# Table of Content

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No.** | **Practical Name** | **Date** | **Page No.** | **Remarks** |
| 6. | Implement 2D transformations of a rectangle. Step By Step Procedural Algorithm   1. Flip the point or shape across the x-axis. 2. Flip the point or shape across the y-axis. 3. Reflect the point or shape across a given line. 4. Skew the point or shape along the x-axis. 5. Skew the point or shape along the y-axis | 19-09-2024 | 20-24 |  |
| 7. | Implement the line clipping using Cohen Sutherland clipping algorithm | 26-09-2024 | 25-29 |  |
| 8. | To write a program to implement the line clipping using Liang Barsky line clipping algorithm. | 17-10-2024 | 30-33 |  |
| 9. | To write a program to implement the polygon clipping using Sutherland Hodgeman polygon clipping algorithm. | 7-11-2024 | 34-37 |  |
| 10. | Implement 3D transformations of a object. Step By Step Procedural Algorithm   1. Enter the choice for transformation. 2. Perform the translation, rotation and scaling of 3d object. 3. Get the needed parameters for the transformation from the user. 4. Incase of rotation, object can be rotated about x or y axis. 5. Display the transmitted object in the screen along with new generated coordinates. | 14-11-2024 | 38- |  |

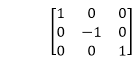
**Practical#6**

**Objective:** Implement 2D transformations of a rectangle.  
Step By Step Procedural Algorithm

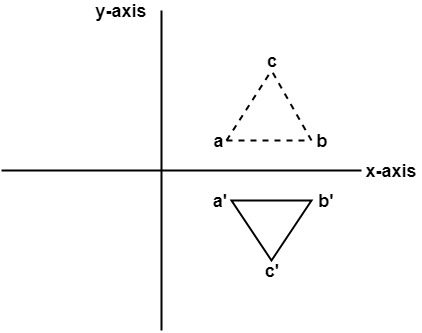
1. Flip the point or shape across the x-axis.
2. Flip the point or shape across the y-axis.
3. Reflect the point or shape across a given line.
4. Skew the point or shape along the x-axis.
5. Skew the point or shape along the y-axis.

**Theory:**

**1. Reflection about x-axis:** The object can be reflected about x-axis with the help of the following matrix



In this transformation value of x will remain same whereas the value of y will become negative. Following figures shows the reflection of the object axis. The object will lie another side of the x-axis.

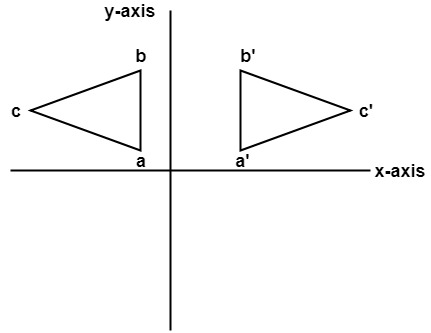


**2. Reflection about y-axis:** The object can be reflected about y-axis with the help of following transformation matrix

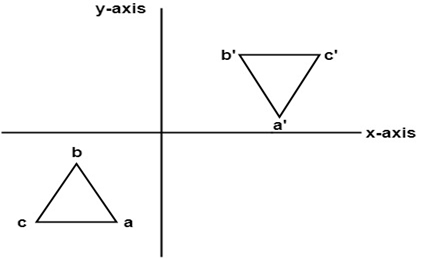
Reflection

Here the values of x will be reversed, whereas the value of y will remain the same. The object will lie another side of the y-axis.

The following figure shows the reflection about the y-axis

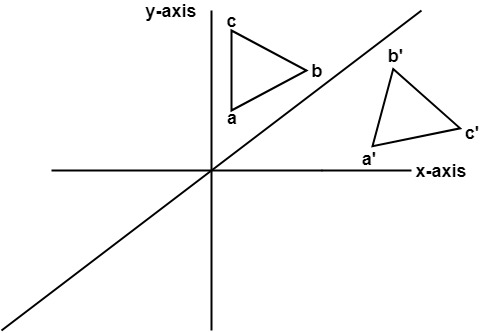


**3. Reflection about an axis perpendicular to xy plane and passing through origin:**  
In the matrix of this transformation is given below

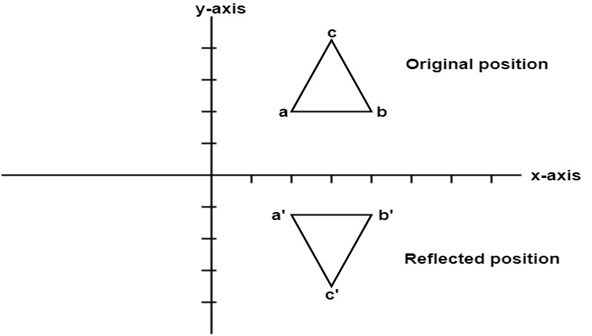
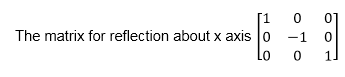
Reflection  


In this value of x and y both will be reversed. This is also called as half revolution about the origin.

