PUNE INSTITUTE OF COMPUTER TECHNOLOGY

SECOND YEAR
INFORMATION TECHNOLOGY
(2019 COURSE)

LABORATORY MANUAL
FOR

COMPUTER GRAPHICS LABORATORY

SEMESTER - III

[Subject code: 214457]

[Prepared By]

Dr. Kavita A. Sultanpure
Dr. Shweta C. Dharmadhikari
Mr. Abhinay G. Dhamankar

INSTITUTE VISION AND MISSION

VISION

Pune Institute of Computer Technology aspires to be the leader inhigher technical education and research of international repute.

MISSION

To be leading and most sought after Institute of education and research in emerging engineering and technology disciplines that attracts, retains and sustains gifted individuals of significant potential.

DEPARTMENT VISION AND MISSION

VISION

The department endeavors to be recognized globally as a center of academic excellence & research in Information Technology.

MISSION

To inculcate research culture among students by imparting information technology related fundamental knowledge, recent technological trends and ethics to get recognized as globally acceptable and socially responsible professionals.

Savitribai Phule Pune University, Pune **Second Year Information Technology (2019 Course)** **214457: Computer Graphics Lab**		
Teaching Scheme:	**Credit Scheme:**	**Examination Scheme:**
Practical (PR) :02hrs/week	**02**	**PR : 25 Marks** **TW: 25 Marks**

Prerequisites: Basic Geometry, Trigonometry, Vectors and Matrices, Data Structures and Algorithms

Course Objectives :

1. To acquaint the learners with the concepts of OpenGL.
2. To acquaint the learners with the basic concepts of Computer Graphics.
3. To implement the various algorithms for generating and rendering the objects.
4. To get familiar with mathematics behind the transformations.
5. To understand and apply various methods and techniques regarding animation.

Course Outcomes :

On completion of this course student will be able to --

CO1: Apply line& circle drawing algorithms to draw the objects.

CO2: Apply polygon filling methods for the object.

CO3: Apply polygon clipping algorithms for the object.

CO4: Apply the 2D transformations on the object.

CO5: Implement the curve generation algorithms.

CO6: Demonstrate the animation of any object using animation principles.

Guidelines for Instructor's Manual

The faculty member should prepare the laboratory manual for all the experiments and it should be made available to students and laboratory instructor/Assistant.

Guidelines for Student's Lab Journal

1. Student should submit term work in the form of journal with write-ups based on specified list of assignments.
2. Practical and Oral Examination will be based on all the assignments in the lab manual
3. Candidate is expected to know the theory involved in the experiment.
4. The practical examination should be conducted if and only if the journal of the candidate is complete in all respects.

Guidelines for Lab /TW Assessment

1. Examiners will assess the student based on performance of students considering the parameters such as timely conduction of practical assignment, methodology adopted for implementation of practical assignment, timely submission of assignment in the form of write-ups along with results of implemented assignment, attendance etc.
2. Examiners will judge the understanding of the practical performed in the examination by asking some questions related to theory & implementation of experiments he/she has carried

out. **3.** Appropriate knowledge of usage of software related to respective laboratory should be checked by the concerned faculty member.
Guidelines for Laboratory Conduction
1. All the assignments should be implemented in C++ with OpenGL libraries. **2.** Assignment 1 (week 1) should cover all the basic functions of openGL to get students familiar with Graphics Environment. Hence, this assignment is not included in Practical Exam. **3.** The different objects/shapes/patterns should be drawn for implementation of drawing algorithm. **4.** All the assignments should explore the conceptual understanding of students. **5.** The keyboard/Mouse interfaces should be used wherever possible.
Guidelines for PRACTICAL EXAM conduction
1. There will be 2 problem statements options and student will have to perform any one. **2.** All the problem statements carry equal weightage.
Virtual Laboratory
• https://cse18-iiith.vlabs.ac.in/ • http://vlabs.iitb.ac.in/vlabs-dev/labs/cglab/index.php
Suggested List of Laboratory Assignments
1. Install and explore the OpenGL -- **CO1**
2. Implement DDA and Bresenham line drawing algorithm to draw: i) Simple Line ii) Dotted Line iii) Dashed Line iv) Solid line ;using mouse interface Divide the screen in four quadrants with center as (0, 0). The line should work for all the slopes positive as well as negative.
3. Implement Bresenham circle drawing algorithm to draw any object. The object should be displayed in all the quadrants with respect to center and radius- **C02**
4. Implement the following polygon filling methods : i) Flood fill / Seed fill ii) Boundary fill ; using mouse click, keyboard interface and menu driven programming**- CO4**
5. Implement Cohen Sutherland polygon clipping method to clip the polygon with respect the viewport and window. Use mouse click, keyboard interface - **CO4**
6.Implement following 2D transformations on the object with respect to axis : – **CO5** i) Scaling ii) Rotation about arbitrary point iii) Reflection
7. Generate fractal patterns using i) Bezier ii) Koch Curve - **CO5**
8. Implement animation principles for any object - **CO6**
Text Books
1. S. Harrington, "Computer Graphics", 2nd Edition, McGraw-Hill Publications, 1987, ISBN 0-07-100472-6

2. D. Rogers, "Procedural Elements for Computer Graphics", 2nd Edition, McGraw-Hill Publications, 1987, ISBN 0-07-047371-4

3. F.S. Hill JR, "Computer Graphics Using OpenGL", Pearson Education

Reference Books

1. Graphics Principles and Practice", 2nd Edition, Pearson Education, 2003, ISBN 81 – 7808 – 038 – 9

2. D.Hearn, M. Baker, "Computer Graphics – C Version", 2nd Edition, Pearson Education, 2002, ISBN81 – 7808 – 794 – 4

3. D. Rogers, J. Adams, "Mathematical Elements for Computer Graphics", 2nd Edition, Tata McGraw-Hill Publication, 2002, ISBN 0 – 07 – 048677 – 8

4. Zhigang Xiang, Roy Plastock, "Computer Graphics", Schaum's Series outlines

5. Shirley, Marschner, "Fundamentals of Computer Graphics", Third Ed, A K Peters SPD

6. D.P. Mukharjee, Debasish Jana, "Computer Graphics Algorithms and implementation", PHI Learning

7. Samuel R. Buss, "3D Computer Graphics", Cambridge University Press

8. Mario Zechner, Robert Green, "Beginning Android 4 Games Development", Apress, ISBN: 978-81-322-0575-3

9. Maurya, "Computer Graphics with Virtual Reality Systems, 2ed.", Wiley, ISBN-9788126550883

10. Foley, "Computer Graphics: Principles & Practice in C", 2e, ISBN 9788131705056, Pearson

214457: COMPUTER GRAPHICS LABORATORY

Prerequisites:

Basic Geometry, Trigonometry, Vectors and Matrices, Data Structures and Algorithms

Course Objectives:

1. To acquaint the learners with the concepts of OpenGL.
2. To acquaint the learners with the basic concepts of Computer Graphics.
3. To implement the various algorithms for generating and rendering the objects.
4. To get familiar with mathematics behind the transformations.
5. To understand and apply various methods and techniques regarding animation.

Assignment No 1

Aim: Install and explore the OpenGL

Prob Statements: Install and explore the OpenGL Functions and Commands

Theory :

How to Install OpenGL on Linux

Part-1 Prepare your Linux Mint operating system for OpenGL Development

1 Open a terminal and enter the following commands to install the necessary libraries for OpenGL development:
Type/Copy/Paste following commands:

```
$ sudo apt-get update
$ sudo apt-get install freeglut3
$ sudo apt-get install freeglut3-dev
$ sudo apt-get install binutils-gold
$ sudo apt-get install g++ cmake
$ sudo apt-get install libglew-dev
$ sudo apt-get install g++
$ sudo apt-get install mesa-common-dev
$ sudo apt-get install build-essential
$ sudo apt-get install libglew1.5-dev libglm-dev
```

2 Get information about the OpenGL and GLX implementations running or not?
$ glxinfo | grep -i opengl
Your will get follwoing information : (**It Means OpenGL Installed**)

```
OpenGL vendor string: NVIDIA Corporation
OpenGL renderer string: GeForce 8800 GT/PCIe/SSE2
OpenGL version string: 2.1.2 NVIDIA 310.44
OpenGL shading language version string: 1.20 NVIDIA via Cg compiler
OpenGL extensions:
```

Part-2 Create your first OpenGL program

To create an OpenGL program, open up a terminal, make a directory, change into the directory and use text editor such as gedit to create your OpenGL source code.
Enter the following commands below.

- Type/Copy/Paste:
  $ mkdir Sample-OpenGL-Programs

(This will create a directory to hold your OpenGL programs)

- Type/Copy/Paste:
  $ cd Sample-OpenGL-Programs

(This will change you into your directory)

- Type/Copy/Paste:
  $ gedit main.c

Copy and paste OR Type the code :

```
#include <GL/freeglut.h>
#include <GL/gl.h>

void renderFunction()
```

```
{
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 1.0, 1.0);
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
glBegin(GL_POLYGON);
glVertex2f(-0.5, -0.5);
glVertex2f(-0.5, 0.5);
glVertex2f(0.5, 0.5);
glVertex2f(0.5, -0.5);
glEnd();
glFlush();
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE);
glutInitWindowSize(500,500);
glutInitWindowPosition(100,100);
glutCreateWindow("OpenGL - First window demo");
glutDisplayFunc(renderFunction);
glutMainLoop();
return 0;
}
```

- Save the file and exit

Part-3 Compile and Run your OpenGL application

1 This command will compile and link your OpenGL libraries.
$ g++ main.c -lglut -lGL -lGLEW -lGLU -o OpenGLExample

2 In order to run the program type the following below:
$ ./OpenGLExample

Header Files
In all of our graphics programs, we will need to include the header file for the OpenGL core library. For most applications we will also need GLU, and on many systems we will need to include the header file for the window system. For instance, with Microsoft Windows, the header file that accesses the WGL routines is **windows.h**. This header file must be listed before the OpenGL and GLU header files because it contains macros needed by the MicrosoftWindows version of the OpenGL libraries. So the source file in this case would begin with

#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>

However, if we use GLUT to handle the window-managing operations, we do not need to include **gl.h** and **glu.h** because GLUT ensures that these will be included correctly. Thus, we can replace the header files for OpenGL and GLU with

#include <GL/glut.h>

(We could include **gl.h** and **glu.h** as well, but doing so would be redundant and could affect program portability.) On some systems, the header files for OpenGL and GLUT routines are found in different places in the filesystem. For instance, on Apple OS X systems, the header file inclusion statement would be

#include <GLUT/glut.h>

In addition, we will often need to include header files that are required by the C++ code. For example,

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

With the ISO/ANSI standard for C++, these header files are called **cstdio**, **cstdlib**, and **cmath**.

Display-Window Management Using GLUT

To get started, we can consider a simplified, minimal number of operations for displaying a picture. Since we are using the OpenGL Utility Toolkit, our first step is to initializeGLUT. This initialization function could also process any command line arguments, but we will not need to use these parameters for our first example programs.We perform the GLUT initialization with the statement

glutInit (&argc, argv);

Next, we can state that a display window is to be created on the screen with a given caption for the title bar. This is accomplished with the function

glutCreateWindow ("An Example OpenGL Program");

where the single argument for this function can be any character string that we want to use for the display-window title. Then we need to specify what the display window is to contain. For this, we create a picture using OpenGL functions and pass the picture definition to the GLUT routine **glutDisplayFunc**, which assigns our picture to the display window. As an example, suppose we have the OpenGL code for describing a line segment in a procedure called **lineSegment**. Then the following function call passes the line-segment description to the display window:

glutDisplayFunc (lineSegment);

But the display window is not yet on the screen. We need one more GLUT function to complete the window-processing operations. After execution of the following statement, all display windows that we have created, including their graphic content, are now activated:

glutMainLoop ( );

This function must be the last one in our program. It displays the initial graphics and puts the program into an infinite loop that checks for input from devices such as a mouse or keyboard. Our first example will not be interactive, so the program will just continue to display our picture until we close the display window. In later chapters, we consider how we can modify our OpenGL programs to handle interactive input. Although the display window that we created will be in some default location and size,wecan set these parameters using additionalGLUTfunctions.We use the **glutInitWindowPosition** function to give an initial location for the upperleft corner of the display window. This position is specified in integer screen

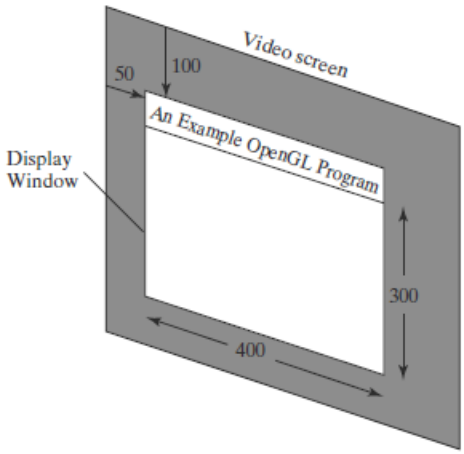


FIGURE 2
A 400 by 300 display window at position (50, 100) relative to the top-left corner of the video display.

coordinates, whose origin is at the upper-left corner of the screen. For instance, the following statement specifies that the upper-left corner of the display window should be placed 50 pixels to the right of the left edge of the screen and 100 pixels down from the top edge of the screen:

glutInitWindowPosition (50, 100);

Similarly, the **glutInitWindowSize** function is used to set the initial pixel width and height of the display window. Thus, we specify a display window with an initial width of 400 pixels and a height of 300 pixels (Fig. 2) with the statement

glutInitWindowSize (400, 300);

After the display window is on the screen, we can reposition and resize it. We can also set a number of other options for the display window, such as buffering and a choice of color modes, with the **glutInitDisplayMode** function.

Arguments for this routine are assigned symbolicGLUTconstants. For example, the following command specifies that a single refresh buffer is to be used for the display window and that we want to use the color mode which uses red, green, and blue (RGB) components to select color values:

glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);

The values of the constants passed to this function are combined using a logical *or* operation. Actually, single buffering and RGB color mode are the default options. But we will use the function now as a reminder that these are the options that are set for our display. Later, we discuss color modes in more detail, as well as other display options, such as double buffering for animation applications and selecting parameters for viewing three-dimensional scenes.

A Complete OpenGL Program
There are still a few more tasks to perform beforewehave all the parts thatweneed for a complete program. For the display window, we can choose a background color.Andweneed to construct a procedure that contains the appropriateOpenGL functions for the picture that we want to display.

Using RGB color values, we set the background color for the display window to be white, as in Figure 2, with the OpenGL function:

glClearColor (1.0, 1.0, 1.0, 0.0);

The first three arguments in this function set the red, green, and blue component colors to the value 1.0, giving us a white background color for the display window. If, instead of 1.0, we set each of the component colors to 0.0, we would get a black background. And if all three of these components were set to the same intermediate value between 0.0 and 1.0, we would get some shade of gray. The fourth parameter in the **glClearColor** function is called the *alpha value* for the specified color. One use for the alpha value is as a "blending" parameter. When we activate the OpenGL blending operations, alpha values can be used to determine the resulting color for two overlapping objects. An alpha value of 0.0 indicates a totally transparent object, and an alpha value of 1.0 indicates an opaque object. Blending operations will not be used for a while, so the value of alpha is irrelevant to our early example programs. For now, we will simply set alpha to 0.0.
Although the **glClearColor** command assigns a color to the display window, it does not put the displaywindowon the screen. To get the assignedwindow color displayed, we need to invoke the following OpenGL function:

glClear (GL_COLOR_BUFFER_BIT);

The argument **GL COLOR BUFFER BIT** is an OpenGL symbolic constant specifying that it is the bit values in the color buffer (refresh buffer) that are to be set to the values indicated in the **glClearColor** function. (OpenGL has several different kinds of buffers that can be manipulated.

In addition to setting the background color for the display window, we canchoose a variety of color schemes for the objects we want to display in a scene. For our initial programming example, we will simply set the object color to be a dark green

glColor3f (0.0, 0.4, 0.2);

The suffix **3f** on the **glColor** function indicates that we are specifying the three RGB color components using floating-point (**f**) values. This function requires that the values be in the range from 0.0 to 1.0, and we have set red = 0.0, green = 0.4, and blue = 0.2.
For our first program, we simply display a two-dimensional line segment. To do this, we need to tell OpenGL how we want to "project" our picture onto the display window because generating a two-dimensional picture is treated by OpenGL as a special case of three-dimensional viewing. So, although we only want to produce a very simple two-dimensional line, OpenGL processes our picture through the full three-dimensional viewing operations. We can set the projection type (mode) and other viewing parameters that we need with the following two functions:

glMatrixMode (GL_PROJECTION);
gluOrtho2D (0.0, 200.0, 0.0, 150.0);

This specifies that an orthogonal projection is to be used to map the contents of a two-dimensional rectangular area of world coordinates to the screen, and that the *x*-coordinate values within this rectangle range from 0.0 to 200.0 with *y*-coordinate values ranging from 0.0 to 150.0. Whatever objects we define within this world-coordinate rectangle will be shown within the display window. Anything outside this coordinate range will not be displayed. Therefore, the GLU function **gluOrtho2D** defines the coordinate reference frame within the display window to be (0.0, 0.0) at the lower-left corner of the display window and (200.0, 150.0) at the upper-right window corner. Since we are only describing a two-dimensional object, the orthogonal projection has no other effect than to "paste" our picture into the display window that we defined earlier. For now, we will use a world-coordinate rectangle with the same aspect ratio as the display window, so that there is no distortion of our picture. Later, we will consider how we can maintain an aspect ratio that does not depend upon the display-window specification. Finally,weneed to call the appropriateOpenGLroutines to create our line segment. The following code defines a two-dimensional, straight-line segment with integer, Cartesian endpoint coordinates (180, 15) and (10, 145).

glBegin (GL_LINES);
glVertex2i (180, 15);
glVertex2i (10, 145);
glEnd ( );

Now we are ready to put all the pieces together. The following OpenGL program is organized into three functions.We place all initializations and related one-time parameter settings in function **init**. Our geometric description of the "picture" that we want to display is in function **lineSegment**, which is the function that will be referenced by the GLUT function **glutDisplayFunc**. And the **main** function contains the GLUT functions for setting up the display window and getting our line segment onto the screen. Figure 3 shows the displaywindow and line segment generated by this program.

```
#include <GL/glut.h>      // (or others, depending on the system in use)

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 0.0);  // Set display-window color to white.

    glMatrixMode (GL_PROJECTION);       // Set projection parameters.
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void lineSegment (void)
{
    glClear (GL_COLOR_BUFFER_BIT);  // Clear display window.

    glColor3f (0.0, 0.4, 0.2);      // Set line segment color to green.
    glBegin (GL_LINES);
        glVertex2i (180, 15);       // Specify line-segment geometry.
        glVertex2i (10, 145);
    glEnd ( );

    glFlush ( );     // Process all OpenGL routines as quickly as possible.
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);                          // Initialize GLUT.
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);    // Set display mode.
    glutInitWindowPosition (50, 100);   // Set top-left display-window position.
    glutInitWindowSize (400, 300);      // Set display-window width and height.
    glutCreateWindow ("An Example OpenGL Program"); // Create display window.

    init ( );                           // Execute initialization procedure.
    glutDisplayFunc (lineSegment);      // Send graphics to display window.
    glutMainLoop ( );                   // Display everything and wait.
}
```

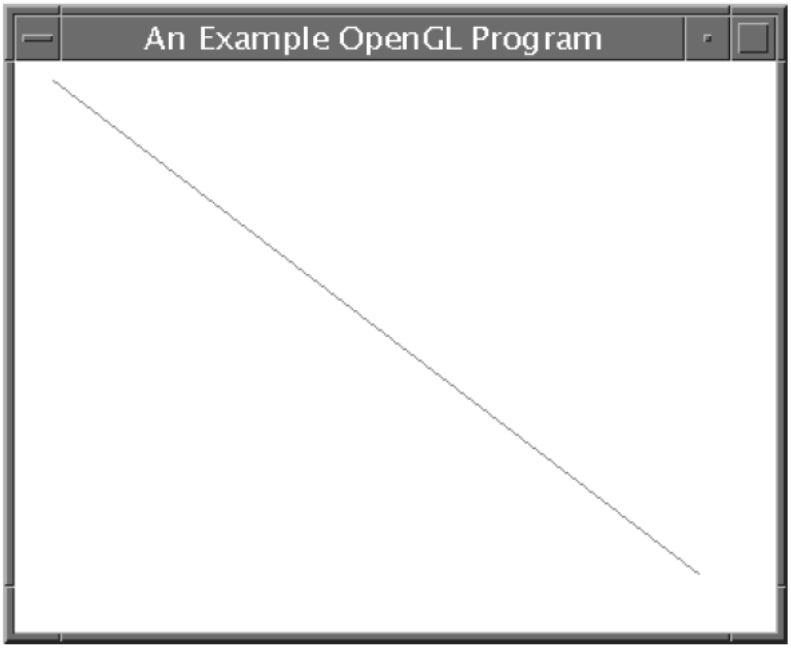


FIGURE 3
The display window and line segment produced by the example program.

