Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report #3

Implementing Clipping Algorithms in Python

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

CE 3rd Year

Submitted to:

Mr. Dhiraj Shrestha

Assistant Professor

Department of Computer Science and Engineering

Submission date: May 10, 2022

Introduction

In Computer Graphics, line clipping is the process of removing the lines or portions of lines outside an area of interest. Here, given a set of lines and a rectangular area of interest, the task is to remove lines that are outside the rectangular window or the view plane. Only those lines that are inside the view plane are visible. Similarly, polygon clipping is the process of removing the portions of the polygon that are outside the viewing plane.

There are several line clipping and polygon clipping algorithms available. In this lab, we'll implement the following clipping algorithms in Python using OpenGL.

1. Cohen Sutherland Line Clipping Algorithm
2. Liang Barsky Line Clipping Algorithm
3. Sutherland Hodgeman Polygon Clipping algorithm

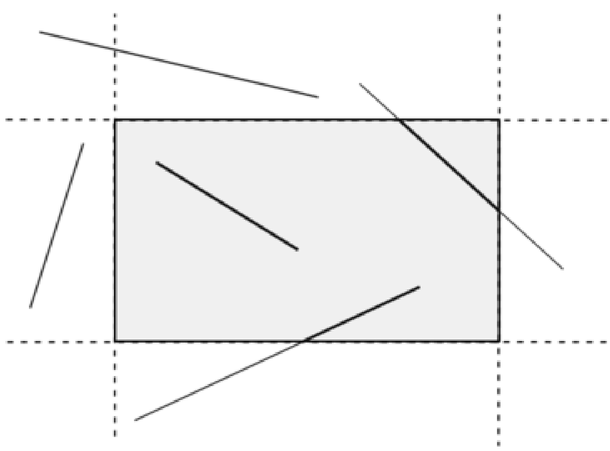

Fig: Line Clipping

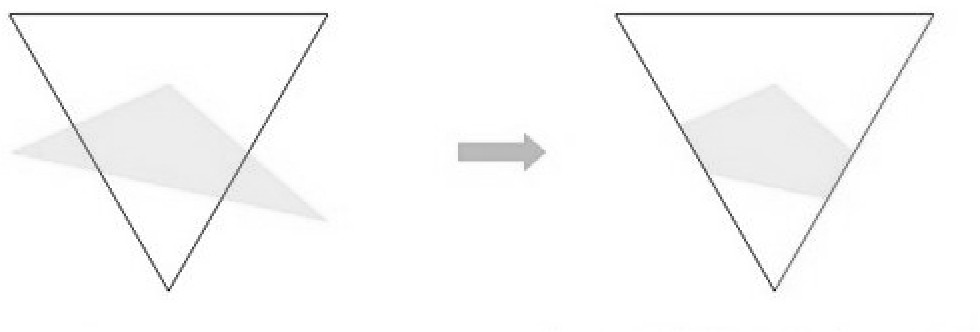

Fig: Polygon Clipping

1. Cohen Sutherland Line Clipping

Cohen Sutherland clipping algorithm uses region code to clip a portion of the line which is not present in the visible region. It divides a region into 9 columns based on the values (X_max, Y_max) and (X_min, Y_min). For all the endpoints of a given line, region codes are assigned.

There can be three different cases for any given line- the line is completely outside, the line is completely inside, and the line is partially inside. In the first case, the line is discarded, in the second case, the whole line is accepted without clipping and in the last case, clipping is performed to accept only the portion of the line that lies inside the clipping window.

