**Kathmandu University**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**



**Lab Report Two**

**[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 14, 2022

Table of Contents

[Chapter 1: Line Drawing Algorithms 1](#_Toc121869778)

[1.1 Introduction 1](#_Toc121869779)

[1.2 Additional Tools 1](#_Toc121869780)

[Chapter 2: Digital Differential Analyzer (DDA) 3](#_Toc121869781)

[2.1 Algorithm 3](#_Toc121869782)

[2.2 Source Code 4](#_Toc121869783)

[2.3 Outputs 7](#_Toc121869784)

[Chapter 3: Bresenham Line Drawing Algorithm 9](#_Toc121869785)

[3.1 Algorithm 9](#_Toc121869786)

[3.2 Source Code 10](#_Toc121869787)

[3.3 Outputs 13](#_Toc121869788)

[Chapter 4: Mid-Point Line Drawing Algorithm 15](#_Toc121869789)

[4.1 Algorithm 15](#_Toc121869790)

[4.2 Source Code 16](#_Toc121869791)

[4.3 Outputs 19](#_Toc121869792)

[Chapter 5: Conclusion 21](#_Toc121869793)

# Chapter 1: Line Drawing Algorithms

## 1.1 Introduction

Line Drawing Algorithms are used within Computer Graphics for approximating a line segment on discrete graphical media. A line segment is displayed within a graphical media through generation of discrete data points between the endpoints of the line segment which when plotted onto the pixels of a window will give an illusion of a connected straight line. In this Lab Work, we will be working on three different but widely popular line drawing algorithms, namely:

* Digital Differential Analyzer (DDA),
* Bresenham Line Drawing Algorithm,
* Mid-Point Line Generation Algorithm.

## 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 Line Drawing Algorithms 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)

Chapter 2: Digital Differential Analyzer (DDA)

## 2.1 Algorithm

The algorithm used to generate the data points between the two end points of a line segment using the Digital Differential Analyzer Line Drawing Algorithm is as follows:

1. Take Start Point (x, y) and End Point (x, y) as inputs.
2. Calculate the following values using these points:
   1. Del X = endPoint(x) – startPoint(x)
   2. Del Y = endPoint(y) – startPoint(y)
3. Perform the following test:
   1. If absolute(Del X) > absolute(Del Y) : set steps = absolute(Del X)
   2. Else : set steps = absolute(Del Y)
4. Calculate the increments of X and Y as:
   1. incX = Del X/steps
   2. incY = Del Y/steps
5. Set xValue to be startPoint(x) and yValue to be startPoint(y)
6. Store (xValue, yValue)
7. Modify xValue, yValue and step values using:
   1. xValue = xValue + incX
   2. yValue = yValue + incY
   3. steps = steps - 1
8. Repeat Step 6 onwards until steps < 0.

## 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 digitalDifferential():

data = []

if len(sys.argv) == 5:

startPoint = [int(sys.argv[1]), int(sys.argv[2])]

endPoint = [int(sys.argv[3]), int(sys.argv[4])]

delX = endPoint[0] - startPoint[0]

delY = endPoint[1] - startPoint[1]

if abs(delX) > abs(delY):

steps = abs(delX)

else:

steps = abs(delY)

incX = delX/steps

incY = delY/steps

xValue = startPoint[0]

yValue = startPoint[1]

while(steps >= 0):

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

xValue = xValue + incX

yValue = yValue + incY

steps = steps - 1

else:

raise Exception("Arguments do not match. Correctly enter the Starting Point and the Ending Point")

return data

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 = digitalDifferential()

data = tonormalized(data, [300,380])

initialize()

glut.glutDisplayFunc(display)

glut.glutPostRedisplay()

glut.glutKeyboardFunc(keyboard)

# enter the mainloop

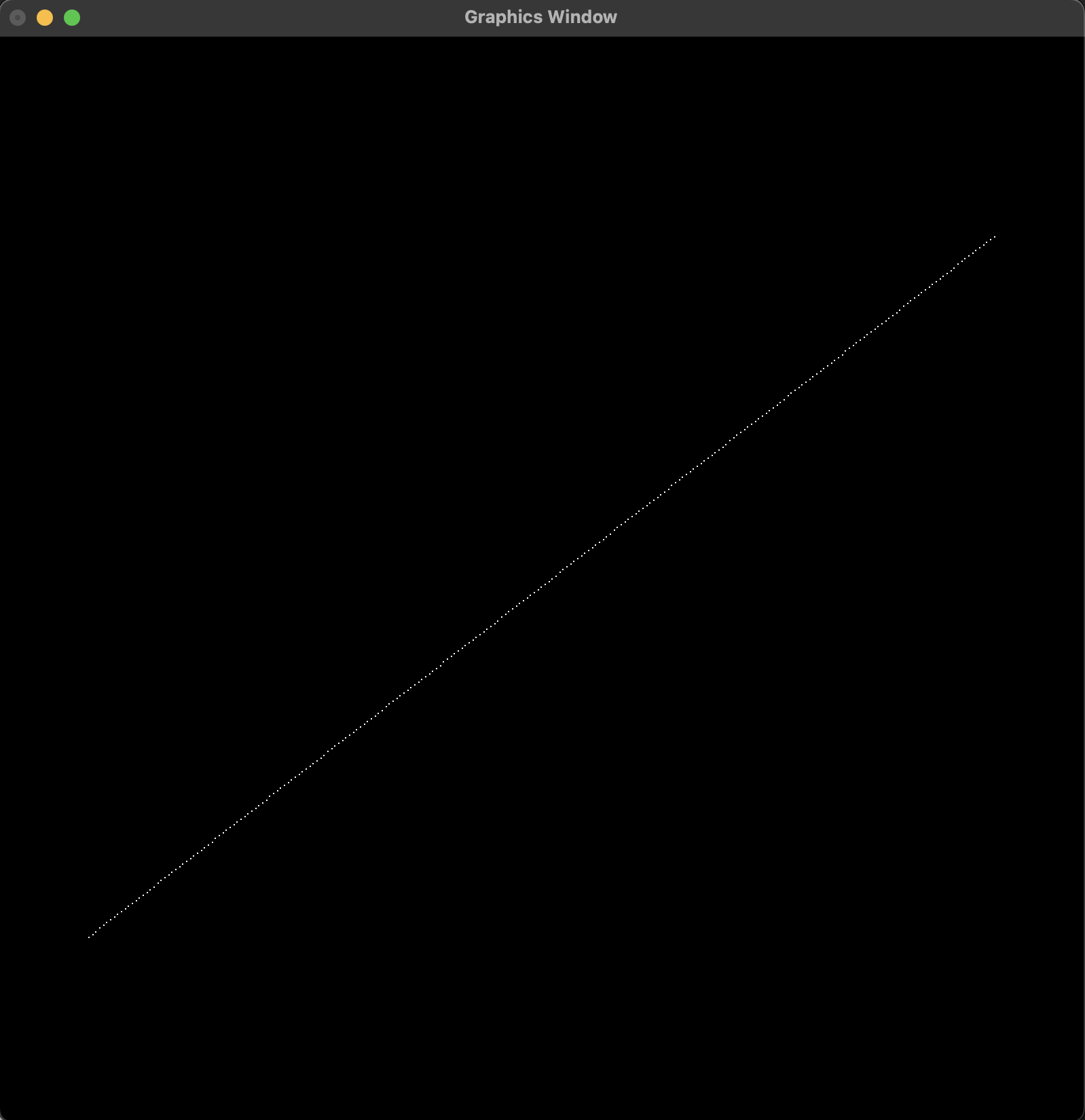
glut.glutMainLoop()