Conclusion : Successfully installed OpenGL and Successfully studies the OpenGL Commands

Assignment No 2

Aim: Understand and Implement DDA and Bresenham's Line Drawing Algorithms using OpenGL.

Prob Statements : Implement DDA and Bresenham line drawing algorithm to draw: i) Simple Line ii) Dotted Line iii) Dashed Line iv) Solid line ;
using mouse interface Divide the screen in four quadrants with center as (0, 0). The line should work for all the slopes positive as well as negative.

Theory :

Theory:

A) DDA (Digital Differential Analyzer) Line Drawing Algorithm

- In computer graphics, a digital differential analyzer (DDA) is hardware or software used for interpolation of variable over an interval between start and end point. DDAs are used for rasterization of lines, triangles and polygons. They can be extended to nonlinear functions, such as perspective correct texture mapping, quadratic curves, and traversing voxels.
- **A linear DDA starts by calculating the smaller of dy or dx for a unit increment of the other. A line is then sampled at unit intervals in one coordinate and corresponding integer values nearest the line path are determined for the other coordinate.**
- **Considering a line with positive slope, if the slope is less than or equal to 1, we sample at unit x intervals (dx=1) and compute successive y values as subscript k takes integer values starting from 0, for the 1st point and increases by 1 until endpoint is reached. y value is rounded off to nearest integer to correspond to a screen pixel.**
- **For lines with slope greater than 1, we reverse the role of x and y i.e., we sample at dy=1 and calculate consecutive x values as similar calculations are carried out to determine pixel positions along a line with negative slope. Thus, if the absolute value of the slope is less than 1, we set dx=1 if i.e., the starting extreme point is at the left.**

B) Bresenham's Line Drawing Algorithm

- The Bresenham's Line drawing algorithm is a scan line algorithm used to determine the intermediate points which determine the line segment between a given initial point and a given final point.
- This algorithm is faster than the DDA algorithm and more efficient as it only involves integer addition, subtraction, multiplication and division.
- These integer calculations have a higher calculation speed than the floating-point calculations and hence, the line is plotted at a higher speed.
- A decision parameter is used to determine the position of the next pixel.
- 2 points are considered while plotting the next ($i+1^{th}$) point. The actual point value is determined by the slope point form equation which can be determined by the 2 given points. The distance value affects the decision parameter and hence the actual distance is not calculated.
- The distance between the actual point and the 2 assumed points is compared. The point which is closer to the actual point is considered for plotting.
- This process is repeated till the final given point is reached.
- The algorithm was initially determined for a line of slope less than 1, which can be generalized by altering a few parameters.

Algorithm:

A) DDA :-

Given-

- **Starting coordinates = (X_0, Y_0)**
- **Ending coordinates = (X_n, Y_n)**

The points generation using DDA Algorithm involves the following steps-

Step-01:

Calculate ΔX, ΔY and M from the given input.

These parameters are calculated as-

- **$\Delta X = X_n - X_0$**
- **$\Delta Y = Y_n - Y_0$**
- **$M = \Delta Y / \Delta X$**

Step-02:

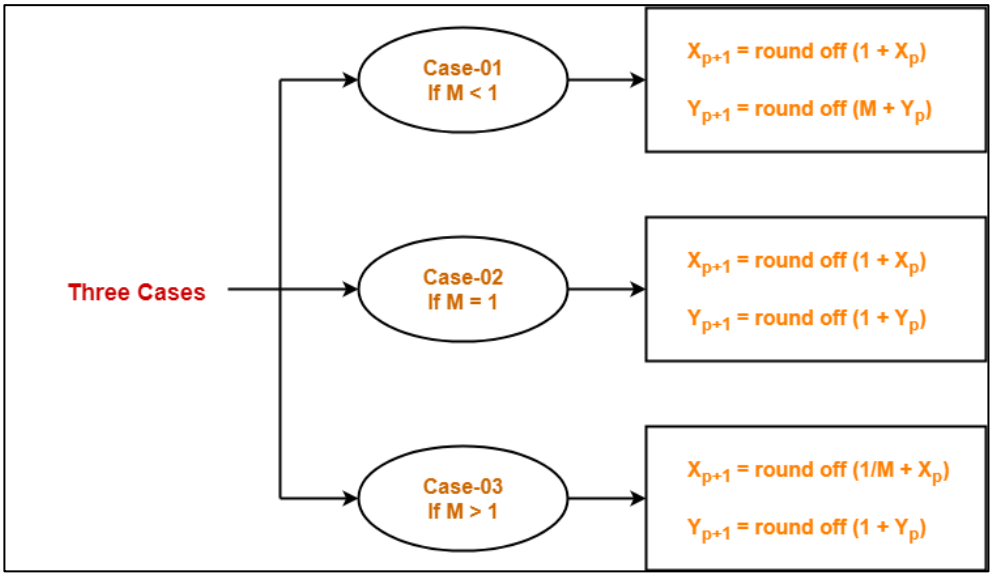
Find the number of steps or points in between the starting and ending coordinates.

if (absolute (ΔX) > absolute (ΔY))

Steps = absolute (ΔX);

else

Steps = absolute (ΔY);



Step-03:

Suppose the current point is (X_p, Y_p) and the next point is (X_{p+1}, Y_{p+1}).

Find the next point by following the below three cases-

Step-04:

Keep repeating Step-03 until the end point is reached or the number of generated new points (including the starting and ending points) equals to the steps count.

B) Bresenham's Algorithm :-

1. START
2. Get Initial Co-ordinates (xi, yi) and final coordinates (xf, yf).
3. Initialize dx and dy to abs (xi, xf) and abs (yi, yf) respectively where abs () represents absolute difference between passed values.
4. Initialize x_change and y_change values on the basis of following conditions.
   a) If xi > xf then x_change = -1 else x_change = 1
   b) If yi > yf then y_change = -1 else y_change = 1
5. If dx is 0, plot vertical line and exit.
6. If dy is 0, plot horizontal line and exit.

7. Initialize x = xi and y = yi.
8. Initialize decision parameter P.
9. If dx > dy:
    a) Set P = 2*dy – dx
    b) Initialize loop variable i to 0.
    c) If P > 0, y = y + y_change, P = P + 2*(dy - dx).
    d) Else, P = P + 2*dy
    e) x = x + x_change
    f) Plot vertex (x, y).
    g) i = i+1
    h) If i < dx, Go To step (c).
10. Else:
    a) Set P = 2*dx – dy
    b) Initialize loop variable i to 0.
    c) If P > 0, x = x + x_change, P = P + 2*(dx - dy)
    d) Else, P = P + 2*dy
    e) y = y + y_change
    f) Plot vertex (x, y)/
    g) i = i + 1.
    h) If i < dy, Go to Step (c).
11. STOP

Conclusion:

1. DDA and Bresenham's Algorithm for line drawing were studied and analyzed mathematically.
2. Above algorithms were implemented using OpenGL library and mouse interface for trapping co-ordinates and assigning menu was used.

Code :-

Bresenhem Algorithm :

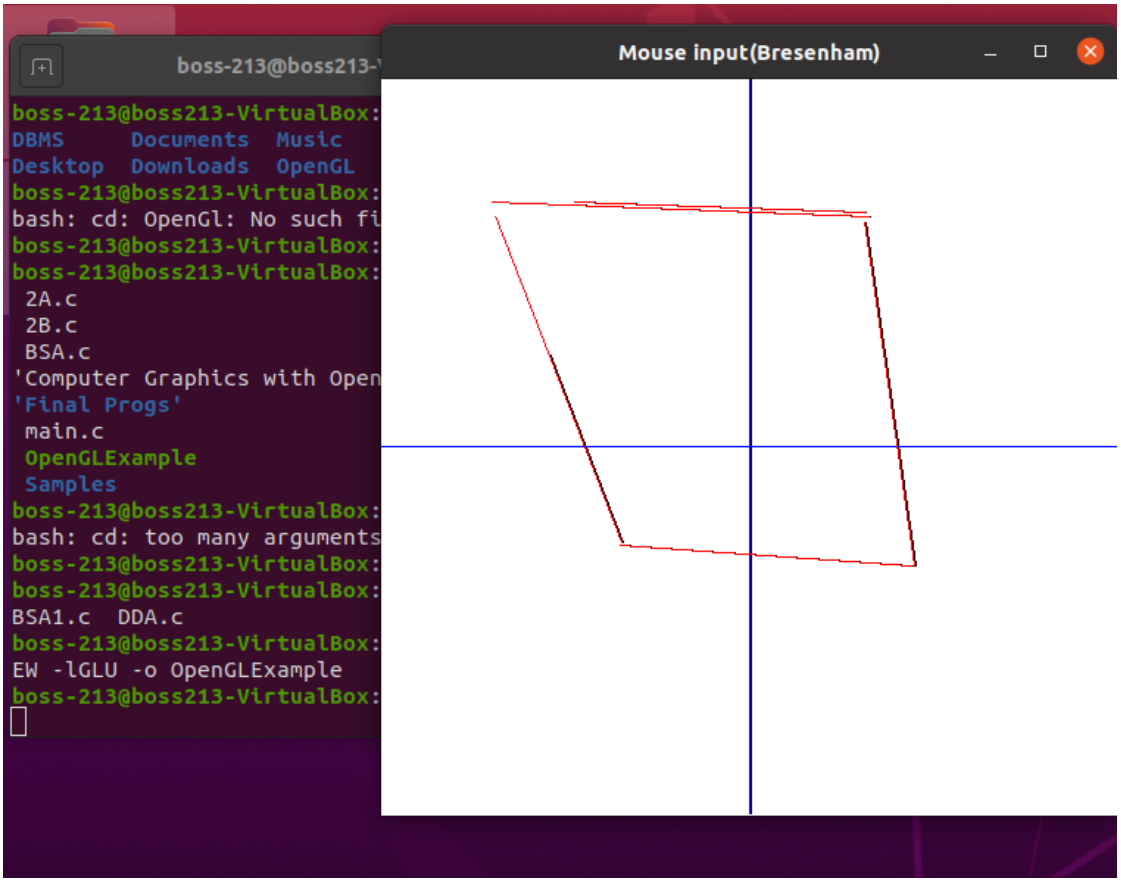
```
#include <GL/freeglut.h>
#include <GL/gl.h>
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;

//Global variables for storing co-ordinates and iteration variable.
int line_co_ordinates[4], i = 0;

void BSA_Algo(int xi, int yi, int xf, int yf)
{
	/*
	 * Input   : Initial (xi, yi) and final co-ordinates of line segment.
	 * Utility : Plot line on the window using Bresenham's line drawing algorithm.
	 * Output  : Line.
	 */
	int P;
	int dx = xf - xi;
	int dy = yf - yi;
	int x = xi, y = yi;
	//Consider absolute value of dx and dy.
	if (dx < 0)
	{
		dx = -dx;
	}
	if (dy < 0)
	{
		dy = -dy;
	}
	//Change of x and y values depends on initial and final points defined by the user.
	int x_change = 1, y_change = 1;
	if(xi > xf)
	{
		x_change = -1;
	}
	if (yi > yf)
	{
		y_change = -1;
	}
	//plot initial point.
	glVertex2i(x, y);
	//For horizontal line, increment y value. Plot the line and return.
	if (dx == 0)
	{
```

```
        glBegin(GL_POINTS);
        for (int i = 1; i<dy; i++)
        {
                y = y + y_change;
                glVertex2i(x, y);
        }
        glEnd();
        return;
}
//For vertical line, increment x value. Plot the line and return.
if (dy == 0)
{
        glBegin(GL_POINTS);
        for (int i = 1; i<dx; i++)
        {
                x = x + x_change;
                glVertex2i(x, y);
        }
        glEnd();
        return;
}
//Calculate vertex position based on descision parameter.
glBegin(GL_POINTS);
if (dx > dy)
{
        P = 2*dy -dx;
        for(int i = 0; i < dx; i++)
        {
                if (P > 0)
                {
                        y += y_change;
                        P = P + 2*(dy - dx);
                }
                else
                {
                        P = P + 2*dy;
                }
                x += x_change;
                glVertex2i(x, y);
        }
}
else
{
        P = 2*dx -dy;
        for(int i = 0; i < dy; i++)
        {
                if (P > 0)
                {
                        x += x_change;
```

```
                        P = P + 2*(dx - dy);
                    }
                    else
                    {
                        P = P + 2*dx;
                    }
                    y += y_change;
                    glVertex2i(x, y);
                }
            }
            glEnd();
}


void mouse_input(int button, int state, int x, int y)
{
        /*
         * Input : Button pressed, state of the button and (x, y) co-ordinates of the mouse
pointer.
         * Utility : Plot the line from initial and final point on the grid.
         * Output : Required line.
         */
         if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN && i < 4)
         {
              line_co_ordinates[i] = x;
              i = i + 1;
              line_co_ordinates[i] = 500 - y;
              i = i + 1;
         }
         if (i == 4)
         {
              glColor3f(1.0, 0.0, 0.0);
              BSA_Algo(line_co_ordinates[0], line_co_ordinates[1], line_co_ordinates[2],
line_co_ordinates[3]);
              glutSwapBuffers();
              i = 0;
         }

}

void render_function()
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 0.0, 1.0);
   gluOrtho2D(0, 500, 0, 500);
   glBegin(GL_POINTS);
   BSA_Algo(0,250,500,250);
   BSA_Algo(250,0,250,500);
```

```
    glEnd();
    glFlush();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Mouse input(Bresenham)");
    glutDisplayFunc(render_function);
    glutMouseFunc(mouse_input);
    glutMainLoop();
    return 0;
}
```

Output :-



DDA Algorithm :

```
#include<GL/glut.h>
#include<stdlib.h>
#include<stdio.h>
```

```
void displayPoint(int x ,int y){
        glColor3f(0,1,0);
        glBegin(GL_POINTS);
                glVertex2i(x,y);
        glEnd();

}

float x01,x2,y01,y2;
int ch;

void SimpleLine(float x1, float y1, float x2 ,float y2){
        float step ;

   float dx = x2-x1;
   float dy = y2-y1;

   if(abs(dx) >abs(dy)){
   step = abs(dx);
   }
   else step = abs(dy);

   float Xinc = dx/(float) step;
   float Yinc = dy/ (float)step;

   float x = x1;
        float y = y1;

   for(int i=0 ; i<=step ;i++){
                displayPoint(x,y);
                x= x + Xinc;
                y= y + Yinc;
   }

 glFlush();

}


void DottedLine(float x1, float y1, float x2 ,float y2){
        float step ;

   float dx = x2-x1;
   float dy = y2-y1;

   if(abs(dx) >abs(dy)){
   step = abs(dx);
   }
```

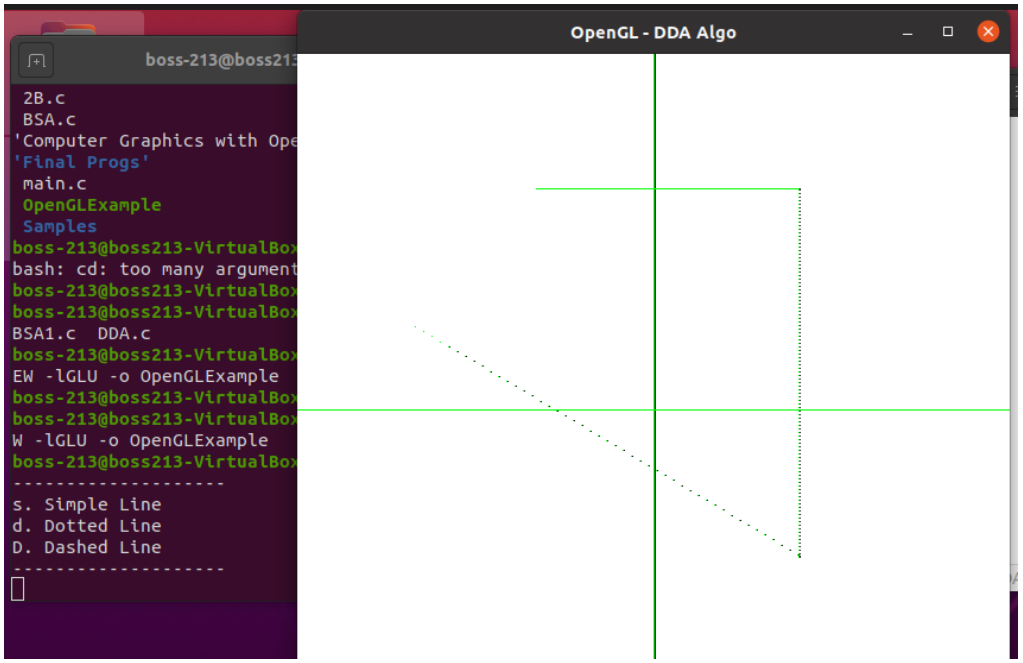
```
    else step = abs(dy);

    float Xinc = dx/(float) step;
    float Yinc = dy/ (float)step;

    float x = x1;
          float y = y1;

          displayPoint(x,y);

    for(int i=0 ; i<=step ;i++){

          x= x + Xinc;
                y= y + Yinc;
          if(i % 3 ==0 ){
                displayPoint(x,y);
          }

    }

 glFlush();

}


void DashedLine(float x1, float y1, float x2 ,float y2){
          float step ;

    float dx = x2-x1;
    float dy = y2-y1;

    if(abs(dx) >abs(dy)){
    step = abs(dx);
    }
    else step = abs(dy);

    float Xinc = dx/(float) step;
    float Yinc = dy/ (float)step;

    float x = x1;
          float y = y1;

          displayPoint(x,y);

    for(int i=0 ; i<=step ;i++){

          x= x + Xinc;
                y= y + Yinc;
          if(i % 7 ==0 ){
```

```
                displayPoint(x,y);
        }

    }

 glFlush();

}


void myMouse(int button, int state, int x, int y ){

        static int xst, yst, pt=0;
        if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN){

                if (pt == 0){
                        xst = x;
                        yst = y;
                        x01 = xst;
                        y01 = yst;
                        pt = pt+1;
                }
                else{

                        x2 = x;
                        y2 = y;

                        if (ch == 1){
                                SimpleLine(xst,yst,x,y);
                        }
                        else if(ch == 2){
                                DottedLine(xst,yst,x,y);
                        }
                        else if (ch == 3) {
                                DashedLine(xst,yst,x,y);
                        }
                        xst=x;
                        yst=y;
                }
        }
        else if (button==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
                pt = 0;
        //Clear Screen
        glFlush();
}

void keyboard(unsigned char key,int x , int y){

                switch(key){
```

```
                        case 's':
                        {
                                ch = 1;
                                glutMouseFunc(myMouse);
                                break;
                        }

                        case 'd':
                        {
                                ch = 2;
                                glutMouseFunc(myMouse);
                                break;
                        }

                        case 'D':
                        {
                                ch = 3;
                                glutMouseFunc(myMouse);
                                break;
                        }
                }

        glutPostRedisplay();
}

void initialize(void)
{

        glClearColor(1.0, 1.0, 1.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);
        // gluOrtho2D(l,r,b,t)
        gluOrtho2D(0,600,600,0);
}


void primitives(void){

        //glClearColor(1.0, 1.0, 1.0, 1.0);
        //glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(1,0,0);
        SimpleLine(0,300,600,300);
        SimpleLine(300,0,300,600);
        glutKeyboardFunc(keyboard);

}


int main(int argc, char** argv)
{
```

```
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE);
        glutInitWindowPosition(0,0);
        glutInitWindowSize(600,600);
        glutCreateWindow("OpenGL - DDA Algo");
        initialize();
        printf("--------------------");
        printf("\ns. Simple Line");
        printf("\nd. Dotted Line");
        printf("\nD. Dashed Line");
        printf("\n--------------------\n");
        glutDisplayFunc(primitives);
        glutMainLoop();
        return 0;
}
```

Output :-



Conclusion : Successfully Implemented DDA and Bresenham's Line Drawing Algorithms using OpenGL

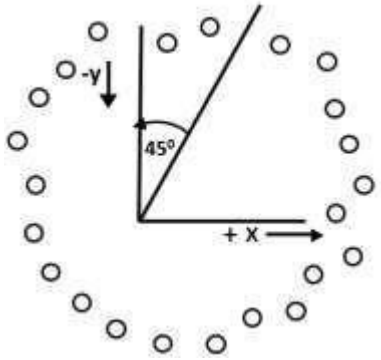
Assignment No 3

Aim: Bresenham's Circle Generation Algorithms using OpenGL.

