214457: Computer Graphics Lab

Examination Scheme

Term Work: 25 Marks

Practical : 25 Marks

List of Laboratory Assignments
1. Install and explore the OpenGL.
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 withcenter 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.
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.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 .

Assignment No 1

Assignment Name:

Computer Graphics(Basic and how to install OpenGL/Glut)

INTRODUCTION

What is Computer Graphics?

Branch of Computer Science.

Computer + Graphs + Pics = Computer Graphics.

Drawing line, Chart, Graphs etc. on the screen using Programming language is computer Graphics.

Computer graphics is the branch of computer science that deals with generating images with the aid of computers. It displays the information in the from of graphics objects such as picture, charts, graphs and diagrams instead of simple text. We can say computer graphics makes it possible to express data in pictorial form.

OpenGL

It is cross-platform, cross-language API for rendering 2D and 3D Graphics(Vector Graphics).

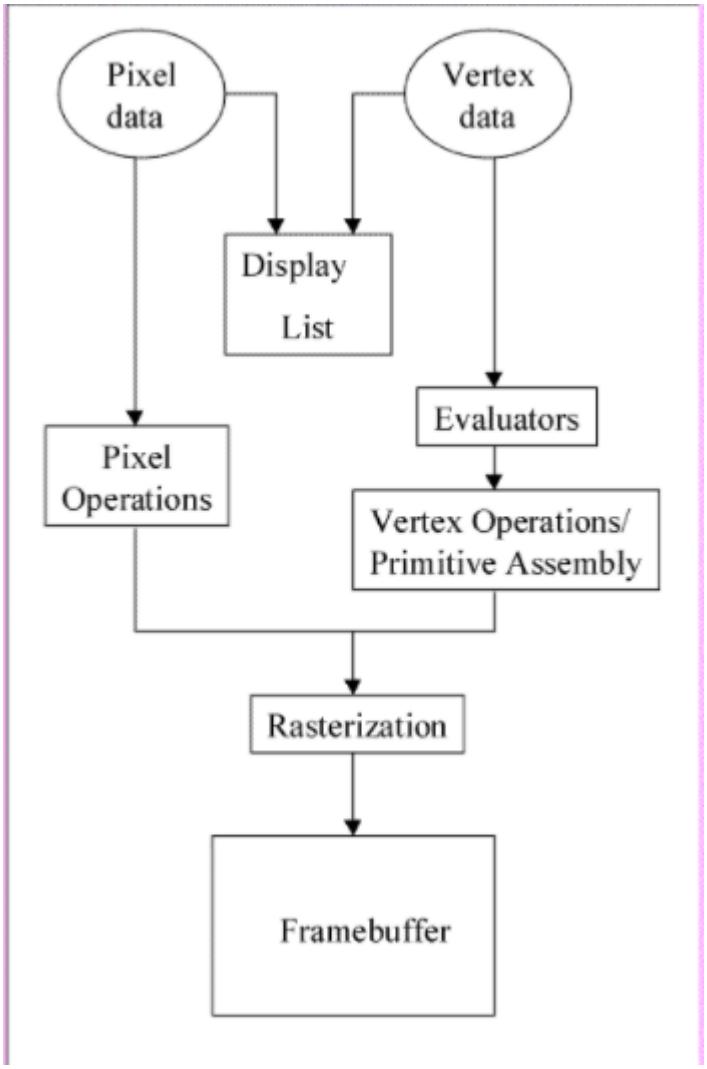
Getting started with OpenGL

Open Graphics Library (OpenGL) is a cross-language (language independent), cross-platform (platform-independent) API for rendering 2D and 3D Vector Graphics(use of polygons to represent image). OpenGL API is designed mostly in hardware.

- **Design :** This API is defined as a set of functions which may be called by the client program. Although functions are similar to those of **C** language but it is language independent.
- **Development :** It is an evolving API and **Khronos Group** regularly releases its new version having some extended feature compare to previous one. GPU vendors may also provide some additional functionality in the form of extension.
- **Associated Libraries :** The earliest version is released with a companion library called OpenGL utility library. But since OpenGL is quite a complex process. So in order to make it easier other library such as OpenGL Utility Toolkit is added which is later superseded by free glut. Later included library were GLEE, GLEW, and gliding.
- **Implementation :** Mesa 3D is an open source implementation of OpenGL. It can do pure software rendering and it may also use hardware acceleration on BSD, Linux, and other platforms by taking advantage of Direct Rendering Infrastructure.

Rendering Pipeline

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. Although this is not a strict rule of how OpenGL is implemented, it provides a reliable guide for predicting what OpenGL will do. Geometric data (verices, line, and polygons) follow a path through the row of boxes that includes evaluators and per-vertex operations, while pixel data (pixels, images and bitmaps) are treated differently for part of the process. Both types of data undergo the same final step (raterization) before the final pixel data is written to the framebuffer.



Display Lists: All data, whether it describes geometry or pixels, can be saved in a display list for current or later use. (The alternative to retaining data in a display list is processing the data immediately-known as immediate mode.) When a display list is

executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode.

Evaluators: All geometric primitives are eventually described by vertices. Evaluators provide a method for deviving the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, colors, and spatial coordinate values from the control points.
Per-Vertex and Primitive Assembly: For vertex data, the next step converts the vertices into primitives. Some types of vertex data are transformed by 4x4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on your screen. In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then viewport and depth operations are applied. The results at this point are geometric primitives, which are transformed with related color and depth vlaues and guidelines for the rasterization step.

Pixel Operations: While geometric data takes one path through the OpenGL rendering papeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked form one of a variety of formats into the proper number of components. Next the data is scaled, biased, processed by a pixel map, and sent to the rasterization step.

Rasterization: Rasterization is the conversion of both geometric and pixel data into fragments. Each fragment square corresponds to a pixel in the framebuffer. Line width, point size, shading model, and coverage calculations to support antialiasing are taken ito consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth values are assigned for each fragment square. The processed fragment is then drawn into the appropriate buffer, where it has finally advanced to be a pixel and achieved its final resting place.

Libraries

OpenGL provides a powerful but primitive set of rendering command, and all higher-level drawing must be done in terms of these commands. There are several libraries that allow you to simplify your programming tasks, including the following:

OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matracies for specific viewing orientations and projections and rendering surfaces.

OpenGL Utility Toolkit (GLUT) is a window-system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window APIs.

Include Files

For all OpenGL applications, you want to include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which also requires inclusion of the glu.h header file. So almost every OpenGL source file begins with:

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

If you are using the OpenGL Utility Toolkit (GLUT) for managing your window manager tasks, you should include:

#include <GL/glut.h>

OpenGl Functions:

- `glutInit:`

Used to initialize GLUT.

- `glutCreateWindow`

Used for creating a top-level window. You can supply the window name as a label while creating the window.

- `glutInitWindowSize`

Used to define the window size. The width and height of the window are specified in pixels while defining the window size.

- `void glutInitWindowPosition`

Used to set the initial window position. The window's *x* and *y* locations are specified in terms of pixels.

- `glutDisplayFunc`

Used to specify the callback function to be executed to display graphics in the current window. For redisplaying the content in the window too, the specified callback function...

Conclusion:

Hence, we have successfully install openGL on Ubuntu

How to install OpenGL / Glut (Ubuntu)

Step 1: sudo apt-get update
To update your basic packages
Step 2: sudo apt-get install build-essential
For installing essential packages.
Step 3: sudo apt-get install freeglut3 freeglut3-dev
Step 4: sudo apt-get install binutils-gold
Step 5: sudo apt-get install g++ cmake
Step 6: sudo apt-get install mesa-common-dev mesa-utils
Step 7: sudo apt-get install libglew-dev libglew1.5-dev libglm-dev
Step 8: glxinfo | grep OpenGL

Create cpp file and write your code

Run on terminal:

- ***g++ MyProg.cpp -lGL -lGLU -lglut (for C++ program)***
- ***./a.out***

Simple Program of Triangle/CODE:

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


// Clears the current window and draws a triangle.
void display() {

  // Set every pixel in the frame buffer to the current clear color.
  glClear(GL_COLOR_BUFFER_BIT);

  // Drawing is done by specifying a sequence of vertices.  The way these
  // vertices are connected (or not connected) depends on the argument to
  // glBegin.  GL_POLYGON constructs a filled polygon.
  glBegin(GL_POLYGON);
    glColor3f(1, 0, 0); glVertex3f(-0.6, -0.75, 0.5);
    glColor3f(0, 1, 0); glVertex3f(0.6, -0.75, 0);
    glColor3f(0, 0, 1); glVertex3f(0, 0.75, 0);
  glEnd();

  // Flush drawing command buffer to make drawing happen as soon as
possible.
  glFlush();
}

// Initializes GLUT, the display mode, and main window; registers
callbacks;
// enters the main event loop.
int main(int argc, char** argv) {
// Use a single buffered window in RGB mode (as opposed to a double-
buffered
```

```
  // window or color-index mode).
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

  // Position window at (80,80)-(480,380) and give it a title.
  glutInitWindowPosition(80, 80);
  glutInitWindowSize(400, 300);
  glutCreateWindow("A Simple Triangle");

  // Tell GLUT that whenever the main window needs to be repainted that it
  // should call the function display().
  glutDisplayFunc(display);