## 

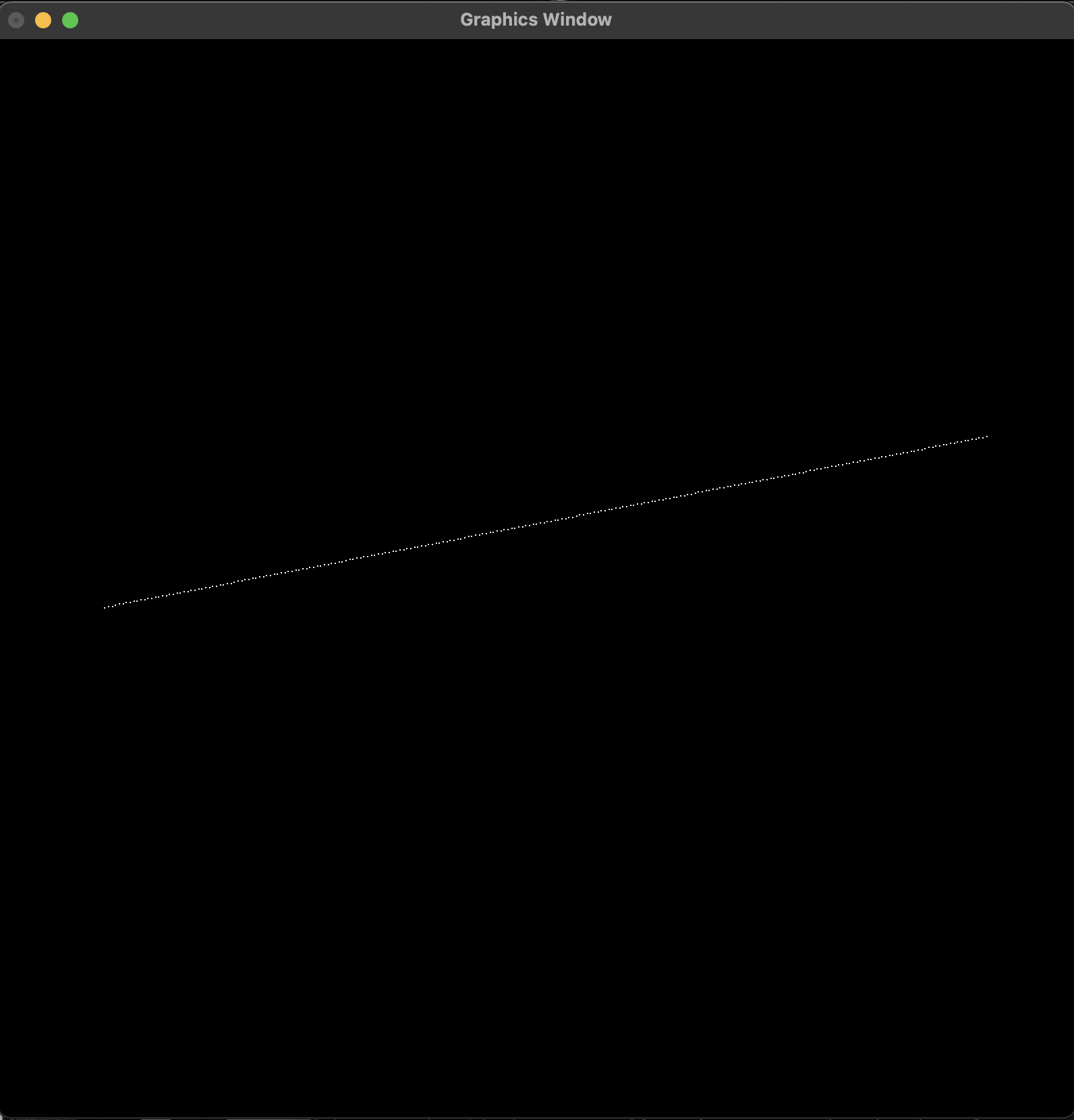
## 2.3 Outputs

The program takes in the starting and ending points of a line segment from the command line arguments. For example: the command python dda.py -125 -125 125 125 is used to initialize the start point as (-125, -125) and end point as (125, 125).

