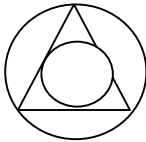


### List of Assignments

Sr. No.	TITLE
	Group A
01	Write C++ program to draw a concave polygon and fill it with desired color using scan fill algorithm. Apply the concept of inheritance.
02	Write C++ program to implement Cohen Southerland line clipping algorithm.
03	Write C++ program to draw the following pattern. Use DDA Line and Bresenham's Circle drawing algorithm. Apply Concept of encapsulation.
	
	Group B
04	Write C++ program to draw 2-D object and perform following basic transformations, Scaling b) Translation c) Rotation. Apply concept of operator overloading.
05	Write C++ program to generate Hilbert curve using concept of fractals.
	Group C
06	Write OpenGL program to draw Sun Rise and Sunset.
07	Write a C++ program to implement bouncing ball using sine wave form. Apply the concept of polymorphism.
	Mini-Projects / Case Study
08	Design and implement game / animation clip / Graphics Editor using open source graphics library. Make use of maximum features of Object Oriented Programming.

**Group A: Experiment No. 01**

**Problem Statement:** Write C++ program to draw a concave polygon and fill it with desired color using scan fill algorithm. Apply the concept of inheritance.

**Objective:** To understand Scan Fill Algorithms

**Outcome:** Implement Scan Fill Algorithm

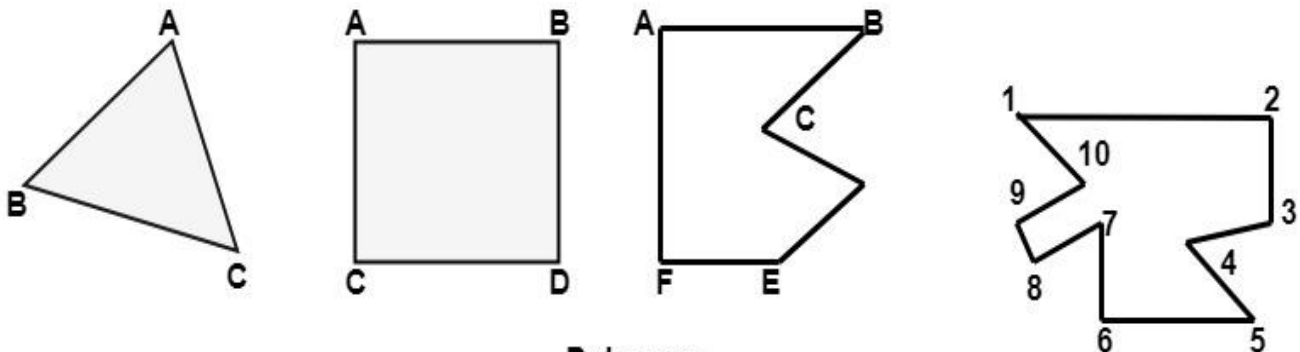
**Theory Concepts:**

Polygon is a representation of the surface. It is primitive which is closed in nature. It is formed using a collection of lines. It is also called as many-sided figure. The lines combined to form polygon are called sides or edges. The lines are obtained by combining two vertices.

Example of Polygon:

1. Triangle
2. Rectangle
3. Hexagon
4. Pentagon

Following figures shows some polygons.



Polygons

**Types of Polygons:**

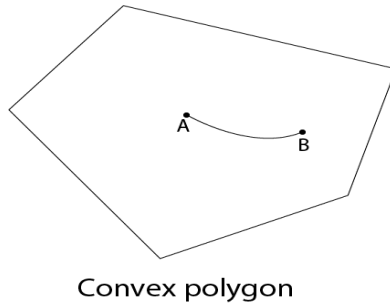
1. Concave
2. Convex.
3. Complex.

A polygon is called convex if line joining any two interior points of the polygon lies inside the polygon. A non-convex polygon is said to be concave. A concave polygon has one interior angle greater than  $180^\circ$ . So that it can be clipped into similar polygons.

A **convex polygon** is a simple polygon whose interior is a convex set. In a convex polygon, all interior angles are less than 180 degrees.