  // Tell GLUT to start reading and processing events.  This function
  // never returns; the program only exits when the user closes the main
  // window or kills the process.
  glutMainLoop();
}
```

Assignment No 2

Assignment Name:

Implement DDA and Bresenham line drawing algorithm to draw:

i) Simple Line

ii) Dotted Line

iii) Dashed 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.

1) DDA Line Drawing Algorithm

DDA stands for Digital Differential Analyzer. It is an incremental method of scan conversion of line. In this method calculation is performed at each step but by using results of previous steps.

DDA Line Drawing Algorithm:

- **Step1:** Start Algorithm
- **Step2:** Declare x_1,y_1,x_2,y_2,dx,dy,x,y as integer variables.
- **Step3:** Enter value of x_1,y_1,x_2,y_2.
- **Step4:** Calculate dx = x_2-x_1
- **Step5:** Calculate dy = y_2-y_1
- **Step6:** If ABS (dx) > =ABS (dy)
  Then step = abs (dx)
  Else steps=abs(dy)
- **Step7:** x_{inc}=dx/step
  y_{inc}=dy/step
  assign x = x_1
  assign y = y_1
- **Step8:** Set pixel (x, y)

- **Step9:** Xnew = $x + x_{inc}$
  Ynew = $y + y_{inc}$
  Set pixels (Round (x), Round (y))
- **Step10:** Repeat step 9 until $x = x_2$
- **Step11:** End Algorithm

Advantages of DDA :

- It is simple and easy to implement algorithm.
- It avoid using multiple operations which have high time complexities.
- It is faster than the direct use of the line equation because it does not use any floating point multiplication and it calculates points on the line.

Disadvantages of DDA :

- It deals with the rounding off operation and floating point arithmetic so it has high time complexity.
- As it is orientation dependent, so it has poor endpoint accuracy.
- Due to the limited precision in the floating point representation it produces cumulative error.

2)Bresenham's Line Algorithm

This algorithm is used for scan converting a line. It was developed by Bresenham. It is an efficient method because it involves only integer addition, subtractions, and multiplication operations. These operations can be performed very rapidly so lines can be generated quickly.

In this method, next pixel selected is that one who has the least distance from true line.

Steps:

1. Accept two endpoints from user and store the left endpoint in (x_0,y_0) as starting point.
2. Plot the point (x_0,y_0)
3. Calculate all constants from two endpoints such as Dx, Dy, 2Dy,

2Dy-2Dx and find the starting value for the G as G= 2Dy-Dx.

4. For each column increment x and decide y-value by checking G>0 condition. If it is true then increment y-value and add (2Dy-2Dx) to current value of G otherwise add(2Dy) to G and don't increment y-value. Plot next point.
5. Repeat step 4 till Dx times.

Advantage:

1. It involves only integer arithmetic, so it is simple.

2. It avoids the generation of duplicate points.

3. It can be implemented using hardware because it does not use multiplication and division

4. It is faster as compared to DDA (Digital Differential Analyzer) because it does not involve floating point calculations like DDA Algorithm.

Disadvantage:

1. This algorithm is meant for basic line drawing only Initializing is not a part of Bresenham's line algorithm. So to draw smooth lines, you should want to look into a different algorithm.

Conclusion:

Hence, We have Successfully implemented DDA and Bresenham's Line Drawing Algorithm.

CODE:

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

int Algo,type;

void Init()
{
    glClearColor(0,0,0,0);
    glColor3f(0,1,0);
    gluOrtho2D(0,640,0,480);
    glClear(GL_COLOR_BUFFER_BIT);
}


int sign(float a){


    if(a==0){
        return 0;
    }
    if(a>0){

        return 1;
    }
    return -1;

}
void B_Line(int x_1,int y_1,int x_2,int y_2,int t){

    float dy, dx, m , P;
    dy = y_2 - y_1;
    dx = x_2 - x_1;

    m = dy/dx;

    P = 2*dy - dx;

    int x = x_1, y = y_1;
    cout<<"\n x1 = "<<x<<" y1 = "<<y;
    if(m<1){
        int cnt=1;
        for(int i=0; i<=dx;i++){

            if(t == 1){
                glBegin(GL_POINTS);
                    glVertex2i(x,y);
                glEnd();
            }
            if(t == 2){

                if(i%2==0){
```

```
                glBegin(GL_POINTS);
                    glVertex2i(x,y);
                glEnd();
            }
        }
        if(t == 3){

            if(cnt <= 10){
                glBegin(GL_POINTS);
                    glVertex2i(x,y);
                glEnd();

            }
            cnt++;
            if(cnt == 15){
                cnt =1;
            }

        }

        if(P<0){

            x = x +1;
            y =y;
            P = P + 2*dy;
        }
        else{

            x= x+1;
            y = y+1;
            P = P + 2*dy - 2*dx;

        }
    }
}
else{
    int cnt = 1;
    for(int i=0;i<=dy;i++){
        if(t == 1){
            glBegin(GL_POINTS);
                glVertex2i(x,y);
            glEnd();
        }
        if(t == 2){

            if(i%2==0){
                glBegin(GL_POINTS);
                    glVertex2i(x,y);
                glEnd();
            }
        }
        if(t == 3){

            if(cnt <= 10){
```

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

                }
                cnt++;
                if(cnt == 15){
                    cnt =1;
                }

            }
            if(P<0){

                x = x;
                y =y+1;
                P = P + 2*dx;
            }
            else{

                x= x+1;
                y = y+1;
                P = P + 2*dx - 2*dy;

            }


        }
    }
    cout<<"\n xlast = "<<x<<" ylast = "<<y;
    glFlush();
}

void DDA_LINE(int x_1,int y_1,int x_2,int y_2, int t){


    float dx,dy,length;
    dx = x_2-x_1;
    dy = y_2-y_1;

    if(abs(dx) >= abs(dy)){

        length = abs(dx);
    }
    else{
        length = abs(dy);
    }

    float xin, yin;
    xin = dx/length;
    yin = dy/length;

    float x,y;

    x = x_1 + 0.5 * sign(xin);
```

```
    y = y_1 + 0.5 * sign(yin);

    int i=0;
    int cnt =1;
    while(i<=length){
        if(t == 1){
            glBegin(GL_POINTS);
                glVertex2i(x,y);
            glEnd();
        }
        if(t == 2){

            if(i%2==0){
                glBegin(GL_POINTS);
                    glVertex2i(x,y);
                glEnd();
            }
        }
        if(t == 3){

            if(cnt <= 10){
                glBegin(GL_POINTS);
                    glVertex2i(x,y);
                glEnd();

            }
            cnt++;
            if(cnt == 15){
                cnt =1;
            }

        }

        x = x + xin;
        y = y + yin;
        i++ ;

    }

    glFlush();

}

void display()
{
    DDA_LINE(0,240,640,240,1);
    B_Line(320,0,320,640,1);


    glFlush();
}
void mymouse(int b,int s, int x, int y)
{
    static int x_s,y_s,x_e,y_e,pt=0;
```

```
    if(b==GLUT_LEFT_BUTTON && s==GLUT_DOWN)
    {
        if(pt==0)
        {
            x_s =x;
            y_s =480 - y;
            pt++;

            glBegin(GL_POINTS);
                glVertex2i(x_s,y_s);
            glEnd();
        }
        else
        {
            x_e=x;
            y_e=480-y;
            cout<<"\n x_1_click "<<x_s<<" y_1_click "<<y_s;
            cout<<"\n x_2_click "<<x_e<<" y_2_click "<<y_e<<"\n";
            glBegin(GL_POINTS);
                glVertex2i(x_e,y_e);
            glEnd();
            if(Algo == 1){
                DDA_LINE(x_s,y_s,x_e,y_e,type);
            }
            if(Algo == 2){
                B_Line(x_s,y_s,x_e,y_e,type);
            }

        }


    }
    else if(b==GLUT_RIGHT_BUTTON && s==GLUT_DOWN)
        {
            pt=0;
        }
    glFlush();
}




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

    cout<<"\n Select the Algorithm \n 1. DDA \n 2. Bresenham's \n";
    cin>>Algo;
    cout<<"Select the Line Type \n 1. Simple Line \n 2. Dotted Line\n 3.
Dashed Line \n";
    cin>>type;

    if((Algo == 1 || Algo == 2 )&&(type==1 || type==2 || type==3)){
```

```
    }
    else{
        cout<<"\n Option enter are wrong \n";
        return 0;
    }

    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowPosition(100,100);
    glutInitWindowSize(640,480);
    glutCreateWindow("DDA-Line");
    Init();
    glutDisplayFunc(display);
    glutMouseFunc(mymouse);

    glutMainLoop();
    return 0;

}
```

Assignment No. 3

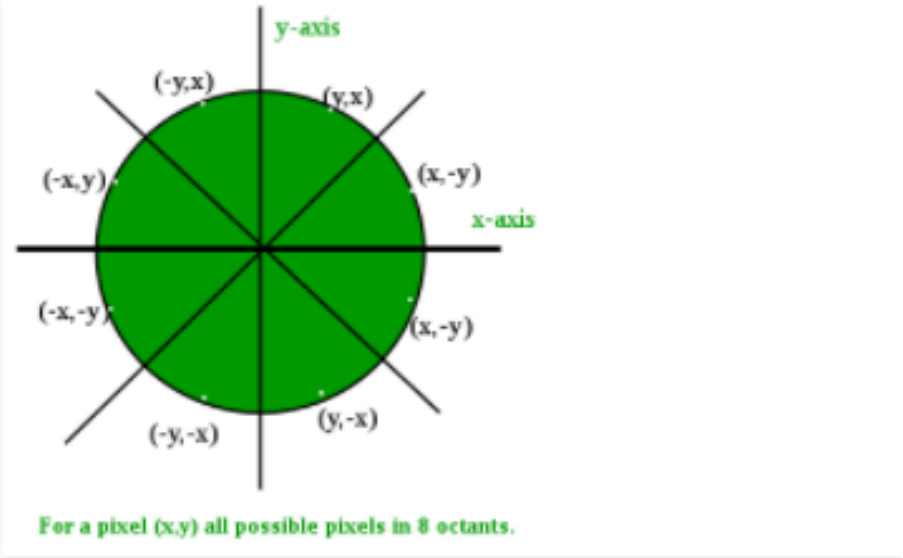
Assignment Name –

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

It is not easy to display a continuous smooth arc on the computer screen as our computer screen is made of pixels organized in matrix form. So, to draw a circle on a computer screen we should always choose the nearest pixels from a printed pixel so as they could form an arc. There are two algorithm to do this:

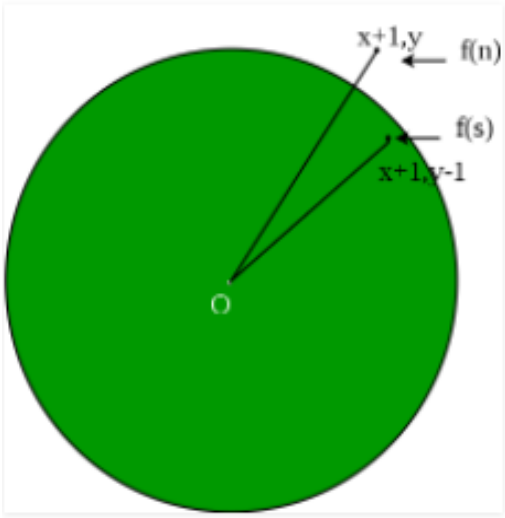
1. Mid-Point circle drawing algorithm
2. Bresenham's circle drawing algorithm

Both of these algorithms uses the key feature of circle that it is highly symmetric. So, for whole 360 degree of circle we will divide it in 8-parts each octant of 45 degree. In order to do that we will use Bresenham's Circle Algorithm for calculation of the locations of the pixels in the first octant of 45 degrees. It assumes that the circle is centered on the origin. So for every pixel (x, y) it calculates, we draw a pixel in each of the 8 octants of the circle as shown below



For a pixel (x,y) all possible pixels in 8 octants.

Now See, how to calculate the next pixel location from a previously known pixel location (x, y). In Bresenham's algorithm at any point (x, y) we have two option either to choose the next pixel in the east i.e. (x+1, y) or in the south east i.e. (x+1, y-1).



And this can be decided by using the decision parameter d as:

- If d > 0, then (x+1, y-1) is to be chosen as the next pixel as it will be closer to the arc.
- else (x+1, y) is to be chosen as next pixel.

Now to draw the circle for a given radius 'r' and centre (xc, yc) We will start from (0, r) and move in first quadrant till x=y (i.e. 45 degree). We should start from listed initial condition:

```
d = 3 - (2 * r)
x = 0
y = r
```

Now for each pixel, we will do the following operations:

1. Set initial values of (xc, yc) and (x, y)
2. Set decision parameter d to d = 3 – (2 * r).
3. call drawCircle(int xc, int yc, int x, int y) function.
4. Repeat steps 5 to 8 until x < = y
5. Increment value of x.
6. If d < 0, set d = d + (4*x) + 6
7. Else, set d = d + 4 * (x – y) + 10 and decrement y by 1.
8. call drawCircle(int xc, int yc, int x, int y) function

Bresenham's Circle Algorithm:

Step 1: Start Algorithm

Step 2: 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

Step 3: Enter the value of r

Step 4: Calculate d = 3 - 2r

Step 5: Initialize x=0

&nbsy= r

Step 6: Check if the whole circle is scan converted

If x > = y

Stope

Step 7: 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)

Step 8: 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

Step 9: Go to step 6

Step 10: Stop Algorithm

Conclusion:

Hence, We have successfully performed Bresemham's circle drawing algorithm.

CODE:

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

int r;

void E_way(int x, int y){

        glBegin(GL_POINTS);
            glVertex2i(x+320,y+240);
            glVertex2i(y+320,x+240);
            glVertex2i(y+320, -x+240);
            glVertex2i(x+320, -y+240);
            glVertex2i(-x+320,-y+240);
            glVertex2i(-y+320,-x+240);
            glVertex2i(-y+320,x+240);
            glVertex2i(-x+320,y+240);
        glEnd();
        glFlush();

}

void B_circle(){

    float d;
    d = 3 - 2*r;

    int x,y;
    x = 0 ;
    y = r ;

    do{
        E_way(x,y);
```

```
        if(d<0){

            d=d+4*x+6;
        }
        else{
            d= d+4*(x-y)+10;
            y=y-1;
        }
        x=x+1;

    }while(x<y);


}

void init(){

    glClearColor(1,1,1,0);
    glColor3f(1,0,0);
    gluOrtho2D(0,640,0,480);
    glClear(GL_COLOR_BUFFER_BIT);
}

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

    cout<<"\n Enter Radius \t ";
    cin>>r;

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    glutInitWindowPosition(100,100);
    glutInitWindowSize(640,480);
    glutCreateWindow("Circle");
    init();
    glutDisplayFunc(B_circle);

    glutMainLoop();

    return 0;
}
```

Assignment No 4

Assignment Name:

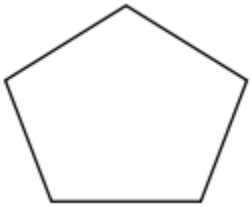
Implement the following polygon filling methods : i) Flood fill / Seed fill ii) Boundary fill ; using mouse click, keyboard interface and menu driven programming-

Prerequisite: Basic of polygon types and polygon filling algorithms.

Relevant Theory:

Polygon

When starting and ending point of any polyline is same i.e. when polyline is closed then it is called polygon

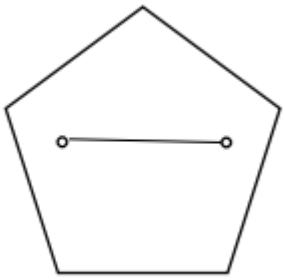


There are two types of polygon

The classification of polygons is based on where the line segment joining any two points within the polygon is going to lie.

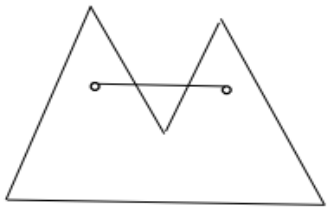
1) **Convex polygon**

It is a polygon in which any two points taken are surely inside the polygon lies complete inside the polygon, then that polygon is called as convex polygon



2) **Concave polygon**

It is a polygon in which the line segment joining any two points within the polygon may not lie completely inside the polygon.



Representation of Polygon

To display polygon on graphic system, it must first be decided, now to represent it. These are 3 approaches to represent polygons accordingly.

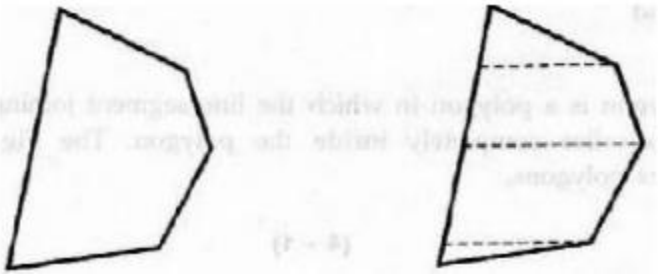
1) Polygon drawing primitive approach

Some graphics device supports polygon drawing primitive approach. Directly drawn polygon in graphics system is stored & that polygon acts as a whole unit.

2) Trapezoid primitive Approach

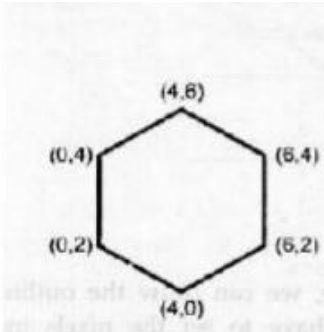
Some graphics devices support trapezoid primitive Trapezoids are formed from two scan line and two scan lines and two line segments.

`



3) Line & Point Approach

Most of the graphics devices to not provide any polygon support at all. In such cases, Polygon can be broken into lines & points and stored the full polygon in the display file The display polygon is then done using line commands.



DF_OP	DF_ x	DF_ y
6	0	2
2	0	4
2	4	6
2	6	4
2	6	2
2	4	0
2	0	2

Convex polygon

A convex polygon is defined as a polygon with all its interior angles less than 180°. This means that all the vertices of the polygon will point outwards, away from the interior of the shape. Think of it as a 'bulging' polygon. Note that a triangle (3-gon) is always convex.

Polygon filling

It is the process of coloring in a fixed area or region. It also means high lighting all the pixels which lie inside the polygon with any color then other background color. There are two basic approaches used to fill the polygon. One way to fill polygon is to start from a given "seed "point known to be inside the polygon & highlight outward from the point in neighboring pixels. Until we encounter the boundary pixel. The approach is called "seed fill." Another approach to fill polygon is "scan line" in which it is checked whether pixel is inside the polygon or outside the polygon.

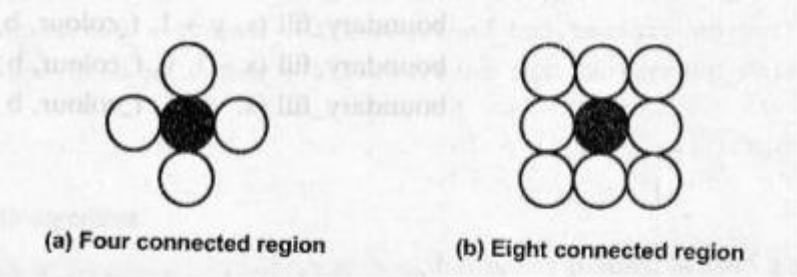
Seed Fill

The seed fill algorithm is further classified as flood algorithm and boundary fill algorithm. Algorithms that fill interior defined regions are called flood-fill algorithms; those that fill boundary defined regions are called boundary fill algorithms or edge fill algorithms.

Boundary fill

algorithm In this algorithm, one point which is surely inside the polygon is taken. This point is called seed point which is nothing but a point from which filling process starts. Boundary defines region may be either 4-connected or 8-connected.

Check the color of pixel If boundary color i.e pixel are not reached, pixels are highlighted (by fill colored) & the process is continued until boundary pixels are reached.



Algorithm

```
Boundary_fill(x,y,f_color,b_color)
{
if(getpixel(x,y)!=b_color &&getpixel(x,y)!=f_color)
{
putpixel(x,y,f_color)
Boundary_fill(x+1,y,f_color,b_color)
Boundary_fill(x,y,+1f_color,b_color)
Boundary_fill(x-1,y,f_color,b_color)
Boundary_fill(x,y-1,f_color,b_color)
 }
 }
```

Flood Fill Algorithm

In this algorithm, areas are filled by replacing a specified interior color instead of searching a specified interior boundary color. Here, instead of checking boundary color, whether pixels are having polygons original color is checked. Using either 4-connected or 8-connected approach step through pixel position until all interior point has been filled.

Algorithm

```
Flood_fill (x,y, old_color, new_color)
{
 If (getpixel(x,y)=old_color)
{
Putpixel (x,y,new color)
```

```
Flood_fill (x+1,y, old_color, new_color)
Flood_fill (x-1,y, old_color, new_color)
Flood_fill (x,y,+1 old_color, new_color)
Flood_fill (x,y-1, old_color, new_color)
Flood_fill (x+1,y+1, old_color, new_color)
Flood_fill (x-1,y-1, old_color, new_color)
Flood_fill (x+1,y-1, old_color, new_color)
Flood_fill (x-1,y+1, old_color, new_color)
 }
}
```

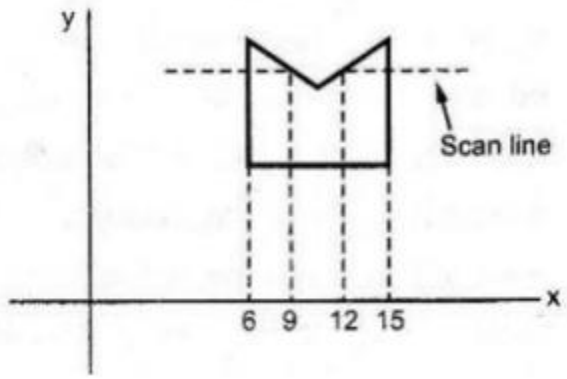
Hence, getpixel function gives color of specified pixel & putpixel function draws pixel specified color.

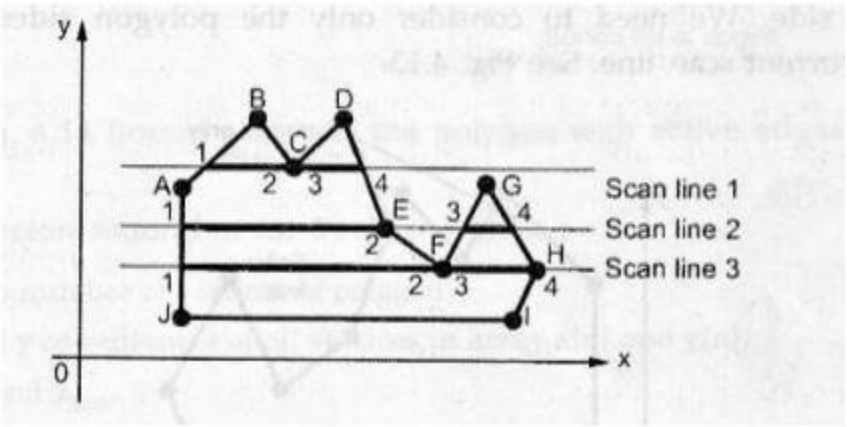
Scan-line algorithm

The following illustrates the scan line algorithm for filling of polygon. Identify rightmost and leftmost pixels of the seed pixel and then drawing a horizontal line between these boundary pixels. This procedure is repeated until complete polygon is filled. We have to stack only beginning position of each horizontal line. For each scan line crossing a polygon, this algorithm locates the intersection points of scan line with polygon edges.

These intersection points are then sorted from left to right & the corresponding positions between each intersection pair are set to the specified fill color. The scan line algorithm first finds the largest & smallest y values of the polygon.

Then it starts with the largest y value & works it may down. Scanning is done from left to right in the manner of a raster display.