**2.3.1 Slope ( |m| >= 1 )**

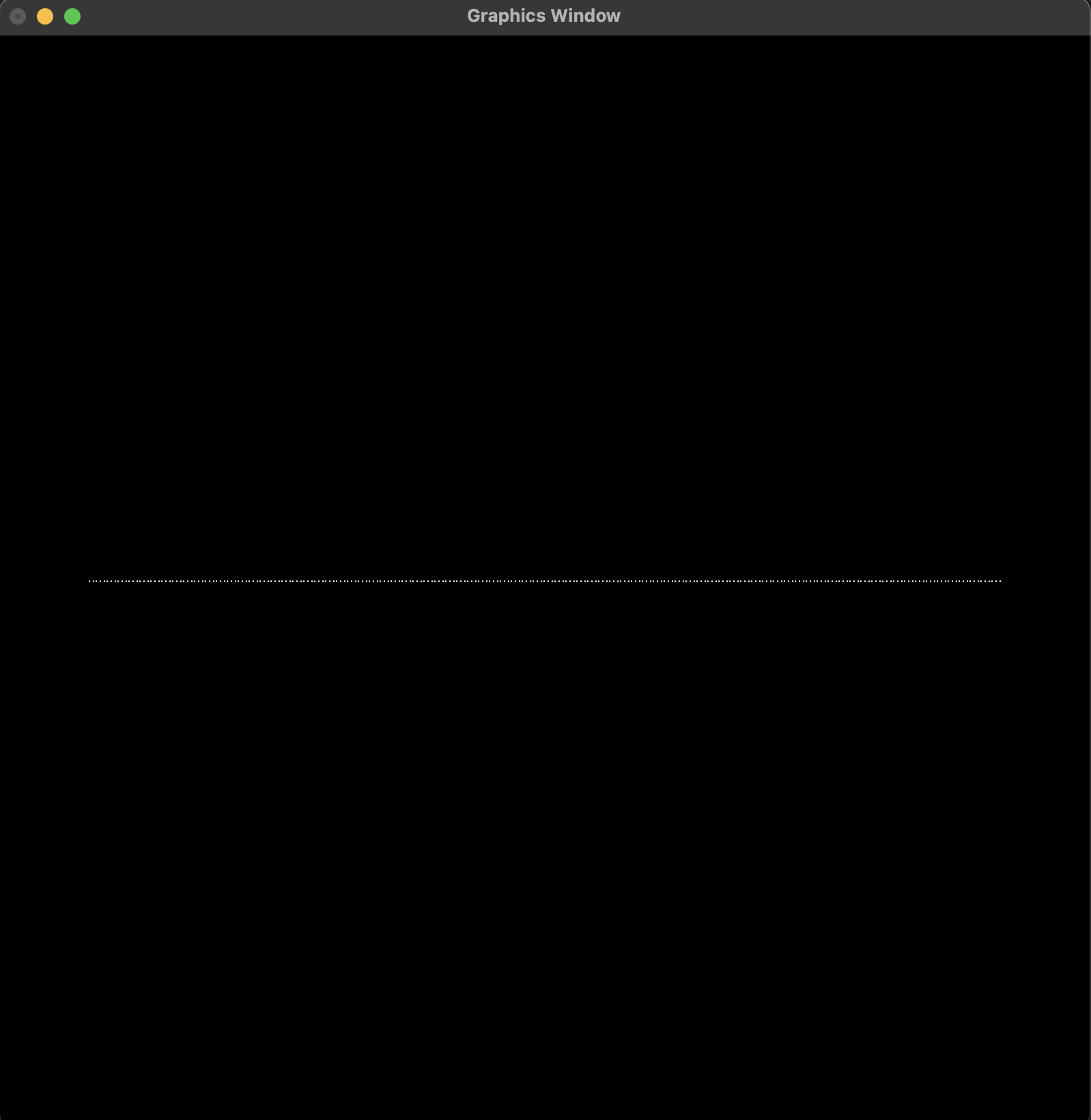
*****fig 2.3.1.1: Line from* (-125, -125) to (125, 120)

**2.3.2 Slope ( |m| < 1 )**



*fig 2.3.2.1: Line from* (-120, -10) to (125, 50)

**2.3.3 Slope ( |m| = 0 )**



*fig 2.3.3.1: Line from* (-120, 0) to (125, 0)

# Chapter 3: Bresenham Line Drawing Algorithm

## 3.1 Algorithm

The algorithm used to generate the data points between the two end points of a line segment having slopes ( both |m| >= 1 and |m| < 1) using Bresenham’s Line Drawing Algorithm is as follows:

1. Take Start Point (x, y) and End Point (x, y) as inputs.
2. Calculate the following values using these points:
3. Del X = | endPoint(x) – startPoint(x) |
4. Del Y = | endPoint(y) – startPoint(y) |
5. Calculate initial decision parameter Pk as:
6. Pk = 2 \* Del Y – Del X
7. Set xValue to startPoint(x) and yValue to startPoint(y)
8. At each (xk, yk) along the line, starting at k = 0:
9. Store (xValue, yValue)
10. Perform the following check:
11. if xValue < endPoint(x) : xValue = xValue + 1
12. Else : xValue = xValue – 1
13. Perform the following check:
14. If Pk < 0 : Pk = Pk + 2 \* Del Y
15. Else : Perform the following check
16. If yValue < endPoint(y) : yValue = yValue + 1
17. Else : yValue = yValue – 1

And, Update Pk = Pk + 2 \* (Del Y – Del X)

1. Repeat Step 6 onwards Del X number of times

## 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 bresenhamAlgo():

data = []

if len(sys.argv) == 5:

startPoint = [int(sys.argv[1]), int(sys.argv[2])]

endPoint = [int(sys.argv[3]), int(sys.argv[4])]

delX = abs(endPoint[0] - startPoint[0])

delY = abs(endPoint[1] - startPoint[1])

print(delY/delX)

