Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Circle and Ellipse Drawing Mid-Point Algorithm

Lab Report Three

[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: Circle and Ellipse Drawing Algorithm

1.1 Introduction

Circle and Ellipse Drawing Algorithms are used within Computer Graphics for approximating a Circular or Elliptical Shape on discrete graphical media. Through the use of mid-point algorithm, Circle or Ellipse are displayed within a graphical media through generation of discrete data points using a Center and Radius (major and minor in case of ellipse). These set of discrete data points when plotted onto the pixels of a window will give an illusion of a connected curved lines forming either a circle or an ellipse. In this Lab Work, we will be working on Mid-Point Algorithm for both Circle and Ellipse.

1.2 Additional Tools

The Programming Language, Graphics Library and Tools used for Generation Algorithms are as follows:

Programming Language: Python 3.10

Graphics Library: PyOpenGL 3.1.6

Window Renderer: GLUT

Helper Library: ctypes, numpy

The data points generated using the Mid-Point Algorithm 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 datasets to normalized form, I have created a helper function named `tonormalized`. It takes in the generated datasets 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)
```

Moreover, a Circle is Symmetric about the 8 Octants and Ellipse is Symmetric about the 4 Quadrants. The Mid-Point Algorithm utilizes this symmetricity property of these shapes to minimize the computation required to generate our data points. Hence, the initial data points generated using the Mid-Point Algorithm are used to generate 7 other data points for a Circle and 3 other data Points for an Ellipse.

So, I have created helper functions for Circle as well as Ellipse to assist in the generation of these Symmetry points.

Code Snippet for Circle Symmetry Helper:

```
def generateOtherPoints(data, center):
    circlePoints = []
    for point in data:
        circlePoints.append([point[0] + center[0], point[1] + center[1], point[2]])
        circlePoints.append([-point[0] + center[0], point[1] + center[1], point[2]])
        circlePoints.append([point[0] + center[0], -point[1] + center[1], point[2]])
        circlePoints.append([-point[0] + center[0], -point[1] + center[1], point[2]])
        circlePoints.append([point[1] + center[0], point[0] + center[1], point[2]])
        circlePoints.append([-point[1] + center[0], point[0] + center[1], point[2]])
        circlePoints.append([point[1] + center[0], -point[0] + center[1], point[2]])
        circlePoints.append([-point[1] + center[0], -point[0] + center[1], point[2]])

    return circlePoints
```

Code Snippet for Circle Symmetry Helper

```
def generateOtherPoints(data, center):
    ellipsePoints = []
    for point in data:
        ellipsePoints.append([point[0] + center[0], point[1] + center[1], point[2]])
        ellipsePoints.append([-point[0] + center[0], point[1] + center[1], point[2]])
        ellipsePoints.append([point[0] + center[0], -point[1] + center[1], point[2]])
        ellipsePoints.append([-point[0] + center[0], -point[1] + center[1], point[2]])

    return ellipsePoints
```

Chapter 2: Mid-Point Circle Drawing Algorithm

2.1 Algorithm

The algorithm used to generate the data points based on the Center and Radius of a circle through the use of Mid-Point Circle Drawing Algorithm is as follows:

1. Take Center (x, y) and Radius r of the Circle as inputs.
2. Check r > 0 : if True Continue with Step 3, else Abort
3. Set xValue to 0 and yValue to r.
4. Set Initial Decision Parameter as: Pk = 1 – r.
5. Store (xValue, yValue)
6. Perform the following test:
    a. If Pk < 0 : update Pk = pk + 2 * xValue + 3
    b. Else : update Pk = Pk * (xValue - yValue) + 5 and set yValue = yValue – 1
7. Update xValue = xValue + 1
8. Repeat Step 5 onwards until xValue > yValue

Note: While storing xValue and yValue, other 7 Symmetric Data Points are calculated and stored as well. Moreover, the Center (x_c, y_c) is added to the respective coordinates of these stored values. So, the 7 Symmetric Points for (xValue, yValue) are:

(-xValue, yValue), (xValue, -yValue), (-xValue, -yValue), (yValue, xValue), (-yValue, xValue), (yValue, -xValue), (-yValue, -xValue)

and, the stored data values are:

(x_c + xValue, y_c + yValue), (x_c - xValue, y_c + yValue),
(x_c + xValue, y_c - yValue), (x_c - xValue, y_c - yValue),
(x_c + yValue, y_c + xValue), (x_c - yValue, y_c + xValue),
(x_c + yValue, y_c - xValue), (x_c - yValue, y_c - xValue)

2.2 Source Code

