Sinhgad Technical Education Society's

SINHGAD COLLEGE OF ENGINEERING, PUNE

S.No. 44/1, Off Sinhgad Road, Vadgaon(BK), Pune- 411041

Accredited by NAAC with Grade 'A'

Sinhgad Institutes

DEPARTMENT OF INFORMATION TECHNOLOGY

e-Certificate

This is to certify that, Mr. / Ms. PIYUSH B KAUTKAR of Class SE -I, Division I, Roll No. 207A029 and PRN No. 72022191K has successfully completed all the practical work and submitted online in the subject Computer Graphics Laboratory and his performance was satisfactory in the said subject as prescribed by Savitribai Phule Pune University, for the academic year 2020-21, Semester-II.

Date: 06/06/2021

Mrs. J. S. Kulkarni	Ganesh Pathak	Dr. S.D. Lokhande
Name of Subject Teacher	Head of Department	Principal

Sinhgad Institutes

SINHGAD COLLEGE OF ENGINEERING, PUNE

DEPARTMENT OF INFORMATION TECHNOLOGY

ACADEMIC YEAR 2020-21, SEMESTER-II

Name of Student	**:**	PIYUSH B KAUTKAR	**PRN No.**	:	72022191K
Student Roll No.	**:**	207A029	**Class**	:	SE -I
Subject	**:**	Computer Graphics Laboratory	**Batch**	:	B

I N D E X

Exp. No.	Date	Name of Experiment	Faculty Remarks, if any
1	04/03/2021	Install and explore the OpenGL.	☐
2	12/03/2021	Implement DDA and Bresenham line drawing algorithm to draw	☐
3	06/04/2021	Implement Bresenham circle drawing algorithm to draw any object. The object should be displayed in all the quadrants with respect to center and	☐
4	09/04/2021	Implement the following polygon filling methods: i) Flood fill / Seed fill ii) Boundary fill; using mouse click, keyboard interface and menu driven	☐
5	14/04/2021	Implement Cohen Sutherland polygon clipping method to clip the polygon with respect the viewport and window. Use mouse click, keyboard interface.	☐
6	21/05/2021	Implement following 2D transformations on the object with respect to axis i) Scaling ii) Rotation about arbitrary point iii) Reflection	☐
7	26/05/2021	Generate fractal patterns using i) Bezier ii) Koch Curve.	☐
8	28/05/2021	Implement animation principles for any object.	☐
9			☐
10			☐
11			☐
12			☐
13			☐
14			☐
15			☐

Name of Subject Teacher	**:**	Mrs. J. S. Kulkarni

214455: COMPUTER GRAPHICS LABORATORY

SEIT (2019 Course)

Semester - II

Teaching Scheme		Examination Scheme	
Practical :	2 Hrs. / Week	Term work :	25 Marks
		Practical	50 Marks

तमसो मा ज्योतिर्गमय

Sinhgad Institutes

LABORATORY MANUAL V 3.0

DEPARTMENT OF INFORMATION TECHNOLOGY

Sinhgad College of Engineering, Pune

2020-2021

VISION

To provide excellent Information Technology education by building strong teaching and research environment.

MISSION

1) To transform the students into innovative, competent and high quality IT professionals to meet the growing global challenges.
2) To achieve and impart quality education with an emphasis on practical skills and social relevance.
3) To endeavor for continuous up-gradation of technical expertise of students to cater to the needs of the society.
4) To achieve an effective interaction with industry for mutual benefits.

PROGRAM EDUCATIONAL OBJECTIVES

The students of Information Technology course after passing out will

1) Graduates of the program will possess strong fundamental concepts in mathematics, science, engineering and Technology to address technological challenges.

2) Possess knowledge and skills in the field of Computer Science & Engineering and Information Technology for analyzing, designing and implementing complex engineering problems of any domain with innovative approaches.

3) Possess an attitude and aptitude for research, entrepreneurship and higher studies in the field of Computer Science & Engineering and Information Technology.

4) Have commitment to ethical practices, societal contributions through communities and life-long learning.

5) Possess better communication, presentation, time management and team work skills leading to responsible & competent professionals and will be able to address challenges in the field of IT at global level.

PROGRAM OUTCOMES

The students in the Information Technology course will attain:

a. an ability to apply knowledge of computing, mathematics including discrete mathematics as well as probability and statistics, science, and engineering and technology;
b. an ability to define a problem and provide a systematic solution with the help of conducting experiments, as well as analyzing and interpreting the data;
c. an ability to design, implement, and evaluate a software or a software/hardware system, component, or process to meet desired needs within realistic constraints;
d. an ability to identify, formulate, and provide systematic solutions to complex engineering problems;
e. an ability to use the techniques, skills, and modern engineering technologies tools, standard processes necessary for practice as a IT professional;
f. an ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer-based systems with necessary constraints and assumptions;
g. an ability to analyze the local and global impact of computing on individuals, organizations and society;
h. an ability to understand professional, ethical, legal, security and social issues and responsibilities;
i. an ability to function effectively as an individual or as a team member to accomplish a desired goal(s);
j. an ability to engage in life-long learning and continuing professional development to cope up with fast changes in the technologies/tools with the help of electives, professional organizations and extra-curricular activities;
k. an ability to communicate effectively in engineering community at large by means of effective presentations, report writing, paper publications, demonstrations;
l. an ability to understand engineering, management, financial aspects, performance, optimizations and time complexity necessary for professional practice;
m. an ability to apply design and development principles in the construction of software systems of varying complexity.

Compliance
Document Control

Reference Code	SCOE-IT / Lab Manual Procedures
Version No	2.0
Compliance Status	Complete
Revision Date	15 Nov 2019
Security Classification	Department Specific
Document Status	Definitive
Review Period	Yearly

Author
Mrs. S. M. Jaybhaye
Mrs.J.S.Kulkarni
Assistant Professor, SCOE(IT)

Document History

Revision No.	**Revision Date**	**Reason For Change**
1.0	15 Dec 2012	University Syllabus Modification - Course 2012
2.0	15 Nov 2016	University Syllabus Modification - Course 2016
3.0	15 Nov 2019	University Syllabus Modification - Course 2019

Summary of Changes to Computer Graphics Lab - V Manual

Sr. No	**Changes**	**Change type**
01	Theory	
02	FAQs	
03	Detailed Conceptual elaboration of each assignment	
04	University Syllabus Modification	

Syllabus

214456 : COMPUTER GRAPHICS LABORATORY
Teaching Scheme: Term Work : 25 Marks
Practical :2 Hours/Week Examination Scheme: Practical : 50 Marks

Prerequisites:

1. Basic Geometry, Trigonometry, Vectors and Matrices
2. Basics of Data Structures and Algorithms

Course Objectives:

1. To acquaint the learners with the basic concepts of Computer Graphics
2. To learn the various algorithms for generating and rendering graphical figures
3. To get familiar with mathematics behind the graphical transformations
4. To understand and apply various methods and techniques regarding projections, animation, Shading, illumination and lighting

Course Outcomes:

On completion of the course, learner will be able to –

1. Apply mathematics and logic to develop Computer programs for elementary graphic operations
2. Develop scientific and strategic approach to solve complex problems in the domain of Computer Graphics
3. Develop the competency to understand the concepts related to Computer Vision and Virtual reality
4. Apply the logic to develop animation and gaming programs

Suggested List of Laboratory Assignments

1. Install and explore the OpenGL.
2. Implement DDA and Bresenham line drawing algorithm to draw:
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
4. Implement the following polygon filling methods: i) Flood fill / Seed fill ii) Boundary fill; using mouse click, keyboard interface and menu driven programming
5. Implement Cohen Sutherland polygon clipping method to clip the polygon with respect the viewport and window. Use mouse click, keyboard interface
6. 6.Implement following 2D transformations on the object with respect to axis
   i) Scaling ii) Rotation about arbitrary point iii) Reflection
7. Generate fractal patterns using i) Bezier ii) Koch Curve
8. Implement animation principles for any object.

INDEX

SCHEDULE

Assig n No.	Assignment Title	No. of Hrs.	Week No.
1	Install and explore the OpenGL.	2	Wk 1
2	Implement DDA and Bresenham line drawing algorithm to draw	2	WK2
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.	4	WK3,WK 4
4	Implement the following polygon filling methods: i) Flood fill / Seed fill ii) Boundary fill; using mouse click, keyboard interface and menu driven programming.	2	WK5
5	Implement Cohen Sutherland polygon clipping method to clip the polygon with respect the viewport and window. Use mouse click, keyboard interface.	2	WK6
6	Implement following 2D transformations on the object with respect to axis i) Scaling ii) Rotation about arbitrary point iii) Reflection	2	WK7
7	Generate fractal patterns using i) Bezier ii) Koch Curve.	4	WK8, WK9
8	Implement animation principles for any object.	2	WK10

Assignment No. : 1

Install and explore the OpenGL.

Assignment No. : 1

Install and explore the OpenGL.

Aim
Install and explore the OpenGL.

Objective(s)	
1	To learn OpenGL Architecture
2	OpenGL Primitives

Theory
OpenGL is a hardware-independent, operating system independent, vendor neutral graphics API specification. Many vendors provide implementations of this specification for a variety of hardware platforms. Bindings exist primarily for the C programming language, but bindings are also available for Fortran and Ada. OpenGL has been designed using a client/server paradigm, allowing the client application and the graphics server controlling the display hardware to exist on the same or separate machines. The network is transparent to the application.OpenGL is window system independent, and therefore contains no windowing operations or mechanisms for user input. Also, OpenGL does not provide direct support for

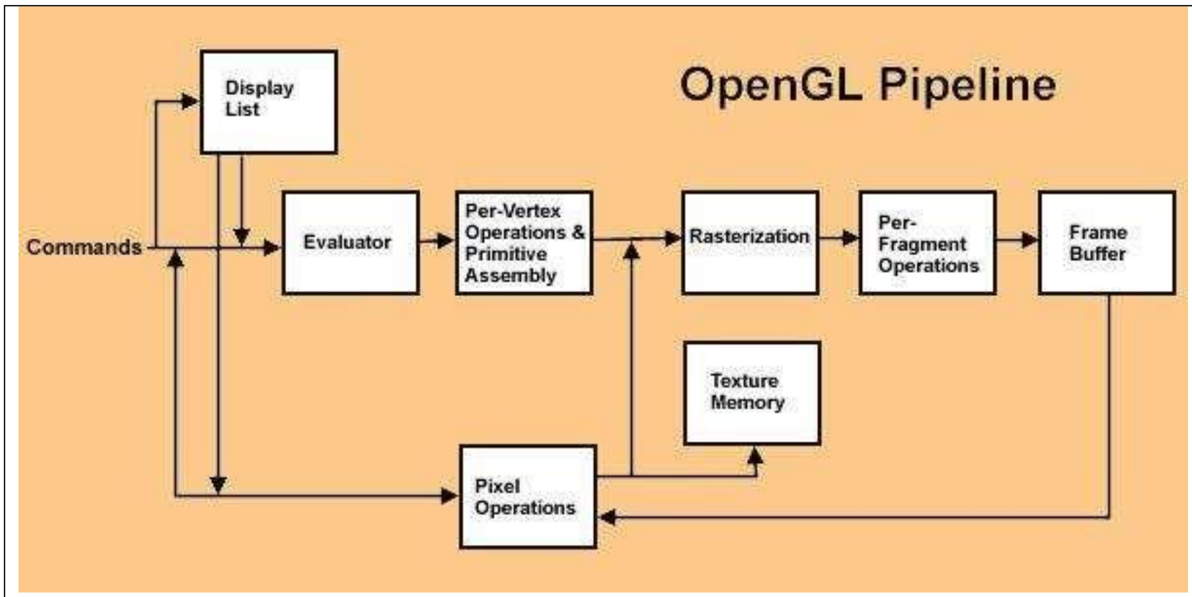
complex geometrical shapes, such as cubes or spheres. These must be built up from supported primitives.

Some features of OpenGL include the following:

- Geometric and raster primitives
- RGBA or color index mode
- Display list or immediate mode
- Viewing and modeling transformations
- Lighting and Shading
- Hidden surface removal (Depth Buffer)
- Anti-aliasing
- Texture Mapping
- Feedback and Selection
- Accumulation Buffer
- Depth Cueing
- Wireframes

OpenGL Pipeline

The OpenGL architecture is structured as a state-based pipeline. Below is a simplified diagram of this pipeline. Commands enter the pipeline from the left.



Commands may either be accumulated in display lists, or processed immediately through the pipeline. Display lists allow for greater optimization and command reuse, but not all commands can be put in display lists.

The first stage in the pipeline is the evaluator. This stage effectively takes any polynomial evaluator commands and evaluates them into their corresponding vertex and attribute commands.

The second stage is the per-vertex operations, including transformations, lighting, primitive assembly, clipping, projection, and viewport mapping.

The third stage is rasterization. This stage produces fragments, which are series of framebuffer addresses and values, from the viewport-mapped primitives as well as bitmaps and pixel rectangles.

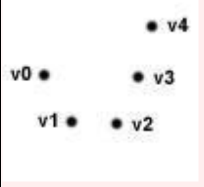
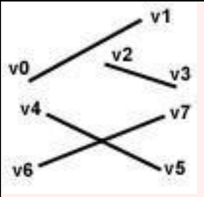
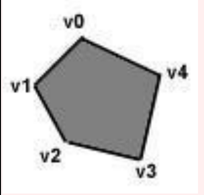
The fourth stage is the per-fragment operations. Before fragments go to the framebuffer, they may be subjected to a series of conditional tests and modifications, such as blending or z-buffering.

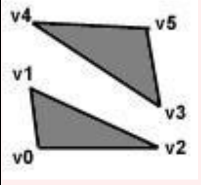
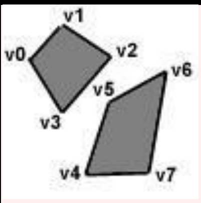
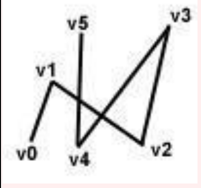
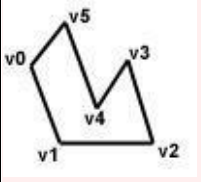
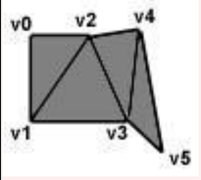
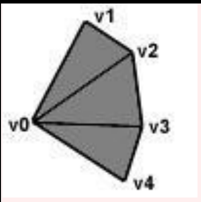
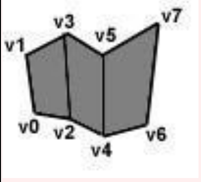
Parts of the framebuffer may be fed back into the pipeline as pixel rectangles. Texture memory may be used in the rasterization process when texture mapping is enabled.

Vertices and Primitives

Most objects (with the exception of pixel rectangles and bitmaps), use Begin/End primitives. Each Begin/End primitive contains a series of vertex data, and may optionally contain normals, texture coordinates, colors, edge flags, and material properties.

There are ten primitive types, as follows:

Points	individual points	
Lines	pairs of vertices interpreted as individual line segments	
Polygon	boundary of a simple, convex polygon	

Triangles	triples of vertices interpreted as triangles	
Quads	quadruples of vertices interpreted as four-sided polygons	
Line Strip	series of connected line segments	
Line Loop	same as above, with a segment added between last and first vertices	
Triangle Strip	linked strip of triangles	
Triangle Fan	linked fan of triangles	
Quad Strip	linked strip of quadrilaterals	

Clipping and Projection

Once a primitive has been assembled, it is subject to arbitrary clipping via user definable clip planes. An OpenGL implementation must provide at least six, and they may be turned on and off independently by the user.

Points are either clipped in our out, depending on whether they fall inside or outside the half-space defined by the clip planes. Lines and polygons, however, may either be 100% clipped, 100% unclipped, or they may fall partially within the clip space. In this latter case, new vertices are automatically place on the clip boundary between pre-existing vertices. Vertex attributes are interpolated.

After clipping, vertices are transformed by the projection matrix (either perspective or orthogonal), and then clipped to the frustum (view space), following the same process as above. Finally, the vertices are mapped to the viewport (screen space).

Rasterization

Rasterization converts the above viewport-mapped primitives into fragments. Fragments consist of pixel location in framebuffer, color, texture coordinates, and depth (z buffer). Depending on the shading mode, vertex attributes are either interpolated across the primitive to all fragments (smooth shading), or all fragments are assigned the same values based on one vertex's attributes (flat shading).

Rasterization is affected by the point and line widths, the line stipple sequence, and the polygon stipple pattern. Antialiasing may be enabled or disabled for each primitive type. If enabled, the alpha color value (if in RGBA mode) or color index (if in CI mode) are modified to reflect sub-pixel coverage.

Pixel rectangles and bitmaps are also rasterized, but they bypass the lighting and geometrical transformations. They are groups of values heading for the framebuffer. They can be scaled, offset, and mapped via lookup tables. The rasterization process produces a rectangle of fragments at a location controlled by the current raster position state variable. The size may be affected by the pixel zoom setting.