**4. Reflection about line y=x:** The object may be reflected about line y = x with the help of following transformation matrix

Reflection  


First of all, the object is rotated at 45°. The direction of rotation is clockwise. After it reflection is done concerning x-axis. The last step is the rotation of y=x back to its original position that is counterclockwise at 45°.

**Code:**

import **matplotlib**.**pyplot** as **plt**

import **numpy** as **np**

def **reflect\_x**(points):

    reflection\_matrix = **np**.**array**([[1, 0],

                                  [0, -1]])

    return points.dot(reflection\_matrix)

def **reflect\_y**(points):

    reflection\_matrix = **np**.**array**([[-1, 0],

                                  [0, 1]])

    return points.dot(reflection\_matrix)

def **reflect\_line\_y\_equals\_x**(points):

    reflection\_matrix = **np**.**array**([[0, 1],

                                  [1, 0]])  *# Swapping x and y*

    return points.dot(reflection\_matrix)

def **shear\_x**(points, shx):

    shearing\_matrix = **np**.**array**([[1, shx],

                                [0, 1]])

    return points.dot(shearing\_matrix)

def **shear\_y**(points, shy):

    shearing\_matrix = **np**.**array**([[1, 0],

                                [shy, 1]])

    return points.dot(shearing\_matrix)

def **plot\_shapes**(original, transformed, title):

**plt**.**figure**()

**plt**.**plot**(\***zip**(\***np**.**vstack**((original, original[0]))), label='Original Shape', linestyle='--', color='blue')

**plt**.**plot**(\***zip**(\***np**.**vstack**((transformed, transformed[0]))), label='Transformed Shape', linestyle='-', color='red')

**plt**.**legend**()

**plt**.**xlabel**('X-axis')

**plt**.**ylabel**('Y-axis')

**plt**.**title**(title)

**plt**.**grid**(True)

**plt**.**axis**('equal')

**plt**.**show**()

def **main**():

*# Asymmetric shape (trapezoid)*

    shape = **np**.**array**([[1, 1], [3, 1], [2, 3], [1, 2]])

*# Reflection across the x-axis*

    reflected\_x = **reflect\_x**(shape)

**plot\_shapes**(shape, reflected\_x, 'Reflection across the X-axis')

*# Reflection across the y-axis*

    reflected\_y = **reflect\_y**(shape)

**plot\_shapes**(shape, reflected\_y, 'Reflection across the Y-axis')

*# Reflection across the line y = x*

    reflected\_line = **reflect\_line\_y\_equals\_x**(shape)

**plot\_shapes**(shape, reflected\_line, 'Reflection across the line y = x')

*# Shearing along the x-axis*

    shx = 1  *# Example shearing factor*

    sheared\_x = **shear\_x**(shape, shx)

**plot\_shapes**(shape, sheared\_x, 'Shearing along the X-axis')

*# Shearing along the y-axis*

    shy = 1  *# Example shearing factor*

    sheared\_y = **shear\_y**(shape, shy)

**plot\_shapes**(shape, sheared\_y, 'Shearing along the Y-axis')

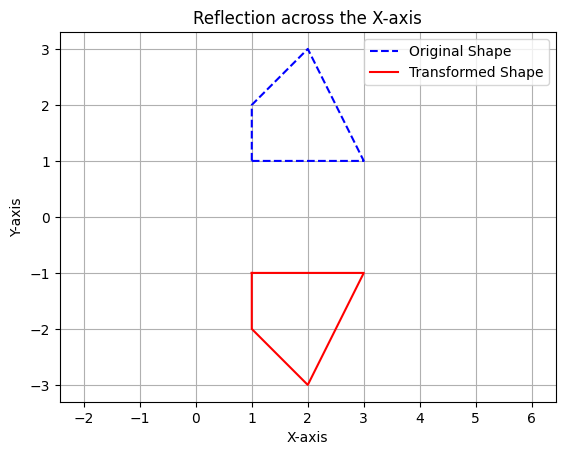
if \_\_name\_\_ == "\_\_main\_\_":

**main**()

**Output:**

**A graph of a triangle and a blue line

Description automatically generated**

****

**A graph with red lines and blue lines

Description automatically generated**

**A graph of reflection across the line

Description automatically generated**

**A graph with red and blue lines

Description automatically generated**

**Practical#7**

**Objective:** Implement the line clipping using Cohen Sutherland clipping algorithm  
Step By Step Procedural Algorithm

1. Get the clip window coordinates.
2. Get the line end points.
3. Draw the window and the line.
4. Remove the line points which are plotted in outside the window.
5. Draw the window with clipped line.