```
import os
import sys
import ctypes
import numpy as np
import OpenGL.GL as gl
import OpenGL.GLUT as glut

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

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

# -- Building Data --
def circleDrawing():
    data = []

    if len(sys.argv) == 6:
        radius = int(sys.argv[1])
        center = [int(sys.argv[2]), int(sys.argv[3])]
        resolution = [int(sys.argv[4]), int(sys.argv[5])]

        if radius > 0:
            xValue = 0
            yValue = radius

            Pk = 1 - radius
            while (xValue <= yValue):

                data.append([xValue, yValue, 1.0])

                if (Pk < 0):
                    Pk = Pk + 2 * xValue + 3
                else:
                    Pk = Pk + 2 * (xValue - yValue) + 5
                    yValue = yValue - 1

                xValue = xValue + 1

            data = generateOtherPoints(data, center)
    else:
        raise Exception("Arguments do not match. Correctly Enter Parameters in
format : [radius, center X, center Y, resoultion X and resoultion Y]")

    return data, resolution


def generateOtherPoints(data, center):
    circlePoints = []
    for point in data:
        circlePoints.append([point[0] + center[0], point[1] + center[1], point[2]])
        circlePoints.append([-point[0] + center[0], point[1] + center[1],
point[2]])
        circlePoints.append([point[0] + center[0], -point[1] + center[1],
point[2]])
```

```
        circlePoints.append([-point[0] + center[0], -point[1] + center[1],
point[2]])
        circlePoints.append([point[1] + center[0], point[0] + center[1], point[2]])
        circlePoints.append([-point[1] + center[0], point[0] + center[1],
point[2]])
        circlePoints.append([point[1] + center[0], -point[0] + center[1],
point[2]])
        circlePoints.append([-point[1] + center[0], -point[0] + center[1],
point[2]])


    return circlePoints

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)

# 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():
    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])

    # 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_POINTS, 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 = circleDrawing()
data = tonormalized(data, resolution)
initialize()

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

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

2.3 Outputs

The program takes in the radius, center and resolution of the screen for which these data are for using the command line arguments. For example: the command `python circledrawing.py 100 50 50 320 320` is used to initialize the Radius as 100 units, Center as (50, 50) and the Screen Resolution as (320, 320).