Bitmaps are similar to pixel rectangles, but the data is binary, only producing fragments when on. This is useful for drawing text in 3D space as part of a scene.

Framebuffer

Fragments produced by rasterization go to the framebuffer where they may be displayed. The framebuffer is a rectangular array of *n* bitplanes. The bitplanes are organized into several logical buffers -- Color, Depth, Stencil, and Accumulation.

The color buffer contains the fragment's color info.The depth buffer contains the fragment's depth info, typically used for z-buffering hidden surface removal.The stencil buffer can be associated with fragments that pass the conditional tests described below and make it into the framebuffer. It can be useful for multiple-pass algorithms.The accumulation buffer is also used for multiple-pass algorithms. It can average the values stored in the color buffer. Full-screen antialiasing can be achieved by jittering the viewpoint. Depth of Field can be achieved by jittering the view angle. Motion blur can be achieved by stepping the scene in time.

Stereo and double-buffering may be supported under OpenGL, depending on the implementation. These would further divide the framebuffer into up to 4 sections -- front and back buffer, left and right. Other auxiliary buffers may be available on some implementations. Any buffers may be individually enabled or disabled for writing. The depths and availabilities of buffers may vary, but must meet the minimum requirement of OpenGL. Each buffer may be individually cleared to a specified value.

OpenGL Commands

1. Primitives

Specify vertices or rectangles:

void `glBegin` (GLenum *mode*);
void `glEnd` (void);
void `glVertex2{sifd}{v}` (TYPE *x*, TYPE *y*);
void `glVertex3{sifd}{v}` (TYPE *x*, TYPE *y*, TYPE *z*);

2. Transform the current matrix:

void `glRotate{fd}` (TYPE *angle*, TYPE *x*, TYPE *y*, TYPE *z*);
void `glTranslate{fd}` (TYPE *x*, TYPE *y*, TYPE *z*);
void `glScale{fd}` (TYPE *x*, TYPE *y*, TYPE *z*);
void `glMultMatrix{fd}` (const TYPE **m*);
void `glFrustum` (GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);
void `glOrtho` (GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);

3. Replace the current matrix:

void `glLoadMatrix{fd}` (const TYPE **m*);
void `glLoadIdentity` (void);

4. Manipulate the matrix stack:

void `glMatrixMode` (GLenum *mode*);

void `glPushMatrix` (void);

void `glPopMatrix` (void);

5. Specify the viewport:

void `glDepthRange` (GLclampd *near*, GLclampd *far*);

void `glViewport` (GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

6. Coloring and Lighting

Set the current color, color index, or normal vector:

void `glColor3{bsifd ubusui}{v}` (TYPE *red*, TYPE *green*, TYPE *blue*);

void `glColor4{bsifd ubusui}{v}` (TYPE *red*, TYPE *green*, TYPE *blue*, TYPE *alpha*);

void `glIndex{sifd}{v}` (TYPE *index*);

void `glNormal3{bsifd}{v}` (TYPE *nx*, TYPE *ny*, TYPE *nz*);

7.Specify a bitmap:

void `glBitmap` (GLsizei *width*, GLsizei *height*, GLfloat *xorig*, GLfloat *yorig*, GLfloat *xmove*, GLfloat *ymove*, const GLubyte **bitmap*);

Specify the dimensions of points or lines:

void `glPointSize` (GLfloat *size*);

void `glLineWidth` (GLfloat *width*);

Control pixel rasterization:

void `glPixelZoom` (GLfloat *xfactor*, GLfloat *yfactor*);

Assignment No. : 2

Implement DDA and Bresenham line drawing algorithm to draw

Assignment No. : 2

Implement DDA and Bresenham line drawing algorithm to draw

Aim
Implement DDA and Bresenham line drawing algorithm to draw

Objective(s)	
1	To learn DDA and Bresenham line drawing
2	To implement line drawing algorithm

Theory
A line connects two points. It is a basic element in graphics. To draw a line, you need two points between which you can draw a line. In the following three algorithms, we refer the one point of line as X0,Y0X0,Y0 and the second point of line as X1,Y1X1,Y1.

DDA Algorithm

Digital Differential Analyzer (DDA) algorithm is the simple line generation algorithm which is explained step by step here.

Step 1 – Get the input of two end points (X0,Y0)(X0,Y0) and (X1,Y1)(X1,Y1).

Step 2 – Calculate the difference between two end points.

```
dx = X1 - X0
dy = Y1 - Y0
```

Step 3 – Based on the calculated difference in step-2, you need to identify the number of steps to put pixel. If dx > dy, then you need more steps in x coordinate; otherwise in y coordinate.

```
if (absolute(dx) > absolute(dy))

  Steps = absolute(dx);

else

  Steps = absolute(dy);
```

Step 4 – Calculate the increment in x coordinate and y coordinate.

```
Xincrement = dx / (float) steps;
Yincrement = dy / (float) steps;
```

Step 5 – Put the pixel by successfully incrementing x and y coordinates accordingly and complete the drawing of the line.

```
for(int v=0; v < Steps; v++)

{

  x = x + Xincrement;

  y = y + Yincrement;

  putpixel(Round(x), Round(y));

}
```

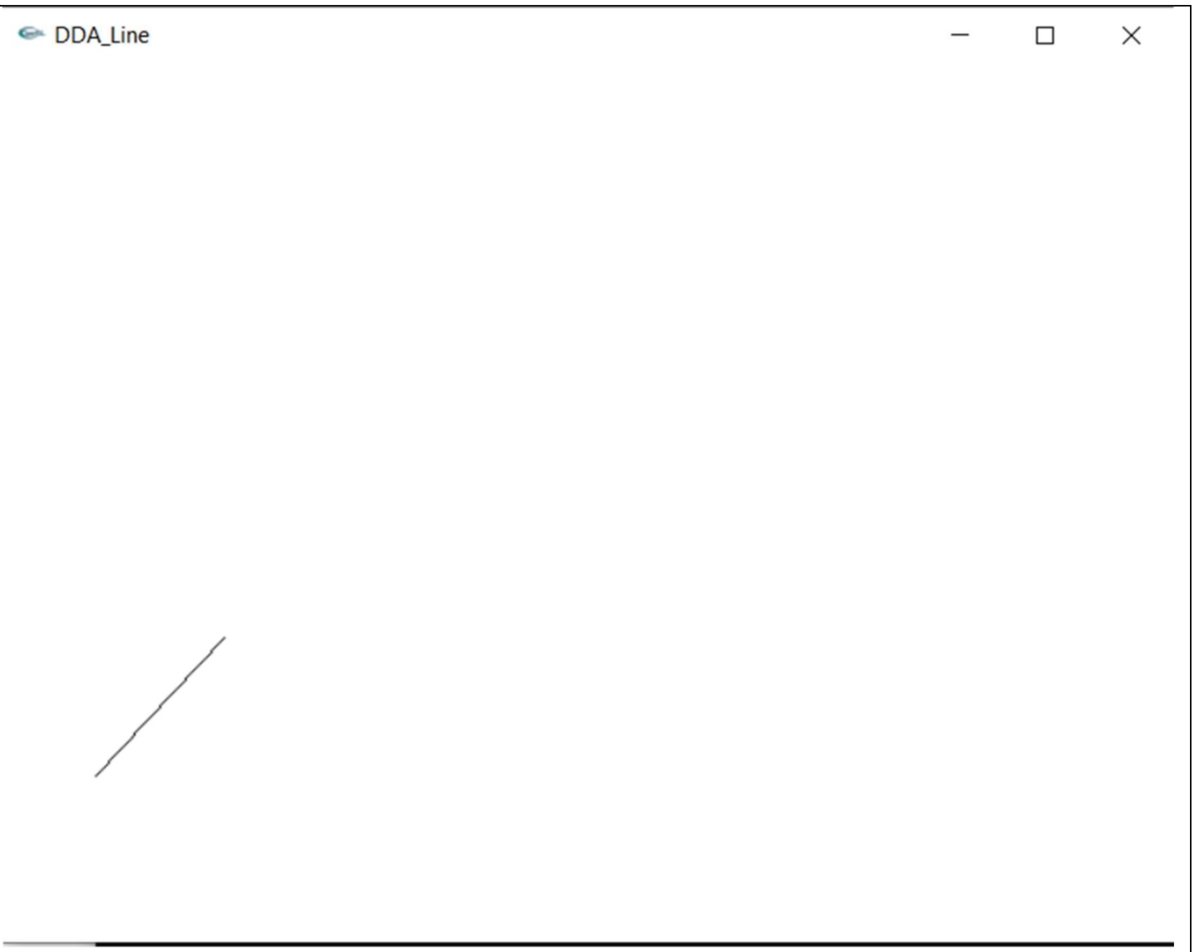
<u>INPUT</u>

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

```
double X1, Y1, X2, Y2;

float round_value(float v)
{
 return floor(v + 0.5);
}
void LineDDA(void)
{
 double dx=(X2-X1);
 double dy=(Y2-Y1);
 double steps;
 float xInc,yInc,x=X1,y=Y1;
 /* Find out whether to increment x or y */
 steps=(abs(dx)>abs(dy))?(abs(dx)):(abs(dy));
 xInc=dx/(float)steps;
 yInc=dy/(float)steps;

 /* Clears buffers to preset values */
 glClear(GL_COLOR_BUFFER_BIT);

 /* Plot the points */
 glBegin(GL_POINTS);
 /* Plot the first point */
 glVertex2d(x,y);
 int k;
 /* For every step, find an intermediate vertex */
 for(k=0;k<steps;k++)
 {
  x+=xInc;
  y+=yInc;
  /* printf("%0.6lf %0.6lf\n",floor(x), floor(y)); */
  glVertex2d(round_value(x), round_value(y));
 }
 glEnd();

 glFlush();
}
void Init()
{
 /* Set clear color to white */
 glClearColor(1.0,1.0,1.0,0);
 /* Set fill color to black */
 glColor3f(0.0,0.0,0.0);
 /* glViewport(0 , 0 , 640 , 480); */
 /* glMatrixMode(GL_PROJECTION); */
 /* glLoadIdentity(); */
 gluOrtho2D(0 , 640 , 0 , 480);
}
int main(int argc, char **argv)
{
 printf("Enter two end points of the line to be drawn:\n");
 printf("\n************************************");
 printf("\nEnter Point1( X1 , Y1):\n");
 scanf("%lf%lf",&X1,&Y1);
 printf("\n************************************");
 printf("\nEnter Point1( X2 , Y2):\n");
 scanf("%lf%lf",&X2,&Y2);

 /* Initialise GLUT library */
 glutInit(&argc,argv);
 /* Set the initial display mode */
 glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
 /* Set the initial window position and size */
 glutInitWindowPosition(0,0);
 glutInitWindowSize(640,480);
 /* Create the window with title "DDA_Line" */
 glutCreateWindow("DDA_Line");
 /* Initialize drawing colors */
 Init();
 /* Call the displaying function */
 glutDisplayFunc(LineDDA);
 /* Keep displaying untill the program is closed */
 glutMainLoop();
}
```

OUTPUT

```
Enter two end points of the line to be drawn:

************************************
Enter Point1( X1 , Y1):
50
90

************************************
Enter Point1( X2 , Y2):
120
165
```



Difference Between DDA Line Drawing Algorithm and Bresenhams Line Drawing Algorithm

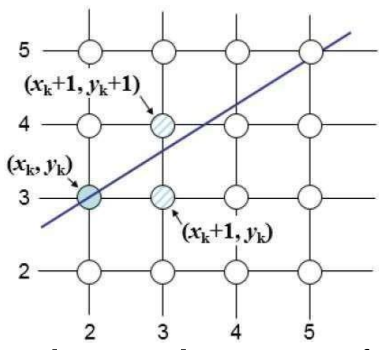
	Digital Differential Analyzer	Bresenhams Line Drawing Algorithm
Arithmetic	DDA algorithm uses **floating points** i.e. **Real Arithmetic**.	Bresenhams algorithm uses **fixed points** i.e. **Integer Arithmetic**.
Operations	DDA algorithm uses **multiplication** and **division** in its operations.	Bresenhams algorithm uses only **subtraction** and **addition** in its operations.
Speed	DDA algorithm is rather **slow**ly than Bresenhams algorithm in line	Bresenhams algorithm is faster than DDA algorithm in line drawing because it

	drawing because it uses real arithmetic (floating-point operations).	performs only addition and subtraction in its calculation and uses only integer arithmetic so it runs significantly **faster**.
Accuracy & Efficiency	DDA algorithm is not as accurate and efficient as Bresenham algorithm.	Bresenhams algorithm is more efficient and much accurate than DDA algorithm.
Drawing	DDA algorithm can draw circles and curves but that are not as accurate as Bresenhams algorithm.	Bresenhams algorithm can draw circles and curves with much more accuracy than DDA algorithm.
Round Off	DDA algorithm round off the coordinates to integer that is nearest to the line.	Bresenhams algorithm does not **round off** but takes the incremental value in its operation.
Expensive	DDA algorithm uses an enormous number of floating-point multiplications so it is expensive.	Bresenhams algorithm is less expensive than DDA algorithm as it uses only addition and subtraction.

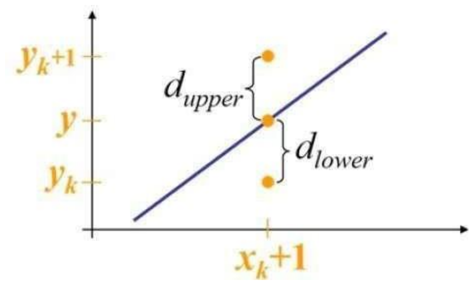
Bresenham's Line Generation

The Bresenham algorithm is another incremental scan conversion algorithm. The big advantage of this algorithm is that, it uses only integer calculations. Moving across the x axis in unit intervals and at each step choose between two different y coordinates.

For example, as shown in the following illustration, from position (2, 3) you need to choose between (3, 3) and (3, 4). You would like the point that is closer to the original line.



At sample position Xk+1,Xk+1, the vertical separations from the mathematical line are labelled as dupperdupper and dlowerdlower.



From the above illustration, the y coordinate on the mathematical line at xk+1xk+1is −

Y = m(XkXk+1) + b

So, dupperdupper and dlowerdlower are given as follows −

$$dlower=y-ykdlower=y-yk$$
$$=m(Xk+1)+b-Yk=m(Xk+1)+b-Yk$$
and

$$dupper=(yk+1)-ydupper=(yk+1)-y$$
$$=Yk+1-m(Xk+1)-b=Yk+1-m(Xk+1)-b$$
You can use these to make a simple decision about which pixel is closer to the mathematical line. This simple decision is based on the difference between the two pixel positions.

$$dlower-dupper=2m(xk+1)-2yk+2b-1dlower-dupper=2m(xk+1)-2yk+2b$$
$$1$$
Let us substitute *m* with *dy/dx* where *dx* and *dy* are the differences between the end-points.

$$dx(dlower-dupper)=dx(2dydx(xk+1)-2yk+2b-1)dx(dlower-dupper)=dx($$
$$2dydx(xk+1)-2yk+2b-1)$$
$$=2dy.xk-2dx.yk+2dy+2dx(2b-1)=2dy.xk-2dx.yk+2dy+2dx(2b-1)$$
$$=2dy.xk-2dx.yk+C=2dy.xk-2dx.yk+C$$
So, a decision parameter PkPk for the *k*th step along a line is given by −
$$pk=dx(dlower-dupper)pk=dx(dlower-dupper)$$
$$=2dy.xk-2dx.yk+C=2dy.xk-2dx.yk+C$$
The sign of the decision parameter PkPk is the same as that of dlower−dupperdlower−dupper.
If pkpk is negative, then choose the lower pixel, otherwise choose the upper pixel.
Remember, the coordinate changes occur along the x axis in unit steps, so you can do everything with integer calculations. At step k+1, the decision parameter is given as −

$$pk+1=2dy.xk+1-2dx.yk+1+Cpk+1=2dy.xk+1-2dx.yk+1+C$$
Subtracting pkpk from this we get −
$$pk+1-pk=2dy(xk+1-xk)-2dx(yk+1-yk)pk+1-pk=2dy(xk+1-xk)-2dx(yk+$$
$$yk)$$
But, xk+1xk+1 is the same as (xk)+1(xk)+1. So −
$$pk+1=pk+2dy-2dx(yk+1-yk)pk+1=pk+2dy-2dx(yk+1-yk)$$
Where, Yk+1–YkYk+1–Yk is either 0 or 1 depending on the sign of PkPk.
The first decision parameter p0p0 is evaluated at (x0,y0)(x0,y0) is given as −
p0=2dy−dxp0=2dy−dx
Now, keeping in mind all the above points and calculations, here is the Bresenham algorithm for slope m < 1 −

Step 1 − Input the two end-points of line, storing the left end-point in (x0,y0)(x0,y0).
Step 2 − Plot the point (x0,y0)(x0,y0).