**Theory:**

In the algorithm, first of all, it is detected whether line lies inside the screen or it is outside the screen. All lines come under any one of the following categories:

1. Visible
2. Not Visible
3. Clipping Case

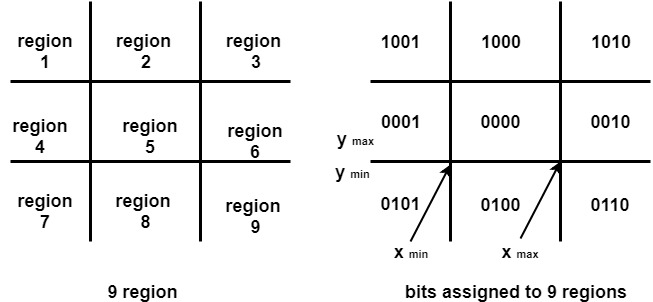
1. Visible: If a line lies within the window, i.e., both endpoints of the line lies within the window. A line is visible and will be displayed as it is.

2. Not Visible: If a line lies outside the window it will be invisible and rejected. Such lines will not display. If any one of the following inequalities is satisfied, then the line is considered invisible. Let A (x1,y2) and B (x2,y2) are endpoints of line.

xmin,xmax are coordinates of the window.

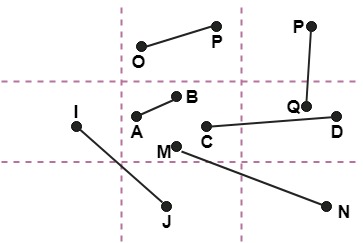
ymin,ymax are also coordinates of the window.  
          x1>xmax  
          x2>xmax  
          y1>ymax  
          y2>ymax  
          x1<xmin  
          x2<xmin  
          y1<ymin  
          y2<ymin

3. Clipping Case: If the line is neither visible case nor invisible case. It is considered to be clipped case. First of all, the category of a line is found based on nine regions given below. All nine regions are assigned codes. Each code is of 4 bits. If both endpoints of the line have end bits zero, then the line is considered to be visible.



The center area is having the code, 0000, i.e., region 5 is considered a rectangle window.

Following figure show lines of various types



Line AB is the visible case  
Line OP is an invisible case  
Line PQ is an invisible line  
Line IJ are clipping candidates  
Line MN are clipping candidate  
Line CD are clipping candidate

**Code:**

import matplotlib.pyplot as plt

*# Define region codes*

INSIDE = 0  *# 0000*

LEFT = 1    *# 0001*

RIGHT = 2   *# 0010*

BOTTOM = 4  *# 0100*

TOP = 8     *# 1000*

*# Compute the region code for a point (x, y)*

def compute\_code(x, y, x\_min, y\_min, x\_max, y\_max):

    code = INSIDE

    if x < x\_min:      *# to the left of rectangle*

        code |= LEFT

    elif x > x\_max:    *# to the right of rectangle*

        code |= RIGHT

    if y < y\_min:      *# below the rectangle*

        code |= BOTTOM

    elif y > y\_max:    *# above the rectangle*

        code |= TOP

    return code

*# Cohen-Sutherland clipping algorithm*

def cohen\_sutherland\_clip(x1, y1, x2, y2, x\_min, y\_min, x\_max, y\_max):

    code1 = compute\_code(x1, y1, x\_min, y\_min, x\_max, y\_max)

    code2 = compute\_code(x2, y2, x\_min, y\_min, x\_max, y\_max)

    accept = False

    while True:

        if code1 == 0 and code2 == 0:

            accept = True

            break

        elif (code1 & code2) != 0:

            break

        else:

            x, y = 0, 0

            if code1 != 0:

                code\_out = code1

            else:

                code\_out = code2

            if code\_out & TOP:

                x = x1 + (x2 - x1) \* (y\_max - y1) / (y2 - y1)

                y = y\_max

            elif code\_out & BOTTOM:

                x = x1 + (x2 - x1) \* (y\_min - y1) / (y2 - y1)

                y = y\_min

            elif code\_out & RIGHT:

                y = y1 + (y2 - y1) \* (x\_max - x1) / (x2 - x1)

