Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre

Lab Report #1

Computer Graphics

[Course Code: COMP 342]

[For the partial fulfillment of 3rd year/2nd Semester in Computer Engineering]

Submitted by:

Sabin Thapa

Roll no. 54

Submitted to:

Mr. Dhiraj Shrestha

Assistant Professor

Department of Computer Science and Engineering

Submission date: April 3, 2022

1. Digital Differential Analyzer Algorithm (DDA)

Introduction

Digital Differential Analyzer is an incremental method of scan conversion of lines where we perform calculations at each step using the results of previous steps. It is called a differential analyzer because it interpolates points based on the difference between the start and end points.

Algorithm

The algorithm for the DDA is as follows:

```
1. Input the two line end-points (x1, y1) and (x2, y2).
2. Calculate:
           dx = x2 - x1
           dy = y2 - y1
3. If |dx| > |dy| then
           stepSize = |dx|
   Else
            stepSize = |dy|
4. x_inc = dx / stepSize         // Calculate increment in x & y for each steps
   y_inc = dy / stepSize
5. x, y = x1, y1                 // Save initial points
   Plot pixel (x,y)
6. Until stepSize do
             x = x + x_inc
             y = y + y_inc
             Plot pixel (Round(x), Round(y)
7. End
```

Source Code

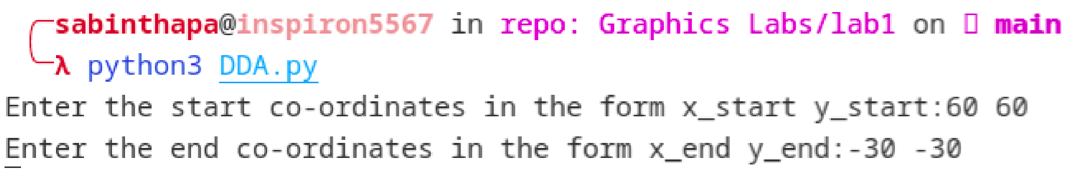
```python
from OpenGL.GL import *

from OpenGL.GLUT import *

from OpenGL.GLU import *

def DDA_Algo(x_start, y_start, x_end, y_end):

   dx = x_end - x_start

   dy = y_end - y_start

   if abs(dx) > abs(dy):

       stepSize = abs(dx)

   else:

       stepSize = abs(dy)

   try:

       x_inc = dx/stepSize

       y_inc = dy/stepSize

   except ZeroDivisionError:

       print("Division by zero!!")

   x = x_start

   y = y_start

   glColor3f(0.0,0.0,1.0) #RGB Color

   glPointSize(4.0) #Point Size

   glBegin(GL_POINTS)
```

```
    for _ in range(stepSize+1):
        glVertex2f(round(x), round(y))
        x += x_inc
        y += y_inc
    glEnd()
    glFlush()
def initialize():
    glutInit()
    glutInitDisplayMode(GLUT_RGBA)
    glutInitWindowSize(600, 600)
    glutInitWindowPosition(400, 400)
    glutCreateWindow("Digital Differential Analyzer Algorithm")
    glClearColor(1.0,1.0,1.0,0.0)
    gluOrtho2D(-100,100,-100,100)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
  #Axes
    glColor3f(0.0,0.0,1.0)
    glPointSize(1.0)
    glBegin(GL_LINES)
    glVertex2f(-100,0)
    glVertex2f(100,0)
    glVertex2f(0,100)
    glVertex2f(0,-100)
```

```
   glEnd()

# driver code

if __name__ ==  "__main__":

    start = input("Enter the start co-ordinates in the form x_start
y_start:").split(' ')

     end  =  input("Enter  the  end  co-ordinates  in  the  form  x_end
y_end:").split(' ')

   x_start,y_start = int(start[0]), int(start[1])

   x_end,y_end = int(end[0]), int(end[1])

   initialize()

   glutDisplayFunc(lambda: DDA_Algo(x_start, y_start, x_end, y_end))

   glutIdleFunc(lambda: DDA_Algo(x_start, y_start, x_end, y_end))

   glutMainLoop()
```

Output

1.



```
╭─sabinthapa@inspiron5567 in repo: Graphics Labs/lab1 on  main
╰─λ python3 DDA.py
Enter the start co-ordinates in the form x_start y_start:60 60
Enter the end co-ordinates in the form x_end y_end:-30 -30
```

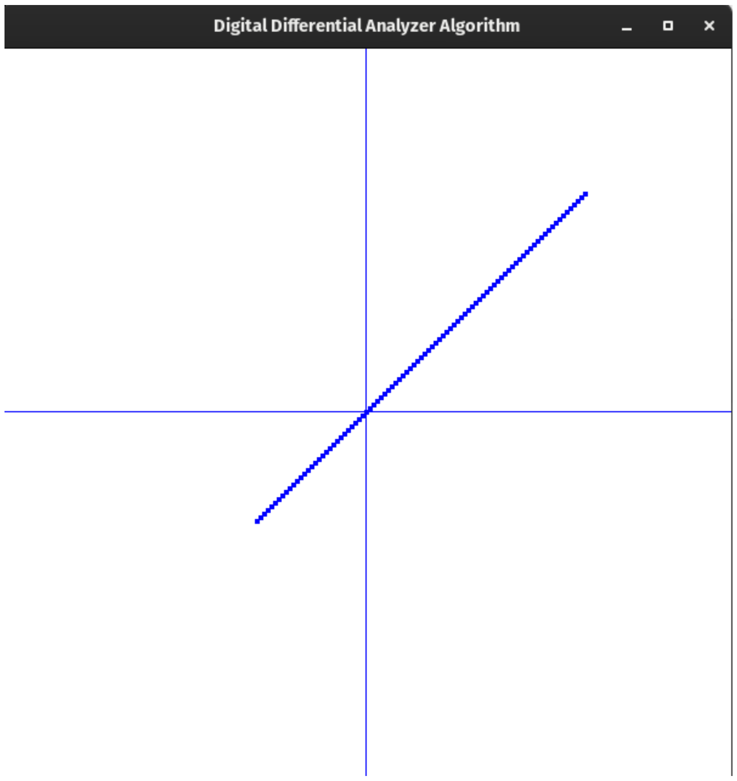
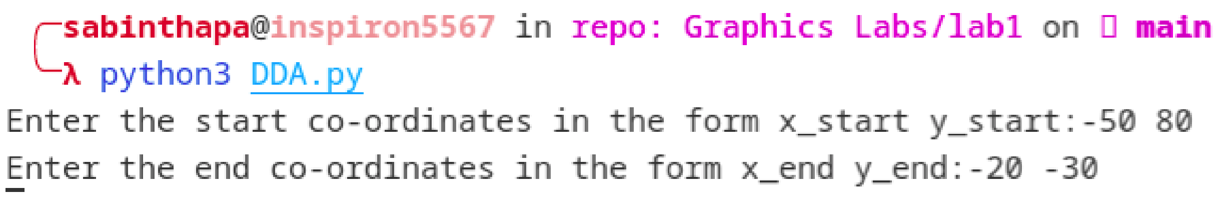