Features of scan line algorithm

1. Scan line algorithm is used for filling of polygons.

2. This algorithm solves the problem of hidden surfaces while generating display scan line.

3. It is used in orthogonal projection.

4. It is non recursive algorithm.

5. In scan line algorithm we have to stack only a beginning position for horizontal pixel scan, instead of stacking all unprocessed neighboring positions around the current position. Therefore it is sufficient algorithm.

Conclusion:- In This way we have studied that how to draw a convex polygon and how to fill a polygon by using different polygon filling algorithm

CODE:

```
#include <iostream>
#include <math.h>
#include <GL/glut.h>
using namespace std;
float R=0,G=0,B=0;
int Algo;
void init(){
glClearColor(1.0,1.0,1.0,0.0);
glMatrixMode(GL_PROJECTION);
gluOrtho2D(0,640,0,480);
}
void floodFill(int x, int y, float *newCol, float *oldcol){
float pixel[3];
glReadPixels(x,y,1,1,GL_RGB,GL_FLOAT,pixel);
```

```
if(oldcol[0]==pixel[0] && oldcol[1]==pixel[1] && oldcol[2]==pixel[2]){
glBegin(GL_POINTS);
glColor3f(newCol[0],newCol[1],newCol[2]);
glVertex2i(x,y);
glEnd();
glFlush();
floodFill(x,y+1,newCol,oldcol);
floodFill(x+1,y,newCol,oldcol);
floodFill(x,y-1,newCol,oldcol);
floodFill(x-1,y,newCol,oldcol);
}
}
void boundaryFill(int x, int y, float* fillColor, float* bc){
float color[3];
glReadPixels(x,y,1.0,1.0,GL_RGB,GL_FLOAT,color);
if((color[0]!=bc[0] || color[1]!=bc[1] || color[2]!=bc[2]) && (fillColor[0]!=color[0] ||
fillColor[1]!=color[1] || fillColor[2]!=color[2]))
{
glColor3f(fillColor[0],fillColor[1],fillColor[2]);
glBegin(GL_POINTS);
glVertex2i(x,y);
glEnd();
glFlush();
boundaryFill(x+1,y,fillColor,bc);
boundaryFill(x-1,y,fillColor,bc);
boundaryFill(x,y+1,fillColor,bc);
boundaryFill(x,y-1,fillColor,bc);
}
return;
}
void mouse(int btn, int state, int x, int y){
y = 480-y;
if(btn == GLUT_LEFT_BUTTON && state == GLUT_DOWN){
float bcol[] = {1,0,0};
float oldcol[] = {1,1,1};
float newCol[] = {R,G,B};
if(Algo==1){
boundaryFill(x,y,newCol,bcol);
}
if(Algo==2){
floodFill(x,y,newCol,oldcol);
}
}
}
void B_Draw(){
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1,0,0);
glBegin(GL_LINE_LOOP);
glVertex2i(150,100);
glVertex2i(300,300);
glVertex2i(450,100);
```

```
glEnd();
glFlush();
}
void F_Draw(){
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_LINES);
glColor3f(1,0,0);glVertex2i(150,100);glVertex2i(300,300);
glEnd();
glBegin(GL_LINE_LOOP);
glColor3f(0,0,1);glVertex2i(300,300);glVertex2i(450,100);
glEnd();
glBegin(GL_LINE_LOOP);
glColor3f(0,0,0);glVertex2i(450,100);glVertex2i(150,100);
glEnd();
glFlush();
}
void goMenu(int value){
switch(value){
case 1:
R = 0, G = 1, B=0;
break;
case 2:
R = 1, G = 1, B=0;
break;
case 3:
R = 0, G = 0, B=1;
break;
}
glutPostRedisplay();
}
int main(int argc, char** argv){
cout<<"\n \t Select the Algorithm ";
cout<<"\n \t 1. Boundary Fill Algorithm ";
cout<<"\n \t 2. Flood Fill Algorithm \n \t";
cin>>Algo;
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(640,480);
glutInitWindowPosition(200,200);
glutCreateWindow("Boundary Fill and Flood Fill");
init();
glutCreateMenu(goMenu);
glutAddMenuEntry("Color 1 Green",1);
glutAddMenuEntry("Color 2 Yellow",2);
glutAddMenuEntry("Color 3 Blue",3);
glutAttachMenu(GLUT_RIGHT_BUTTON);
if(Algo==1){
glutDisplayFunc(B_Draw);
}
if(Algo==2){
glutDisplayFunc(F_Draw);
```

```
}
glutMouseFunc(mouse);
glutMainLoop();
return 0;
}
```

Assignment No. 5

Aim:

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

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).

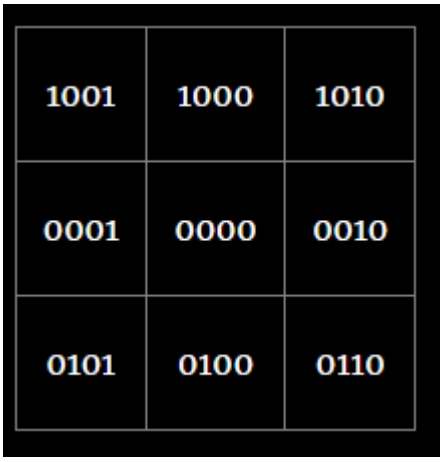
The algorithm was developed in 1967 during flight simulator work by Danny Cohen and Ivan Sutherland

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.



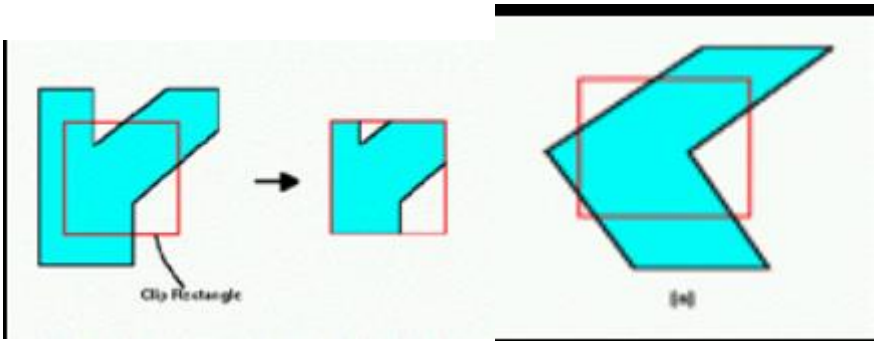
Sutherland Hodgman Polygon Clipping

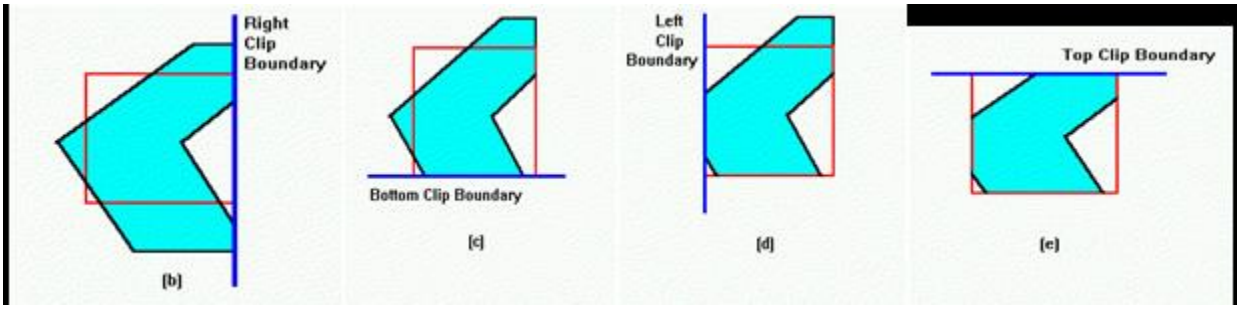
It is used for clipping polygons. It works by extending each line of the convex clip polygon in turn and selecting only vertices from the subject polygon those are on the visible side.

An algorithm that clips a polygon must deal with many different cases. The case is particularly note worthy 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.

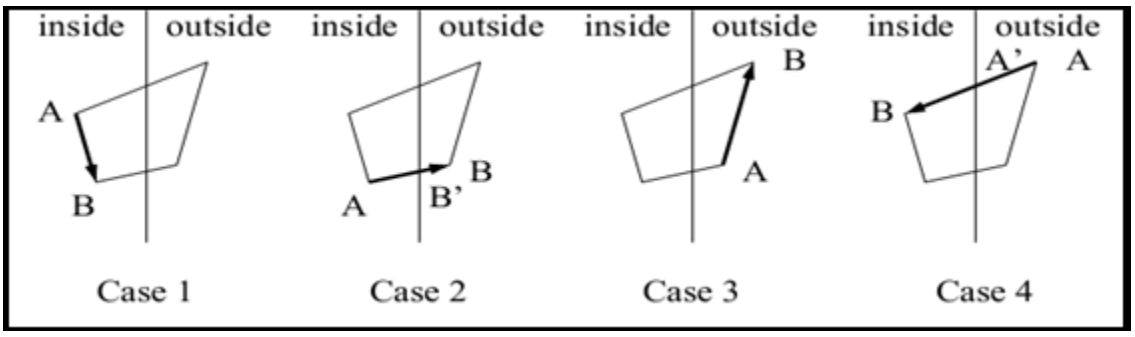
This process is repeated iteratively for each clip polygon side, using the output list from one stage as the input list for the next. Once all sides of the clip polygon have been processed, the final generated list of vertices defines a new single polygon that is entirely visible. Note that if the subject 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.





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.



he Sutherland-Hodgeman Polygon-Clipping Algorithms clips a given polygon successively against the edges of the given clip-rectangle. These clip edges are denoted with e1, e2, e3, and e4, here. The closed polygon is represented by a list of its verteces (v1 to vn; Since we got 15 verteces in the example shown above, vn = v15.).