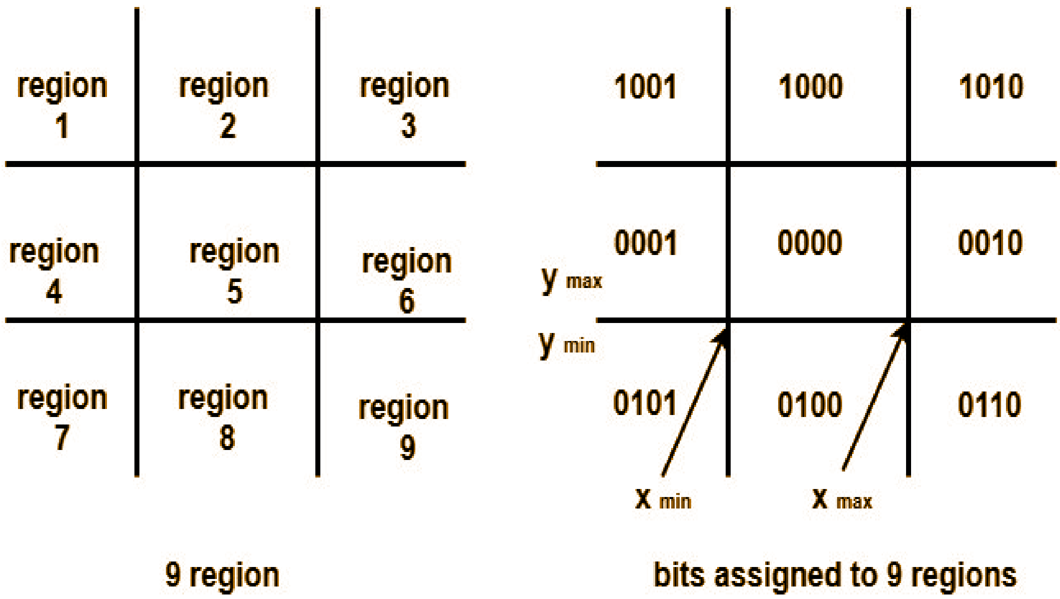

Fig: Region codes in Cohen Sutherland Line Clipping

Algorithm

1. Assign a region code for each endpoint.
2. If both endpoints have a region code 0000, then trivially accept these lines.
3. Else, perform the logical AND operation for both region codes.
   If the result is NOT 0000 → trivially reject the line.
   Else (i.e, result = 0000, need clipping)
   I. Choose an endpoint of the line that is outside the window.

   II. Find the intersection point at the window boundary(based on region code).

III. Replace endpoint with the intersection point and update the region code.

IV. Repeat step 2 until we find a clipped line either trivially accepted or trivially rejected.

4. Repeat step 1 for other lines.

Python Implementation

```
from OpenGL.GL import *

from OpenGL.GLUT import *

from OpenGL.GLU import *

''' Clippping Window '''

XW_MIN = -200

XW_MAX = 200

YW_MIN = -200

YW_MAX = 200


''' Region Codes '''

INSIDE = 0  #0000

LEFT = 1    #0001

RIGHT = 2   #0010

BOTTOM = 4  #0100

TOP = 8     #1000

def draw_window():

   glColor3f(0.0, 0.5, 1.0)

   glLineWidth(3)

   glBegin(GL_LINE_LOOP)

   glVertex2f(XW_MIN,YW_MIN)

   glVertex2f(XW_MAX,YW_MIN)

   glVertex2f(XW_MAX,YW_MAX)
```

```
    glVertex2f(XW_MIN,YW_MAX)
    glEnd()

def initialization():
    glutInit()
    glutInitDisplayMode(GLUT_RGBA)
    glutInitWindowSize(600, 600)
    glutInitWindowPosition(300, 300)
    glutCreateWindow("Line Clipping - Cohen Sutherland")
    glClearColor(1.0,1.0,1.0,0.0)
    gluOrtho2D(-300,300,-300,300)
    glClear(GL_COLOR_BUFFER_BIT)
    glPointSize(1.0)

class Cohen_Sutherland:
    def __init__(self, x1, y1, x2, y2):
        self.x1, self.y1, self.x2, self.y2 = x1, y1, x2, y2
        self.draw_line()
        self.line_clipping()

    #initial line - red
    def draw_line(self):
        glLineWidth(3)
        glColor3f(1.0, 0.0, 0.0)
        glBegin(GL_LINES)
        glVertex2f(self.x1, self.y1)
        glVertex2f(self.x2, self.y2)
        glEnd()
```

```python
#computes TBRL code for the endpoint
def compute_code(self, x, y):
    code = INSIDE
    if x < XW_MIN:
        code |= LEFT
    elif x > XW_MAX:
        code |= RIGHT
    if y < YW_MIN:
        code |= BOTTOM
    elif y > YW_MAX:
        code |= TOP
    return code

def line_clipping(self):
    region_code_1 = self.compute_code(self.x1, self.y1)
    region_code_2 = self.compute_code(self.x2, self.y2)
    partially_inside = False

    while True:
        #Line completely inside
        if region_code_1 == 0 and region_code_2 == 0:
            partially_inside = True
            break
        #Line completely outside
        elif (region_code_1 & region_code_2)!=0:
            break
        #Line needs clipping
        else:
```

```
                    x = 1.0

                    y = 1.0

                    if region_code_1 != 0:

                        code_to_clip = region_code_1

                    else:

                        code_to_clip = region_code_2


                    #finding intersection points

                    if code_to_clip & TOP:

                     x = self.x1 + ((self.x2 - self.x1) / (self.y2 - self.y1)) * (YW_MAX - self.y1)

                        y = YW_MAX

                    elif code_to_clip & BOTTOM:

                         x = self.x1 + ((self.x2 - self.x1) / (self.y2 - self.y1)) * (YW_MIN - self.y1)

                        y = YW_MIN

                    elif code_to_clip & RIGHT:

                         y = self.y1 + ((self.y2 - self.y1) / (self.x2 - self.x1)) * (XW_MAX - self.x1)

                        x = XW_MAX

                    elif code_to_clip & LEFT:

                         y = self.y1 + ((self.y2 - self.y1) / (self.x2 - self.x1)) * (XW_MIN - self.x1)

                        x = XW_MIN


        # replacing outside points with calculated intersection points

                    if code_to_clip == region_code_1:

                        self.x1 = x

                        self.y1 = y

                            region_code_1 = self.compute_code(self.x1, self.y1)
```

```
                    else:
                        self.x2 = x
                        self.y2 = y
                        region_code_2 = self.compute_code(self.x2, self.y2)

            if partially_inside:
                #draw line in green
                glColor3f(0.0, 1.0, 0.0)
                glLineWidth(3)
                glBegin(GL_LINES)
                glVertex2f(self.x1, self.y1)
                glVertex2f(self.x2, self.y2)
                glEnd()

    def LineClipping():
        draw_window()
        line = Cohen_Sutherland(int(line_inputs[0]), int(line_inputs[1]), int(line_inputs[2]), int(line_inputs[3]))
        glFlush()

    if __name__ == "__main__":
        line_inputs = input("Enter the line coordinates in the form x1 y1 x2 y2 : ").split(' ')
        initialization()
        glutDisplayFunc(LineClipping)
        glutMainLoop()
```

