Savitribai Phule Pune University

CG LAB MANUAL

(S.E. IT)

Prepared By

Prof. Kapil Wagh

Asst Prof (NMIET)

Nutan Maharashtra Institute of Engineering and Technology

Assignment 1: *Study of Install and explore the OpenGL*

AIM: To learn all the basic functions of openGL. Also install and configure opengl.

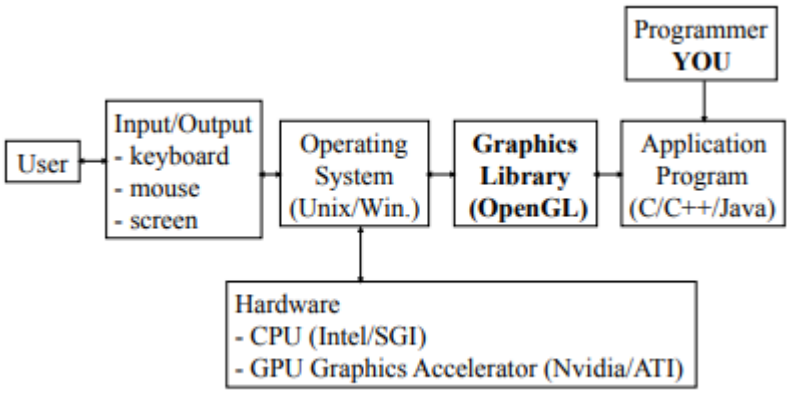
Objective:

- To get familiar with the openGL
- Use of various in built openGL functions
- To use Various functions available in glu and glut libraries
- To know about displaying of any object onto screen(polygon)

Theory:

What is OpenGL?

Application programmers interface (API) for 2D/3D graphics



Why OpenGL?

Open standard for graphics based applications

- originally developed by SGI as 'GL' graphics library
- Released as an open-standard
- Widely used for interactive graphics applications Animation/VR/Games

Platform independent library of low-level graphics functions

- Approx. 250 distinct commands for 3D graphics
- Hardware accelerated for particular platform
- Very fast 3D rendering

What OpenGL does'nt do:

No functions dependent on a particular platform No high-level functions for object description etc.
Utility libraries to support platform dependent functions GLU/GLUT

A Simple Example OpenGL Program: Square

```
#include “gl/gl.h” /* include functions from gl library */
main()
{
/* call my function to initialise a draw window here */
/* OpenGL code to draw a square */
glClearColor(1.0,0.0,0.0,0.0); /* set window to red (r,g,b,a) */
glClear(GL_COLOR_BUFFER_BIT); /* clear window */
glOrtho(0.0,1.0, 0.0,1.0,-1.0,1.0); /* setup 3d coordinate space */
glColor3f(0.0,0.0,1.0); /* set drawing colour blue (r,g,b) */
glBegin(GL_POLYGON); /* specify a polygon */
glVertex3f(0.25,0.25,0.0) /* vertex 1 (x,y,z) */
glVertex3f(0.75,0.25,0.0) /* vertex 2 (x,y,z) */
glVertex3f(0.75,0.75,0.0) /* vertex 3 (x,y,z) */
glVertex3f(0.25,0.75,0.0) /* vertex 4 (x,y,z) */
glEnd();

glFlush(); /* draw all objects */
/* call myfunction to update window and handle events */
```

Result of Simple Example Code

OpenGL Syntax

All OpenGL commands have the prefix ‘gl’

glClear()

glColor3f()

glVertex3f()

Constants are defined with prefix 'GL' & use '_' to separate words

GL_COLOR_BUFFER_BIT

Type information is appended to the end of the command

glColor3f(r,g,b) - a colour of 3 floating point components

glVertex3f(x,y,z) - a vertex with 3 floating point coordinates

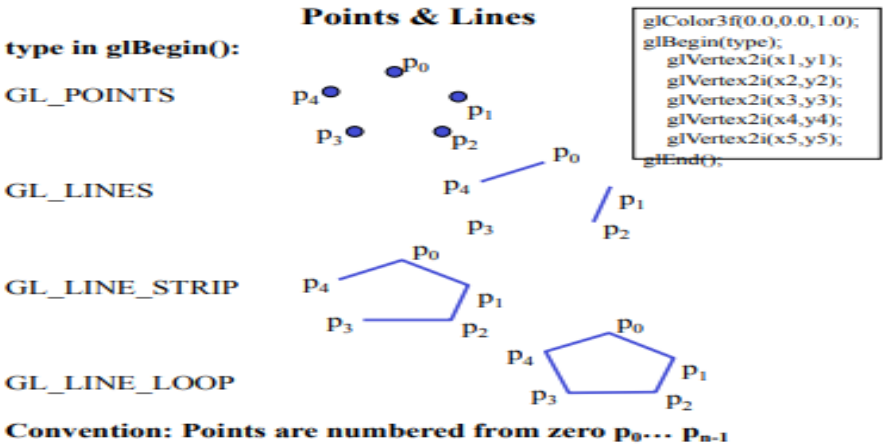
glVertex2f(x,y) - a vertex with 2 floating point coordinates

Different versions of the same function exist for different types

glVertex2i(p,q) - vertex with 2 integer coordinates

Suffix	Type	OpenGL Type	C type
b	8-bit integer	GLbyte	short
i	32-bit integer	GLint	int or long
f	32-bit real	GLfloat	float
d	64-bit real	GLdouble	double
ui	32-bit unsigned int	GLuint	unsigned int
+ others			

Use OpenGL Types to avoid problems



Convention: Points are numbered from zero $p_0 \cdots p_{n-1}$

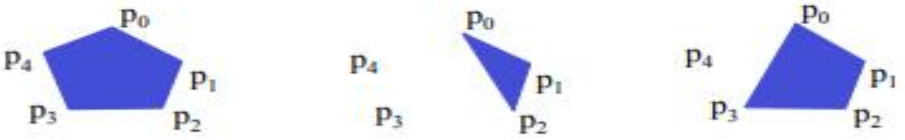
Polygons

Must be: **'Flat'** All vertices lie in a plane
'Simple' Polygon edges do not intersect
'Convex' All point are on one side of any edge