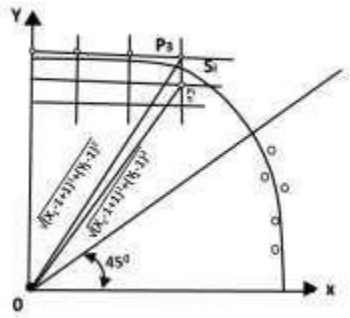
Prob Statements : Implement Bresenham circle drawing algorithm to draw any object. The object should be displayed in all the quadrants with respect to center and radius.

Theory :

Scan-Converting a circle using Bresenham's algorithm works as follows: Points are generated from 90° to 45°, moves will be made only in the +x & -y directions as shown in fig:



The best approximation of the true circle will be described by those pixels in the raster that falls the least distance from the true circle. We want to generate the points from



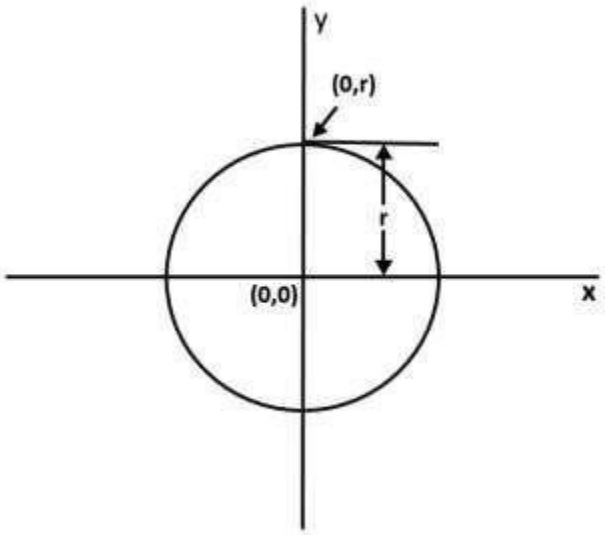
90° to 45°. Assume that the last scan-converted pixel is P1 as shown in fig. Each new point closest to the true circle can be found by taking either of two actions.

1. Move in the x-direction one unit or
2. Move in the x- direction one unit & move in the negative y-direction one unit.

Let D (Si) is the distance from the origin to the true circle squared minus the distance to point P3 squared. D (Ti) is the distance from the origin to the true circle squared minus the distance to point P2 squared. Therefore, the following expressions arise.

$D(Si)=(xi-1+1)^2+ yi-1^2 -r^2$
$D(Ti)=(xi-1+1)^2+(yi-1 -1)^2-r^2$

Since D (Si) will always be +ve & D (Ti) will always be -ve, a decision variable d may be defined as follows:



di=D (Si)+ D (Ti)

Therefore,
di=(xi-1+1)2+ yi-12 -r2+(xi-1+1)2+(yi-1 -1)2-r2

From this equation, we can drive initial values of di as

If it is assumed that the circle is centered at the origin, then at the first step x = 0 & y = r.

Therefore,

$di=(0+1)^2+r^2 -r^2+(0+1)^2+(r-1)^2-r^2$
$=1+1+r^2-2r+1-r^2$
$= 3 - 2r$

Thereafter, if d_i<0,then only x is incremented. xi+1=xi+1

di+1=di+ 4xi+6

& if di≥0,then x & y are incremented

xi+1=xi+1 yi+1 =yi+ 1
di+1=di+ 4 (xi-yi)+10

Algorithm :

Step1: Start Algorithm

Step2: Declare p, q, x, y, r, d variables p, q are coordinates of the center of the circle r is the radius of the circle Step3: Enter the value of r

Step4: Calculate d = 3 - 2r

Step5: Initialize x= r

Step6: Check if the whole circle is scan converted
If x > = y
Stop

Step7: Plot eight points by using concepts of eight-way symmetry. The center is at (p, q). Current active pixel is (x, y).
putpixel (x+p, y+q)
putpixel (y+p, x+q) putpixel (-y+p, x+q) putpixel (-x+p, y+q) putpixel (-x+p, -y+q)
putpixel (-y+p, -x+q)
putpixel (y+p, -x+q)
putpixel (x+p, -y-q)

Step8: Find location of next pixels to be scanned
If d < 0
then d = d + 4x + 6
increment x = x + 1 If
d ≥ 0
then d = d + 4 (x - y) + 10
increment x = x + 1
decrement y = y - 1 **Step9:** Go to
step 6

Step10: Stop Algorithm

Conclusion :

Learned to implement Bresenham's circle drawing algorithm

Advantages

- It is a simple algorithm.
- It can be implemented easily
- It is totally based on the equation of circle i.e. $x^2 + y^2 = r^2$

Disadvantages

- There is a problem of accuracy while generating points.
- This algorithm is not suitable for complex and high graphic images.

CODE :

```
#include <iostream>
#include <GL/glut.h>
#include <math.h>

using namespace std;

//Defalut radius of circle int
cx=300,cy=300,R=70; bool
flag=1;

//Color struct      struct
color{
        GLubyte r,g,b;
};

//init function for init. void
init()
{
      glClearColor(1,1,1,0);
glClear(GL_COLOR_BUFFER_BIT);
gluOrtho2D(0,600,0,600);
        glColor3f(0,0,0);
}

//ploat the pixel (x,y) void
plotpixel(int x,int y)
{
        glPointSize(1.5);
        glBegin(GL_POINTS);
            glVertex2i(x,y);
```

```
glEnd();
        glFlush();
}

//ploat the points using the circle sym. void
octant(int xc,int yc,int x,int y)
{
        plotpixel(xc+x,yc+y); plotpixel(xc+y,yc+x);
                                plotpixel(xc+y,yc-
x);     plotpixel(xc+x,yc-y);

        plotpixel(xc-x,yc-y);       plotpixel(xc-y,yc- x);
        plotpixel(xc-y,yc+x);       plotpixel(xc-
x,yc+y);

}
//mid point circle drawing      void
circleMP(int xc,int yc,int r)
{
        int p=1-r,x=0,y=r;
        //loop til the x become y equal to radius (r,r)
        while(x<y)
        {
                octant(xc,yc,x,y);
                x++;
            if(p>0)             //if p>0 decrement the y and 2(x-y)+1
        y--,p+=2*(x-y)+1;               else                    //if p<=0 add 2x+1 to p
                    p+=2*x+1;
        }
}

//convert the rad to deg
double ang(int q)
{
        return (double)q*3.142/180;
}
void plottofill(int x,int y,color c)
{
        glPointSize(1.0);
glColor3ub(c.r,c.g,c.b);

        glBegin(GL_POINTS);
            glVertex2i(x,y);
glEnd();
        glFlush();
```

```
}
void seedfill(int x,int y,color oc,color nc)
{
        color c;
      glReadPixels(x,y,1,1,GL_RGB,GL_UNSIGNED_BYTE,&c);
if(c.r==oc.r&&c.b==oc.b&&c.g==oc.g)
        {
            plottofill(x,y,nc); seedfill(x+1,y,oc,nc);
                        seedfill(x- 1,y,oc,nc);
                        seedfill(x,y+1,oc,nc);
      seedfill(x,y-1,oc,nc);
        }
}
//Draw all the Cirlces
void drawcircles(int x,int y,int r)
{
        circleMP(x,y,r);

      circleMP(x+2*r,y,r);        circleMP(x-
2*r,y,r);

      circleMP(x+2*r*cos(ang(60)),y+2*r*sin(ang(60)),r);
circleMP(x-2*r*cos(ang(60)),y+2*r*sin(ang(60)),r); circleMP(x-
2*r*cos(ang(60)),y-2*r*sin(ang(60)),r);
        circleMP(x+2*r*cos(ang(60)),y-2*r*sin(ang(60)),r);


        circleMP(x,y,3*r);


        circleMP(x,y,(float)2*r-r*(0.20));
}

//Display Function
void draw() {

}
//Clear the whole screen
void clear_screen()
{
         glClearColor(1,1,1,0);
glClear(GL_COLOR_BUFFER_BI
T);
}

//Mouse click function
```

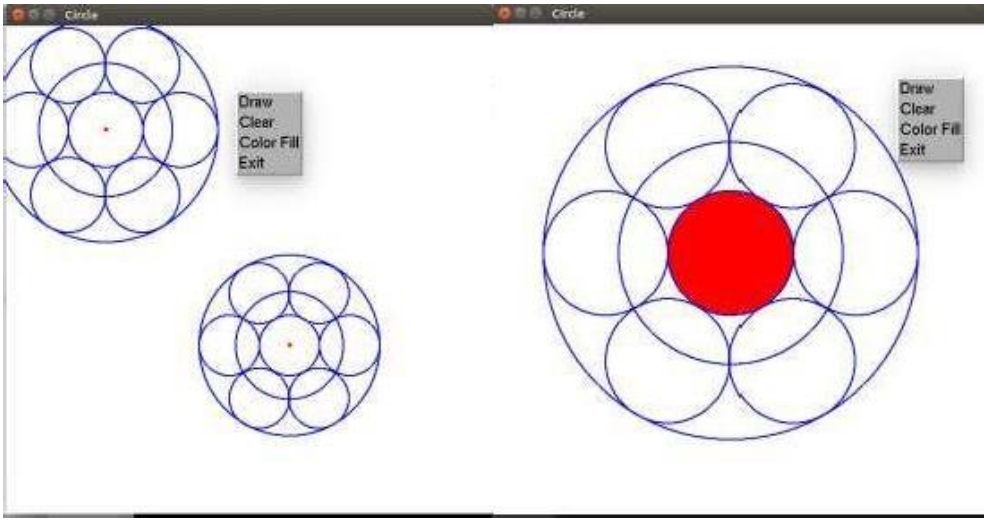
```
void mouseClick(int button,int state,int x,int y)
{
        cout<<"Mouse Clicked"<<endl;
```

```
        //First point to get the xc,yc
        if(flag&&button==GLUT_LEFT_BUTTON&&state==GLUT_
                                        DOWN)
        {
                cout<<"Center Found"<<endl;
                cx=x,cy=600-y;
                glPointSize(5.0);
                glColor3f(1,0,0);
                glBegin(GL_POINTS);
                        glVertex2i(x,600-y);
                glEnd();
                glFlush();
                flag=0;
        }
        //find the radius of the circle
        else if
        (!flag&&button==GLUT_LEFT_BUTTON&&state==GLUT_DOWN)
        {
                cout<<"Ohhho !!, I got a radius"<<endl;
                glColor3f(0,0,1);
                glPointSize(1.0);
                glBegin(GL_POINTS);
                        glVertex2i(x,600-y);
                glEnd();
                glFlush();
                R=abs(x-cx);
                flag=1;
        }
}


//Menu function void
menu(int ch) {
        color oc={255,255,255};
        color nc={255,0,0};
        switch(ch)
        {
                case 1:
                        drawcircles(cx,cy,R);

case 2:
                    case 4:

case 3:
```

```
break;                  ered Circle"<<endl; seedfill(cx+5,cy,oc,nc);
                        break;
        c
l
e
a
r
_
s
c
r
e
e
n
(
)
;
b
r
e
a
k
;

        c
        o
        u
        t
        <
        <
        "
        F
        i
        l
        l
        t
        h
        e
        C
        e
        n
        t
```

```
    exit(0);                    break;
            }

}
int main(int agrc,char ** agrv)
{
            glutInit(&agrc,agrv);
          glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowPosition(0,0); glutInitWindowSize(600,600);
            glutCreateWindow("Circle");
            init();
          glutDisplayFunc(draw);
glutCreateMenu(menu);
glutAddMenuEntry("Draw",1);
glutAddMenuEntry("Clear",2);
glutAddMenuEntry("Color Fill",3);
glutAddMenuEntry("Exit",4);
glutAttachMenu(GLUT_RIGHT_BUTTON);
glutMouseFunc(mouseClick);
            glutMainLoop();
}
```

OUTPUT :



Conclusion :

Successfully Implemented Bresenham's Circle Generation Algorithms using OpenGL

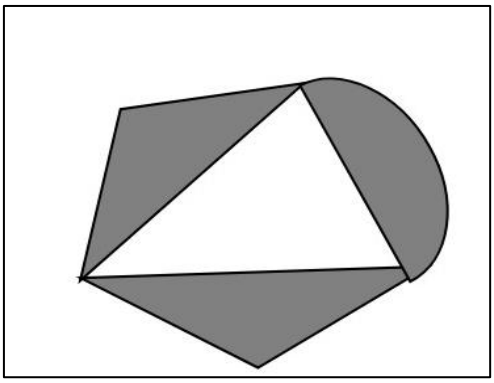
Assignment No 4

Aim: Polygon Filling Algorithms using OpenGL.

Prob Statements : Implement the following polygon filling methods : i) Flood fill / Seed fill ii) Boundary fill ; using mouse click, keyboard interface and menu driven programming

Theory :

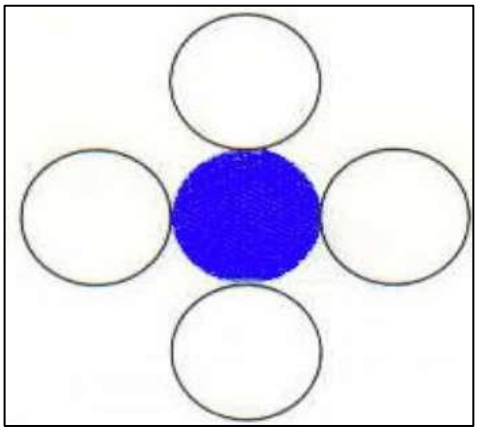
1) Flood Fill / Seed Fill Algorithm :-
In this method, a point or seed which is inside region is selected. This point is called a seed point. Then four connected approaches or eight connected approaches is used to fill with specified color.
The flood fill algorithm has many characters similar to boundary fill. But this method is more suitable for filling multiple colors boundary. When boundary is of many colors and interior is to be filled with one color we use this algorithm.



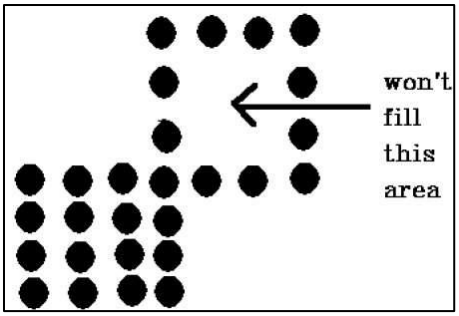
In fill algorithm, we start from a specified interior point (x, y) and reassign all pixel values are currently set to a given interior color with the desired color. Using either a 4-connected or 8-connected approaches, we then step through pixel positions until all interior points have been repainted.

2) Boundary Fill Algorithm :-
The boundary fill algorithm works as its name. This algorithm picks a point inside an object and starts to fill until it hits the boundary of the object. The color of the boundary and the color that we fill should be different for this algorithm to work.
In this algorithm, we assume that color of the boundary is same for the entire object. The boundary fill algorithm can be implemented by 4-connected pixels or 8-connected pixels.

(i) 4-Connected Polygon :-
In this technique 4-connected pixels are used as shown in the figure. We are putting the pixels above, below, to the right, and to the left side of the current pixels and this process will continue until we find a boundary with different color.



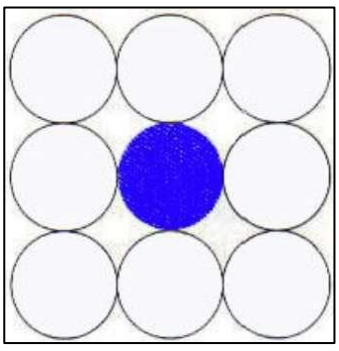
There is a problem with this technique. Consider the case as shown below where we tried to fill the entire region. Here, the image is filled only partially. In such cases, 4-connected pixels technique cannot be used.



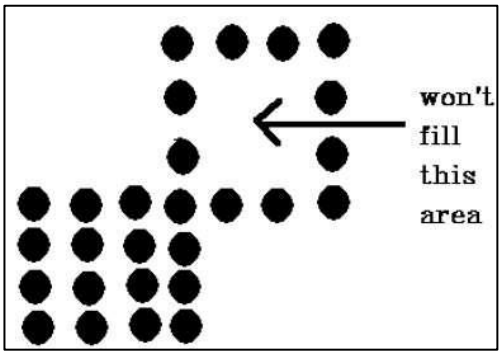
(ii) 8-Connected Polygon :-

In this technique 8-connected pixels are used as shown in the figure. We are putting pixels above, below, right and left side of the current pixels as we were doing in 4-connected technique.

In addition to this, we are also putting pixels in diagonals so that entire area of the current pixel is covered. This process will continue until we find a boundary with different color.



The 4-connected pixel technique failed to fill the area as marked in the following figure which won't happen with the 8-connected technique.



Algorithm:

1) Flood Fill / Seed Fill Algorithm :-

1. **Procedure floodfill (x, y,fill_ color, old_color: integer)**
2. **If (getpixel (x, y)=old_color)**
3. **{**
4. **setpixel (x, y, fill_color);**
5. **fill (x+1, y, fill_color, old_color);**
6. **fill (x-1, y, fill_color, old_color);**
7. **fill (x, y+1, fill_color, old_color);**
8. **fill (x, y-1, fill_color, old_color);**
9. **}**
10. **}**

2) Boundary Fill Algorithm :-

(i) 4-Connected Polygon :-

Step 1 – Initialize the value of seed point seedx, seedyseedx, seedy, fcolor and dcol.
Step 2 – Define the boundary values of the polygon.
Step 3 – Check if the current seed point is of default color, then repeat the steps 4 and 5 till the boundary pixels reached. If getpixel(x, y) = dcol then repeat step 4 and 5

Step 4 – Change the default color with the fill color at the seed point.
setPixel(seedx, seedy, fcol)
Step 5 – Recursively follow the procedure with four neighborhood points.
FloodFill (seedx – 1, seedy, fcol, dcol)
FloodFill (seedx + 1, seedy, fcol, dcol)
FloodFill (seedx, seedy - 1, fcol, dcol)
FloodFill (seedx – 1, seedy + 1, fcol, dcol)
Step 6 – Exit

(ii) 8-Connected Polygon :-

Step 1 – Initialize the value of seed point seedx, seedyseedx, seedy, fcolor and dcol.
Step 2 – Define the boundary values of the polygon.
Step 3 – Check if the current seed point is of default color then repeat the steps 4 and 5 till the boundary pixels reached
If getpixel(x,y) = dcol then repeat step 4 and 5
Step 4 – Change the default color with the fill color at the seed point.
setPixel(seedx, seedy, fcol)
Step 5 – Recursively follow the procedure with four neighbourhood points
FloodFill (seedx – 1, seedy, fcol, dcol)
FloodFill (seedx + 1, seedy, fcol, dcol)
FloodFill (seedx, seedy - 1, fcol, dcol)
FloodFill (seedx, seedy + 1, fcol, dcol)
FloodFill (seedx – 1, seedy + 1, fcol, dcol)
FloodFill (seedx + 1, seedy + 1, fcol, dcol)
FloodFill (seedx + 1, seedy - 1, fcol, dcol)
FloodFill (seedx – 1, seedy - 1, fcol, dcol)
Step 6 – Exit

Advantages Flood Fill :-

- Flood fill colors an entire area in an enclosed figure through interconnected pixels using a single color.
- It is an easy way to fill color in the graphics. One just takes the shape and starts flood fill.
- The algorithm works in a manner so as to give all the pixels inside the boundary the same color leaving the boundary and the pixels outside.
- Flood Fill is also sometimes referred to as Seed Fill as you plant a seed and more and more seeds are planted by the algorithm. Each seed takes the responsibility of giving the same color to the pixel at which it is positioned.

Disadvantages of Flood Fill :-

- Very slow algorithm
- May be fail for large polygons
- Initial pixel required more knowledge about surrounding pixels.

Disadvantages of Boundary-Fill over Flood-Fill :-

- In boundary-fill algorithms each pixel must be compared against both the new colour and the boundary colour. In flood-fill algorithms each pixel need only be compared against the new colour. Therefore flood-fill algorithms are slightly faster.
- Boundary-fill algorithms can leak. There can be no leakage in flood-fill algorithms.

Advantages of Boundary-Fill over Flood-Fill :-

- Flood-fill regions are defined by the whole of the region. All pixels in the region must be made the same colour when the region is being created. The region cannot be translated, scaled or rotated.

- 4-connected boundary-fill regions can be defined by lines and arcs. By translating the line and arc endpoints we can translate, scale and rotate the whole boundary-fill region. Therefore 4-connected boundary-fill regions are better suited to modelling.