Outputs

Output 1

```
(env) sabinthapa_win@pop-os:~/Desktop/repos/Graphics-Labs/lab3$ python3 CohenSutherland.py
Enter the line coordinates in the form x1 y1 x2 y2 : -230 -230 280 280
```

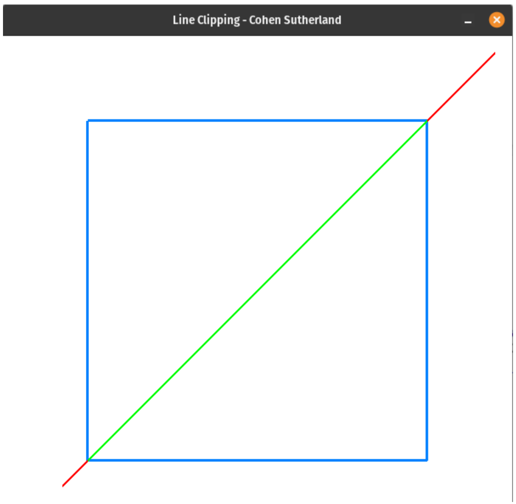

Fig: Line partially outside

Output 2

```
(env) sabinthapa_win@pop-os:~/Desktop/repos/Graphics-Labs/lab3$ python3 CohenSutherland.py
Enter the line coordinates in the form x1 y1 x2 y2 : -250 250 250 250
```

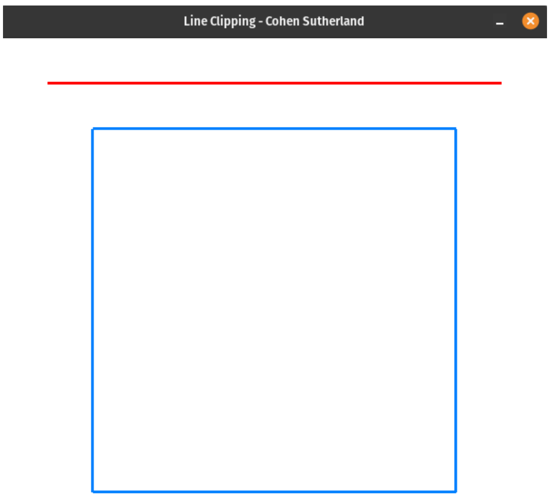

Fig: Line completely outside

Output 3

```
(env) sabinthapa_win@pop-os:~/Desktop/repos/Graphics-Labs/lab3$ python3 CohenSutherland.
Enter the line coordinates in the form x1 y1 x2 y2 : -100 -100 150 150
```