Fig 1.1: DDA Output 1

2.



```
╭─sabinthapa@inspiron5567 in repo: Graphics Labs/lab1 on  main
╰─λ python3 DDA.py
Enter the start co-ordinates in the form x_start y_start:-50 80
Enter the end co-ordinates in the form x_end y_end:-20 -30
```

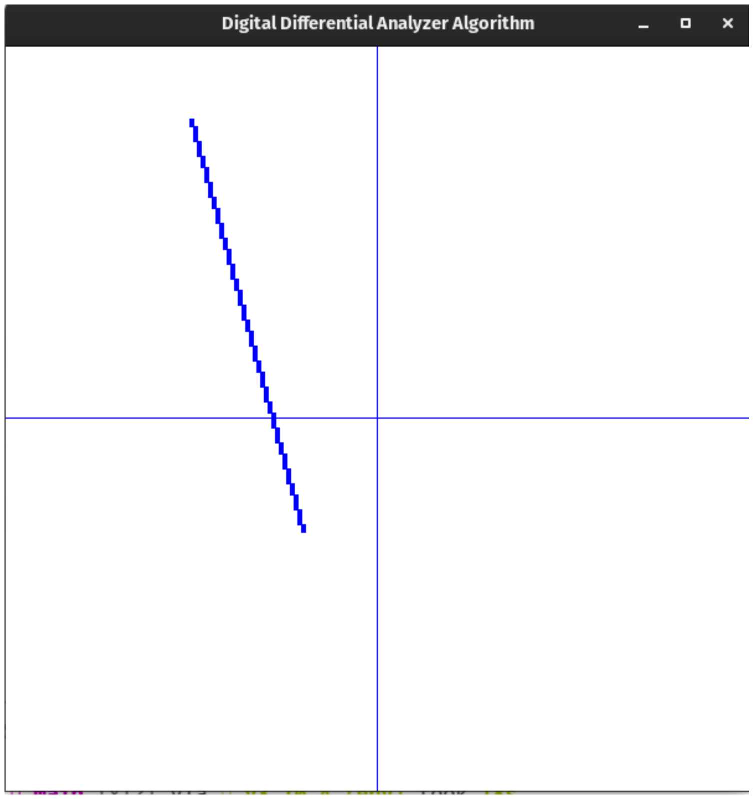
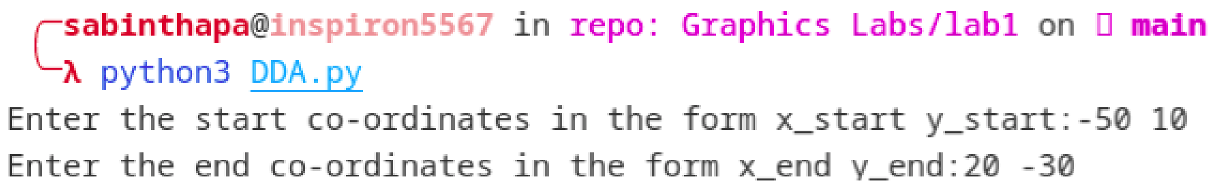


Fig 1.2: DDA Output 2

3.



```
╭─sabinthapa@inspiron5567 in repo: Graphics Labs/lab1 on  main
╰─λ python3 DDA.py
Enter the start co-ordinates in the form x_start y_start:-50 10
Enter the end co-ordinates in the form x_end y_end:20 -30
```

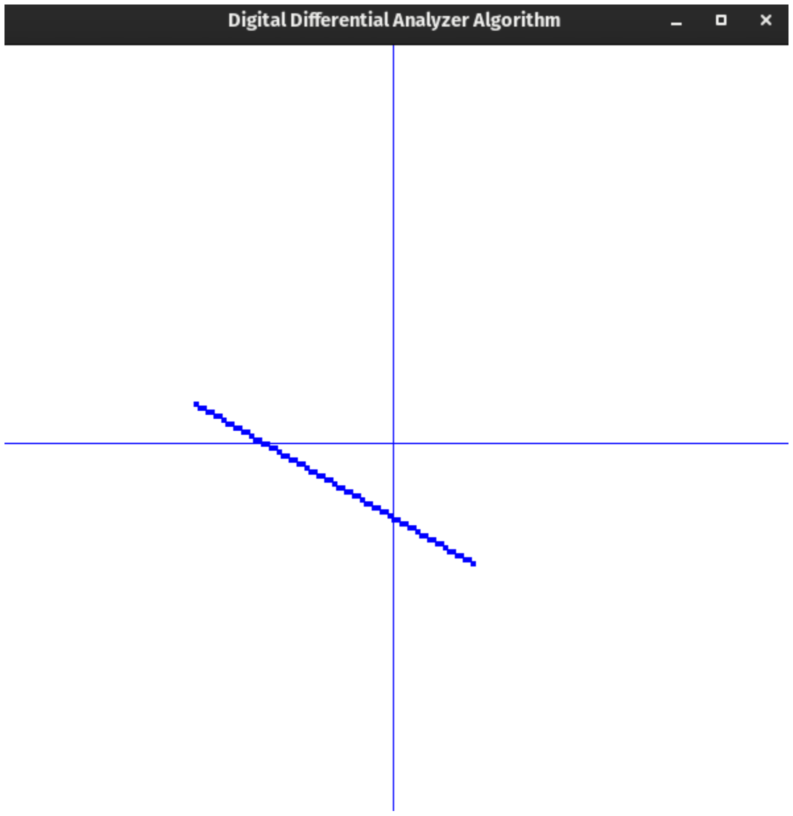
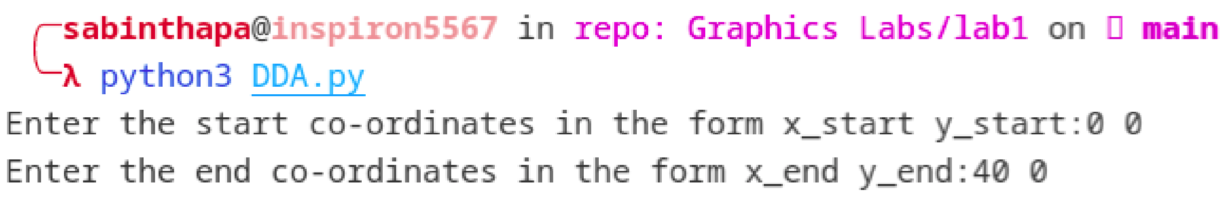


Fig 1.3: DDA Output 3

4.



```
╭─sabinthapa@inspiron5567 in repo: Graphics Labs/lab1 on  main
╰─λ python3 DDA.py
Enter the start co-ordinates in the form x_start y_start:0 0
Enter the end co-ordinates in the form x_end y_end:40 0
```