2.3.1 Circle with Radius 100 and Center at (50, 50)

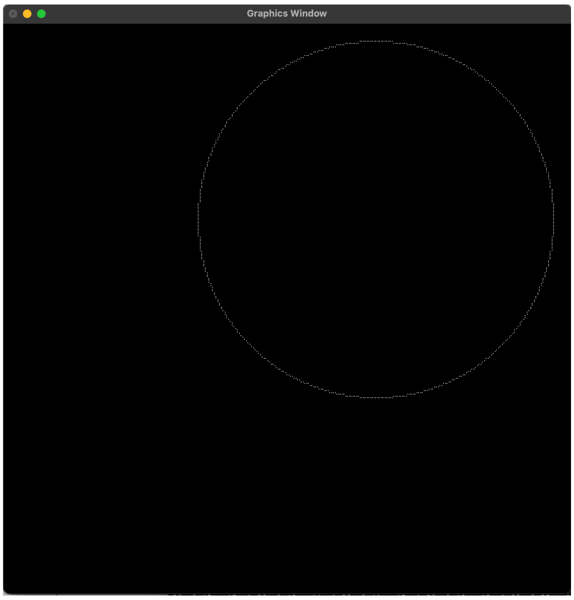

fig 2.3.1.1: Circle at Center (50,50) with Radius 100

2.3.2 Circle with Radius 100 and Center at (0, 0)

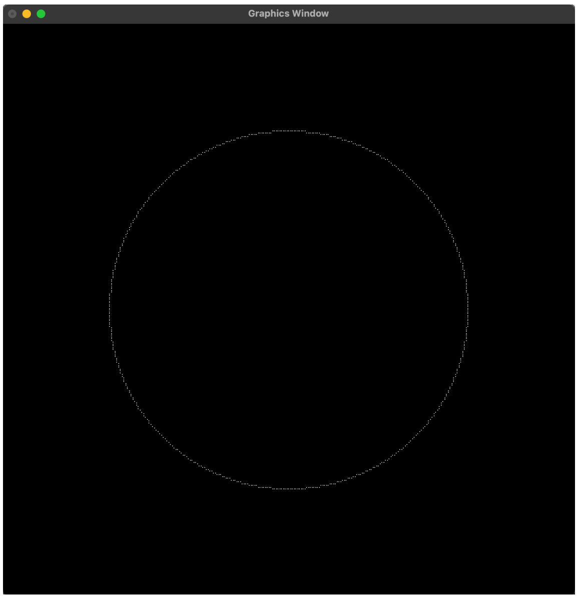

fig 2.3.2.1: Circle at Center (0,0) with Radius 100

Chapter 3: Mid-Point Ellipse Drawing Algorithm

3.1 Algorithm

The algorithm used to generate the data points based on the Center, Major Radius and the Minor Radius of an Ellipse through the use of Mid-Point Ellipse Drawing Algorithm is as follows:

1. Take Center (x, y) and Major Radius rx and Minor Radius ry of the Ellipse as inputs.
2. We Start with the Region 1 of Ellipse: | Slope | < 1
3. Set xValue to 0 and yValue to ry.
4. Set Initial Decision Parameter as: P1k = (ry * ry) + (0.25 * rx * rx) – (rx * rx * ry)
5. Compute delX and delY as:
    a. delX = 2 * ry * ry * xValue
    b. delY = 2 * rx * rx * yValue
6. Store (xValue, yValue)
7. Update xValue = xValue + 1
8. Perform the following test:
    a. If P1k < 0 :
        i. Update delX = delX + 2 * ry * ry
        ii. Update P1k = P1k + delX + ry * ry
    b. Else :
        i. Update yValue = yValue – 1
        ii. Update delX = delX + 2 * rx * rx
        iii. Upate delY = delY – 2 * rx * rx

        iv. Update P1k = P1k + delX – delY + ry * ry

9. Repeat Step 6 onwards while delX < delY

10. Now we are at the Start of Region 2 of Ellipse

11. Compute P2k as:

    P2k = ry * ry * (xValue + 0.5) * (xValue + 0.5) + rx * rx * (yValue -1) * (yValue -1)

12. Store (xValue, yValue)

13. Update yValue = yValue - 1

14. Perform the following test

    a. If P2k < 0 :

        i. Update xValue = xValue + 1

        ii. Update delX = delX + 2 * ry * ry

        iii. Update delY = delY – 2 * rx * rx

        iv. Update P2k = P2k + delX – delY = rx * rx

    b. Else :

        i. Update delY = delY = 2 * rx * rx

        ii. Update P2k = P2k + rx * rx – delY

15. Repeat Step 12 onwards while yValue >= 0

Note: While storing xValue and yValue, other 3 Symmetric Data Points are calculated and stored as well. Moreover, the Center (x_c, y_c) is added to the respective coordinates of these stored values. So, the 3 Symmetric Points for (xValue, yValue) are: **(-xValue, yValue), (xValue, -yValue), (-xValue, -yValue)**

and, the stored data values are: **(x_c + xValue, y_c + yValue), (x_c - xValue, y_c + yValue), (x_c + xValue, y_c - yValue), (x_c - xValue, y_c - yValue)**