Allows for fast polygon rendering algorithm implement in hardware

Type for glBegin():

GL_POLYGON GL_TRIANGLES GL_QUADS



Convention: Polygons are specified in anticlockwise vertex order

Task1: Configuring opengl with Visual Studio

Step1. Project Properties -> C/C++ ->General->Additional Include Directories->edit->

<path : C:\Users\OS LAB\Documents\Glew and Glut\freeglut\include\GL;C:\Users\OS LAB\Documents\Glew and Glut\glew-1.11.0\include\GL>

Step2. Project Properties -> Linker -> Input -> edit-> type : freeglut.lib ->enter->glew32.lib

Step3. Project Properties ->Linker->General->edit-> <path : C:\Users\OS LAB\Documents\Glew and Glut\freeglut\lib;C:\Users\OS LAB\Documents\Glew and Glut\glew-1.11.0\lib>

Step4. Copy freeglut.dll and glew32.dll files from Glew and Glut folder to project folder

Assignment 2: *To implement DDA and Bresenham's line drawing algorithm for i>Dashed line ii>Dotted Line iii>Simple Line iv> Solid Line in all quadrants. Divide Screen in quadrants.*

Objective:

- To be able to implement dda and bresenhams line drawing algorithm
- To be able to Divide screen in quadrants.
- To be able to display line each quadrants
- To be able to display all types of lines (dashed, dotted, simple, solid)

DDA (Digital Differential Analyzer):

Scan conversion line algorithm based on Δ x or Δ y. Sample the line at unit interval in one co-ordinate and determine corresponding integer value. Faster method than y= mx+ b using but may specifies inaccurate pixel section. Rounding off operations and floating point arithmetic operations are time consuming.DDA is used in drawing straight line to form a line, triangle or polygon in computer graphics. DDA analyzes samples along the line at regular interval of one coordinate as the integer and for the other coordinate it rounds off the integer that is nearest to the line. Therefore as the line progresses it scan the first integer coordinate and round the second to nearest integer. Therefore a line drawn using DDA for x coordinate it will be x0 to x1 but for y coordinate it will be y = ax+ b and to draw function it will be fn(x, y rounded off).

DDA LINE DRAWING ALGORITHM

1. Start.

2. Declare variables x,y,x1,y1,x2,y2,k,dx,dy,s,xi,yi and also declare gdriver=DETECT,gmode.

3. Initialise the graphic mode with the path location in TC folder.

4. Input the two line end-points and store the left end-points in (x1,y1).

5.	Load (x1,y1) into the frame buffer;that is,plot the first point. put x=x1,y=y1.
6.	Calculate dx=x2-x1 and dy=y2-y1.
7.	If abs(dx) > abs(dy), then steps=abs(dx).
8.	Otherwise steps= abs(dy).
9.	Then xi=dx/steps and yi=dy/steps.
10.	Start from k=0 and continuing till k<steps, the points will be
i.	x=x+xi.
ii.	y=y+yi.
11.	Plot pixels using putpixel at points (x,y) in specified colour.
12.	Close Graph.
13.	Stop.

BRESENHAM'S LINE DRAWING ALGORITHM

This was developed by J.E.Bresenham in 1962 and it is much accurate and much more efficient than DDA. An algorithm which determines which order to form a close approximation to a straight line between two given points. It is commonly used to draw lines on a computer screen, as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures. It scans the coordinates but instead of rounding

them off it takes the incremental value in account by adding or subtracting and therefore can be used for drawing circle and curves. Therefore if a line is to be drawn between two points x and y then next coordinates will be(xa+1, ya) and (xa+1, ya+1) where a is the incremental value of the next coordinates and difference between these two will be calculated by subtracting or adding the equations formed by them.

- **Step 1** : Except the two end points of Line from User.
- **Step 2** : Calculate the slope(m) of the required Line.
- **Step 3** : Identify the value of slope(m).
- **Step 3.1** : If slope(m) is Less than 1 i.e: m < 1
  * **Step 3.1.1** : Calculate the constants dx, dy, 2dy, and (2dy – 2dx) and get the first value for the decision parameter as -
  * p0 = 2dy − dx
  * **Step 3.1.2** : At each Xk along the line, starting at k = 0, perform the following test −
  * If pk < 0, the next point to plot is (xk + 1,yk) and
  pk+1 = pk + 2dy
  else
  * plot (xk,yk + 1)
  * pk+1 = pk + 2dy − 2dx
  * **Step 3.1.3** : Repeat step 4(dx - 1) times.
- **Step 3.2** : If slope(m) is greater than or equal to 1 i.e: m >= 1
  * **Step 3.2.1** : Calculate the constants dx, dy, 2dy, and (2dy – 2dx) and get the first value for the decision parameter as -
  * p0 = 2dx − dy
  * **Step 3.2.2** : At each Yk along the line, starting at k = 0, perform the following test −
  * If pk < 0, the next point to plot is (xk,yk + 1) and
  pk+1 = pk + 2dx
  else
  * plot (xk + 1,yk)
  * pk+1 = pk + 2dx − 2dy
  * **Step 3.2.3** : Repeat step 4(dy - 1) times.
- **Step 3.3** : Exit.