Code :-

```
#include<stdio.h>
#include<GL/gl.h>
#include<GL/glu.h>
#include<GL/glut.h>
#include<math.h>

/*draw chess pattern rotate it and fill it with different colours*/

typedef struct pixel
{
	GLubyte r,g,b;
}pixel;

pixel f_color,b_color;

float mat1[20][3];
float ans1[20][3];
float trans1[3][3];
int ch=1;

void initial_co()
{
	int i,y,x;

	y=90;
	//horizontal lines
	for(i=0;i<10;i+=2)
	{
		//first point
		mat1[i][0]=90;
		mat1[i][1]=y;
		mat1[i][2]=1;

		//second point
		mat1[i+1][0]=210;
		mat1[i+1][1]=y;
		mat1[i+1][2]=1;

		y+=30;
	}

	x=90;
	//vertical lines
	for(i;i<20;i+=2)
	{
```

```
		//first point
		mat1[i][0]=x;
		mat1[i][1]=90;
		mat1[i][2]=1;

		//second point
		mat1[i+1][0]=x;
		mat1[i+1][1]=210;
		mat1[i+1][2]=1;

		x+=30;
	}
}

void rotate_fig()
{
	int i,j,k;
	float theta;
	theta=45*3.14/180;

	/*----------------translation to origin ------------------------------*/
	for(i=0;i<3;i++)
	{
		for(j=0;j<3;j++)
		{
			if(i==j)
				trans1[i][j]=1;
			else
				trans1[i][j]=0;
		}
	}
	trans1[2][0]=trans1[2][1]=-150;
	/*
		trans1= 1  0  0
			0  1  0
			tx ty 1
	*/

	for(i=0;i<20;i++)
	{
		for(j=0;j<3;j++)
		{
			ans1[i][j]=0;
			for(k=0;k<3;k++)
				ans1[i][j]+=mat1[i][k]*trans1[k][j];
		}
	}

	/*----------------------rotation at origin--------------------------------*/
```

```
for(i=0;i<3;i++)
{
	for(j=0;j<3;j++)
	{
		if(i==j)
			trans1[i][j]=1;
		else
			trans1[i][j]=0;
	}
}

trans1[0][0]=trans1[1][1]=cos(theta);
trans1[0][1]=sin(theta);
trans1[1][0]=-sin(theta);
/*
	trans1=  cos sin 0
		-sin cos 0
		 0    0  1
*/

for(i=0;i<20;i++)
{
	for(j=0;j<3;j++)
	{
		mat1[i][j]=0;
		for(k=0;k<3;k++)
			mat1[i][j]+=ans1[i][k]*trans1[k][j];
	}
}

/*----------------------translation back-----------------------------*/
for(i=0;i<3;i++)
{
	for(j=0;j<3;j++)
	{
		if(i==j)
			trans1[i][j]=1;
		else
			trans1[i][j]=0;
	}
}
trans1[2][0]=trans1[2][1]=150;

for(i=0;i<20;i++)
{
	for(j=0;j<3;j++)
	{
		ans1[i][j]=0;
		for(k=0;k<3;k++)
```

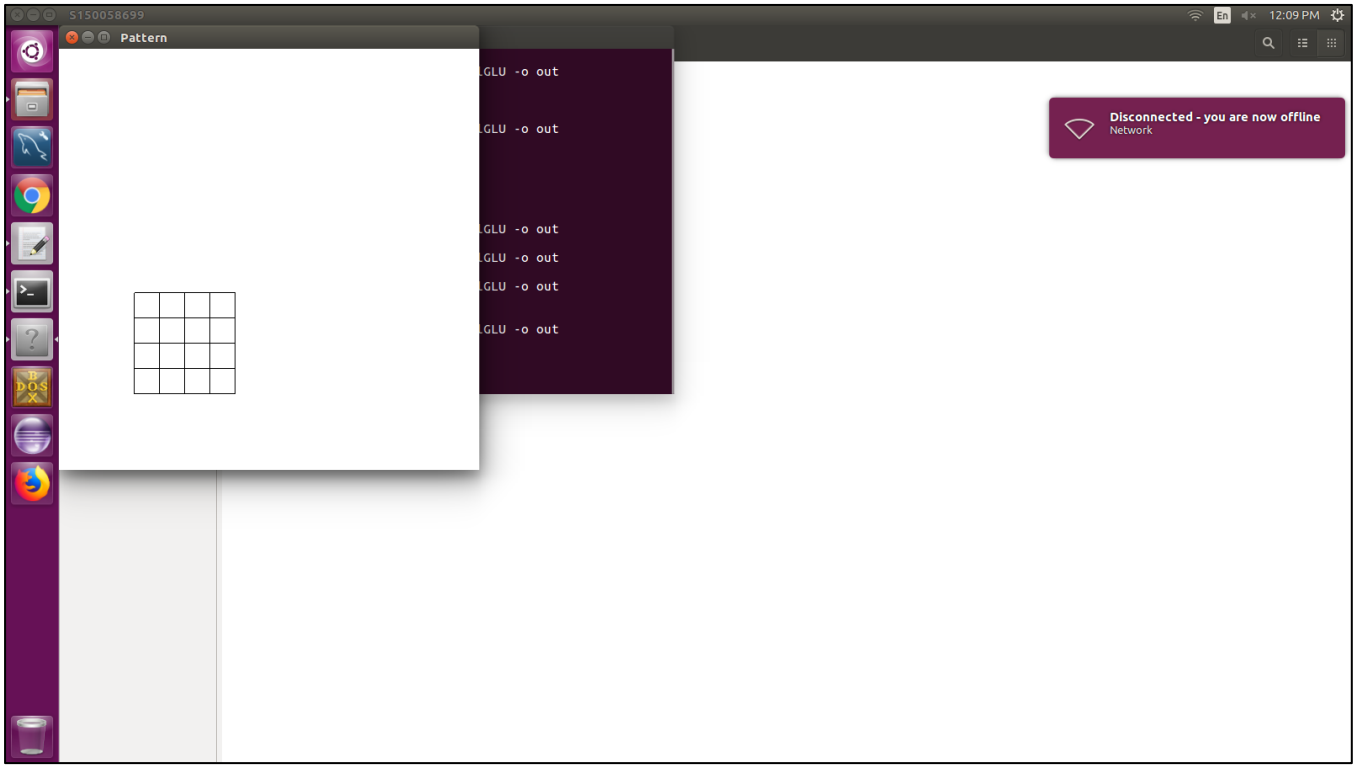
```
                                ans1[i][j]+=mat1[i][k]*trans1[k][j];
                }
        }

}

void boundary_fill(int x,int y)
{
        pixel c;

        glReadPixels(x,y,1,1,GL_RGB,GL_UNSIGNED_BYTE,&c);//values are put into c

        //if color not equal to backgroung color and filling color put color
        if((c.r!=b_color.r || c.g!=b_color.g || c.b!=b_color.b )&&(c.r!=f_color.r || c.g!=f_color.g ||
c.b!=f_color.b ))
        {
                glColor3ub(f_color.r,f_color.g,f_color.b);//set fill color for pixel
                glBegin(GL_POINTS);
                        glVertex2d(x,y);//put pixel
                glEnd();
                glFlush();
                boundary_fill(x+1,y);//right pixel
                boundary_fill(x-1,y);//left pixel
                boundary_fill(x,y+1);//upper pixel
                boundary_fill(x,y-1);//lower pixel
        }
}

void before()
{
        int i;
        initial_co();
        glBegin(GL_LINES);//draws the new figure
                for(i=0;i<20;i+=2)
                {
                        glVertex2f(mat1[i][0],mat1[i][1]);
                        glVertex2f(mat1[i+1][0],mat1[i+1][1]);
                }
        glEnd();
        glFlush();
}

void figure()
{
        glClear(GL_COLOR_BUFFER_BIT);
        int i;
        float factor=30*cos(45*3.14/180);

        rotate_fig();//rotates the figure about the middle point (150,150)
```

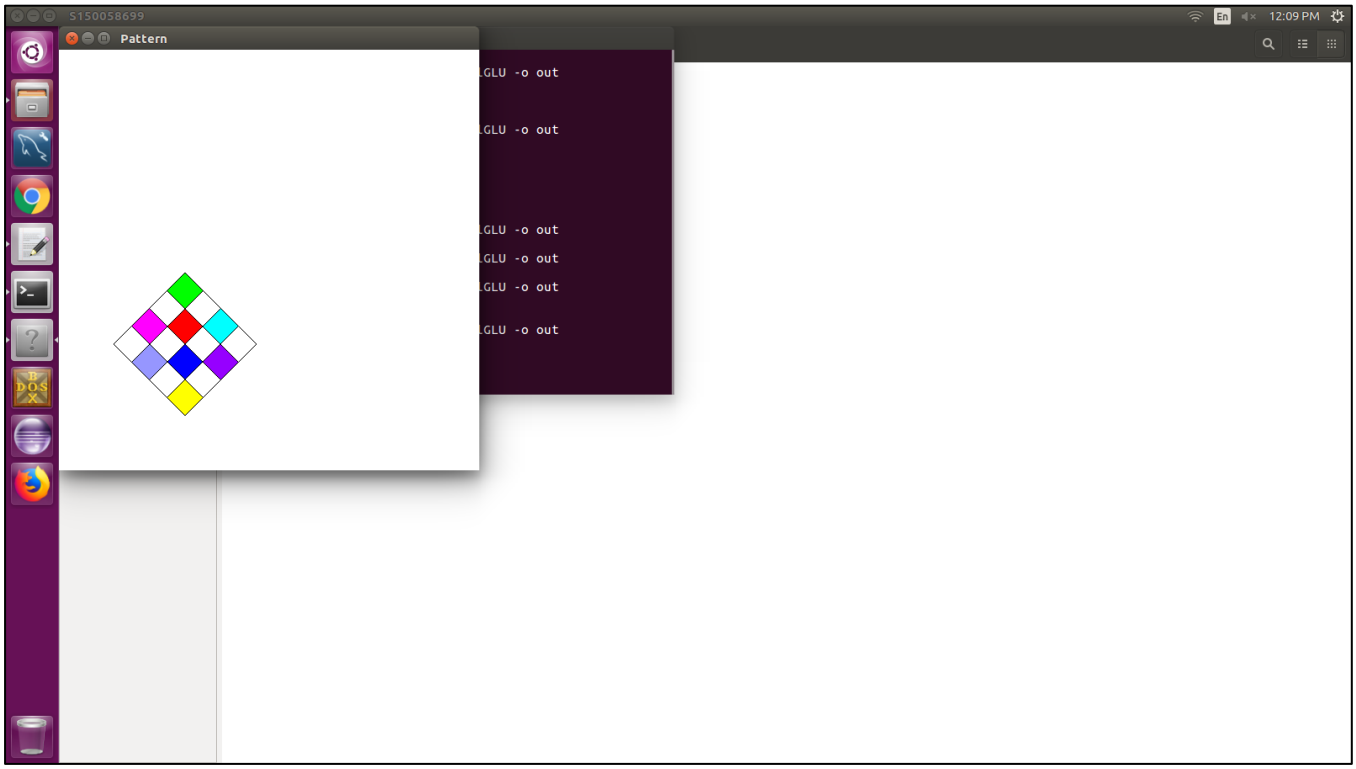
```
glBegin(GL_LINES);//draws the new figure
        for(i=0;i<20;i+=2)
        {
                glVertex2f(ans1[i][0],ans1[i][1]);
                glVertex2f(ans1[i+1][0],ans1[i+1][1]);
        }
glEnd();
glFlush();

//filling the boxes with colours
//red
boundary_fill(150,150+factor);

//green
f_color.r=0;
f_color.g=255;
f_color.b=0;
boundary_fill(150,150+3*factor);

//blue
f_color.r=0;
f_color.g=0;
f_color.b=255;
boundary_fill(150,150-factor);

//yellow
f_color.r=255;
f_color.g=255;
f_color.b=0;
boundary_fill(150,150-3*factor);

//light blue
f_color.r=0;
f_color.g=255;
f_color.b=255;
boundary_fill(150+2*factor,150+factor);

//pink
f_color.r=255;
f_color.g=0;
f_color.b=255;
boundary_fill(150-2*factor,150+factor);

//purple
f_color.r=150;
f_color.g=0;
f_color.b=255;
boundary_fill(150+2*factor,150-factor);
```

```
	//light violet
	f_color.r=150;
	f_color.g=150;
	f_color.b=255;
	boundary_fill(150-2*factor,150-factor);
}

void mouse_click(int btn,int state,int x,int y)
{
	//left click shows and changes the figure
	if(btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
	{
		switch(ch)
		{
		case 1:
			before();//initial figure
			ch=2;
			break;
		case 2:
			figure();//after transformation
			ch=3;
			break;
		case 3:
			break;
		}
	}
}

void init_func()//empty function doesnt do anything
{
	glFlush();
}

void Init()
{
	glClearColor(1.0,1.0,1.0,0.0);//sets the background colour
	glClear(GL_COLOR_BUFFER_BIT);
	glColor3f(0.0,0.0,0.0);//sets the drawing colour
	gluOrtho2D(0,500,0,500);//sets the co ordinates
}

int main(int argc,char **argv)
{

	//border color
	b_color.r=b_color.g=b_color.b=0;

	//fill color
```

```
        f_color.r=255;
        f_color.g=0;
        f_color.b=0;

        glutInit(&argc,argv);//initializing the library
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);//setting the display mode
        glutInitWindowPosition(0,0);//position of the window
        glutInitWindowSize(500,500);//size of the window
        glutCreateWindow("Pattern");//name of the window
        Init();//initializes the background colour and co ordinates
        glutDisplayFunc(init_func);//displays the function
        glutMouseFunc(mouse_click);//to display before and after figures
        glutMainLoop();//keeps the program open until closed
        return 0;
}
```

Output :-





Conclusion: Successfully implemented Flood fill Polygon Filling Algorithms using OpenGL.

Assignment No 5

Aim: Polygon Clipping Algorithms using OpenGL.

Prob Statements : Implement Cohen Sutherland polygon clipping method to clip the polygon with respect the viewport and window. Use mouse click, keyboard interface

Theory :

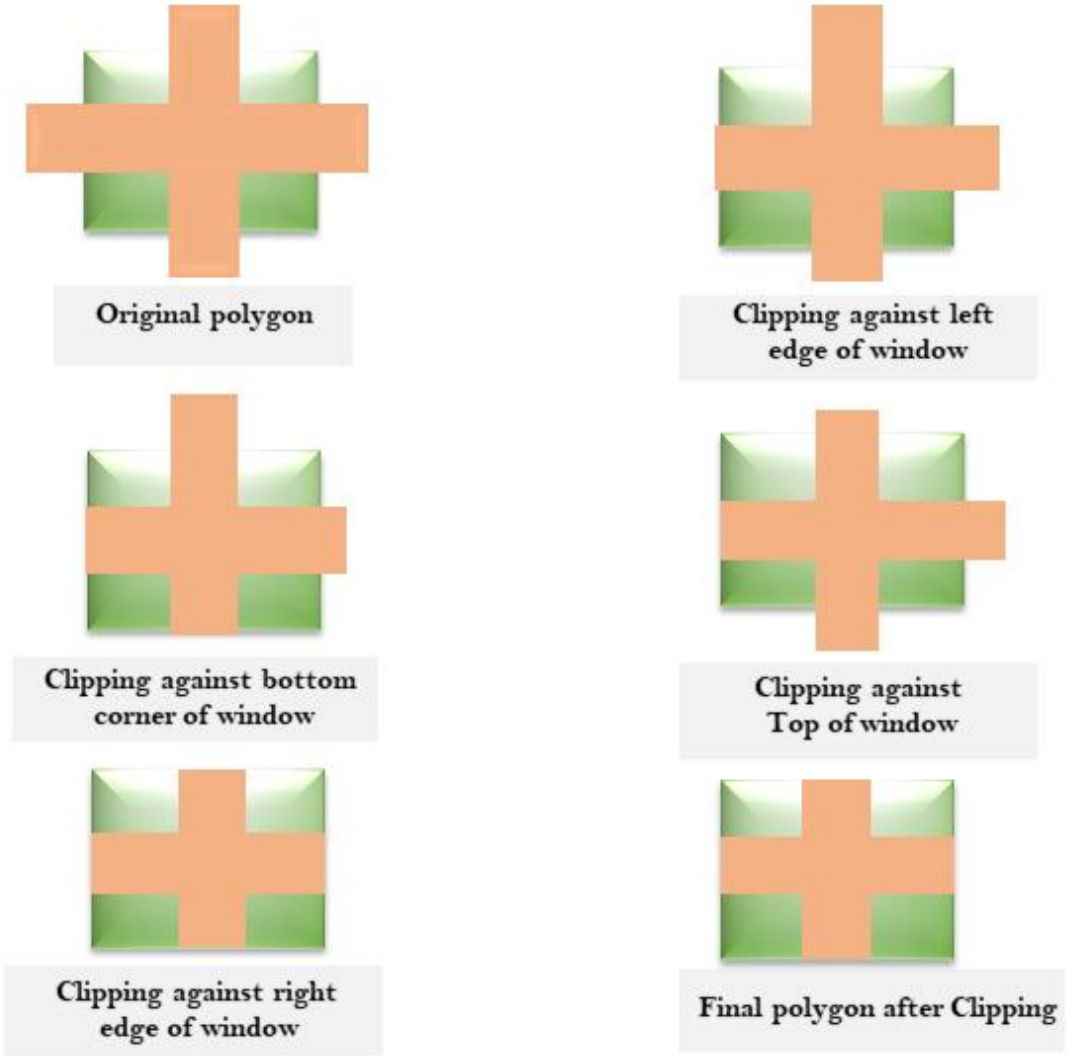
A) Sutherland – Hodgeman Algorithm for Polygon Clipping.

Sutherland - Hodgeman Polygon Clipping is performed by processing the boundary of polygon against each window corner or edge. First of all, entire polygon is clipped against one edge, then resulting polygon is considered, then the polygon is considered against the second edge, so on for all four edges.

Four possible situations while processing

1. If the first vertex is an outside the window, the second vertex is inside the window. Then second vertex is added to the output list. The point of intersection of window boundary and polygon side (edge) is also added to the output line.
2. If both vertexes are inside window boundary. Then only second vertex is added to the output list.
3. If the first vertex is inside the window and second is an outside window. The edge which intersects with window is added to output list.
4. If both vertices are the outside window, then nothing is added to output list.

Following figures shows original polygon and clipping of polygon against four windows.



Disadvantage of Cohen Hodgeman Algorithm:

This method requires a considerable amount of memory. The first of all polygons are stored in original form. Then clipping against left edge done and output is stored. Then clipping against right edge done, then top edge. Finally, the bottom edge is clipped. Results of all these operations are stored in memory. So wastage of memory for storing intermediate polygons.

B) Cohen Sutherland Algorithm for Line Clipping

Cohen Sutherland Line Clipping Algorithm:

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 lie within the window. A line is visible and will be displayed as it is.

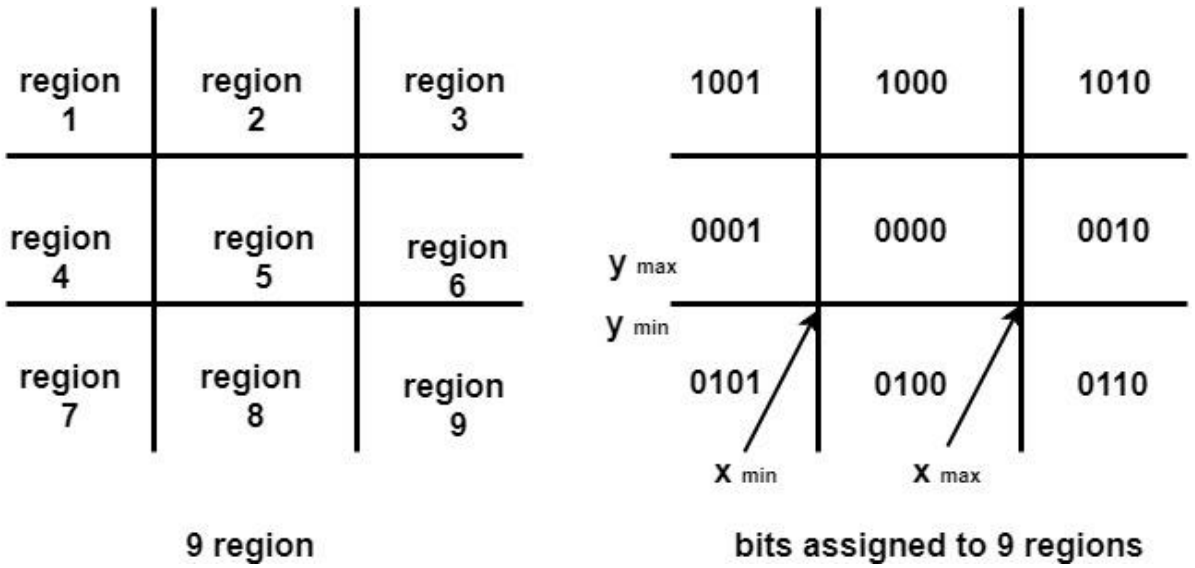
2. Not Visible: If a line lies outside the window, it will be invisible and rejected. Such lines will not display. If any one of the following inequalities is satisfied, then the line is considered invisible. Let A (x_1, y_2) and B (x_2, y_2) are endpoints of line.

x_{min}, x_{max} are coordinates of the window.

y_{min}, y_{max} are also coordinates of the window.