Pk = 2 \* delY - delX

xValue = startPoint[0]

yValue = startPoint[1]

for i in range(0, delX + 1):

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

if xValue < endPoint[0]:

xValue = xValue + 1

else:

xValue = xValue - 1

if Pk < 0:

Pk = Pk + 2 \* delY

else:

if yValue < endPoint[1]:

yValue = yValue + 1

else:

yValue = yValue - 1

Pk = Pk + 2 \* (delY - delX)

else:

raise Exception("Arguments do not match. Correctly enter the Starting Point and the Ending Point")

return data

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 = bresenhamAlgo()

data = tonormalized(data, [300,300])

initialize()

glut.glutDisplayFunc(display)

glut.glutPostRedisplay()

glut.glutKeyboardFunc(keyboard)

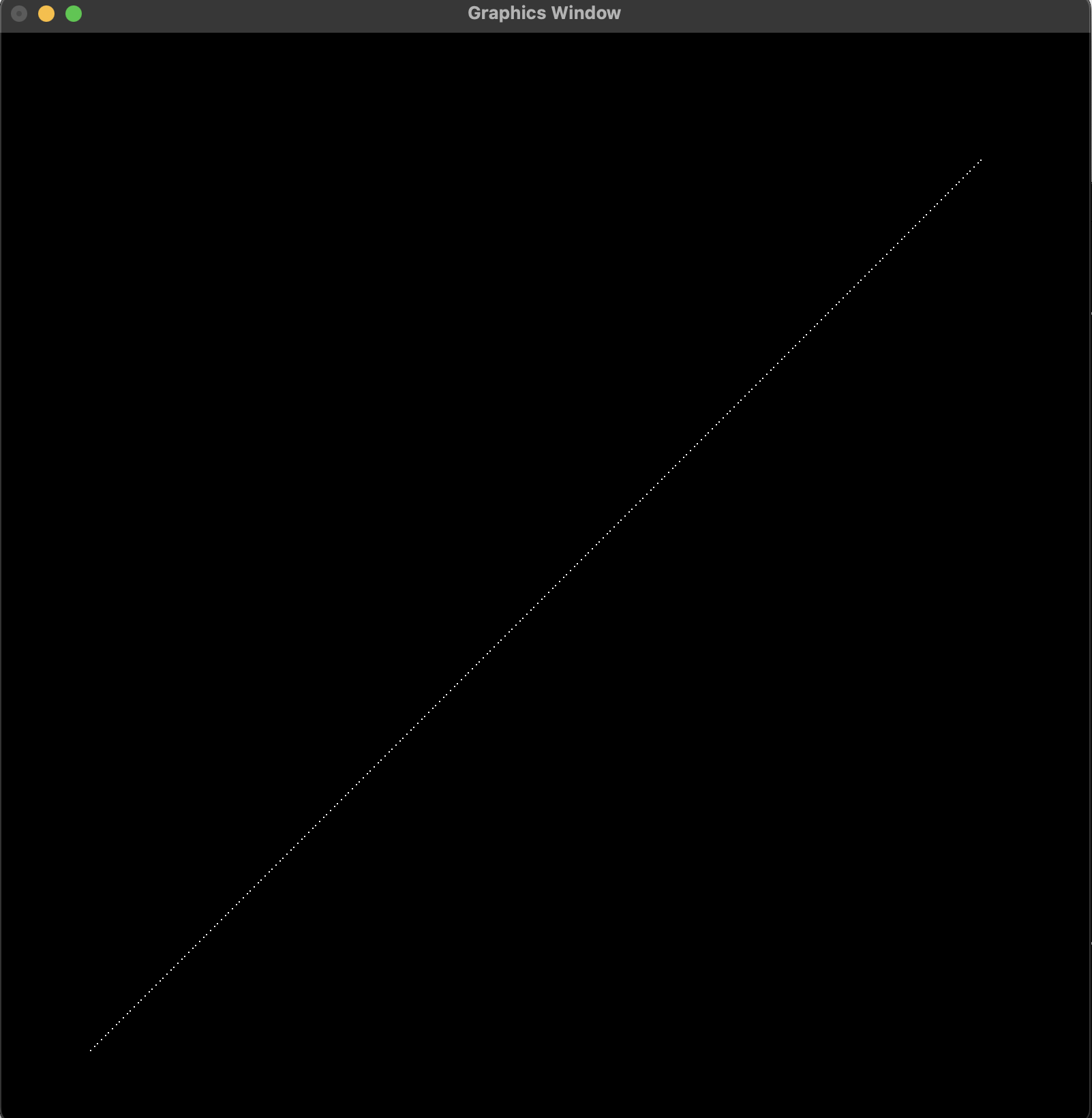
# enter the mainloop

glut.glutMainLoop()