Algorithm to change line style in any dda/bresenham

A. To print dashed line

```
if (j % 7 == 0)
{
x = x + 3;
y = y + 3;
glBegin(GL_POINTS);
glVertex2i(x, y);
glEnd();
}
else
{
glBegin(GL_POINTS);
glVertex2i(x, y);
glEnd();
}
j++;
```

B. To print dotted line

```
x = x + 1; // to show dotted line we skipped alternate points
y = y + 1; // to show dotted line we skipped alternate points
```

C. Simle Line

Use dda program as it is

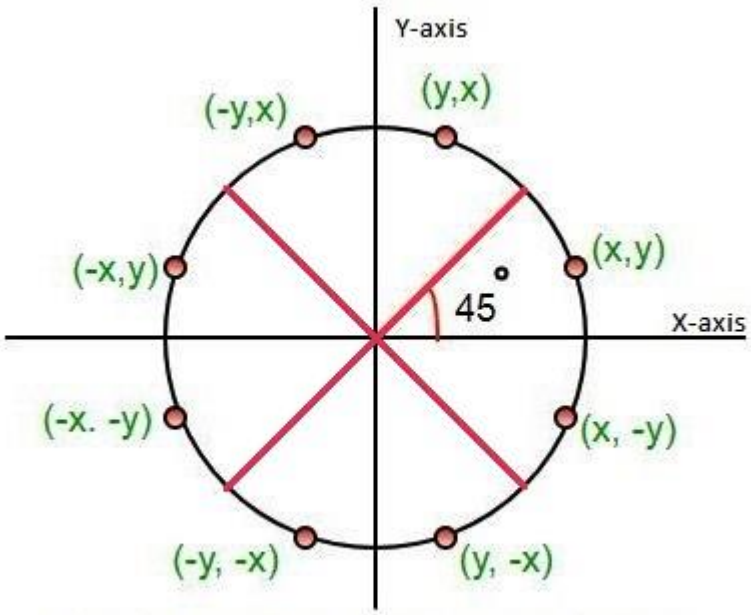
D. Print neighbour points of each point

Assignment 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.*

Bresenham's Circle Drawing Algorithm is a circle drawing algorithm that selects the nearest pixel position to complete the arc. The unique part of this algorithm is that is uses only integer arithmetic which makes it, significantly, faster than other algorithms using floating point arithmetic in classical processors.

As per Eight way symmetry property of circle, circle can be divided into **8 octants** each of **45-degrees**.
The Algorithm calculate the location of pixels in the first octant of 45 degrees and extends it to the other 7 octants. For every pixel (x, y), the algorithm draw a pixel in each of the 8 octants of the circle as shown below :

Assumption : Center of Cirle is Origin.
Following image illustrates the 8 octants with corresponding pixel:



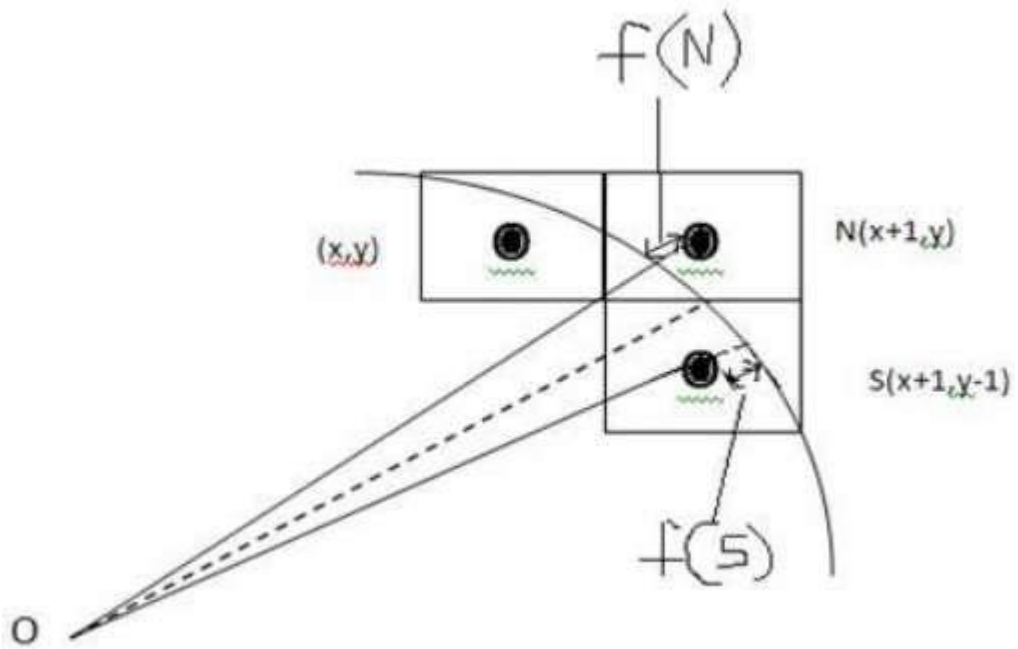
For each pixel (x,y) all possible pixels in 8 octants

Point (x, y) is in the first octant and is on the circle.

To calculate the next pixel location, can be either:

- N (x+1, y)

- S (x+1, y-1)



It is decided by using the decision parameter d as:

- If **d <= 0**, then **N(x+1, y)** is to be chosen as next pixel.
- If **d > 0**, then **S(x+1, y-1)** is to be chosen as the next pixel.

Draw the circle for a given radius 'r' and centre (xc, yc) starting from (0, r) and move in first quadrant till x=y (i.e. 45 degree) ,first octant.

Initial conditions :

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

Steps:

- **Step 1** : Set initial values of (xc, yc) and (x, y)
- **Step 2** : Calculate decision parameter d to d = 3 – (2 * r).
- **Step 3** : call displayBresenhmCircle(int xc, int yc, int x, int y) method to display initial(0,r) point.
- **Step 4** : Repeat steps 5 to 8 until x < = y
- **Step 5** : Increment value of x.
- **Step 6** : If d < 0, set d = d + (4*x) + 6

- **Step 7** : Else, set d = d + 4 * (x – y) + 10 and decrement y by 1.
- **Step 8** : call displayBresenhmCircle(int xc, int yc, int x, int y) method.
- **Step 9** : Exit.