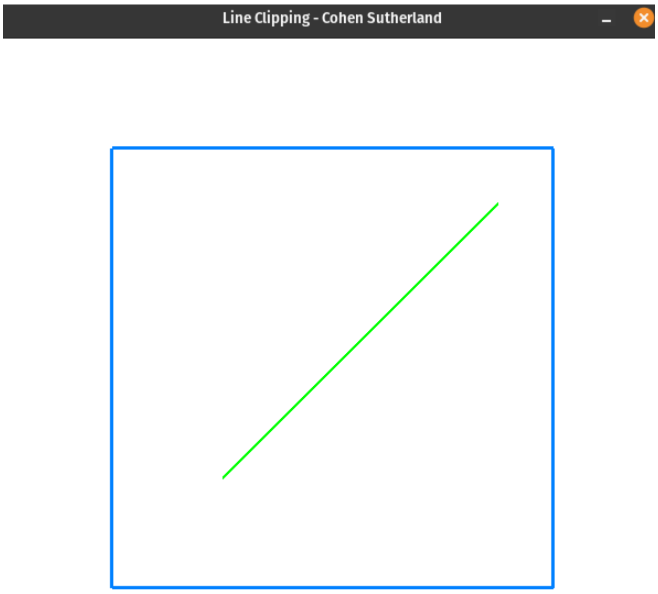

Fig: Line completely inside

2. Liang Barsky Line Clipping

The Liang Barsky Line Clipping algorithm is more efficient than Cohen Sutherland line clipping algorithm. It is considered to be the faster parametric line-clipping algorithm. Here, we consider the parametric equation of the line and the inequalities describing the range of the clipping window which is used to determine the intersections between the line and the clip window.

The advantages of Line Barsky over Cohen Sutherland line clipping are:

1. Intersection calculations are reduced.
2. It requires only one division to update parameters t1 and t2.
3. Window intersections of the line are computed only once.

Algorithm

1. Read two endpoints of the line, $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$.
2. Read two corners (left-top and right-bottom) of the window, say (X_{wmin}, Y_{wmax}, X_{wmax}, Y_{wmin}).
3. Calculate the values of the parameters p_i and q_i for i = 1,2,3,4 such that

   $p_1 = -\Delta x \qquad q_1 = x_1 - x_{wmin}$

   $p_2 = \Delta x \qquad q_2 = x_{wmax} - x_1$

   $p_3 = -\Delta y \qquad q_3 = y_1 - y_{wmin}$

   $p_4 = \Delta y \qquad q_4 = y_{wmax} - y_1$

4. If $p_i = 0$ then

   {The line is parallel to i^{th} boundary.

   Now, if $q_i < o$ then
   {
   Line is completely outside the boundary, hence discard the line segment and goto stop.
   }
   Else
   {
   {

   Check whether the line is horizontal or vertical and accordingly check the line endpoint with corresponding boundaries. If line endpoint/s lie within the bounded area then use them to draw lines otherwise use boundary coordinates to draw lines. Go to stop.

   }

}

5. Initialize values for t_1 and t_2 as

   $t_1 = 0$ and $t_2 = 1$
6. Calculate values q_i/p_i for i = 1,2,3,4
7. Select values q_i/p_i where $p_i < 0$ and assign maximum out of them as t_1.
8. Select values of qi/pi where $p_i > 0$ and assign minimum out of them as t_2.
9. If $(t_1<t_2)$

   {

   $xx_1 = x_1 + t_1\Delta x$

   $xx_2 = x_1 + t_2\Delta x$

   $yy_1 = y_1 + t_1\Delta y$

   $yy_2 = y_1 + t_2\Delta y$

   Draw line (xx_1, yy_1, xx_2, yy_2)

   }
10. Stop.