## 3.3 Outputs

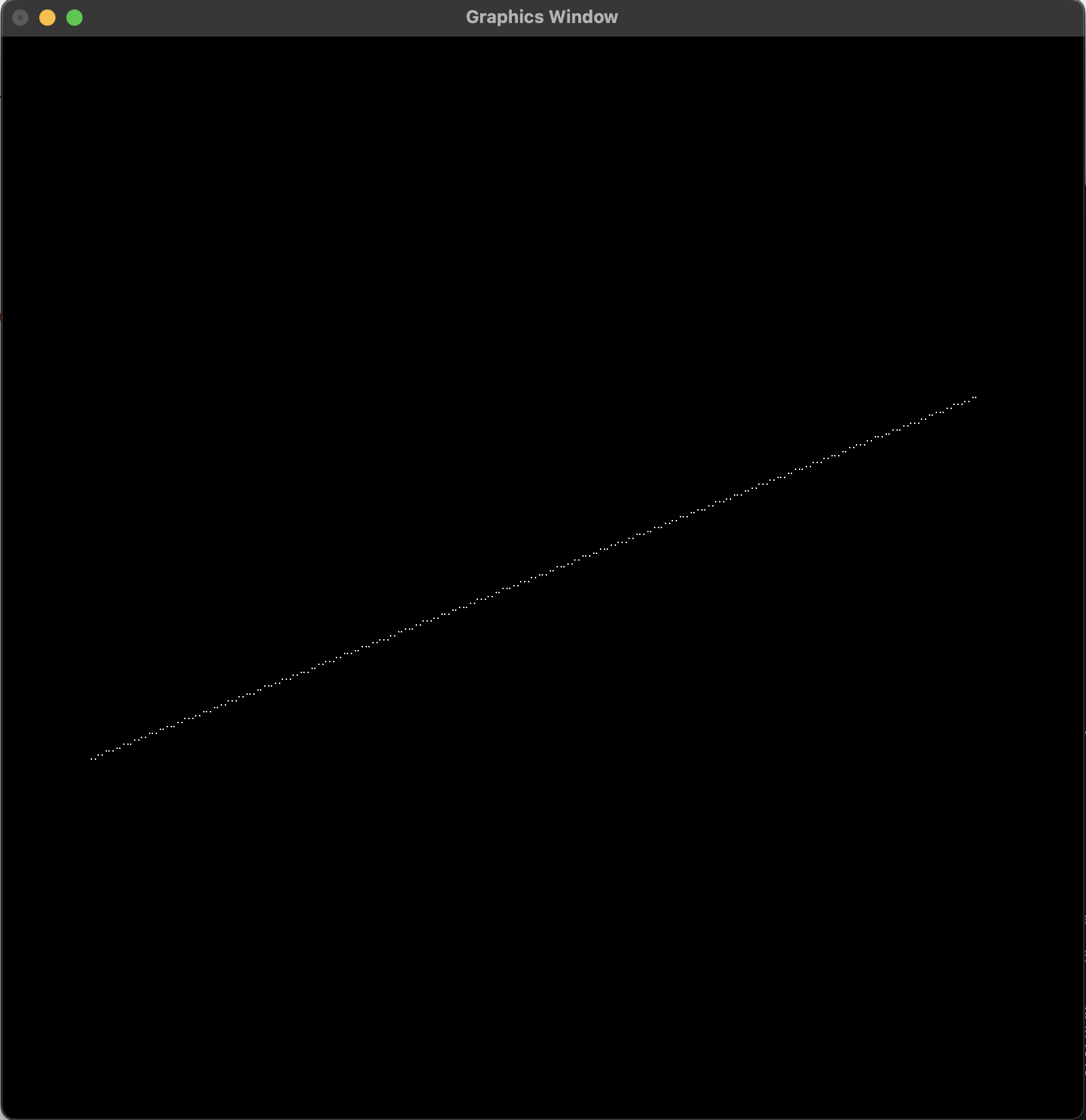
The program takes in the starting and ending points of a line segment from the command line arguments. For example: the command python bresenham.py -125 -125 125 125 is used to initialize the start point as (-125, -125) and end point as (125, 125).

**3.3.1 Slope ( |m| >= 1 )**

****

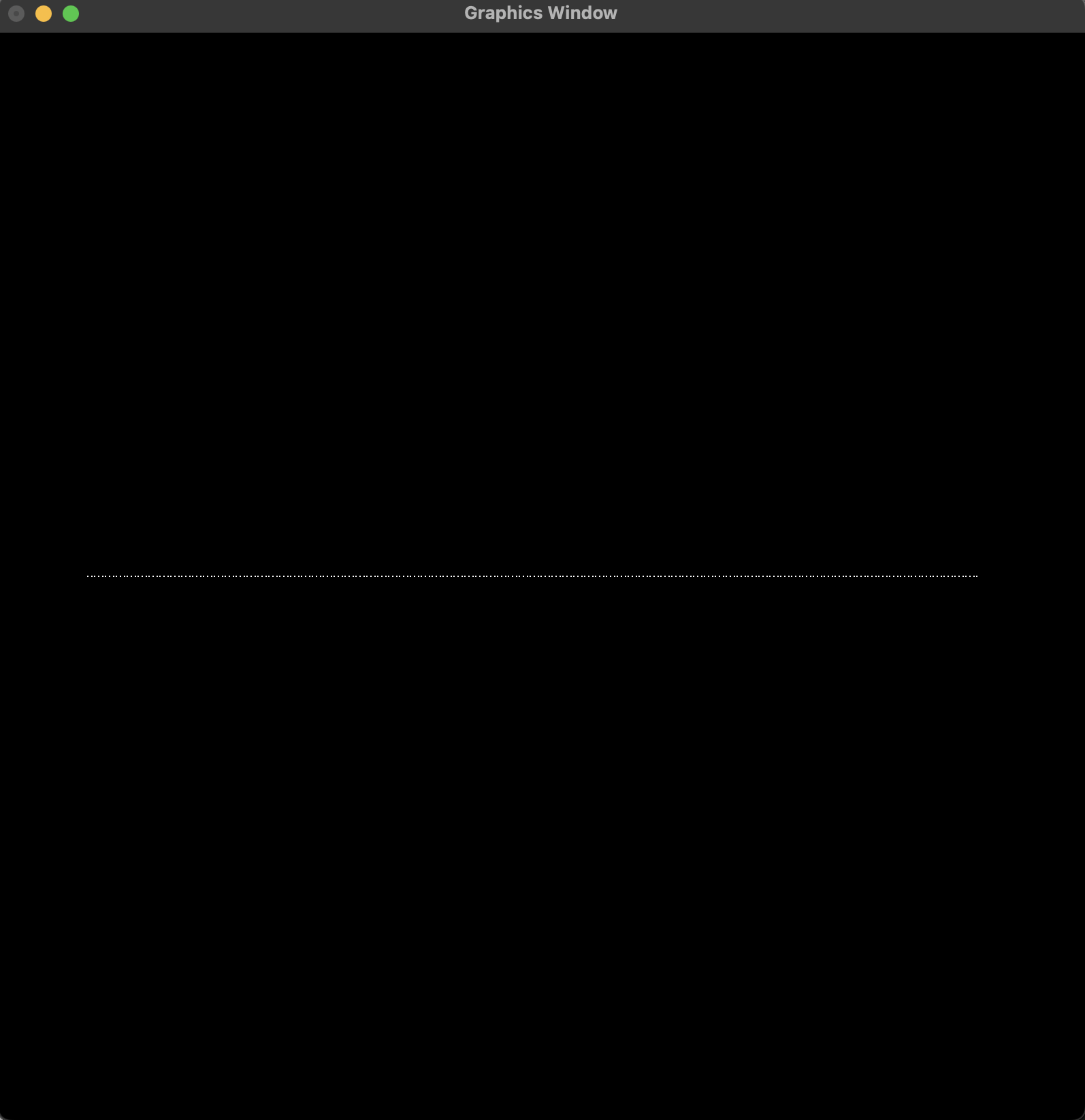
*fig 3.3.1.1:* (-125, -125) to (125, 120)