Assignment 4: . *Implement the following polygon filling methods (1 week, 2 hrs) i) Flood fill / Seed fill ii) Boundary fill using mouse click, keyboard interface and menu driven programming*

Boundary Fill

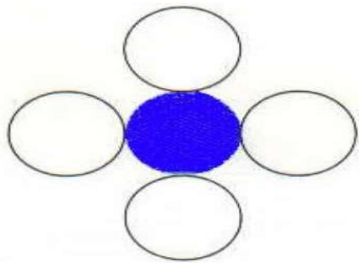
It follows an approach where the region filling begins from some extent residing inside the region and paint the inside towards the boundary. In case the boundary contains single colour the fill algorithm continues within the outward direction pixel by pixel until the boundary colour is encountered. The boundary-fill algorithm is often mainly implemented within the interactive painting packages, where the inside points are easily chosen. The functioning of the boundary-fill starts by accepting the coordinates of an indoor point (x, y), a boundary colour and fill colour becomes the input. Beginning from the (x, y) the method checks neighbouring locations to spot whether or not they are a part of the boundary colour. If they're not from the boundary colour, then they're painted with the fill colour, and their adjacent pixels are tested against the condition. The process ends when all the pixels up until the boundary colour for the world are checked.

Flood Fill

Flood fill algorithm is also known as a seed fill algorithm. It determines the area which is connected to a given node in a multi-dimensional array. This algorithm works by filling or recolouring a selected area containing different colours at the inside portion and therefore the boundary of the image. It is often illustrated by a picture having a neighbourhood bordered by various distinct colour regions. To paint such regions, we will replace a specific interior colour instead of discovering a boundary colour value. This is the rationale the approach is understood because of the flood-fill algorithm. Now, there are two methods which will be used for creating endless boundary by connecting pixels – 4-connected and 8-connected approach. In the 4-connected method, the pixel can have at maximum four neighbours that are positioned at the proper, left, above and below the present pixel. On the contrary, in the 8-connected method, it can have eight, and the neighbouring positions are checked against the four diagonal pixels. So, any of the 2 methods are often wont to repaint the inside points.

4-Connected Polygon

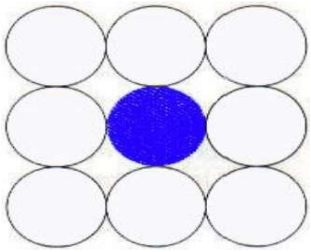
In this technique 4-connected pixels are used as shown in the figure. We are putting the pixels above, below, to the right, and to the left side of the current pixels and this process will continue until we find a boundary with different color.



8-Connected Polygon

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

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



Flood-fill Algorithm	Boundary-fill Algorithm
It can process the image containing more than one boundary colours.	It can only process the image containing single boundary colour.
Flood-fill algorithm is comparatively slower than the Boundary-fill algorithm.	Boundary-fill algorithm is faster than the Flood-fill algorithm.
In Flood-fill algorithm a random colour can be used to paint the interior portion then the old one is replaced with a new one.	In Boundary-fill algorithm Interior points are painted by continuously searching for the boundary colour.
It requires huge amount of memory.	Memory consumption is relatively low in Boundary-fill algorithm.
Flood-fill algorithms are simple and efficient.	The complexity of Bounndary-fill algorithm is high.

Flood Fill Algorithm

```
void floodfill4(int x, int y, float oldcolor[3], float newcolor[3])
{
   float color[3];
   getPixel(x, y, color);
   if (color[0] == oldcolor[0] && (color[1]) == oldcolor[1] && (color[2]) == oldcolor[2])
   {
      setPixel(x, y, newcolor);
      floodfill4(x + 1, y, oldcolor, newcolor);
      floodfill4(x - 1, y, oldcolor, newcolor);
      floodfill4(x, y + 1, oldcolor, newcolor);
      floodfill4(x, y - 1, oldcolor, newcolor);
   }
}
```

Boundary Fill Algorithm

```
void boundaryFill4(int x, int y, float fillColor[3], float borderColor[3])
 {
     float interiorColor[3];

     getPixel(x, y, interiorColor);

     if ((interiorColor[0] != borderColor[0] && interiorColor[1] != borderColor[1] &&
          interiorColor[2] != borderColor[2]) && (interiorColor[0] != fillColor[0] &&
           interiorColor[1] != fillColor[1] && interiorColor[2] != fillColor[2]))
      {
         setPixel(x, y, fillColor);
         boundaryFill4(x + 1, y, fillColor, borderColor);
         boundaryFill4(x - 1, y, fillColor, borderColor);
```

```
        boundaryFill4(x, y - 1, fillColor, borderColor);
        boundaryFill4(x, y + 1, fillColor, borderColor);
    }
}
```

Assignment 5: *Implement Cohen Suterland polygon clipping method to clip the polygon with respect the viewport and window. Use mouse click, keyboard interface*

A polygon can be clipped by processing its boundary as a whole against each window edge. This is achieved by processing all polygon vertices against each clip rectangle boundary in turn. beginning with the original set of polygon vertices, we could first clip the polygon against the left rectangle boundary to produce a new sequence of vertices. The new set of vertices could then be successively passed to a right boundary clipper, a top boundary clipper and a bottom boundary clipper, as shown in figure (l). At each step a new set of polygon vertices is generated and passed to the next window boundary clipper. This is the fundamental idea used in the Sutherland - Hodgeman algorithm.