                x = x\_max

            elif code\_out & LEFT:

                y = y1 + (y2 - y1) \* (x\_min - x1) / (x2 - x1)

                x = x\_min

            if code\_out == code1:

                x1, y1 = x, y

                code1 = compute\_code(x1, y1, x\_min, y\_min, x\_max, y\_max)

            else:

                x2, y2 = x, y

                code2 = compute\_code(x2, y2, x\_min, y\_min, x\_max, y\_max)

    if accept:

        return (x1, y1, x2, y2)

    else:

        return None

def plot\_line(x1, y1, x2, y2, label):

    plt.plot([x1, x2], [y1, y2], label=label)

def main():

    x\_min = float(input("Enter the x\_min of the clip window: "))

    y\_min = float(input("Enter the y\_min of the clip window: "))

    x\_max = float(input("Enter the x\_max of the clip window: "))

    y\_max = float(input("Enter the y\_max of the clip window: "))

    x1 = float(input("Enter the x1 of the line: "))

    y1 = float(input("Enter the y1 of the line: "))

    x2 = float(input("Enter the x2 of the line: "))

    y2 = float(input("Enter the y2 of the line: "))

    plt.figure()

    plt.plot([x\_min, x\_max, x\_max, x\_min, x\_min], [y\_min, y\_min, y\_max, y\_max, y\_min], 'k-', label='Clip Window')

    plot\_line(x1, y1, x2, y2, label='Original Line')

    clipped\_line = cohen\_sutherland\_clip(x1, y1, x2, y2, x\_min, y\_min, x\_max, y\_max)

    if clipped\_line:

        plot\_line(\*clipped\_line, label='Clipped Line')

    plt.xlabel('X-axis')

    plt.ylabel('Y-axis')

    plt.title('Cohen-Sutherland Line Clipping Algorithm')

    plt.legend()

    plt.grid(True)

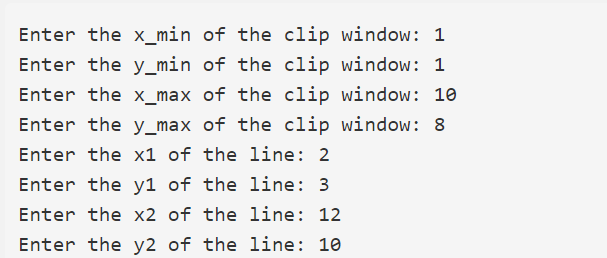
    plt.axis('equal')

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Input:**

****

**Output:**

A graph with a line and a line

Description automatically generated

**Practical#8**

**Objective:** To write a program to implement the line clipping using Liang Barsky line clipping algorithm.

Step By Step Procedural Algorithm

1. Get the clip window coordinates.
2. Get the line end points.
3. Draw the window and the line.
4. Remove the line points which are plotted in outside the window.
5. Draw the window with clipped line.

**Theory:**

The Liang-Barsky algorithm is a line clipping algorithm. This algorithm is more efficient than Cohen–Sutherland line clipping algorithm and can be extended to 3-Dimensional clipping. This algorithm is considered to be the faster parametric line-clipping algorithm. The following concepts are used in this clipping:

1. The parametric equation of the line.
2. The inequalities describing the range of the clipping window which is used to determine the intersections between the line and the clip window.

The parametric equation of a line can be given by,

X = x1 + t(x2-x1)

Y = y1 + t(y2-y1)

Where, t is between 0 and 1. Then, writing the point-clipping conditions in the parametric form:

xwmin <= x1 + t(x2-x1) <= xwmax

ywmin <= y1 + t(y2-y1) <= ywmax

The above 4 inequalities can be expressed as,

tpk <= qk

Where k = 1, 2, 3, 4 (correspond to the left, right, bottom, and top boundaries, respectively). The p and q are defined as,

p1 = -(x2-x1), q1 = x1 - xwmin (Left Boundary)

p2 = (x2-x1), q2 = xwmax - x1 (Right Boundary)

p3 = -(y2-y1), q3 = y1 - ywmin (Bottom Boundary)

p4 = (y2-y1), q4 = ywmax - y1 (Top Boundary)

When the line is parallel to a view window boundary, the p value for that boundary is zero. When pk < 0, as t increase line goes from the outside to inside (entering). When pk > 0, line goes from inside to outside (exiting). When pk = 0 and qk < 0 then line is trivially invisible because it is outside view window. When pk = 0 and qk > 0 then the line is inside the corresponding window boundary. Using the following conditions, the position of line can be determined:

| **Condition** | **Position of line** |
| --- | --- |
| pk = 0 | parallel to the clipping boundaries |
| pk = 0 and qk < 0 | completely outside the boundary |
| pk = 0 and qk >= 0 | inside the parallel clipping boundary |
| pk < 0 | line proceeds from outside to inside |
| pk > 0 | line proceeds from inside to outside |

Parameters t1 and t2 can be calculated that define the part of line that lies within the clip rectangle. When,

1. pk < 0, maximum(0, qk/pk) is taken.
2. pk > 0, minimum(1, qk/pk) is taken.