**3.3.2 Slope ( |m| < 1 )**

****

*fig 3.3.2.1:* (-125, -50) to (125, 50)

**3.3.3 Slope ( |m| = 0 )**

****

*fig 3.3.3.1:* (-125, 0) to (125, 0)

# Chapter 4: Mid-Point Line Drawing Algorithm

## 4.1 Algorithm

The algorithm used to generate the data points between the two end points of a line segment having slopes ( both |m| >= 1 and |m| < 1) using Mid-Point Line Drawing Algorithm is as follows:

1. Take Start Point (x, y) and End Point (x, y) as inputs.
2. Swap points if endPoint(x) < startPoint(x).
3. Calculate the following values using these points:
   1. Del X = endPoint(x) – startPoint(x)
   2. Del Y = endPoint(y) – startPoint(y)
   3. Slope = Del Y/ Del X
4. Perform the following Check:
   1. If | Slope | < 1 : set Check = False and Pk = Del Y – (Del X / 2)
   2. Else : set Check = True and Pk = Del X – (Del Y / 2)
5. Set xValue to startPoint(x) and yValue to startPoint(y)
6. At each (xk, yk) along the line, starting at k = 0:
7. Store (xValue , yValue)
8. Perform the following:
   1. If Check is true : yValue = yValue + 1 and Perform another test:
      1. If Pk < 0 : Update Pk = Pk + Del X
      2. Else : Update Pk = Pk + Del X – Del Y and xValue = xValue + 1
   2. Else : xValue = xValue + 1 and Perform another test:
      1. If Pk < 0 : Update Pk = Pk + Del Y
      2. Else : Update Pk = Pk + Del Y – Del X and yValue = yValue + 1
9. Repeat Step 7 onwards until:

[yValue > endPoint(y) if Check is True] or [xValue > endPoint(x) if Check is False]

## 4.2 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;

}

"""

# 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 --

def midpointAlgo():

data = []

if len(sys.argv) == 5:

startPoint = [int(sys.argv[1]), int(sys.argv[2])]

endPoint = [int(sys.argv[3]), int(sys.argv[4])]

if(startPoint[0] > endPoint[0]) or (startPoint[1] > endPoint[1]):

pointStore = startPoint

startPoint = endPoint

endPoint = pointStore

xValue = startPoint[0]

yValue = startPoint[1]

delX = endPoint[0] - startPoint[0]

delY = endPoint[1] - startPoint[1]

if abs(delX) > abs(delY):

check = False

Pk = delY - (delX/2)

else:

check = True

Pk = delX - (delY/2)

while((yValue <= endPoint[1])) if (check) else (xValue <= endPoint[0]):

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

if check:

yValue = yValue + 1

if Pk < 0:

Pk = Pk + delX

else:

Pk = Pk + delX - delY

xValue = xValue + 1

else:

xValue = xValue + 1

if Pk < 0:

Pk = Pk + delY

else:

Pk = Pk + delY - delX

yValue = yValue + 1

else:

raise Exception("Arguments do not match. Correctly enter the Starting Point and the Ending Point")

return data

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)

# 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 = midpointAlgo()

data = tonormalized(data, [300,300])

initialize()

glut.glutDisplayFunc(display)

glut.glutPostRedisplay()

glut.glutKeyboardFunc(keyboard)

