Final Lab Report

Course Title: Computer Graphics Laboratory

Course Code: CSE-304

Submitted To

Dr. Morium Akter

Professor

Department of Computer Science and Engineering

Jahangirnagar University

Savar, Dhaka-1342

Submitted By

Oywon Islam

Roll No: 370

Exam Roll: 220430

Session: 2021-2022

Date of submission: 10.10.2025

Lab 01:

Experiment Title: Color Model Conversion: RGB, CMY, and Grayscale

Objective: The purpose of this lab is to understand and implement color space conversions in computer graphics by creating C++ programs to:

1. Convert RGB colors to CMY.
2. Convert CMY colors to RGB.
3. Convert RGB colors to Grayscale (Black & White).

Theory: Color spaces are ways to represent colors in computer graphics. This lab focuses on three conversions:

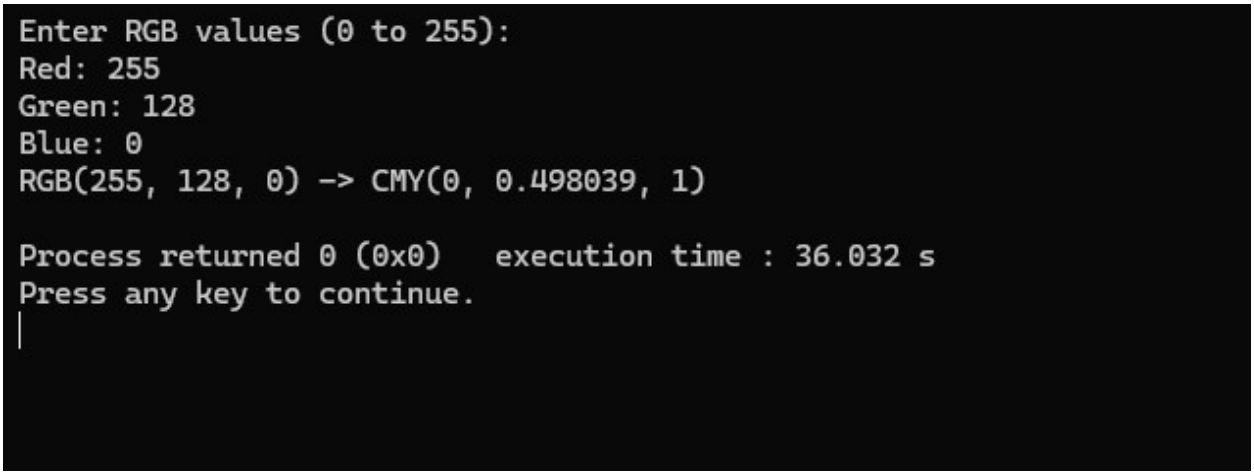
1. RGB to CMY:
    a. RGB (Red, Green, Blue) is used for digital displays, with values from 0 to 255.
    b. CMY (Cyan, Magenta, Yellow) is used for printing, with values from 0 to 1.
    c. Formula: C = 1 - (R/255), M = 1 - (G/255), Y = 1 - (B/255).
    d. Example: RGB(255, 128, 0) becomes CMY(0, 0.498, 1).
2. CMY to RGB:
    a. Reverses the RGB to CMY process.
    b. Formula: R = (1 - C) * 255, G = (1 - M) * 255, B = (1 - Y) * 255.
    c. Example: CMY(0, 0.5, 1) becomes RGB(255, 127, 0).
3. RGB to Grayscale:
    a. Converts RGB to a single gray value for black-and-white images.
    b. Uses a weighted average since human eyes are more sensitive to green: Gray = 0.3 * (R/255) + 0.6 * (G/255) + 0.1 * (B/255).
    c. The gray value is scaled to 0-255 and applied to all RGB channels.
    d. Example: RGB(255, 128, 0) becomes Grayscale(153, 153, 153).

Task 1: RGB to CMY

C++ Code:

```
#include <iostream>
using namespace std;

void rgbToCmy(float r, float g, float b) {
// Convert to 0-1 range and subtract from 1
    float c = 1.0 - (r / 255.0);
float m = 1.0 - (g / 255.0);
    float y = 1.0 - (b / 255.0);

    // Show result
    cout << "RGB(" << r << ", " << g << ", " << b << ") -> CMY("
<< c << ", " << m << ", " << y << ")\n";
}

int main() {
    float r, g, b;
    cout << "Enter RGB values (0 to 255):\n";
    cout << "Red: "; cin >> r;
cout << "Green: "; cin >> g;
    cout << "Blue: "; cin >> b;

    // Check if values are valid
    if (r < 0 || r > 255 || g < 0 || g > 255 || b < 0 || b > 255)
{       cout << "Error: Values must be 0 to 255!\n";
return 1;
    }

    rgbToCmy(r, g, b);
    return 0;
}
```

Output :


Enter RGB values (0 to 255):
Red: 255
Green: 128
Blue: 0
RGB(255, 128, 0) -> CMY(0, 0.498039, 1)
Process returned 0 (0x0)   execution time : 36.032 s
Press any key to continue.

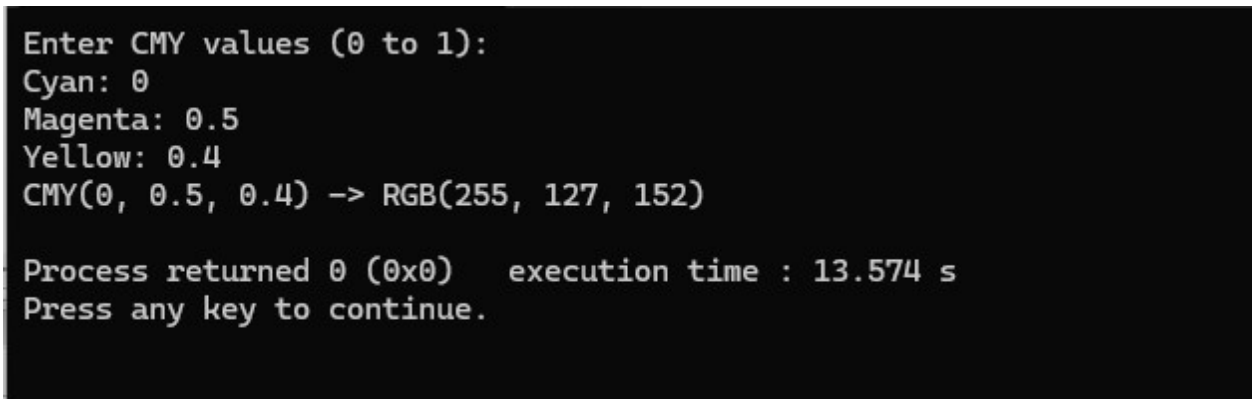
Task 2: CMY to RGB

C++ Code:

```
#include <iostream>
using namespace std;

void cmyToRgb(float c, float m, float y) {
   // Convert to RGB
int r = (1.0 - c) * 255;
int g = (1.0 - m) * 255;
   int b = (1.0 - y) * 255;

   // Show result
   cout << "CMY(" << c << ", " << m << ", " << y << ") -> RGB("
```

```
        << r << ", " << g << ", " << b << ")\n";
}

int main() {     float c, m, y;     cout <<
"Enter CMY values (0 to 1):\n";     cout
<< "Cyan: "; cin >> c;     cout <<
"Magenta: "; cin >> m;
   cout << "Yellow: "; cin >> y;

   // Check if values are valid     if (c < 0 || c > 1 ||
m < 0 || m > 1 || y < 0 || y > 1) {         cout <<
"Error: Values must be 0 to 1!\n";         return 1;
   }

   cmyToRgb(c, m, y);
return 0;
}
```

Output :


Enter CMY values (0 to 1):
Cyan: 0
Magenta: 0.5
Yellow: 0.4
CMY(0, 0.5, 0.4) -> RGB(255, 127, 152)
Process returned 0 (0x0) execution time : 13.574 s
Press any key to continue.

Task 3: RGB to Black & White (Grayscale)

C++ Code:

```
#include using namespace std;

void rgbToGrayscale(float r, float g, float b) { // Simple grayscale formula float gray = 0.3
* (r / 255.0) + 0.6 * (g / 255.0) + 0.1 * (b / 255.0); int grayValue = gray * 255; // Convert
back to 0-255

// Show result cout << "RGB(" << r << ", " << g << ", " << b << ")
-> Grayscale("
    << grayValue << ", " << grayValue << ", " << grayValue << ")\n";



}

int main() { float r, g, b; cout << "Enter RGB values (0 to 255):\n"; cout << "Red: "; cin
>> r; cout << "Green: "; cin >> g; cout << "Blue: "; cin >> b;

// Check if values are valid
if (r < 0 || r > 255 || g < 0 || g > 255 || b < 0 || b > 255)
{    cout << "Error: Values must be 0 to 255!\n";
return 1;
}

rgbToGrayscale(r, g, b);
return 0;



}
```

Output Screenshot:

Lab 02:

Experiment Title:Julia Set and Mandelbrot Set Visualization

Objective: The purpose of this lab is to explore fractals by implementing C++ programs to generate:

1. Julia Set
2. Mandelbrot Set The fractals are displayed as ASCII art in the console for simplicity.

Theory: Fractals are shapes that repeat at different scales, created using complex numbers. This lab implements two fractals:

1. Julia Set:
    a. Uses the formula: z(n+1) = z(n)^2 + c, where c is a fixed complex number (e.g., -0.4 + 0.6i).
    b. Each point in a grid is mapped to a complex number z.
    c. Iterate up to 20 times. If |z| stays below 2, the point is in the set (print '*'). If it grows large, it's outside (print space).
    d. The result is a unique, self-similar pattern.
2. Mandelbrot Set:
    a. Uses the same formula: z(n+1) = z(n)^2 + c, but c is the point's complex coordinate, and z starts at 0.
    b. Points that don't escape after 20 iterations are in the set (print '*'), others are outside (print space).
    c. The set forms a cardioid with circular patterns.

The fractals are shown in a 40x20 ASCII grid, where '*' represents points inside the set.

Task 1 Julia Set:

C++ Code:

```
#include <iostream>
#include <complex> using
namespace std;
```

```
int main() {     float real, imag;     cout << "Enter real and
imaginary parts for Julia constant (c): ";     cin >> real >> imag;

   complex<float> c(real, imag);

   for (float y = -1.5; y <= 1.5; y += 0.1) {
for (float x = -1.5; x <= 1.5; x += 0.05) {
complex<float> z(x, y);            int i = 0;
while (abs(z) < 2 && i < 20) {
            z = z * z + c;
            i++;
         }
         cout << (i == 20 ? "*" : " ");
      }
      cout << endl;
   }

   return 0;
}
```

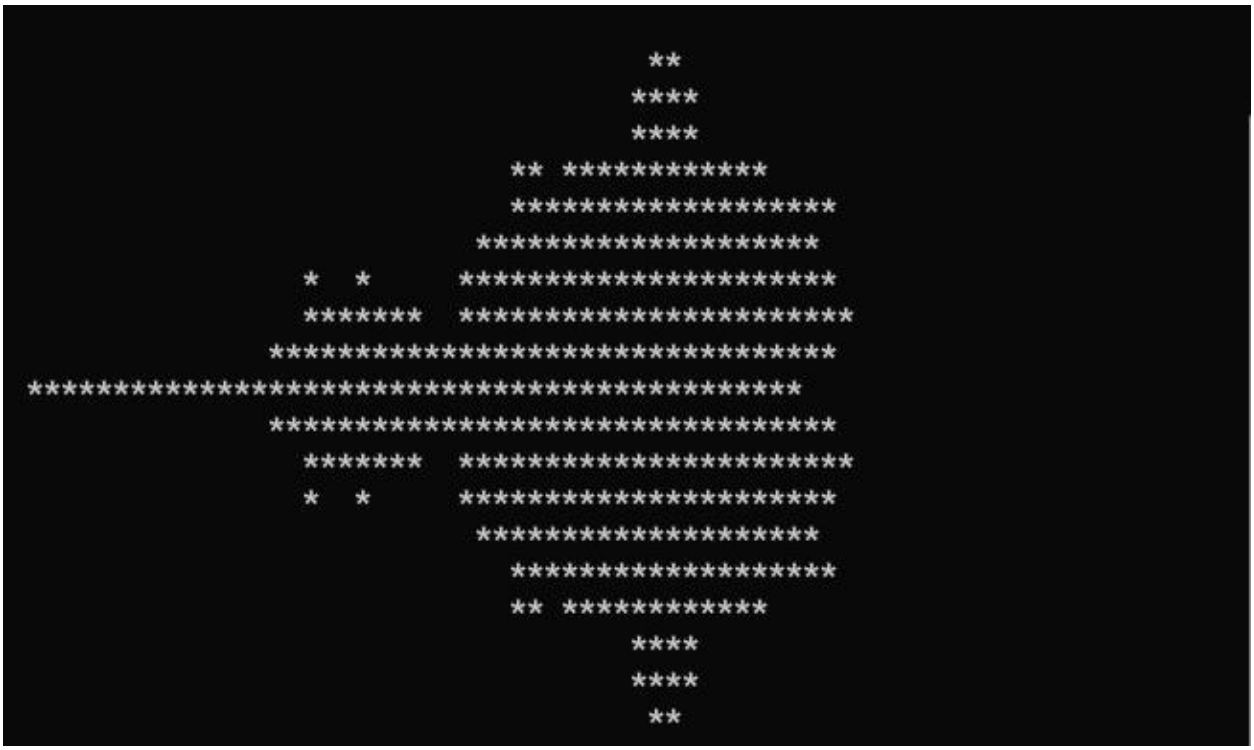
Output Screenshot:

Task 2: Mandelbrot Set

C++ Code:

```
#include <iostream> #include
<complex>
using namespace std;

int main() {     for (float y = -1.5; y <= 1.5; y
+= 0.1) {        for (float x = -2.0; x <= 1.0; x
+= 0.05) {            complex<float> c(x, y);
complex<float> z(0, 0);            int i = 0;
while (abs(z) < 2 && i < 20) {
            z = z * z + c;
            i++;
          }
          cout << (i == 20 ? "*" : " ");
       }
       cout << endl;
return 0;}
```

Output Screenshot:



LAB-3

Experiment Title: Scan Conversion of Point and Line

Tools: Python (with matplotlib)

1 Scan Conversion of a Point

Objective:
To plot a single point on the screen at coordinates specified by the user.

Theory:
Scan conversion of a point is the simplest form of rendering in computer graphics. The point (x, y) is mapped to the nearest pixel on the raster grid. The coordinate is provided by the user and directly illuminated in the display.