Clipping is computed successively for each edge. The output list of the previous clipping run is used as the inputlist for the next clipping run. 1st run: Clip edge: e1; inputlist = {v1, v2, ..., v14, v15}, the given polygon

Conclusion:

Hence, We have performed Cohen Sutherland polygon clipping method to clip the polygon with respect the view-port and window

CODE:

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

using namespace std;

int wxmin = 200,wxmax=500,wymax=350, wymin=100;
int points[10][2];
int edge;

void init(){
   glClearColor(1.0,1.0,1.0,0.0);
   glMatrixMode(GL_PROJECTION);
   gluOrtho2D(0,640,0,480);
   glClear(GL_COLOR_BUFFER_BIT);
}


void Draw(){
   glClearColor(1.0,1.0,1.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.2,0.2,1);
   glBegin(GL_POLYGON);
      for(int i=0; i<edge; i++)
      {
         glVertex2i(points[i][0],points[i][1]);
      }
   glEnd();
   glFlush();
```

```
   glColor3f(0,1,0);
   glBegin(GL_LINE_LOOP);
      glVertex2i(200,100);
      glVertex2i(500,100);
      glVertex2i(500,350);
      glVertex2i(200,350);
   glEnd();
   glFlush();

}
int BottomCliping(int e){

float m=0;
int x=0,k=0;
int t[10][2];

   for(int i=0; i<e; i++){
      if(points[i][1] < wymin){

         if(points[i+1][1]  < wymin){

         }
         else if(points[i+1][1] > wymin){
            float x1,x2;
            float y1,y2;
            x1 = points[i][0];
            y1 = points[i][1];
            x2 = points[i+1][0];
            y2 = points[i+1][1];
            x = ((1/((y2-y1)/(x2-x1))) * (wymin - y1) )+ x1;
            t[k][0] = x;
            t[k][1] = wymin;
            k++;

         }

      }
      else if(points[i][1]>wymin){

         if(points[i+1][1] > wymin){
            t[k][0] = points[i][0];
            t[k][1] = points[i][1];
            k++;
         }
         else if(points[i+1][1] < wymin){
            float x1,x2;
            float y1,y2;
```

```
            x1 = points[i][0];
            y1 = points[i][1];
            x2 = points[i+1][0];
            y2 = points[i+1][1];

            x = ((1/((y2-y1)/(x2-x1))) * (wymin - y1) )+ x1;

            t[k][0] = x1;
            t[k][1] = y1;
            k++;
            t[k][0] = x;
            t[k][1] = wymin;
            k++;


         }

      }

   }
   cout<<"k = "<<k;
   for(int i=0; i<10;i++)
   {
      points[i][0] = 0;
      points[i][1] = 0;

   }

   for(int i=0; i<k;i++)
   {
      cout<<"\n"<<t[i][0]<<" "<<t[i][1];
      points[i][0] = t[i][0];
      points[i][1] = t[i][1];

   }
   points[k][0] = points[0][0];
   points[k][1] = points[0][1];
   return k;

}




int TopCliping(int e){
```

```
float m=0;
int x=0,k=0;
int t[10][2];

   for(int i=0; i<e; i++){
      if(points[i][1] > wymax){

         if(points[i+1][1]  > wymax){

         }
         else if(points[i+1][1] < wymax){
            float x1,x2;
            float y1,y2;
            x1 = points[i][0];
            y1 = points[i][1];
            x2 = points[i+1][0];
            y2 = points[i+1][1];
            x = ((1/((y2-y1)/(x2-x1))) * (wymax - y1) )+ x1;
            t[k][0] = x;
            t[k][1] = wymax;
            k++;

         }

      }
      else if(points[i][1]<wymax){

         if(points[i+1][1] < wymax){
            t[k][0] = points[i][0];
            t[k][1] = points[i][1];
            k++;
         }
         else if(points[i+1][1] > wymax){
            float x1,x2;
            float y1,y2;
            x1 = points[i][0];
            y1 = points[i][1];
            x2 = points[i+1][0];
            y2 = points[i+1][1];

            x = ((1/((y2-y1)/(x2-x1))) * (wymax - y1) )+ x1;

            t[k][0] = x1;
            t[k][1] = y1;
            k++;
            t[k][0] = x;
            t[k][1] = wymax;
```

```
            k++;


        }

    }
  }
  cout<<"k = "<<k;
  for(int i=0; i<10;i++)
  {
    points[i][0] = 0;
    points[i][1] = 0;

  }

  for(int i=0; i<k;i++)
  {
    cout<<"\n"<<t[i][0]<<" "<<t[i][1];
    points[i][0] = t[i][0];
    points[i][1] = t[i][1];

  }
  points[k][0] = points[0][0];
  points[k][1] = points[0][1];
  return k;

}

int leftCliping(int e){

float m=0;
int y=0, k = 0;
int t[10][2];
  for(int i=0;i<e;i++)
  {

    if(points[i][0] < wxmin){

      if(points[i+1][0] < wxmin){
        cout<<"\n Test 1";

      }
      else if (points[i+1][0] > wxmin){
        cout<<"\n Test 2";
        float x1,x2;
        float y1,y2;
```

```
            x1 = points[i][0];
            y1 = points[i][1];
            x2 = points[i+1][0];
            y2 = points[i+1][1];
            y = (((y2-y1)/(x2-x1)) * (wxmin - x1) )+ y1;
            t[k][0] = wxmin;
            t[k][1] = y;
            k++;
        }
    }
    else if(points[i][0] > wxmin){

        if(points[i+1][0] > wxmin){

            t[k][0] = points[i][0];
            t[k][1] = points[i][1];
            k++;
        }
        else if(points[i+1][0] < wxmin){


            float x1,x2;
            float y1,y2;
            x1 = points[i][0];
            y1 = points[i][1];
            x2 = points[i+1][0];
            y2 = points[i+1][1];

            y = ((y2-y1)/(x2-x1)*(wxmin - x1)) + y1;

            t[k][0] = x1;
            t[k][1] = y1;
            k++;
            t[k][0] = wxmin;
            t[k][1] = y;
            k++;
        }


    }
}
cout<<"k = "<<k;
for(int i=0; i<10;i++)
{
    points[i][0] = 0;
    points[i][1] = 0;
```

```
    }

    for(int i=0; i<k;i++)
    {
        cout<<"\n"<<t[i][0]<<" "<<t[i][1];
        points[i][0] = t[i][0];
        points[i][1] = t[i][1];

    }
    points[k][0] = points[0][0];
    points[k][1] = points[0][1];
    return k;
}

int RightCliping(int e){

float m=0;
int y=0, k = 0;
int t[10][2];
    for(int i=0;i<e;i++)
    {

        if(points[i][0] > wxmax){

            if(points[i+1][0] > wxmax){


            }
            else if(points[i+1][0] < wxmax){

                float x1,x2;
                float y1,y2;
                x1 = points[i][0];
                y1 = points[i][1];
                x2 = points[i+1][0];
                y2 = points[i+1][1];
                y = (((y2-y1)/(x2-x1)) * (wxmax - x1) )+ y1;
                t[k][0] = wxmax;
                t[k][1] = y;
                k++;
            }

        }
        else if(points[i][0] < wxmax){

            if(points[i+1][0] < wxmax){
```

```
            t[k][0] = points[i][0];
            t[k][1] = points[i][1];
            k++;
        }
        else if(points[i+1][0] > wxmax){

            float x1,x2;
            float y1,y2;
            x1 = points[i][0];
            y1 = points[i][1];
            x2 = points[i+1][0];
            y2 = points[i+1][1];

            y = ((y2-y1)/(x2-x1)*(wxmax - x1)) + y1;
            t[k][0] = x1;
            t[k][1] = y1;
            k++;
            t[k][0] = wxmax;
            t[k][1] = y;
            k++;
        }
      }
   }
   cout<<"k = "<<k;
   for(int i=0; i<10;i++)
   {
      points[i][0] = 0;
      points[i][1] = 0;

   }
   for(int i=0; i<k;i++)
   {
      cout<<"\n"<<t[i][0]<<" "<<t[i][1];
      points[i][0] = t[i][0];
      points[i][1] = t[i][1];

   }
   points[k][0] = points[0][0];
   points[k][1] = points[0][1];
   return k;
}

void P_C(){

   Draw();
}
```

```
void goMenu(int value){

    switch(value){

        case 1:
            edge = leftCliping(edge);
            Draw();
            break;
        case 2:
            edge = RightCliping(edge);
            Draw();
            break;
        case 3:
            edge = TopCliping(edge);
            Draw();
            break;
        case 4:
            edge = BottomCliping(edge);
            Draw();
            break;

    }
    glutPostRedisplay();
}

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


    cout<<"\n Enter No of edges of polygon  ";
    cin>>edge;

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

        cout<<"\n Enter point "<<i<<" x space y ";
        cin>>points[i][0]>>points[i][1];

    }
    points[edge][0] = points[0][0];
    points[edge][1] = points[0][1];
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(640,480);
    glutInitWindowPosition(200,200);
    glutCreateWindow("Polygon Clipping");
    init();
```

```
    glutCreateMenu(goMenu);

        glutAddMenuEntry("Left",1);
        glutAddMenuEntry("Right",2);
        glutAddMenuEntry("Top",3);
        glutAddMenuEntry("Bottom",4);
        glutAttachMenu(GLUT_RIGHT_BUTTON);

       glutDisplayFunc(P_C);

