Lab Report : 03

Title: Computer Graphics Lab
Course code: CSE-304
3rd Year 1st Semester

Date of Submission: **9.07.2023**

Submitted to-

Prof. Dr. Mohammad Shorif Uddin *Professor* *Department of Computer Science and Engineering* *Jahangirnagar University* *Savar, Dhaka-1342*	**Dr. Morium Akther** *Associate Professor* *Department of Computer Science and Engineering* *Jahangirnagar University* *Savar, Dhaka-1342*

Sl	Class Roll	Registration Number	Name
01	388	20200650758	Md.Tanvir Hossain Saon

Experiment No.07

Name Of the Experiment: Scan Conversion of a line object from(0,0) to (100,50).
i) Rotating 30 Degree
ii) Scale it to to 50%
iii)Translate it on x axis by 75px

Introduction:

Scan conversion is an essential process in computer graphics that involves the conversion of geometric objects represented in continuous coordinate systems to discrete coordinate systems, such as pixels on a display. One commonly encountered geometric object is a line, which can be defined by two endpoints. In this discussion, we will explore the scan conversion of a line object from the coordinates (0,0) to (100,50), followed by three transformations: rotation, scaling, and translation.

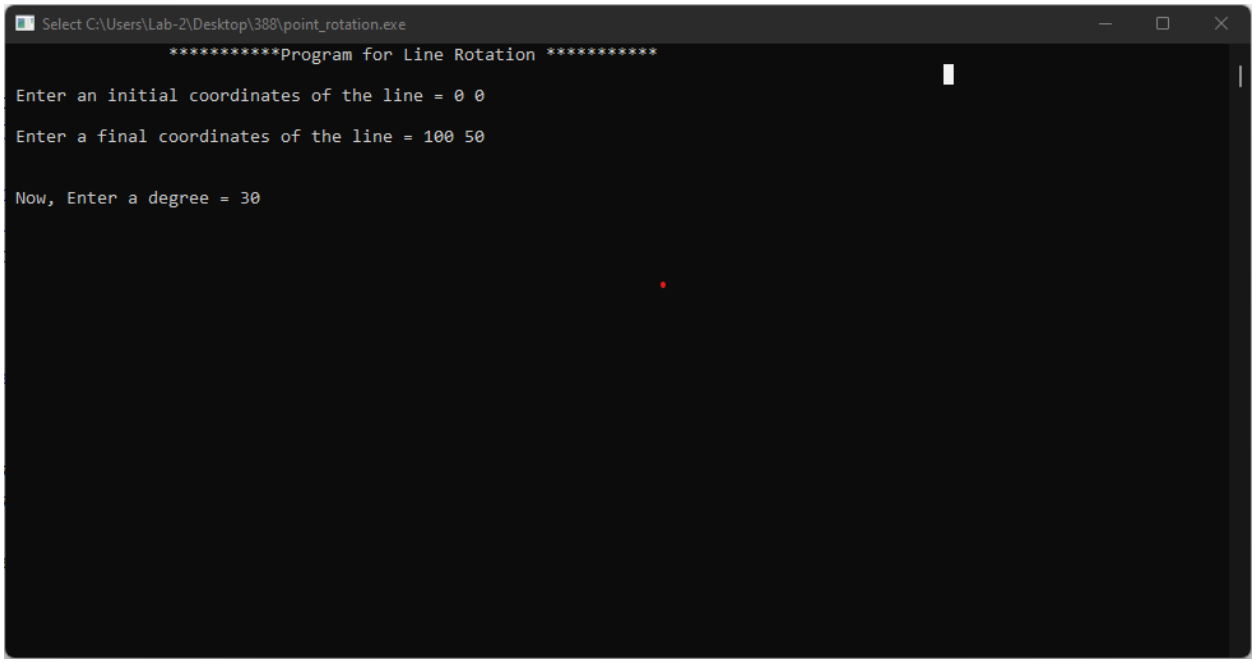
i) Rotating 30 Degree:

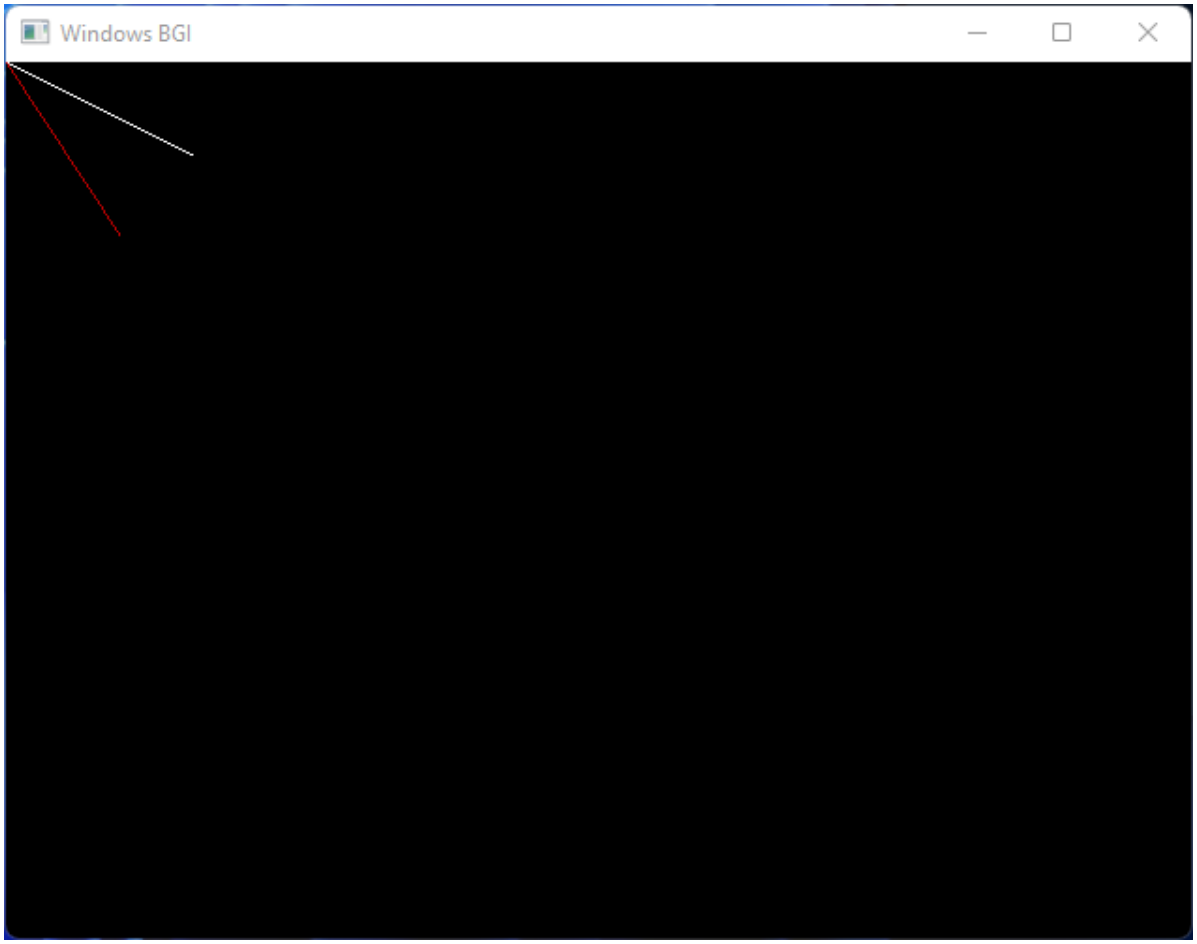
Source Code:

```
#include<graphics.h>
#include<stdio.h>
#include<conio.h>
#include<math.h>
int main()
{
    int gd=DETECT,gm;
    int pivot_x,pivot_y,x,y;
    double degree,radian;
    int rotated_point_x,rotated_point_y;
    initgraph(&gd,&gm,"C://TURBOC3//BGI");
    cleardevice();
    printf("\t\t***********Program for Line Rotation ***********
\n");
    printf("\n Enter an initial coordinates of the line = ");
    scanf("%d %d",&pivot_x,&pivot_y);
    printf("\n Enter a final coordinates of the line = ");
    scanf("%d %d",&x,&y);
    line(pivot_x,pivot_y,x,y);
```

```
    printf("\n\n Now, Enter a degree = ");
    scanf("%lf",&degree);
    radian=degree*0.01745;
    rotated_point_x=(int)(pivot_x
+((x-pivot_x)*cos(radian)-(y-pivot_y)*sin(radian)));
    rotated_point_y=(int)(pivot_y
+((x-pivot_x)*sin(radian)+(y-pivot_y)*cos(radian)));
    setcolor(RED);
    line(pivot_x,pivot_y,rotated_point_x,rotated_point_y);
    getch();
    closegraph();
}
```