3.2 Source Code

```
import os
import sys
import ctypes
import numpy as np
import OpenGL.GL as gl
import OpenGL.GLUT as glut

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

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

# -- Building Data --
def ellipseDrawing():
    data = []
    if len(sys.argv) >= 7:
        radius = [int(sys.argv[1]), int(sys.argv[2])]
        center = [int(sys.argv[3]), int(sys.argv[4])]
        resolution = [int(sys.argv[5]), int(sys.argv[6])]
        rX, rY = radius
        rXSquared = rX ** 2
        rYSquared = rY ** 2

        xValue = 0
        yValue = rY
        P1k = rYSquared + (1/4) * (rXSquared) - (rXSquared * rY)
        delX = 2 * rYSquared * xValue
        delY = 2 * rXSquared * yValue

        while (delX < delY):
            data.append([xValue, yValue, 1.0])
            xValue = xValue + 1
            if (P1k < 0):
                delX = delX + 2 * rYSquared
                P1k = P1k + delX +  rYSquared
            else:
                yValue = yValue - 1
                delX = delX + 2 * rYSquared
                delY = delY - 2 * rXSquared
                P1k = P1k + delX - delY + rYSquared

        P2k = rYSquared * (xValue + 1/2) * (xValue + 1/2) + rXSquared * (yValue -
1) * (yValue - 1) - rXSquared*rYSquared

        while (yValue >= 0):

            data.append([xValue, yValue, 1.0])

            yValue = yValue - 1

            if P2k < 0:
                xValue = xValue + 1
                delX = delX + (2 * rYSquared)
                delY = delY - (2 * rXSquared)
```

```
                P2k = P2k + delX - delY + rXSquared
            else:
                delY = delY - 2 * rXSquared
                P2k = P2k + rXSquared - delY

        data = generateOtherPoints(data, center)
    else:
        raise Exception("Arguments do not match. Correctly Enter Parameters in
format : [ major radius X, minor radius Y, center X, center Y, resoultion X and
resoultion Y]")

    return data, resolution

def generateOtherPoints(data, center):
    ellipsePoints = []
    for point in data:
        ellipsePoints.append([point[0] + center[0], point[1] + center[1],
point[2]])
        ellipsePoints.append([-point[0] + center[0], point[1] + center[1],
point[2]])
        ellipsePoints.append([point[0] + center[0], -point[1] + center[1],
point[2]])
        ellipsePoints.append([-point[0] + center[0], -point[1] + center[1],
point[2]])

    return ellipsePoints

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)

# 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)
```

```python
    return program

# 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)

    # 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])