    glutMainLoop();
    return 0;
}
```

Assignment No. 6

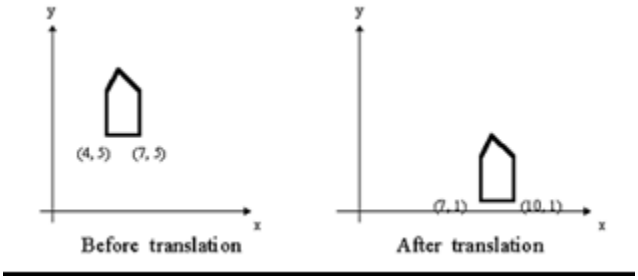
AIM:

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

Prerequisite: Basic primitives of computer graphics and concept of OOP.

Learning Objective: How to transform given polygon in 2D.

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

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:

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' = xcosθ -y sinθ

y' = xsinθ+ ycosθ

or in matrix form:

P' = R • P,

R-rotation matrix.

The matrix for rotating a point about an origin in a 2D plane is defined as:

$$R_\beta = \begin{bmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{bmatrix}$$

Thus the rotation of a 2D vector in a plane is done as follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Formula: X = xcosA - ysinA

Y = xsinA + ycosA,

A is the angle of rotation.

The above formula will rotate the point around the origin.

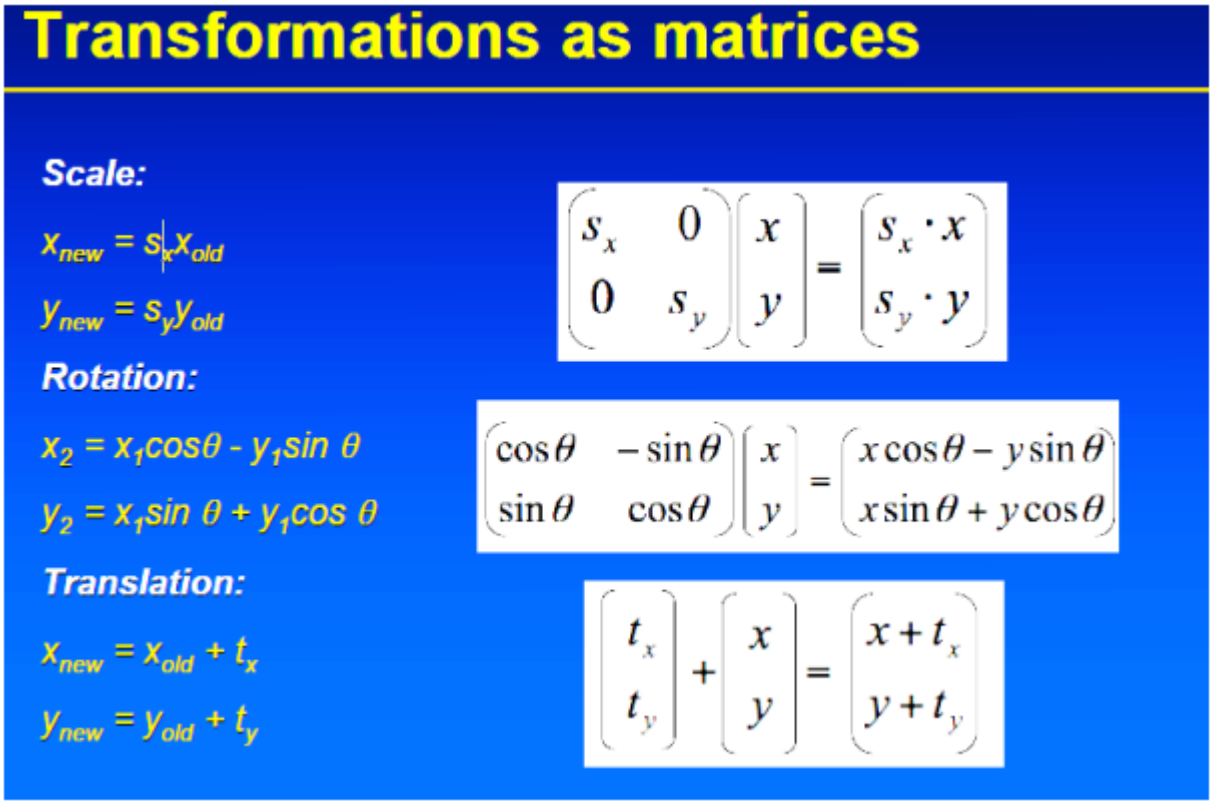
To rotate around a different point, the formula:

X = cx + (x-cx)*cosA - (y-cy)*sinA,

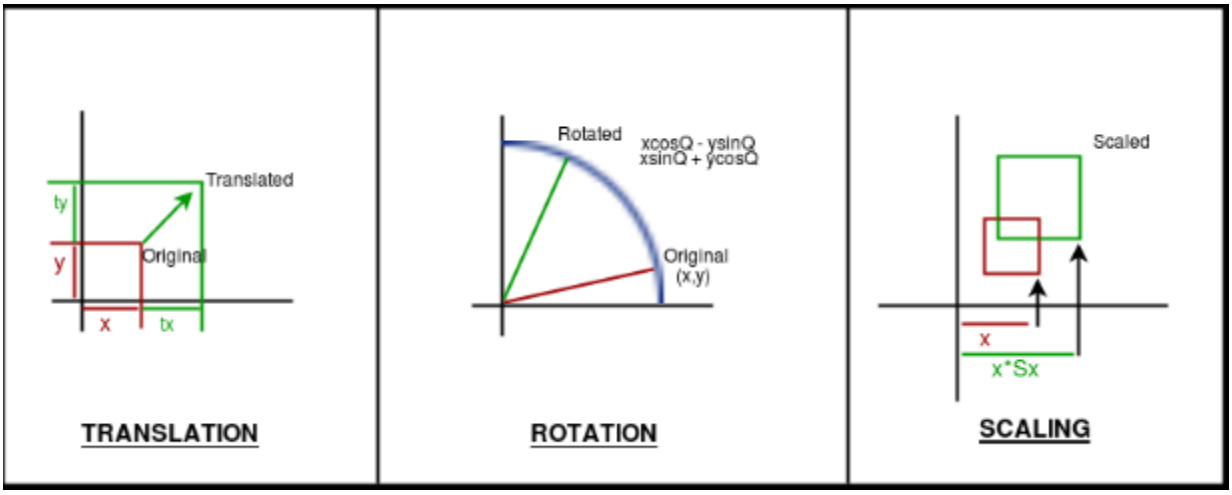
Y = cx + (x-cx)*sinA + (y-cy)*cosA,

cx, cy is centre coordinates,

A is the angle of rotation.



The OpenGL function is **glRotatef (A, x, y, z).**



Reflection: It is a transformation which produces a mirror image of an object. The mirror image can be either about x-axis or y-axis. The object is rotated by180°.

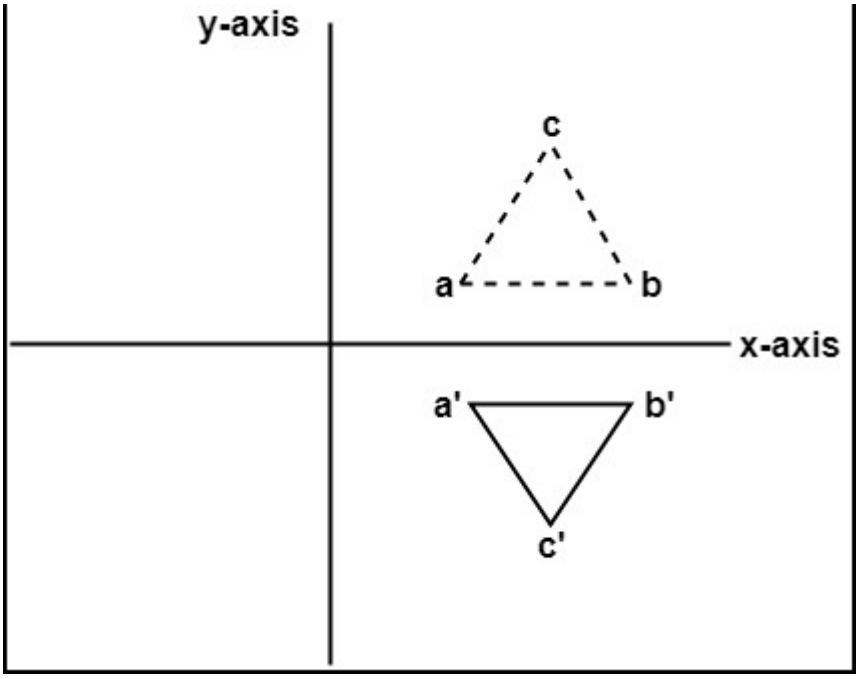
Types of Reflection:

1. Reflection about the x-axis
2. Reflection about the y-axis
3. Reflection about an axis perpendicular to xy plane and passing through the origin
4. Reflection about line y=x

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

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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

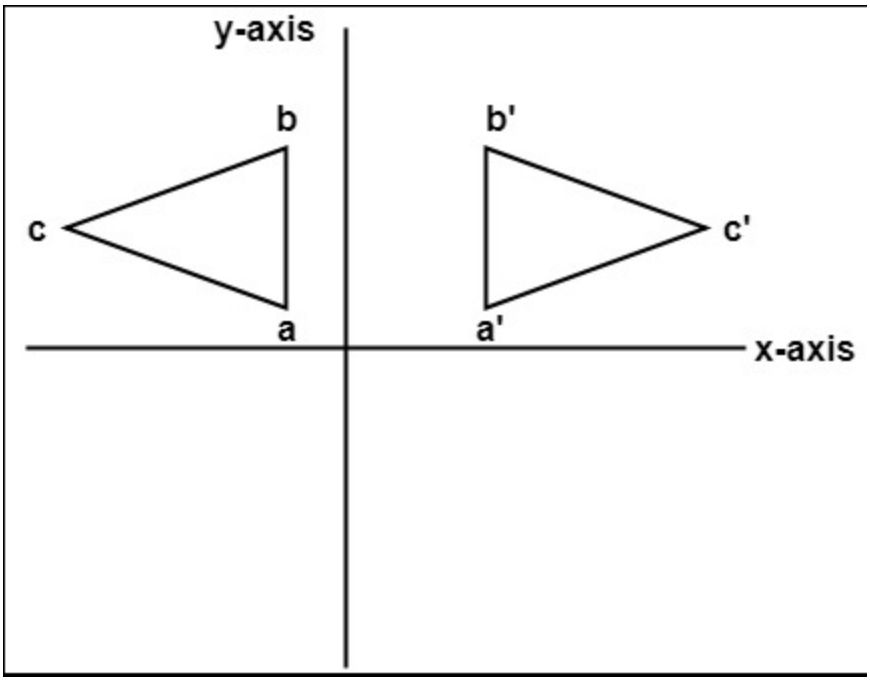


2. Reflection about y-axis: The object can be reflected about y-axis with the help of following transformation matrix

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

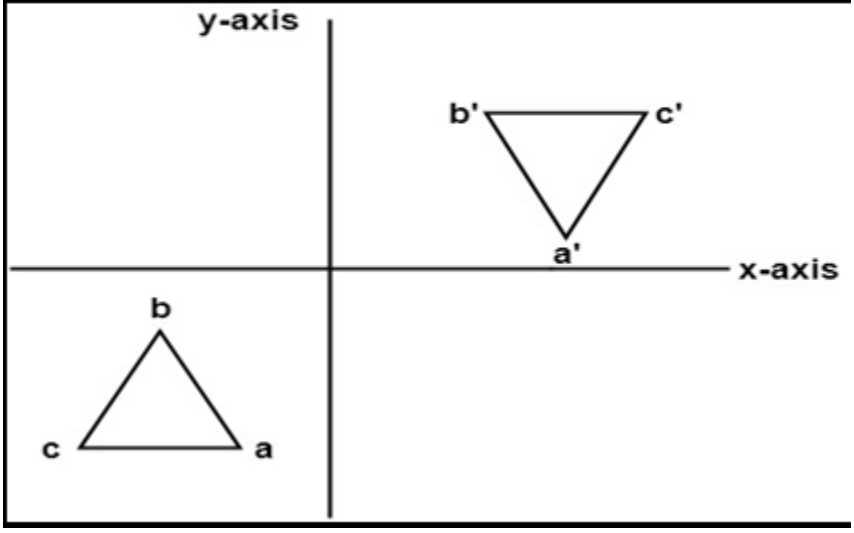
Here the values of x will be reversed, whereas the value of y will remain the same. The object will lie another side of the y-axis.

The following figure shows the reflection about the y-axis



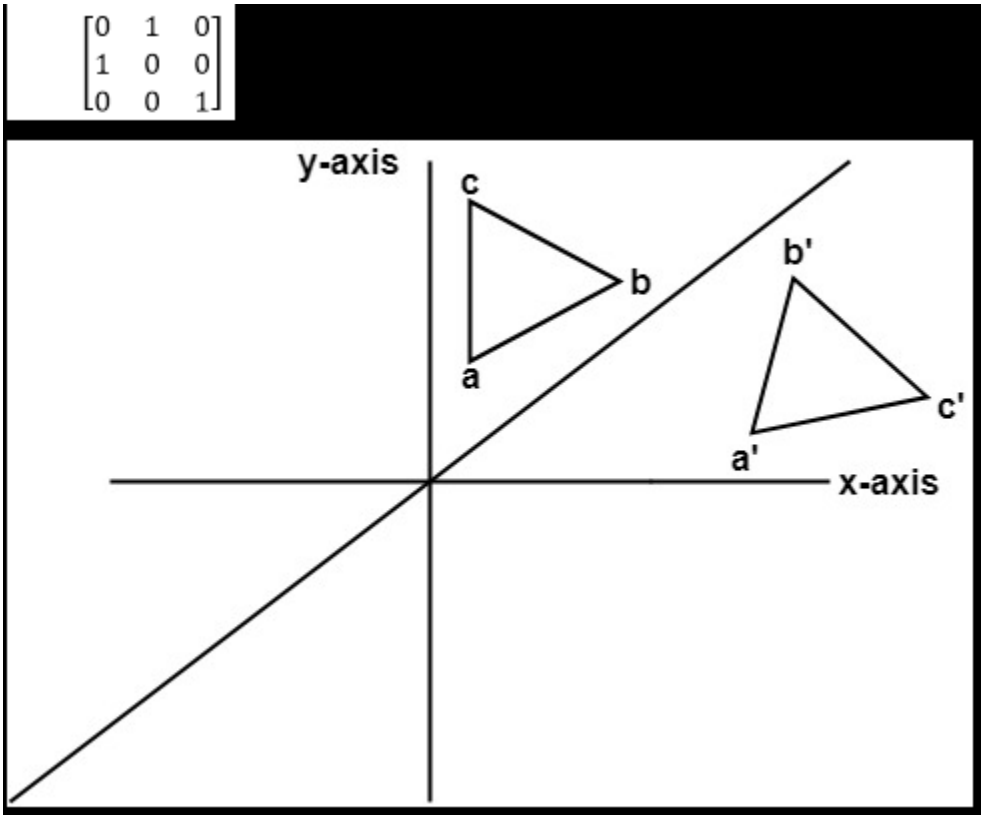
3. Reflection about an axis perpendicular to xy plane and passing through origin:
In the matrix of this transformation is given below

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



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

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



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

Example: A triangle ABC is given. The coordinates of A, B, C are given as

A (3 4)
B (6 4)
C (4 8)

Find reflected position of triangle i.e., to the x-axis.

Conclusion:-

In This way we have studied that how to transform a given polygon in 2D.

CODE:

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

using namespace std;

int edge;
vector<int> xpoint;
vector<int> ypoint;

int ch;

double round(double d){

   return floor(d + 0.5);
}

void init(){
   glClearColor(1.0,1.0,1.0,0.0);
   glMatrixMode(GL_PROJECTION);
   gluOrtho2D(0,640,0,480);
   glClear(GL_COLOR_BUFFER_BIT);
}

void translation(){
   int tx, ty;
   cout<<"\t Enter Tx, Ty \n";
   cin>> tx>> ty;

   //Translate the point
   for(int i=0;i<edge;i++){

      xpoint[i] = xpoint[i] + tx;
      ypoint[i] = ypoint[i] + ty;

   }


   glBegin(GL_POLYGON);
      glColor3f(0,0,1);
```

```
        for(int i=0;i<edge;i++){
            glVertex2i(xpoint[i],ypoint[i]);
        }
    glEnd();
    glFlush();
}

void rotaion(){
    int cx, cy;
    cout<<"\n Enter Ar point x , y ";
    cin >> cx >> cy;

    cx = cx+320;
    cy = cy+240;
    glColor3f(0.0, 1.0, 0.0);
    glBegin(GL_POINTS);
        glVertex2i(cx,cy);
    glEnd();
    glFlush();

    double the;
    cout<<"\n Enter thetha ";
    cin>>the;
    the = the * 3.14/180;

    glColor3f(0,0,1.0);
    glBegin(GL_POLYGON);
        for(int i=0;i<edge;i++){
            glVertex2i(round(((xpoint[i] - cx)*cos(the) - ((ypoint[i]-cy)*sin(the))) + cx),
                round(((xpoint[i] - cx)*sin(the) + ((ypoint[i]-cy)*cos(the))) + cy));
        }
    glEnd();
    glFlush();
}

void scale(){

    glColor3f(1.0,0,0);
    glBegin(GL_POLYGON);
        for(int i=0;i<edge;i++){
            glVertex2i(xpoint[i]-320,ypoint[i]-240);
        }
    glEnd();
```

```
    glFlush();
    cout<<"\n\tIn Scaling whole screen is 1st Qudrant \n";
    int sx, sy;
    cout<<"\t Enter sx, sy \n";
    cin>> sx>> sy;

    //scale the point
    for(int i=0;i<edge;i++){

        xpoint[i] = (xpoint[i]-320) * sx;
        ypoint[i] = (ypoint[i]-240) * sy;
    }

    glColor3f(0,0,1.0);
    glBegin(GL_POLYGON);
        for(int i=0;i<edge;i++){
            glVertex2i(xpoint[i],ypoint[i]);
        }
    glEnd();
    glFlush();
}

void reflection(){
    char reflection;
    cout<<"Enter Reflection Axis \n";
    cin>> reflection;

    if(reflection == 'x' || reflection == 'X'){

        glColor3f(0.0,0.0,1.0);
        glBegin(GL_POLYGON);
            for(int i=0;i<edge;i++){
                glVertex2i(xpoint[i], (ypoint[i] * -1)+480);
            }
        glEnd();
        glFlush();

    }
    else if(reflection == 'y' || reflection == 'Y'){
        glColor3f(0.0,0.0,1.0);
        glBegin(GL_POLYGON);
            for(int i=0;i<edge;i++){
                glVertex2i((xpoint[i] * -1)+640,(ypoint[i]));
```