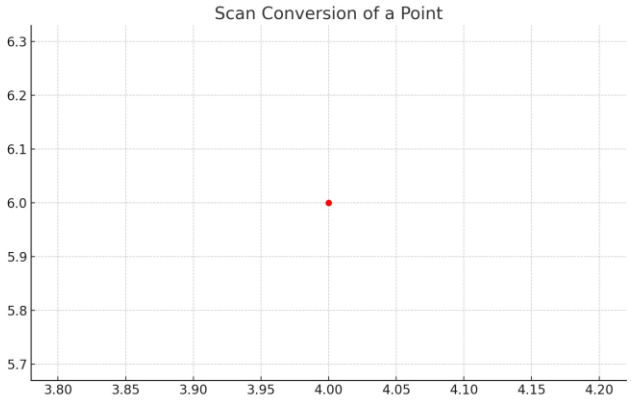
Code (Python):

```
import matplotlib.pyplot as plt

x = int(input("Enter x coordinate of point: "))
y = int(input("Enter y coordinate of point: "))

plt.plot(x, y, 'ro')
plt.title("Scan Conversion of a Point")
plt.grid(True)
plt.show()
```

Output Screenshot :

Input:
Enter x coordinate of point: 4
Enter y coordinate of point: 6

Output:



A single red dot appears at position (4, 6) on the grid.

Observation:

- The point is correctly plotted at the specified location.
- The matplotlib grid helps to visually verify accuracy.
- No computational complexity; direct mapping to raster.

2 Scan Conversion of a Line

Objective:

To draw a line between two points using:
- Line equation
- DDA (Digital Differential Analyzer)
- Bresenham's algorithm

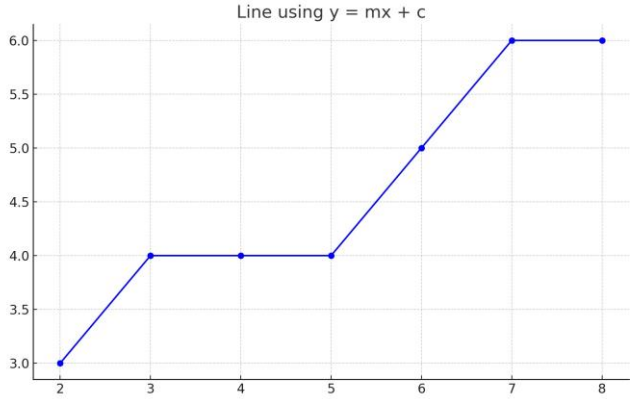
Theory:

Line Equation (y = mx + c): Slope m = (y2 - y1) / (x2 - x1), c = y1 - m * x1.
DDA: Steps = max(|dx|, |dy|), x_inc = dx / steps, y_inc = dy / steps.
Bresenham: Uses integer operations, decision parameter p guides pixel choice.

Code (Python) - Line using Equation:

```
import matplotlib.pyplot as plt

x1 = int(input("Enter x1: "))
y1 = int(input("Enter y1: "))
x2 = int(input("Enter x2: "))
y2 = int(input("Enter y2: "))

m = (y2 - y1) / (x2 - x1)
c = y1 - m * x1

x_values = []
y_values = []

for x in range(x1, x2 + 1):
    y = round(m * x + c)
    x_values.append(x)
    y_values.append(y)

plt.plot(x_values, y_values, 'bo-')
plt.title("Line using y = mx + c")
plt.grid(True)
plt.show()
```

Output :

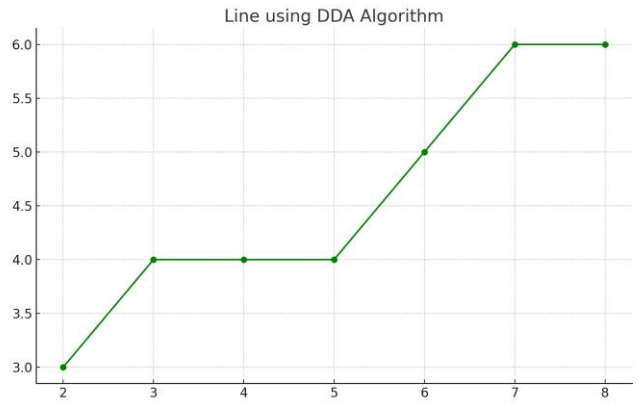


Code (Python) - DDA Algorithm:

```
import matplotlib.pyplot as plt

x1 = int(input("Enter x1: "))
y1 = int(input("Enter y1: "))
x2 = int(input("Enter x2: "))
y2 = int(input("Enter y2: "))

dx = x2 - x1
dy = y2 - y1
steps = max(abs(dx), abs(dy))

x_inc = dx / steps
y_inc = dy / steps

x = x1
y = y1

x_values = []
y_values = []

for i in range(int(steps) + 1):
    x_values.append(round(x))
```

```
    y_values.append(round(y))
x += x_inc
    y += y_inc

plt.plot(x_values, y_values, 'go-')
plt.title("Line using DDA Algorithm")
plt.grid(True)
plt.show()
```

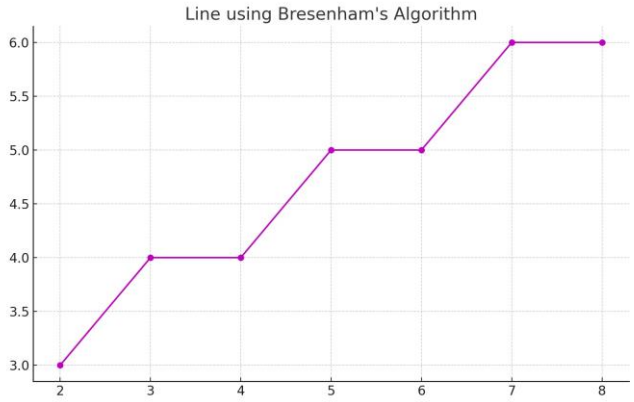
Output :



Code (Python) - Bresenham's Algorithm:

```
import matplotlib.pyplot as plt

x1 = int(input("Enter x1: "))
y1 = int(input("Enter y1: "))
x2 = int(input("Enter x2: "))
y2 = int(input("Enter y2: "))

x, y = x1, y1
```

```
dx = x2 - x1
dy = y2 - y1

x_values = [x]
y_values = [y]

p = 2 * dy - dx

while x < x2:
    x += 1
    if p < 0:
        p += 2 * dy
else:
        y += 1
        p += 2 * (dy - dx)
    x_values.append(x)
    y_values.append(y)

plt.plot(x_values, y_values, 'mo-')
plt.title("Line using Bresenham's Algorithm")
plt.grid(True)
plt.show()
```



Observation:

- All methods generate the same set of points visually.
- Equation method may suffer rounding inaccuracies.
- DDA uses floating-point calculations, slightly costlier.
- Bresenham is most efficient due to integer arithmetic.

LAB-4:

Experiment Title: Scan Conversion of Bresenham's Circle, Midpoint Circle, and Midpoint Ellipse Algorithms with Comparative Analysis

Table of Contents

Objectives

- Implement scan conversion of a circle using Bresenham's Circle Algorithm.
- Implement scan conversion of a circle using the Midpoint Circle Algorithm. - Compare the results of the two circle algorithms in terms of accuracy and efficiency.
- Implement scan conversion of an ellipse using the Midpoint Ellipse Algorithm.

Introduction

Scan conversion is a fundamental process in computer graphics where geometric shapes are converted into raster pixels for display. Circles and ellipses are common shapes, and efficient algorithms like Bresenham's Circle, Midpoint Circle, and Midpoint Ellipse algorithms help render these shapes with high accuracy while minimizing computational overhead. This report covers the implementation and comparison of these algorithms.

Task –1: Scan Conversion Using Bresenham's Circle Algorithm

Theory

Bresenham's Circle Algorithm calculates points for one-eighth of a circle using an integer-based decision parameter and reflects them to draw the entire circle.
It avoids floating-point calculations, increasing performance and precision.

Initial decision parameter:
$d = 3 - 2r$

Update rules:
If $d < 0$, then $d = d + 4x + 6$

Else d = d + 4(x - y) + 10, y = y - 1
In both cases, x = x + 1

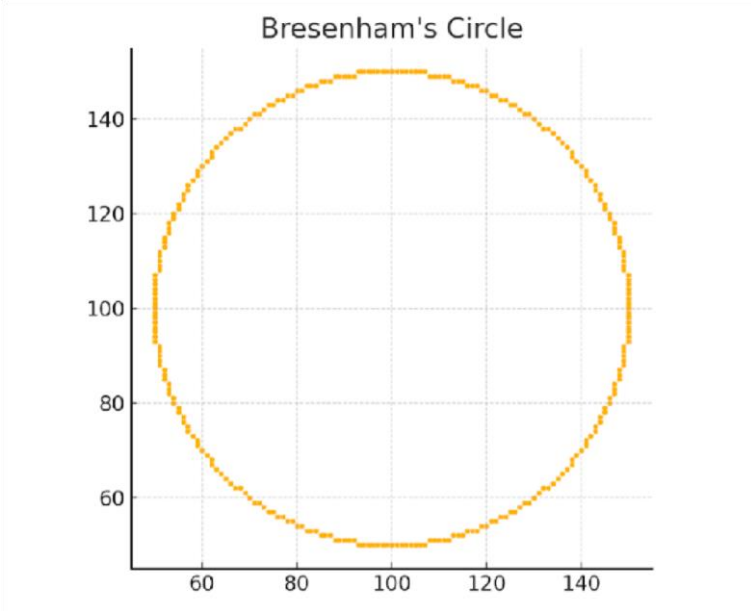
Python Implementation

```
import matplotlib.pyplot as plt

def plot_circle_points(xc, yc, x, y):
    return [(xc+x, yc+y), (xc-x, yc+y), (xc+x, yc-y), (xc-
x,
yc-y),
            (xc+y, yc+x), (xc-y, yc+x), (xc+y, yc-x), (xc-
y,
yc-x)]

def bresenham_circle(xc, yc, r):
    x, y = 0, r
    d = 3 - 2 * r
    points = []
    while x <= y:
        points += plot_circle_points(xc, yc, x, y)
        if d < 0:
            d += 4 * x + 6
        else:
            d += 4 * (x - y) + 10
            y -= 1
        x += 1
    return points

xc = int(input("Enter center x: "))
yc = int(input("Enter center y: "))
r = int(input("Enter radius: "))

points = bresenham_circle(xc, yc, r)
x_vals, y_vals = zip(*points)
```

```
plt.scatter(x_vals, y_vals, s=5)
plt.title("Scan Conversion: Bresenham's Circle")
plt.gca().set_aspect('equal')
plt.grid(True)
plt.show()
```

Sample Input:
Center: (100, 100)
Radius: 50

Output Screenshot:



Observation:
- Efficient integer arithmetic.
- Produces a smooth, symmetric circle.

Task-2 : Scan Conversion Using Midpoint Circle Algorithm

Theory

The Midpoint Circle Algorithm also uses symmetry and an integer-based midpoint decision parameter to plot pixels closest to the ideal circle.

Initial decision parameter:
$d = 1 - r$

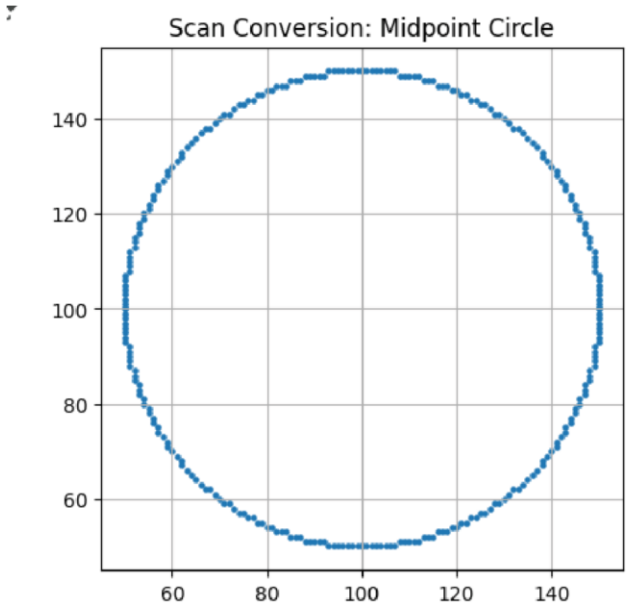
Update rules:
If $d < 0$, then $d = d + 2x + 3$ Else
$d = d + 2(x - y) + 5$, $y = y - 1$
Always increment x.

Python Implementation

```python
import matplotlib.pyplot as plt

def plot_circle_mid_points(xc, yc, x, y):
    return [(xc+x, yc+y), (xc-x, yc+y), (xc+x, yc-y), (xc-x, yc-y),
            (xc+y, yc+x), (xc-y, yc+x), (xc+y, yc-x), (xc-y, yc-x)]

def midpoint_circle(xc, yc, r):
    x, y = 0, r
    d = 1 - r
    points = []
    while x <= y:
        points += plot_circle_mid_points(xc, yc, x, y)
        if d < 0:
            d += 2 * x + 3
        else:
            d += 2 * (x - y) + 5
            y -= 1
        x += 1
    return points

xc = int(input("Enter center x: "))
yc = int(input("Enter center y: "))
r = int(input("Enter radius: "))

points = midpoint_circle(xc, yc, r)
x_vals, y_vals = zip(*points)
plt.scatter(x_vals, y_vals, s=5)
plt.title("Scan Conversion: Midpoint Circle")
plt.gca().set_aspect('equal')
plt.grid(True)
plt.show()
```

Sample Input:
Center: (100, 100)
Radius: 50

Output Screenshot:



Observation:

- Similar output to Bresenham's algorithm.
- Midpoint decision logic.
- Slightly simpler calculations.

Comparison Between Bresenham's and Midpoint Circle Algorithms

Feature	Bresenham Circle	Midpoint Circle
Type of Arithmetic	Integer-only	Integer-only
Decision Logic	More optimized	Midpoint-based
Output Appearance	Identical	Identical

Symmetry Used	8-way	8-way
Efficiency	Very High	High
Practical Use	Hardware rasterization	Teaching, software apps

Task –3 Scan Conversion Using Midpoint Ellipse Algorithm

Theory

The Midpoint Ellipse Algorithm plots ellipses by dividing the process into two regions based on slope and using different decision parameters for each.