If t1 > t2, the line is completely outside the clip window and it can be rejected. Otherwise, the endpoints of the clipped line are calculated from the two values of parameter t.

Line CD are clipping candidate

**Code:**

import matplotlib.pyplot as plt

def liang\_barsky(x\_min, y\_min, x\_max, y\_max, x1, y1, x2, y2):

dx = x2 - x1

dy = y2 - y1

p = [-dx, dx, -dy, dy]

q = [x1 - x\_min, x\_max - x1, y1 - y\_min, y\_max - y1]

t\_enter = 0.0

t\_exit = 1.0

for i in range(4):

if p[i] == 0:

if q[i] < 0:

return None

else:

t = q[i] / p[i]

if p[i] < 0:

if t > t\_enter:

t\_enter = t

else:

if t < t\_exit:

t\_exit = t

if t\_enter > t\_exit:

return None

x1\_clip = x1 + t\_enter \* dx

y1\_clip = y1 + t\_enter \* dy

x2\_clip = x1 + t\_exit \* dx

y2\_clip = y1 + t\_exit \* dy

return x1\_clip, y1\_clip, x2\_clip, y2\_clip

x\_min, y\_min = 20, 20

x\_max, y\_max = 80, 80

x1, y1 = 10, 30

x2, y2 = 90, 60

clipped\_line = liang\_barsky(x\_min, y\_min, x\_max, y\_max, x1, y1, x2, y2)

plt.figure(figsize=(8, 6))

plt.plot([x\_min, x\_max, x\_max, x\_min, x\_min], [y\_min, y\_min,

y\_max, y\_max, y\_min], 'b', label='Clipping Window')

if clipped\_line is not None:

x1\_clip, y1\_clip, x2\_clip, y2\_clip = clipped\_line

plt.plot([x1, x2], [y1, y2], 'r', label='Original Line')

plt.plot([x1\_clip, x2\_clip], [y1\_clip, y2\_clip], 'g', label='Clipped Line')

plt.title('Liang-Barsky Line Clipping Algorithm')

plt.legend()

else:

window

plt.title('Line is outside the clipping window')

plt.xlabel('X-axis')

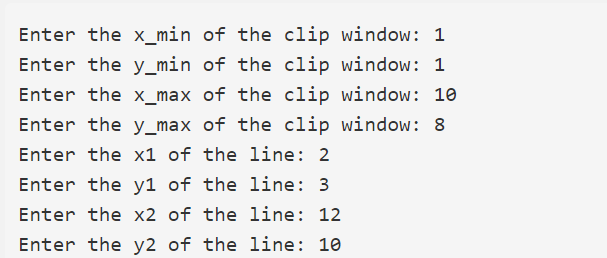
plt.ylabel('Y-axis')

plt.grid()

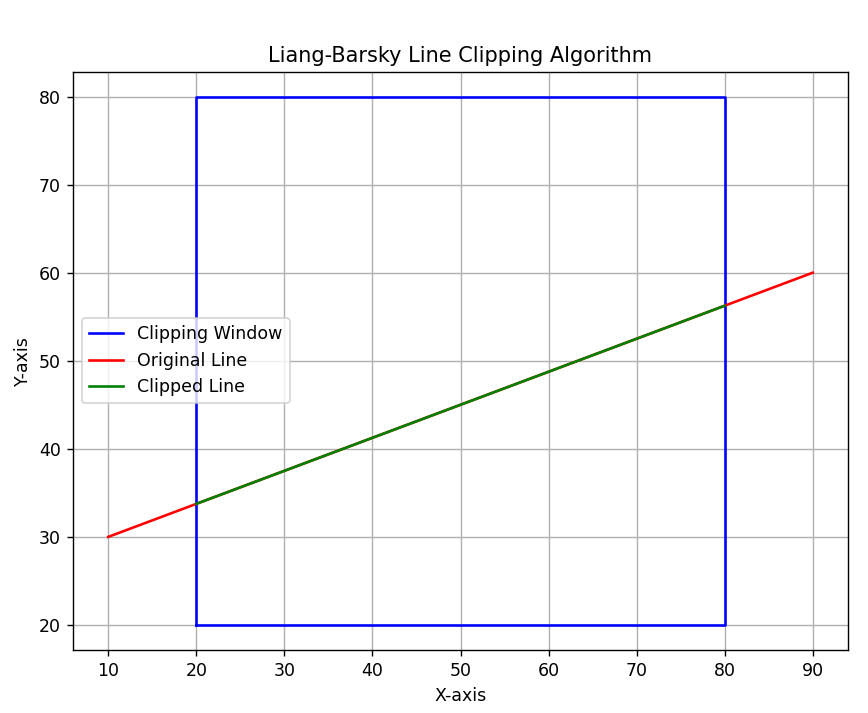
plt.axis('equal')