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

void Draw(){

    if(ch==2 || ch==3 || ch==4){
        glColor3f(1.0,0,0);
        glBegin(GL_LINES);
            glVertex2i(0,240);
            glVertex2i(640,240);
        glEnd();
        glColor3f(1.0,0,0);
        glBegin(GL_LINES);
            glVertex2i(320,0);
            glVertex2i(320,480);
        glEnd();
        glFlush();

        glColor3f(1.0,0,0);
        glBegin(GL_POLYGON);
            for(int i=0;i<edge;i++){
                glVertex2i(xpoint[i],ypoint[i]);
            }
        glEnd();
        glFlush();
    }
    if(ch==1){
        scale();
    }
    else if(ch == 2){
        rotaion();
    }
    else if( ch == 3){
        reflection();
    }
    else if (ch == 4){
        translation();
    }
}
```

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

   cout<<"\n \t Enter 1) Scaling ";
   cout<<"\n \t Enter 2) Rotation about arbitrary point";
   cout<<"\n \t Enter 3) Reflection";
   cout<<"\n \t Enter 4) Translation  \n \t";

   cin>>ch;

   if(ch==1 || ch==2 || ch==3 || ch==4){

      cout<<"Enter No of edges \n";
      cin>> edge;

      int xpointnew, ypointnew;
      cout<<" Enter"<< edge <<" point of polygon \n";
      for(int i=0;i<edge;i++){

         cout<<"Enter "<< i << " Point ";
         cin>>xpointnew>>ypointnew;

         xpoint.push_back(xpointnew+320);
         ypoint.push_back(ypointnew+240);

      }
         glutInit(&argc, argv);
         glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
         glutInitWindowSize(640,480);
         glutInitWindowPosition(200,200);
         glutCreateWindow("2D");
         init();
         glutDisplayFunc(Draw);

      glutMainLoop();
         return 0;
      }
      else{
         cout<<"\n \t Check Input run again";
      return 0;
   }
}
```

Assignment No 7

Assignment Name:

Generate fractal patterns using i) Bezier ii) Koch Curve

Prerequisite: Circle generation

Learning Objective: Generating curve.

Theory:

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 −

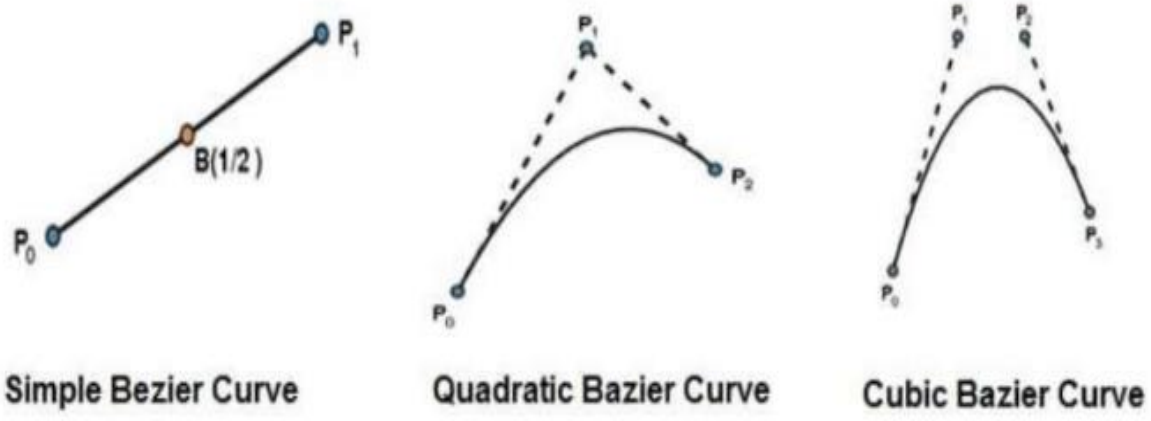
$$\sum k=0nPiBni(t)\sum k=0nPiBin(t)$$

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

$$Bni(t)=(ni)(1-t)n-itiBin(t)=(ni)(1-t)n-iti$$

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

1. They generally follow the shape of the control polygon, which consists of the segments joining the control points.

2. They always pass through the first and last control points.

3. They are contained in the convex hull of their defining control points.

4. 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.

5. A Bezier curve generally follows the shape of the defining polygon.

6. The direction of the tangent vector at the end points is same as that of the vector determined by first and last segments.

7. The convex hull property for a Bezier curve ensures that the polynomial smoothly follows the control points.

8. No straight line intersects a Bezier curve more times than it intersects its control polygon.

9. They are invariant under an affine transformation.

10. Bezier curves exhibit global control means moving a control point alters the shape of the whole curve.

11. A given Beziercurve 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

Fractals are geometric objects. Many real-world objects like ferns are shaped like fractals. Fractals are formed by iterations. Fractals are self-similar.

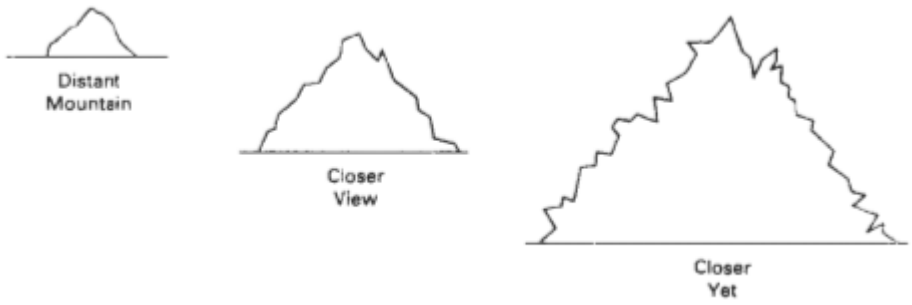
In computer graphics, we use fractal functions to create complex object

The object representations uses 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.

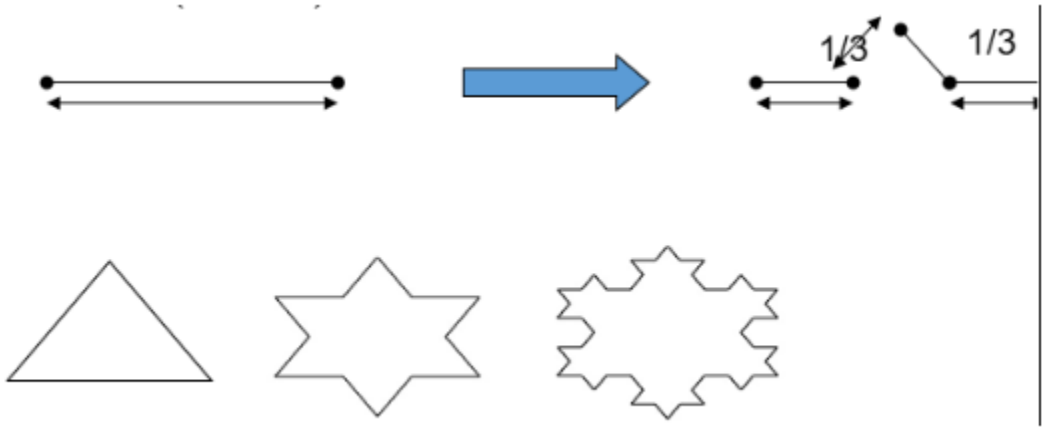
In computer graphics, we use fractal functions to create complex objects



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

Conclusion:

In This way we have studied generating Bezier and Koch Curve.