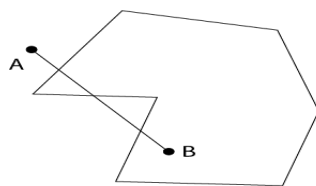
The following properties of a simple polygon are all equivalent to convexity:

- Every internal angle is less than or equal to 180 degrees.
- Every line segment between two vertices remains inside or on the boundary of the polygon.



Convex polygon

A **concave polygon** will always have an interior angle greater than 180 degrees. It is possible to cut a concave polygon into a set of convex polygons. You can draw at least one straight line through a concave polygon that crosses more than two sides.



Concave polygon

**Complex polygon** is a polygon whose sides cross over each other one or more times.

### Inside outside test (Even- Odd Test):

We assume that the vertex list for the polygon is already stored and proceed as follows.

1. Draw any point outside the range  $X_{\min}$  and  $X_{\max}$  and  $Y_{\min}$  and  $Y_{\max}$ . Draw a scan line through P up to a point A under study.
2. If this scan line
  - i. Does not pass through any of the vertices then its contribution is equal to the number of times it intersects the edges of the polygon. Say C if
    - a) C is odd then A lies inside the polygon.
    - b) C is even then it lies outside the polygon.
  - ii. If it passes through any of the vertices then the contribution of this intersection say V is,
    - a) Taken as 2 or even. If the other points of the two edges lie on one side of the scan line.
    - b) Taken as 1 if the other end points of the 2 edges lie on the opposite sides of the scan- line.
    - c) Here will be total contribution is  $C + V$ .

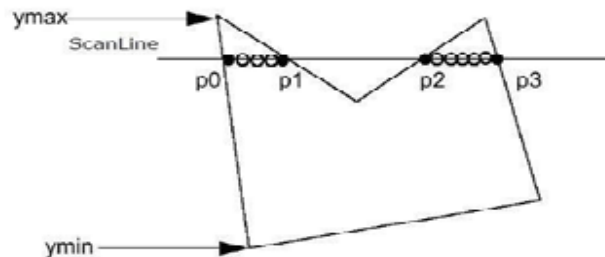
### Polygon Filling:

For filling polygons with particular colors, you need to determine the pixels falling on the border of the polygon and those which fall inside the polygon.

### Scan fill algorithm:

A scan-line fill of a region is performed by first determining the intersection positions of the boundaries of the fill region with the screen scan lines  $v$ . Then the fill colors are applied to each section of a scan line that lies within the interior of the fill region. The scan-line fill algorithm identifies the same interior regions as the odd-even rule.

It is an image space algorithm. It processes one line at a time rather than one pixel at a time. It uses the concept area of coherence. This algorithm records edge list, active edge list. So accurate bookkeeping is necessary. The edge list or edge table contains the coordinate of two endpoints. Active Edge List (AEL) contain edges a given scan line intersects during its sweep. The active edge list (AEL) should be sorted in increasing order of x. The AEL is dynamic, growing and shrinking.



### **Algorithm:**

**Step 1:** Start algorithm

**Step 2:** Initialize the desired data structure

- Create a polygon table having color, edge pointers, coefficients
- Establish edge table contains information regarding, the end point of edges, pointer to polygon, inverse slope.
- Create Active edge list. This will be sorted in increasing order of x.
- Create a flag F. It will have two values either on or off.

**Step 3:** Perform the following steps for all scan lines

- Enter values in Active edge list (AEL) in sorted order using y as value
- Scan until the flag, i.e. F is on using a background color
- When one polygon flag is on, and this is for surface S1 enter color intensity as I1 into refresh buffer
- When two or image surface flag are on, sort the surfaces according to depth and use intensity value S<sub>n</sub> for the nth surface. This surface will have least z depth value
- Use the concept of coherence for remaining planes.

**Step 4:** Stop Algorithm

### **Conclusion:**

After completion of experiment students are able to implement Scan Fill Algorithm.

### **Questions:**

- Which are the different approaches to fill a polygon?
- What are advantages and drawbacks of scan line polygon fill algorithm?

**Group A: Experiment No. 02**

**Problem Statement:** Write C++ program to implement Cohen Sutherland line clipping algorithm

**Objective:**

To understand Cohen Sutherland Line Clipping Algorithm.

**Outcome:**

Implement Cohen Sutherland Line Clipping Algorithm.