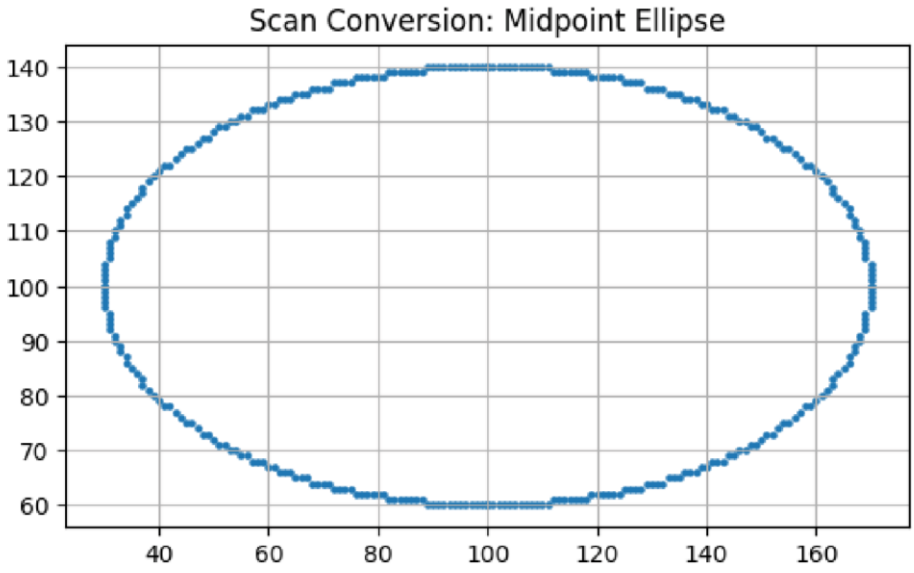
Python Implementation

```
import matplotlib.pyplot as plt

def midpoint_ellipse(xc, yc, a, b):
    x, y = 0, b
    a2, b2 = a*a, b*b
    dx, dy = 2*b2*x, 2*a2*y
    d1 = b2 - a2 * b + 0.25 * a2
    points = []

    while dx < dy:
        points += [(xc+x, yc+y), (xc-x, yc+y), (xc+x, yc-y), (xc-x, yc-y)]
        if d1 < 0:
            x += 1
            dx += 2*b2
            d1 += dx + b2
        else:
            x += 1
            y -= 1
            dx += 2*b2
            dy -= 2*a2
            d1 += dx - dy + b2

    d2 = b2*(x + 0.5)**2 + a2*(y - 1)**2 - a2*b2
    while y >= 0:
        points += [(xc+x, yc+y), (xc-x, yc+y), (xc+x, yc-y), (xc-x, yc-y)]
        if d2 > 0:
            y -= 1
            dy -= 2*a2
```

```
            d2 += a2 - dy
        else:
            y -= 1
            x += 1
            dx += 2*b2
            dy -= 2*a2
            d2 += dx - dy + a2
    return points

xc = int(input("Enter center x: "))
yc = int(input("Enter center y: "))
a = int(input("Enter semi-major axis a: "))
b = int(input("Enter semi-minor axis b: "))

points = midpoint_ellipse(xc, yc, a, b)
x_vals, y_vals = zip(*points)
plt.scatter(x_vals, y_vals, s=5)
plt.title("Scan Conversion: Midpoint Ellipse")
plt.gca().set_aspect('equal')
plt.grid(True)
plt.show()
```

Sample Input:
Center: (100, 100)
Semi-major axis: 70
Semi-minor axis: 40

Output Screenshot:



Observation:
- Produces smooth, symmetric ellipses.
- Efficient region-based decision logic.

Discussion

Both circle algorithms produce visually identical and smooth circles using integer calculations, avoiding floating-point overhead.
Bresenham's algorithm is slightly more optimized, making it preferred for hardware implementations.
Midpoint algorithm is intuitive and widely used in teaching and software rendering.
The ellipse algorithm extends the midpoint approach to handle two regions separately, ensuring accuracy and symmetry in elliptical shapes.

Conclusion

This lab successfully implemented and compared scan conversion algorithms for circles and ellipses. Bresenham's Circle and Midpoint Circle algorithms both efficiently rasterize circles, with minor differences in performance. The Midpoint Ellipse algorithm accurately renders ellipses by handling different

slope regions. These foundational algorithms are crucial for efficient shape rendering in computer graphics.

LAB-5

Experiment Title:2D Geometric Transformations: Translation, Rotation, and Scaling.

Objective

To implement and visualize 2D geometric transformations — Translation, Rotation, and Scaling — on a single point using Python and Matplotlib. The objective is to understand how coordinate changes affect object placement and shape in 2D space.

1. Translation

Theory

Translation moves a point in 2D space by shifting it horizontally and/or vertically.

Let a point be P(x, y) and the translation factors be Tx and Ty.

The translated point P' is calculated as:

x' = x + Tx y' = y + Ty

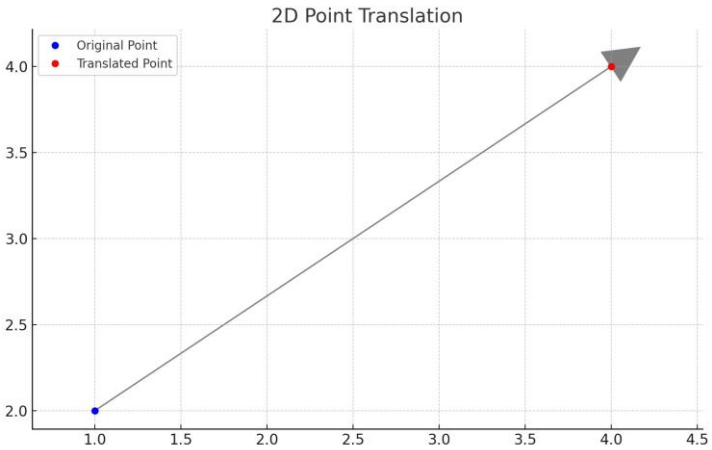
This transformation does not affect the size, shape, or orientation.

Code Snippet

```
import matplotlib.pyplot as plt



# Function to translate a point

def translate_point(x, y, tx, ty):

  return x + tx, y + ty
```

```
# Input
x = float(input("Enter original x: "))
y = float(input("Enter original y: "))

tx = float(input("Enter translation Tx: "))
ty = float(input("Enter translation Ty: "))



# Calculate translated point
new_x, new_y = translate_point(x, y, tx, ty)



# Plot original and translated points
plt.plot(x, y, 'bo', label="Original Point")
plt.plot(new_x, new_y, 'ro', label="Translated Point")



# Draw simple arrow from original to translated
plt.arrow(x, y, tx, ty, head_width=0.2, head_length=0.2, color='gray')



# Show plot
plt.axis('equal')

plt.grid(True)
plt.title("2D Point Translation")
plt.legend()
plt.show()
```

Output



Observation

The point moves based on Tx and Ty without changing size or shape.

2. Rotation

Theory

Rotation transforms a point around the origin by a certain angle θ in counter-clockwise direction.

Let a point be P(x, y) and the angle be θ (in degrees).
Converted to radians: $\theta\_rad = \theta \times \pi / 180$

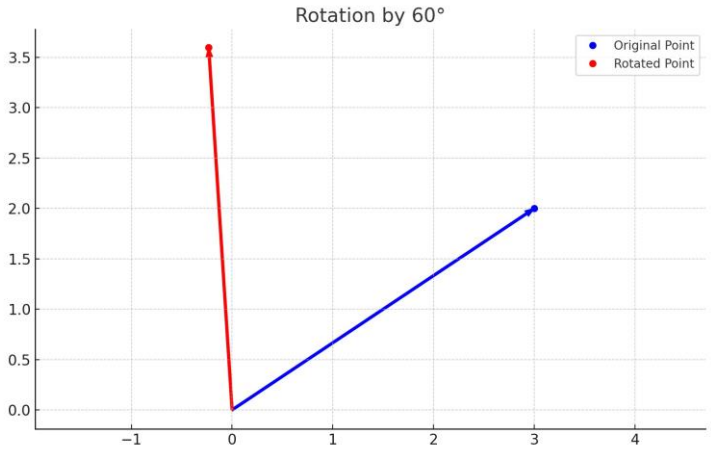
The rotated point P' is:
$x' = x * \cos(\theta) - y * \sin(\theta)$ $y' = x * \sin(\theta) + y * \cos(\theta)$
Rotation keeps the distance from the origin the same.

Code Snippet

```
import math

import matplotlib.pyplot as plt



# Function to rotate a point around origin
```

```
def rotate_point(x, y, angle_deg):

   angle_rad = math.radians(angle_deg)

   new_x = x * math.cos(angle_rad) - y * math.sin(angle_rad)

   new_y = x * math.sin(angle_rad) + y * math.cos(angle_rad)
return new_x, new_y




# Input

x = float(input("Enter x: "))
y = float(input("Enter y: "))


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



# Rotate

new_x, new_y = rotate_point(x, y, angle)
print(f"Rotated Point: ({new_x:.2f}, {new_y:.2f})")




# Plot

plt.plot(x, y, 'bo', label="Original Point")
plt.plot(new_x, new_y, 'ro', label="Rotated Point")


plt.arrow(0, 0, x, y, color='blue', width=0.02, length_includes_head=True)

plt.arrow(0, 0, new_x, new_y, color='red', width=0.02, length_includes_head=True)



plt.grid(True)

plt.axis('equal')

plt.title(f"Rotation by {angle}°")
plt.legend()


plt.show()
```

Output



Observation

The point rotates about the origin. Orientation changes, distance remains same.

3. Scaling

Theory

Scaling modifies the position of a point relative to the origin by stretching or shrinking it.

Let a point be P(x, y) and the scaling factors be Sx and Sy.

The scaled point P' is calculated as:

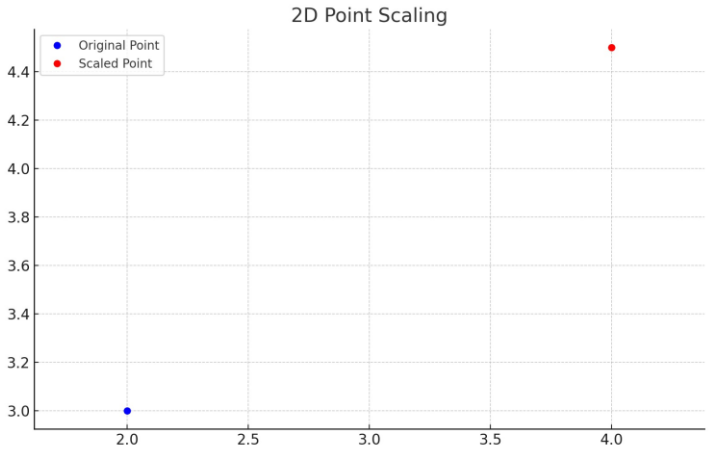
x' = x * Sx y' = y * Sy

If Sx = Sy, it’s uniform scaling. Otherwise, it’s differential scaling.

Code Snippet

```
import matplotlib.pyplot as plt



# Function to scale a point

def scale_point(x, y, sx, sy):

  return x * sx, y * sy



# Input

x = float(input("Enter x: "))

y = float(input("Enter y: ")) sx =
float(input("Enter scaling factor in x (Sx): "))


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



# Calculate scaled point

new_x, new_y = scale_point(x, y, sx, sy)
print(f"Scaled Point: ({new_x}, {new_y})")




# Plot original and scaled points

plt.plot(x, y, 'bo', label="Original Point")
plt.plot(new_x, new_y, 'ro', label="Scaled Point")


plt.grid(True)

plt.axis('equal')

plt.title("2D Point Scaling")

plt.legend()
plt.show()
```

Output:



Observation

Scaling affects distance from origin. Equal scaling keeps shape, unequal distorts.

Conclusion

This lab demonstrated 2D geometric transformations using Python: Translation, Rotation, and Scaling. Translation shifts point position, Rotation changes its orientation around the origin, and Scaling alters its size relative to the origin. These operations are fundamental in computer graphics.

LAB-6

Experiment Title: Advanced 2D Geometric Transformations

Objective

To understand and implement advanced 2D geometric transformations such as Scaling (with respect to origin), Mirror Reflection (about axes), Inverse Coordinate Transformation, and Composite Transformation using Python. The experiment aims to visualize how transformation matrices manipulate the position and orientation of 2D points.

Theory

1. Scaling with Respect to Origin:

Scaling changes the size of an object relative to the origin. Given a point (x, y) and scaling factors Sx and Sy, the transformed coordinates are:

x' = x *
Sx y' = y
* Sy

If Sx = Sy, it's uniform scaling; otherwise, it's differential scaling.

2. Mirror Reflection:

Reflection creates a mirror image of an object about an axis.

- About X-axis: (x', y') = (x, -y)
- About Y-axis: (x', y') = (-x, y)
- About Origin: (x', y') = (-x, -y)

3. Inverse Coordinate System:

An inverse transformation reverses the effect of a previous transformation. For scaling, rotation, or translation, applying the inverse matrix restores the original coordinates.

4. Composite Transformation:

Multiple transformations can be combined into a single transformation matrix. If T1, T2, T3 are transformations, then composite T = T1 × T2 × T3. The order of multiplication affects the final result.