$x_1 > x_{max}$
$x_2 > x_{max}$
$y_1 > y_{max}$
$y_2 > y_{max}$
$x_1 < x_{min}$
$x_2 < x_{min}$
$y_1 < y_{min}$
$y_2 < y_{min}$

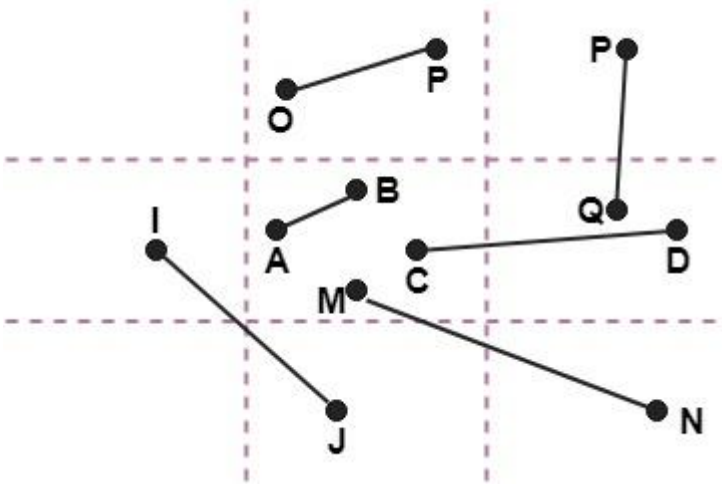
3. Clipping Case: If the line is neither visible case nor invisible case. It is considered to be clipped case. First of all, the category of a line is found based on nine regions given below. All nine regions are assigned codes. Each code is of 4 bits. If both endpoints of the line have end bits zero,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



9 region bits assigned to 9 regions

The centre 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

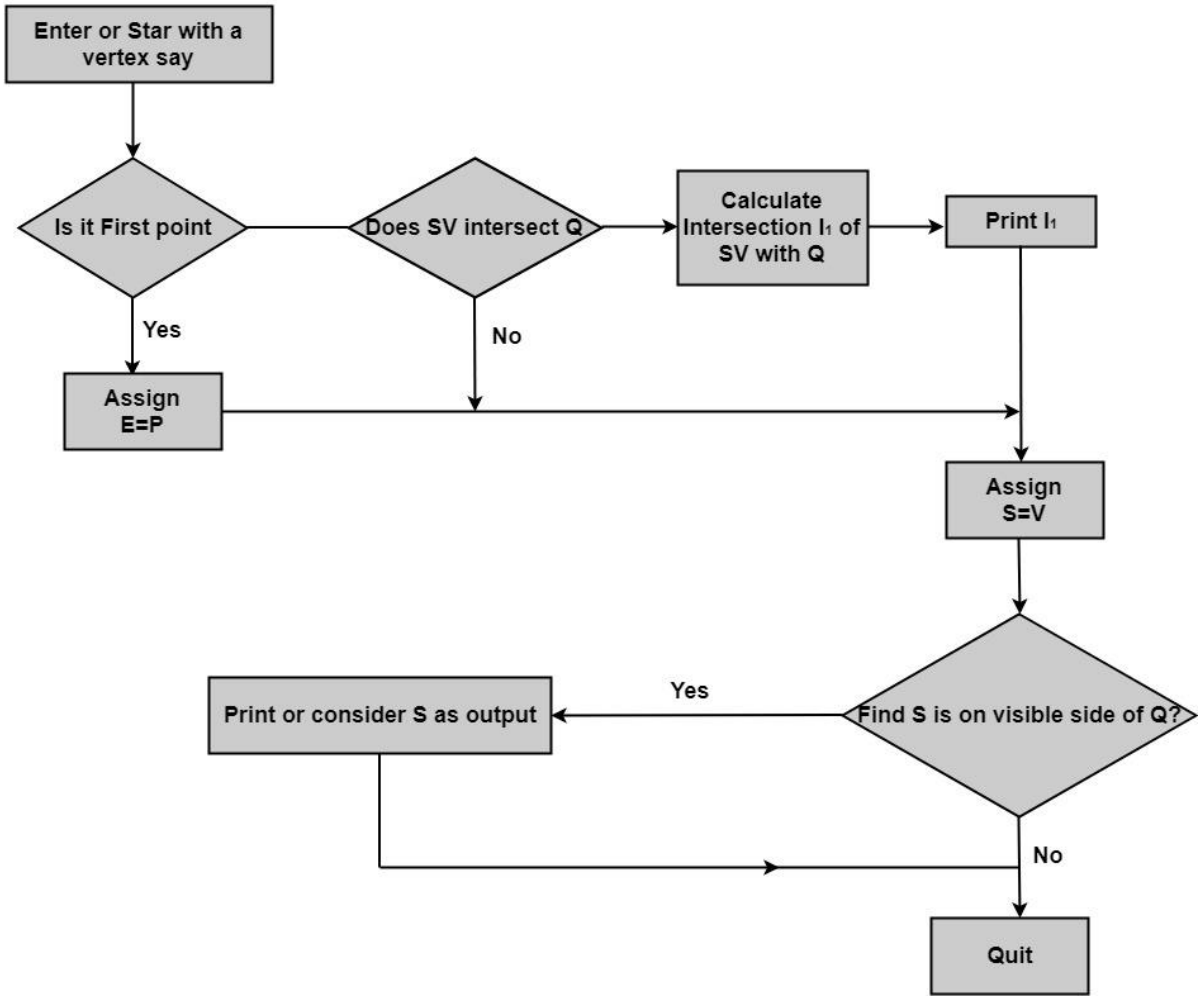
Advantage of Cohen Sutherland Line Clipping:

1. It calculates end-points very quickly and rejects and accepts lines quickly.
2. It can clip pictures much large than screen size.

Algorithm:

A) Sutherland Hodgeman's Polygon Clipping Algorithm:

Sutherland Hodgemen Algorithm:



B) Cohen Sutherland Line Clipping algorithm:

1. Calculate positions of both endpoints of the line
2. Perform OR operation on both of these end-points
3. If the OR operation gives 0000
   Then
   line is considered to be visible
   else
   Perform AND operation on both endpoints
   If And ≠ 0000
   then the line is invisible
   else
   And=0000
   Line is considered the clipped case.

4. If a line is clipped case, find an intersection with boundaries of the window
   $m = (y_2-y_1) \times (x_2-x_1)$

a) If bit 1 is "1" line intersects with left boundary of rectangle window
   $y_3=y_1+m(x-X_1)$

where X = X_{wmin}
where X_{wmin} is the minimum value of X co-ordinate of window

b) If bit 2 is "1" line intersect with right boundary
$y_3=y_1+m(X-X_1)$
where X = X_{wmax}
where X more is maximum value of X co-ordinate of the window

c) If bit 3 is "1" line intersects with bottom boundary
$X_3=X_1+(y-y_1)/m$
where y = y_{wmin}
y_{wmin} is the minimum value of Y co-ordinate of the window

d) If bit 4 is "1" line intersects with the top boundary
$X_{3=X}1+(y-y_1)/m$
where y = y_{wmax}
y_{wmax} is the maximum value of Y co-ordinate of the window

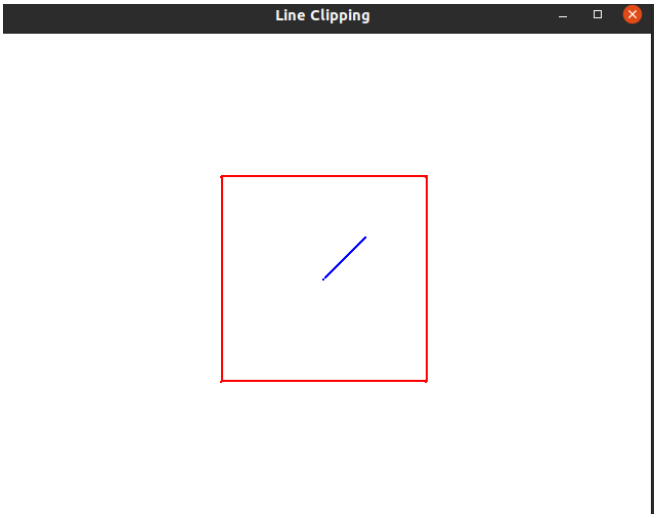
Input:

1. The program asks for the initial and final co-ordinates of the line and the line is plotted after the creation of the window.
2. The viewport is highlighted in the window and the line is plotted in blue.
3. To run the algorithm, 'C' key must be pressed.
4. The Inputs are given for the following cases:
    a) Both end points lie inside the viewport. [(0, 0) and (40, 40)]
    b) One end point lies inside and the other end point lies outside the viewport. [(0, 0) and (320, 240)]

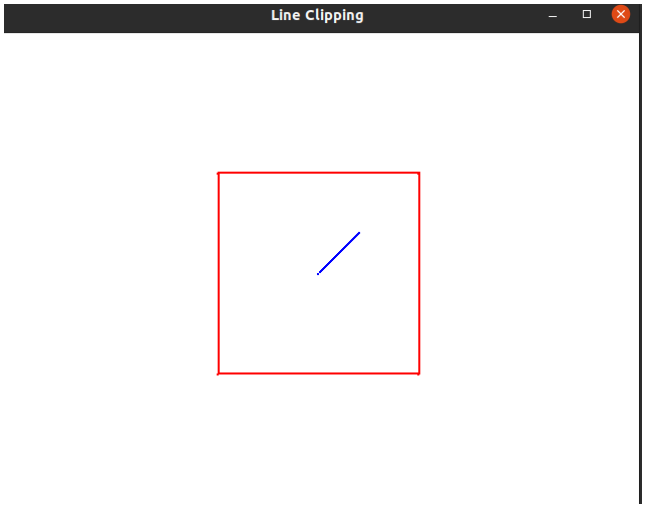
Output:

A) Both end points lie inside the viewport [(0, 0) and (40, 40)]

Input:

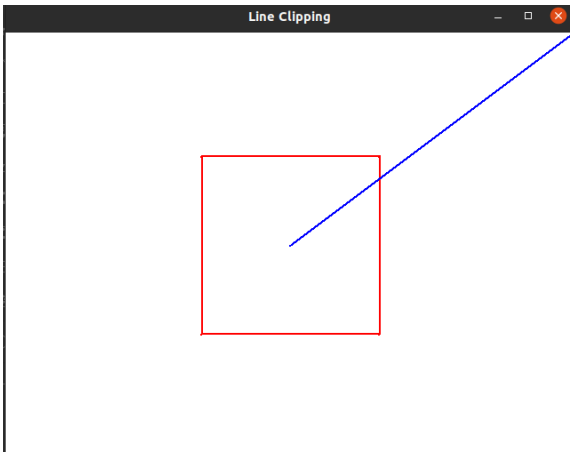


Output:

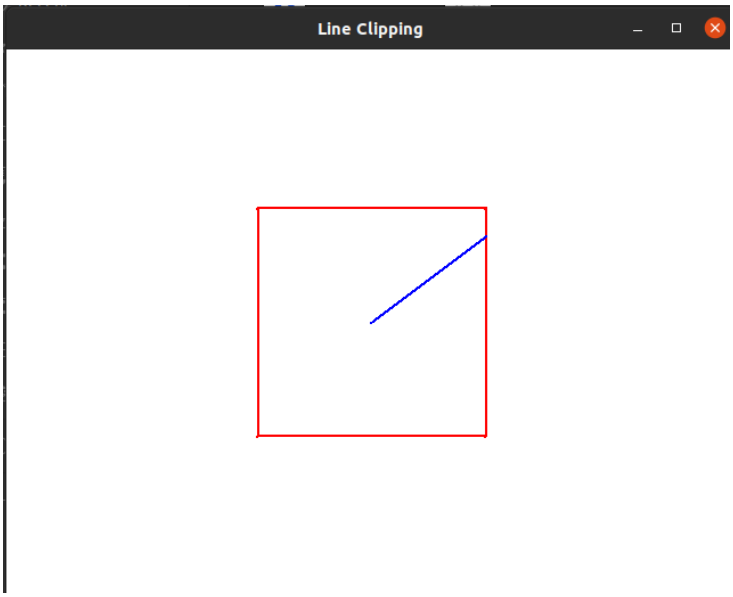


B) One end point lies inside and the other end point lies outside the view port. [(0, 0) and (320, 240)]

Input:



Output:



Program:

```
#include<stdio.h>     //initial inclusions
#include<GL/gl.h>
#include<GL/glu.h>
#include<GL/glut.h>
#include<math.h>
float xd1,yd1,xd2,yd2; //storing values for end points of line
int ymax=100; //initializing window coordinates
int ymin=-100;
int xmax=100;
int xmin=-100;
static int p=0;
void disp(); //declaring display function
float round_value(float v) //function to round value to next greater float
{
        return (v+0.5);
}
void plotpoint(float a,float b)
{
        glBegin(GL_POINTS);
        glVertex2f(a,b);
        glEnd();
}
void dda(float X1,float Y1,float X2,float Y2) //dda algorithm
{
        /*
         * Input : Initial and final co-ordinates of line points.
         * Utility : plot line using Digital Differential Analyzer
         * Output : Line on initialized window.
         */
        float dx,dy,x,y,xinc,yinc;//initializations
```

```
        int k,steps;
        dx=X2-X1;                       //difference of x coordinates
        dy=Y2-Y1;                       //difference of y coordinates
        steps=abs(dx)>abs(dy)?abs(dx):abs(dy); //calculation of number of steps
        xinc=dx/(float)steps; //value for incrementing x
        yinc=dy/(float)steps; //value for incrementing y
        x=X1,y=Y1;
        plotpoint(x,y);                 //function to plot point on window
        for(k=0;k<steps;k++) //loop to plot points
        {
                x+=xinc;                //incrementing x by xinc
                y+=yinc;                //incrementing y by yinc
                plotpoint(round_value(x),round_value(y)); //plotting point
        }
        glFlush();
}
int code(int x,int y)
{
        /*
         * Input : x and y coordinates of the point.
         * Utility : Determine outcode  for given point.
         * Output : Out code.
         */
        int c=0;
        if(y>ymax) c=8;         //if greater than ymax set code to 8
        if(y<ymin) c=4;         //if less than ymin set code to 4
        if(x>xmax) c=c|2;       //if greater than xmax set code to 2
        if(x<xmin) c=c|1;       //if less than ymin set code to 1
        return c;
}
void cohen(float x1,float y1,float x2,float y2)     //implementing cohen-sutherland
algorithm
{
        int c1=code(x1,y1);             //checking for outcode of point 1
        int c2=code(x2,y2);                     //checking for outcode of point 2
        float m=(y2-y1)/(x2-x1);        //checking slope of line
        while((c1|c2)>0)                        //iterating loop till c1|c2>0
        {
                if((c1 & c2)>0)                 //if both lie completely outside the
window
                {
                        disp();
                        return;
                }

                int c;
                float xi=x1;
                float yi=y1;
                c=c1;
```

```
                float x,y;
                if(c==0)                                    //checking if outcode is
equal to 0
                {
                        c=c2;                               //assigning outcode of c2
                        xi=x2;                              //assigning x coordinate of
c2
                        yi=y2;                              //assigning y coordinate of
c2
                }
                if((c & 8)>0)                           //checking if  c&8 >0 ( greater than
ymax)
                {
                        y=ymax;                                     //assigning new
values to x and y
                        x=xi+1.0/(m*(ymax-yi));
                }
                 if((c & 4)>0)                          //checking if c> 4 >0 (less than
ymin)
                {
                        y=ymin;                                     //assigning new
values to x and y
                        x=xi+1.0/(m*(ymin-yi));
                }
                 if((c & 2)>0)                          //checking if  c&2 >0 ( greater than
xmax)
                {
                        x=xmax;
                        y=yi+m*(xmax-xi);
                }
                 if((c & 1)>0)                          //checking if  c&1 >0 (less than
xmin)
                {
                        x=xmin;
                        y=yi+m*(xmin-xi);
                }
                if(c==c1)                                   //checking code and
assigning new values
                {
                        xd1=x;
                        yd1=y;
                        c1=code(xd1,yd1);
                }
                if(c==c2)                                   //checking code and
assigning new values
                {
                        xd2=x;
                        yd2=y;
                        c2=code(xd2,yd2);
```

```
                }
        }
        p++;
        disp();                                         //calling display function
again to display new line
}
void mykey(unsigned char ch,int x,int y)
{
        if(ch=='c')
        {
                cohen(xd1,yd1,xd2,yd2);                 //if character c is pressed calling
algorithm
                glFlush();

        }
}
void disp()
{
        glClear(GL_COLOR_BUFFER_BIT);//clearing buffer
        glColor3f(1.0,0.0,0.0);                         //assigning color
        dda(xmin,ymin,xmax,ymin);               //creating window using dda algorithm to
draw lines
        dda(xmax,ymin,xmax,ymax);
        dda(xmax,ymax,xmin,ymax);
        dda(xmin,ymax,xmin,ymin);

        glColor3f(0.0,0.0,1.0);                         //assigning color for line
        dda(xd1,yd1,xd2,yd2);                           //drawing line
        glFlush();

}
void init()
{
        glClearColor(1.0,1.0,1.0,0);            //clearing background color to new color
        glClear(GL_COLOR_BUFFER_BIT);                   //clearing buffer
        glPointSize(2);                                 //assigning point size
        gluOrtho2D(-320,320,-240,240);
        glFlush();

}
int main(int argc,char **argv)
{
        printf("Window coordinates are (-100,100,-100,100)\n");
        printf("\nEnter coordinates of the line(limits : -320,320,-240,240) \nAfter entering
enter c to clip\n");
        printf("\nCoordinates of first point");
        printf("\nX1: ");
        scanf("%f",&xd1);                               //accepting value of x1
        printf("\nY1: ");                               //accepting value of y1
```

```
        scanf("%f",&yd1);
        printf("\nCoordinates of second point");
        printf("\nX2: ");
        scanf("%f",&xd2);                          //accepting value of x2
        printf("\nY2: ");                          //accepting value of y2
        scanf("%f",&yd2);

        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowPosition(100,100);
        glutInitWindowSize(640,480);
        glutCreateWindow("Line Clipping");
        init();
        glutDisplayFunc(disp);
        glutKeyboardFunc(mykey);
        glutMainLoop();
        return 0;
}
```

Conclusion: Successfully implemented Cohen Sutherland Polygon Clipping Algorithms using OpenGL.

Assignment No 6

Aim: 2 D Transformation using OpenGL.

Prob Statements : Implement the following polygon filling methods : i) Flood fill / Seed fill ii) Boundary fill ; using mouse click, keyboard interface and menu driven programming

Implement following 2D transformations on the object with respect to axis : – i) Scaling

ii) Rotation about arbitrary point iii) Reflection

Theory:

Transformation means changing some graphics into something else by applying rules. We can have various types of transformations such as translation, scaling up or down, rotation, shearing, etc. When a transformation takes place on a 2D plane, it is called 2D transformation.

Transformations play an important role in computer graphics to reposition the graphics on the screen and change their size or orientation.

Homogenous Coordinates

To perform a sequence of transformation such as translation followed by rotation and scaling, we need to follow a sequential process −

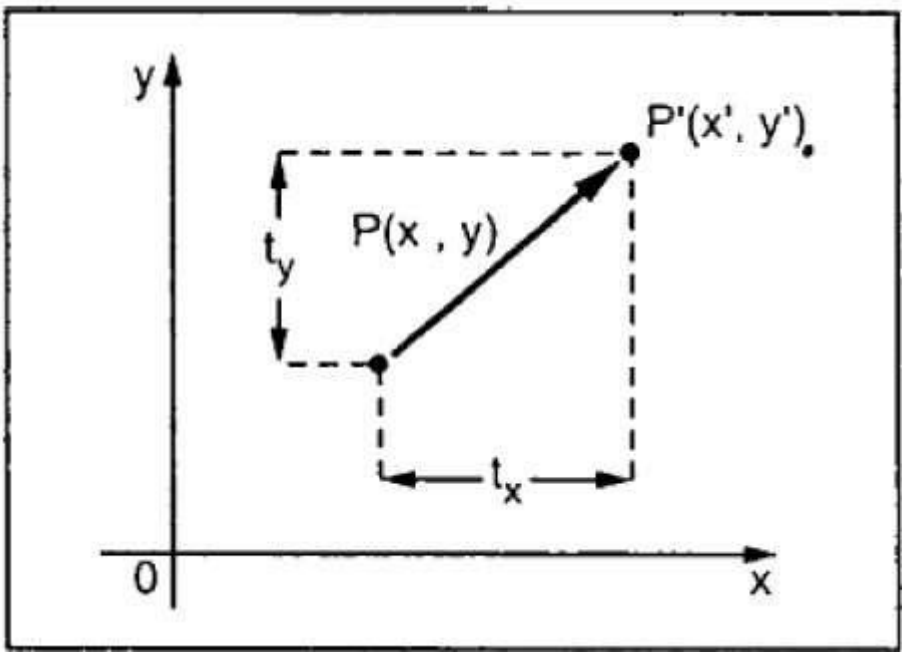
- Translate the coordinates,
- Rotate the translated coordinates, and then
- Scale the rotated coordinates to complete the composite transformation.