Python Implementation

```
from OpenGL.GL import *

from OpenGL.GLUT import *

from OpenGL.GLU import *


''' Clipping window '''

XW_MIN = -200

XW_MAX = 200

YW_MIN = -200

YW_MAX = 200


def draw_window():

    glColor3f(0.0, 0.5, 1.0)

    glBegin(GL_LINE_LOOP)

    glVertex2f(XW_MIN,YW_MIN)

    glVertex2f(XW_MAX,YW_MIN)

    glVertex2f(XW_MAX,YW_MAX)

    glVertex2f(XW_MIN,YW_MAX)
```

```python
    glEnd()

def initialization():
    glutInit()
    glutInitDisplayMode(GLUT_RGBA)
    glutInitWindowSize(600, 600)
    glutInitWindowPosition(300, 300)
    glutCreateWindow("Line Clipping - Liang Barsky")
    glClearColor(1.0,1.0,1.0,0.0)
    gluOrtho2D(-300,300,-300,300)
    glClear(GL_COLOR_BUFFER_BIT)
    glPointSize(1.0)

class LiangBarsky:
    def __init__(self, x1, y1, x2, y2):
        self.x1, self.y1, self.x2, self.y2 = x1, y1, x2, y2
        self.draw_line()
        self.line_clipping()

    #initial line - red color
    def draw_line(self):
        glLineWidth(3)
        glColor3f(1.0, 0.0, 0.0)
        glBegin(GL_LINES)
        glVertex2f(self.x1, self.y1)
        glVertex2f(self.x2, self.y2)
        glEnd()
```

```
    def line_clipping(self):
        dx = self.x2 - self.x1
        dy = self.y2 - self.y1
        pks = [-dx, dx, -dy, dy]
        qks = [self.x1 - XW_MIN, XW_MAX - self.x1, self.y1 - YW_MIN, YW_MAX - self.y1]
        u1, u2 = 0, 1

        for (pk, qk) in zip(pks, qks):
        # For all the boudaries
        #if line is parallel to any axes and lies outside
            if pk == 0 and qk<0:
                return
            if pk == 0:
                continue
            u = qk / pk
            if pk < 0:
                u1 = max(u1, u)
            else:
                u2 = min(u2, u)
        #line not completely outside
        if u1 <= u2:
            x1, y1 = self.x1, self.y1
            self.x1 = x1 + u1 * dx
            self.x2 = x1 + u2 * dx
            self.y1 = y1 + u1 * dy
            self.y2 = y1 + u2 * dy

            #green line
```

```python
            glColor3f(0.0, 1.0, 0.0)
            glLineWidth(3)
            glBegin(GL_LINES)
            glVertex2f(self.x1, self.y1)
            glVertex2f(self.x2, self.y2)
            glEnd()


def line_clipping():
    draw_window()
    line = LiangBarsky(int(line_inputs[0]), int(line_inputs[1]), int(line_inputs[2]), int(line_inputs[3]))
    glFlush()


if __name__ == "__main__":
    line_inputs = input("Enter the line coordinates in the form x1 y1 x2 y2 : ").split(' ')
    initialization()
    glutDisplayFunc(line_clipping)
    glutMainLoop()
```

Outputs

Output 1

```
(env) sabinthapa_win@pop-os:~/Desktop/repos/Graphics-Labs/lab3$ python3 LiangBarskey.py
Enter the line coordinates in the form x1 y1 x2 y2 : -150 100 150 100
```

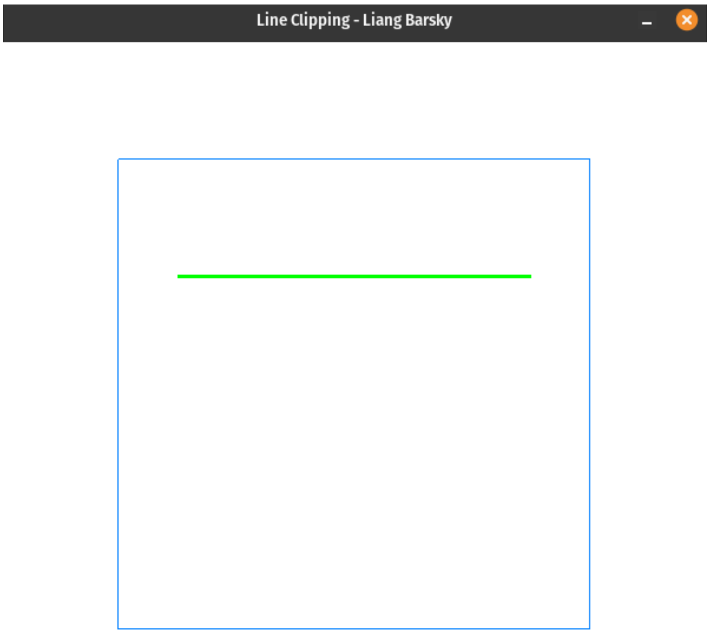

Fig: Line completely inside

Output 2

```
(env) sabinthapa_win@pop-os:~/Desktop/repos/Graphics-Labs/lab3$ python3 LiangBarskey.py
Enter the line coordinates in the form x1 y1 x2 y2 : -250 0 250 0
```