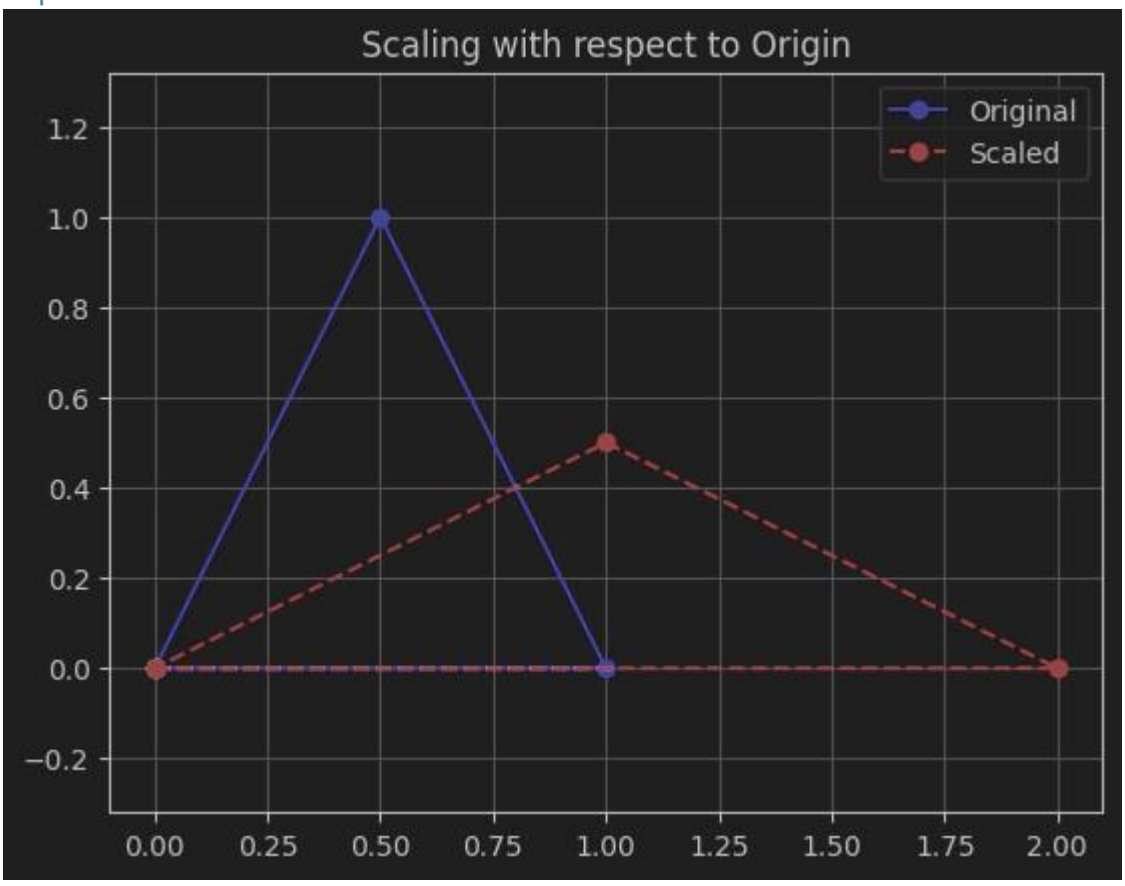
Code Implementation

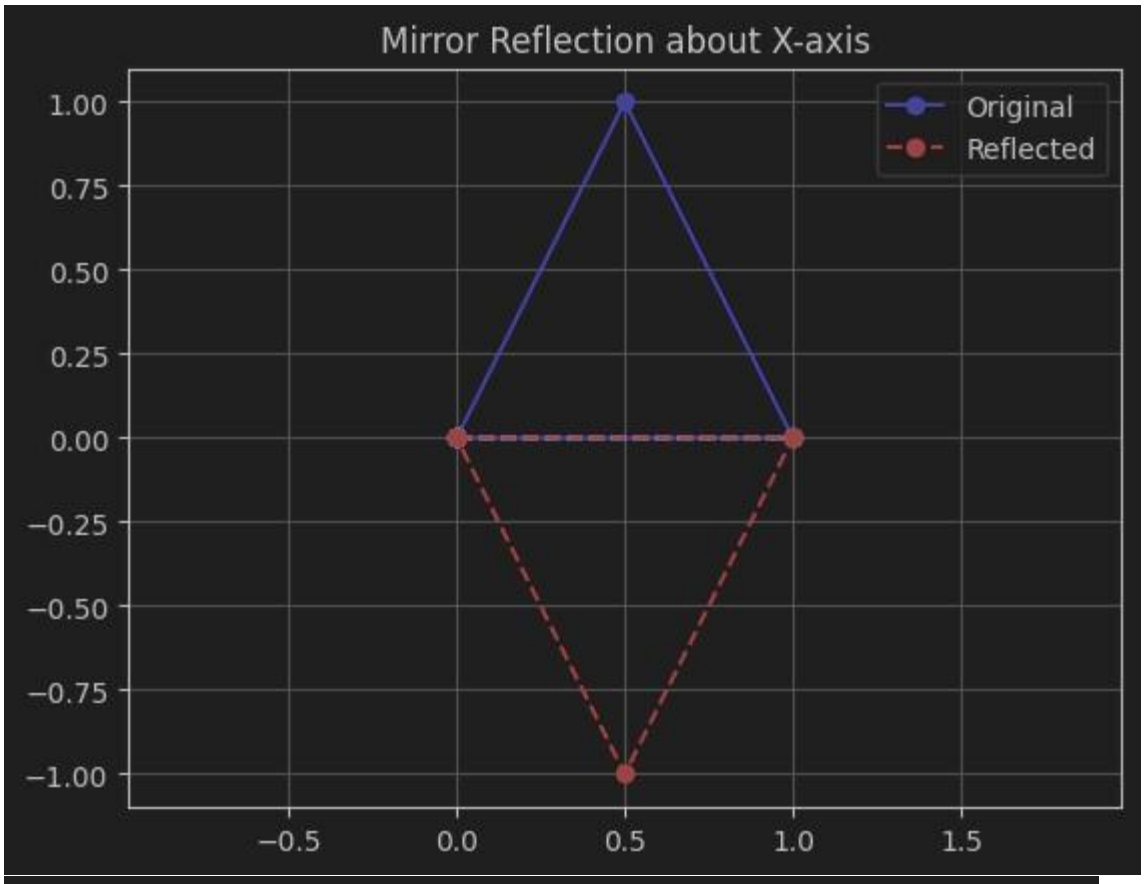
```
import numpy as np
import
matplotlib.pyplot as plt

# Function for
scaling def
scale_point(x, y, sx,
sy):
  return x * sx, y * sy
```

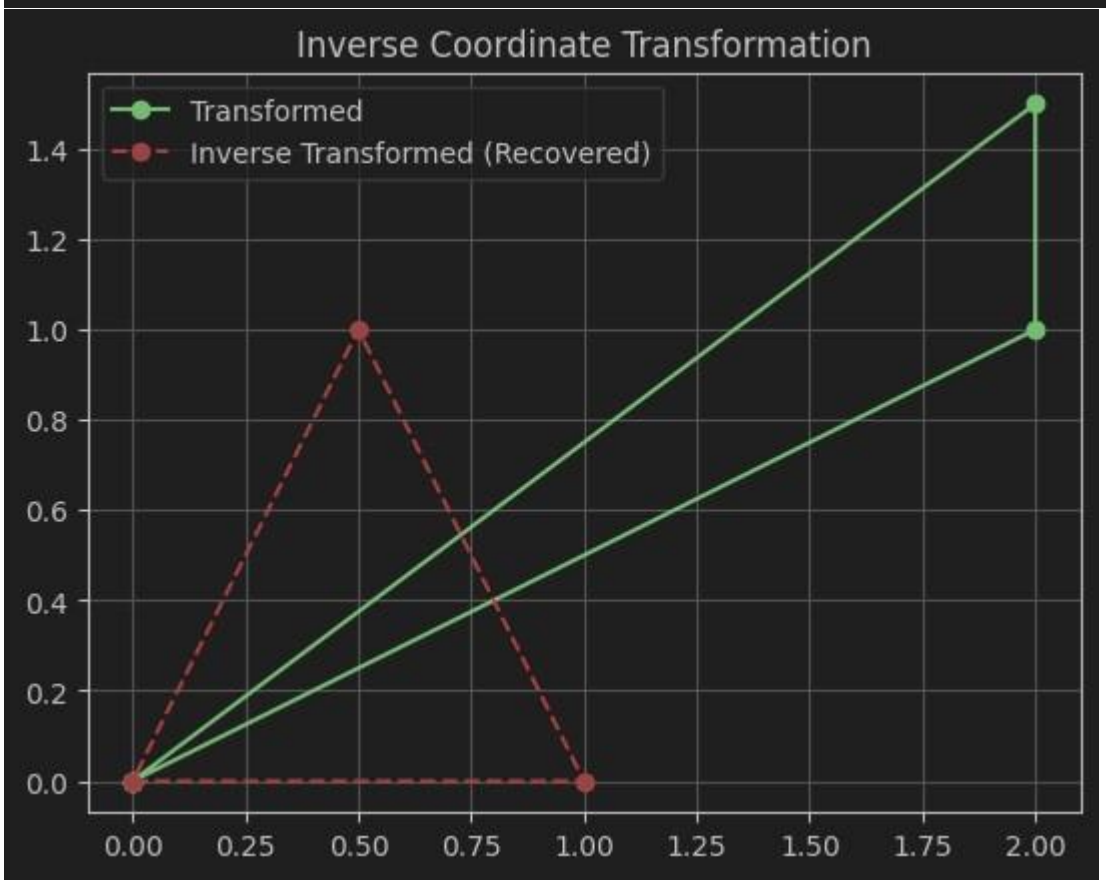
```
# Function for
reflection def
reflect_point(x, y,
axis):    if axis == 'x':
return x, -y    elif
axis == 'y':
return -x, y    elif
axis == 'origin':
      return -x, -y

# Function for inverse
transformation def
inverse_scale(x, y, sx, sy):
   return x / sx, y / sy

# Function for composite transformation (Scaling +
Reflection) def composite_transform(x, y, sx, sy,
axis):
   x1, y1 = scale_point(x, y,
sx, sy)    x2, y2 =
reflect_point(x1, y1, axis)
return x2, y2

# Input x = float(input("Enter x: ")) y =
float(input("Enter y: ")) sx =
float(input("Enter scaling factor Sx: "))
sy = float(input("Enter scaling factor
Sy: ")) axis = input("Enter reflection
axis (x/y/origin): ")

# Apply transformations scaled_x, scaled_y =
scale_point(x, y, sx, sy) reflected_x, reflected_y =
reflect_point(x, y, axis) inverse_x, inverse_y =
inverse_scale(scaled_x, scaled_y, sx, sy)
composite_x, composite_y =
composite_transform(x, y, sx, sy, axis)
```

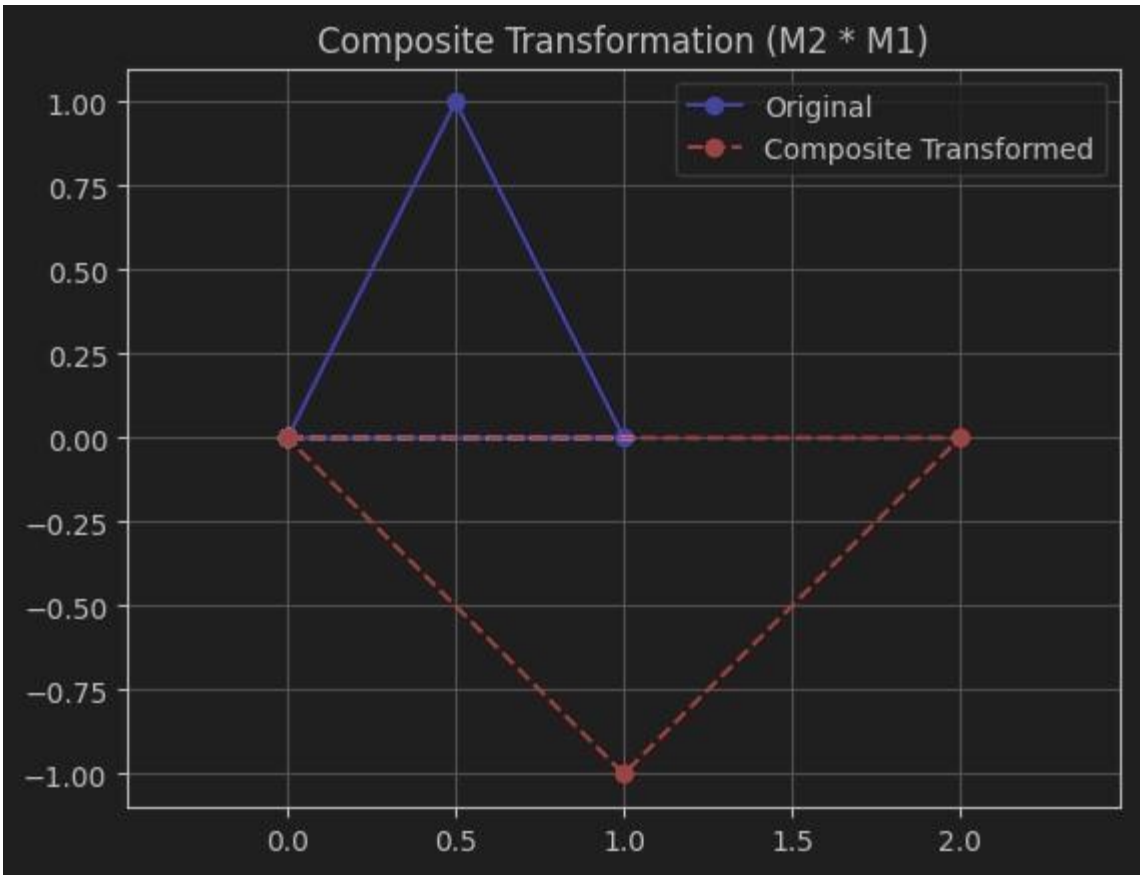
```
# Plot results plt.figure(figsize=(6,6)) plt.plot(x, y, 'bo',
label="Original Point") plt.plot(scaled_x, scaled_y, 'ro',
label="Scaled Point") plt.plot(reflected_x, reflected_y, 'go',
label="Reflected Point") plt.plot(inverse_x, inverse_y, 'yo',
label="Inverse Transformed Point") plt.plot(composite_x,
composite_y, 'mo', label="Composite Transformed Point")

plt.grid(True) plt.axis('equal')
plt.legend() plt.title("Advanced 2D
Geometric Transformations") plt.show()
```

Output Screenshot




Mirror Reflection about X-axis
Original
Reflected
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−0.5
0.0
0.5
1.0
1.5


Inverse Coordinate Transformation
Transformed
Inverse Transformed (Recovered)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00



Observation

1. Scaling changes the distance of a point from the origin based on scaling factors.
2. Reflection flips the point across the specified axis or origin.
3. Inverse scaling restores the original coordinates by applying reciprocal scaling.
4. Composite transformation shows the combined effect of scaling followed by reflection.
5. Transformation matrices provide a powerful way to manipulate objects in 2D space.

Conclusion

The experiment successfully demonstrated advanced 2D transformations including scaling, reflection, inverse, and composite transformations using Python and Matplotlib. By observing point movements and transformations, one can understand how transformation matrices affect the geometric properties of 2D objects.

LAB-7

Experiment Title:**Window to Viewport Mapping, Point Clipping,Line Clipping using Cohen-Sutherland Algorithm and line Clipping using Midpoint Subdivision**

Objective

The objective of this experiment is to implement and demonstrate the transformation of coordinates from a window (world coordinate system) to a viewport (device coordinate system) in computer graphics. This involves mapping a given point from the window boundaries to the corresponding position in the viewport, visualizing the process using graphical representation to understand scaling and translation in 2D graphics.

Theory

window to viewport mapping :

In computer graphics, window to viewport mapping is a crucial transformation technique used to display a portion of the world coordinate system (the "window") onto a physical display area (the "viewport"). The window defines the rectangular region of the scene we wish to view, specified by minimum and maximum x and y coordinates (xw_min, xw_max, yw_min, yw_max). The viewport is the rectangular area on the screen where this window is rendered, defined similarly (xv_min, xv_max, yv_min, yv_max).

The mapping involves scaling and translating points from window coordinates to viewport coordinates. For a point (x, y) in the window:

- Scaling factor in x-direction: sx = (xv_max - xv_min) / (xw_max - xw_min)
- Scaling factor in y-direction: sy = (yv_max - yv_min) / (yw_max - yw_min)
- Translated x in viewport: xv = xv_min + (x - xw_min) * sx
- Translated y in viewport: yv = yv_min + (y - yw_min) * sy

This ensures that the entire window fits proportionally into the viewport, handling aspects like zooming, panning, and aspect ratio preservation. This transformation is fundamental in clipping algorithms (e.g., Cohen-Sutherland) and rendering pipelines.

Point Clipping:

Checks if a point lies inside a rectangular clipping window defined by (xmin ,ymin)(x_{\min}, y_{\min})(xmin ,ymin) and (xmax ,ymax)(x_{\max}, y_{\max})(xmax ,ymax). Points outside are discarded; points inside are displayed.

Line Clipping using Cohen-Sutherland Algorithm:

Assigns region codes to line endpoints and uses bitwise operations to determine whether the line is fully inside, fully outside, or partially inside the window. Partially inside lines are clipped until they fit.

Line Clipping using Midpoint Subdivision:

Recursively divides a line segment at its midpoint and checks which portion lies inside the clipping window. Subdivision continues until the line is entirely inside or outside.