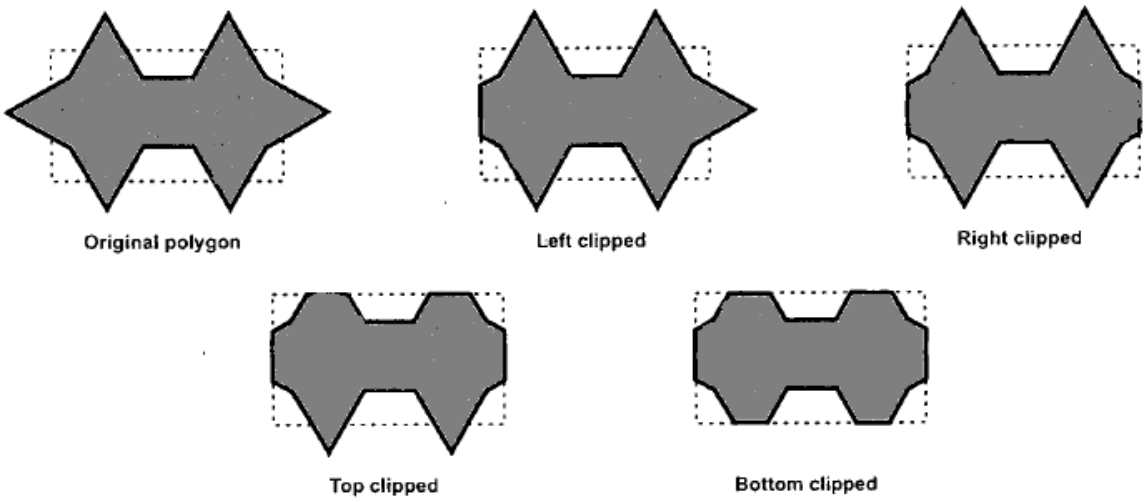


Fig. (l) Clipping a polygon against successive window boundaries

The output of the algorithm is a list of polygon vertices all of which are on the visible side of a clipping plane. Such each edge of the polygon is individually compared with the clipping plane. This is achieved by processing two vertices of each edge of the polygon around the clipping boundary or plane. This results in four possible

relationships between the edge and the clipping boundary or Plane. (See Fig. m).

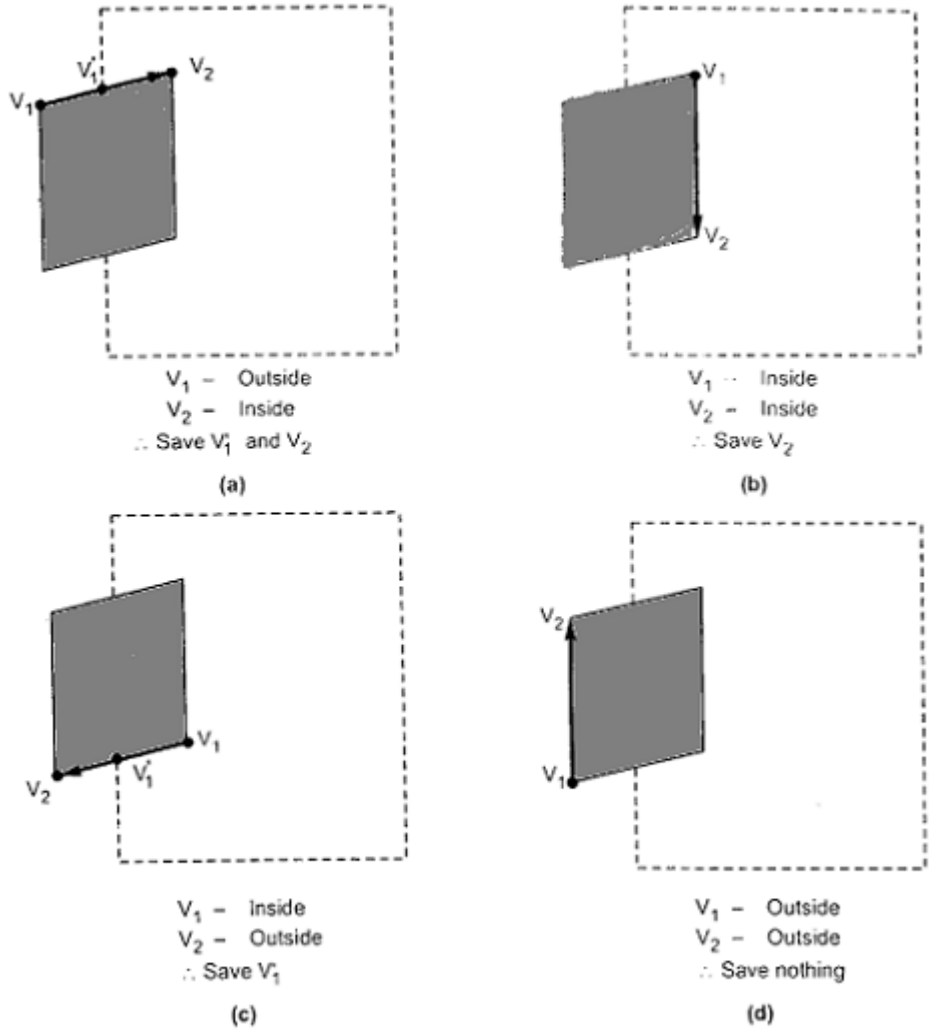


Fig. (m) Processing of edges of the polygon against the left window boundary

1. If the first vertex of the edge is outside the window boundary and the second vertex of the edge is inside then the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list (See Fig. m (a)).
2. If both vertices of the edge are inside the window boundary, only the second vertex is added to the output vertex list. (See Fig. m (b)).
3. If the first vertex of the edge is inside the window boundary and the second vertex of the edge is outside, only the edge intersection with the window boundary is added to the output vertex list. (See Fig. m (c)).
4. If both vertices of the edge are outside the window boundary, nothing is added to the output list. (See Fig. m (d)).

Once all vertices are processed for one clip window boundary, the output list of vertices is clipped against the next window boundary. Going through above four cases we can realize that there are two key processes in this algorithm.

1. Determining the visibility of a point or vertex (Inside - Outside test) and
2. Determining the intersection of the polygon edge and the clipping plane.

One way of determining the visibility of a point or vertex is described here. Consider that two points A and B define the window boundary and point under consideration is V, then these three points define a plane. Two vectors which lie in that plane are AB and AV. If this plane is considered in the xy plane, then the vector cross product AV x AB has only a component given by

$$(x_V - x_A)\,(y_B - y_A) - (y_V - y_A)\,(x_B - x_A)$$

The sign of the z component decides the position of Point V with respect to window boundary.