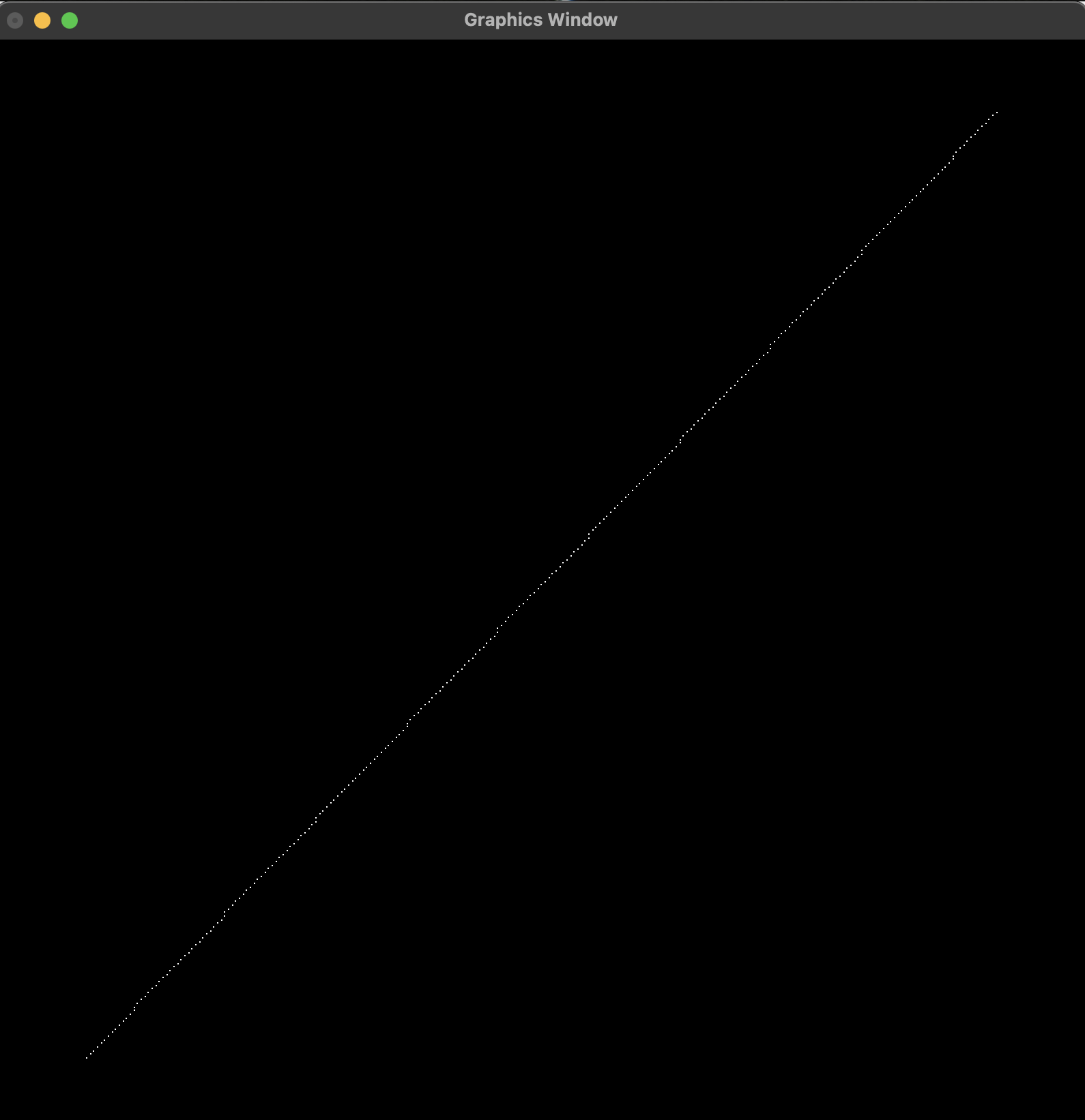
# enter the mainloop

glut.glutMainLoop()

## 4.3 Outputs

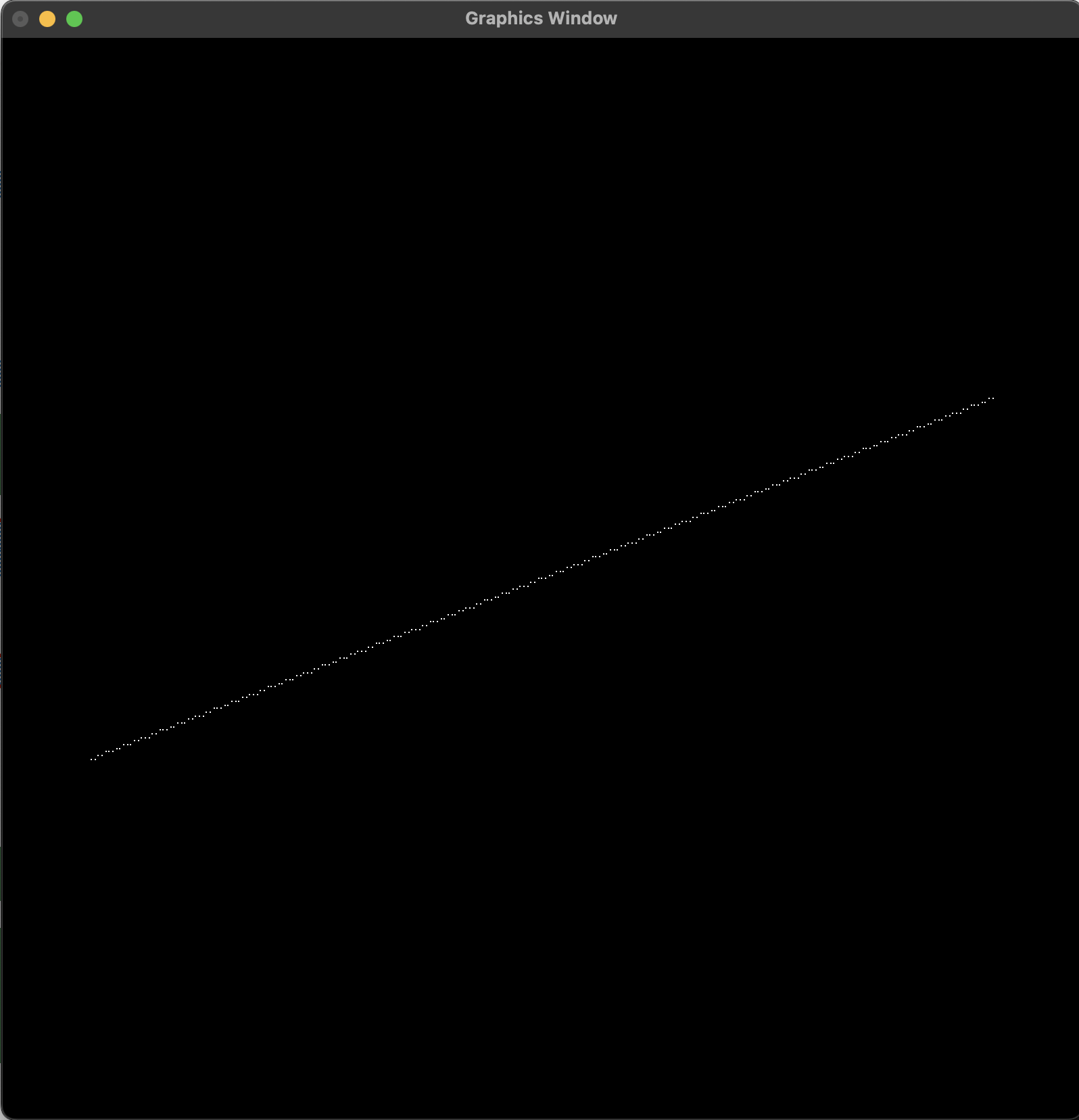
The program takes in the starting and ending points of a line segment from the command line arguments. For example: the command python midpoint.py -125 -125 125 125 is used to initialize the start point as (-125, -125) and end point as (125, 125).