Code:

```
import matplotlib.pyplot as plt

# ------------------------------

# 1. Window to Viewport Mapping

# ------------------------------

def window_to_viewport(xw, yw, wx_min, wy_min, wx_max, wy_max, vx_min, vy_min,
vx_max, vy_max):

    sx = (vx_max - vx_min) / (wx_max - wx_min)

sy = (vy_max - vy_min) / (wy_max - wy_min)

xv = vx_min + (xw - wx_min) * sx     yv =

vy_min + (yw - wy_min) * sy     return xv, yv


# ------------------------------

# 2. Point Clipping

# ------------------------------

def point_clip(x, y, xmin, ymin, xmax, ymax):

    return xmin <= x <= xmax and ymin <= y <= ymax


# ------------------------------

# 3. Line Clipping: Cohen-Sutherland

# ------------------------------

INSIDE, LEFT, RIGHT, BOTTOM, TOP = 0, 1, 2, 4, 8


def compute_code(x, y, xmin, ymin, xmax, ymax):

    code = INSIDE     if x < xmin:

code |= LEFT     elif x > xmax:

code |= RIGHT     if y < ymin:
```

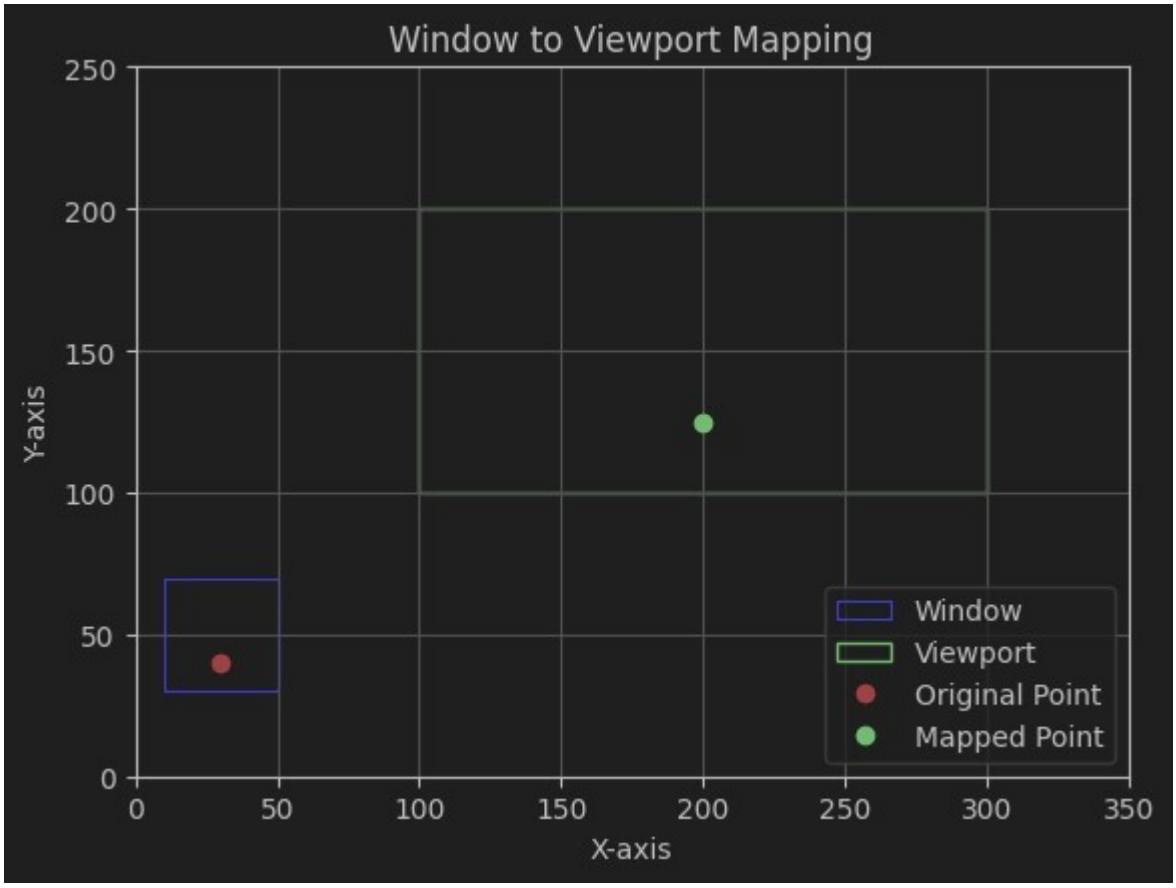
```
code |= BOTTOM     elif y >
ymax: code |= TOP     return
code


def cohen_sutherland_clip(x1, y1, x2, y2, xmin, ymin, xmax, ymax):
   code1 = compute_code(x1, y1, xmin, ymin, xmax, ymax)
code2 = compute_code(x2, y2, xmin, ymin, xmax, ymax)     while
True:
       if code1 == 0 and code2 == 0:
           return [(x1, y1, x2, y2)]
elif code1 & code2 != 0:
           return []
else:
           code_out = code1 if code1 != 0 else code2
if code_out & TOP:
               x = x1 + (x2 - x1) * (ymax - y1) / (y2 - y1)
y = ymax             elif code_out & BOTTOM:
x        = x1 + (x2 - x1) * (ymin - y1) / (y2 - y1)
y = ymin             elif code_out & RIGHT:
y        = y1 + (y2 - y1) * (xmax - x1) / (x2 - x1)
x = xmax             elif code_out & LEFT:
               y = y1 + (y2 - y1) * (xmin - x1) / (x2 - x1)
x = xmin             if code_out == code1:

               x1, y1 = x, y
               code1 = compute_code(x1, y1, xmin, ymin, xmax, ymax)
           else:
```

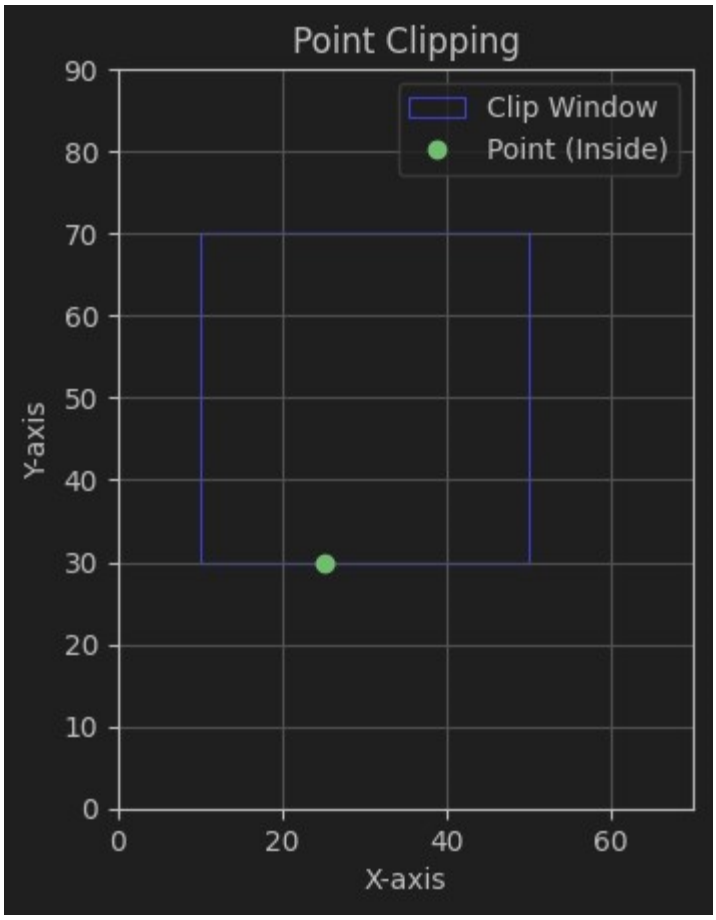
```
            x2, y2 = x, y
            code2 = compute_code(x2, y2, xmin, ymin, xmax, ymax)


# ------------------------------
# 4. Line Clipping: Midpoint Subdivision
# ------------------------------
def midpoint_clip(x1, y1, x2, y2, xmin, ymin, xmax, ymax, tol=0.01):
    if (xmin <= x1 <= xmax and ymin <= y1 <= ymax) and (xmin <= x2 <= xmax and ymin <= y2 <= ymax):
        return [(x1, y1, x2, y2)]
    elif not ((xmin <= x1 <= xmax and ymin <= y1 <= ymax) or (xmin <= x2 <= xmax and ymin <= y2 <= ymax)):
        return []
else:
        xm, ym = (x1 + x2) / 2, (y1 + y2) / 2
if abs(x1 - x2) < tol and abs(y1 - y2) < tol:
            return []
        return midpoint_clip(x1, y1, xm, ym, xmin, ymin, xmax, ymax, tol) + \
midpoint_clip(xm, ym, x2, y2, xmin, ymin, xmax, ymax, tol)


# ------------------------------
# Main: Example Usage
# ------------------------------
if __name__ == "__main__":
    # Window and viewport      wx_min, wy_min, wx_max,
wy_max = 0, 0, 100, 100      vx_min, vy_min, vx_max,
vy_max = 200, 200, 400, 400      xw, yw = 30, 40
```

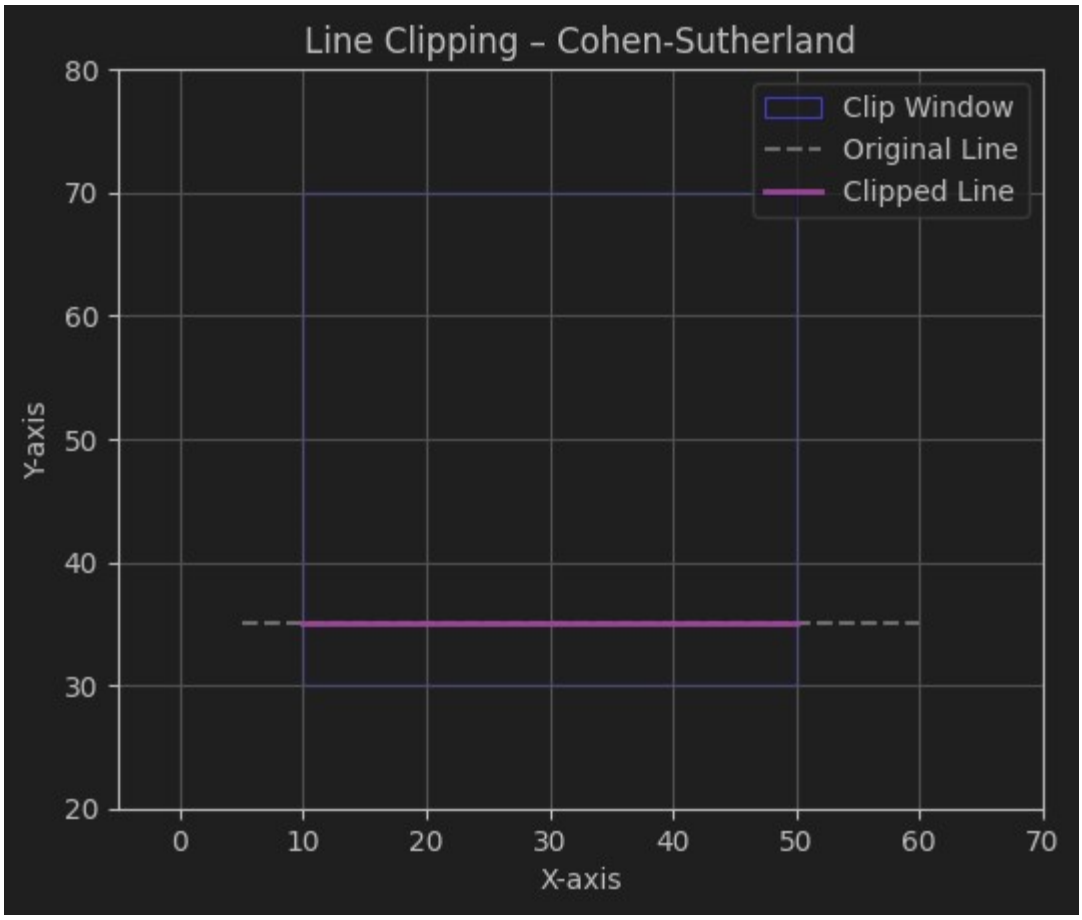
```
    xv, yv = window_to_viewport(xw, yw, wx_min, wy_min, wx_max, wy_max, vx_min,
vy_min, vx_max, vy_max)
    print(f"Viewport coordinates: ({xv}, {yv})")


    # Point clipping      xmin, ymin, xmax,
ymax = 10, 10, 50, 50      points = [(20, 20),
(60, 40), (30, 60)]      for x, y in points:
        print(f"Point ({x},{y}) is {'inside' if point_clip(x,y,xmin,ymin,xmax,ymax) else
'outside'}")


    # Cohen-Sutherland line clipping      line_cs =
cohen_sutherland_clip(5, 5, 40, 60, xmin, ymin, xmax, ymax)
print("Cohen-Sutherland clipped line:", line_cs)


    # Midpoint Subdivision line clipping      line_mp =
midpoint_clip(5, 5, 40, 60, xmin, ymin, xmax, ymax)
print("Midpoint Subdivision clipped line:", line_mp)


    # Visualization      fig,
ax = plt.subplots()
    # Draw clipping window
    rect = plt.Rectangle((xmin, ymin), xmax-xmin, ymax-ymin, edgecolor='black',
facecolor='none', linewidth=2)      ax.add_patch(rect)      # Plot points      for x, y
in points:
        color = 'green' if point_clip(x,y,xmin,ymin,xmax,ymax) else
'red'          ax.plot(x, y, 'o', color=color)      # Plot Cohen-Sutherland
line      for x1, y1, x2, y2 in line_cs:
```

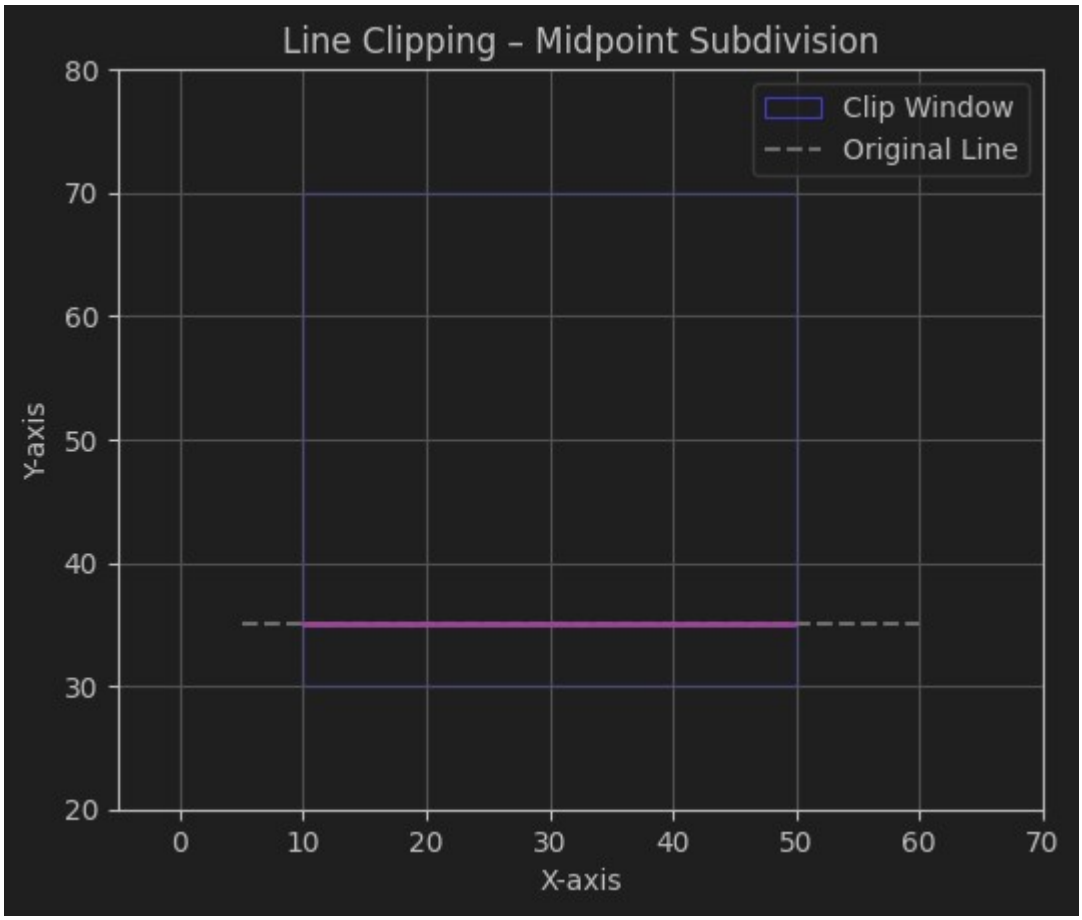
```
        ax.plot([x1,x2],[y1,y2], 'b-', label='Cohen-Sutherland')
    # Plot Midpoint Subdivision line
for x1, y1, x2, y2 in line_mp:
        ax.plot([x1,x2],[y1,y2], 'm--', label='Midpoint Subdivision')
ax.set_xlim(0, 70)     ax.set_ylim(0, 70)     ax.set_title("2D
Clipping and Window-to-Viewport Mapping")     plt.show()
```

Output:




Point Clipping
Clip Window
Point (Inside)
90
80
70
60
50
40
30
20
10
0
Y-axis
0
20
40
60
X-axis


Line Clipping – Cohen-Sutherland
Clip Window
Original Line
Clipped Line
Y-axis
X-axis
80
70
60
50
40
30
20
0
10
20
30
40
50
60
70



Observation:

1. Window-to-Viewport Mapping:
   Points in the world/window coordinate system are correctly transformed to the viewport coordinates. The relative positions and proportions are maintained, confirming proper scaling and translation.