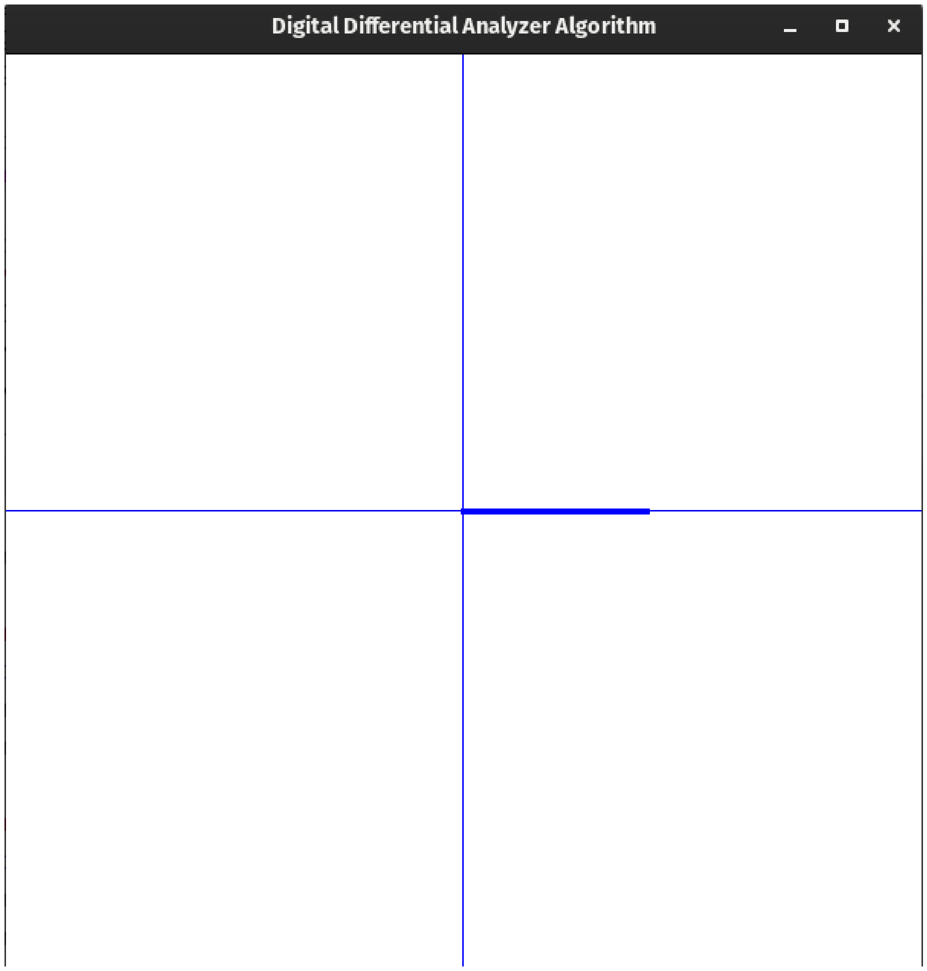


Fig 1.4: DDA Output 4

2. Bresenham Line Algorithm (BLA)

Introduction

Bresenham's line drawing algorithm is an efficient algorithm over the DDA, as it involves only integer calculations. As the integer calculations can be performed very rapidly, the line can be generated quickly as well. We select the next pixel which is closer to the scan line.

Algorithm

1. Given two endpoints (x1, y1) and (x2,y2) as start and end points.
2. Set x = x1, y = y1.
3. Plot pixel (x,y).
4. Calculate the constants

    dx = x2 - x1

    dy = y2 - y1

    m = dy/dx, m is the slope of the line

    2dx-2dy and 2dy-2dx

5. Obtain the value of decision parameter as:

    p_k = 2 * dy-dx if |m| < 1

    p_k= 2 * dx-dy if |m| >=1

6. At each point along the line, starting at k=0, perform the following tests:

```
If |m| < 1:
    If $p_k$>= 0:
        Plot($x_{k+1}$, $y_k + 1$) if y1<y2  else
        Plot($x_{k+1}$, $y_k - 1$)
        $p_{k+1} = p_k + 2dy - 2dx$ if y1 < y2 else
```

```
            $p_{k+1} = p_k - 2dy - 2dx$

        Else if $p_k < 0$:

            Plot($x_{k+1}, y_k$)

            $p_{k+1} = p_k + 2dy$ if y1 < y2 else

            $p_{k+1} = p_k - 2dy$

    Else if |m| >= 1:

        If $p_k >= 0$:

            Plot($x_k + 1, y_k$) if x1 < x2  else

            Plot($x_k - 1, y_k$)

            $p_{k+1} = p_k + 2dx - 2dy$ if x1 < x2 else

            $p_{k+1} = p_k - 2dx - 2dy$

        Else if $p_k < 0$:

            Plot($x_k, y_k + 1$)

            $p_{k+1} = p_k + 2dx$ if x1 < x2 else

            $p_{k+1} = p_k - 2dx$
```

7. Repeat step 6 for:
    a. dx times if |m| < 1
    b. dy times if |m| >= 1
8. End

Source Code

```
from OpenGL.GL import *

from OpenGL.GLUT import *

from OpenGL.GLU import *


def Bresenham_Algo(x_start, y_start, x_end, y_end):

   dx = x_end - x_start

   dy = y_end - y_start

   x = x_start

   y = y_start


   if abs(dx) > abs(dy) and x_end < x_start:

       dx, dy = -dx, -dy

       x_end, x_start  = x_start, x_end

       y_end, y_start = y_start, y_end


   elif abs(dx) <= abs(dy) and y_end <  y_start:

       dy, dx = -dy, -dx

       y_end, y_start = y_start, y_end

       x_end, x_start = x_start, x_end


   glPointSize(4.0) #Point Size

   glBegin(GL_POINTS)
```