Step 3 – Calculate the constants dx, dy, 2dy, and (2dy – 2dx) and get the first value for the decision parameter as –

$$p_0 = 2dy - dx$$ p0=2dy−dx

Step 4 – At each X_kXk along the line, starting at k = 0, perform the following test –

If $p_k$pk < 0, the next point to plot is (x_k+1, y_k)(xk+1,yk) and

$$p_{k+1} = p_k + 2dy$$ pk+1=pk+2dy

Otherwise,

$$p_{k+1} = p_k + 2dy - 2dx$$ pk+1=pk+2dy−2dx

Step 5 – Repeat step 4 (dx – 1) times.

For m > 1, find out whether you need to increment x while incrementing y each time.

After solving, the equation for decision parameter P_kPk will be very similar, just the x and y in the equation gets interchanged.

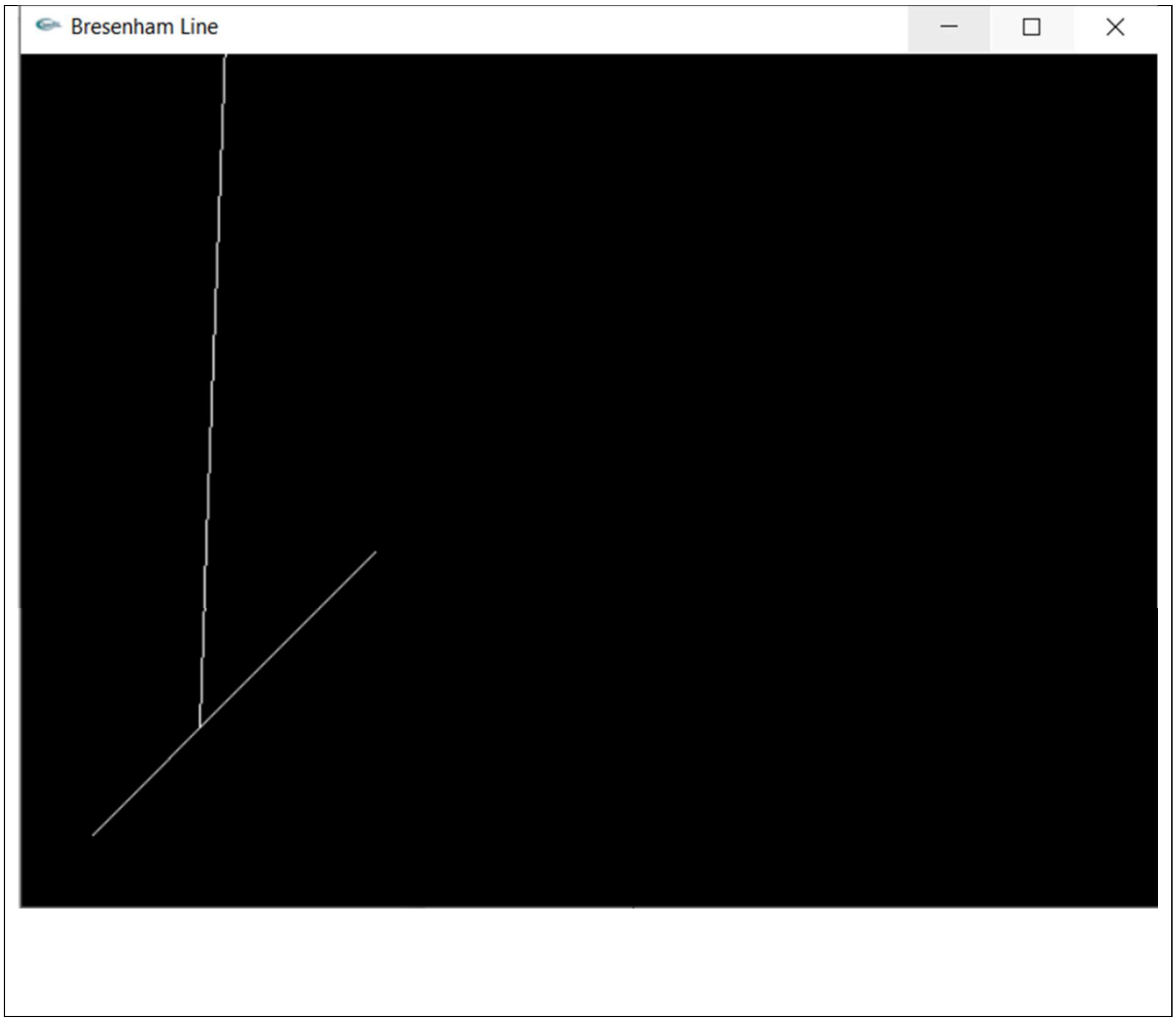
Input
Starting and ending point of line

```
#include<windows.h>
#include <GL/glut.h>
#include<math.h>
#include<stdlib.h>
#include<iostream>
#include<stdio.h>
using namespace std;
int sign(float arg)
{
if(arg<0)
return -1;
else if(arg==0)
return 0;
else
return 1;
}

void Bre(int X1,int Y1,int X2,int Y2)
{
  float dx=abs(X2-X1);
float dy=abs(Y2-Y1);
int  s1,s2,exc,y,x,i;
float g,temp;
x=X1;
```

```
y=Y1;
        s1=sign(X2-X1);
        s2=sign(Y2-Y1);

        glBegin(GL_POINTS);

        if(dy>dx)
        {
        temp=dx;
        dx=dy;
        dy=temp;
        exc=1;
        }
        else
        {
        exc=0;
        }
        g=2*dy-dx;
        i=1;
        while(i<=dx)
        {
                glVertex2d(x,y);
                while(g>=0)
                {
                        if(exc==1)
                        x=x+s1;
                        else
                        y=y+s2;
                        g=g-2*dx;
                }
                if(exc==1)
                y=y+s2;
                else
                x=x+s1;
        g=g+2*dy;
        i++;
        }
         glEnd();
        glFlush();
}



 void display()
 {
```

```
          Bre(40,40,200,200);
        Bre(250,4000,100,100);

}


 void myInit(void)
{         glClearColor(1.0,1.0,1.0,0.0);
          glColor3f(1,1,1);
          glPointSize(1.0);
          glMatrixMode(GL_PROJECTION);
          glLoadIdentity();
          gluOrtho2D(0.0,640.0,0.0,480.0);
}
int main(int argc,char **argv)
{
          glutInit(&argc,argv);
          glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
          glutInitWindowSize(640,480);
          glutInitWindowPosition(100,150);
          glutCreateWindow("Bresenham Line ");
          glutDisplayFunc(display);
          myInit();
          glutMainLoop();
          return 0;
}
```

Output
Given pattern in the problem statement



Lab. Based FAQ
Explain DDA Explain Bresenham Difference Between DDA and Bresenham

Assignment No. : 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.

Assignment No. : 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.

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

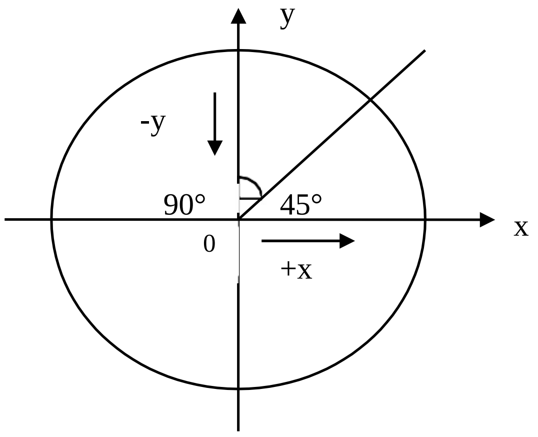
Objective(s)	
1	To Learn Circle drawing slgorithm

Theory

Bresenham's Circle Drawing:

A circle is a symmetrical figure. It has eight - way symmetry. If we know any single point of circle we can plot all remaining seven pixels using 8- way symmetry.
This algorithm considers 8 –way symmetry of circle and generates the whole circle.

1/8th part of circle i.e. from 90° to 45° is drawn, during this x increments in positive direction and y increments in negative direction. In this algorithm we have to select proper pixel which is either on the circle or closed to the circle port.



Decision Variable is given as d = 3 -2r

Algorithm:
1. Input radius r
2. Plot a point at (0, r)
3. Calculate the initial value of the decision parameter as

$$p_0 = 5/4 - r \approx 1 - r$$

```
4. At each position $x_k$, starting at k = 0, perform the following test:
        if $p_k < 0$
            plot point at $(x_{k+1}, y_k)$
            compute new $p_{k+1} = p_k + 2x_{k+1} + 1$
        else
            plot point at $(x_{k+1}, y_k - 1)$
            compute new $p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$
        where $x_{k+1} = x_k + 1$ and $y_{k+1} = y_k - 1$

5. Determine symmetry points in the other seven octants and plot points
6. Repeat steps 4 and 5 until $x \geq y$
```

Symmetry of Circle:

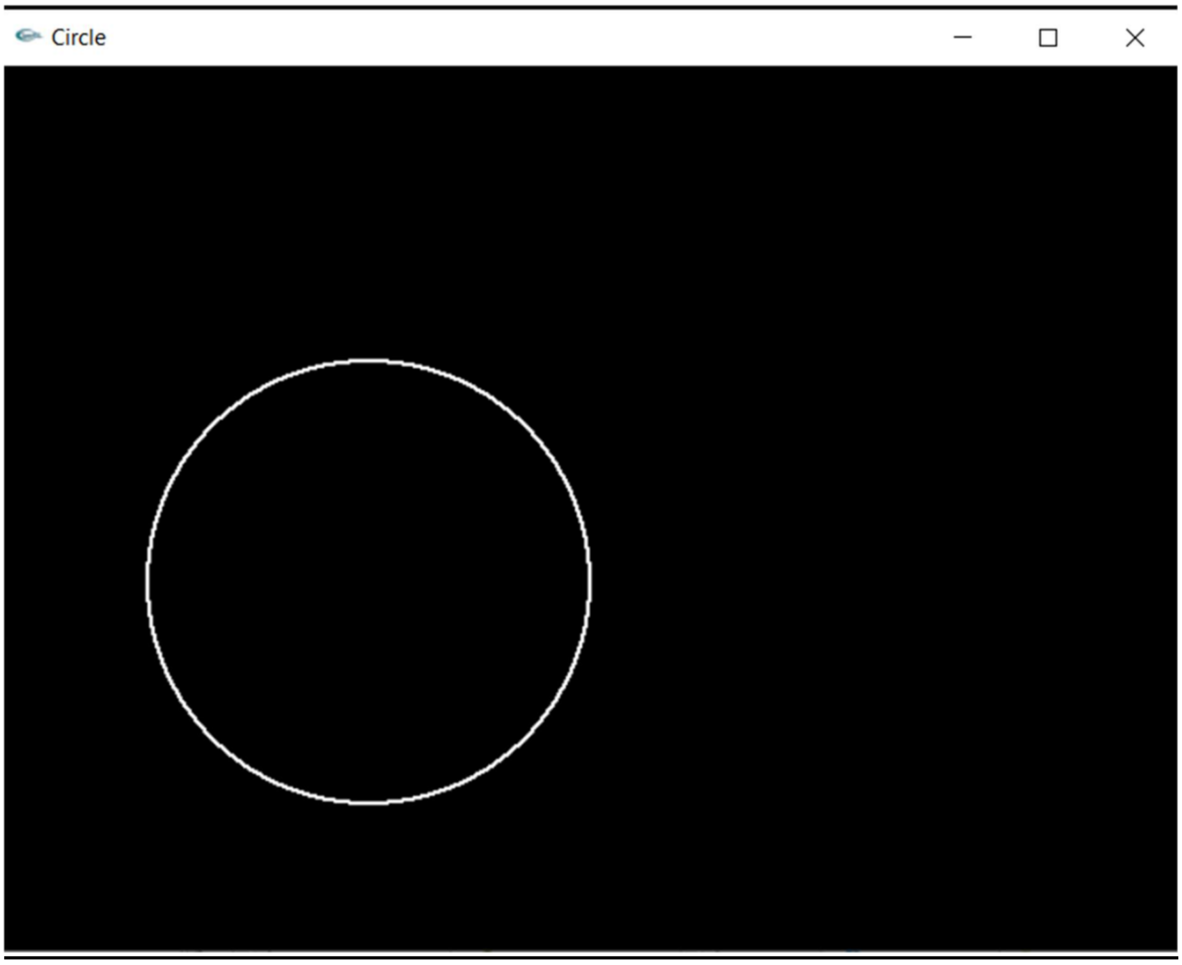
```
void symmetry (double xc,double yc,double x,double y)
{
  glBegin(GL_POINTS);
  glVertex2i(xc+x, yc+y);
  glVertex2i(xc+x, yc-y);
  glVertex2i(xc+y, yc+x);
  glVertex2i(xc+y, yc-x);
  glVertex2i(xc-x, yc-y);
  glVertex2i(xc-y, yc-x);
  glVertex2i(xc-x, yc+y);
  glVertex2i(xc-y, yc+x);
  glEnd();
}


void circle(double x1,double y1,double r)
{
  int x=0,y=r;
  float pk=(5.0/4.0)-r;
  /* Plot the points */
  /* Plot the first point */
  symmetry(x1,y1,x,y);
  int k;
  /* Find all vertices till x=y */
}
while(x < y)
  {
    x = x + 1;
    if(pk < 0)
      pk = pk + 2*x+1;
    else
    {
      y = y - 1;
      pk = pk + 2*(x - y) + 1;
    }
    symmetry(x1,y1,x,y);
  }
  glFlush();
```

Input

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



void symmetry(double xc,double yc,double x,double y)
{
  glBegin(GL_POINTS);

  glVertex2i(xc+x, yc+y);
  glVertex2i(xc+x, yc-y);
  glVertex2i(xc+y, yc+x);
  glVertex2i(xc+y, yc-x);
  glVertex2i(xc-x, yc-y);
  glVertex2i(xc-y, yc-x);
  glVertex2i(xc-x, yc+y);
  glVertex2i(xc-y, yc+x);
  glEnd();
}


void circle(double x1,double y1,double r)
{
  int x=0,y=r;
  float pk=(5.0/4.0)-r;

  /* Plot the points */
  /* Plot the first point */
  symmetry(x1,y1,x,y);
  int k;
  /* Find all vertices till x=y */
  while(x < y)
  {
    x = x + 1;
    if(pk < 0)
      pk = pk + 2*x+1;
    else
    {
      y = y - 1;
      pk = pk + 2*(x - y) + 1;
    }
    symmetry(x1,y1,x,y);
  }
  glFlush();

}

  void display()
  {

        circle(200,200,120);


}



 void myInit(void)
{         glClearColor(1.0,1.0,1.0,0.0);
          glColor3f(1,1,1);
          glPointSize(2.0);
          glMatrixMode(GL_PROJECTION);
          glLoadIdentity();
          gluOrtho2D(0.0,640.0,0.0,480.0);
}
```

```
int main(int argc,char **argv)
{
	glutInit(&argc,argv);
	glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
	glutInitWindowSize(640,480);
	glutInitWindowPosition(100,150);
	glutCreateWindow("Circle");
	glutDisplayFunc(display);
	myInit();
	glutMainLoop();
	return 0;
}
```

Output



Assignment No. : 4

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

Assignment No. : 4

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

Objective(s)	
	Use seed fill algorithm to fill

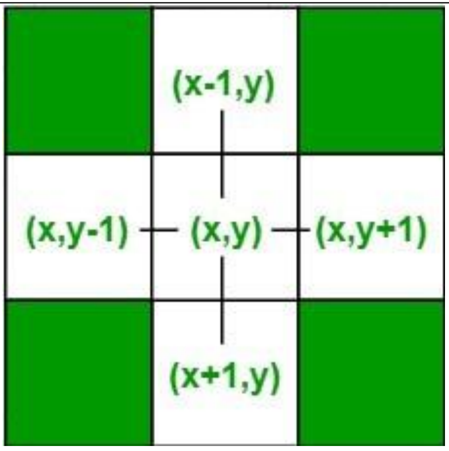
Theory
Boundary Fill Algorithm starts at a pixel inside the polygon to be filled and paints the interior proceeding outwards towards the boundary. This algorithm works **only if** the color with which the region has to be filled and the color of the boundary of the region are different. If the boundary is of one single color, this approach proceeds outwards pixel by pixel until it hits the boundary of the region. ***Boundary Fill Algorithm is recursive in nature.*** It takes an interior point(x, y), a fill color, and a boundary color as the input. The algorithm starts by checking the color of (x, y). If it's color is not equal to the fill color and the boundary color, then it is painted with the fill color and the function is called for all the neighbours of (x, y). If a point is found to be of fill color or of boundary color, the function does not call its neighbours and returns. This process continues until all points up to the boundary color for the region have been tested.

The boundary fill algorithm can be implemented by 4-connected pixels or 8-connected pixels.

4-connected pixels : After painting a pixel, the function is called for four neighboring points. These are the pixel positions that are right, left, above and below the current pixel. Areas filled by this method are called 4-connected. Below given is the algorithm

Algorithm :

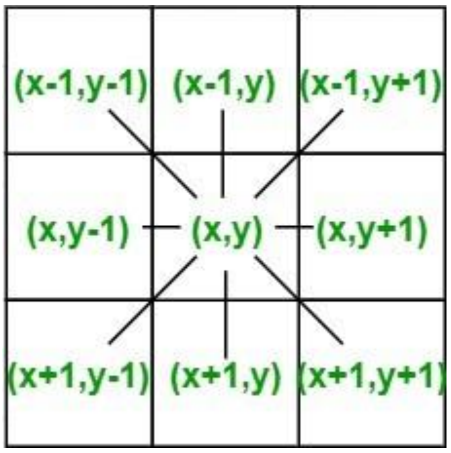
```
void boundaryFill4(int x, int y, int fill_color,int boundary_color)
{
    if(getpixel(x, y) != boundary_color &&
       getpixel(x, y) != fill_color)
    {
        putpixel(x, y, fill_color);
        boundaryFill4(x + 1, y, fill_color, boundary_color);
        boundaryFill4(x, y + 1, fill_color, boundary_color);
        boundaryFill4(x - 1, y, fill_color, boundary_color);
        boundaryFill4(x, y - 1, fill_color, boundary_color);
    }
}
```



8-connected pixels: More complex figures are filled using this approach. The pixels to be tested are the 8 neighboring pixels, the pixel on the right, left, above, below and the 4 diagonal pixels. Areas filled by this method are called 8-connected. Below given is the algorithm

Algorithm:

```
void boundaryFill8(int x, int y, int fill_color,int boundary_color)
{
    if(getpixel(x, y) != boundary_color &&
       getpixel(x, y) != fill_color)
    {
        putpixel(x, y, fill_color);
        boundaryFill8(x + 1, y, fill_color, boundary_color);
        boundaryFill8(x, y + 1, fill_color, boundary_color);
        boundaryFill8(x - 1, y, fill_color, boundary_color);
        boundaryFill8(x, y - 1, fill_color, boundary_color);
        boundaryFill8(x - 1, y - 1, fill_color, boundary_color);
        boundaryFill8(x - 1, y + 1, fill_color, boundary_color);
        boundaryFill8(x + 1, y - 1, fill_color, boundary_color);
        boundaryFill8(x + 1, y + 1, fill_color, boundary_color);
    }
}
```



Flood Fill algorithm

```
void FloodFill4(int x, int y, color newcolor, color oldColor)
{
  if(ReadPixel(x, y) == oldColor)
  {
    SetColor(newcolor);
  SetPixel(x, y);
  FloodFill4(x+1, y, newcolor, oldColor);
     FloodFill4(x-1, y, newcolor, oldColor);
     FloodFill4(x, y+1, newcolor, oldColor);
     FloodFill4(x, y-1, newcolor, oldColor);
  }
}
```

Input

```
#include<iostream>
#include<math.h>
#include<windows.h>
```

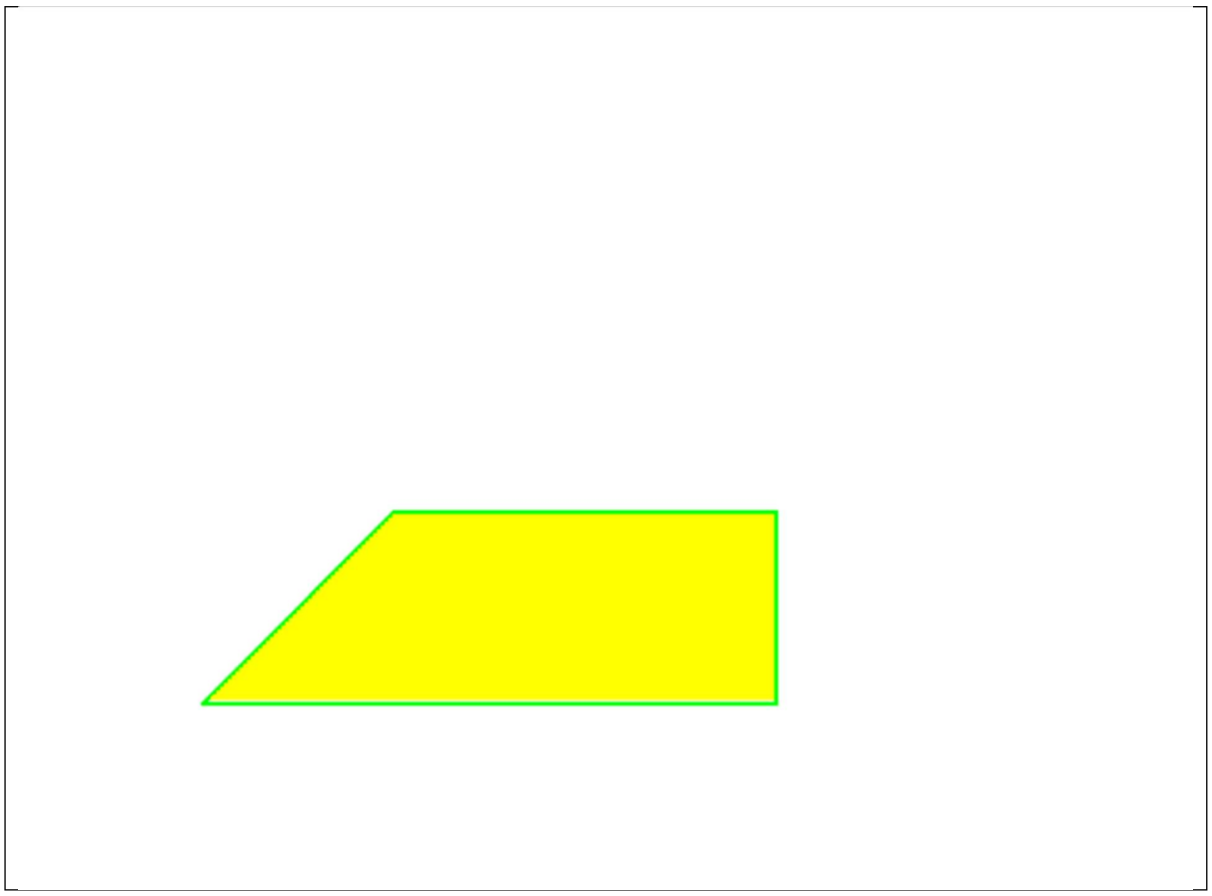
```
#include<GL/glut.h>

using namespace std;

int a, b, Dx, Dy, temp, interchange, e; //Specifying parameters for algorithms
int s1, s2;

void myInit() { //Initialization of viewing parameters
	glClearColor(1.0, 1.0, 1.0, 0.0);
	glMatrixMode(GL_PROJECTION);
	gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}

int sign(int a) { //Function to determine sign of a number
	if (a > 0) {
		return 1;
	}
	else if (a < 0) {
		return -1;
	}
}

void plot(int a, int b) { //Function to plot points
	glBegin(GL_POINTS);
	glVertex2i(a, b);
	glEnd();
	glFlush();
}

void DrawPolygon(int x, int y, int m, int n) { //Function to draw polygon using DDA algorithm
	a = x; b = y;
	Dx = abs(m - x);
	Dy = abs(n - y);
	s1 = sign(m - x);
	s2 = sign(n - y);
	if (Dy > Dx) {
		temp = Dx;
		Dx = Dy;
		Dy = temp;
		interchange = 1;
	}
	else {
		interchange = 0;
	}
	e = 2 * Dy - Dx;
	plot(a, b);
```

```
        for (int i = 1; i <= Dx; i++) {
                plot(a, b);
                while (e >= 0) {
                        if (interchange == 1)
                                a = a + s1;
                        else
                                b = b + s2;

                        e = e - 2 * Dx;
                }
                if (interchange == 1) {
                        b = b + s2;
                }
                else {
                        a = a + s1;
                }
                e = e + 2 * Dy;

        }
}

void putpixel(int c, int d, float* fillColor) { //Function to color polygon
        glColor3f(fillColor[0], fillColor[1], fillColor[2]); //Setting the color buffer
        glBegin(GL_POINTS);
        glVertex2i(c, d);
        glEnd();
        glFlush();
}

void boundaryFill4(int x, int y, float* fillColor, float* boundarycolor) { //Boundary fill
algorithm
        float color[3];
        glReadPixels(x, y, 1.0, 1.0, GL_RGB, GL_FLOAT, color);
        if ((color[0] != boundarycolor[0] || color[1] != boundarycolor[1] || color[2] !=
boundarycolor[2]) && (
                color[0] != fillColor[0] || color[1] != fillColor[1] || color[2] != fillColor[2])) {

                putpixel(x, y, fillColor);

                boundaryFill4(x + 1, y, fillColor, boundarycolor); //Recurssive calls
                boundaryFill4(x - 2, y, fillColor, boundarycolor);

                boundaryFill4(x, y + 2, fillColor, boundarycolor);
                boundaryFill4(x, y - 2, fillColor, boundarycolor);
        }
}
```

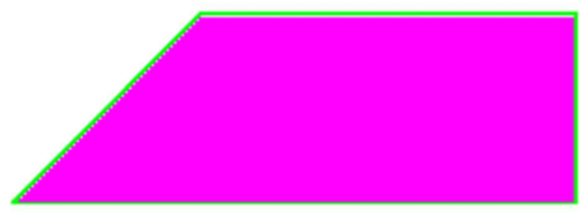
```
void Floodfill4(int x, int y, float* fillColor, float* interiorColor) { //Floodfill algorithm
        float color[3];
        glReadPixels(x, y, 1.0, 1.0, GL_RGB, GL_FLOAT, color);

        if (color[0] == interiorColor[0] && color[1] == interiorColor[1] && color[2] ==
interiorColor[2]) {

                putpixel(x, y, fillColor);

                Floodfill4(x + 1, y, fillColor, interiorColor); //Recurssive calls
                Floodfill4(x - 1, y, fillColor, interiorColor);
                Floodfill4(x, y + 1, fillColor, interiorColor);
                Floodfill4(x, y - 1, fillColor, interiorColor);
        }
        else {
                return;
        }
}

void myMouse(int button, int state, int x, int y) { //Function to select seed point
        y = 480 - y;
        if (button == GLUT_LEFT_BUTTON)
        {
                if (state == GLUT_DOWN)
                {
                        float boundaryCol[] = { 0,1,0 };

                        float fillcolor[] =  { 1,0,1  };
                        boundaryFill4(x, y, fillcolor, boundaryCol);
                }
        }

}

void myMouse1(int button, int state, int x, int y) {
        y = 480 - y;
        if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
                float interiorcolor[] = { 1,1,1 };
                float fillcolor[] = { 1,1,0 };
                Floodfill4(x, y, fillcolor, interiorcolor);
        }

}

void myDisplay() { //Draws polygon on the screen
```

```
	glLineWidth(3.0);
	glPointSize(2.0);
	glClear(GL_COLOR_BUFFER_BIT);
	glColor3f(0, 1, 0);
	DrawPolygon(100, 100, 200, 200);
	DrawPolygon(200, 200, 400, 200);
	DrawPolygon(400, 200, 400, 100);
	DrawPolygon(400, 100, 100, 100);
	glEnd();
	glFlush();
}

void myMenu(int item) { //Menu function
	if (item == 1) {
		glutMouseFunc(myMouse);
	}
	else if (item == 2) {
		glutMouseFunc(myMouse1);
	}

}

int main(int argc, char** argv) {
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB); //Initializing display mode for
window
	glutInitWindowSize(640, 480);
	glutInitWindowPosition(200, 200);
	glutCreateWindow("Polygon Filling");
	glutDisplayFunc(myDisplay);
	glutCreateMenu(myMenu);
	glutAttachMenu(GLUT_RIGHT_BUTTON);
	glutAddMenuEntry("Boundary fill",1);
	glutAddMenuEntry("Flood fill",2);
	myInit();
	glutMainLoop();
	return 0;
}
```

Output
Flood fill / Seed fill



Boundary fill



Lab. Based FAQ

1. Explain concave and convex polygons?
2. Explain different tests used for checking whether pixel is inside or outside the polygon.
3. Explain different ways of filling the polygon.
4. What is meant by Active Edge Table and Global Edge Table.

Practice Assignments :

Draw different objects such as house and apply following algorithms for filling the objects.

1. Scan line polygon filling algorithm
2. Flood fill polygon filling algorithm(Recursive and non recursive)
3. Boundary fill polygon filling algorithm(Recursive and non recursive)

Assignment No.: 6

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

Assignment No.: 6

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

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

Objective(s)	
1	Implement Cohen Sutherland lgorithm to clip any given line .
2	Implement Sutherland Hodgman algorithm to clip any given polygon.
3	Provide the vertices of the polygon to be clipped and pattern of clipping interactively

Theory

Line Clipping – Cohen Sutherland

In computer graphics, '***line clipping***' is the process of removing lines or portions of lines outside of an area of interest. Typically, any line or part thereof which is outside of the viewing area is removed.
The Cohen–Sutherland algorithm is a computer graphics algorithm used for line clipping. The algorithm divides a two-dimensional space into 9 regions (or a three-dimensional space into 27 regions), and then efficiently determines the lines and portions of lines that are visible in the center region of interest (the viewport).
The algorithm was developed in 1967 during flight simulator work by Danny Cohen and Ivan Sutherland

The design stage includes, excludes or partially includes the line based on where:

- Both endpoints are in the viewport region (bitwise OR of endpoints == 0): trivial accept.
- Both endpoints share at least one non-visible region which implies that the line does not cross the visible region. (bitwise AND of endpoints != 0): trivial reject.
- Both endpoints are in different regions: In case of this nontrivial situation the algorithm finds one of the two points that is outside the viewport region (there will be at least one point outside). The intersection of the outpoint and extended viewport border is then calculated (i.e. with the parametric equation for the line) and this new point replaces the outpoint. The algorithm repeats until a trivial accept or reject occurs.

The numbers in the figure below are called outcodes. The outcode is computed for each of the two points in the line. The outcode will have four bits for two-dimensional clipping, or six bits in the three-dimensional case. The first bit is set to 1 if the point is above the viewport. The bits in the 2D outcode represent: Top, Bottom, Right, Left. For example the outcode 1010 represents a point that is top-right of the viewport. Note that the outcodes for endpoints **must** be recalculated on each iteration after the clipping occurs.

1001	1000	1010
0001	0000	0010
0101	0100	0110

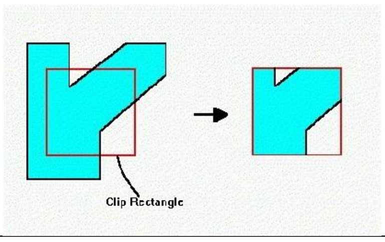
Sutherland Hodgman Polygon Clipping

It is used for clipping polygons. It works by extending each line of the convex *clip polygon* in turn and selecting only vertices from the *subject polygon* those are on the visible side.

An algorithm that clips a polygon must deal with many different cases. The case is particularly noteworthy in that the concave polygon is clipped into two separate polygons. All in all, the task of clipping seems rather complex. Each edge of the polygon must be tested against each edge of the clip rectangle; new edges must be added, and existing edges must be discarded, retained, or divided. The algorithm begins with an input list of all vertices in the subject polygon. Next, one side of the clip polygon is extended infinitely in both directions, and the path of the subject polygon is traversed. Vertices from the input list are inserted into an output list if they lie on the visible side of the extended clip polygon line, and new vertices are added to the output list where the subject polygon path crosses the extended clip polygon line.

This process is repeated iteratively for each clip polygon side, using the output list from one stage as the input list for the next. Once all sides of the clip polygon have been processed, the final generated list of vertices defines a new single polygon that is entirely visible. Note that if the subject polygon was concave at vertices outside the clipping polygon, the new polygon may have coincident (i.e., overlapping) edges – this is acceptable for rendering, but not for other applications such as computing shadows.

The following example illustrate a simple case of polygon clipping



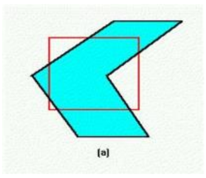
Sutherland and Hodgman's polygon-clipping algorithm uses a divide-and-conquer strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to clip a polygon against a single infinite clip edge. Four clip edges, each defining one boundary of the clip rectangle, successively clip a polygon against a clip rectangle.

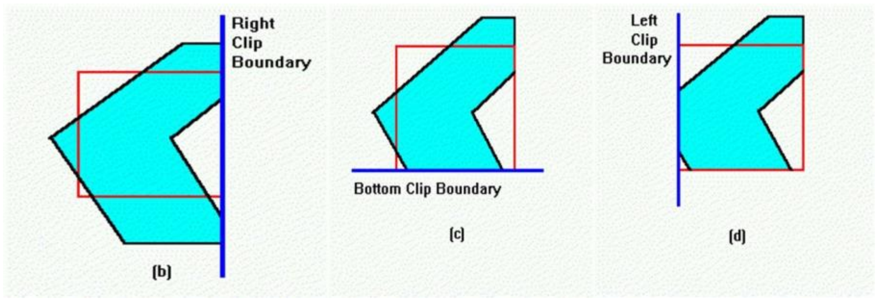
Note the difference between this strategy for a polygon and the Cohen-Sutherland algorithm for clipping a line: The polygon clipper clips against four edges in succession, whereas the line clipper tests the outcode to see which edge is crossed, and clips only when necessary.

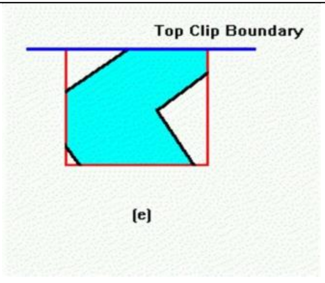
Steps of Sutherland-Hodgman's polygon-clipping algorithm

- Polygons can be clipped against each edge of the window one at a time. Windows/edge intersections, if any, are easy to find since the X or Y coordinates are already known.
- Vertices which are kept after clipping against one window edge are saved for clipping against the remaining edges.
- Note that the number of vertices usually changes and will often increases.
- We are using the Divide and Conquer approach
- After clipped by the right and bottom clip boundaries.

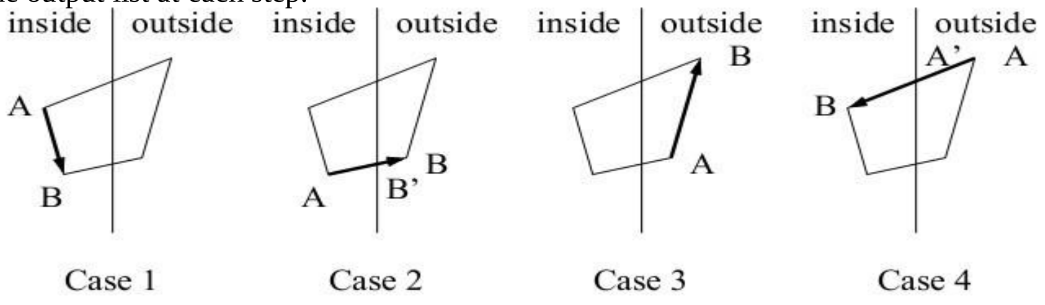
The original polygon and the clip rectangle.







Clipping polygons would seem to be quite complex. A single polygon can actually be split into multiple polygons .The Sutherland-Hodgman algorithm clips a polygon against all edges of the clipping region in turn. The algorithm steps from vertex to vertex, adding 0, 1, or 2 vertices to the output list at each step.

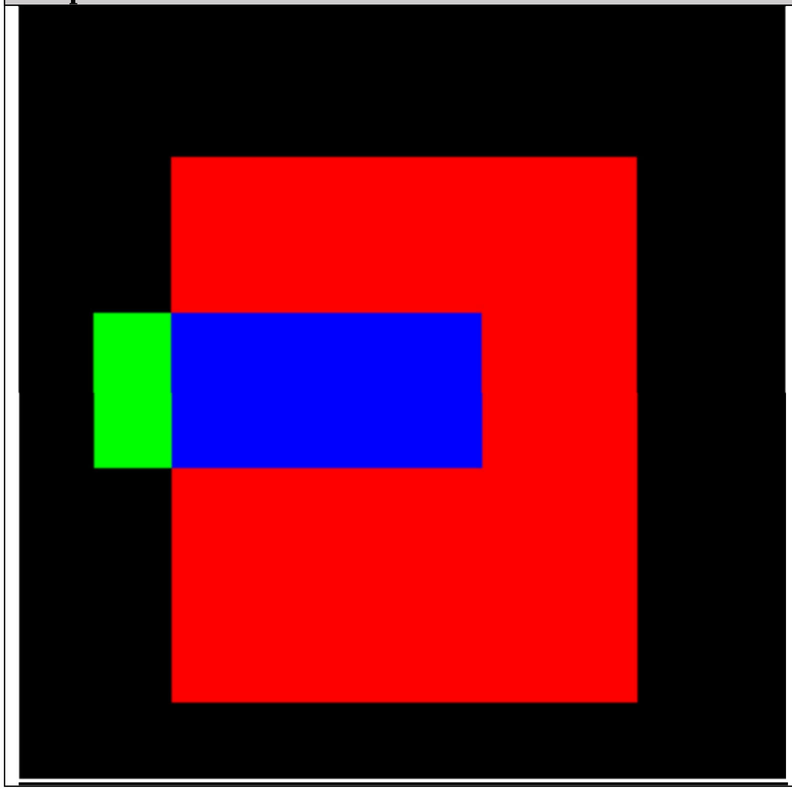


Input