To shorten this process, we have to use 3×3 transformation matrix instead of 2×2 transformation matrix. To convert a 2×2 matrix to 3×3 matrix, we have to add an extra dummy coordinate W.

In this way, we can represent the point by 3 numbers instead of 2 numbers, which is called **Homogenous Coordinate** system. In this system, we can represent all the transformation equations in matrix multiplication. Any Cartesian point PX, YX, Y can be converted to homogenous coordinates by P' (X_h, Y_h, h).

Translation

A translation moves an object to a different position on the screen. You can translate a point in 2D by adding translation coordinate (t_x, t_y) to the original coordinate X, YX, Y to get the new coordinate X′, Y′X′, Y′.



From the above figure, you can write that −

$$X' = X + t_x$$

$$Y' = Y + t_y$$

The pair (t_x, t_y) is called the translation vector or shift vector. The above equations can also be represented using the column vectors.

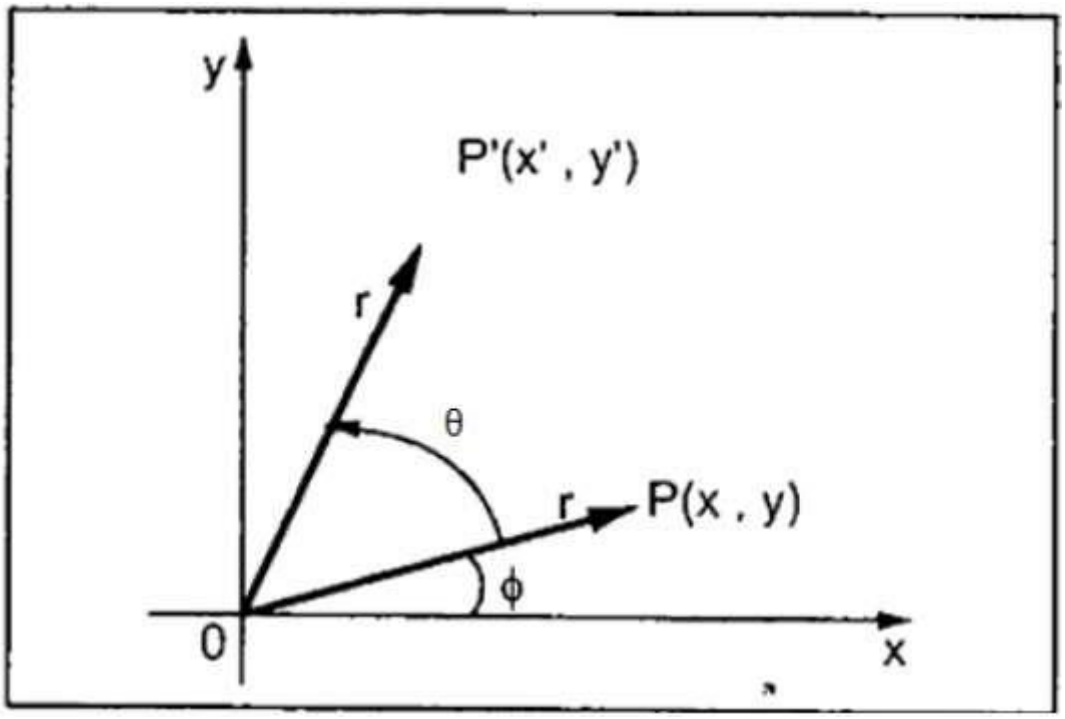
P=[X][Y]P=[X][Y] p' = [X′][Y′][X′][Y′]T = [tx][ty][tx][ty]

We can write it as −

$$P' = P + T$$

Rotation

In rotation, we rotate the object at particular angle θ from its origin. From the following figure, we can see that the point PX, YX, Y is located at angle φ from the horizontal X coordinate with distance r from the origin.
Let us suppose you want to rotate it at the angle θ. After rotating it to a new location, you will get a new point P' X′, Y′X′, Y′.



Using standard trigonometric the original coordinate of point PX,YX,Y can be represented as −

$$X=r\cos\phi \ldots\ldots(1) X=r\cos\phi \ldots\ldots(1)$$
$$Y=r\sin\phi \ldots\ldots(2) Y=r\sin\phi \ldots\ldots(2)$$

Same way we can represent the point P' X′,Y′X′,Y′ as −

$$x'=r\cos(\phi+\theta)=r\cos\phi\cos\theta-r\sin\phi\sin\theta \ldots\ldots(3) x'=r\cos(\phi+\theta)=r\cos\phi\cos\theta-r\sin\phi\sin\theta \ldots\ldots(3)$$
$$y'=r\sin(\phi+\theta)=r\cos\phi\sin\theta+r\sin\phi\cos\theta \ldots\ldots(4) y'=r\sin(\phi+\theta)=r\cos\phi\sin\theta+r\sin\phi\cos\theta \ldots\ldots(4)$$

Substituting equation 11 & 22 in 33 & 44 respectively, we will get

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

Representing the above equation in matrix form,

$$[X'Y']=[XY][\cos\theta-\sin\theta\sin\theta\cos\theta]OR[X'Y']=[XY][\cos\theta\sin\theta-\sin\theta\cos\theta]OR$$

P' = P . R

Where R is the rotation matrix

$$R=[\cos\theta-\sin\theta\sin\theta\cos\theta]R=[\cos\theta\sin\theta-\sin\theta\cos\theta]$$

The rotation angle can be positive and negative.

For positive rotation angle, we can use the above rotation matrix. However, for negative angle rotation, the matrix will change as shown below −

$$R=[\cos(-\theta)-\sin(-\theta)\sin(-\theta)\cos(-\theta)]R=[\cos(-\theta)\sin(-\theta)-\sin(-\theta)\cos(-\theta)]$$
$$=[\cos\theta\sin\theta-\sin\theta\cos\theta](\because\cos(-\theta)=\cos\theta and \sin(-\theta)=-\sin\theta)=[\cos\theta-\sin\theta\sin\theta\cos\theta](\because\cos(-\theta)=\cos\theta and \sin(-\theta)=-\sin\theta)$$

Scaling

To change the size of an object, scaling transformation is used. In the scaling process, you either expand or compress the dimensions of the object. Scaling can be achieved by

multiplying the original coordinates of the object with the scaling factor to get the desired result.

Let us assume that the original coordinates are X,YX,Y, the scaling factors are (S_X, S_Y), and the produced coordinates are X′,Y′X′,Y′. This can be mathematically represented as shown below −

$$\mathbf{X' = X \cdot S_X \text{ and } Y' = Y \cdot S_Y}$$

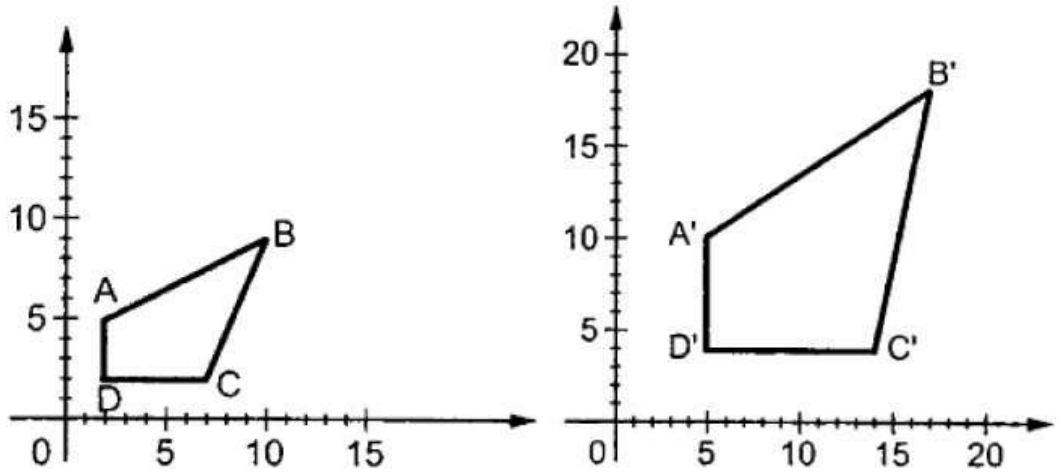
The scaling factor S_X, S_Y scales the object in X and Y direction respectively. The above equations can also be represented in matrix form as below −

(X′Y′)=(XY)[Sx00Sy](X′Y′)=(XY)[Sx00Sy]

OR

$$\mathbf{P' = P \cdot S}$$

Where S is the scaling matrix. The scaling process is shown in the following figure.

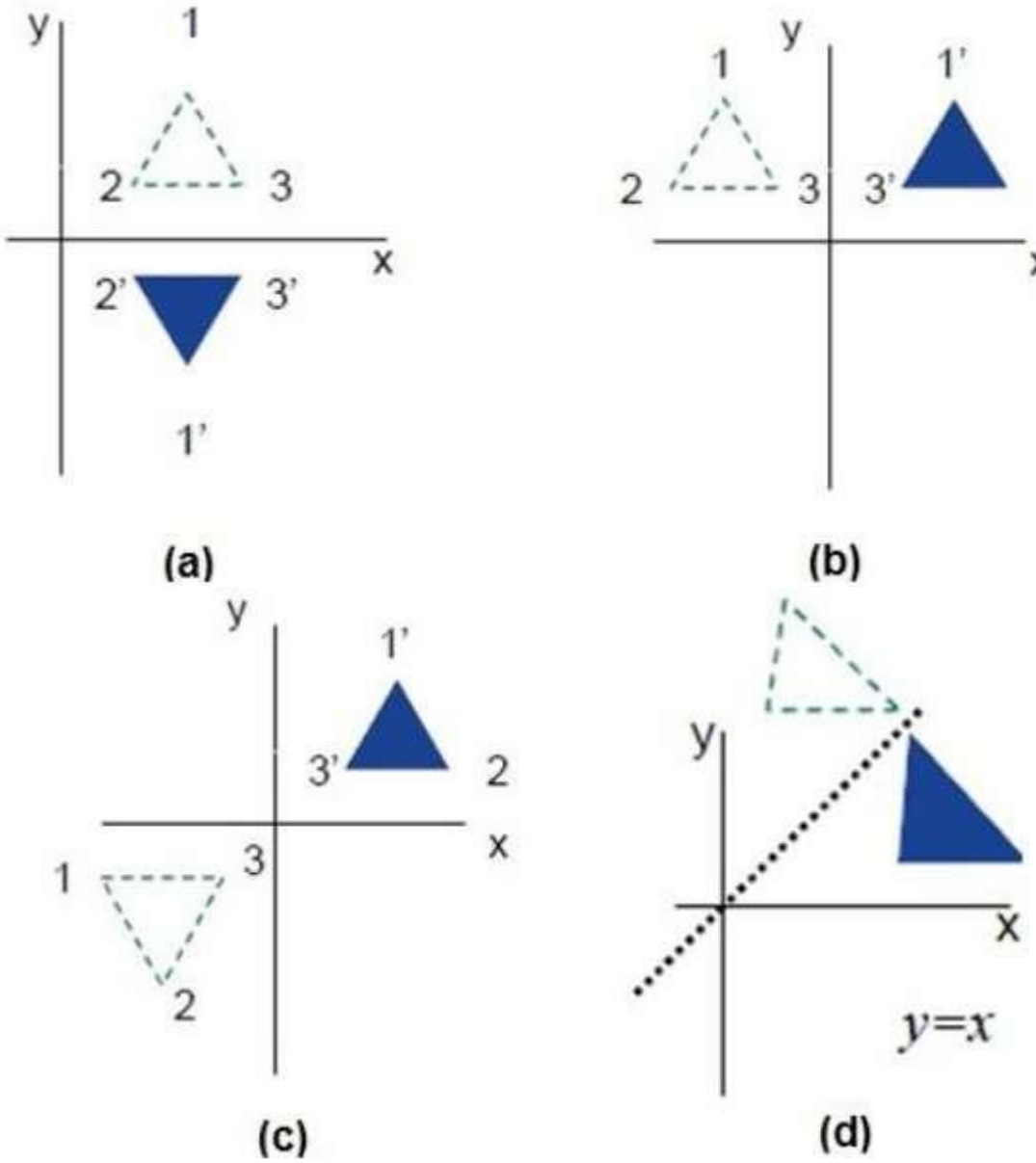


If we provide values less than 1 to the scaling factor S, then we can reduce the size of the object. If we provide values greater than 1, then we can increase the size of the object.

Reflection

Reflection is the mirror image of original object. In other words, we can say that it is a rotation operation with 180°. In reflection transformation, the size of the object does not change.

The following figures show reflections with respect to X and Y axes, and about the origin respectively.

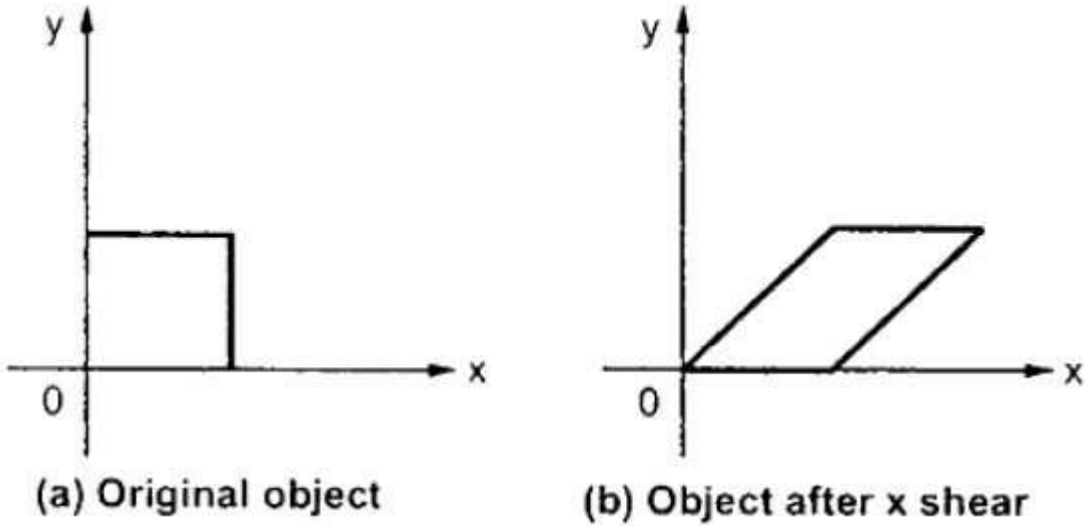


Shear

A transformation that slants the shape of an object is called the shear transformation. There are two shear transformations **X-Shear** and **Y-Shear**. One shifts X coordinates values and other shifts Y coordinate values. However; in both the cases only one coordinate changes its coordinates and other preserves its values. Shearing is also termed as **Skewing**.

X-Shear

The X-Shear preserves the Y coordinate and changes are made to X coordinates, which causes the vertical lines to tilt right or left as shown in below figure.



(a) Original object (b) Object after x shear

The transformation matrix for X-Shear can be represented as −

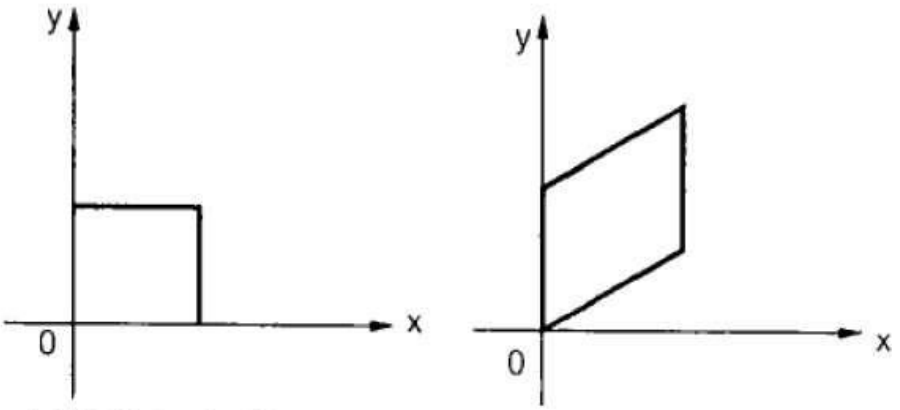
$$Xsh=\begin{bmatrix}1 & 0 & 0\\ shx & 1 & 0\\ 0 & 0 & 1\end{bmatrix}Xsh=\begin{bmatrix}1 & shx & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$Y' = Y + Sh_y \cdot X$$

$$X' = X$$

Y-Shear

The Y-Shear preserves the X coordinates and changes the Y coordinates which causes the horizontal lines to transform into lines which slopes up or down as shown in the following figure.



(a) Original object (b) Object after y shear

The Y-Shear can be represented in matrix from as −

$$Ysh\begin{bmatrix}1 & shy & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}Ysh\begin{bmatrix}1 & 0 & 0\\ shy & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$X' = X + Sh_x \cdot Y$$

$$Y' = Y$$

Composite Transformation

If a transformation of the plane T1 is followed by a second plane transformation T2, then the result itself may be represented by a single transformation T which is the composition of T1 and T2 taken in that order. This is written as T = T1·T2.

Composite transformation can be achieved by concatenation of transformation matrices to obtain a combined transformation matrix.

A combined matrix −

[T][X] = [X] [T1] [T2] [T3] [T4] .... [Tn]

Where [Ti] is any combination of

- Translation
- Scaling
- Shearing
- Rotation
- Reflection

The change in the order of transformation would lead to different results, as in general matrix multiplication is not cumulative, that is [A] . [B] ≠ [B] . [A] and the order of multiplication. The basic purpose of composing transformations is to gain efficiency by applying a single composed transformation to a point, rather than applying a series of transformation, one after another.

For example, to rotate an object about an arbitrary point (X_p, Y_p), we have to carry out three steps −

- Translate point (X_p, Y_p) to the origin.
- Rotate it about the origin.
- Finally, translate the center of rotation back where it belonged.

Algorithm:
A) Scaling

1. START
2. Create a 3x3 scaling matrix S as:
   | S_x 0 0 |
   | 0 S_y 0 |
   | 0 0 1 |
   where S_x and S_y are the scaling factors in x and y directions respectively.
3. For all points located on the polygon, generate a n x 3 matrix where each row in the matrix represents a point on the polygon in the following manner:
4. [x, y, 1] where (x, y) are the co-ordinates of the point.
5. Multiply the generated point matrix with the scaling matrix to obtain the transformation matrix.
6. Plot the transformation matrix.