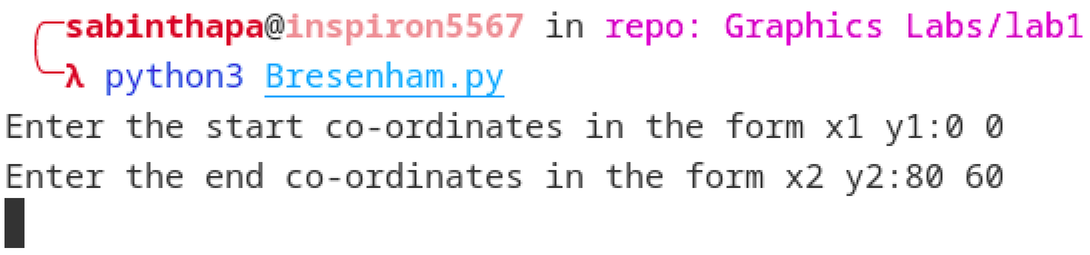
```
'''RGB COLOR'''

   glColor3f(0.0,0.0,1.0)

   '''Starting Point'''

   glVertex2f(x, y)


   ''' Case: |slope| < 1'''

   if abs(dx) > abs(dy):

       p = 2*dy-dx

       for i in range(0, abs(dx)+1):

           x+=1

           if(p >= 0):

               y = y+1 if y_start < y_end else y-1

               glVertex2f(x, y)

               p = dec+2*dy-2*dx if y_start < y_end else p-2*dy-2*dx

           else:

               glVertex2f(x,y)

               p = dec+2*dy if y_start < y_end else p-2*dy


   #Case: |slope| > 1

   else:

       p=2*dx-dy

       for i in range(0, abs(dy)+1):

           y+=1
```

```
            if (p>=0):

                x = x+1 if x_start < x_end else x-1

                glVertex2f(x, y)

                p = p+2*dx-2*dy if x_start < x_end else p-2*dx-2*dy

            else:

                glVertex2f(x, y)

                p = p+2*dx if x_start < x_end else p-2*dx

        glEnd()


    glFlush()


def initialize():

    glutInit()

    glutInitDisplayMode(GLUT_RGBA)

    glutInitWindowSize(600, 600)

    glutInitWindowPosition(300, 300)

    glutCreateWindow("Bresenham Line Drawing Algorithm")

    glClearColor(1.0,1.0,1.0,0.0)

    gluOrtho2D(-100,100,-100,100)

      glClear(GL_COLOR_BUFFER_BIT  |  GL_DEPTH_BUFFER_BIT)  #to  clear
everything drawn previously


    #Axes

    glColor3f(0.0,0.0,1.0)
```

```
   glPointSize(1.0)

   glBegin(GL_LINES)

   glVertex2f(-100,0)

   glVertex2f(100,0)

   glVertex2f(0,100)

   glVertex2f(0,-100)

   glEnd()




if __name__ == "__main__":

     start = input("Enter the start co-ordinates in the form x1 y1:").split(' ')

      end = input("Enter the end co-ordinates in the form x2 y2:").split(' ')

   x1,y1 = int(start[0]), int(start[1])

   x2,y2 = int(end[0]), int(end[1])

   initialize()

   glutDisplayFunc(lambda: Bresenham_Algo(x1, y1, x2, y2))

   glutIdleFunc(lambda: Bresenham_Algo(x1, y1, x2, y2))

   glutMainLoop()
```

Output

1. Output 1 for |m| < 1



```
┌─sabinthapa@inspiron5567 in repo: Graphics Labs/lab1
└─λ python3 Bresenham.py
Enter the start co-ordinates in the form x1 y1:0 0
Enter the end co-ordinates in the form x2 y2:80 60
█
```

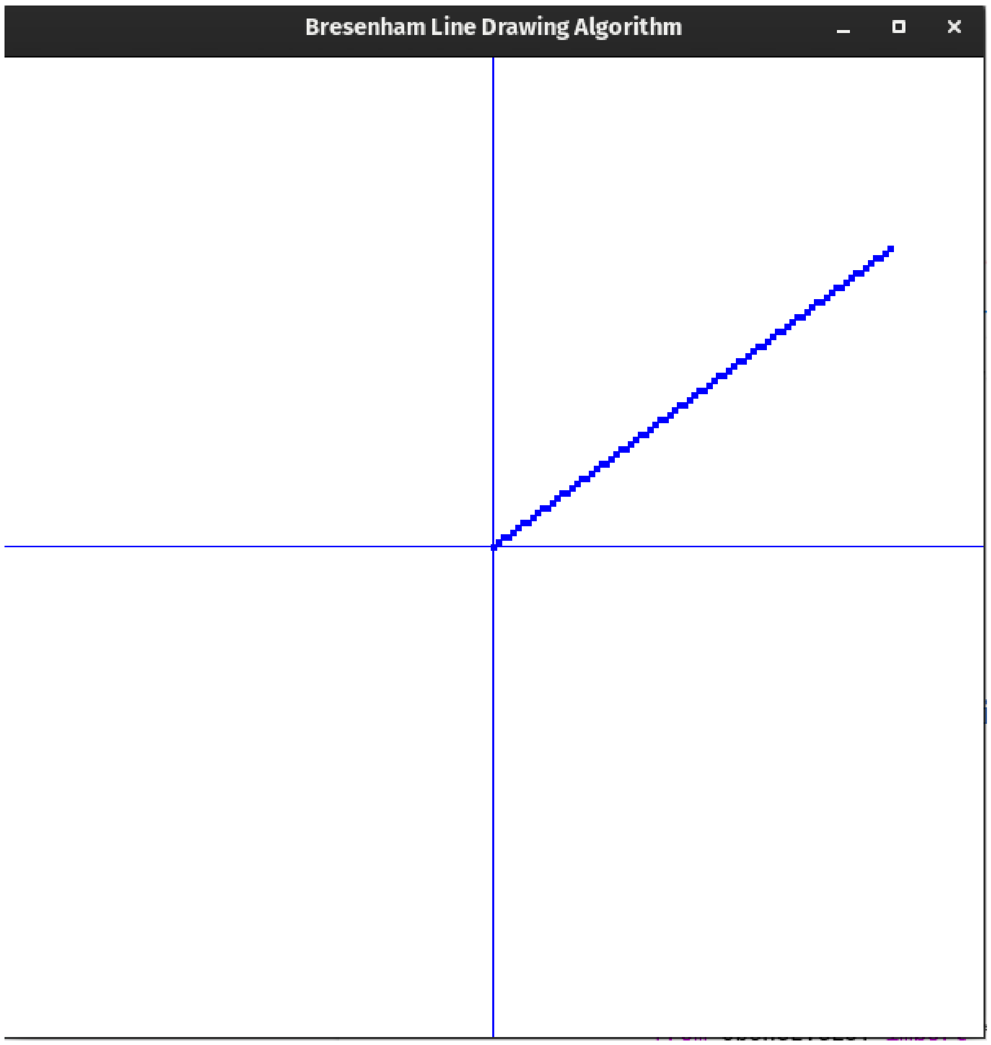
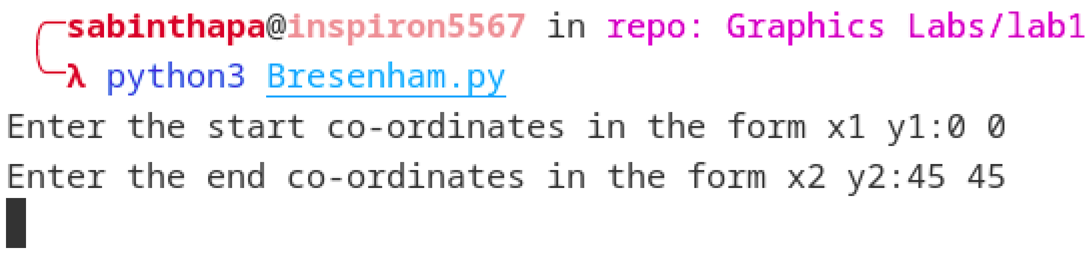