**Theory Concepts:**

**Line Clipping:**

It is performed by using the line clipping algorithm. The line clipping algorithms are:

1. Cohen Sutherland Line Clipping Algorithm
2. Midpoint Subdivision Line Clipping Algorithm
3. Liang-Barsky Line Clipping Algorithm

**Cohen Sutherland Line Clipping Algorithm:**

In the algorithm, first of all, it is detected whether line lies inside the screen or it is outside the screen. All lines come under any one of the following categories:

1. Visible
2. Not Visible
3. Clipping Case

**1. Visible:** If a line lies within the window, i.e., both endpoints of the line lies within the window. A line is visible and will be displayed as it is.

**2. Not Visible:** If a line lies outside the window it will be invisible and rejected. Such lines will not display. If any one of the following inequalities is satisfied, then the line is considered invisible. Let A ( $x_1, y_1$ ) and B ( $x_2, y_2$ ) are  $x_{min}, x_{max}$  are coordinates of the window.

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

$$x_1 > x_{max}$$

$$x_2 > x_{max}$$

$$y_1 > y_{max}$$

$$y_2 > y_{max}$$

$$x_1 < x_{min}$$

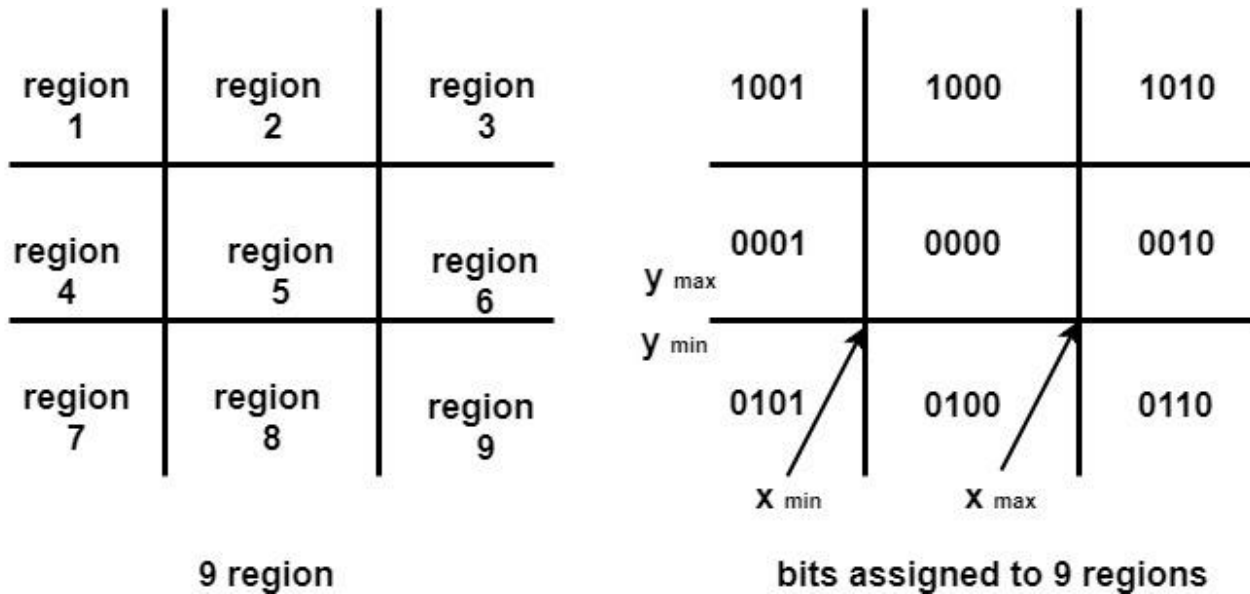
$$x_2 < x_{min}$$

$$y_1 < y_{min}$$

$$y_2 < y_{min}$$

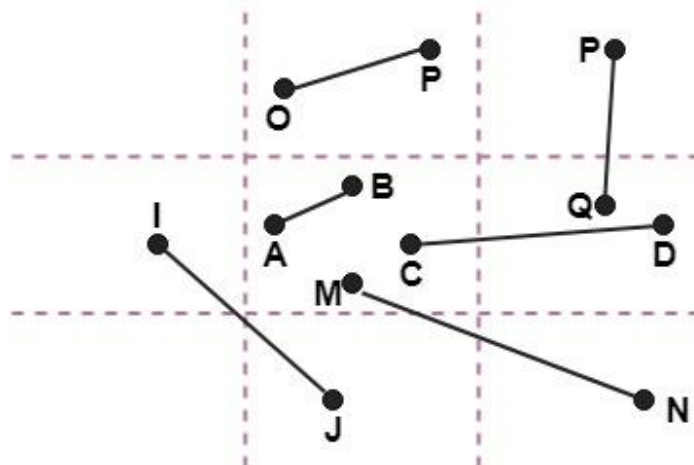
endpoints of line.