```
#include <windows.h>
#include <gl/glut.h>

struct Point{
   float x,y;
} w[4],oVer[4];
int Nout;

void drawPoly(Point p[],int n){
   glBegin(GL_POLYGON);
   for(int i=0;i<n;i++)
      glVertex2f(p[i].x,p[i].y);
   glEnd();
}

bool insideVer(Point p){
      if((p.x>=w[0].x)&&(p.x<=w[2].x))
         if((p.y>=w[0].y)&&(p.y<=w[2].y))
            return true;
      return false;
```

```
}

void addVer(Point p){
   oVer[Nout]=p;
   Nout=Nout+1;
}

Point getInterSect(Point s,Point p,int edge){
   Point in;
   float m;
   if(w[edge].x==w[(edge+1)%4].x){ //Vertical Line
      m=(p.y-s.y)/(p.x-s.x);
      in.x=w[edge].x;
      in.y=in.x*m+s.y;
   }
   else{//Horizontal Line
      m=(p.y-s.y)/(p.x-s.x);
      in.y=w[edge].y;
      in.x=(in.y-s.y)/m;
   }
   return in;
}

void clipAndDraw(Point inVer[],int Nin){
   Point s,p,interSec;
   for(int i=0;i<4;i++)
   {
      Nout=0;
      s=inVer[Nin-1];
      for(int j=0;j<Nin;j++)
      {
         p=inVer[j];
         if(insideVer(p)==true){
            if(insideVer(s)==true){
               addVer(p);
            }
            else{
               interSec=getInterSect(s,p,i);
               addVer(interSec);
               addVer(p);
            }
         }
         else{
            if(insideVer(s)==true){
               interSec=getInterSect(s,p,i);
               addVer(interSec);
            }
          }
          s=p;
        }
        inVer=oVer;
        Nin=Nout;
    }
    drawPoly(oVer,4);
}

void init(){
    glClearColor(0.0f,0.0f,0.0f,0.0f);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0,100.0,0.0,100.0,0.0,100.0);
    glClear(GL_COLOR_BUFFER_BIT);
    w[0].x =20,w[0].y=10;
    w[1].x =20,w[1].y=80;
    w[2].x =80,w[2].y=80;
    w[3].x =80,w[3].y=10;
}
void display(void){
    Point inVer[4];
    init();
    // As Window for Clipping
    glColor3f(1.0f,0.0f,0.0f);
    drawPoly(w,4);
    // As Rect
    glColor3f(0.0f,1.0f,0.0f);
    inVer[0].x =10,inVer[0].y=40;
    inVer[1].x =10,inVer[1].y=60;
    inVer[2].x =60,inVer[2].y=60;
    inVer[3].x =60,inVer[3].y=40;
    drawPoly(inVer,4);
    // As Rect
    glColor3f(0.0f,0.0f,1.0f);
    clipAndDraw(inVer,4);
    // Print
    glFlush();
}

int main(int argc,char *argv[]){
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(400,400);
    glutInitWindowPosition(100,100);
    glutCreateWindow("Polygon Clipping!");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Output



Lab. Based FAQ

1. What is clipping?
2. What do you mean by interior and exterior clipping?
3. Explain how exterior clipping is useful in multiple window environments
4. The Sutherland-Hodgman algorithm can be used to clip lines against a non rectangular boundary. What uses might this have? What modifications to the algorithm would be necessary? What restrictions would apply to the shape of clipping region?
5. Explain why Sutherland-Hodgman algorithm works only for convex clipping regions?

Assignment No.: 6

Implement following 2D transformations on the object with respect to axis

i) Scaling ii) Rotation about arbitrary point iii) Reflection

Assignment No. : 6

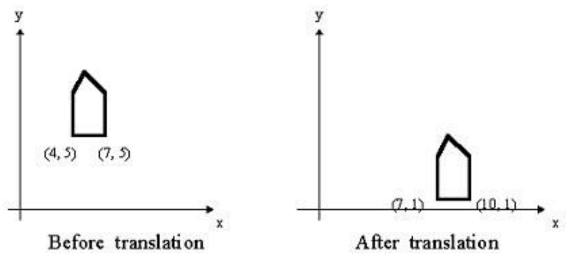
Implement following 2D transformations on the object with respect to axis
i) Scaling ii) Rotation about arbitrary point iii) Reflection

Aim
Implement following 2D transformations on the object with respect to axis i) Scaling ii) Rotation about arbitrary point iii) Reflection

Objective(s)	
1	2 D Homogeneous coordinate system
2	Understand and Implement 2D transformations in Laboratory.

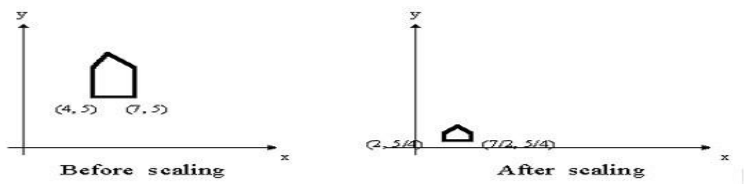
Theory

1. **Translation:** Translation is defined as moving the object from one position to another *position along straight line path.*



We can move the objects based on translation distances along x and y axis. tx denotes translation distance along x-axis and ty denotes translation distance along y axis.
Translation Distance: It is nothing but by how much units we should shift the object from one location to another along x, y-axis.
Consider (x,y) are old coordinates of a point. Then the new coordinates of that same point (x’,y’) can be obtained as follows:
X’=x+tx
Y’=y+ty

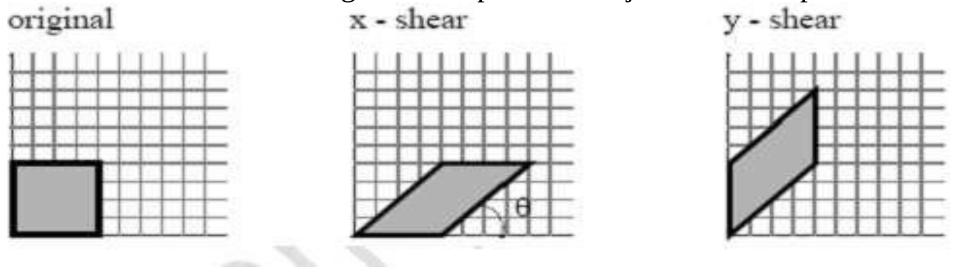
2. **Scaling:** scaling refers to changing the size of the object either by increasing or decreasing. We will increase or decrease the size of the object based on scaling factors along x and y -axis. If (x, y) are old coordinates of object, then new coordinates of object after applying scaling transformation are obtained as:
x’=x*sx
y’=y*sy.
sx and sy are scaling factors along x-axis and y-axis. we express the above equations in matrix form as:



3. Rotation : A rotation repositions all points in an object along a circular path in the plane centered at the
pivot point. We rotate an object by an angle theta
New coordinates after rotation depend on both x and y
$x' = x\cos\theta - y\sin\theta$
$y' = x\sin\theta + y\cos\theta$
or in matrix form:
$P' = R \bullet P$,
R-rotation matrix.

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

4. Shear:
1. Shear is the translation along an axis by an amount that increases linearly with another axis (Y). It produces shape distortions as if objects were composed of layers that are caused to slide over each other.
2. Shear transformations are very useful in creating italic letters and slanted letters from regular letters.
3. Shear transformation changes the shape of the object to a slant position.



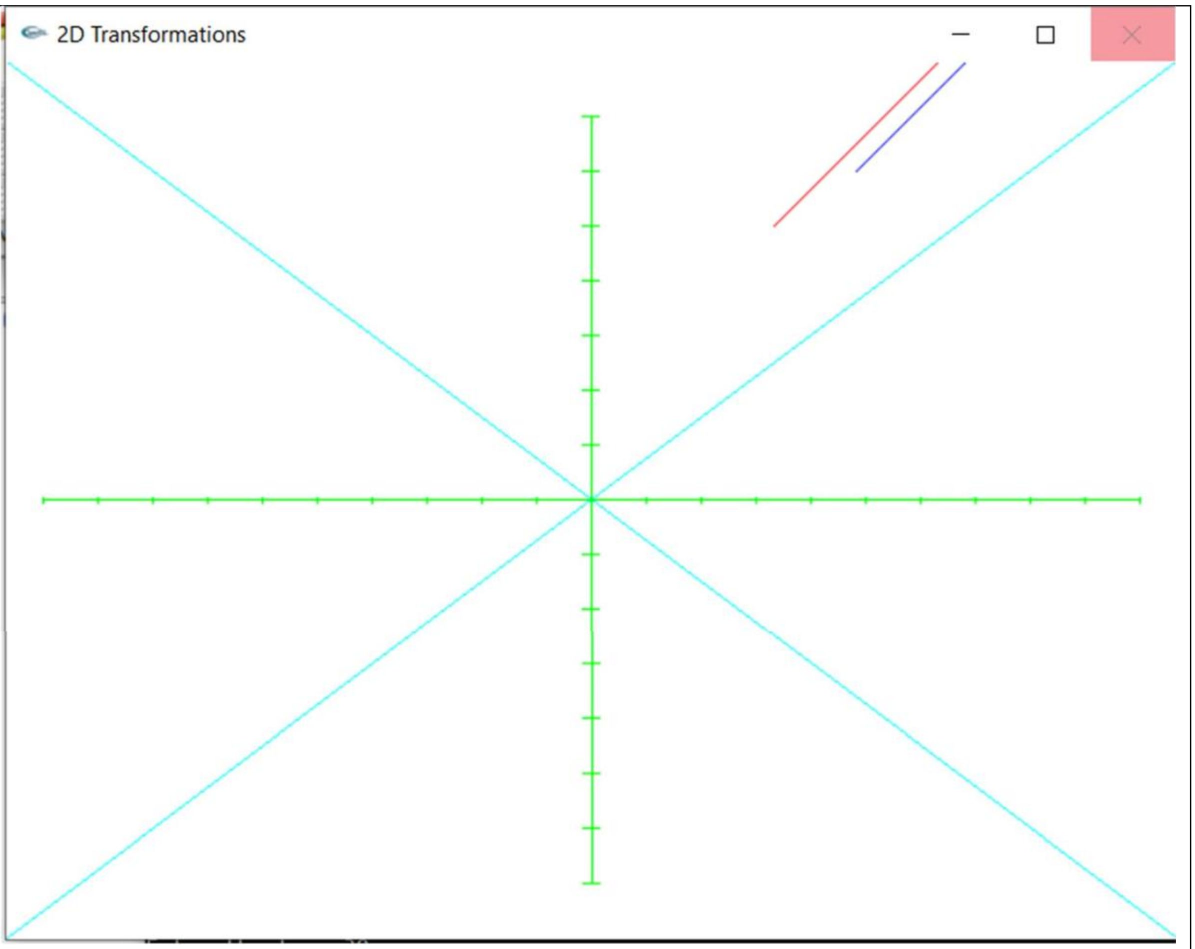
4. Shear transformation is of 2 types:
a. X-shear: changing x-coordinate value and keeping y constant
$x'=x+shx*y$
$y'=y$
b. Y-shear: changing y coordinates value and keeping x constant
$x'=x$
$y'=y+shy*x$
shx and shy are shear factors along x and y-axis