Fig 2.1: Bresenham Algorithm with slope < 1

2. Output 2 for |m| >=1



```
┌─sabinthapa@inspiron5567 in repo: Graphics Labs/lab1
└─λ python3 Bresenham.py
Enter the start co-ordinates in the form x1 y1:0 0
Enter the end co-ordinates in the form x2 y2:45 45
```

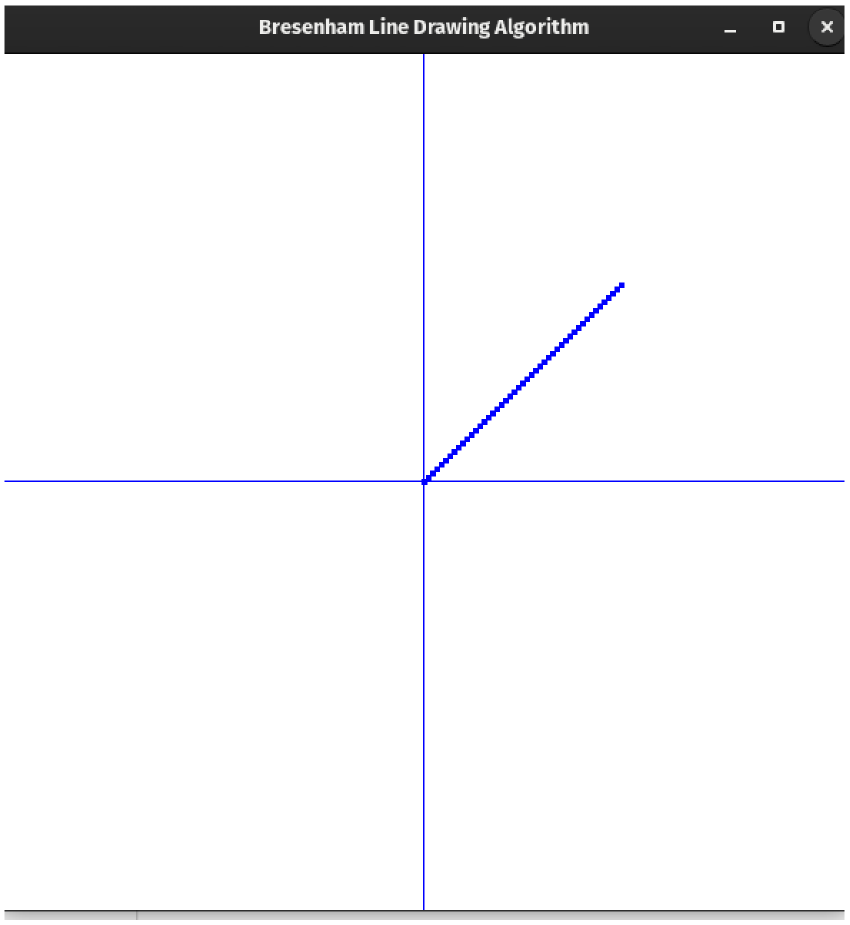
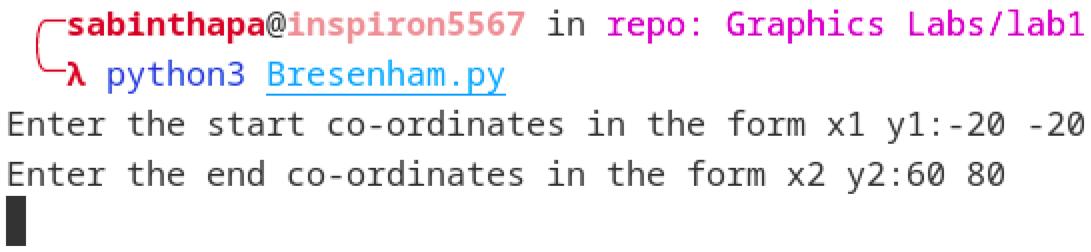


Fig 2.1: Bresenham Algorithm with slope = 1



```
╭─sabinthapa@inspiron5567 in repo: Graphics Labs/lab1
╰─λ python3 Bresenham.py
Enter the start co-ordinates in the form x1 y1:-20 -20
Enter the end co-ordinates in the form x2 y2:60 80
```

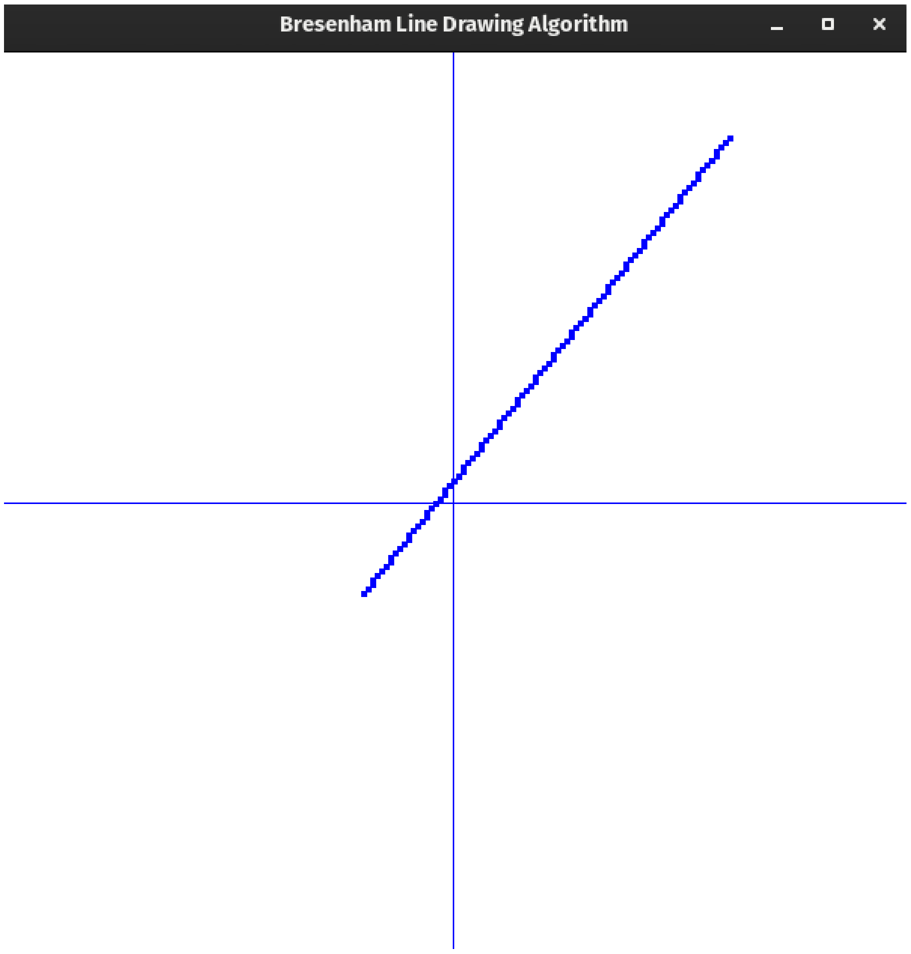


Fig 2.2: Bresenham Algorithm with slope > 1

3. Midpoint Circle Drawing Algorithm

Introduction

It is an algorithm used to determine the points needed to rasterize a circle. We use the algorithm to calculate the perimeter points of the circle on the first octant and use the 8 point symmetry of the circle to calculate other points in other octants.