If z is:

Positive - Point is on the right side of the window boundary.

Zero - Point is on the window boundary.

Negative - Point is on the left side of the window boundary.

Sutherland-Hodgeman Polygon Clipping Algorithm:-

1. Read coordinates of all vertices of the Polygon.
2. Read coordinates of the dipping window
3. Consider the left edge of the window
4. Compare the vertices of each edge of the polygon, individually with the clipping plane.
5. Save the resulting intersections and vertices in the new list of vertices according to four possible relationships between the edge and the clipping boundary.
6. Repeat the steps 4 and 5 for remaining edges or the clipping window. Each time the resultant list of vertices is successively passed to process the next edge of the clipping window.
7. Stop.
8. Example :- For a polygon and clipping window shown in figure below give the list of vertices after each boundary clipping.

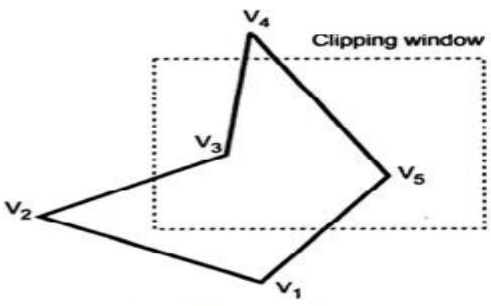


Fig. Before Clipping

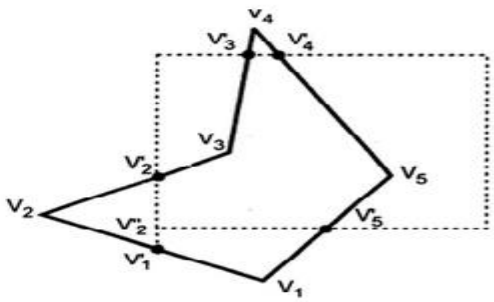


Fig. After Boundary Clipping

Solution:- Original polygon vertices are V1, V2, V3, V4, and V5. After clipping each boundary the new vertices are as shown in figure above.

After left clipping	:	$V_1, V_1', V_2', V_3, V_4, V_5$
After right clipping	:	$V_1, V_1', V_2', V_3, V_4, V_5$
After top clipping	:	$V_1, V_1', V_2', V_3, V_3', V_4', V_5$
After bottom clipping	:	$V_2'', V_2', V_3, V_3', V_4', V_5, V_5'$

Assignment 6: *Implement following 2D transformations on the object with respect to axis*

a. Scaling

b. Rotation about arbitrary point

c. Translation

Scaling

To change the size of an object, scaling transformation is used. In the scaling process, you either expand or compress the dimensions of the object. Scaling can be achieved by multiplying the original coordinates of the object with the scaling factor to get the desired result.

Let us assume that the original coordinates are X,Y, the scaling factors are (SX, SY), and the produced coordinates are X',Y'. This can be mathematically represented as shown below −

X' = X . SX and Y' = Y . SY

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

$$\begin{pmatrix} X' \\ Y' \end{pmatrix} = \begin{pmatrix} X \\ Y \end{pmatrix} \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