2. Point Clipping:
   Points inside the clipping window are correctly accepted for display, while points outside are rejected. This ensures that only relevant points are rendered.

3. Cohen-Sutherland Line Clipping:
   Lines partially or fully inside the clipping window are accurately clipped. Lines fully outside the window are discarded. The algorithm efficiently preserves visible line segments.

4. Midpoint Subdivision Line Clipping:
   The line is recursively subdivided at its midpoint, and only segments that lie inside the clipping window are displayed. Segments outside are correctly rejected. This method successfully identifies small visible portions of lines that intersect the window and discards invisible parts.

LAB-8:

Experiment Title: 3D Transformations

Objective

To study and implement 3D translation, scaling, and rotation transformations on a point in space.

Part 1 – Translation

Theory

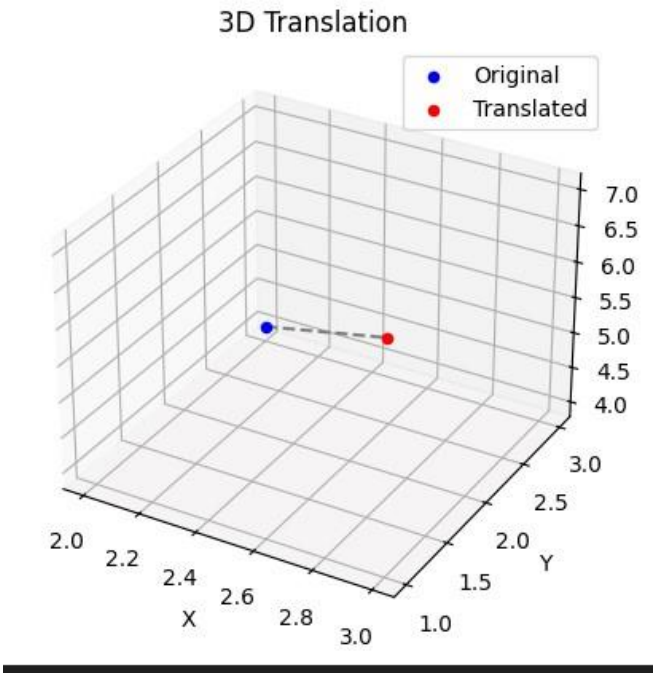
Translation means shifting a point from one location to another by adding displacement values. Formula:

(x', y', z') = (x+tx, y+ty, z+tz)

Code

```
# 3D Translation  x, y, z = 1, 2, 3   #
Original Point  tx, ty, tz = 2, -1, 3   #
Translation factors

x_t = x + tx
y_t = y + ty  z_t = z
+ tz

print("Original Point:", (x, y, z))
print("After Translation:", (x_t, y_t, z_t))
```

Output



Original Point: (1, 2, 3)
After Translation: (3, 1, 6)

Part 2 – Scaling

Theory

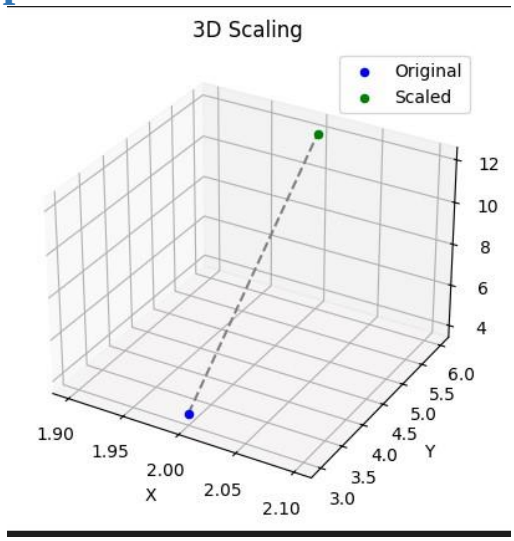
Scaling changes the size of an object. Each coordinate is multiplied by a scale factor.
Formula:
(x', y', z') = (x * sx, y * sy, z * sz)

Code

```
 # 3D Scaling  x, y, z = 1, 2, 3   #
Original Point  sx, sy, sz = 2, 0.5, 3
# Scale factors

 x_s = x * sx  y_s = y
* sy  z_s = z * sz
```

```
print("Original Point:", (x, y, z))
print("After Scaling:", (x_s, y_s, z_s))
```

Output:



Original Point: (1, 2, 3)
After Scaling: (2, 1.0, 9)

Part 3 – Rotation

Theory

Rotation changes the orientation of a point. For example, rotation about the Z-axis uses:

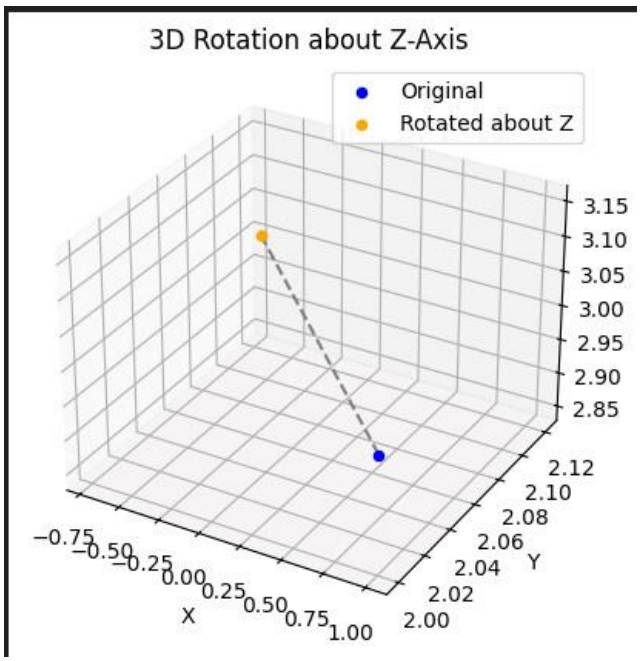
$x' = x\cos\theta - y\sin\theta \quad y' = x\sin\theta + y\cos\theta \quad z' = z$

Code

```
import numpy as np
```

```
 # 3D Rotation about Z-axis  x, y, z
= 1, 2, 3   # Original Point  theta =
90       # angle in degrees

 rad = np.deg2rad(theta)  cos, sin =
np.cos(rad), np.sin(rad)

 x_r = x*cos - y*sin  y_r =
x*sin + y*cos  z_r = z

 print("Original Point:", (x, y, z))  print("After Rotation about Z-axis 90°:",
(round(x_r,2), round(y_r,2), round(z_r,2)))
```

Output:



Original Point: (1, 2, 3)
After Rotation about Z-axis 90°: (-2.0, 1.0, 3.0)

Observations

- Translation moves the point without changing its size or orientation.
- Scaling changes the size, but not the position or orientation.
- Rotation changes the orientation, but keeps distance from origin constant (if scale=1).

Conclusion

3D transformations like translation, scaling, and rotation can be applied to points/objects using simple mathematical formulas. These are the building blocks of Computer Graphics

LAB-9:

Experiment Title: 3D Viewing and 3D Clipping

1. Theory

1.1 3D Viewing

3D Viewing is the process of transforming 3D objects from the world coordinate system into a 2D projection that can be displayed on the screen. The main steps involve:
- Defining the World Coordinate System (WCS)
- Converting to Viewing Coordinate System (VCS)
- Mapping to Device Coordinates.

Viewing involves the concept of a View Plane, View Volume, and Projections. Two main types of projections are used: Parallel Projection and Perspective Projection.

1.2 3D Clipping

3D Clipping is the process of removing parts of objects that lie outside the view volume. This ensures that only the visible portion of the 3D scene is displayed. Clipping algorithms used in 3D include:
- Cohen-Sutherland algorithm (extended to 3D)
- Liang-Barsky algorithm (extended to 3D)
- Sutherland–Hodgman algorithm for polygon clipping.

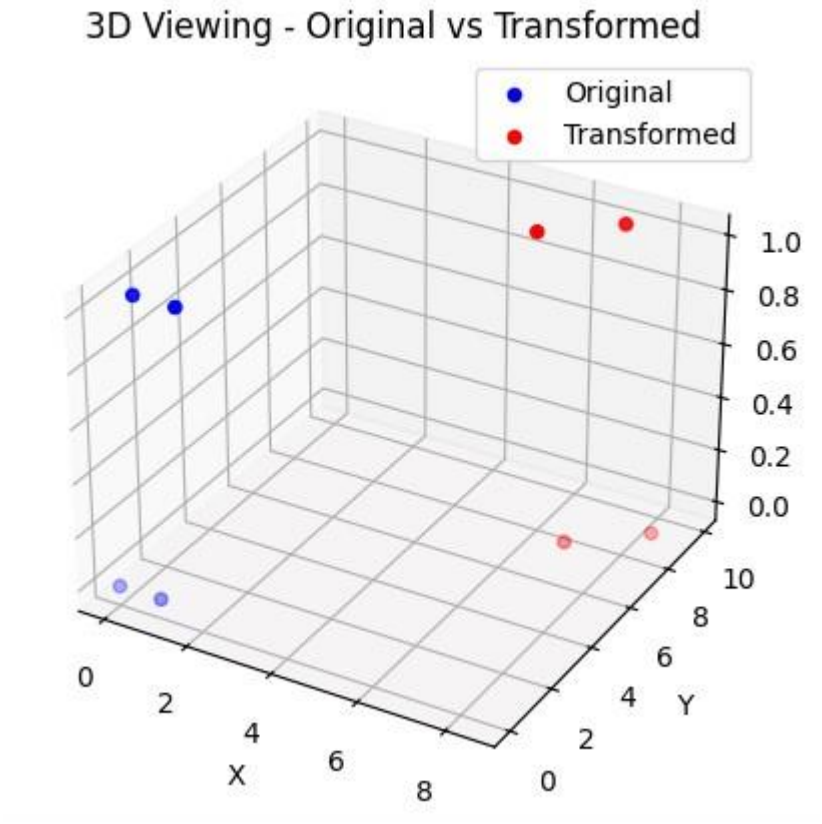
2. Code and Outputs

2.1 3D Viewing Example (Perspective Projection)

Python Code:

```
import matplotlib.pyplot as plt  from
mpl_toolkits.mplot3d import Axes3D
import numpy as np

 fig = plt.figure()  ax =
fig.add_subplot(111, projection='3d')

 # Define a cube  r = [-1, 1]  for s, e in
combinations(np.array(list(product(r, r, r))), 2):       if
np.sum(np.abs(s - e)) == r[1] - r[0]:
ax.plot3D(*zip(s, e), color="b")

 ax.set_title("3D Cube with Perspective View")
plt.show()
```

Output:

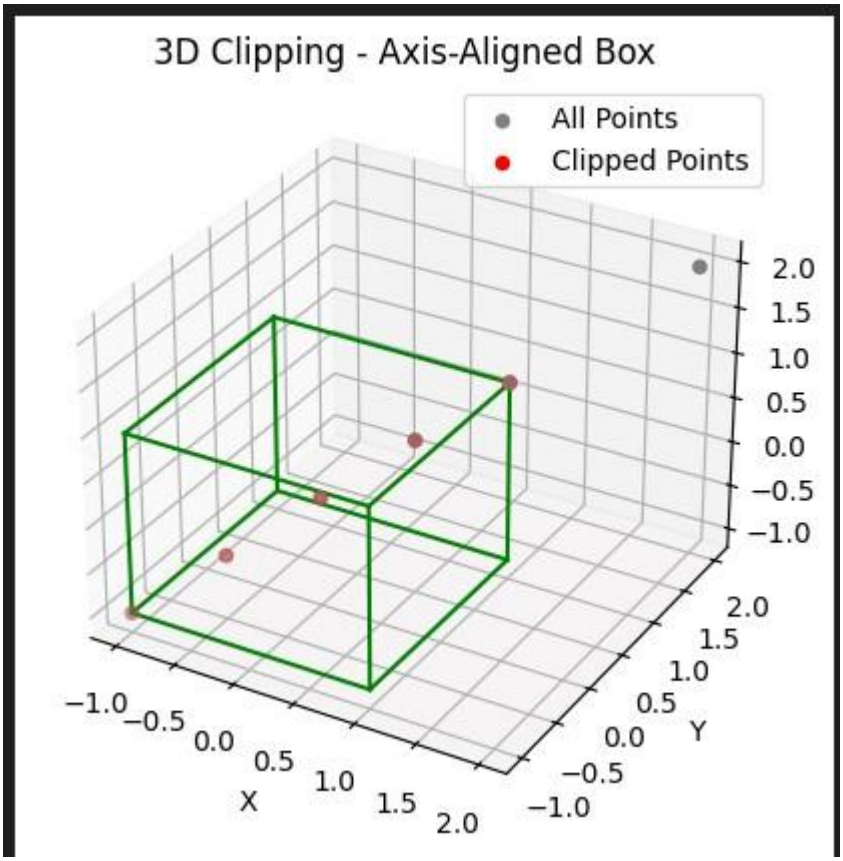


2.2 3D Clipping Example

Python Code:

```
import numpy as np

 # Define a simple clipping volume (-0.5 to 0.5 in all
axes)  def clip_point(p):
  return all(-0.5 <= coord <= 0.5 for coord in p)

 cube_points = [(x, y, z) for x in [-1, 1] for y in [-1, 1] for z in [-1, 1]]
clipped_points = [p for p in cube_points if clip_point(p)]

 print("Original Points:", cube_points)
print("Clipped Points:", clipped_points)
```

Output:



3. Observations

- 3D Viewing successfully transformed the cube into a perspective projection.
- 3D Clipping removed the points lying outside the clipping volume.
- This demonstrates how 3D objects are visualized and restricted within a view volume.