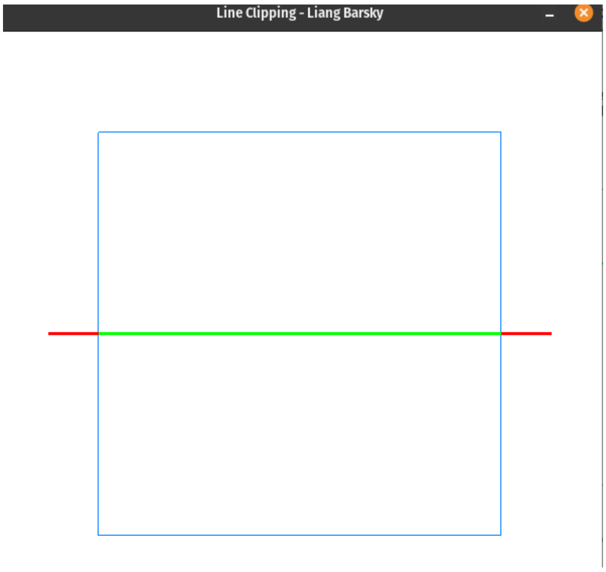

Fig: Line not completely inside

Output 3

```
(env) sabinthapa_win@pop-os:~/Desktop/repos/Graphics-Labs/lab3$ python3 LiangBarskey.py
Enter the line coordinates in the form x1 y1 x2 y2 : -250 250 250 250
```

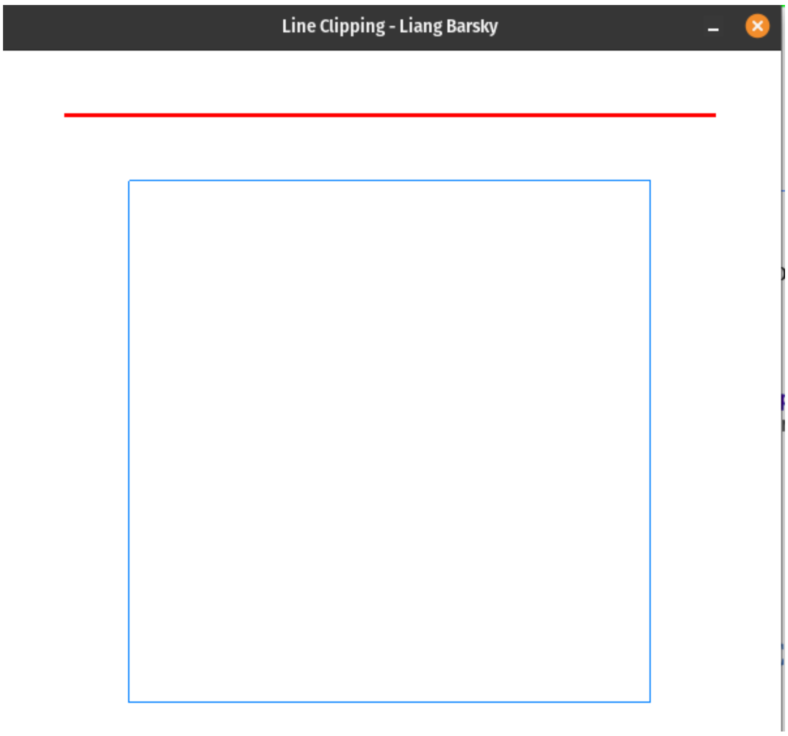

Fig: Line completely outside

3. Sutherland Hodgeman Line Clipping

Sutherland Hodgeman is a very popular polygon clipping algorithm. It is performed by processing the boundary of a polygon against each window corner or edge. Firstly, the entire polygon is clipped against one edge, then the resulting polygon is considered, then the polygon is considered against the second edge. The four possible situations while processing are:

1. **Both vertices are inside.**

   In this case, only the second vertex is added to the output list
2. **First vertex is outside while the second one is inside.**

   In this case, both the point of intersection of the dege with the clip boundary is added to the output list.
3. **First vertex is inside while the second one is outside.**

   In this case, only the point of intersection of the edge with the clip boundary is added to the output list.
4. **Both vertices are outside**

   In this case, no vertices are added to the output list.