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



The center area is having the code, 0000, i.e., region 5 is considered a rectangle window.

Following figure show lines of various types



Line AB is the visible case  
 Line OP is an invisible case  
 Line PQ is an invisible line  
 Line IJ are clipping candidates  
 Line MN are clipping candidate  
 Line CD are clipping candidate

**Advantage of Cohen Sutherland Line Clipping:**

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

**Algorithm of Cohen Sutherland Line Clipping:**

**Step1:** Calculate positions of both endpoints of the line

**Step2:** Perform OR operation on both of these end-points

**Step3:** If the OR operation gives 0000

Then

line is considered to be visible

else

Perform AND operation on both endpoints

If And  $\neq$  0000

then the line is invisible

else

And = 0000

Line is considered the clipped case.

**Step4:** If a line is clipped case, find an intersection with boundaries of the window

$$m = (y_2 - y_1) / (x_2 - x_1)$$

(a) If bit 1 is "1" line intersects with left boundary of rectangle window

$$y_3 = y_1 + m(x - X_1)$$

where  $X = X_{wmin}$

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

(b) If bit 2 is "1" line intersect with right boundary

$$y_3 = y_1 + m(X - X_1)$$

where  $X = X_{wmax}$

where X more is maximum value of X co-ordinate of the window

(c) If bit 3 is "1" line intersects with bottom boundary

$$X_3 = X_1 + (y - y_1) / m$$

where  $y = y_{wmin}$

$y_{wmin}$  is the minimum value of Y co-ordinate of the window

(d) If bit 4 is "1" line intersects with the top boundary

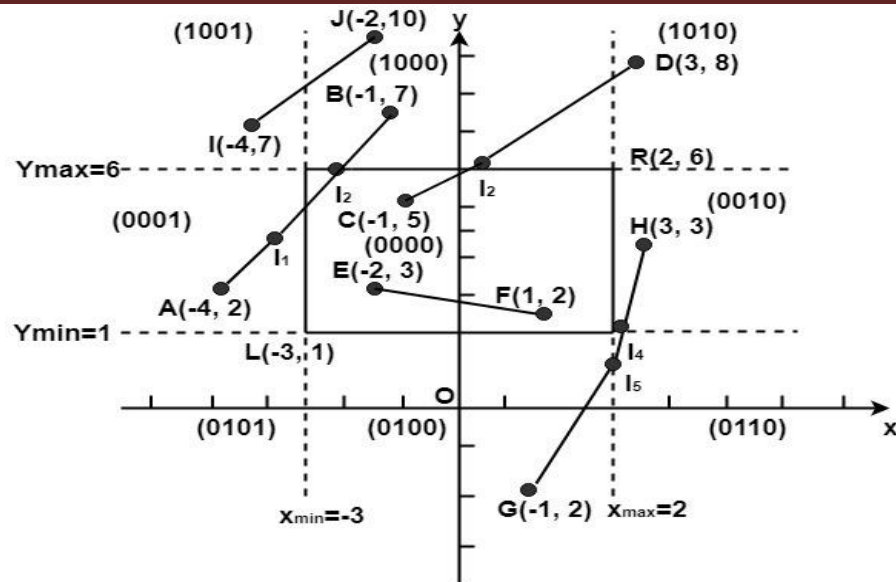
$$X_3 = X_1 + (y - y_1) / m$$

where  $y = y_{wmax}$

$y_{wmax}$  is the maximum value of Y co-ordinate of the window

**Example of Cohen-Sutherland Line Clipping Algorithm:**

Let R be the rectangular window whose lower left-hand corner is at L (-3, 1) and upper right-hand corner is at R (2, 6). Find the region codes for the endpoints in fig:



The region code for point  $(x, y)$  is set according to the scheme

Bit 1 =  $\text{sign}(y - y_{\max}) = \text{sign}(y - 6)$       Bit 3 =  $\text{sign}(x - x_{\max}) = \text{sign}(x - 2)$   
 Bit 2 =  $\text{sign}(y_{\min} - y) = \text{sign}(1 - y)$       Bit 4 =  $\text{sign}(x_{\min} - x) = \text{sign}(-3 - x)$

Here

$$\left\{ \begin{array}{ll} \text{sign}(a) = 1 & \text{if } a \text{ is positive} \\ 0 & \text{otherwise} \end{array} \right\}$$

So

A (-4, 2) → 0001	F (1, 2) → 0000
B (-1, 7) → 1000	G (-1, -2) → 0100
C (-1, 5) → 0000	H (3, 3) → 0100
D (3, 8) → 1010	I (-4, 7) → 1001
E (-2, 3) → 0000	J (-2, 10) → 1000

We place the line segments in their appropriate categories by testing the region codes found in the problem.

**Category1 (visible):** EF since the region code for both endpoints is 0000.

**Category2 (not visible):** IJ since  $(1001) \text{ AND } (1000) = 1000$  (which is not 0000).

**Category 3 (candidate for clipping):** AB since  $(0001) \text{ AND } (1000) = 0000$ , CD since  $(0000) \text{ AND } (1010) = 0000$ , and GH. Since  $(0100) \text{ AND } (0010) = 0000$ .

The candidates for clipping are AB, CD, and GH.

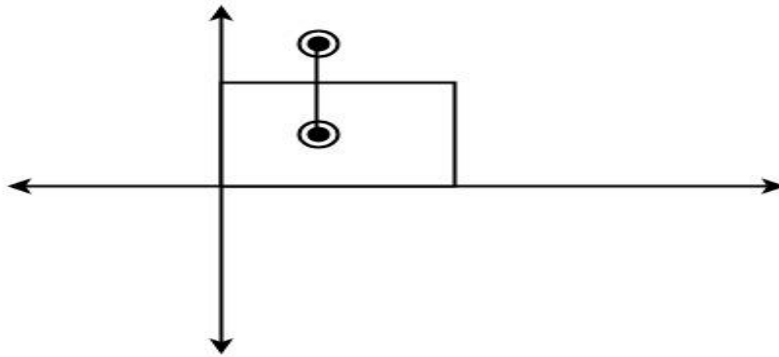


In clipping AB, the code for A is 0001. To push the 1 to 0, we clip against the boundary line  $x_{\min} = -3$ . The resulting intersection point is  $I_1 (-3, 3\frac{2}{3})$ . We clip (do not display)  $AI_1$  and  $I_1 B$ . The code for  $I_1$  is 1001. The clipping category for  $I_1 B$  is 3 since  $(0000) \text{ AND } (1000) \text{ is } (0000)$ . Now B is outside the window (i.e., its code is 1000), so we push the 1 to a 0 by clipping against the line  $y_{\max} = 6$ . The resulting intersection is  $I_2 (-1\frac{3}{5}, 6)$ . Thus  $I_2 B$  is clipped. The code for  $I_2$  is 0000. The remaining segment  $I_1 I_2$  is displayed since both endpoints lie in the window (i.e., their codes are 0000).

For clipping CD, we start with D since it is outside the window. Its code is 1010. We push the first 1 to a 0 by clipping against the line  $y_{\max} = 6$ . The resulting intersection  $I_3$  is  $(\frac{3}{5}, 6)$ , and its code is 0000. Thus  $I_3 D$  is clipped and the remaining segment  $CI_3$  has both endpoints coded 0000 and so it is displayed.

For clipping GH, we can start with either G or H since both are outside the window. The code for G is 0100, and we push the 1 to a 0 by clipping against the line  $y_{\min} = 1$ . The resulting intersection point is  $I_4 (2\frac{1}{5}, 1)$  and its code is 0010. We clip  $GI_4$  and work on  $I_4 H$ . Segment  $I_4 H$  is not displaying since  $(0010) \text{ AND } (0010) = 0010$ .

The area code of 1st point is 0000  
The area code of 2nd point is 1000  
Line is perfect candidate for clipping



### Conclusion:

Upon completion Students will be able to implement Cohen Sutherland Line Clipping Algorithm.