plt.show()

**Input:**

****

**Output:**



**Practical#9**

**Objective:** To write a program to implement the polygon clipping using Sutherland Hodgeman polygon clipping algorithm.

Step By Step Procedural Algorithm

1. Get the clip window coordinates.
2. Get the polygon end points.
3. Draw the window and the polygon.
4. Remove the polygon outside edges which are plotted in outside the window.
5. Draw the window with clipped polygon

**Theory:**

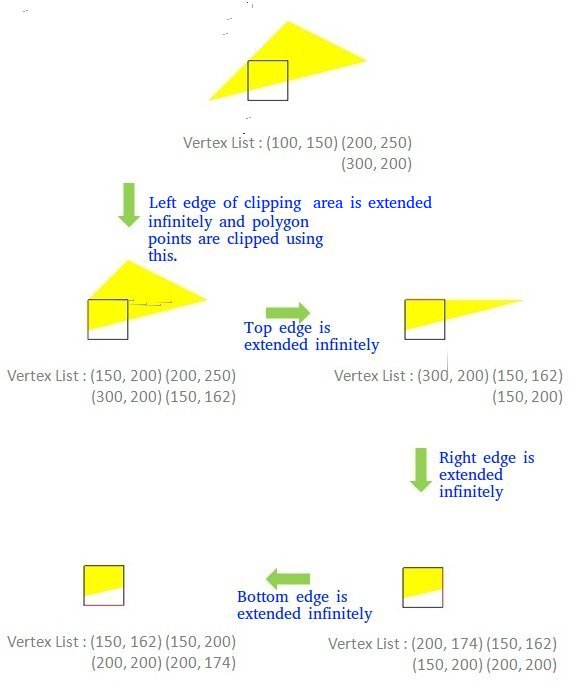
A convex polygon and a convex clipping area are given. The task is to clip polygon edges using the Sutherland–Hodgman Algorithm. Input is in the form of vertices of the polygon in **clockwise order**

How to clip against an edge of clipping area?

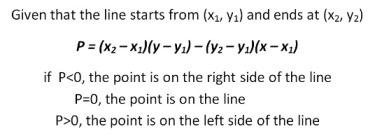
The edge (of clipping area) is extended infinitely to create a boundary and all the vertices are clipped using this boundary. The new list of vertices generated is passed to the next edge of the clip polygon in clockwise fashion until all the edges have been used.

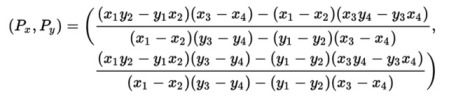
There are four possible cases for any given edge of given polygon against current clipping edge e.

1. Both vertices are inside : Only the second vertex is added to the output list
2. First vertex is outside while second one is inside : Both the point of intersection of the edge with the clip boundary and the second vertex are added to the output list
3. First vertex is inside while second one is outside : Only the point of intersection of the edge with the clip boundary is added to the output list
4. Both vertices are outside : No vertices are added to the output list



There are two sub-problems that need to be discussed before implementing the algorithm:- To decide if a point is inside or outside the clipper polygon If the vertices of the clipper polygon are given in clockwise order then all the points lying on the right side of the clipper edges are inside that polygon. This



can be calculated using : To find the point of intersection of an edge with the clip boundary If two points of each line(1,2 & 3,4) are known, then their point of intersection can be calculated using the formula :-