Algorithm

For each boundary of the clipping window, perform the following:

1. Find the case among in-to-out, out-to-in, in-to-in, out-to-out for each edge (taken two endpoints at a time) of the polygon.
2. According to the case, calculate the intersection point and vertices in a new output list based on the following four possible situations:
   a. If the first vertex is outside the window boundary and the second vertex is inside, both the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list.
   b. If both input vertices are inside the window boundary, only the second vertex is added to the output vertex list.
   c. If the first vertex is inside the window boundary and the second vertex is outside, only the edge intersection with the window boundary is added to the output vertex list.
   d. If both input vertices are outside the window boundary, nothing is added to the output list.
3. Find all the vertices by repeating a and b for all the edges.
4. The output list contains the final value of the resulting vertices of the clipped polygon.

Python Implementation

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

''' Window '''
XW_MIN = -200
XW_MAX = 200
YW_MIN = -200
YW_MAX = 200
LEFT = 0
RIGHT = 1
BOTTOM = 2
TOP = 3
IN_TO_IN = 0
IN_TO_OUT = 1
OUT_TO_IN = 2
OUT_TO_OUT = 3
def initialization():
   glutInit()
   glutInitDisplayMode(GLUT_RGBA)
   glutInitWindowSize(600, 600)
   glutInitWindowPosition(350, 350)
   glutCreateWindow("Polygon Clipping - Sutherland Hodgeman")
   glClearColor(1.0,1.0,1.0,0.0)
   gluOrtho2D(-350,350,-350,350)
   glClear(GL_COLOR_BUFFER_BIT)
   glLineWidth(3)
```

```
def draw_window():
    glColor3f(0.0, 0.5, 1.0)
    glBegin(GL_LINE_LOOP)
    glVertex2f(XW_MIN,YW_MIN)
    glVertex2f(XW_MAX,YW_MIN)
    glVertex2f(XW_MAX,YW_MAX)
    glVertex2f(XW_MIN,YW_MAX)
    glEnd()

def get_case(boundary, p1, p2):
    (x1,y1) = p1 #tuple unpacking
    (x2,y2) = p2
    if boundary == LEFT:
        if x1>=XW_MIN and x2>=XW_MIN:
            return IN_TO_IN
        elif x1>=XW_MIN:
            return IN_TO_OUT
        elif x2>=XW_MIN:
            return OUT_TO_IN
        else:
            return OUT_TO_OUT
    elif boundary == RIGHT:
        if x1<=XW_MAX and x2<=XW_MAX:
            return IN_TO_IN
        elif x1<=XW_MAX:
            return IN_TO_OUT
        elif x2<=XW_MAX:
            return OUT_TO_IN
```

```
        else:
            return OUT_TO_OUT
    elif boundary == BOTTOM:
        if y1>=YW_MIN and y2>=YW_MIN:
            return IN_TO_IN
        elif y1>=YW_MIN:
            return IN_TO_OUT
        elif y2>=YW_MIN:
            return OUT_TO_IN
        else:
            return OUT_TO_OUT
    elif boundary == TOP:
        if y1<=YW_MAX and y2<=YW_MAX:
            return IN_TO_IN
        elif y1<=YW_MAX:
            return IN_TO_OUT
        elif y2<=YW_MAX:
            return OUT_TO_IN
        else:
            return OUT_TO_OUT

def find_intersection(boundary, p1, p2):
    (x1,y1) = p1
    (x2,y2) = p2
    m=0
    if x1 != x2:
        m = (y2-y1)/(x2-x1)
    if boundary == LEFT:
```

```
        return (XW_MIN, y1+m*(XW_MIN-x1))
    elif boundary == RIGHT:
        return (XW_MAX, y1+m*(XW_MAX-x1))
    elif boundary == BOTTOM:
        if x1==x2:
            return (x1,YW_MIN)
        else:
            return (x1+(YW_MIN-y1)/m, YW_MIN)
    elif boundary == TOP:
        if x1==x2:
            return (x1,YW_MAX)
        else:
            return (x1+(YW_MAX-y1)/m, YW_MAX)

def polygon_clipper(points):
    for boundary in range(4):
        new_points = []
        for i in range(len(points)):
            p1 = points[i]
            p2 = points[(i+1)%(len(points))]

            case = get_case(boundary,p1,p2)

            if case == IN_TO_IN:
                new_points.append(p2)
            elif case == IN_TO_OUT:
                p = find_intersection(boundary,p1,p2)
                new_points.append(p)
```

```
            elif case == OUT_TO_IN:
                p = find_intersection(boundary,p1,p2)
                new_points.append(p)
                new_points.append(p2)

        points = new_points
    return points
def draw_polygon(points):
    for i in range(len(points)):
        (x1,y1) = points[i]
        (x2,y2) = points[(i+1)%len(points)]
        glBegin(GL_LINES)
        glVertex2f(x1, y1)
        glVertex2f(x2, y2)
        glEnd()

def SutherlandHodgeman():
    draw_window()
    points = data
    #initial
    glColor3f(1.0, 0.0, 0.0)
    draw_polygon(points)
    #new
    new_points = polygon_clipper(points)
    glColor3f(0.0, 1.0, 0.0)
    draw_polygon(new_points)
    glFlush()
```