OR

P' = P . S

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

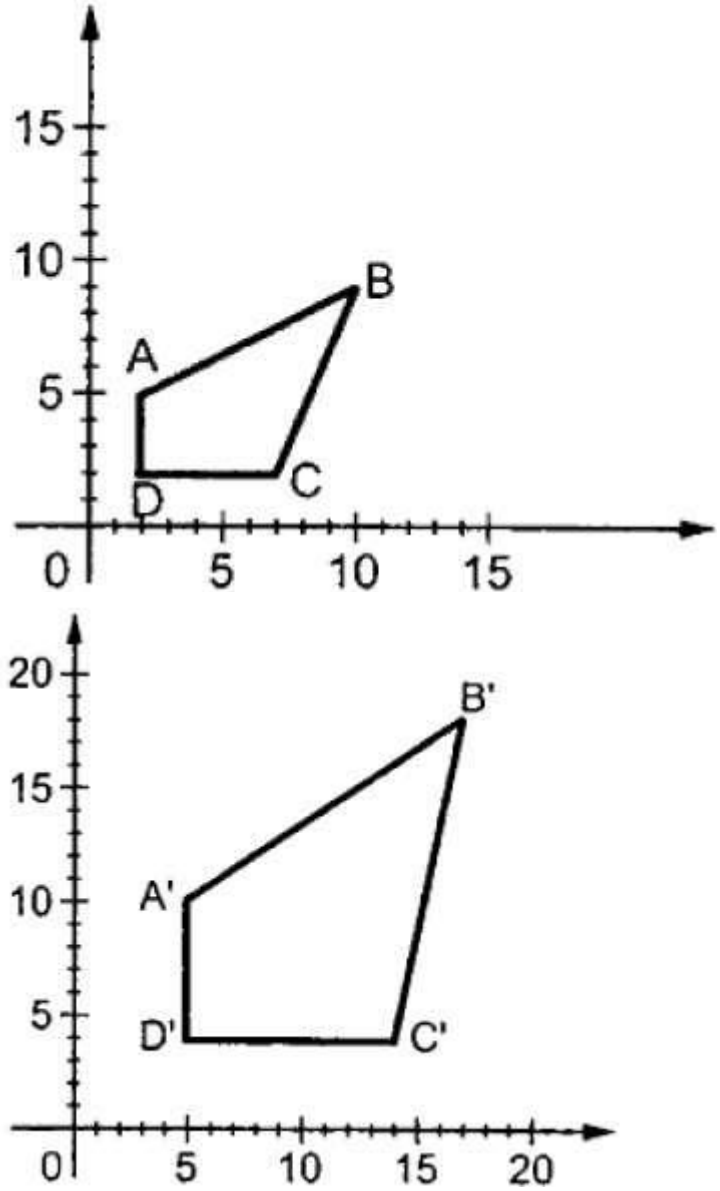
Before ScalingAfter Scaling

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

OR

P' = P . S

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

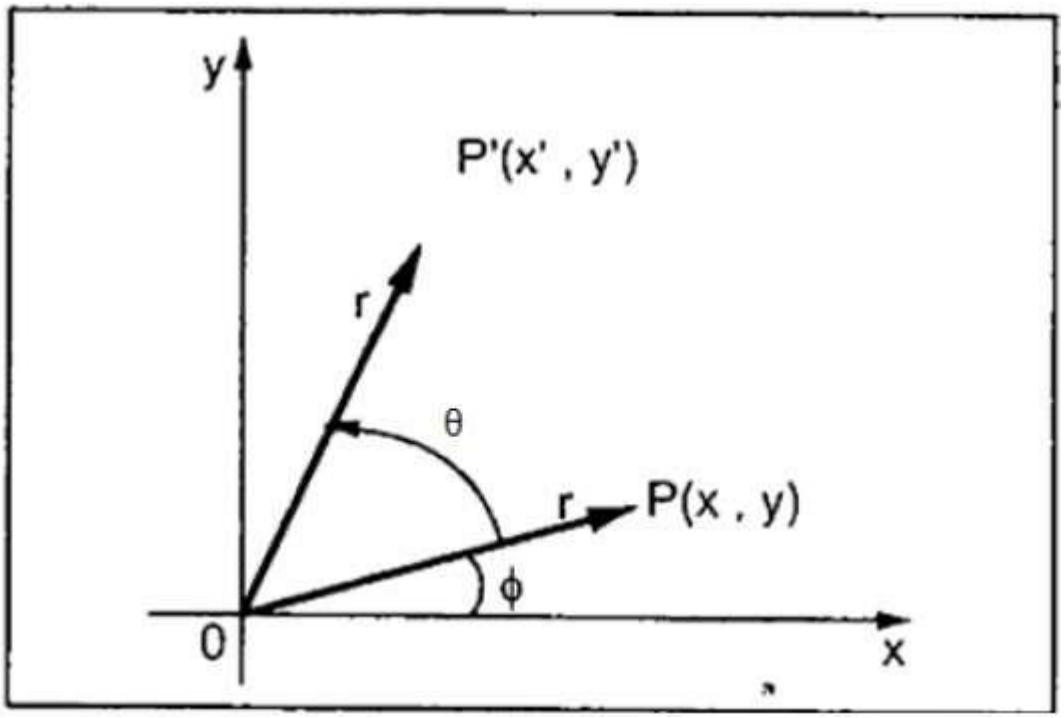


If we provide values less than 1 to the scaling factor S, then we can reduce the size of the object. If we provide values greater than 1, then we can increase the size of the object.

Rotation

In rotation, we rotate the object at particular angle θ thetatheta from its origin. From the following figure, we can see that the point PX,YX,Y is located at angle φ from the horizontal X coordinate with distance r from the origin.

Let us suppose you want to rotate it at the angle θ. After rotating it to a new location, you will get a new point P' X′,Y′X′,Y′.



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

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

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

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

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

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

Representing the above equation in matrix form,

$$[X'Y'] = [XY]\begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} OR$$

$$P' = P \cdot R$$

Where R is the rotation matrix

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

The rotation angle can be positive and negative.

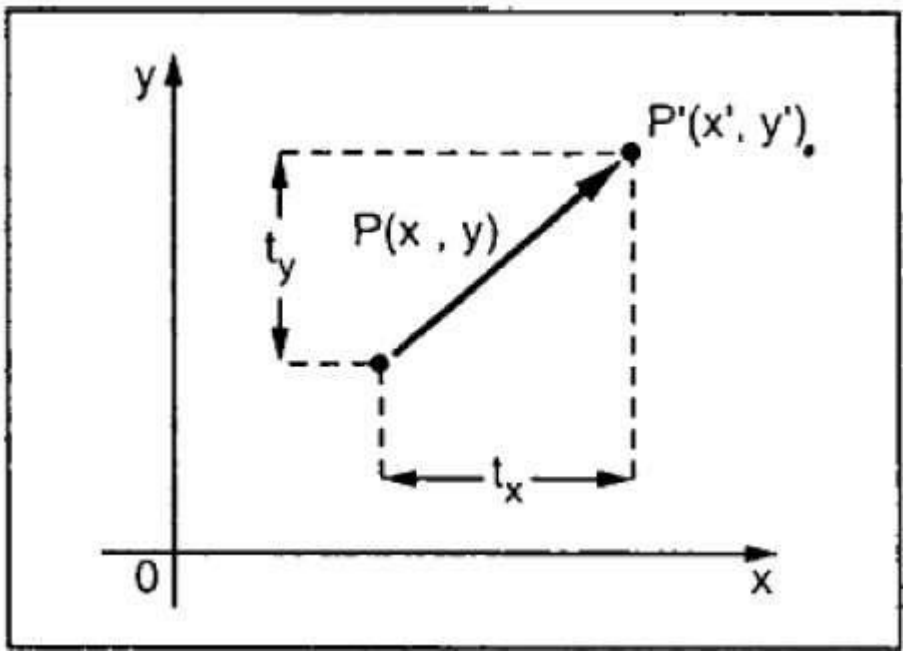
For positive rotation angle, we can use the above rotation matrix. However, for negative angle rotation, the matrix will change as shown below −

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

$$= \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} (\because cos(-\theta) = cos\theta \; and \; sin(-\theta) = -sin\theta)$$

Translation

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



From the above figure, you can write that −

$$X' = X + t_x$$

$$Y' = Y + t_y$$

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

$$P = \begin{bmatrix} X \\ Y \end{bmatrix} \qquad P' = \begin{bmatrix} X' \\ Y' \end{bmatrix} \qquad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

We can write it as −

$$P' = P + T$$

Assignment 7: *Generate fractal patterns using a. Bezier b. Koch Curve*

Koch Curve:

1. The Koch snowflake (also known as the Koch curve, star, or island) is a mathematical curve and one of the earliest fractal curves to have been described.
2. A Koch curve is a fractal generated by a replacement rule. This rule is, at each step, to replace the middle 131/3 of each line segment with two sides of a right triangle having sides of length equal to the replaced segment.
3. This quantity increases without bound; hen
4. ce the Koch curve has infinite length.
5. However, the curve still bounds a finite area.
6. We can prove this by noting that in each step, we add an amount of area equal to the area of all the equilateral triangles created.