7. END

B) Translation

1. START
2. Get the X and Y translation factor from the user.
3. Create a 3x3 scaling matrix S as:
   | 1 0 0 |
   | 0 1 0 |
   | T_x T_y 1 |
   where T_x and T_y are the scaling factors in x and y directions respectively.
4. For all points located on the polygon, generate a n x 3 matrix where each row in the matrix represents a point on the polygon in the following manner:
5. [x, y, 1] where (x, y) are the co-ordinates of the point.
6. Multiply the generated point matrix with the translation matrix to obtain the transformation matrix.
7. Plot the transformation matrix.
8. END

B) Rotation

1. START
2. Get the angle of rotation as theta
3. Create a 2x2 scaling matrix S as:

   | cos(θ) sin(θ) |
   | -sin(θ) cos(θ) |

4. For all points located on the polygon, generate a n x 2 matrix where each row in the matrix represents a point on the polygon in the following manner:
   [x, y] where (x, y) are the co-ordinates of the point.
5. Multiply the generated point matrix with the translation matrix to obtain the transformation matrix.
6. Plot the transformation matrix.
7. END

Program:

```
#include<GL/glut.h>
#include<stdio.h>

#include<math.h>

static int flag;

int length, xi, yi, choice;
double angle, ET[3][3], ETResult[3][3];
double Rh[4][4], RhResult[4][4];
```

```
//------------------DRAW-------------//

void drawET(double ET[3][3])
{
	int i;

	glBegin(GL_LINE_LOOP);
	for(i=0;i<3;i++)
	{
	 glVertex2i(ET[i][0],ET[i][1]);
	}
	glEnd();
}

void drawR(double Rh[4][4])
{
	int i;

	glBegin(GL_LINE_LOOP);
	for(i=0;i<4;i++)
	{
	 glVertex2i(Rh[i][0],Rh[i][1]);
	}
	glEnd();
}
//----------------------------Display---------------------//

void Display()
{
   glClearColor(0,0,0,0);
   glClear(GL_COLOR_BUFFER_BIT);

   glLoadIdentity();
   gluOrtho2D(-320,320,-240,240);//note

   glColor3f(1,1,1);
   glBegin(GL_LINES);
      glVertex2d(-320,0);
      glVertex2d(320,0);
      glVertex2d(0,-240);
      glVertex2d(0,240);
   glEnd();

   glColor3f(1,0,0);
   if(flag == 0)
    drawET(ET);
   else if(flag == 1)
    drawR(Rh);
```

```
    glFlush();
}


//---------MULTIPLY--------------//

void mult3X3(double ET[3][3],double temp[3][3])
{
 double sum;
 int i,j,k;
    for(i=0;i<3;i++)
    {
       for(j=0;j<3;j++)
        {
         sum=0;
            for(k=0;k<3;k++)
            {
               sum=sum+ET[i][k]*temp[k][j];
            }
            ETResult[i][j]=sum;
        }
    }
}

void mult4X4(double Rh[4][4],double temp[4][4])
{
 double sum;
    int i,j,k;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
         sum=0;
            for(k=0;k<4;k++)
            {
                sum=sum+Rh[i][k]*temp[k][j];
  }
            RhResult[i][j]=sum;
        }
 }
}

//----------Translation-----------//

void translationET()
{
 double tx,ty,temp[3][3];
```

```
    printf("\nTranslating Equilateral triangle");
    printf("\nEnter Tx: ");
    scanf("%lf",&tx);
    printf("\nEnter Ty: ");
    scanf("%lf",&ty);

 temp[3][3]={0};
    temp[0][0]=1;
    temp[1][1]=1;
    temp[2][2]=1;
    temp[2][0]=tx;
    temp[2][1]=ty;

    mult3X3(ET,temp);
    glColor3f(0.0,1.0,0.0);
    drawET(ETResult);
}

void translationRh()
{
 double tx,ty,temp[4][4];

    printf("\nTranslating Rhombus");
    printf("\nEnter Tx: ");
    scanf("%lf",&tx);
    printf("\nEnter Ty: ");
 scanf("%lf",&ty);

    temp[4][4]={0};
    temp[0][0]=1;
    temp[1][1]=1;
    temp[2][2]=1;
 temp[3][3]=1;
 temp[3][0]=tx;
    temp[3][1]=ty;

    mult4X4(Rh,temp);
    glColor3f(0.0,1.0,0.0);
    drawR(RhResult);
}

//------------Rotation------------//

void rotationET()
{
 double rx,ry,angle, temp[3][3];

    printf("\n**ROTATION**\n");
    printf("\nArbitrary Point (x,y) : ");
```

```
    scanf("%lf %lf",&rx,&ry);
    printf("\nAngle (in degrees) : ");
    scanf("%lf",&angle);

    angle=angle*(M_PI/180);

    temp[3][3]={0};
    temp[0][0]=cos(angle);
    temp[0][1]=sin(angle);
    temp[1][0]=-sin(angle);
    temp[1][1]=cos(angle);
    temp[2][0]=(-(rx*cos(angle))+(ry*sin(angle))+rx);
    temp[2][1]=(-(rx*sin(angle))-(ry*cos(angle))+ry);
    temp[2][2]=1;

    mult3X3(ET,temp);
    glColor3f(0.0,1.0,0.0);
    drawET(ETResult);
}

void rotationRh()
{
 double rx,ry,angle, temp[4][4];

    printf("\nRotating Rhombus");
    printf("\nArbitrary Point (x,y): ");
    scanf("%lf %lf",&rx,&ry);
    printf("\nAngle (in degree): ");
    scanf("%lf",&angle);

    angle=angle*(M_PI/180);

    temp[4][4]={0};
    temp[0][0]=cos(angle);
    temp[0][1]=sin(angle);
    temp[1][0]=-sin(angle);
    temp[1][1]=cos(angle);
    temp[2][2]=1;
    temp[3][0]=(-(rx*cos(angle))+(ry*sin(angle))+rx);
    temp[3][1]=(-(rx*sin(angle))-(ry*cos(angle))+ry);
    temp[3][3]=1;

    mult4X4(Rh,temp);
    glColor3f(0.0,1.0,0.0);
    drawR(RhResult);
}

//----------Scaling-------------//
```

```
void scaleET()
{
 double sx,sy, temp[3][3];

    printf("\nScaling Equilateral triangle");
    printf("\nSx: ");
    scanf("%lf",&sx);
    printf("\nSy: ");
    scanf("%lf",&sy);

    temp[3][3]={0};
 temp[0][0]=sx;
 temp[1][1]=sy;
 temp[2][2]=1;

    mult3X3(ET,temp);
    glColor3f(1.0,1.0,0.0);
    drawET(ETResult);
}

void scaleRh()
{
    double sx,sy,temp[4][4];

    printf("\nScaling Rhombus");
    printf("\nSx: ");
    scanf("%lf",&sx);
    printf("\nSy: ");
    scanf("%lf",&sy);

    temp[4][4]={0};
 temp[0][0]=sx;
 temp[1][1]=sy;
 temp[2][2]=1;
 temp[3][3]=1;

    mult4X4(Rh,temp);
    glColor3f(1.0,1.0,0.0);
    drawR(RhResult);
}

//-----------Shearing-------------//
void shearET()
{
 double xs,ys,temp[3][3];

    printf("\nShear Equilateral triangle");
    printf("\nPress 1: X - Shear");
    printf("\nPress 2: Y - Shear");
```

```
    printf("\nEnter your Choice: ");
    scanf("%d",&choice);

    temp[3][3]={0};

    switch(choice)
    {
     case 1: printf("\nX-shear value: ");
       scanf("%lf",&xs);
              temp[0][0]=1;
              temp[1][0]=xs;
              temp[1][1]=1;
              temp[2][2]=1;
              break;
  case 2: printf("\nY-shear value: ");
       scanf("%lf",&ys);
    temp[0][0]=1;
    temp[0][1]=ys;
    temp[1][1]=1;
    temp[2][2]=1;
              break;
    }

    mult3X3(ET,temp);
    glColor3f(1.0,1.0,0.0);
    drawET(ETResult);
}

void shearRh()
{
 double xs,ys,temp[4][4];

    printf("\nPress 1: X - Shear");
    printf("\nPress 2: Y - Shear");
    printf("\nEnter your Choice: ");
    scanf("%d",&choice);

    temp[4][4]={0};

    switch(choice)
    {
     case 1: printf("\nX-shear value: ");
              scanf("%lf",&xs);
              temp[0][0]=1;
              temp[1][0]=xs;
              temp[1][1]=1;
              temp[2][2]=1;
              temp[3][3]=1;
              break;
```

```
  case 2: printf("\nY-shear value: ");
      scanf("%lf",&ys);
            temp[0][0]=1;
            temp[0][1]=ys;
            temp[1][1]=1;
            temp[2][2]=1;
            temp[3][3]=1;
            break;
 }
   mult4X4(Rh,temp);
   glColor3f(0.0,1.0,0.0);
   drawR(RhResult);
}

//----------MENU--------------//
void Menu(int item)
{
 switch(item)
   {
      case 1: if(choice==1)
               translationET();
              else
               translationRh();

              break;

      case 2: if(choice==1)
               rotationET();
              else
               rotationRh();

              break;
      case 3: if(choice==1)
               scaleET();
              else
               scaleRh();

              break;
      case 4: if(choice==1)
              {
               shearET();
              }
              else
              {
               shearRh();
              }
              break;
      case 5:
        exit(0);
```

```
        break;
    }
}

//----------MAIN-------------------//

int main(int argc,char** argv)
{
    printf("\n*MENU");
    printf("\n1. To draw Equilateral Triangle");
    printf("\n2. To draw Rhombus");
    printf("\n3. To Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);

    switch(choice)
    {
     int i, j;

       case 1:
         flag = 0;
         printf("\nEnter X co-ordinate of a Base point: ");
         scanf("%d",&xi);
         printf("\nEnter Y co-ordinate of the Base point: ");
          scanf("%d",&yi);
          printf("\nEnter length of sides: ");
          scanf("%d",&length);

          for(i=0;i<3;i++)
               {
                for(j=0;j<3;j++)
                  {
                     ET[i][j]=1;
                  }
               }
          ET[0][0]=xi;
          ET[0][1]=yi;
          ET[1][0]=xi+length;
          ET[1][1]=yi;
          ET[2][0]=length/2+xi;
          ET[2][1]=(sqrt(3)/2*length)+yi;
          break;

        case 2:
          flag = 1;
          printf("\nEnter X co-ordinates of a Base point: ");
          scanf("%d",&xi);
          printf("\nEnter Y co-ordinates of the Base point: ");
          scanf("%d",&yi);
```

```
        printf("\nEnter length of sides: ");
        scanf("%d",&length);
        printf("\nEnter angle of Rhombus (in degrees): ");
        scanf("%lf",&angle);
        angle = angle * M_PI / 180;

        for(i=0;i<4;i++)
             {
              for(j=0;j<4;j++)
                {
                   Rh[i][j]=1;
                }
             }
        Rh[0][0]=xi;
        Rh[0][1]=yi;
        Rh[1][0]=xi+length;
        Rh[1][1]=yi;
        Rh[2][0]=length+xi+length*cos(angle);
        Rh[2][1]=yi+length*sin(angle);
        Rh[3][0]=xi+length*cos(angle);
        Rh[3][1]=yi+length*sin(angle);
        break;

      case 3:
        exit(0);
        break;

      default:printf("\nInvalid Input!");
        break;
}

   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE);
   glutInitWindowSize(640,480);
   glutInitWindowPosition(0,0);
   glutCreateWindow("2D - TRANSFORMATIONS");

   glutDisplayFunc(Display);

   glutCreateMenu(Menu);
    glutAddMenuEntry("1.Translation",1);
    glutAddMenuEntry("2.Rotation",2);
    glutAddMenuEntry("3.Scaling",3);
    glutAddMenuEntry("4.Shear",4);
    glutAddMenuEntry("5.EXIT",5);
   glutAttachMenu(GLUT_RIGHT_BUTTON);

   glutMainLoop();
   return 0;
```

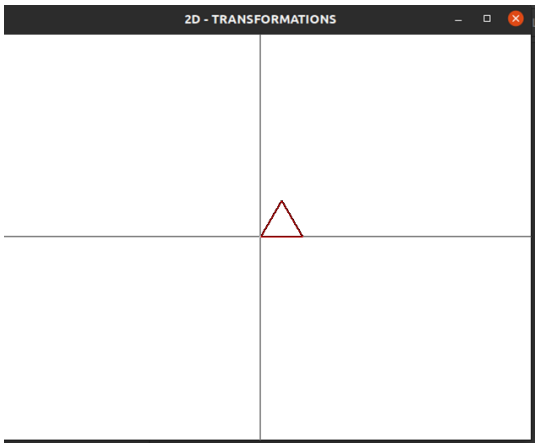
}

Input:

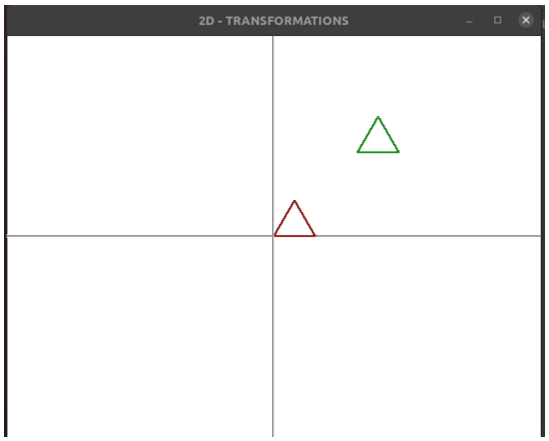
1. The program plots Equilateral triangle or a rhombus.
2. For the equilateral Triangle, the point of origin is given as (0, 0) and the size of one side is given as 50px.
3. For the rhombus, the point of origin is given as (0,0), angle is given as 60 degrees and the side length is given as 60px.
4. The transformation is selected using the menu.

Output:

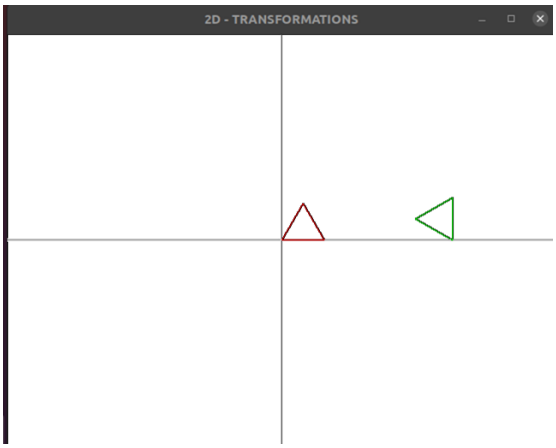
1. Triangle



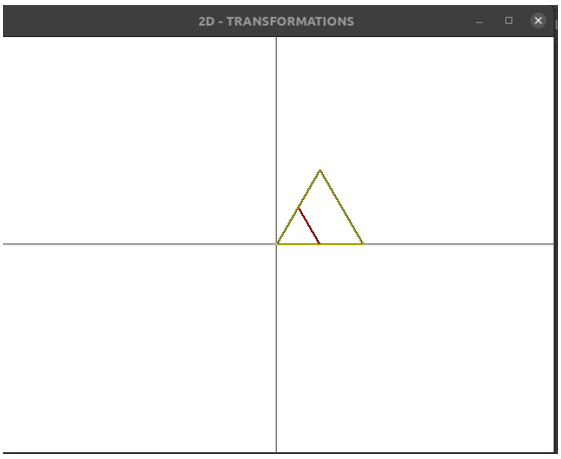
a) Translation:



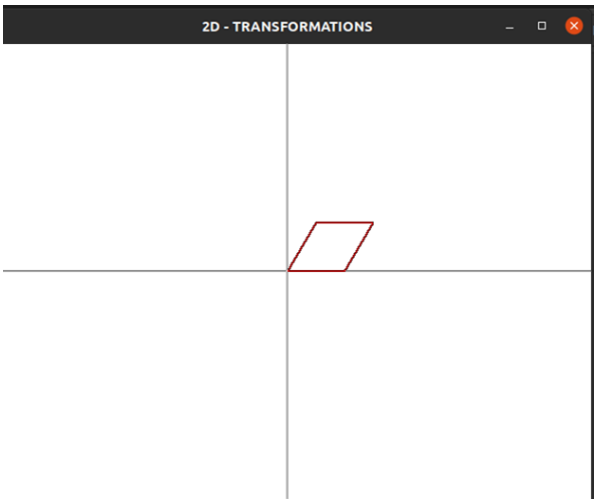
b) Rotation:



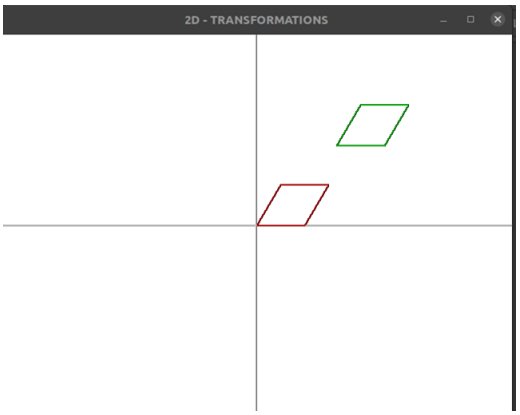
c) Scaling



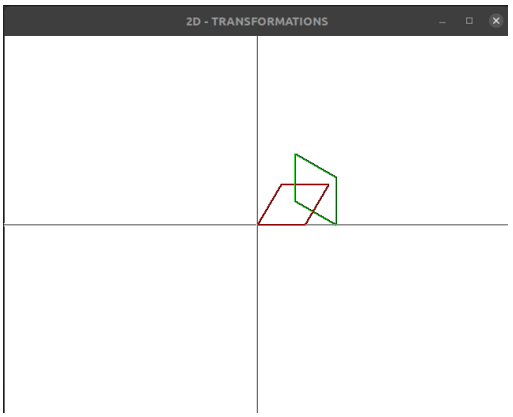
2. Rhombus



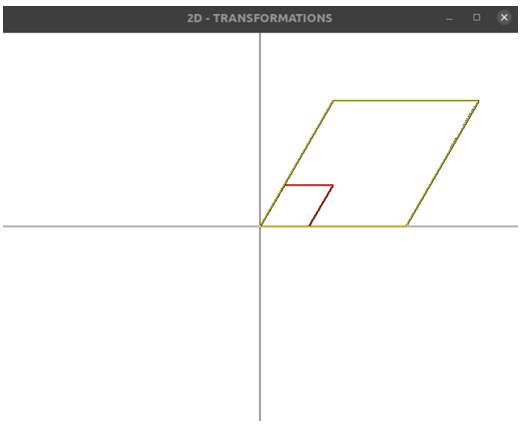
a) Translation



b) Rotation



c) Scaling



Conclusion:

1. Successfully implemented 2D transformation.
2. Matrix multiplication was used to implement the transformations.

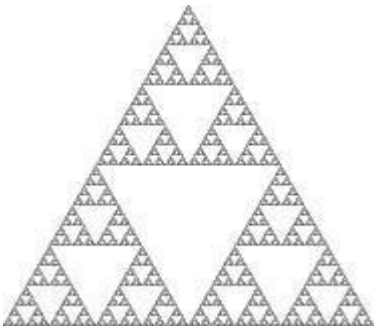
Assignment No 7

Aim: Generate fractal patterns using OpenGL.

Prob Statements : Generate fractal patterns using i) Bezier ii) Koch Curve

Theory:

A fractal is a never-ending pattern. Fractals are infinitely complex patterns that are selfsimilar across different scales. They are created by repeating a simple process over and over in an ongoing feedback loop. Driven by recursion, fractals are images of dynamic systems – the pictures of Chaos. Geometrically, they exist in between our familiar dimensions. Fractal patterns are extremely familiar, since nature is full of fractals. For instance: trees, rivers, coastlines, mountains, clouds, seashells, hurricanes, etc. Abstract fractals – such as the Mandelbrot Set – can be generated by a computer calculating a simple equation over and over.



Bezier Curves

Bezier curve is discovered by the French engineer Pierre Bézier. These curves can be generated under the control of other points. Approximate tangents by using control points are used to generate curve. The Bezier curve can be represented mathematically as –

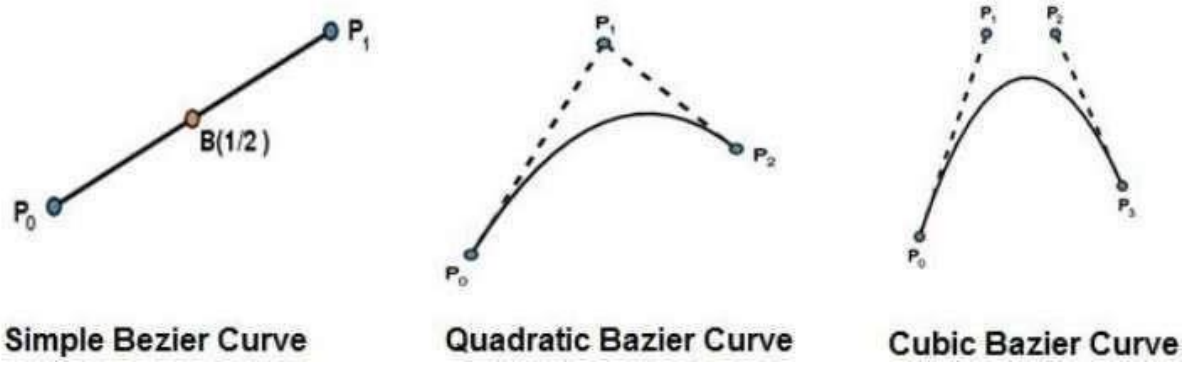
$$\sum k=0nPiBni(t)\sum k=0nPiBin(t)$$

Where pipi is the set of points and Bni(t)Bin(t) represents the Bernstein polynomials which are given by –

$$B_{ni}(t)=(ni)(1-t)n-¡tiBin(t)=(ni)(1-t)n-iti$$

Where n is the polynomial degree, i is the index, and t is the variable.

The simplest Bézier curve is the straight line from the point P0P0 to P1P1. A quadratic Bezier curve is determined by three control points. A cubic Bezier curve is determined by four control points.



Simple Bezier Curve Quadratic Bazier Curve Cubic Bazier Curve

Properties of Bezier Curves

Bezier curves have the following properties –

- They generally follow the shape of the control polygon, which consists of the segments joining the control points.
- They always pass through the first and last control points.
- They are contained in the convex hull of their defining control points.
- The degree of the polynomial defining the curve segment is one less that the number of defining polygon point. Therefore, for 4 control points, the degree of the polynomial is 3, i.e. cubic polynomial.
- A Bezier curve generally follows the shape of the defining polygon.
- The direction of the tangent vector at the end points is same as that of the vector determined by first and last segments.
- The convex hull property for a Bezier curve ensures that the polynomial smoothly follows the control points.
- No straight line intersects a Bezier curve more times than it intersects its control polygon.
- They are invariant under an affine transformation.
- Bezier curves exhibit global control means moving a control point alters the shape of the whole curve.
- A given Bezier curve can be subdivided at a point t=t0 into two Bezier segments which join together at the point corresponding to the parameter value t=t0.