### Questions:

1. What is the limitation of Cohen Sutherland Line Clipping algorithm?
2. Give another example of Cohen Sutherland Line Clipping?

**Group A: Experiment No. 03****Problem Statement:**

Write C++ program to draw the following pattern. Use DDA line and Bresenham's circle drawing algorithm. Apply the concept of encapsulation

**Objective:**

To understand and implement DDA line and Bresenham's circle drawing algorithm.

**Outcome:**

Implement computer graphics programs in C++ using line and circle algorithm.

**Theory Concepts:****Graphics:**

Graphics are defined as any sketch or a drawing or a special network that pictorially represents some meaningful information. Computer Graphics is used where a set of images needs to be manipulated or the creation of the image in the form of pixels and is drawn on the computer.

**Line generation algorithm:**

In any 2-Dimensional plane, if we connect two points  $(x_0, y_0)$  and  $(x_1, y_1)$ , we get a line segment. But in the case of computer graphics, we cannot directly join any two coordinate points, for that, we should calculate intermediate points coordinates and put a pixel for each intermediate point, of the desired color with the help of functions like `putpixel(x, y, K)` ., where  $(x, y)$  is our co-ordinate and K denotes some color.

Examples:

**Input:** For line segment between (2, 2) and (6, 6)

**Output:** we need (3, 3) (4, 4) and (5, 5) as our intermediate points.

**Input:** For line segment between (0, 2) and (0, 6)

**Output:** we need (0, 3) (0, 4) and (0, 5) as our intermediate points.

For using graphics functions, our system output screen is treated as a coordinate system where the coordinate of the top-left corner is (0, 0) and as we move down our y-ordinate increases, and as we move right our x-ordinate increases for any point  $(x, y)$ . Now, for generating any line segment we need intermediate points and for calculating them.

For this first we can define a straight line with the help of the following equation.

$$y = mx + c$$

Where,

$(x, y)$  = axis of the line.

$m$  = Slope of the line.

$c$  = Interception point

Let us assume we have two points of the line  $(x_1, y_1)$  and  $(x_2, y_2)$ . Now, we will put values of the two points in straight line equation, and we get

$$y = mx + c$$

$$y_2 = mx_2 + c \quad \dots (1)$$

$$y_1 = mx_1 + c \quad \dots (2)$$

We have from equation (1) and (2)

$$y_2 - y_1 = mx_2 - mx_1$$

$$y_2 - y_1 = m (x_2 - x_1)$$

The value of  $m = (y_2 - y_1) / (x_2 - x_1)$

$$m = \Delta y / \Delta x$$

We can use a basic algorithm called,

1. DDA (Digital Differential Analyzer) Line Drawing Algorithm
2. Bresenham's Line Drawing Algorithm

### DDA (Digital Differential Analyzer) Line Drawing Algorithm:

The Digital Differential Analyzer helps us to interpolate the variables on an interval from one point to another point. We can use the digital Differential Analyzer algorithm to perform rasterization on polygons, lines, and triangles.

#### Algorithm:

**Step 1:** Read the input of the two end points of the line as  $(x_1, y_1)$  and  $(x_2, y_2)$  such that they are not equal.

**Step 2:** Calculate  $dx = x_2 - x_1$  and  $dy = y_2 - y_1$

**Step 3:** if  $(\text{abs}(dx) > \text{abs}(dy))$

step=dx;

else

step=dy;

**Step 4:**  $x_{inc} = dx / \text{step};$

$y_{inc} = dy / \text{step};$

**Step 5:**

for( $i = 0$ ;  $i \leq \text{step}$ ;  $i++$ )

{  
putpixel( $x_1, y_1$ )

$x_1 = x_1 + x_{inc}$

$y_1 = y_1 + y_{inc}$

}

#### Example:

A line has a starting point (1,7) and ending point (11,17). Apply the Digital Differential Analyzer algorithm to plot a line.

Solution: We have two coordinates,

Starting Point =  $(x_1, y_1) = (1, 7)$

Ending Point =  $(x_2, y_2) = (11, 17)$

Step 1: First, we calculate  $\Delta x$ ,  $\Delta y$  and  $m$ .

$$\Delta x = x_2 - x_1 = 11 - 1 = 10$$

$$\Delta y = y_2 - y_1 = 17 - 7 = 10$$

$$m = \Delta y / \Delta x = 10 / 10 = 1$$

Step 2: Now, we calculate the number of steps.

$$\Delta x = \Delta y = 10$$

Then, the number of steps = 10

Step 3: We get  $m = 1$ , Third case is satisfied.