```python
if __name__ == "__main__":
    choice = input("Enter the polygon coordinates in the form [(x1, y1), (x2, y2), ...] : \n").strip()[1:-1]
    choice = choice.replace("(", "")
    choice = choice.replace(")", "")
    choice = choice.split(",")
    data = []

    for i in range(0, len(choice), 2):
        data.append((int(choice[i]), int(choice[i+1])))

    initialization()
    glutDisplayFunc(SutherlandHodgeman)
    glutMainLoop()
```

Output

Output 1

```
(env) sabinthapa_win@pop-os:~/Desktop/repos/Graphics-Labs/lab3$ python3 SutherlandHodgeman.py
Enter the polygon coordinates in the form [(x1, y1), (x2, y2), ...] :
[(-250, 130),( 100, 250), (250, 150), (150, -250), (-150, -150)]
```

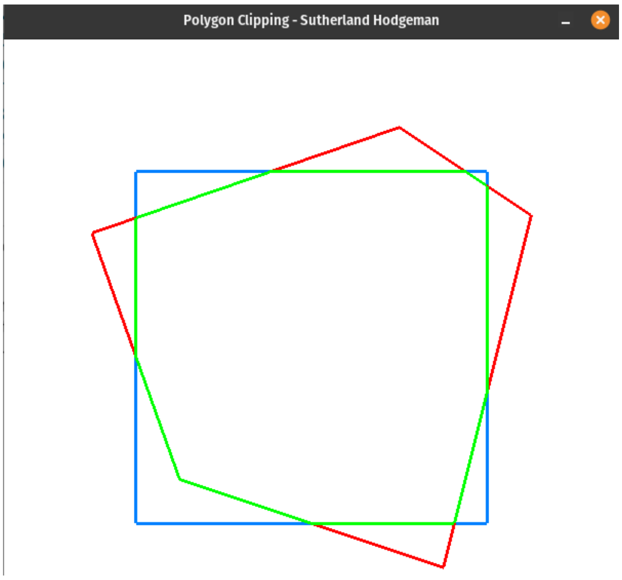

Fig: Polygon partially outside

Output 2

```
(env) sabinthapa_win@pop-os:~/Desktop/repos/Graphics-Labs/lab3$ python3 SutherlandHodgeman.py
Enter the polygon coordinates in the form [(x1, y1), (x2, y2), ...] :
[(-250, 80),( 50, 250), (170, 80), (100, -250), (-100, -100)]
```

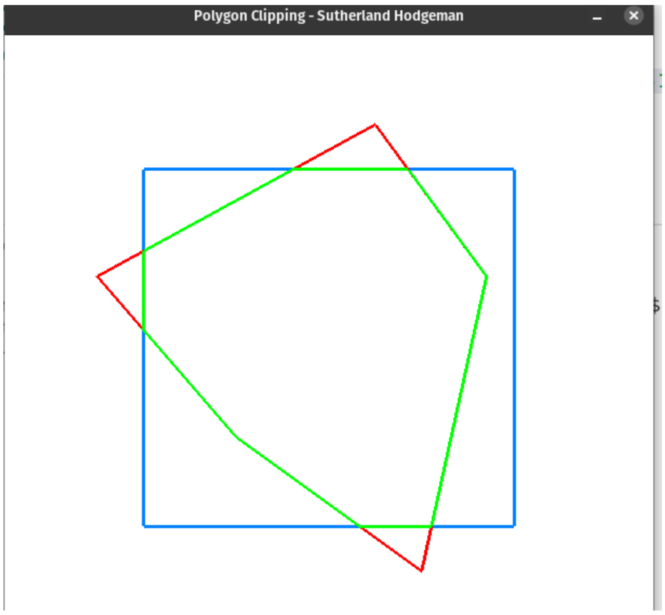

Fig: Polygon partially outside

Conclusion

In this way, in the third lab of Computer Graphics, the clipping algorithms for lines and polygons were implemented using OpenGL in Python. The algorithms, along with their source codes and outputs are discussed above. This lab has helped us visualize how computer graphics is used to render only the portion of images on the screen and cut out the portion that lie outside the view plane.