Koch Curve

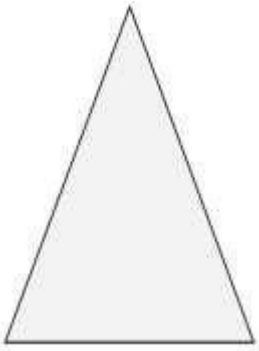
The Koch snowflake (also known as the Koch curve, Koch star, or Koch island) is a mathematical curve and one of the earliest fractal curves to have been described. It is based on the Koch curve, which appeared in a 1904 paper titled “On a continuous curve without tangents, constructible from elementary geometry” by the Swedish mathematician Helge von Koch.

The progression for the area of the snowflake converges to 8/5 times the area of the original triangle, while the progression for the snowflake’s perimeter diverges to infinity. Consequently, the snowflake has a finite area bounded by an infinitely long line.

Step1:

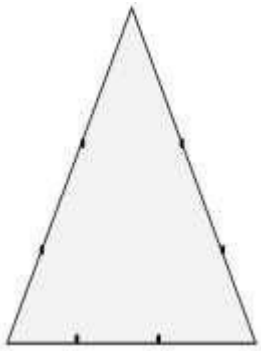
Draw an equilateral triangle. You can draw it with a compass or protractor, or just eyeball it if you don’t want to spend too much time drawing the snowflake.

It’s best if the length of the sides are divisible by 3, because of the nature of this fractal. This will become clear in the next few steps.



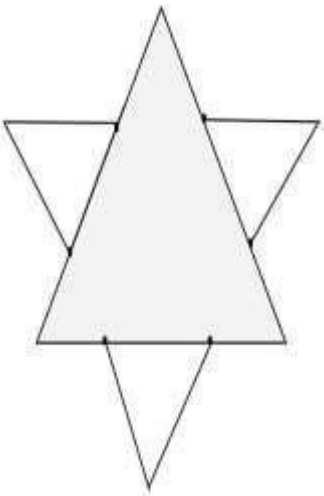
Step2:

Divide each side in three equal parts. This is why it is handy to have the sides divisible by three.



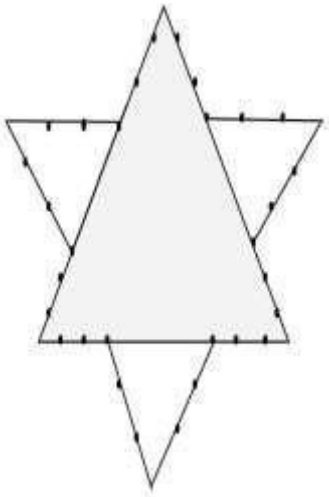
Step3:

Draw an equilateral triangle on each middle part. Measure the length of the middle third to know the length of the sides of these new triangles.



Step4:

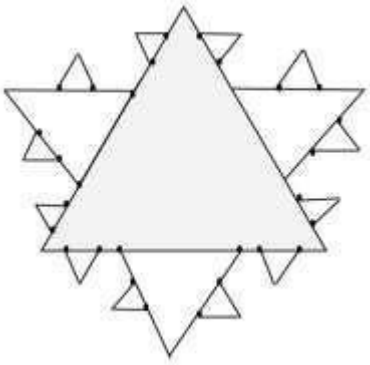
Divide each outer side into thirds. You can see the 2nd generation of triangles covers a bit of the first. These three line segments shouldn't be parted in three.



Step5:

Draw an equilateral triangle on each middle part.

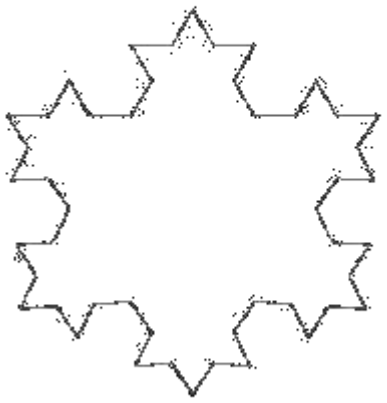
Note how you draw each next generation of parts that are one 3rd of the mast one.

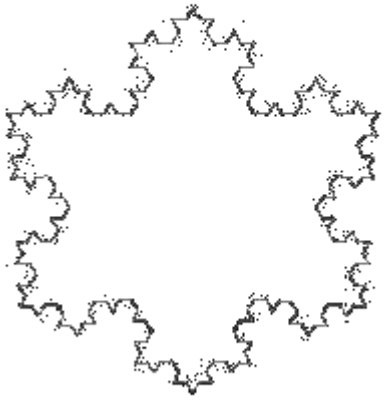


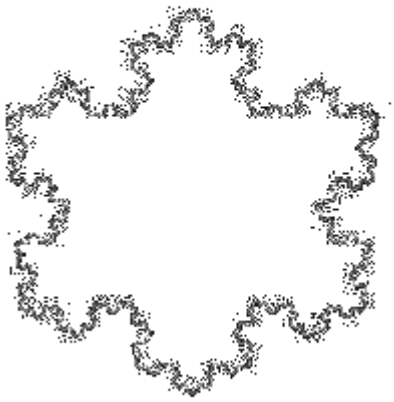
```
#include <iostream>
#include <GL/glut.h>
#include <GL/freeglut.h>
#include <math.h> using
namespace std;

#define RADIAN (3.14/180)
#define XMAX 1400
#define YMAX
  900 void
Initialize(); void
draw();
void draw_koch(float,float,float,float,int);

void Initialize()
{
        glClear(GL_COLOR_BUFFER_BIT);
glClearColor(0.0,0.0,0.0,0.0);
glColor3f(1.0,1.0,1.0);
          gluOrtho2D(0.0,XMAX,0.0,YMAX);
}
void draw(int n)
{

          glBegin(GL_LINES);
                  draw_koch(600,100,800,
                                  400,n);
                  draw_koch(800,400,400,
                                  400,n);
                  draw_koch(400,400,600,
                                  100,n);
          glEnd();
          glFlush();
}

void draw_koch(float xa,float ya,float xb,float yb,int n)
{       float xc,xd,yc,yd,midx,midy;

        xc = (2*xa+xb)/3;
yc = (2*ya+yb)/3;               xd
= (2*xb+xa)/3;        yd =
(2*yb+ya)/3;

           midx = xc + ((xd-xc)*cos(60*RADIAN)) + ((yd-yc)*sin(60*RADIAN)); midy
           = yc - ((xd-xc)*sin(60*RADIAN)) + ((yd-yc)*cos(60*RADIAN));

          if(n>0)
```

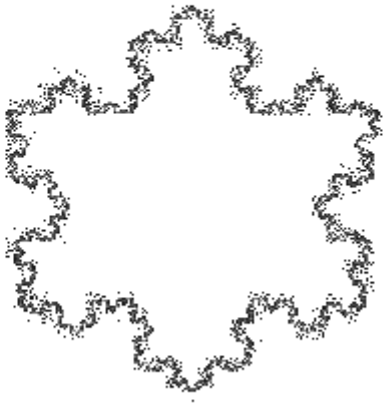
```
        {
            draw_koch(xa,ya,xc,yc,n-1);
draw_koch(xc,yc,midx,midy,n-1);          draw_koch(midx,midy,xd,yd,n-1);
                draw_koch(xd,yd,xb,yb,n-1);
        }

        else
        {
                glVertex2f(xa,ya);
                glVertex2f(xc,yc);

                glVertex2f(xc,yc);
                glVertex2f(midx,midy);

                glVertex2f(midx,midy);
                glVertex2f(xd,yd);

            glVertex2f(xd,yd);
glVertex2f(xb,yb);
        }
}
int main(int argc , char ** argv)
{
        int n;
        cout<<"\n Enter For How Many Iterations You Want to Draw ?::";
        cin>>n;
        glutInit( &argc , argv);
      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(XMAX,YMAX);
glutInitWindowPosition(0,0);
        glutCreateWindow("KOCH
        CURVE");

      Initialize(); draw(n);
            glutMainLoop();
        return 0;
}
```

OUTPUT :









Conclusion:

Successfully implemented Koch Curve.

Assignment No 8

Aim: Animation using OpenGL.

Prob Statements : Implement animation principles for any object

Theory:

Theory:

Animation is a method in which figures are manipulated to appear as moving images. In traditional animation, images are drawn or painted by hand on transparent celluloid sheets to be photographed and exhibited on film. Today, most animations are made with computer-generated imagery (CGI). Computer animation can be very detailed 3D animation, while 2D computer animation (which may have the look of traditional animation) can be used for stylistic reasons, low bandwidth, or faster real-time renderings. Other common animation methods apply a stop motion technique to two and three-dimensional objects like paper cutouts, puppets, or clay figures.

Commonly, the effect of animation is achieved by a rapid succession of sequential images that minimally differ from each other. The illusion—as in motion pictures in general—is thought to rely on the phi phenomenon and beta movement, but the exact causes are still uncertain. Analog mechanical animation media that rely on the rapid display of sequential images include the phénakisticope, zoetrope, flip book, praxinoscope, and film. Television and video are popular electronic animation media that originally were analog and now operate digitally. For display on the computer, techniques like animated GIF and Flash animation were developed.

Animation is more pervasive than many people know. Apart from short films, feature films, television series, animated GIFs, and other media dedicated to the display of moving images, animation is also prevalent in video games, motion graphics, user interfaces, and visual effects.

The physical movement of image parts through simple mechanics—for instance moving images in magic lantern shows—can also be considered animation. The mechanical manipulation of three-dimensional puppets and objects to emulate living beings has a very long history in automata. Electronic automata were popularized by Disney as animatronics.
Animators are artists who specialize in creating animation

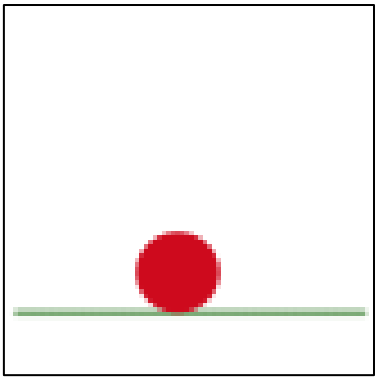
The 12 Principles of Animation is a group of key teachings for the professional animator. The list has served Disney animators since the 1930s and was outlined by Ollie Johnston and Frank Thomas in the 1981 book The Illusion of Life: Disney Animation. Many of these foundational ideas are still utilized in classrooms and studios around the world almost 40 years later. While technology and industries have evolved with new and different ideas being integrated into animation, the principles can still be seen in movies and web design today.
So what are the 12 Principles of Animation?

1. **Squash and stretch**
2. **Anticipation**
3. **Staging**
4. **Straight-ahead action and pose-to-pose**
5. **Follow through and overlapping action**

6. **Slow in and slow out**
7. **Arc**
8. **Secondary action**
9. **Timing**
10. **Exaggeration**
11. **Solid drawing**
12. **Appeal**

We caught up with Animation faculty Alex Salsberg to get his take on the Principles and if they play a role in the classes he teaches and his own animation work.

What's your take on the 12 Principles?
While I don't think they're the only important things to learn about animation, I think the 12 Principles are a really good launching point, especially for students studying to be professional animators. I think they've stuck around for a reason, even if that reason is sometimes to "learn the rules before you break them."



Source Code:

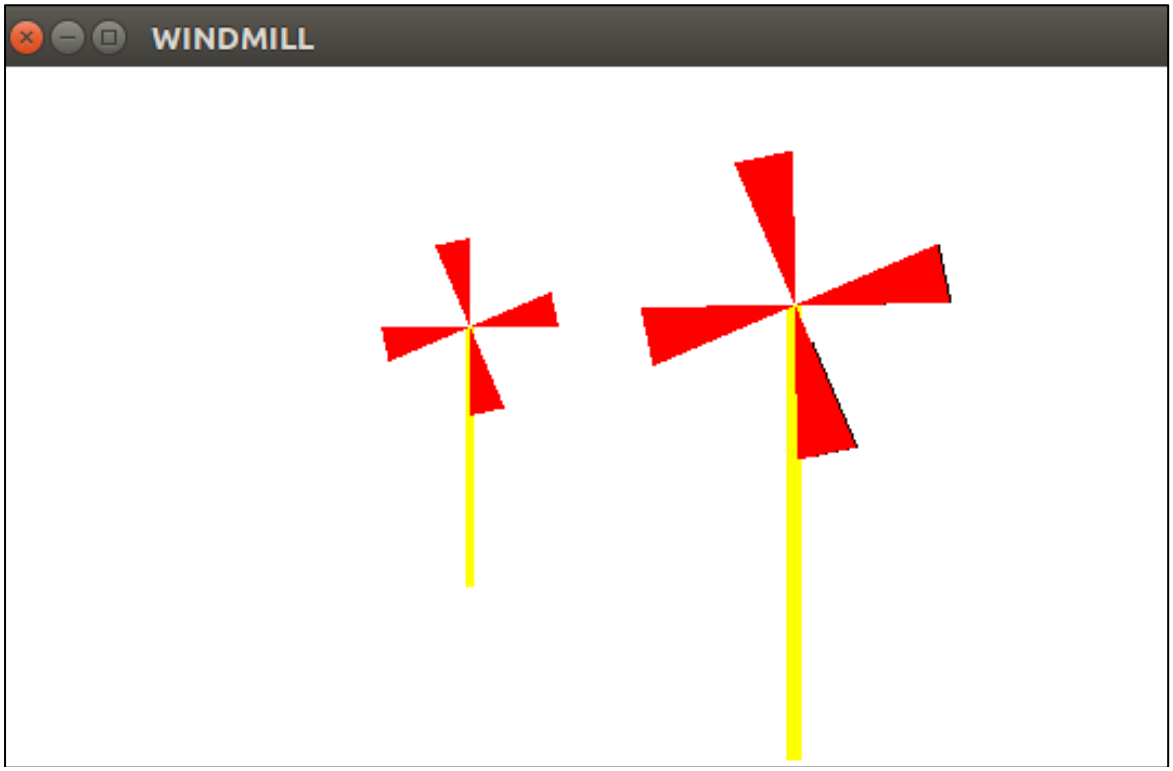
```
#include <GL/gl.h>
#include <GL/glut.h>
#include <math.h>


                        //global variable diclaration
int frameNumber = 0;
        //frame no


void drawWindmill()
        //Function to draw windmill
{

        int i;

        glColor3f(1.0,1.0,0.0);
        //red green blue

        glBegin(GL_POLYGON);
```

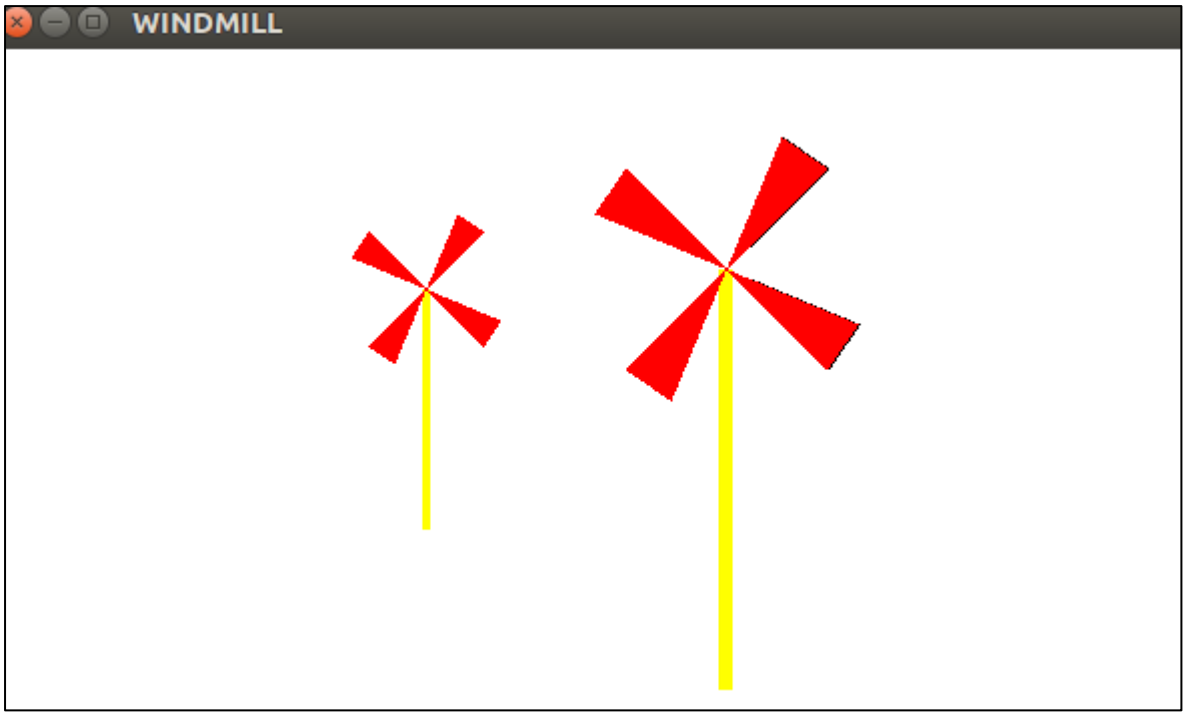
```
        glVertex2f(-0.05f, 0);
        //for drawing rectangular base part
        glVertex2f(-0.05f, 3);
        glVertex2f(0.05f, 3);
        glVertex2f(0.05f, 0);

    glEnd();

        glTranslatef(0,3,0);
        //x,y,z

        glColor3f(1.0,0.0,0.0);
        //red,green,blue (RED PLATES OF WINDMILL)

        glRotated(frameNumber * (180.0/45), 0, 0, 1);                    //(angle,x,y,z)

                for (i = 0; i < 4; i++)                                          //LOOP
TO DRAW FOUR PLATES
                {

                                glRotated(90, 0, 0, 1);
        //90,0,0,Z

                                glBegin(GL_POLYGON);

                                glVertex2f(0,0);
        //FOR DRAWING TYIANGLULAR PLATE

                                glVertex2f(1.0f, 0.2f);

                                glVertex2f(1.0f,-0.2f);

                        glEnd();
            }
}

void display()
         //DISPLAY FUNCTION
{
        glClear(GL_COLOR_BUFFER_BIT);


        glLoadIdentity();
        //TAKES IDENTITY MATRIX

        glPushMatrix();
                //PUSH MATRIX

        glTranslated(2.2,1.6,0);
        //SET POSITION OF WINDMILL
```

```
        glScaled(0.4,0.4,1);
        //SCALLING WINDMILL WITH POINT (0.4,0.4,1)

        drawWindmill();
                //FUNCTION CALL TO DRAW WINDMILL

        glPopMatrix();
                //POP MATRIX


        glPushMatrix();
                //PUSH MATRIX

        glTranslated(3.7,0.8,0);
        //SET POSITION OF WINDMILL

        glScaled(0.7,0.7,1);
        //SCALLING WINDMILL WITH POINT(0.7,0.7,1)

        drawWindmill();
                //FUNCTION CALL TO DRAW WINDMILL

        glPopMatrix();
                //POP MATRIX



        glutSwapBuffers();
        //SWAP BUFFER
}

void doFrame(int v)
{
   frameNumber++;
        //INCREMENT FRAME NO

   glutPostRedisplay();
        //POST REDISPLAY

   glutTimerFunc(10,doFrame,0);
}

void init()
                //FUNCTION INITIALISATION
{
        glClearColor(0,0,0,0);

        glMatrixMode(GL_PROJECTION);
        //MATRIX MODE  FOR PROJECTION

        glLoadIdentity();
        //LOADS IDENTITY MATRIX
```

```
        glOrtho(0, 7, -1, 4, -1, 1);                                    //MIN
X,MAX X,MIN Y,MAX Y,MIN Z,MAX Z VALUE

        glMatrixMode(GL_MODELVIEW);
        //MATRIX MODE FOR MODEL VIEW
}

int main(int argc, char** argv)                                         //MAIN
FUNCTION
{
   glutInit(&argc, argv);

   glutInitDisplayMode(GLUT_DOUBLE);

   glutInitWindowSize(700,500);                                         //DEFINED
WINDOW SIZE 700*500

   glutInitWindowPosition(100,100);                                     //DEFINED
WINDOW POSITION 100,100

   glutCreateWindow("WINDMILL");
        //NAME OF WINDOW

   init();
                //FIRSTLY CALL TO INTIALISE VALUE

   glutDisplayFunc(display);
        //DISPLAY

   glutTimerFunc(200,doFrame,0);                                //TIMER FUNC

   glutMainLoop();

   return 0;
}
```

Output :





Conclusion:

Successfully implemented Animated Windmill .