Algorithm

```
1. Given the radius r and circle center ($x_c$, $y_c$).

2. Set x= 0 and y =r.

3. Calculate the initial decision parameter as:

    p0 = 1 - r if r is an integer

    p0 = 5/4 - r, if r is a floating point

4. At each $x_k$ position, starting at k=0 perform the following test:

    a. If $p_k < 0$:

        i. $P_{k+1} = p_k + 2x_{k+1} +1$

        ii. plot( $x_k$+1 +$x_c$ , $y_k$ + $y_c$ )

    b. Else:

        i. $P_{k+1} = p_k + 2x_{k+1} +1 - 2y_{k+1}$ , where $y_{k+1} = y_k - 1$ & $x_{k+1} = x_k + 1$

        ii. plot( $x_k$+1 +$x_c$ , $y_k$ - 1+ $y_c$ )

5. Define symmetry points in other 7 octants as:

    (x,y) ->(x,-y), (-x,y), ( -x,-y), (y,x), (-y,x), (y,-x), (-y,-x),

6. Repeat steps 4-5 until x>y which is the stopping condition.

7. Stop
```

Source Code

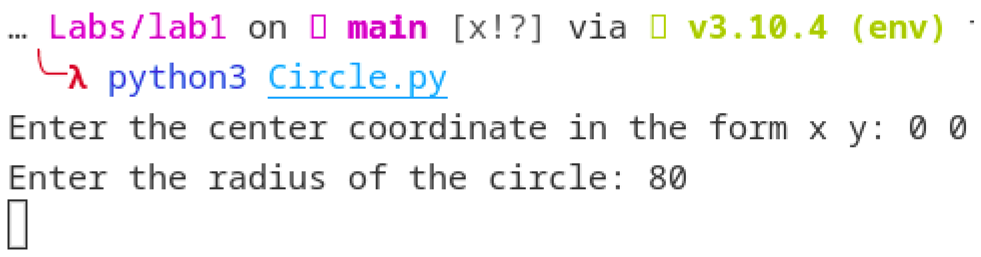
```
from OpenGL.GL import *

from OpenGL.GLUT import *

from OpenGL.GLU import *


'''Stopping Criteria'''

def stop_criteria(x,y):

   return x>y


#plots all the 8 symmetric points for a circle's point

def plot_symmetric_pixels(x,y,x_center,y_center):

   glColor3f(0.0,1.0,1.0)

   glPointSize(4.0)

   glBegin(GL_POINTS)


   glVertex2f(x+x_center, y+y_center)

   glVertex2f(-x+x_center, y+y_center)

   glVertex2f(x+x_center, -y+y_center)

   glVertex2f(-x+x_center, -y+y_center)

   glVertex2f(y+x_center, x+y_center)

   glVertex2f(y+x_center, -x+y_center)

   glVertex2f(-y+x_center, x+y_center)

   glVertex2f(-y+x_center, -x+y_center)
```

```
    glEnd()


def Circle_Algo(x_center, y_center, r):
    x = 0
    y = r
      pk = 1-r if isinstance(r,int) else 5/4-r  #decision
parameter

    while not stop_criteria(x, y):
        x=x+1
        if pk<0:
            pk = pk + 2*x + 1
            plot_symmetric_pixels(x,y, x_center, y_center)
        else:
            y=y-1
            pk = pk + 2*x - 2*y + 1
            plot_symmetric_pixels(x,y, x_center, y_center)


    glFlush()


def initialize():
    glutInit()
    glutInitDisplayMode(GLUT_RGBA)
    glutInitWindowSize(600, 600)
    glutInitWindowPosition(300, 300)

    glutCreateWindow("Midpoint Circle Drawing Algorithm")



    glClearColor(1.0,1.0,1.0,0.0)

    gluOrtho2D(-200,200,-200,200)



     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) #clears
everything previously drawn



    glColor3f(0.0,0.0,1.0) #sets RGB color

    glPointSize(1.0) #sets point size



    glBegin(GL_LINES)

    glVertex2f(-200,0)

    glVertex2f(200,0)

    glVertex2f(0,200)

    glVertex2f(0,-200)

    glEnd()






if __name__ == "__main__":

     center = input("Enter the center coordinate in the form x
y: ").split(' ')
```

```
    radius = float(input("Enter the radius of the circle: "))

    x_center,y_center = int(center[0]), int(center[1])



    initialize()



                                              glutDisplayFunc(lambda:
Circle_Algo(x_center,y_center,radius))

    glutIdleFunc(lambda: Circle_Algo(x_center,y_center,radius))



    glutMainLoop()
```

Output

1. Circle centered at origin with radius 80:



```
… Labs/lab1 on  main [x!?] via  v3.10.4 (env)
  ╰─λ python3 Circle.py
Enter the center coordinate in the form x y: 0 0
Enter the radius of the circle: 80
▯
```

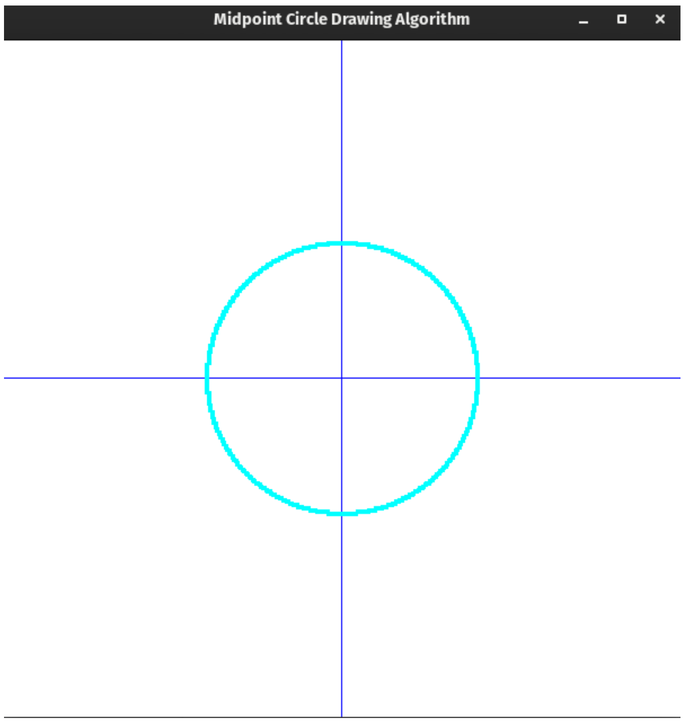
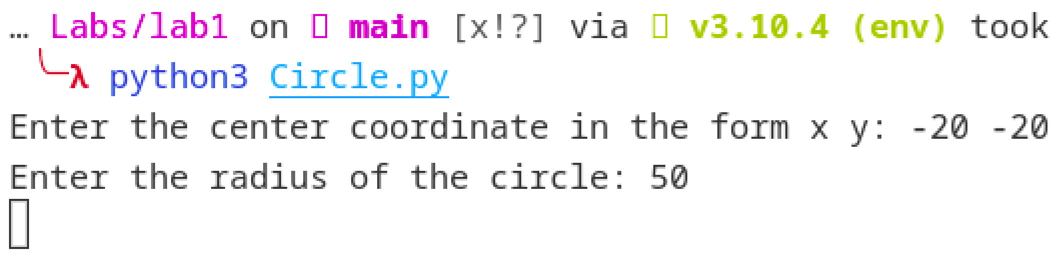