Construction:

1. The Koch snowflake can be constructed by starting with an equilateral triangle, then recursively altering each line segment as follows:
    - Divide the line segment into three segments of equal length.
    - Draw an equilateral triangle that has the middle segment from step 1 as its base and points outward.

- Remove the line segment that is the base of the triangle from step 2.

- After one iteration of this process, the resulting shape is the outline of a hexagram.

1. The Koch snowflake is the limit approached as the above steps are followed over and over again.
2. The Koch curve originally described by Koch is constructed with only one of the three sides of the original triangle.
3. In other words, three Koch curves make a Koch snowflake.

Bezier Curve:

A Bezier curve is a mathematically defined curve used in two-dimensional graphic applications like adobe Illustrator, Inkscape etc. The curve is defined by four points: **the initial position** and **the terminating position** i.e **P0** and **P3** respectively (which are called “anchors”) and **two separate middle points** i.e **P1** and **P2**(which are called “handles”) in our example. Bezier curves are frequently used in computer graphics, animation, modelling etc.

How do we Represent Bezier Curves Mathematically?

Bezier 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 –

Where is the set of points and represents the Bernstein polynomials i.e. Blending Function which are given by –
Where **n** is the polynomial order, **i** is the index, and **u/t** is the variable which has from **0** to **1**.

Let us define our cubic bezier curve mathematically.

So a bezier curve id defined by a set of control points to where n is called its order(n = 1 for linear, n = 2 for quadratic, etc.). The first and last control points are always the endpoints of the curve; however, the intermediate control points (if any) generally do not lie on the curve.
For cubic bezier curve order(n) of polynomial is **3**,

index(i) vary from **i = 0 to i = n i.e. 3** and u will vary from.

Cubic Bezier Curve function is defined as :

$$P(u) = P_0 B_0^3(u) + P_1 B_1^3(u) + P_2 B_2^3(u) + P_3 B_3^3(u)$$

```
Cubic Bezier Curve blending function are defined as :
```

$$B_0^3(u) = \binom{3}{0}(1-u)^{3-0}u^0 \equiv 1(1-u)^3u^0$$

$$B_1^3(u) = \binom{3}{1}(1-u)^{3-1}u^1 \equiv 3(1-u)^2u^1$$

$$B_2^3(u) = \binom{3}{2}(1-u)^{3-2}u^2 \equiv 3(1-u)^1u^2$$

$$B_3^3(u) = \binom{3}{3}(1-u)^{3-3}u^3 \equiv 1(1-u)^0u^3$$

So

$$P(u) = (1-u)^3P_0 + 3u^1(1-u)^2P_1 + 3(1-u)^1u^2P_2 + u^3P_3$$

and

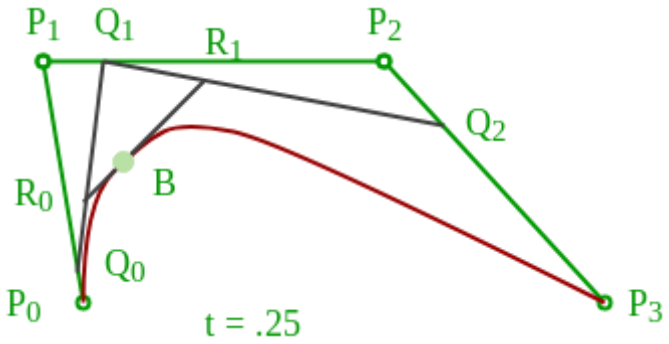
$$P(u) = \{x(u), y(u)\}$$

Now,

$$x(u) = (1-u)^3x_0 + 3u^1(1-u)^2x_1 + 3(1-u)^1u^2x_2 + u^3x_3$$

$$y(u) = (1-u)^3y_0 + 3u^1(1-u)^2y_1 + 3(1-u)^1u^2y_2 + u^3y_3$$

So and Now,
So we will calculate curve x and y pixel by incrementing value of u by **0.0001**.

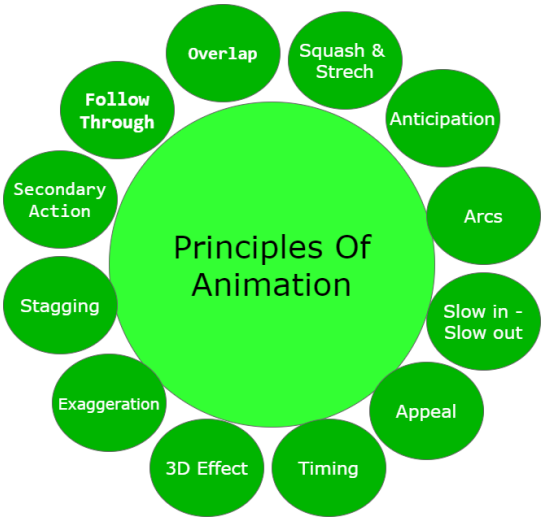


Construction of a cubic Bézier curve

Assignment 8: *Implement animation principles for any object*

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 Stretch**

   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:

1. Preparation phase
2. Movement phase
3. Finish

1. 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.

2. **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.

3. **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.

4. **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.

5. **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.

1. **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.

2. **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.

3. **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.

4. **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.

5. **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.

6. **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.