**Code:**

import matplotlib.pyplot as plt

import numpy as np

MAX\_POINTS = 20

def x\_intersect(x1, y1, x2, y2, x3, y3, x4, y4):

num = (x1\*y2 - y1\*x2) \* (x3-x4) - (x1-x2) \* (x3\*y4 - y3\*x4)

den = (x1-x2) \* (y3-y4) - (y1-y2) \* (x3-x4)

return num/den

def y\_intersect(x1, y1, x2, y2, x3, y3, x4, y4):

num = (x1\*y2 - y1\*x2) \* (y3-y4) - (y1-y2) \* (x3\*y4 - y3\*x4)

den = (x1-x2) \* (y3-y4) - (y1-y2) \* (x3-x4)

return num/den

def clip(poly\_points, poly\_size, x1, y1, x2, y2):

new\_points = np.zeros((MAX\_POINTS, 2), dtype=int)

new\_poly\_size = 0

for i in range(poly\_size):

k = (i+1) % poly\_size

ix, iy = poly\_points[i]

kx, ky = poly\_points[k]

i\_pos = (x2-x1) \* (iy-y1) - (y2-y1) \* (ix-x1)

k\_pos = (x2-x1) \* (ky-y1) - (y2-y1) \* (kx-x1)

if i\_pos < 0 and k\_pos < 0:

new\_points[new\_poly\_size] = [kx, ky]

new\_poly\_size += 1

elif i\_pos >= 0 and k\_pos < 0:

new\_points[new\_poly\_size] = [x\_intersect(x1, y1, x2, y2, ix, iy, kx, ky),

y\_intersect(x1, y1, x2, y2, ix, iy, kx, ky)]

new\_poly\_size += 1

new\_points[new\_poly\_size] = [kx, ky]

new\_poly\_size += 1

elif i\_pos < 0 and k\_pos >= 0:

new\_points[new\_poly\_size] = [x\_intersect(x1, y1, x2, y2, ix, iy, kx, ky),

y\_intersect(x1, y1, x2, y2, ix, iy, kx, ky)]

new\_poly\_size += 1

else:

clipped\_poly\_points = np.zeros((new\_poly\_size, 2), dtype=int)

for i in range(new\_poly\_size):

clipped\_poly\_points[i] = new\_points[i]

return clipped\_poly\_points, new\_poly\_size

def suthHodgClip(poly\_points, poly\_size, clipper\_points, clipper\_size):

for i in range(clipper\_size):

k = (i+1) % clipper\_size

poly\_points, poly\_size = clip(poly\_points, poly\_size, clipper\_points[i][0],

clipper\_points[i][1], clipper\_points[k][0],

clipper\_points[k][1])

for i in range(poly\_size):

print('(', poly\_points[i][0], ', ', poly\_points[i][1], ')')

if \_\_name\_\_ == "\_\_main\_\_":

poly\_size = 3

poly\_points = np.array([[100,150], [200,250], [300,200]])

clipper\_size = 4

clipper\_points = np.array([[150,150], [150,200], [200,200], [200,150]])

suthHodgClip(poly\_points, poly\_size, clipper\_points, clipper\_size)Input:

**Output:**

A group of numbers with different expressions

Description automatically generated with medium confidence

**Practical#10**

**Objective:** Implement 3D transformations of a object.  
Step By Step Procedural Algorithm  
1. Enter the choice for transformation.  
2. Perform the translation, rotation and scaling of 3d object.  
3. Get the needed parameters for the transformation from the user.  
4. Incase of rotation, object can be rotated about x or y axis.  
5. Display the transmitted object in the screen along with new generated coordinates.

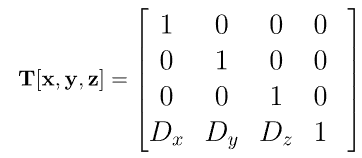
**Theory:**

**3-D Transformation:** In very general terms a 3D model is a mathematical representation of a physical entity that occupies space. In more practical terms, a 3D model is made of a description of its shape and a description of its color appearance.3-D Transformation is the process of manipulating the view of a three-D object with respect to its original position by modifying its physical attributes through various methods of transformation like Translation, Scaling, Rotation, Shear, etc.

**Properties of 3-D Transformation:**

* Lines are preserved,
* Parallelism is preserved,
* Proportional distances are preserved.

**Translation:**It is the process of changing the relative location of a 3-D object with respect to the original position by changing its coordinates. Translation transformation matrix in the 3-D image is shown as –