Input
Enter an object of any shape for transformation

Output

Translation

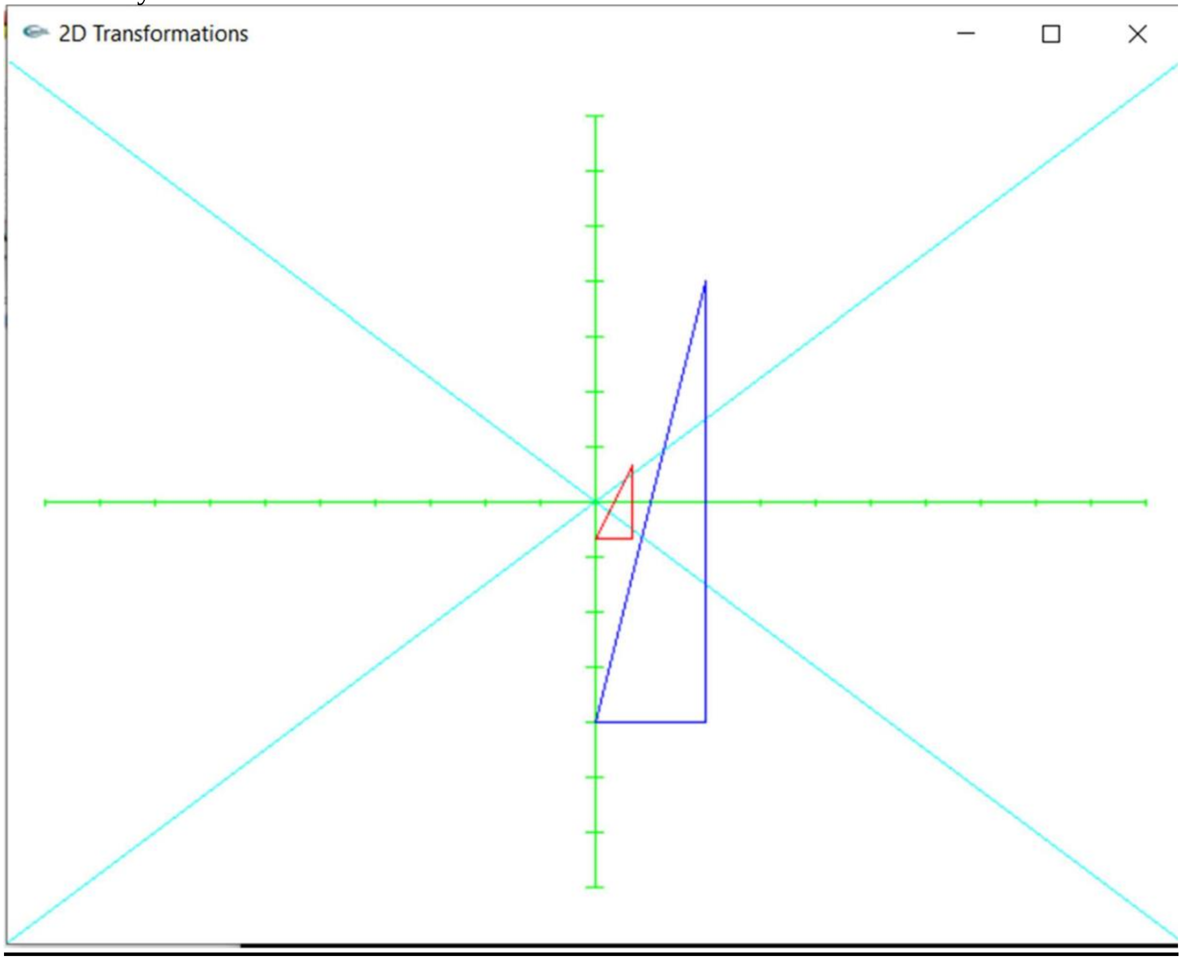
```
***********  2D  Transformations  *********

Enter the no of vertices of polygon : 2
Enter the coordinate :
x1,y1 : 100
150

x2,y2 : 200
250

***********MENU**********

1) Translation
2) Scaling
3) Rotation
4) Rotation of arbitary
5) Shearing
6) Reflection
Enter your choice : 1

The polygon before translation
Enter the tx : 45

Enter the ty : 30
```



Scaling

```
*********** 2D Transformations *********

Enter the no of vertices of polygon : 3
Enter the coordinate :
x1,y1 : 0
-20

x2,y2 : 20
-20

x3,y3 : 20
20

***********MENU**********

1) Translation
2) Scaling
```

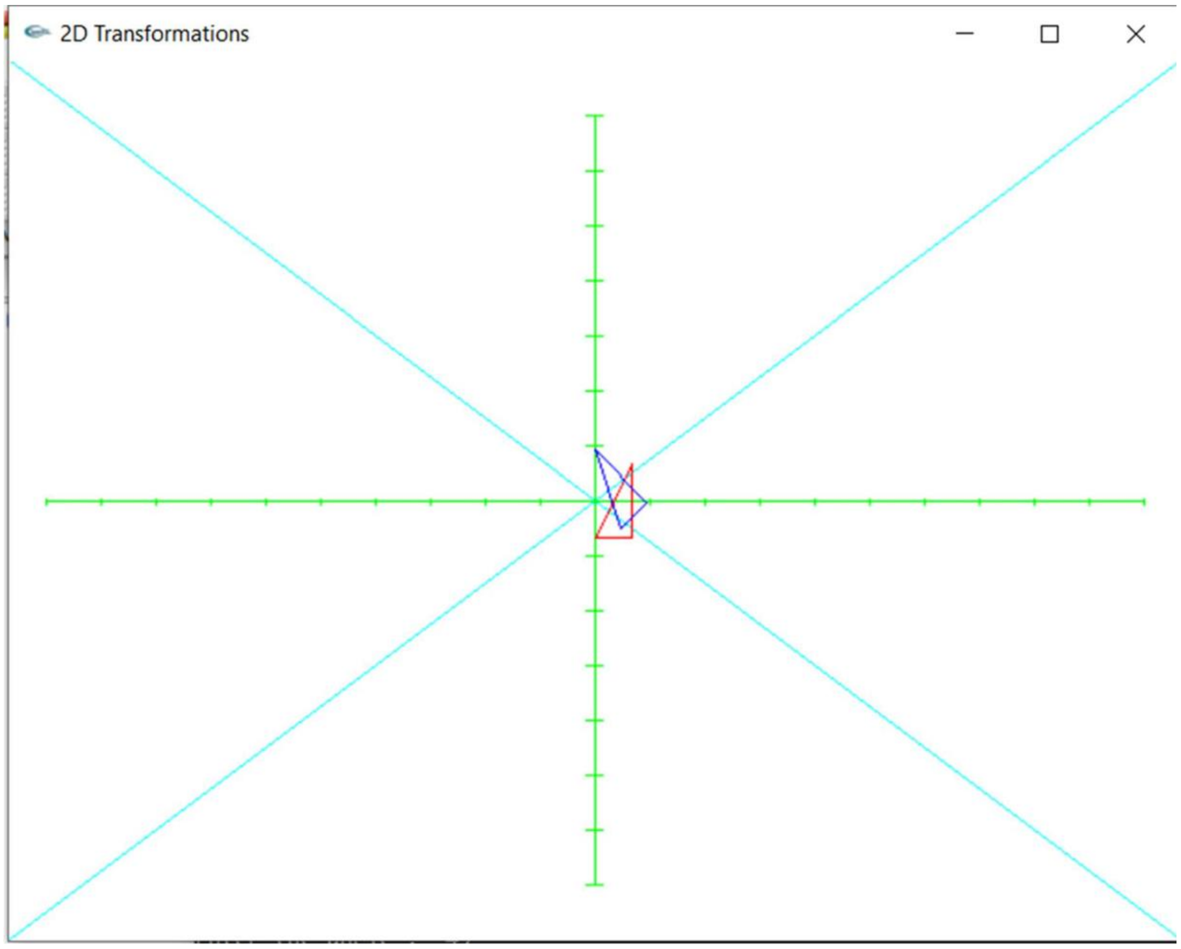
```
3) Rotation
4) Rotation of arbitary
5) Shearing
6) Reflection
Enter your choice : 2

The polygon before scaling
Enter the sx3

Enter the sy6
```



Rotation

```
*********** 2D Transformations *********

Enter the no of vertices of polygon : 3
Enter the coordinate :
x1,y1 : 0
-20

x2,y2 : 20
-20

x3,y3 : 20
20

***********MENU**********

1) Translation
2) Scaling
3) Rotation
4) Rotation of arbitary
5) Shearing
6) Reflection
Enter your choice : 3

The polygon befor rotation
Enter the angle : 45

Press 1 for anticlockwise and 2 for clockwise : 1
```

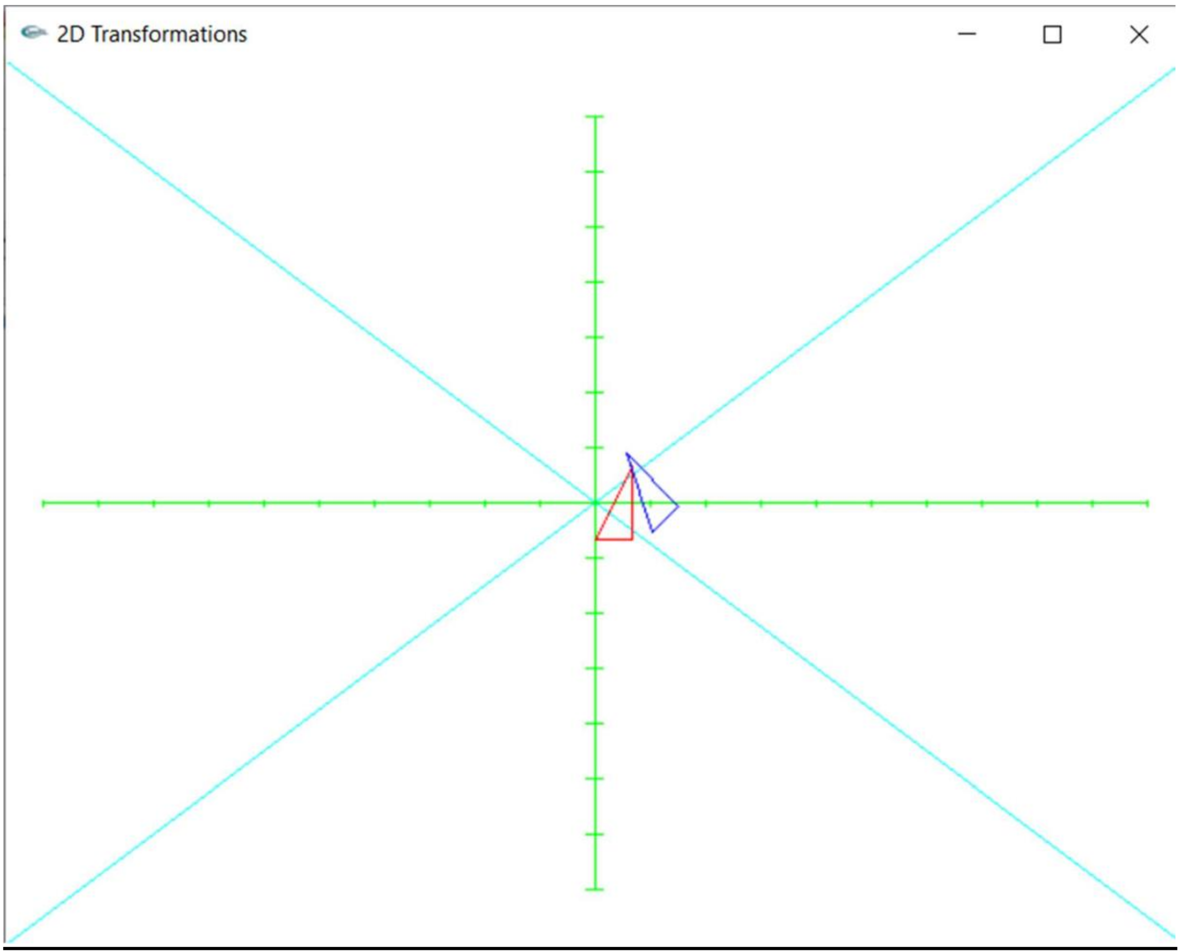
a



Rotation of arbitary

```
*********** 2D Transformations *********

Enter the no of vertices of polygon : 3
Enter the coordinate :
x1,y1 : 0
-20

x2,y2 : 20
-20

x3,y3 : 20
20

***********MENU**********
```

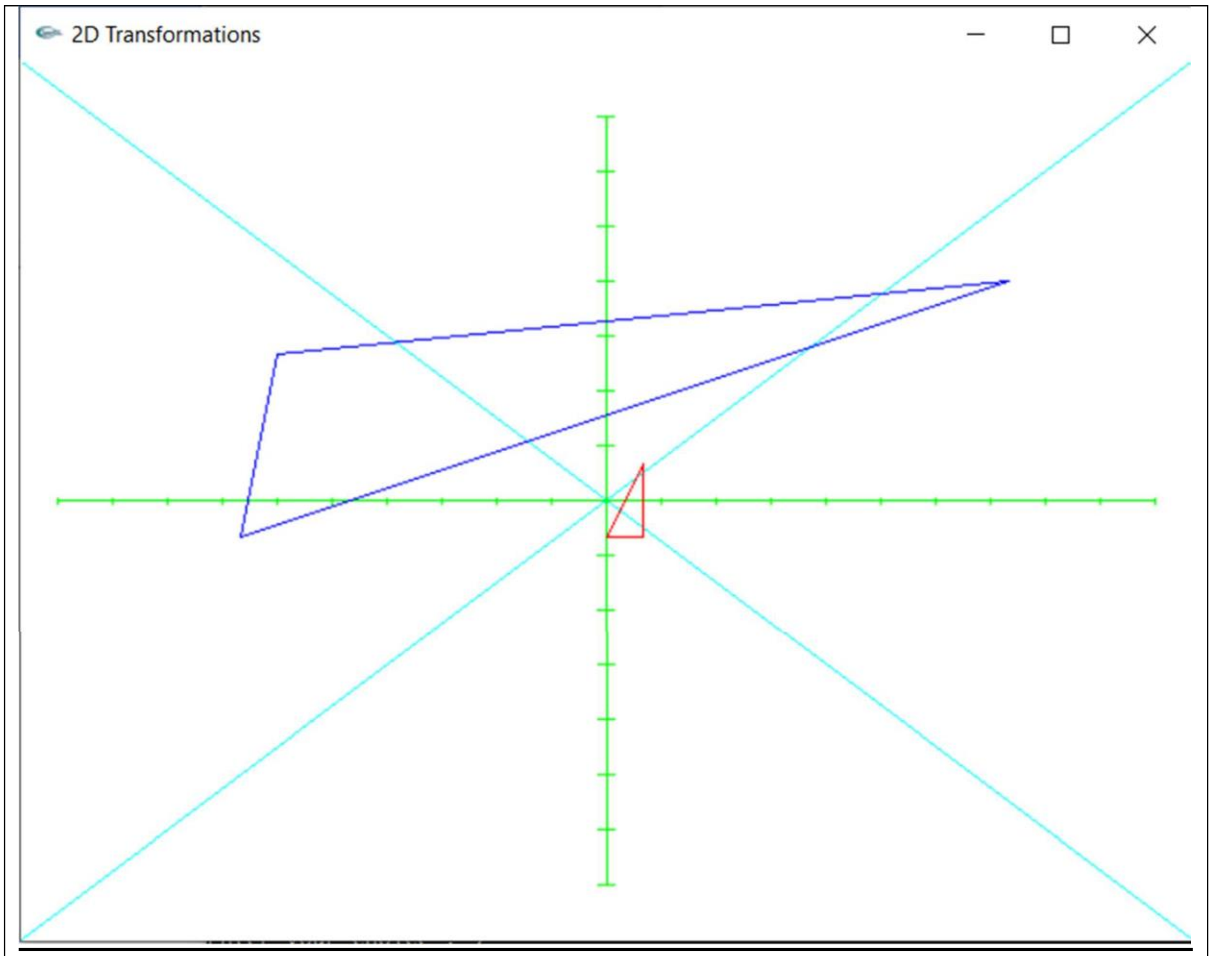
```
1) Translation
2) Scaling
3) Rotation
4) Rotation of arbitary
5) Shearing
6) Reflection
Enter your choice : 4

The polygon befor rotation
Enter the angle : 45

Press 1 for anticlockwise and 2 for clockwise : 1
Enter the x and y coordinate : 10
20
```



Shearing

```
***********  2D  Transformations   *********
```

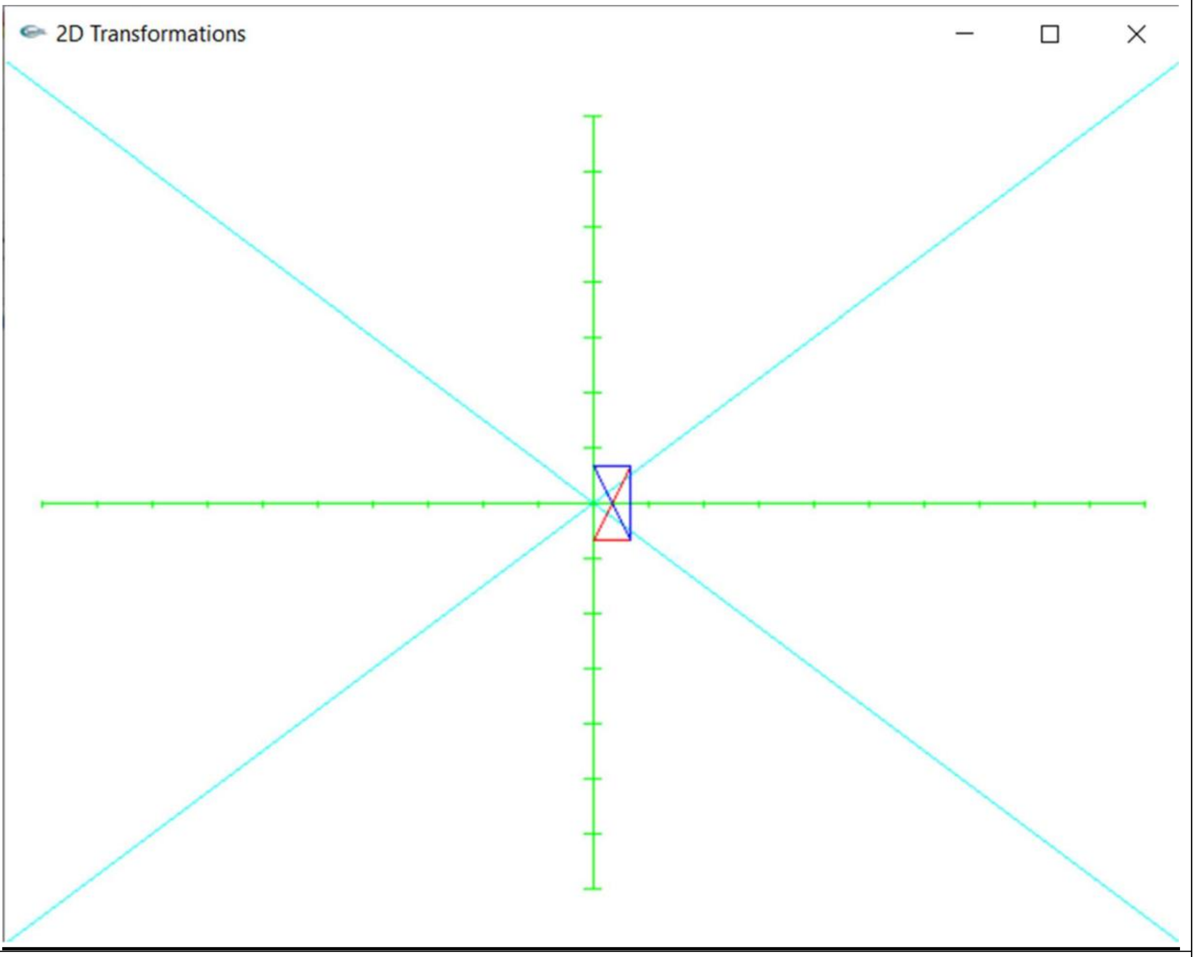
```
Enter the no of vertices of polygon : 3
Enter the coordinate :
x1,y1 : 0
-20

x2,y2 : 20
-20

x3,y3 : 20
20

***********MENU**********

1) Translation
2) Scaling
3) Rotation
4) Rotation of arbitary
5) Shearing
6) Reflection
Enter your choice : 5

The polygon before Shearing
Enter the Shx : 5

Enter the Shy : 10
```