OUTPUT:





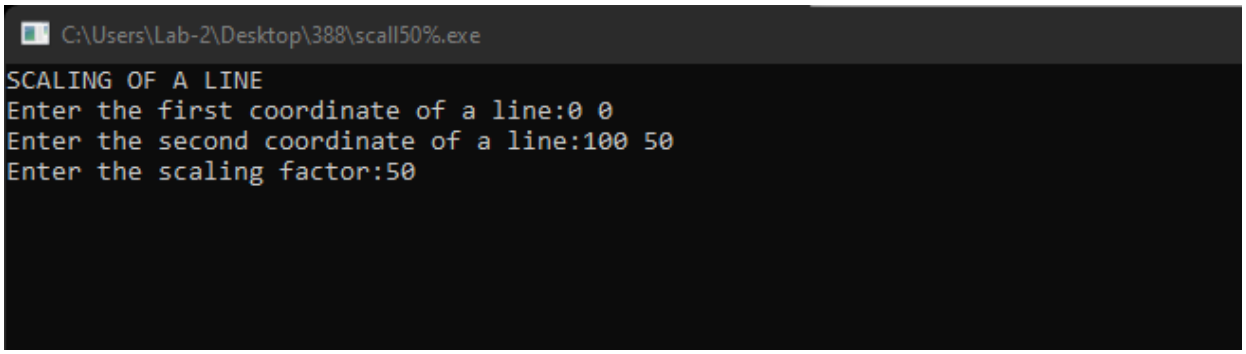
ii) Scale it to to 50%:

Source Code:

```
#include <iostream>
#include <conio.h>
#include <graphics.h>
using namespace std;
int main()
{
     int gd=DETECT,gm;
     float x1,y1,x2,y2,sx,sy,s;
     initgraph(&gd,&gm,"C:\\Tc\\BGI");
     cout<<"SCALING OF A LINE\n";
     cout<<"Enter the first coordinate of a line:";
     cin>>x1>>y1;
```

```
        cout<<"Enter the second coordinate of a line:";
        cin>>x2>>y2;
        line(x1,y1,x2,y2);
        cout<<"Enter the scaling factor:";
        cin>>s;
        sx=s/100,sy=s/100;
        setcolor(RED);
        x1=x1*sx;
        y1=y1*sy;
        x2=x2*sx;
        y2=y2*sy;
        line(x1,y1,x2,y2);
        getch();
        closegraph();
}
```

OUTPUT:

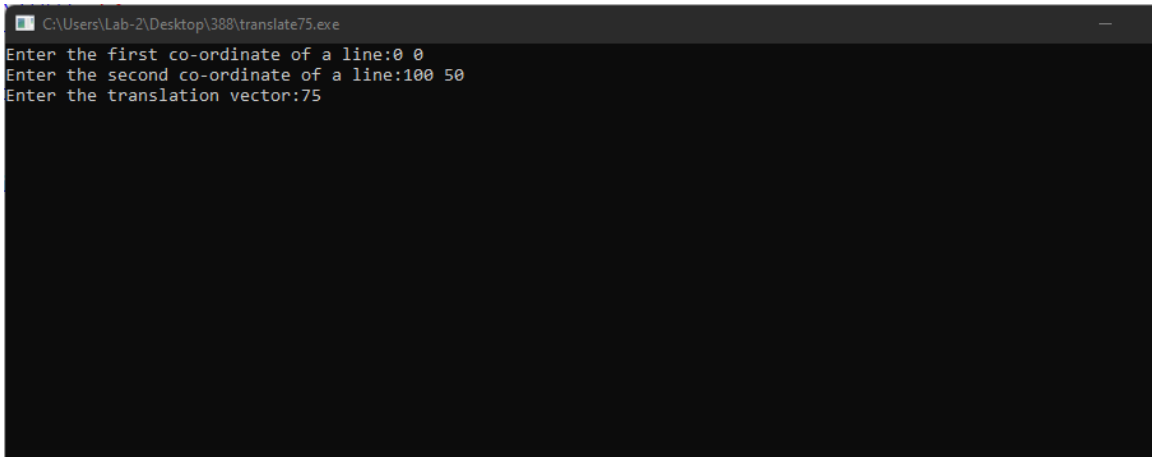


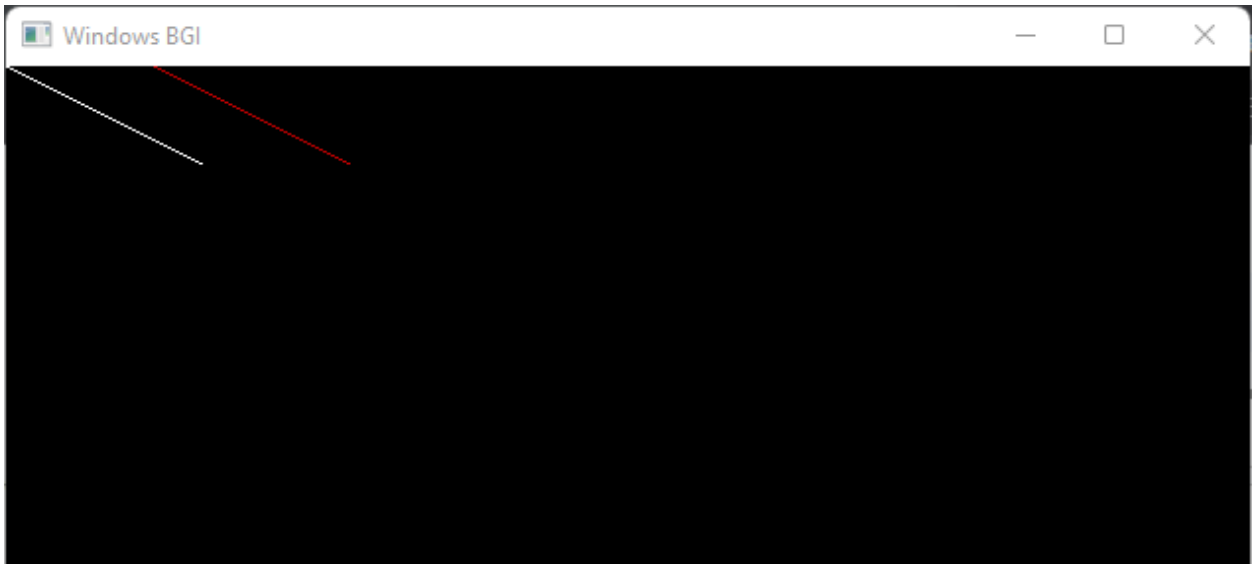
iii)Translate it on x axis by 75px:

Source Code:

```
#include <iostream>
#include <conio.h>
#include <graphics.h>
using namespace std;
int main()
{
      int gd=DETECT,gm,x1,x2,y1,y2,tx,ty;
      initgraph(&gd,&gm,"C:\\Tc\\BGI");
      cout<<"Enter the first co-ordinate of a line:";
      cin>>x1>>y1;
      cout<<"Enter the second co-ordinate of a line:";
      cin>>x2>>y2;
      line(x1,y1,x2,y2);
      cout<<"Enter the translation vector:";
      cin>>tx;
      setcolor(RED);
      x1=x1+tx;
      x2=x2+tx;
      line(x1,y1,x2,y2);
      getch();
      closegraph();
}
```

OUTPUT:





Discussion:

i) Rotating 30 Degrees:

To rotate the line object by 30 degrees, we need to apply a rotation transformation. The rotation can be performed using a rotation matrix or trigonometric functions. Assuming the rotation is performed around the origin (0,0), each coordinate of the line can be transformed using the following equations:

$x' = x * \cos(\theta) - y * \sin(\theta)$

$y' = x * \sin(\theta) + y * \cos(\theta)$

Where (x, y) represents the original coordinates of a point on the line, (x', y') represents the rotated coordinates, and θ is the angle of rotation (30 degrees in this case).

ii) Scaling to 50%:

Scaling involves resizing an object by multiplying or dividing its dimensions by a scaling factor. In this case, we want to scale the line to 50% of its original size. To achieve this, we can multiply each coordinate of the line by the scaling factor (0.5). The scaling transformation can be expressed as:

$x' = x * Sx$

$y' = y * Sy$

Where (x, y) represents the original coordinates of a point on the line, (x', y') represents the scaled coordinates, and Sx = Sy = 0.5 represents the scaling factors in the x and y directions.

iii) Translating on the x-axis by 75 pixels:

Translation involves shifting an object in a particular direction by adding or subtracting a fixed amount from its coordinates. In this case, we want to translate the line object by 75 pixels along the x-axis. To achieve this, we add the translation amount to the x-coordinate of each point on the line while keeping the y-coordinate unchanged. The translation transformation can be expressed as:

x' = x + Tx

y' = y

Where (x, y) represents the original coordinates of a point on the line, (x', y') represents the translated coordinates, and Tx represents the translation amount in the x direction (75 pixels in this case).

By applying these transformations successively, we can obtain the final scan-converted line object after rotation, scaling, and translation operations have been performed. These transformations are fundamental techniques used in computer graphics to manipulate and transform objects in a 2D space.

Experiment No.08

Name of the Experiment: Draw a kite using Bresenham's Algorithm.

Introduction:

Bresenham's Algorithm is a commonly used algorithm in computer graphics for drawing lines on a raster display. It is known for its efficiency in determining which pixels to activate to achieve the best approximation of a line between two given points. In this discussion, we will explore the use of Bresenham's Algorithm to draw a kite shape on a 2D grid.

Source Code:

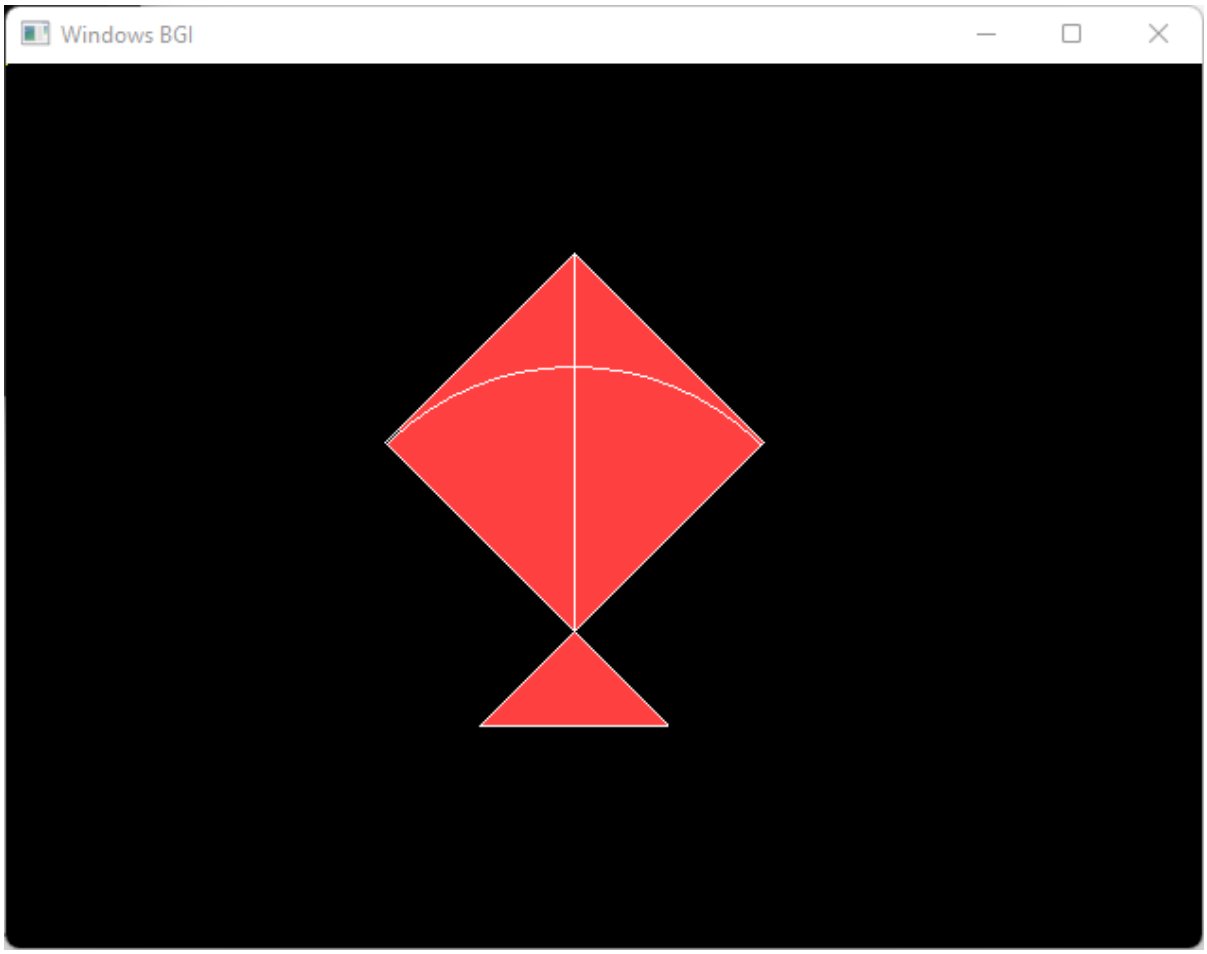
```
#include <graphics.h>
#include <iostream>
#include <conio.h>
#include <math.h>
using namespace std;
void kite()
{
    line(200, 200, 300, 100);
    line(300, 100, 400, 200);
    line(400, 200, 300, 300);
    line(300, 100, 300, 300);
    line(300,300,200,200);
    arc(300, 300, 45, 135, 140);
    setfillstyle(SOLID_FILL, 12);
    floodfill(301, 105, WHITE);
    floodfill(299, 105, WHITE);
    floodfill(299, 275, WHITE);
    floodfill(301, 275, WHITE);
    line(300, 300, 250, 350);
    line(250, 350, 350, 350);
    line(300, 300, 350, 350);
    floodfill(300, 310, WHITE);
}
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");

    kite();

    getch();
    closegraph();

    return 0;
}
```

OUTPUT:



Discussion:

Drawing a kite using Bresenham's Algorithm involves breaking down the shape into individual line segments and then using the algorithm to draw each segment. A kite typically consists of two intersecting diagonals that form an X shape, with the top and bottom edges connecting the endpoints of the diagonals.

To draw the kite, we can start by defining the coordinates of the endpoints for each line segment. Let's assume we have the following coordinates:

Endpoint 1: (x1, y1) Endpoint 2: (x2, y2)

Using Bresenham's Algorithm, we can determine which pixels to activate to approximate the line segment between Endpoint 1 and Endpoint 2. The algorithm takes into account the slope of the line and selects the closest pixel to the ideal line path.

Once we have the line segments for the diagonals and the top and bottom edges, we can draw each segment separately using Bresenham's Algorithm. By repeating this process for all line segments, we can construct the kite shape on the 2D grid.

It's important to note that Bresenham's Algorithm is typically used for drawing straight lines, so if you want to create a more precise kite shape with curved edges, you may need to use other techniques such as Bezier curves or other algorithms specific to curve drawing.

In conclusion, Bresenham's Algorithm can be utilized to draw a kite shape by breaking it down into individual line segments and applying the algorithm to each segment. By activating the appropriate pixels, we can approximate the desired kite shape on a 2D grid.