Fig 3.1: Circle centered at origin

2. Circle centered at (-20, -20) with radius 50



```
… Labs/lab1 on □ main [x!?] via □ v3.10.4 (env) took
 ╰─λ python3 Circle.py
Enter the center coordinate in the form x y: -20 -20
Enter the radius of the circle: 50
□
```

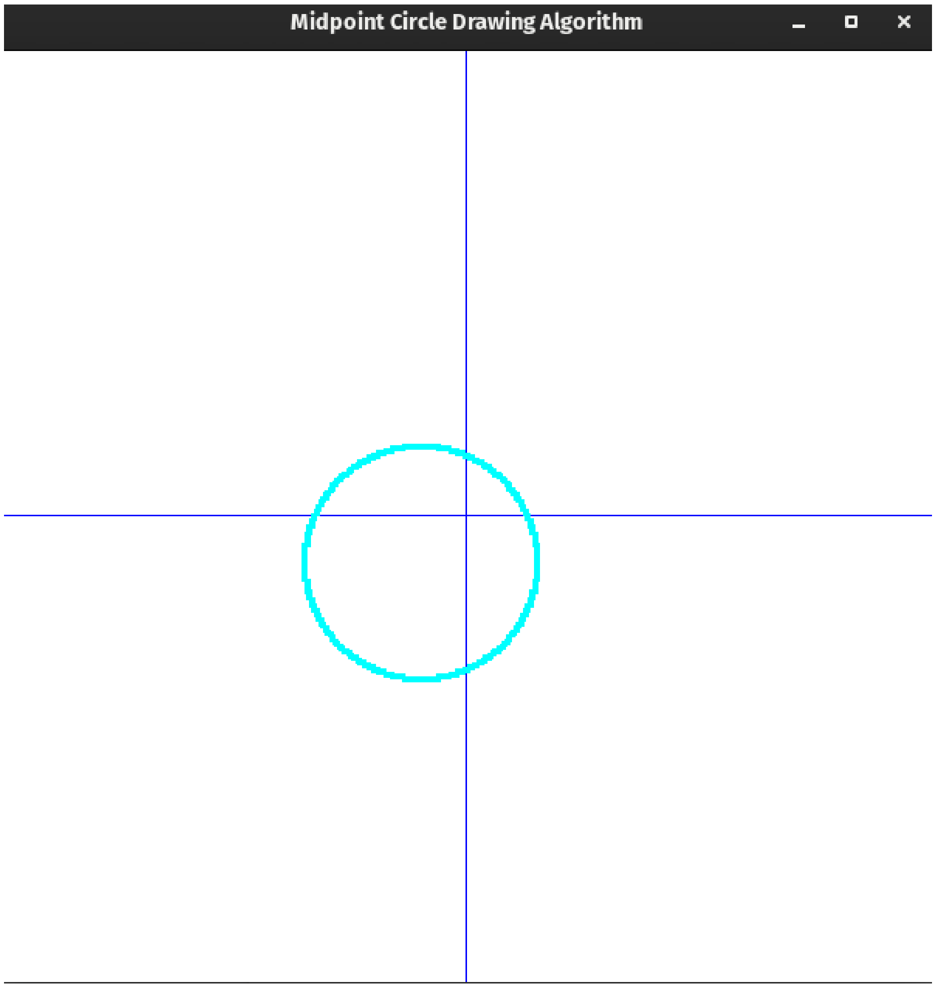


Fig 3.2: Circle centered at (-20, -20)

4. Pie-Chart using OpenGL functions

Introduction

Pie-Chart is a graph that represents data in a circular graph. It can be generated using the algorithms discussed above: the midpoint circle generating algorithm can be used to generate the circle of the pie-chart and we can use a line drawing algorithm to create slices of the pie chart.

Algorithm

1. Start
2. Accept the values of theta such that the sum is less than 360.
3. Plot the outer circle of the pie-chart using mid-point circle algorithm with:

   $(x_c\ ,y_c)$ as the centers of the circle with radius r.

4. Starting with initial angle 0, draw the line starting from the center to the circumference of the circle (0, r).
5. For each user input, theta, find the points on the circumference as:

   x = x cos(theta) - y*sin(theta)

   y = xsin(theta) + y cost(theta)

6. For each coordinate obtained from equation (5), draw lines from the center of the circle connecting to the coordinate point (x,y) at an interval of 0.1 until the first user input angle theta is reached. Generate a random color for the lines to fill each slice in the pie chart.
7. Update the starting angle theta to the first user input and repeat steps 5 and 6.
8. End.