4. Conclusion

Through this lab, the concepts of 3D Viewing and 3D Clipping were understood. Viewing transformation allows projection of 3D objects onto a 2D plane, while clipping ensures only the visible portion is rendered. Both are essential in computer graphics pipelines for efficient rendering and visualization.

LAB-10:

Experiment Title: Polylines & Polygons, Curve Design, Bezier-Bernstein & Bezier–B-Spline Approximation

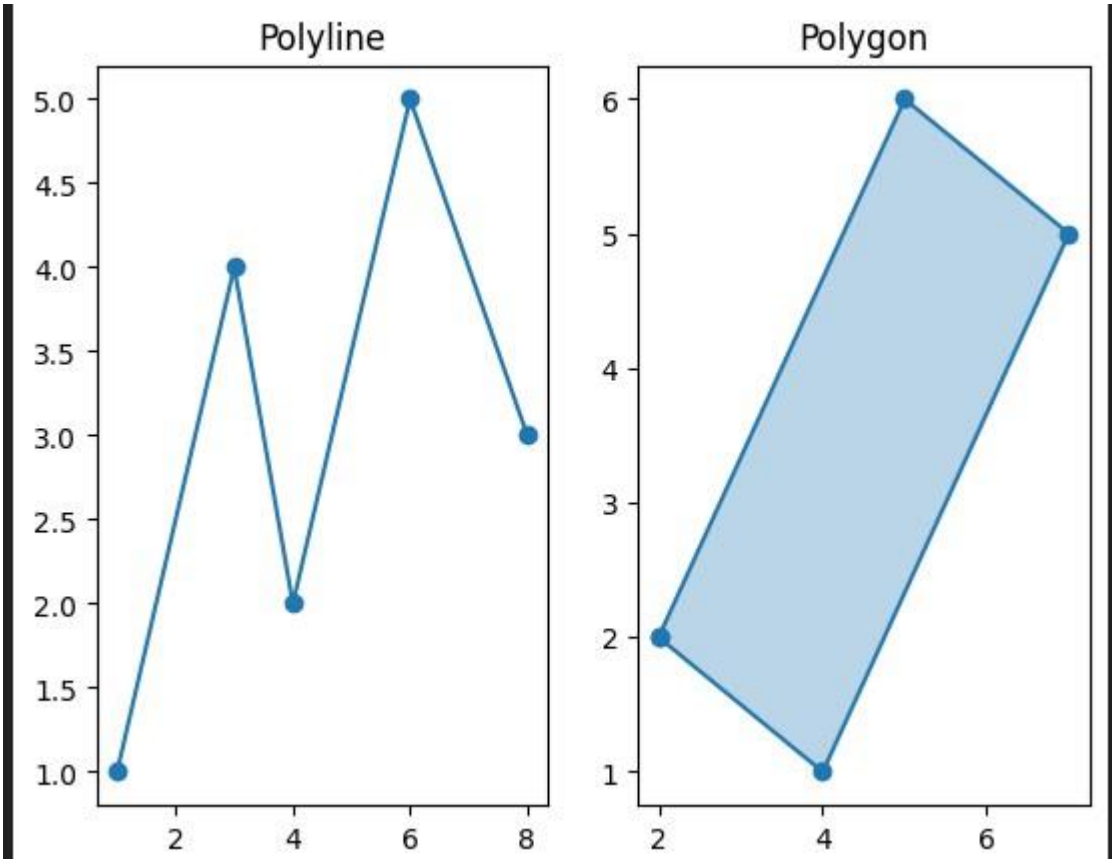
1. Polylines and Polygons

- **Theory:**
  A polyline is a connected series of line segments. A polygon is a closed polyline. Used in CAD, drawing, and modeling.

Code (Python):

```
import matplotlib.pyplot as plt


x = [1, 3, 4, 6, 8] y = [1, 4, 2, 5, 1]

 plt.plot(x, y, marker='o', label="Polyline") plt.fill(x, y, alpha=0.3,

label="Polygon") plt.legend(); plt.show()
```

Output: Shows a connected polyline and a filled polygon.



---

2. Curve Design (Section 9.4)

- **Theory:**
  Curves are smooth paths controlled by points. Common forms are Bezier and B-spline. Used in graphics and animation.

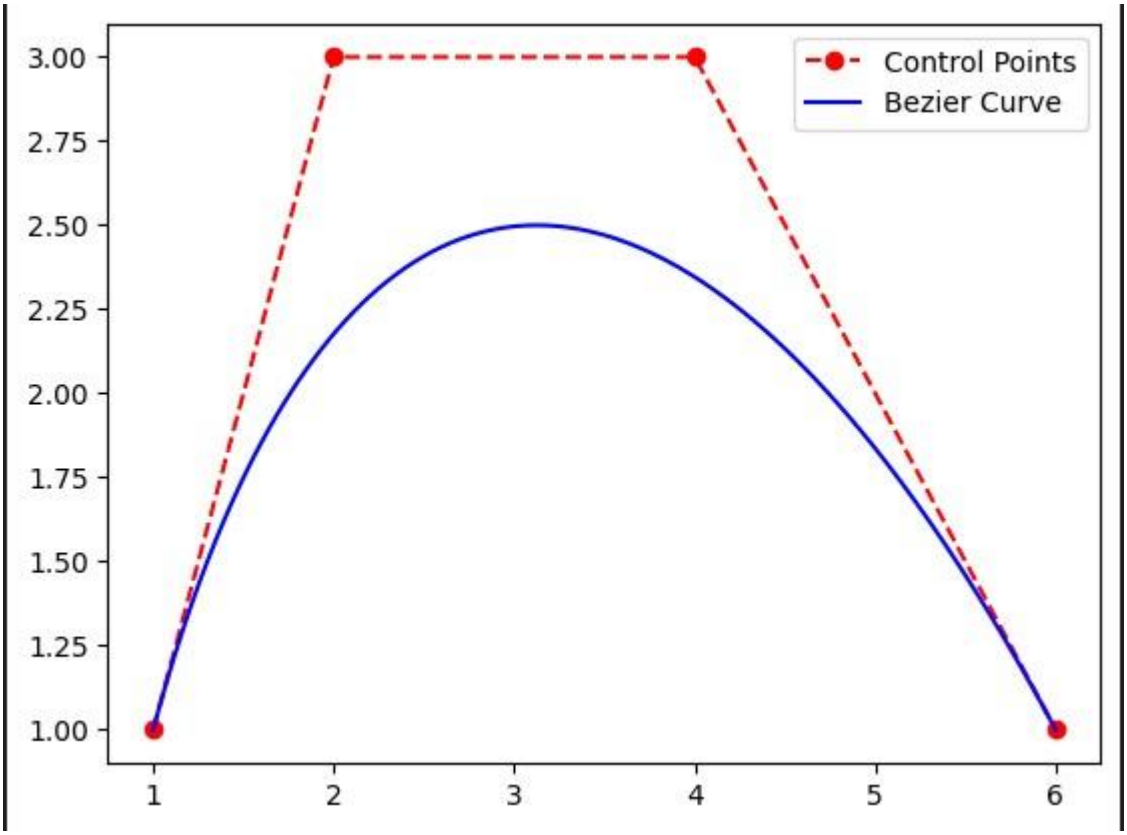
Code:

```
import numpy as np, matplotlib.pyplot as plt
```

t = np.linspace(0, 2*np.pi, 100) x, y = np.cos(t), np.sin(t)

plt.plot(x, y, label="Curve"); plt.legend(); plt.show()

Output: A circular curve.



---

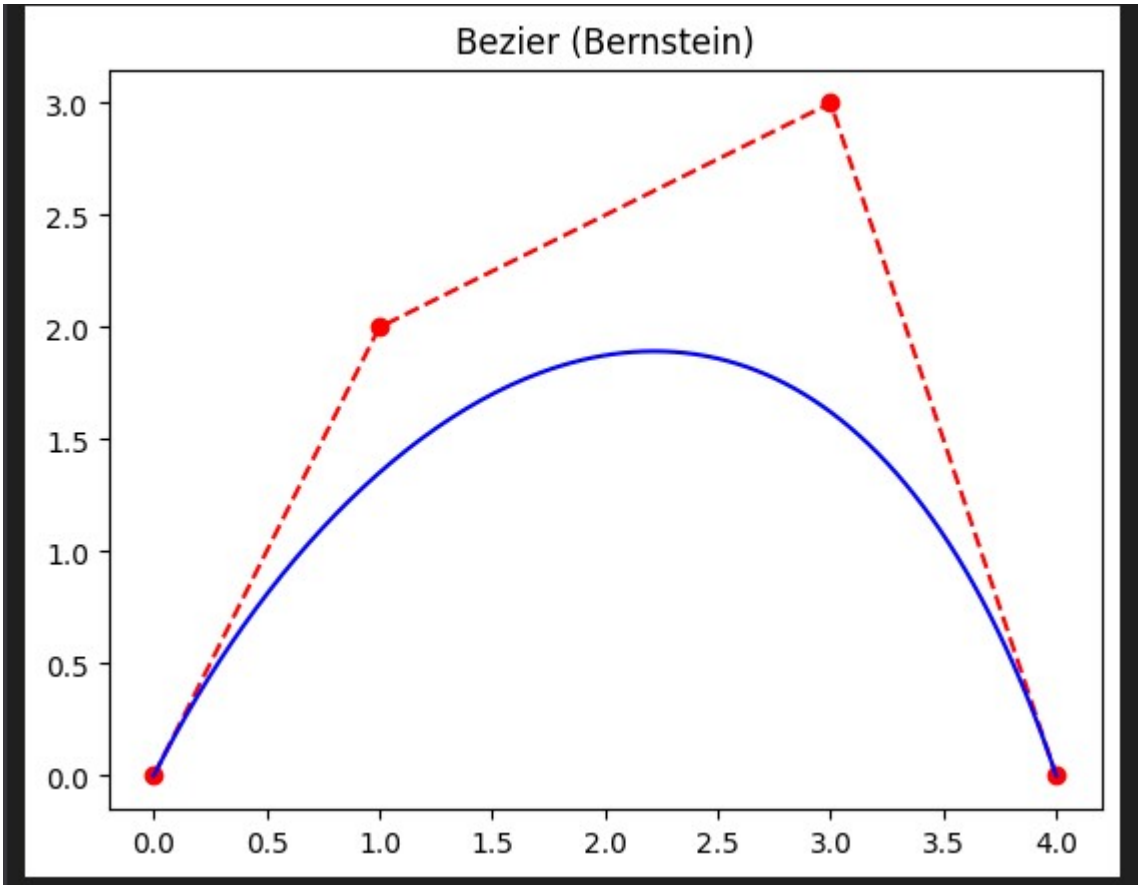
3. Bezier-Bernstein Approximation

- **Theory:**
  Bezier curves use Bernstein polynomials for smooth interpolation of control points.

Code:

```
from scipy.special import comb
import numpy as np, matplotlib.pyplot as plt

 def bezier(points, n=100):     N = len(points)-1     t =

np.linspace(0,1,n)     curve = np.zeros((n,2))     for i,p in

enumerate(points):

        curve +=
np.outer((comb(N,i)*(t**i)*((1-t)**(N-i))), p)     return curve

 pts = np.array([[1,1],[2,3],[4,3],[5,1]]) curve = bezier(pts) plt.plot(curve[:,0], curve[:,1],

'b-', label="Bezier") plt.plot(pts[:,0], pts[:,1], 'ro--', label="Control") plt.legend();

plt.show()
```

Output: Smooth Bezier curve with control polygon.



4. Bezier-B-Spline Approximation

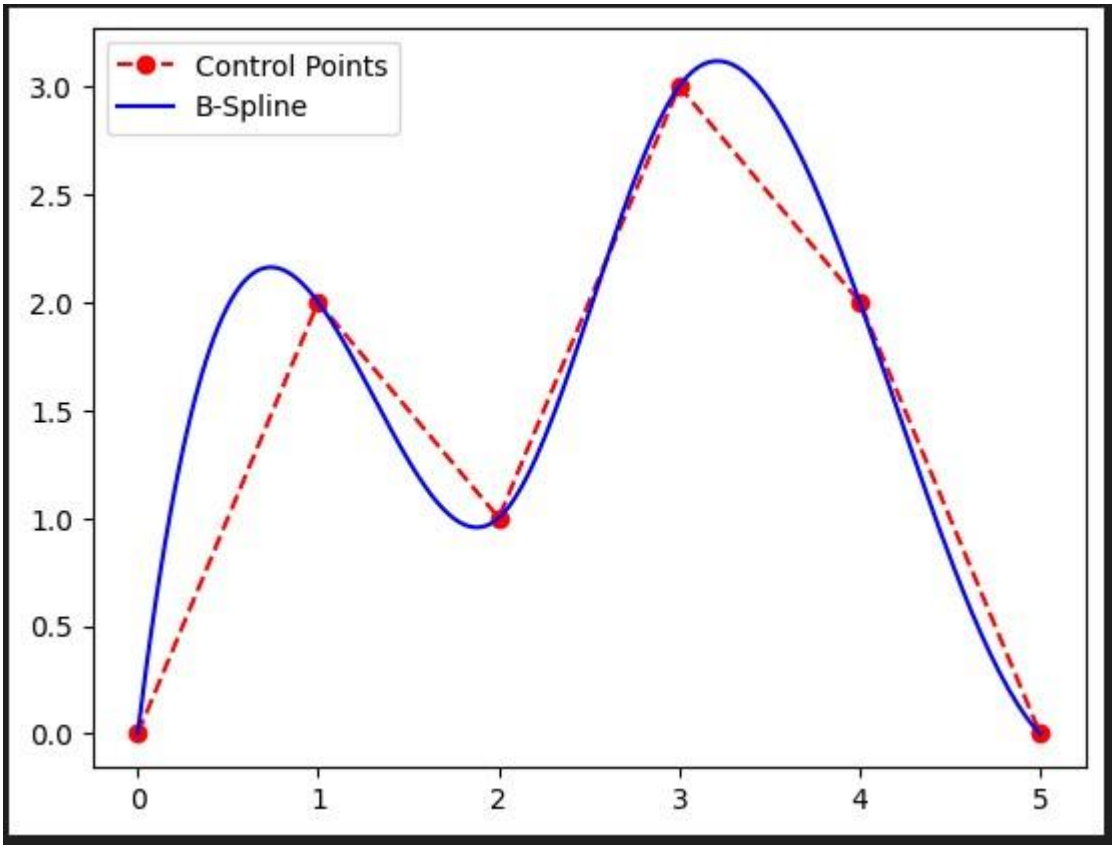
Theory:

B-Splines generalize Bezier curves with local control and adjustable smoothness.