    # 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_POINTS, 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 = ellipseDrawing()
data = tonormalized(data, resolution)
initialize()

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

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

3.3 Outputs

The program takes in the major radius, minor radius, center and resolution of the screen for which these data are for using the command line arguments. For example: the command `python ellipsedrawing.py 100 50 0 0 320 320` is used to initialize the Major Radius (Rx) as 100 units, Minor Radius (Ry) as 50 units Center as (0, 0) and the Screen Resolution as (320, 320).

3.3.1 Ellipse with Major Radius 100, Minor Radius 50, and Center at (0, 0)

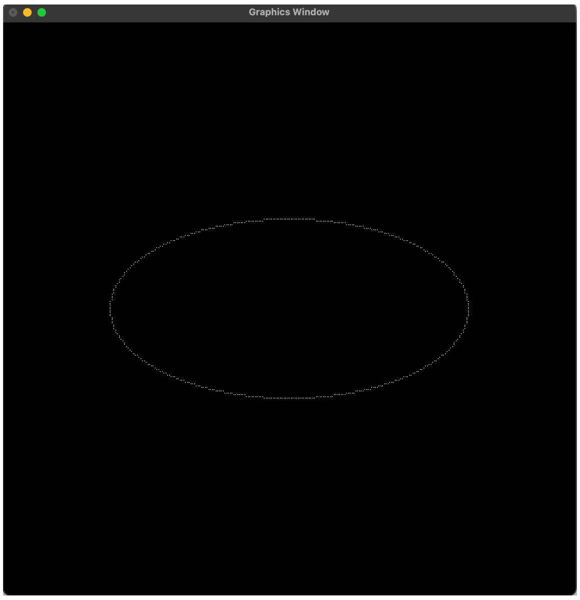

fig 3.3.1.1: Ellipse at Center (0,0) with Major Radius 100, Minor Radius 50

3.3.2 Ellipse with Major Radius 100, Minor Radius 50, and Center at (25, 25)

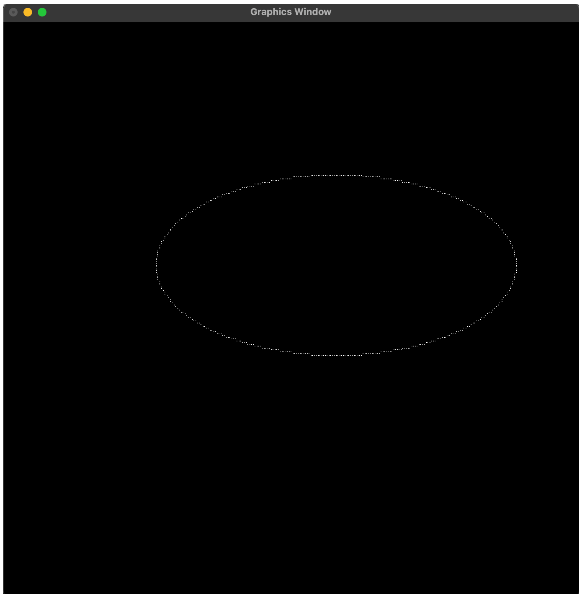

fig 3.3.2.1: Ellipse at Center (25,25) with Major Radius 100, Minor Radius 50

3.3.3 Ellipse with Major Radius 50, Minor Radius 100, and Center at (0, 0)

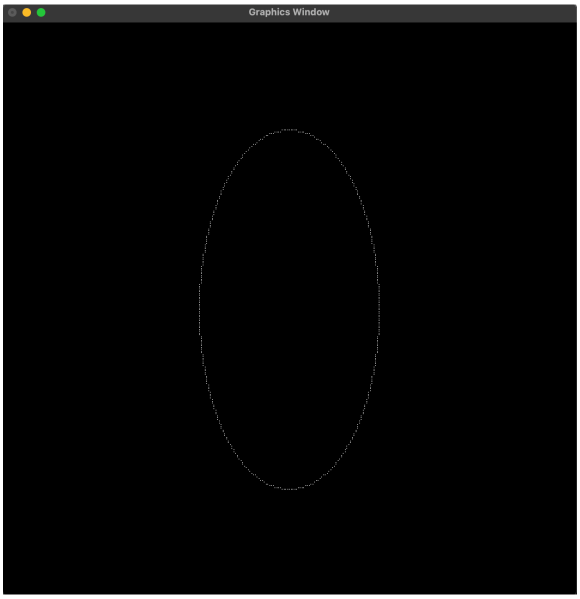

fig 3.3.3.1: Ellipse at Center (0,0) with Major Radius 50, Minor Radius 100

3.3.4 Ellipse with Major Radius 50, Minor Radius 100, and Center at (25, 25)

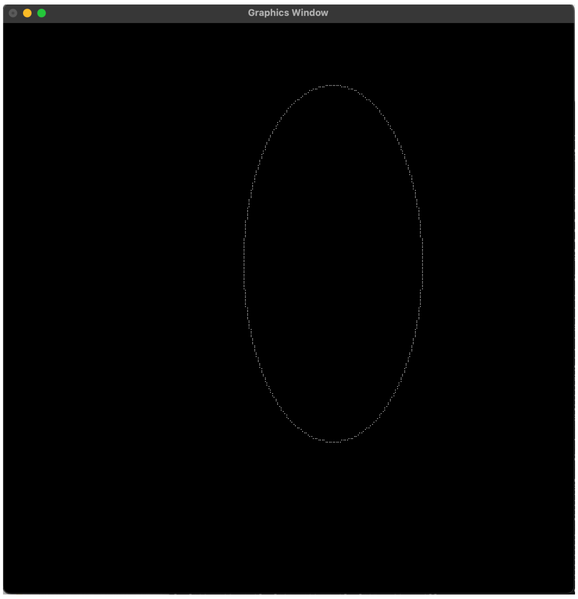

fig 3.3.4.1: Ellipse at Center (25,25) with Major Radius 50, Minor Radius 100

Chapter 5: Conclusion

Through this Lab Work, I was able to study the details of Mid-Point drawing algorithm in drawing a Circle and an Ellipse whilst also recognizing the need to identify and carefully pick between the given pixel choices in order to draw a simple connected curved line segment to properly depict the curvature of these shapes. The written programs use the gl.GL_POINTS primitive supported by OpenGL to demonstrate the creation of an approximately correct line segment on the graphical media instead of the gl.GL_LINES primitive. This is done so to correctly portray the usage of the Mid-Point algorithm as it focuses on creation of shapes through illumination of individual pixels, or in our case individual points. So, using the gl.GL_LINES to draw a line segments connecting the generated points would defeat the purpose of implementing the given algorithm.

Moreover, as seen in the outputs if denser points were to be identified by increasing the resolution of the display during the normalization phase we would observe circle and ellipses that appear to be joint with no gaps in-between the plotted points. However, this was not pursued for this lab work as its main purpose was to visualize that drawing a circle or ellipse using the Mid-Point algorithm is nothing but plotting a set of closely identified points.