Reflection

```
*********** 2D Transformations *********

Enter the no of vertices of polygon : 3
Enter the coordinate :
x1,y1 : 0
-20

x2,y2 : 20
-20

x3,y3 : 20
20

***********MENU**********

1) Translation
```

```
2) Scaling
3) Rotation
4) Rotation of arbitary
5) Shearing
6) Reflection
Enter your choice : 6

The polygon before Reflection
--------------------------------
        1. Against X-axis
        2. Against Y-axis
        3. Against Origin
        4. X = Y
        5. X = -Y
        Enter you Choice : 1
```



Lab. Based FAQ
1. What is homogeneous co-ordinate system

Practice Assignment
1. Draw a house and perform all the above transformations.

Assignment No.: 7

Generate fractal patterns using i) Bezier ii) Koch Curve.

Assignment No. : 7

Generate fractal patterns using i) Bezier ii) Koch Curve.

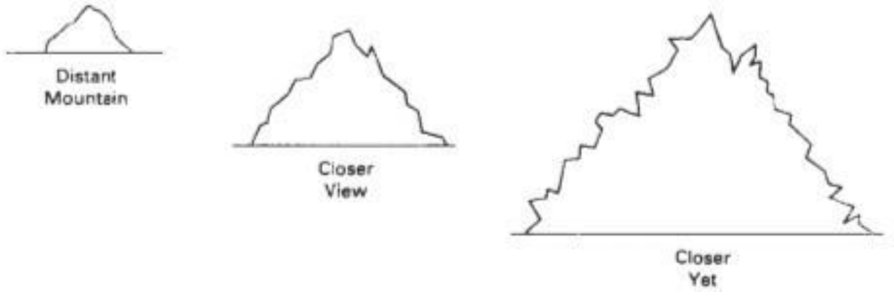
Aim
Generate fractal patterns using i) Bezier ii) Koch Curve.

Objective(s)	
1	Understand different types of Curve. Implement different objects using Koch curve fractal pattern

Theory

Fractals are geometric objects. Many real-world objects like ferns are shaped like fractals. Fractals are formed by iterations. Fractals are self-similar. In computer graphics, we use fractal functions to create complex object. The object representations use Euclidean-geometry methods; that is, object shapes were described with equations. These methods are adequate for describing manufactured objects: those that have smooth surfaces and regular shapes. But natural objects, such as mountains and clouds, have irregular or fragmented features, and Euclidean methods do not realistically model these objects. Natural objects can be realistically described with fractal-geometry methods, where procedures rather than equations are used to model objects.

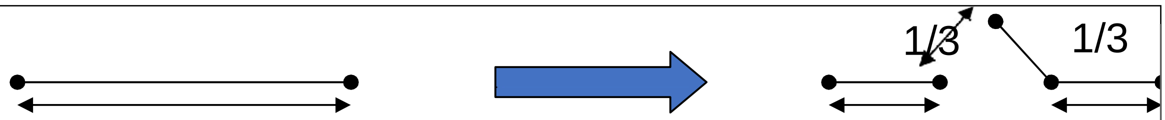
In computer graphics, fractal methods are used to generate displays of natural objects and visualizations. The self-similarity properties of an object can take different forms, depending on the choice of fractal representation. In computer graphics, we use fractal functions to create complex objects

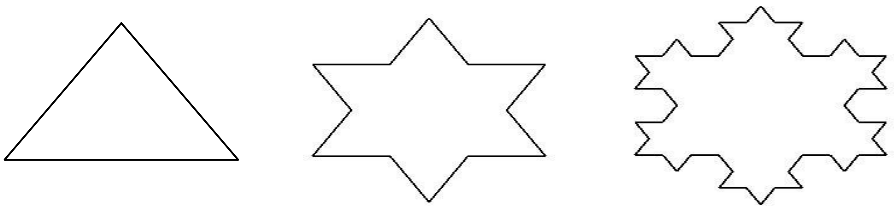


A mountain outlined against the sky continues to have the same jagged shape as we view it from a closer and closer. We can describe the amount of variation in the object detail with a number called the fractal dimension.

Examples: In graphics applications, fractal representations are used to model terrain, clouds, water, trees and other plants, feathers, fur, and various surface textures, and just to make pretty patterns. In other disciplines, fractal patterns have been found in the distribution of stars, river islands, and moon craters; in rain fields; in stock market variations; in music; in traffic flow; in urban property utilization; and in the boundaries of convergence regions for numerical- analysis techniques

Koch Fractals (Snowflakes)





- **Add Some Randomness:**
- The fractals we've produced so far seem to be very regular and "artificial".
- To create some realism and variability, simply change the angles slightly sometimes based on a random number generator.
- For example, you can curve some of the ferns to one side.
- For example, you can also vary the lengths of the branches and the branching factor.

Terrain (Random Mid-point Displacement):

- Given the heights of two end-points, generate a height at the mid-point.
- Suppose that the two end-points are a and b. Suppose the height is in the y direction, such that the height at a is y(a), and the height at b is y(b).
- Then, the height at the mid-point will be:

$y_{mid} = (y(a)+y(b))/2 + r$, where

r is the random offset

- This is how to generate the random offset r:

$r = sr_g|b-a|$, where

s is a user-selected "roughness" factor, and

r_g is a Gaussian random variable with mean 0 and variance 1

Input

Select the object, Select the pattern, How many times the pattern is to be repeated.

Bezier Curve :

Part 1: Rendering a Quadratic Bézier Curve

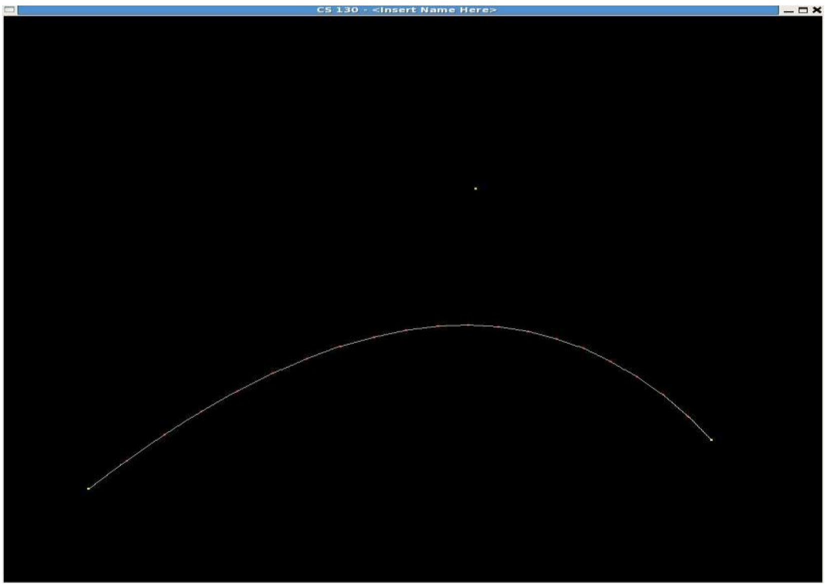
Consider the parametric equation for the straight line:

$$L=(1-t)P_0+tP_1$$

This is a Bézier curve with two control points and a degree of 1 (linear). A quadratic Bézier curve has three control points and a degree of 2. This is a curve influenced by the control points P0, P1, and P2. Or another phrase, it is influence by vectors from P0 to P1 and P1 to P2. The parametric equation can be generated by recursively substituting the generalized parametric equation for the straight line:

$$L=(1-t)P_{01}+tP_{12}$$
to
$$L=(1-t)[(1-t)P_0+tP_1]+t[(1-t)P_1+tP_2]$$

Once simplified, you can substitute t values to locate the exact point on the Bézier curve. We will render lines using the DDA algorithm and incrementing by small dt from [0,1]. Now, write a function that takes in three control points and renders the Bézier curve using line rendering. Provide a mouse event handler where three mouse clicks (in order) will generate the Bézier curve. Be aware that part 2 will be a modification of part 1 to render a Bézier curve with many control points. An example of a Bézier curve is shown in the image below where the number of lines used to render the curve is 20.



Part 2: Generalization for Bézier Curves

Working through a formula for a cubic Bézier curve (degree n = 3), we can find that the coefficients for each term are binomial coefficients (1 3 3 1). Each control point, P, is given a specific weight. At a certain t, the weighted points Pi are aggregated into a single point on the Bézier curve. The generalization for a point on the Bézier curve for t from [0,1] is below:

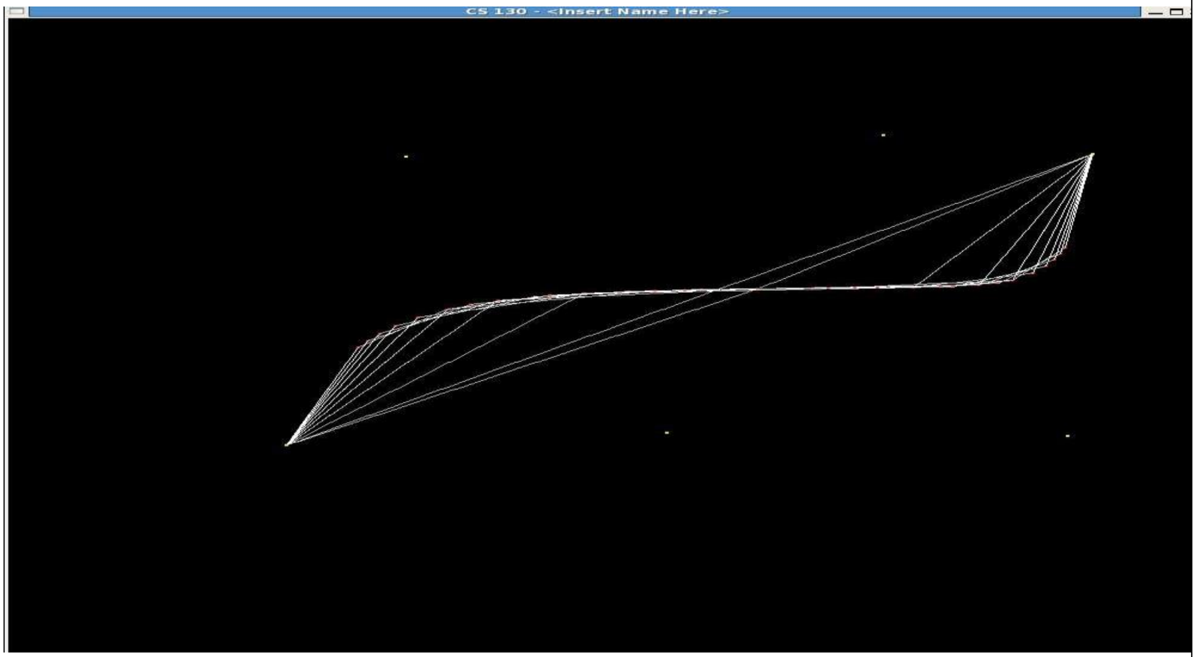
$$B(t)=\sum_{i=0}^{n}\binom{n}{i}(1-t)^{n-i}t^{i}P_i$$

Pay close attention here - the degree, n, is one less than the number of control points!

Modify your code such that your Bézier curve can take in a list of control points and the maximum number of line segments used to render the curve. Change your mouse handler function to continuously add mouse coordinates to a list so that your function can render them.

Hint: First write a function to calculate any specific point on curve B for any t, given all the control points (clicks) set so far.

Show the TA that your Bézier curve can be rendered with more than 3 control points. An example of what multiple calls to your Bézier render function can do is shown below varying with the number of line segments and 6 control points:



Input-

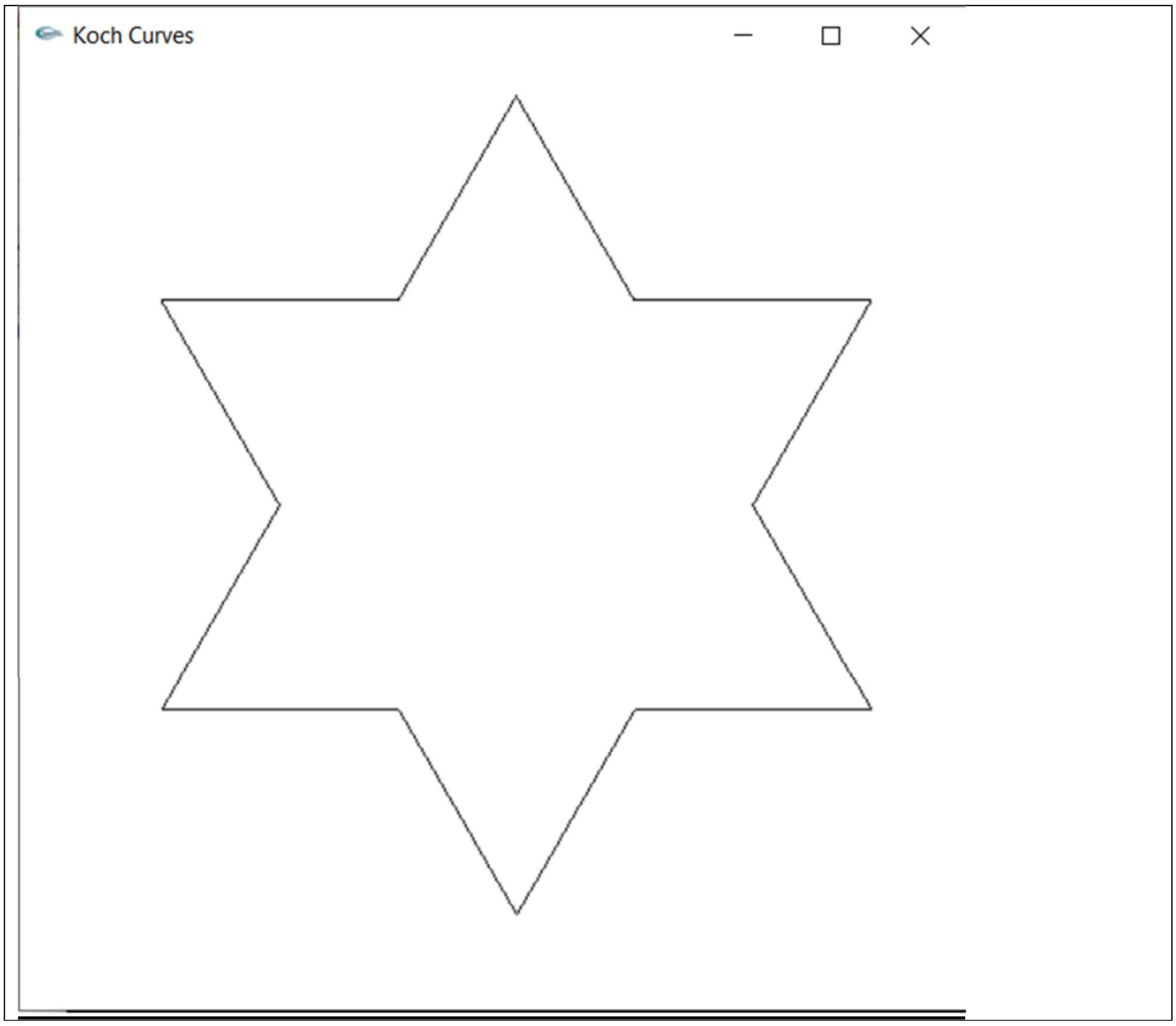
```
#include<windows.h>
#include <GL/glut.h>

#include <math.h>


GLfloat oldx=-0.7,oldy=0.5;


void drawkoch(GLfloat dir,GLfloat len,GLint iter)
{
```