CODE FOR Bezier Curves

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

using namespace std;

int x[4],y[4];

void init(){
   glClearColor(1.0,1.0,1.0,0.0);
   glMatrixMode(GL_PROJECTION);
   gluOrtho2D(0,640,0,480);
   glClear(GL_COLOR_BUFFER_BIT);
}
void putpixel(double xt,double yt )
{
   glColor3f(1,0,0);
   glBegin(GL_POINTS);
         glVertex2d(xt,yt);
      glEnd();
      glFlush();
}

void Algorithm(){

   glColor3f(0,1,0);
   glBegin(GL_LINES);
      glVertex2i(x[0],y[0]);
      glVertex2i(x[1],y[1]);
      glVertex2i(x[1],y[1]);
      glVertex2i(x[2],y[2]);
      glVertex2i(x[2],y[2]);
      glVertex2i(x[3],y[3]);
   glEnd();
   glFlush();
```

```
   double t;
   for (t = 0.0; t < 1.0; t += 0.0005)
   {
      double xt = pow(1-t, 3) * x[0] + 3 * t * pow(1-t, 2) * x[1] + 3 * pow(t, 2) * (1-t) * x[2] + pow(t, 3) *
x[3];
      double yt = pow(1-t, 3) * y[0] + 3 * t * pow(1-t, 2) * y[1] + 3 * pow(t, 2) * (1-t) * y[2] + pow(t, 3) *
y[3];
      putpixel(xt, yt);
   }

}

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

   cout<<"\n \t Enter The Four Points x space y ";
   for(int i=0;i<4;i++){
      cout<<"\n \t Enter x and y for "<<i<<" = ";
      cin>>x[i]>>y[i];
   }

   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(640,480);
   glutInitWindowPosition(200,200);
   glutCreateWindow("Bezier 4 point");
   init();
   glutDisplayFunc(Algorithm);

   glutMainLoop();
   return 0;
}
```

CODE FOR Koch Curve

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

using namespace std;


double x,y,len,angle;
int it;

void init(){
   glClearColor(1.0,1.0,1.0,0.0);
```

```
   glMatrixMode(GL_PROJECTION);
   gluOrtho2D(0,640,0,480);
   glClear(GL_COLOR_BUFFER_BIT);
}

void line1(int x1, int y11, int x2,int y2){

   glColor3f(0,1,0);
   glBegin(GL_LINES);
      glVertex2i(x1,y11);
      glVertex2i(x2,y2);
   glEnd();
   glFlush();
}
void k_curve(double x, double y, double len, double angle, int it){

   if(it>0){

      len /=3;
      k_curve(x,y,len,angle,(it-1));
      x += (len * cosl(angle * (M_PI)/180));
      y += (len * sinl(angle * (M_PI)/180));
      k_curve(x,y, len, angle+60,(it-1));
      x += (len * cosl((angle + 60) * (M_PI)/180));
      y += (len * sinl((angle + 60) * (M_PI)/180));
      k_curve(x,y, len, angle-60,(it-1));
      x += (len * cosl((angle - 60) * (M_PI)/180));
      y += (len * sinl((angle - 60) * (M_PI)/180));
      k_curve(x,y,len,angle,(it-1));
   }
   else
   {
      line1(x,y,(int)(x + len * cosl(angle * (M_PI)/180) + 0.5),(int)(y + len * sinl(angle * (M_PI)/180) +
0.5));
   }
}

void Algorithm(){
   k_curve(x,y,len,angle,it);

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

   cout<<"\n Enter Starting Point x space y ";
   cin>>x>>y;

   cout <<"\n Lenght of line  and space angle of line";
```

```
    cin>>len>>angle;

    cout<<"\n No. of ittration ";
    cin>>it;

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(640,480);
    glutInitWindowPosition(200,200);
    glutCreateWindow("Koch");
    init();
    glutDisplayFunc(Algorithm);

    glutMainLoop();
    return 0;
}
```

Assignment No 8

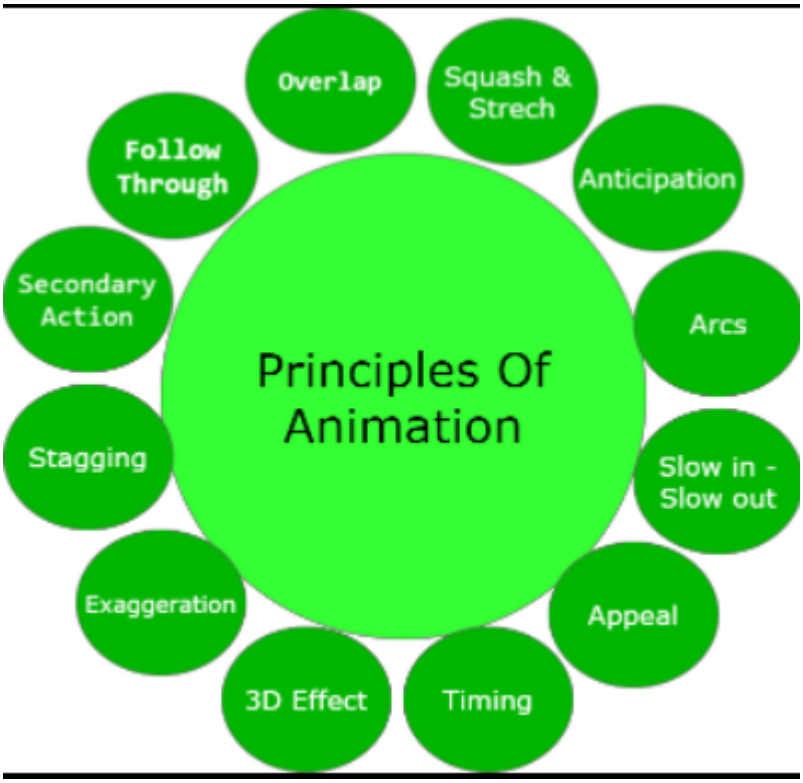
Aim:

Implement animation principles for any object

Theory:

Animation is defined as a series of images rapidly changing to create an illusion of movement. We replace the previous image with a new image which is a little bit shifted. Animation Industry is having a huge market nowadays. To make an efficacious animation there are some principles to be followed.

Principle of Animation:



There are 12 major principles for an effective and easy to communicate animation.

1. **Squash and Strech:**
   This principle works over the physical properties that are expected to change in any process. Ensuring proper squash and stretch makes our animation more convincing.

For Example: When we drop a ball from height, there is a change in its physical property. When the ball touches the surface, it bends slightly which should be depicted in animation properly.

2. **Anticipation:**
   Anticipation works on action.Animation has broadly divided into 3 phases:

3. **1.** Preparation phase

4. **2.** Movement phase

3. Finish

In Anticipation, we make our audience prepare for action. It helps to make our animation look more realistic.

For Example: Before hitting the ball through the bat, the actions of batsman comes under anticipation. This are those actions in which the batsman prepares for hitting the ball.

5. **Arcs:**
   In Reality, humans and animals move in arcs. Introducing the concept of arcs will increase the realism. This principle of animation helps us to implement the realism through projectile motion also.

For Example, The movement of the hand of bowler comes under projectile motion while doing bowling.

6. **Slow in-Slow out:**
   While performing animation, one should always keep in mind that in reality object takes time to accelerate and slow down. To make our animation look realistic, we should always focus on its slow in and slow out proportion.

For Example, It takes time for a vehicle to accelerate when it is started and similarly when it stops it takes time.

7. **Appeal:**
   Animation should be appealing to the audience and must be easy to understand. The syntax or font style used should be easily understood and appealing to the audience. Lack of symmetry and complicated design of character should be avoided.

8. **Timing:**

Velocity with which object is moving effects animation a lot. The speed should be handled with care in case of animation.

For Example, An fast-moving object can show an energetic person while a slow-moving object can symbolize a lethargic person. The number of frames used in a slowly moving object is less as compared to the fast-moving object.

9. **3D Effect:**
   By giving 3D effects we can make our animation more convincing and effective. In 3D Effect, we convert our object in a 3-dimensional plane i.e., X-Y-Z plane which improves the realism of the object.

For Example, a square can give a 2D effect but cube can give a 3D effect which appears more realistic.

10. **8. Exaggeration:**
    Exaggeration deals with the physical features and emotions. In Animation, we represent emotions and feeling in exaggerated form to make it more realistic. If there is more than one element in a scene then it is necessary to make a balance between various exaggerated elements to avoid conflicts.

11. **Stagging:**
    Stagging is defined as the presentation of the primary idea, mood or action. It should always be in presentable and easy to manner. The purpose of defining principle is to avoid unnecessary details and focus on important features only. The primary idea should always be clear and unambiguous.

12. **Secondary Action:**
    Secondary actions are more important than primary action as they represent the animation as a whole. Secondary actions support the primary or main idea.

For Example, A person drinking a hot tea, then his facial expressions, movement of hands, etc comes under the secondary actions.

13. **Follow Through:**
    It refers to the action which continues to move even after the completion of action. This type of action helps in the generation of more idealistic animations.

For Example: Even after throwing a ball, the movement of hands continues.

14. **Overlap:**
    It deals with the nature in which before ending the first action, the second action starts.

For Example: Consider a situation when we are drinking Tea from the right hand and holding a sandwich in the left hand. While drinking a tea, our left-hand start showing movement towards the mouth which shows the interference of the second action before the end of the first action.

Conclusion:

Hence, In this way we have studied different animation principles.

CODE:

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

using namespace std;


int x=0;
int flag=0;
void init(){
    glClearColor(1.0,1.0,1.0,0.0);
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(0,640,0,480);
}

void object1(){

    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(1,0,0);
    glBegin(GL_POLYGON);
        glVertex2i(x,220);
        glVertex2i(x+40,220);
        glVertex2i(x+40,260);
        glVertex2i(x,260);
    glEnd();

    glutSwapBuffers();

}

void timer(int){

    glutPostRedisplay();
    glutTimerFunc(1000/60,timer,0);

    if(flag == 0){
        x = x+3;
    }
    if(flag == 1){
        x = x-3;
    }
    if(x==600){
        flag = 1;
```

```
    }
    if(x == 0){
        flag = 0;
    }

}
int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(640,480);
    glutInitWindowPosition(200,200);
    glutCreateWindow("Animation");
    init();
    glutDisplayFunc(object1);
    glutTimerFunc(1000,timer,0);
    glutMainLoop();
    return 0;
}
```