Where Dx, Dy, Dz are the translation distances.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d.art3d import Poly3DCollection

def get\_transformation\_choice():

    print("Choose transformation:")

    print("1. Translation")

    print("2. Rotation")

    print("3. Scaling")

    choice = int(input("Enter choice (1/2/3): "))

    return choice

def get\_parameters(choice):

    if choice == 1:

        tx = float(input("Enter translation along x: "))

        ty = float(input("Enter translation along y: "))

        tz = float(input("Enter translation along z: "))

        return (tx, ty, tz)

    elif choice == 2:

        axis = input("Enter axis of rotation (x/y): ").lower()

        angle = float(input("Enter angle of rotation (in degrees): "))

        return (axis, angle)

    elif choice == 3:

        sx = float(input("Enter scaling factor along x: "))

        sy = float(input("Enter scaling factor along y: "))

        sz = float(input("Enter scaling factor along z: "))

        return (sx, sy, sz)

def translate(vertices, tx, ty, tz):

    translation\_matrix = np.array([[1, 0, 0, tx],

                                   [0, 1, 0, ty],

                                   [0, 0, 1, tz],

                                   [0, 0, 0, 1]])

    vertices = np.hstack((vertices, np.ones((vertices.shape[0], 1))))

    transformed\_vertices = vertices.dot(translation\_matrix.T)

    return transformed\_vertices[:, :3]

def rotate(vertices, axis, angle):

    angle = np.radians(angle)

    if axis == 'x':

        rotation\_matrix = np.array([[1, 0, 0, 0],

                                    [0, np.cos(angle), -np.sin(angle), 0],

                                    [0, np.sin(angle), np.cos(angle), 0],

                                    [0, 0, 0, 1]])

    elif axis == 'y':

        rotation\_matrix = np.array([[np.cos(angle), 0, np.sin(angle), 0],

                                    [0, 1, 0, 0],

                                    [-np.sin(angle), 0, np.cos(angle), 0],

                                    [0, 0, 0, 1]])

    vertices = np.hstack((vertices, np.ones((vertices.shape[0], 1))))

    transformed\_vertices = vertices.dot(rotation\_matrix.T)

    return transformed\_vertices[:, :3]

def scale(vertices, sx, sy, sz):

    scaling\_matrix = np.array([[sx, 0, 0, 0],

                               [0, sy, 0, 0],

                               [0, 0, sz, 0],

                               [0, 0, 0, 1]])

    vertices = np.hstack((vertices, np.ones((vertices.shape[0], 1))))

    transformed\_vertices = vertices.dot(scaling\_matrix.T)

    return transformed\_vertices[:, :3]

def display\_object(original\_vertices, transformed\_vertices, faces):

    fig = plt.figure()

    ax = fig.add\_subplot(111, projection='3d')

    # Original object

    poly3d\_original = [[original\_vertices[vertice] for vertice in face] for face in faces]

    ax.add\_collection3d(Poly3DCollection(poly3d\_original, facecolors='cyan', linewidths=1, edgecolors='r', alpha=.25))

    # Transformed object

    poly3d\_transformed = [[transformed\_vertices[vertice] for vertice in face] for face in faces]

    ax.add\_collection3d(Poly3DCollection(poly3d\_transformed, facecolors='magenta', linewidths=1, edgecolors='b', alpha=.25))

    ax.set\_xlabel('X')

    ax.set\_ylabel('Y')

    ax.set\_zlabel('Z')

    plt.show()

# Example usage

vertices = np.array([[0, 0, 0],

                     [1, 0, 0],

                     [1, 1, 0],

                     [0, 1, 0],

                     [0, 0, 1],

                     [1, 0, 1],

                     [1, 1, 1],

                     [0, 1, 1]])

faces = [[0, 1, 2, 3],

         [4, 5, 6, 7],

         [0, 1, 5, 4],

         [2, 3, 7, 6],

         [0, 3, 7, 4],

         [1, 2, 6, 5]]

choice = get\_transformation\_choice()

params = get\_parameters(choice)

if choice == 1:

    tx, ty, tz = params

    transformed\_vertices = translate(vertices, tx, ty, tz)

elif choice == 2:

    axis, angle = params

    transformed\_vertices = rotate(vertices, axis, angle)

elif choice == 3:

    sx, sy, sz = params

    transformed\_vertices = scale(vertices, sx, sy, sz)

display\_object(vertices, transformed\_vertices, faces)

**Output:**

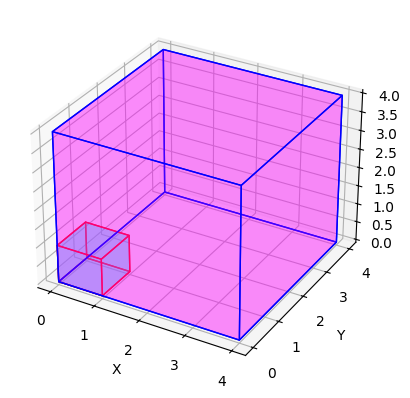


Fig – Output Scaling

A graph of a box with red and blue boxes

Description automatically generated

Fig – Output Translation

A diagram of a cube

Description automatically generated

Fig – Output Rotation along x – axis

A graph of a cube with a red line

Description automatically generated

Fig – Output Rotation along y – axis