**4.3.1 Slope ( |m| >= 1 )**

****

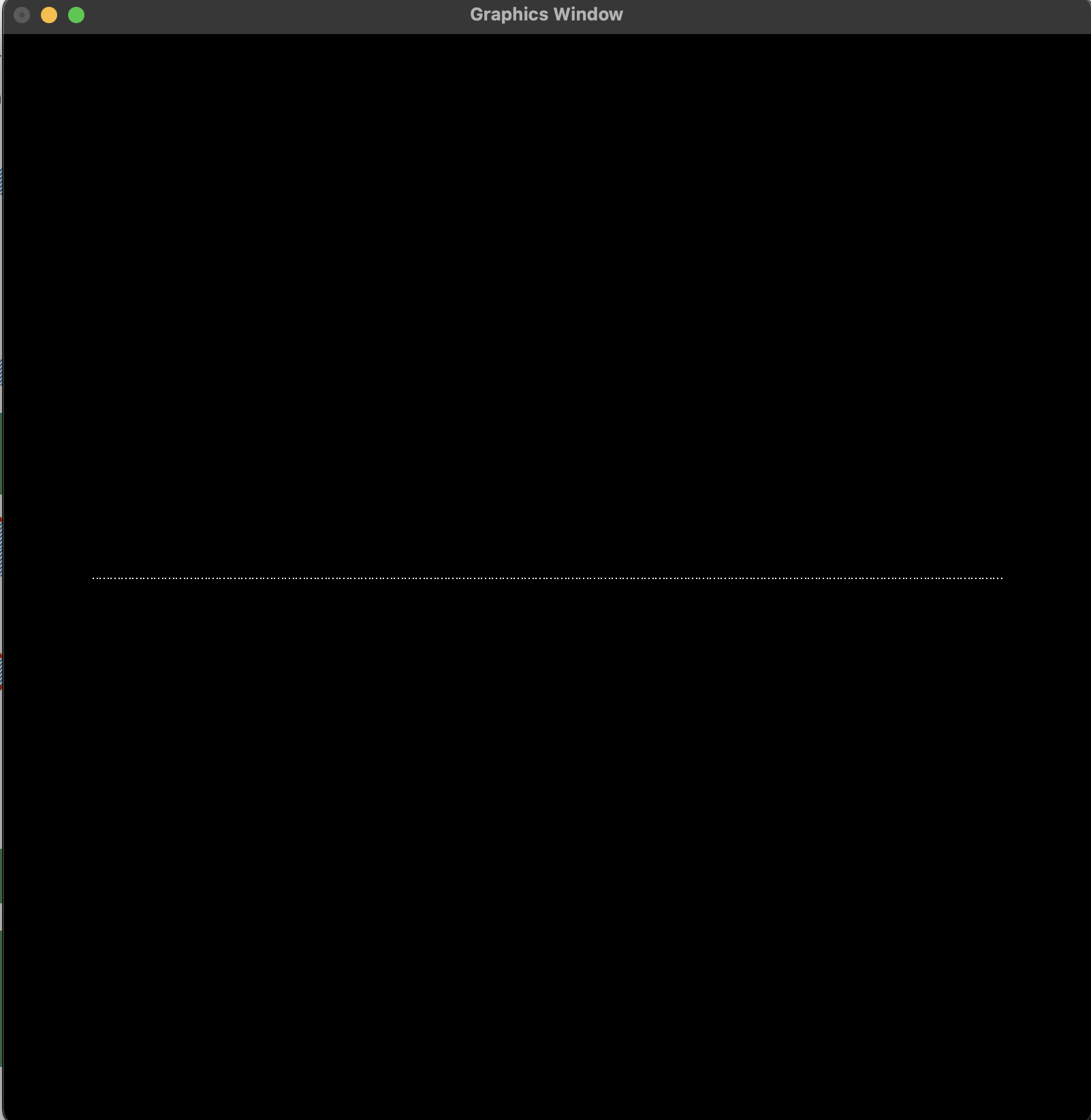
*fig 4.3.1.1:* (-125, -130) to (125, 130)

**4.3.2 Slope ( |m| < 1 )**

****

*fig 4.3.2.1:* (-125, -50) to (125, 50)

**4.3.3 Slope ( |m| = 0 )**

****

*fig 4.3.3.1:* (-125, 0) to (125, 0)

# Chapter 5: Conclusion

Through this Lab Work, I was able to study the details of various popular line drawing algorithms whilst also recognizing the need to identify and carefully pick between the given pixel choices in order to draw a simple connected line segment. 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 these three line drawing algorithms. As known, the line drawing algorithms focuses on creation of lines through illumination of individual pixels, or in our case individual points. So, using the gl.GL\_LINES to draw a line itself would defeat the purpose of using these algorithms.

Moreover, as seen in the output of the various line drawing algorithms if denser points were to be identified by increasing the resolution of the display during the normalization phase we would observe a line appears 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 plotting a line is nothing but plotting a set of closely identified points.