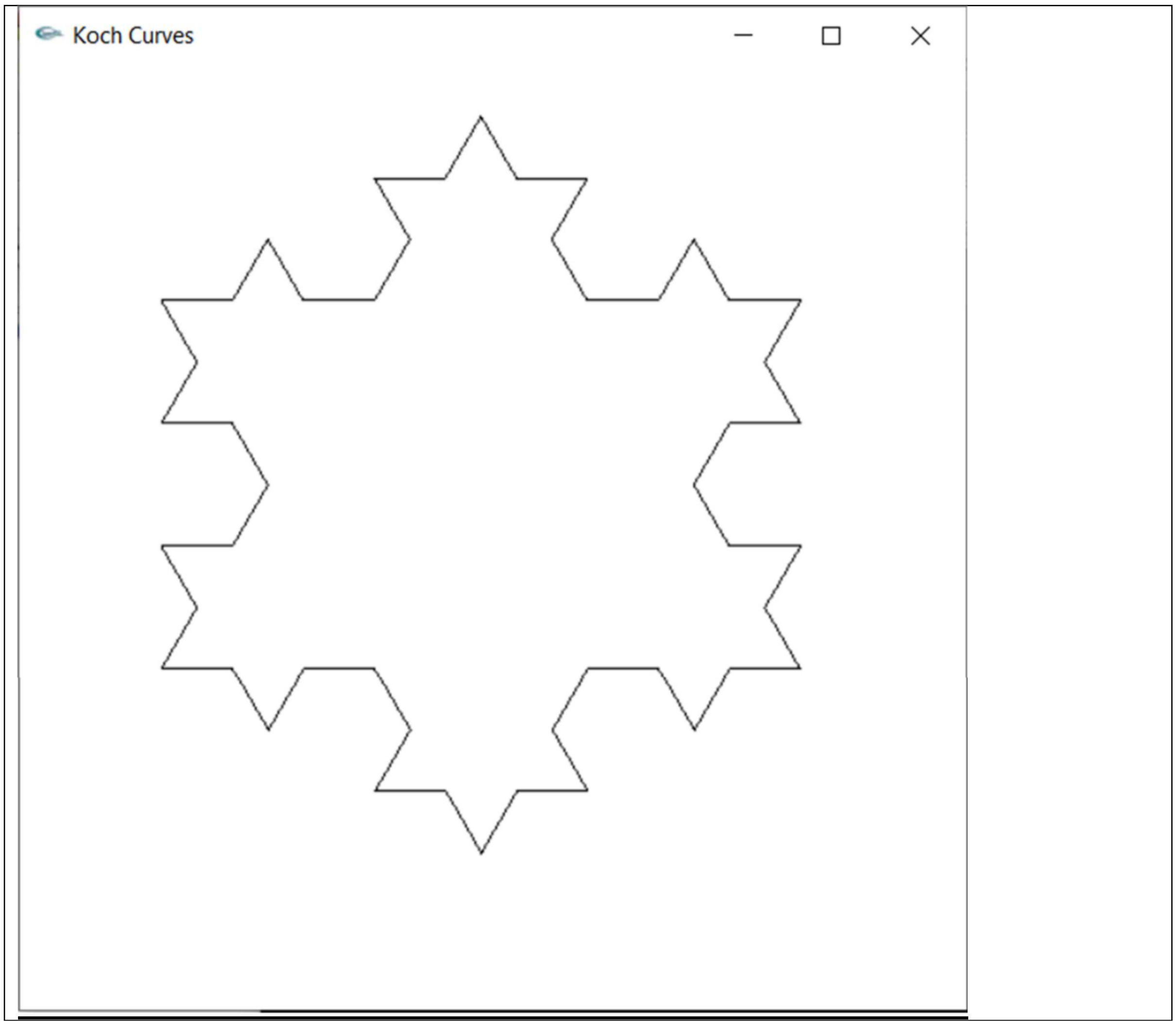
```
GLdouble dirRad = 0.0174533 * dir ;

GLfloat newX = oldx + len * cos(dirRad);

            GLfloat newY = oldy + len * sin(dirRad);

            if (iter==0)
         {

            glVertex2f(oldx, oldy);

            glVertex2f(newX, newY);

            oldx = newX;

            oldy = newY;

      }

else
    {

            iter--;
            //draw the four parts of the side _/\_

              drawkoch(dir, len, iter);

             dir += 60.0;

           drawkoch(dir, len, iter);

           dir -= 120.0;

          drawkoch(dir, len, iter);

         dir += 60.0;

 drawkoch(dir, len, iter);

}

}

void display()
```

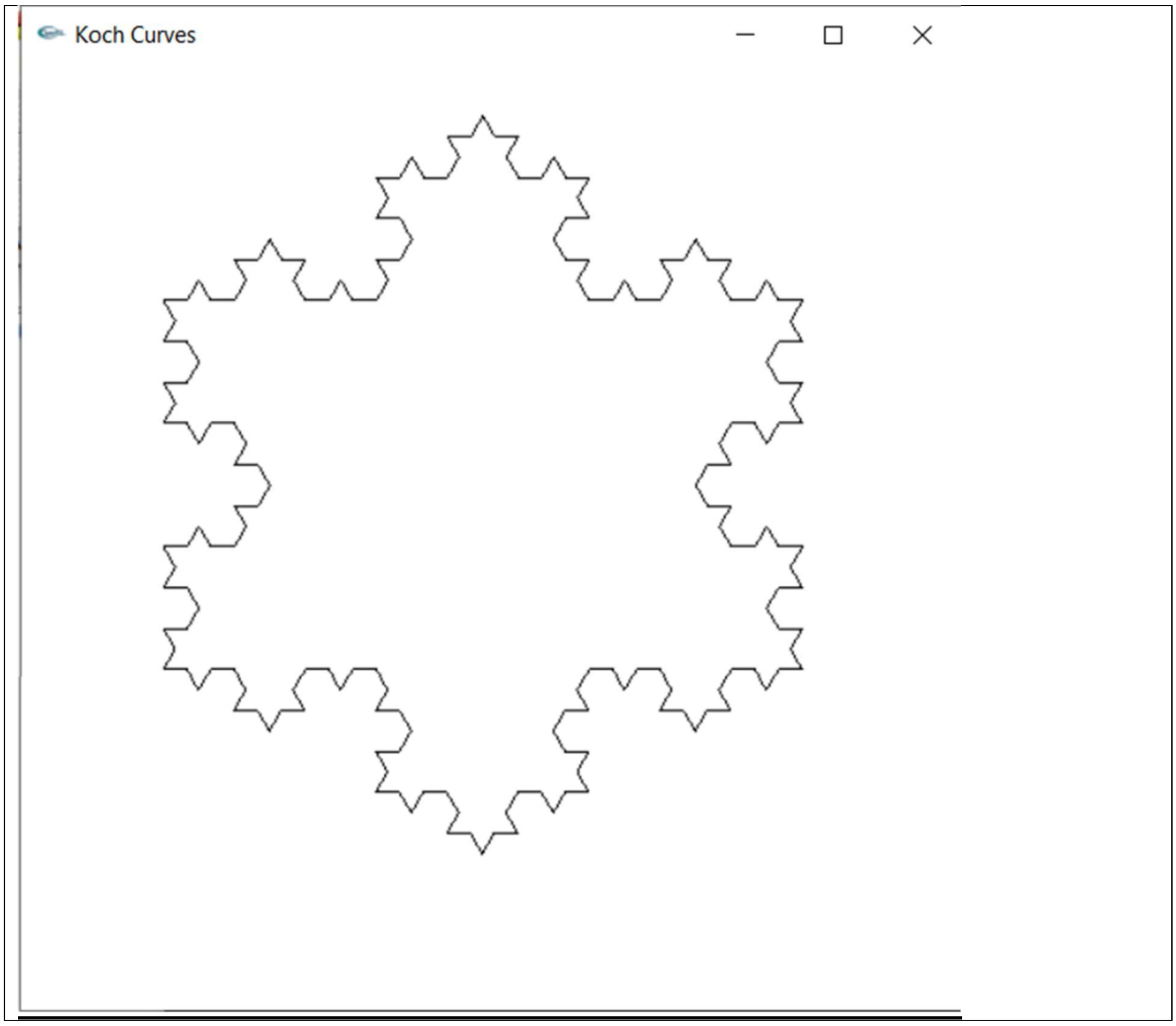
```
{
        glClearColor(1.0,1.0,1.0,0);
        glColor3f(0.0, 0.0, 0.0);
 glClear( GL_COLOR_BUFFER_BIT );
 glBegin(GL_LINES);

            drawkoch(0.0,0.5,1);
      drawkoch(-120.0, 0.5, 1);
  drawkoch(120.0,0.5,1);
 /*
 drawkoch(0.0,0.15,2);
 drawkoch(-120.0, 0.15, 2);
drawkoch(120.0,0.15,2);

/*
 drawkoch(0.0,0.05,3);
 drawkoch(-120.0, 0.05, 3);
 drawkoch(120.0,0.05,3);
*/
 glEnd();
 glFlush();
}

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

```
glutInit(&argc,argv);

glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);

glutInitWindowSize(500,500);

glutInitWindowPosition(0,0);

glutCreateWindow("Koch Curves");

glutDisplayFunc(display);

glutMainLoop();


}
```

Output


Koch Curves


Koch Curves



Lab. Based FAQ
Give examples of Koch curve?

Lab. Based Assignments
Draw Different patterns using Koach curves

Assignment No.: 8

Implement animation principles for any object.

Assignment No.: 8

Animation: Implement any one of the following animation assignments,

i) Clock with pendulum

ii) National Flag hoisting

iii) Vehicle/boat locomotion

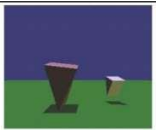
iv) Falling Water drop into the water and generated waves after impact

v) Kaleidoscope views generation (at least 3 colorful patterns)

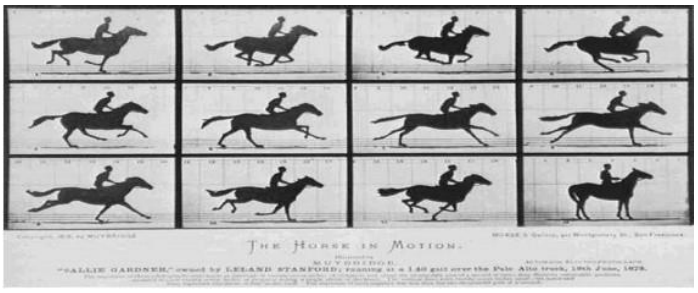
Aim
Implement animation principles for any object.

Objective(s)	
1	To learn different types of animation
2	To learn OpenGL function which support for animation

Theory
Motion can bring the simplest of characters to life. Even simple polygonal shapes can convey a number of human qualities when animated: identity, character, gender, mood, intention, emotion, and so on. Very simple



A movie is a sequence of frames of still images. For video, the frame rate is typically 24 frames per second. For film, this is 30 frames per second.



In general, animation may be achieved by specifying a model with n parameters that identify degrees of freedom that an animator may be interested in such as

• polygon vertices,

• spline control,

• joint angles,

• muscle contraction,

• camera parameters, or color.

With n parameters, this results in a vector q in n-dimensional state space. Parameters may be varied to generate animation. A model's motion is a trajectory through its state space or a set of motion curves for each parameter over time, i.e. ~q(t), where t is the time of the current frame.

Every animation technique reduces to specifying the state space trajectory.

The basic animation algorithm is then: for t=t1 to tend: render(~q(t)).

Modeling and animation are loosely coupled. Modeling describes control values and their actions.

Animation describes how to vary the control values. There are a number of animation techniques,
including the following:

• User driven animation

– Keyframing

– Motion capture

• Procedural animation

– Physical simulation

– Particle systems

– Crowd behaviors

• Data-driven animation

Keyframing

Keyframing is an animation technique where motion curves are interpolated through states at times, (~q1, ..., ~qT), called keyframes, specified by a user

Kinematics

Kinematics describe the properties of shape and motion independent of physical forces that cause motion. Kinematic techniques are used often in keyframing, with an animator either setting joint parameters explicitly with forward kinematics or specifying a few key joint orientations and having the rest computed automatically with inverse kinematics.

Forward Kinematics

With forward kinematics, a point p is positioned by p = f(_) where_is a state vector ($\theta 1$, $\theta 2$, ...θn) specifying the position, orientation, and rotation of all joints.

For the above example, p

p = (l1 cos($\theta 1$) + l2 cos($\theta 1 + \theta 2$), l1 sin($\theta 1$) + l2 sin($\theta 1 +$
$\theta 2$)).

Inverse Kinematics

With inverse kinematics, a user specifies the position of the end effector, p, and the algorithm has to evaluate the required give p. That is = f−1(p).

Usually, numerical methods are used to solve this problem, as it is often nonlinear and either underdetermined or over determined. A system is underdetermined when there is not a unique solution, such as when there are more equations than unknowns. A system is over determined when it is inconsistent and has no solutions. Extra constraints are necessary to obtain unique and stable solutions. For example, constraints may be placed on the range of joint motion and the solution may be required to minimize the
kinetic energy of the system.

Motion Capture

In motion capture, an actor has a number of small, round markers attached to his or her body that reflect light in frequency ranges that motion capture cameras are specifically designed to pick up.

INPUT

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

const double PI = 3.141592654;
int frameNumber = 0;

void drawDisk(double radius) {
        int d;
        glBegin(GL_POLYGON);
        for (d = 0; d < 32; d++) {
                double angle = 2*PI/32 * d;
                glVertex2d( radius*cos(angle), radius*sin(angle));
        }
        glEnd();
}

void drawship() {
        glColor3f(1.5,1,0);
        glBegin(GL_POLYGON);
        glVertex2f(1.0f,5);//x1
        glVertex2f(-2.0f,4.5);//x2
        glVertex2f(1.0f,4);//y2
        glEnd();

        glColor3f(1.5,0,1);
        glBegin(GL_LINES);
        glVertex2f(1.0f,2);//x1
        glVertex2f(1.0f,4);//y2
        glEnd();

        glColor3f(1,0,0);
        glBegin(GL_POLYGON);
        glVertex2f(-2.0f,0);//x1
        glVertex2f(2.0f,0);//x2
        glVertex2f(3.0f,2);//y2
        glVertex2f(-3.0f,2);//y1

        glEnd();
}



void drawSun() {
        int i;
        glColor3f(1.0,0.8,0.0);
        for (i = 0; i < 20; i++) {
                glRotatef( 360 / 20, 0, 0, 1 );
                glBegin(GL_LINES);
                glVertex2f(0, 0);
                glVertex2f(0.75f, 0);
                glEnd();
        }
        drawDisk(0.5);
        glColor3f(0,0,0);
}

void display() {

        glClear(GL_COLOR_BUFFER_BIT);
        glLoadIdentity();



        glColor3f(0, 0.6f, 0.2f);
        glBegin(GL_POLYGON);
        glVertex2f(-3,-1);
        glVertex2f(1.5f,1.65f);
        glVertex2f(5,-1);
        glEnd();

        glBegin(GL_POLYGON);
        glVertex2f(-3,-1);
        glVertex2f(3,2.1f);
        glVertex2f(7,-1);
```

```
glEnd();
glBegin(GL_POLYGON);
glVertex2f(0,-1);
glVertex2f(6,1.2f);
glVertex2f(20,-1);
glEnd();



glColor3f(0.2f, 0.2f, 1.0f);
glBegin(GL_POLYGON);
glVertex2f(0,-0.4f);
glVertex2f(7,-0.4f);
glVertex2f(7,0.4f);
glVertex2f(0,0.4f);
glEnd();



glPushMatrix();
glTranslated(1.8,3,0);
glRotated(-frameNumber*2.7,0,0,1);
drawSun();
glPopMatrix();


glPushMatrix();

glTranslated(-3 + 15*(frameNumber % 300) / 300.0, 0, 0);
glScaled(0.3,0.3,1);
```

```
        drawship();
        glPopMatrix();

        glutSwapBuffers();

}

void doFrame(int v) {
   frameNumber++;
   glutPostRedisplay();
   glutTimerFunc(30,doFrame,0);
}

void init() {
        glClearColor(0.6f, 0.6f, 1.0f, 0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0, 7, -1, 4, -1, 1);
        glMatrixMode(GL_MODELVIEW);
}


int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE);
   glutInitWindowSize(700,500);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Animation");

   init();
```

```
    glutDisplayFunc(display);
    glutTimerFunc(200,doFrame,0);


    glutMainLoop();
    return 0;
}
```

<u>OUTPUT</u>