Source Code

```
from Circle import Circle_Algo

from OpenGL.GL import *

from OpenGL.GLUT import *

from OpenGL.GLU import *

import math

import random


def plot_lines(x1, y1, x2, y2):

   glLineWidth(3.0)

   glBegin(GL_LINES)

   glVertex2f(x1,y1)

   glVertex2f(x2,y2)

   glEnd()


def convert_to_radian(angle_in_degrees):

   return - math.pi /180 * angle_in_degrees


def draw_pie_chart(theta_values):

   '''Using Midpoint Circle Algo to draw the Circle'''

   Circle_Algo(0, 0, 150)

   x1 = 0

   y1 = 150
```

```
    angle = 0

    # Plot lines for each theta in the input

    for i, theta in enumerate(theta_values):

        #Plot lines from previous angle to the next angle

        for j in range(angle, angle+theta):

            k=0

            # Plot lines at every 0.1 interval

            while k<1:

                radian_val = convert_to_radian(j+k)

        # Use the degree to find the coordinates on the circle

                x  =  round(x1*  math.cos(radian_val)  -  y1*
math.sin(radian_val))

                 y  =  round(x1*  math.sin(radian_val)  +  y1*
math.cos(radian_val))

    # Generate random RGB color to fill the pie chart sections

                r = ((i+1)/4)%2

                g = ((i+1)/2)%2

                b = (i+1)%2

                glColor3f(r,g,b)

                plot_lines(0,0,x,y)

                k+=0.1


        angle += theta
```

```
def initialize():

   glutInit()

   glutInitDisplayMode(GLUT_RGBA)

   glutInitWindowSize(600, 600)

   glutInitWindowPosition(300, 300)

   glutCreateWindow("PieChart")

   glClearColor(1.0,1.0,1.0,0.0)

   gluOrtho2D(-200,200,-200,200)

   glPointSize(4.0)

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

if __name__ == "__main__":

   accept_input = True

   theta_values = []

      '''Take theta angles as inputs to create a pie chart'''

   while(accept_input):

      inputs = input("Enter the values of theta degrees in pie
      charts separated by space: ").split(' ')

       for inp in inputs:

           theta_values.append(int(inp))

       #Stop if the theta values exceed 360

       if sum(theta_values) > 360:

           accept_input = False

           print("Sum of angles exceeded 360 degrees.")
```

```
else:

    initialize()

    glutDisplayFunc(lambda: draw_pie_chart(theta_values))

    glutIdleFunc(lambda: draw_pie_chart(theta_values))

    glutMainLoop()
```

Output

1. Pie Chart with four 90 degree slices.

```
…s Labs/lab1 on  main [x!?] via  v3.10.4 (env) took 17s
✦ ╰─λ python3 PieChart.py
Enter the values of theta degrees in pie charts separated
 by space: 90 90 90 90
▯
```

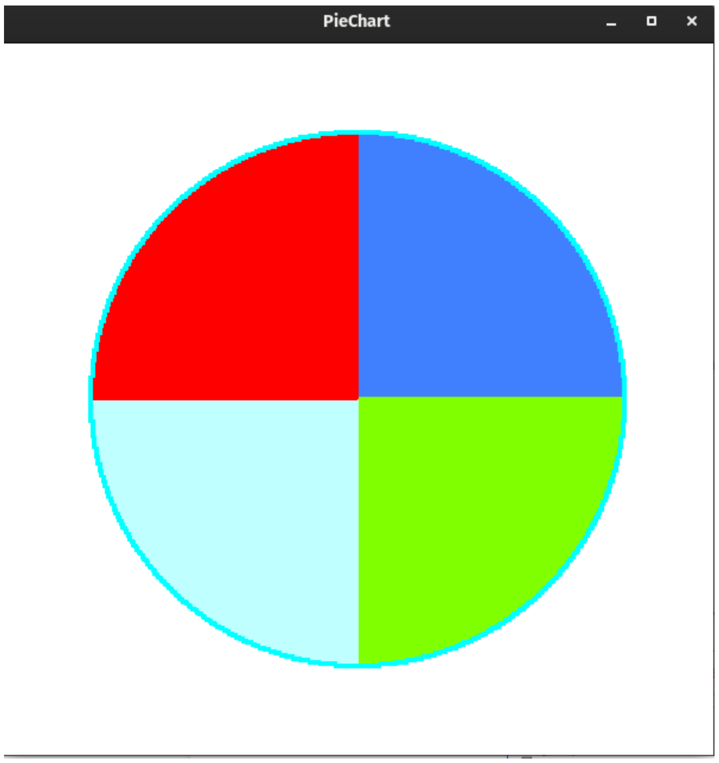
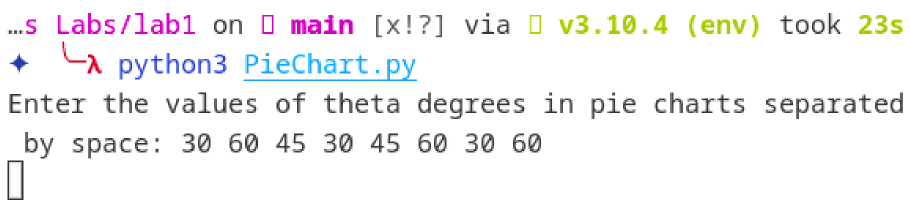


Fig 4.1: Pie Chart 1

2. Pie Chart with random theta inputs.



```
…s Labs/lab1 on  main [x!?] via  v3.10.4 (env) took 23s
✦ ╰─λ python3 PieChart.py
Enter the values of theta degrees in pie charts separated
 by space: 30 60 45 30 45 60 30 60
```

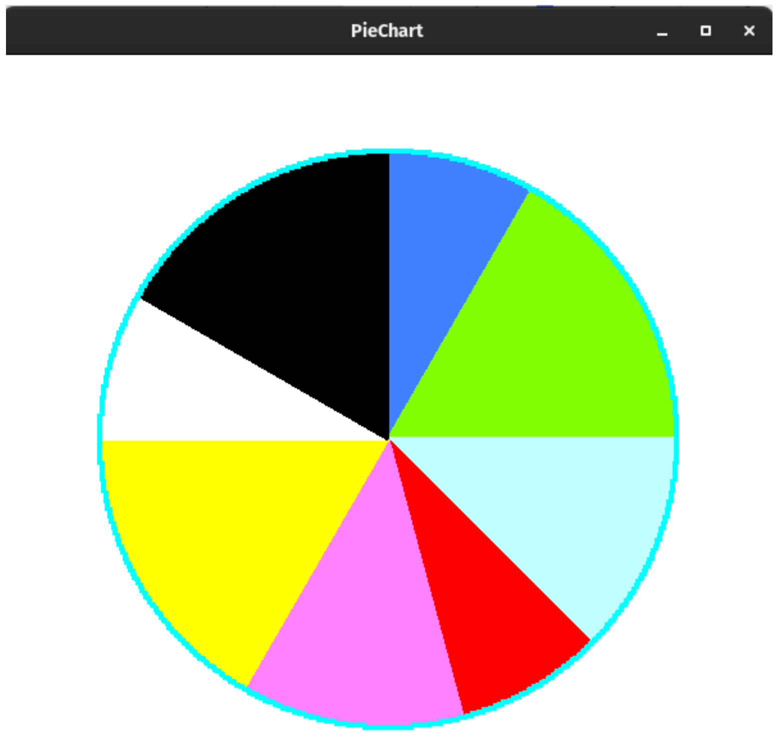


Fig 4.2: Pie Chart 2

Conclusion

In this way, in the first lab of Computer Graphics, we got to become familiar with the OpenGL library and implemented various raster algorithms as discussed above. OpenGL is a cross-language and a cross-platform API for rendering 2D and 3D graphics. I've implemented the algorithms in Python because of the ease and the versatility of the language. We started from drawing a point, then a line, then a circle and finally used all the concepts to draw a pie-chart. Furthermore, we gained good knowledge and experience on presenting graphical elements on the screen at a pixel level.