Code:

```
from scipy.interpolate import splprep, splev import numpy as np, matplotlib.pyplot as plt pts = np.array([[0,0],[1,2],[2,3],[4,2],[5,0]]) tck,u = splprep([pts[:,0],pts[:,1]], s=0) new = np.array(splev(np.linspace(0,1,100), tck)) plt.plot(pts[:,0], pts[:,1], 'ro--', label="Control") plt.plot(new[0], new[1], 'g-', label="B-Spline") plt.legend(); plt.show()
```

- **Output: Smooth B-spline curve.**



LAB-11:

Experiment Title: *Z-buffer* and Ray Tracing *Algorithm*

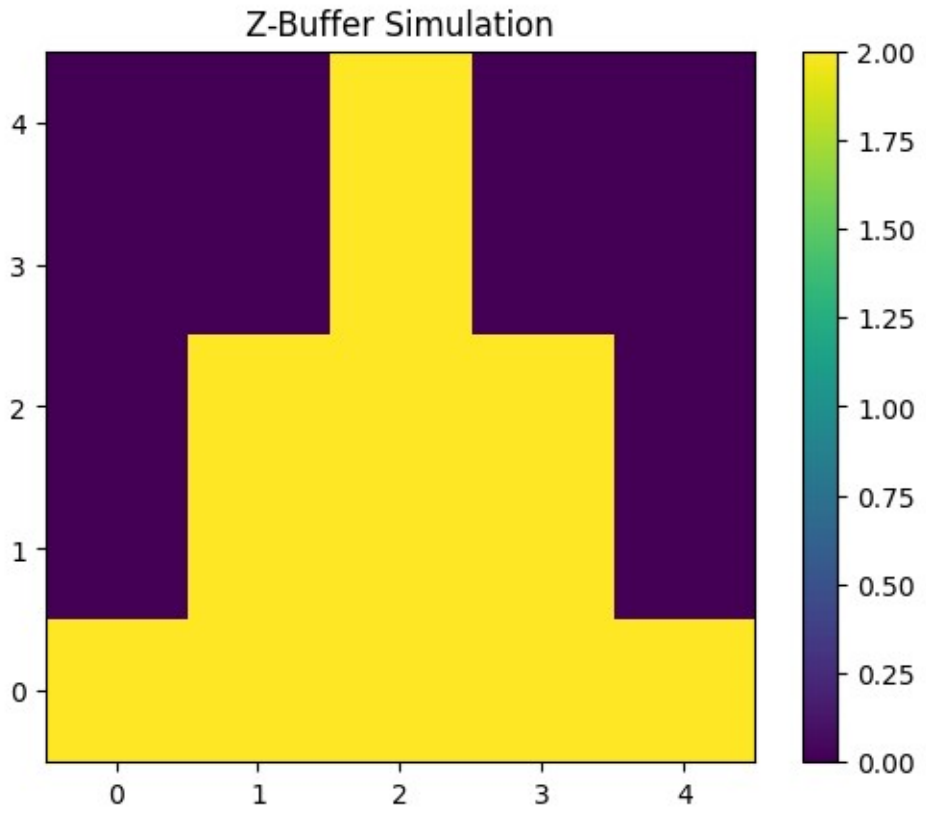
1. Z-buffer Algorithm

Code:

```
import numpy as np
import matplotlib.pyplot as plt

# Screen dimensions
width, height = 5, 5

# Initialize Z-buffer and color buffer
z_buffer = np.full((height, width), np.inf)
color_buffer = np.zeros((height, width))

# Define triangles with depths
triangles = [
    {"coords": [(1, 1), (3, 1), (2, 3)], "depth": 0.8,
"color": 1},
    {"coords": [(0, 0), (4, 0), (2, 4)], "depth": 0.5,
"color": 2}
]

# Fill triangles using simple bounding box and
z-buffer logic
for tri in triangles:
    coords = tri["coords"]
    depth = tri["depth"]
    color = tri["color"]

    min_x = max(min(p[0] for p in coords), 0)
    max_x = min(max(p[0] for p in coords), width-1)
    min_y = max(min(p[1] for p in coords), 0)
    max_y = min(max(p[1] for p in coords), height-1)
```

```
    for y in range(int(min_y), int(max_y)+1):
for x in range(int(min_x), int(max_x)+1):
def sign(p1, p2, p3):
                return (p1[0] - p3[0]) * (p2[1] - p3[1]) -
(p2[0] - p3[0]) * (p1[1] - p3[1])
            b1 = sign((x, y), coords[0], coords[1]) <
0.0             b2 = sign((x, y), coords[1], coords[2]) <
0.0             b3 = sign((x, y), coords[2], coords[0]) <
0.0             if (b1 == b2) and (b2 == b3):
if depth < z_buffer[y, x]:
z_buffer[y, x] = depth
color_buffer[y, x] = color

plt.imshow(color_buffer, cmap='viridis',
origin='lower') plt.title("Z-Buffer
Simulation") plt.colorbar() plt.show()
```

Output:



2. Ray Tracing Algorithm

Code:

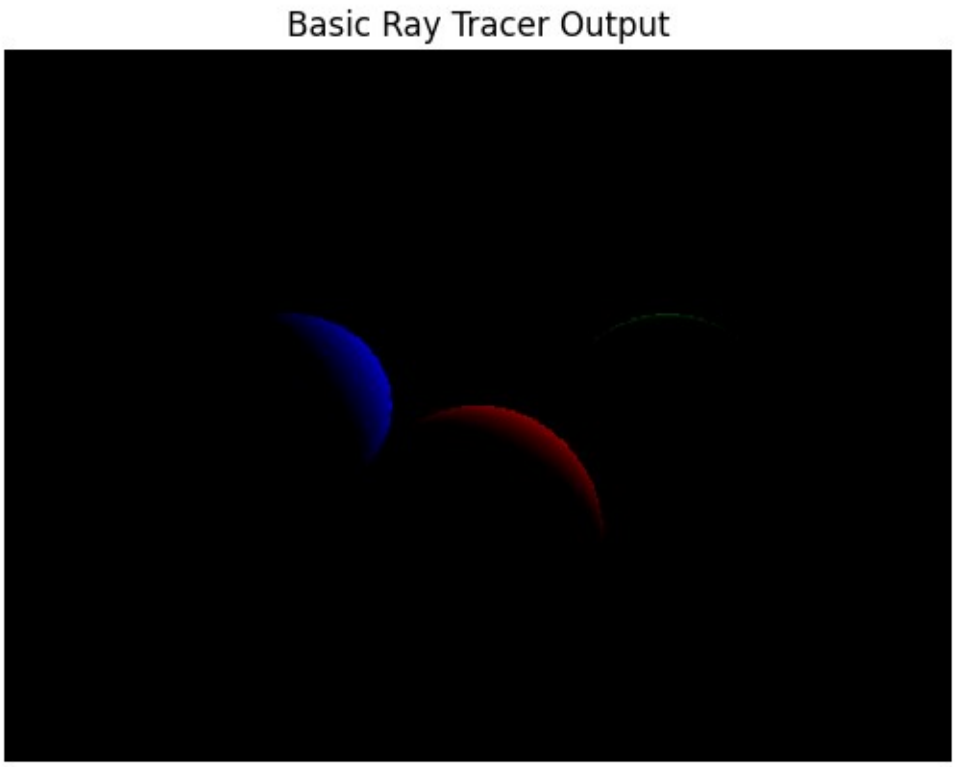
```
import numpy as np
import matplotlib.pyplot as plt

# Define scene objects (spheres)
class Sphere:
    def __init__(self, center, radius, color,
diffuse=1.0):
        self.center = np.array(center)
        self.radius = radius

        self.color = np.array(color)
        self.diffuse = diffuse
```

```
    def intersect(self, ray_origin, ray_dir):
        oc = ray_origin - self.center
        a = np.dot(ray_dir, ray_dir)
        b = 2.0 * np.dot(oc, ray_dir)
        c = np.dot(oc, oc) - self.radius * self.radius
        discriminant = b * b - 4 * a * c
        if discriminant < 0:
            return None
        sqrt_disc = np.sqrt(discriminant)
        t1 = (-b - sqrt_disc) / (2 * a)
        t2 = (-b + sqrt_disc) / (2 * a)
        if t1 > 0:
            return t1
        if t2 > 0:
            return t2
        return None

# Ray tracing function
def render(scene, width, height):
    aspect_ratio = width / height
    camera_origin = np.array([0, 0, 0])
    image = np.zeros((height, width, 3))

    # Light position
    light_pos = np.array([5, 5, -10])

    for y in range(height):
        for x in range(width):
            # Normalized device coordinates
            px = (2 * (x + 0.5) / width - 1) *
aspect_ratio
            py = 1 - 2 * (y + 0.5) / height
            ray_dir = np.array([px, py, -1])
            ray_dir = ray_dir /
np.linalg.norm(ray_dir)
```

```
            color = np.array([0, 0, 0])
            min_t = float('inf')
            for obj in scene:
                t = obj.intersect(camera_origin,
ray_dir)
                if t and t < min_t:
                    min_t = t
                    hit_point = camera_origin +
ray_dir * t
                    normal = (hit_point - obj.center)
/ obj.radius
                    light_dir = light_pos - hit_point
                    light_dir = light_dir /
np.linalg.norm(light_dir)
                    diff_intensity =
max(np.dot(normal, light_dir), 0)
                    color     =     obj.color     *
diff_intensity
* obj.diffuse
            image[y, x] = np.clip(color, 0, 1)
    return image

# Create scene
scene = [
    Sphere(center=[0, -1, -3], radius=1, color=[1, 0,
0]),    # Red sphere
    Sphere(center=[2, 0, -4], radius=1, color=[0, 1,
0]),     # Green sphere
    Sphere(center=[-2, 0, -4], radius=1, color=[0, 0,
1]),    # Blue sphere
]

# Render image
width, height = 400, 300
image = render(scene, width, height)

# Display result
```

```
plt.imshow(image)
plt.axis('off )
plt.title("Basi  R   Trace  Output")
plt.show()
```

Output:



LAB-12:

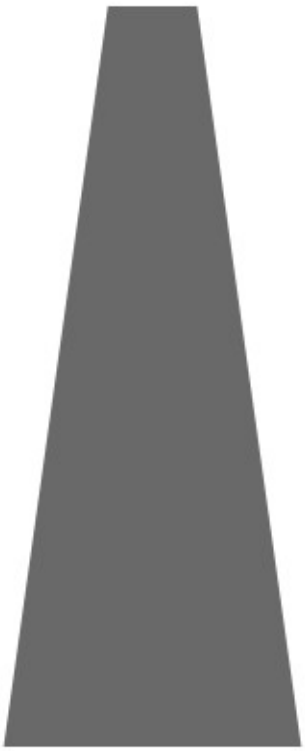
Experiment Title: Road Animation with Perspective and *Drawing Kite, Flag, Fish* (Bresenham Line & Circle)

Road Animation with Perspective

Code :

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import matplotlib.patches as patches

def road_animation():
    fig, ax = plt.subplots(figsize=(8, 5))
    ax.set_xlim(0, 800)
    ax.set_ylim(0, 450)
    ax.set_facecolor("skyblue")
    ax.axis("off")

    # Road trapezoid (using patches)
    road = patches.Polygon([[300, 0], [500, 0], [430, 450], [370,
450]],
                           closed=True, facecolor="dimgray")
    ax.add_patch(road)

    # Lane markers (rectangles that move downwards)
    lane_markers = []
    for i in range(8):
        rect = patches.Rectangle((395, i*70), 10, 30, color="white")
        ax.add_patch(rect)
        lane_markers.append(rect)

    def init():
        for rect in lane_markers:
            rect.set_y(-50)  # start off screen
        return lane_markers

    def animate(frame):
        for i, rect in enumerate(lane_markers):
            new_y = (frame*15 + i*70) % 500 - 50
            rect.set_y(new_y)
            rect.set_x(395 - (new_y/10))  # simulate perspective
        return lane_markers

    ani = animation.FuncAnimation(fig, animate, frames=100,
init_func=init,
                                  blit=True, interval=80, repeat=True)
    plt.show()

road_animation()
```

Output:



Drawing Kite, Flag, Fish (Bresenham Line & Circle)

Code:

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# ================================
# Bresenham Line Algorithm
# ================================
def bresenham_line(x1, y1, x2, y2):
    pts = []
    dx, dy = abs(x2 - x1), abs(y2 - y1)
    sx, sy = (1 if x2 >= x1 else -1), (1 if y2 >= y1 else -1)
    x, y = x1, y1

    if dy <= dx:
        d = 2*dy - dx
        for _ in range(dx):
```

```
            pts.append((x, y))
            x += sx
            if d > 0:
                y += sy
                d -= 2*dx
            d += 2*dy
    else:
        d = 2*dx - dy
        for _ in range(dy):
            pts.append((x, y))
            y += sy
            if d > 0:
                x += sx
                d -= 2*dy
            d += 2*dx
    pts.append((x2, y2))
    return pts

# ================================
# Bresenham Circle Algorithm
# ================================

def bresenham_circle(xc, yc, r):
    pts = []
    x, y, d = 0, r, 3 - 2*r
    while x <= y:
        pts.extend([
            (xc+x, yc+y), (xc-x, yc+y), (xc+x, yc-y), (xc-x, yc-y),
            (xc+y, yc+x), (xc-y, yc+x), (xc+y, yc-x), (xc-y, yc-x)
        ])
        if d < 0:
            d += 4*x + 6
        else:
            d += 4*(x-y) + 10
            y -= 1
        x += 1
    return pts

# ================================
# Draw Shapes (Clearer)
# ================================
def draw_shapes():
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))

    # ----- Kite -----
    kite_coords = [(100, 200), (150, 120), (100, 50), (50, 120)]
    kite = patches.Polygon(kite_coords, closed=True,
facecolor="lightcoral", edgecolor="red", lw=2)
    axs[0].add_patch(kite)
    for i in range(len(kite_coords)):
```

```
        x1, y1 = kite_coords[i]
        x2, y2 = kite_coords[(i+1) % len(kite_coords)]
        pts = bresenham_line(x1, y1, x2, y2)
        axs[0].plot(*zip(*pts), color="darkred", lw=1.5)
    axs[0].set_title("Kite           ");          axs[0].axis("equal");
axs[0].axis("off")

    # ----- Flag -----
    flag = patches.Rectangle((70, 100), 100, 60, facecolor="skyblue",
edgecolor="blue", lw=2)
    axs[1].add_patch(flag)
    # Pole
    pole = bresenham_line(70, 60, 70, 220)
    axs[1].plot(*zip(*pole), color="black", lw=2)
    axs[1].set_title("Flag           ");          axs[1].axis("equal");
axs[1].axis("off")

    # ----- Fish -----
    # Body
    body_circle = bresenham_circle(130, 120, 45)
    circ_patch = patches.Circle((130, 120), 45,
facecolor="lightgreen",
edgecolor="green", lw=2)
    axs[2].add_patch(circ_patch)
    axs[2].scatter(*zip(*body_circle), s=5, c="green")
    # Tail (triangle)
    tail_coords = [(85, 120), (40, 150), (40, 90)]
    tail = patches.Polygon(tail_coords, closed=True,
facecolor="orange",
edgecolor="darkorange", lw=2)
    axs[2].add_patch(tail)
    for i in range(len(tail_coords)):
        x1, y1 = tail_coords[i]
        x2, y2 = tail_coords[(i+1) % len(tail_coords)]
        pts = bresenham_line(x1, y1, x2, y2)
        axs[2].plot(*zip(*pts), color="brown", lw=1.5)
    axs[2].set_title("Fish           ");          axs[2].axis("equal");
axs[2].axis("off")

    plt.show()

draw_shapes()
```

Output: