution value and divides each item in the list by the resolution value. It returns the list with the new values. This code could be used for normalizing a list of data for a specific resolution.

2.3 altList() Function

```
def altList(lst1, lst2):
    return [sub[item] for item in range(len(lst2))
                      for sub in [lst1, lst2]]
```

This code is used to create an alternate list using two lists. The function altList takes in two lists, lst1 and lst2, and returns a list with the items of lst1 and lst2 alternating. For example, altList([1,2,3], [4,5,6]) would return [1,4,2,5,3,6].

Chapter 3: Mid- Point Circle Drawing Algorithm

3.1. Algorithm for Circle Drawing

Following is the algorithm used to draw a circle using mid-point circle drawing algorithm:

Step 1: Take inputs radius (r) and center (xc, yc)

Step 2: Obtain the first point on the circumference of the circle (x+ xc, y+yc)

Step 3: Set the initial decision parameter to pk = 1-r

Step 4: At each xk position, starting at k=0, perform the following test:

a. If pk< 0:
    i. The next pixel to be plotted : (x, y + 1)
    ii. pk= pk 2 * y +1

b. If pk> 0:
    i. The next pixel to be plotted : (x-1, y+1)
    ii. Pk= pk +2 * y – 2 * x + 1

c. Stop the loop if x< y

Step 5: Plot the new values of x and y

Step 6: Plot the corresponding octant symmetry points for the plotted point using the transformation:

(x, y) is mapped to (y, x) , (-y, x), (-x, y), (y, -x), (x, -y), (-y, -x), (-x, -y)

Step 7: Repeat from step 4 to 6 until x>= y.

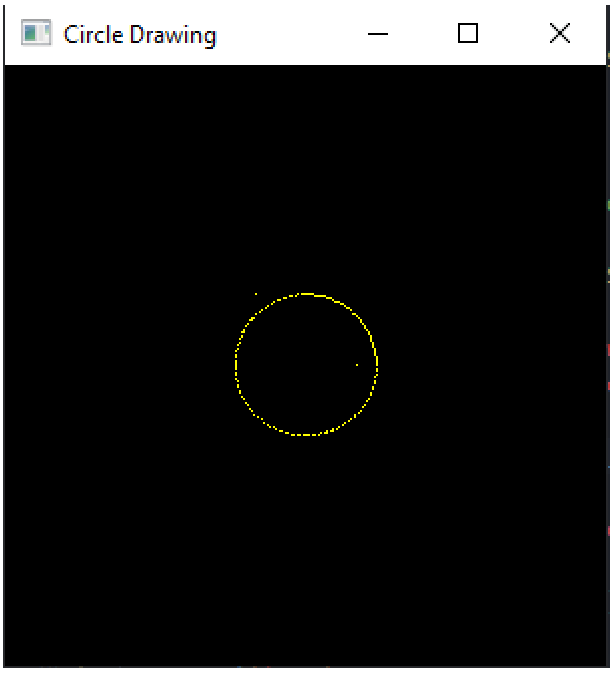
3.2. Source Code

```python
import glfw
import numpy as np
from OpenGL.GL import *
from OpenGL.GL.shaders import compileProgram, compileShader
from helpers import toNVC2, altList


RESOLUTION = 300


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


def midPointCircle(x_centre, y_centre, r, resolution):
    x_points = []
    y_points = []
    x = 0
    y = r
    p = 1 - r
    while x <= y:
        x_points.append(x)
        y_points.append(y)
        if p < 0:
            p = p + 2 * x + 3
        else:
            p = p + (2 * (x - y)) + 5
            y = y - 1
        x = x + 1

    neg_x_list = [-i + x_centre for i in x_points]
    neg_y_list = [-i + y_centre for i in y_points]
    x_points = [i + x_centre for i in x_points]
    y_points = [i + y_centre for i in y_points]

    midPointPoints = (
        altList(x_points, y_points)
        + altList(neg_x_list, y_points)
        + altList(neg_x_list, neg_y_list)
        + altList(x_points, neg_y_list)
        + altList(y_points, x_points)
        + altList(neg_y_list, x_points)
        + altList(neg_y_list, neg_x_list)
        + altList(y_points, neg_x_list)
    )
    return toNVC2(midPointPoints, resolution)


def main():

    vertex_src = """
   #version 330

   layout(location=0) in vec3 aPos;

   void main(){

    gl_Position =vec4(aPos,1.0f);

   }

  """
    fragment_src = """
  #version 330

  out vec4 FragColor;

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

  }

  """
    if not glfw.init():
        raise Exception("glfw cannot be initialised")

    window = glfw.create_window(RESOLUTION, RESOLUTION, "Circle
Drawing", None, None)

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

    glfw.set_window_pos(window, RESOLUTION, RESOLUTION)

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

    temp = midPointCircle(50, 50, 70, RESOLUTION)

    vertices = np.array(temp, dtype=np.float32)

    render_count = round(len(temp))

    print(temp)
    indices = np.array([i for i in range(1, len(vertices))],
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_STATIC_DRAW)

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

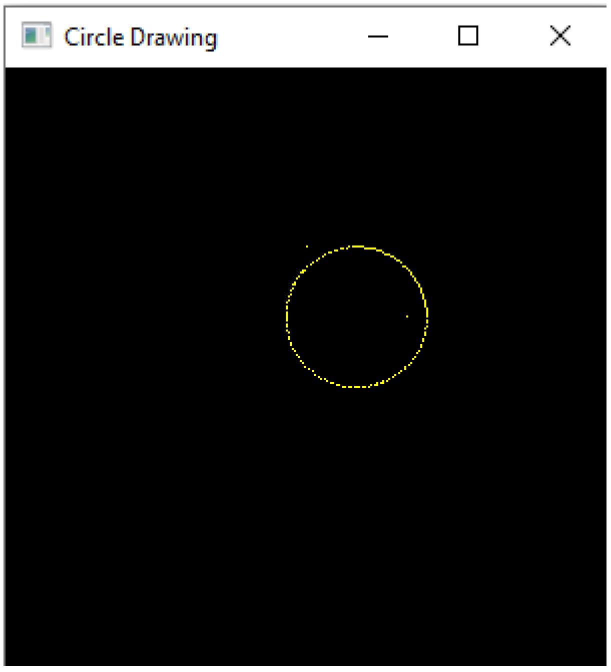
```
    glEnableVertexAttribArray(0)
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_TRUE, 0, 
ctypes.c_void_p(0))

    glUseProgram(shader)

    print(render_count)

    while not glfw.window_should_close(window):

        glfw.poll_events()

        glDrawElements(GL_POINTS, len(indices), GL_UNSIGNED_BYTE, None)

        glfw.swap_buffers(window)

    glfw.terminate()


main()
```

3.3. Output

3.3.1. Circle at center (0,0)



3.3.2. Circle at center (50, 50)



Chapter 4: Mid-point Ellipse Drawing Algorithm

4.1. Algorithm

Following is the algorithmic step for mid point ellipse drawing:

Step 1: Take major axis, minor axis and center of the ellipse as inputs: rx, ty, xc, yc respectively.

Step 2: We start of region 1 of the ellipse:

a. pk1 = (ry * ry) – (rx * rx * ry) + (0.25 * rx * rx)

b. dx = 2 * ry * ry *x

c. dx = 2 * rx * rx * y

Step 3: For any given pixel (x, y), the next pixel to be plotted is either (x + 1, y) or (x+1, y-1). This can be decided by following the steps below.

```
a. Plot (x + xc, y + yc) and also plot the 4-way symmetry points using the
   transformation (x, y) mapped to (-x, y), (x, -y), (-x, -y)
b. If pk1 < 0
    i. x = x + 1
    ii. Recalculate decision parameters as
        1. dx = dx + (2 * ry * ry)
        2. pk1 = pk1 + dx + (ry * ry)
c. else
    i. x = x + 1
    ii. y = y – 1
    iii. Recalculate decision parameters as
        1. dx = dx + (2 * ry * ry)
        2. dy = dy - (2 * rx * rx)
        3. pk1 = pk1 + dx – dy + (ry * ry)
```

Step 4: Repeat step 3 while dx < dy

Step 5: We start of region 2 of the ellipse:

```
a. pk2 = ((ry * ry) * ((x + 0.5) * (x + 0.5))) + ((rx * rx) * ((y - 1) * (y - 1)))
- (rx * rx * ry * ry)
b. dx = 2 * ry * ry *x
```

c. dx = 2 * rx * rx * y

Step 6: For any given pixel (x, y), the next pixel to be plotted is either (x, y-1) or (x+1, y-1). This can be decided by following the steps below.

a. Plot (x + xc, y + yc) and also plot the 4-way symmetry points using the transformation (x, y) mapped to (-x, y), (x, -y), (-x, -y)

b. If pk1 < 0

i. Calculate new point as (x, y-1)

ii. Recalculate decision parameters as

1. dy = dy - (2 * rx * rx)

2. pk2 = pk2 + (rx * rx) – dy

c. else

i. Calculate new point as (x+1, y-1)

ii. Recalculate decision parameters as

1. dx = dx + (2 * ry * ry)

2. dy = dy - (2 * rx * rx)

3. pk2 = pk2 + dx - dy + (rx * rx)

Step 7: Repeat step 6 while y>=0

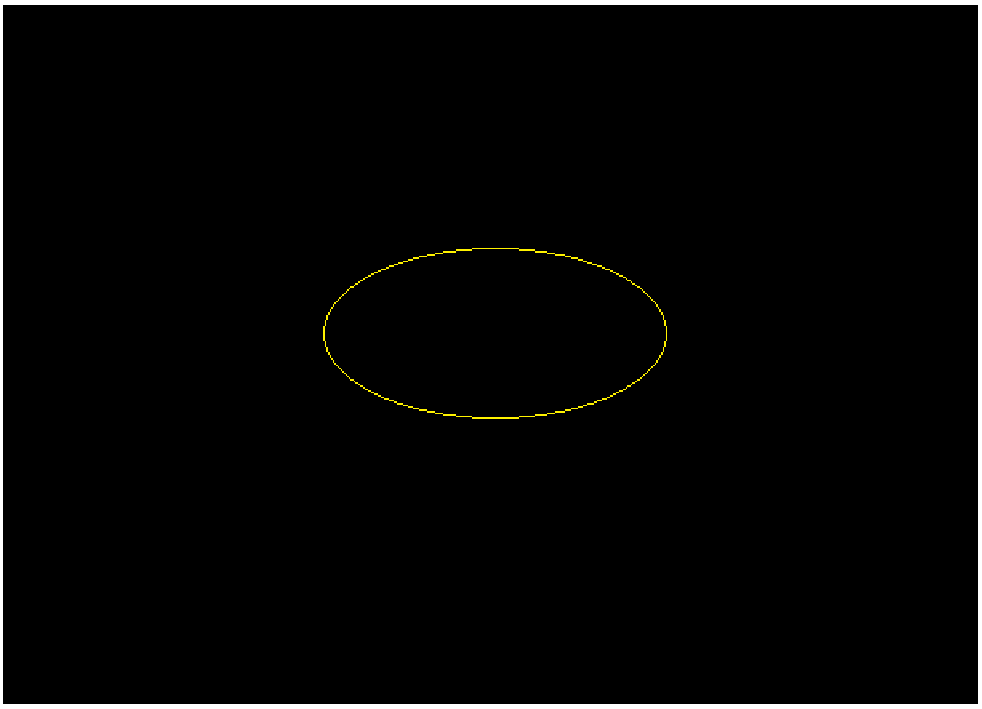
4.2. Source Code

```python
import glfw
import numpy as np
from OpenGL.GL import *
from OpenGL.GL.shaders import compileProgram, compileShader
from helpers import toNVC2, toNVC, altList
```

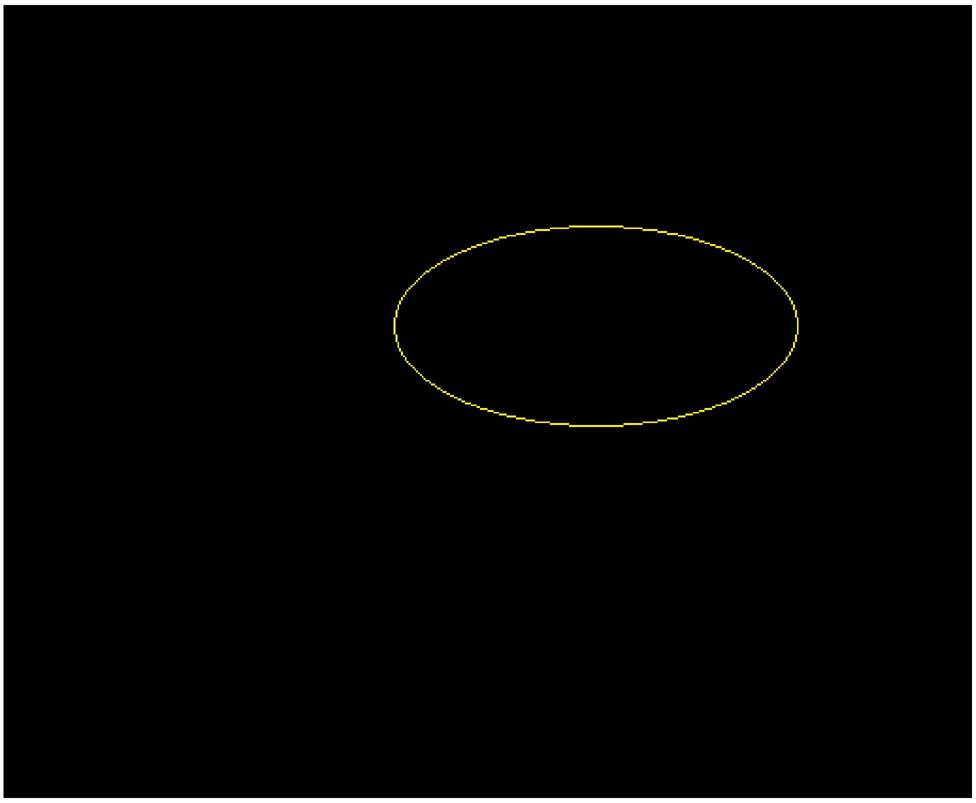
```
resolution = 600


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


def ellipse_algo(xc, yc, rx, ry, resolution):
    x = 0
    y = ry
    x_coordinates = []
    y_coordinates = []

    d1 = (ry * ry) - (rx * rx * ry) + (0.25 * rx * rx)
    dx = 2 * ry * ry * x
    dy = 2 * rx * rx * y

    while dx < dy:
        x_coordinates.append(x + xc)
        x_coordinates.append(-x + xc)
        y_coordinates.append(y + yc)
        y_coordinates.append(-y + yc)

        if d1 < 0:
            x += 1
            dx = dx + (2 * ry * ry)
            d1 = d1 + dx + (ry * ry)
        else:
            x += 1
            y -= 1
            dx = dx + (2 * ry * ry)
            dy = dy - (2 * rx * rx)
            d1 = d1 + dx - dy + (ry * ry)

    # Decision parameter of region 2
    d2 = (
        ((ry * ry) * ((x + 0.5) * (x + 0.5)))
        + ((rx * rx) * ((y - 1) * (y - 1)))
        - (rx * rx * ry * ry)
    )

    # Plotting points of region 2
    while y >= 0:

        # printing points based on 4-way symmetry
        x_coordinates.append(x + xc)
        x_coordinates.append(-x + xc)
        y_coordinates.append(y + yc)
        y_coordinates.append(-y + yc)
        # Checking and updating parameter
        # value based on algorithm
        if d2 > 0:
            y -= 1
            dy = dy - (2 * rx * rx)
            d2 = d2 + (rx * rx) - dy
        else:
            y -= 1
            x += 1
            dx = dx + (2 * ry * ry)
            dy = dy - (2 * rx * rx)
            d2 = d2 + dx - dy + (rx * rx)

    return toNVC2(altList(x_coordinates, y_coordinates), resolution)


def main():

    vertex_src = """
    #version 330
    layout (location=0) in vec2 a_position;
    void main(){
        gl_Position=vec4(a_position,0.0f, 1.0f);
    }
    """

    fragment_src = """

    #version 330

    out vec4 FragColor;

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

    """

    # checking and initializing glfw library
    if not glfw.init():
        raise Exception("glfw cannot be initialised")

    # creating window, width, height, name, monitor, share
    window = glfw.create_window(resolution, resolution, "Ellipse", None,
None)

    # check if window
    if not window:
        glfw.terminate()
        raise Exception("glfw window cannot be created!")

    # set size callback and window is resized
    glfw.set_window_size_callback(window, window_resize)
    # context initializes opengl  a state machine that stores all data
related to rendering
    glfw.make_context_current(window)

    dda_call = ellipse_algo(0, 0, 20, 90, resolution)

    vertices = np.array(dda_call, dtype=np.float32)

    render_count = round(len(dda_call) / 2)

    print(render_count)

    indices = np.array([i for i in range(1, render_count + 1)],
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_STATIC_DRAW)

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

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

    glUseProgram(shader)
    while not glfw.window_should_close(window):
        glfw.poll_events()
        glDrawElements(GL_POINTS, len(indices), GL_UNSIGNED_BYTE, None)

        glfw.swap_buffers(window)

    glfw.terminate()

main()
```

4.3. Output

4.3.1. Ellipse at center (0,0)



4.3.2. Ellipse at center (50,50)



Chapter 5: Conclusion

In conclusion, this lab report has demonstrated the use of mid-point circle and ellipse drawing algorithms using the GL_POINTS primitive from the OpenGL library to draw circles and ellipses. It has also shown that there is no direct way to draw circles in Modern OpenGL, but there are potential methods such as anti-aliasing and drawing with GL_TRIANGLES that can be used to improve the visuals of the output.