Now move to next step.

Step 4: We will repeat step 3 until we get the endpoints of the line.

Step 5: Stop

$x_k$	$y_k$	$x_{k+1}$	$y_{k+1}$	$(x_{k+1}, y_{k+1})$
1	7	2	8	(2, 8)
		3	9	(3, 9)
		4	10	(4, 10)
		5	11	(5, 11)
		6	12	(6, 12)
		7	13	(7, 13)
		8	14	(8, 14)
		9	15	(9, 15)
		10	16	(10, 16)
		11	17	(11, 17)

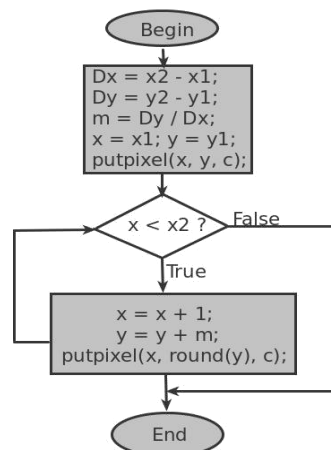
#### Advantages:

- It is a simple algorithm to implement.
- It is a faster algorithm than the direct line equation.
- We cannot use the multiplication method in Digital Differential Analyzer.
- Digital Differential Analyzer algorithm tells us about the overflow of the point when the point changes its location.

#### Disadvantages:

- The floating-point arithmetic implementation of the Digital Differential Analyzer is time-consuming.
- The method of round-off is also time-consuming.
- Sometimes the point position is not accurate

#### Flowchart for DDA Line Drawing Algorithm:



#### Circle generation algorithm:

The circle is also an important primitive in all the graphics packages. A circle can be defined by its center and radius. There is an important property of a circle which is its symmetry. You have to find points of the circle for only one octant. Points for other octants can be derived easily.

The equation of circle is  $x^2 + y^2 = r^2$ , where  $r$  is radius.

**Bresenham's circle drawing algorithm:**

**Step1:** Start Algorithm

**Step2:** Declare p, q, x, y, r, d variables

p, q are coordinates of the center of the circle

r is the radius of the circle

**Step3:** Enter the value of r

**Step4:** Calculate  $d = 3 - 2r$

**Step5:** Initialize  $x=0$

&nbsy= r

**Step6:** Check if the whole circle is scan converted

If  $x \geq y$

Stop

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

putpixel (x+p, y+q)

putpixel (y+p, x+q)

putpixel (-y+p, x+q)

putpixel (-x+p, y+q)

putpixel (-x+p, -y+q)

putpixel (-y+p, -x+q)

putpixel (y+p, -x+q)

putpixel (x+p, -y-q)

**Step8:** Find location of next pixels to be scanned

If  $d < 0$

then  $d = d + 4x + 6$

increment  $x = x + 1$

If  $d \geq 0$

then  $d = d + 4(x - y) + 10$

increment  $x = x + 1$

decrement  $y = y - 1$

**Step9:** Go to step 6

**Step10:** Stop Algorithm

**Example:**

Plot 6 points of circle using Bresenham Algorithm. When radius of circle is 10 units. The circle has centre (50, 50).

**Solution:** Let  $r = 10$  (Given)

**Step1:** Take initial point (0, 10)

$d = 3 - 2r$

$d = 3 - 2 * 10 = -17$

$$d < 0$$

$$\begin{aligned}\therefore d &= d + 4x + 6 \\ &= -17 + 4(0) + 6 \\ &= -11\end{aligned}$$

**Step2:** Plot (1, 10)

$$\begin{aligned}d &= d + 4x + 6 (\because d < 0) \\ &= -11 + 4(1) + 6 \\ &= -1\end{aligned}$$

**Step3:** Plot (2, 10)

$$\begin{aligned}d &= d + 4x + 6 (\because d < 0) \\ &= -1 + 4 \times 2 + 6 \\ &= 13\end{aligned}$$

**Step4:** Plot (3, 9)  $d$  is  $> 0$  so  $x = x + 1$ ,  $y = y - 1$

$$\begin{aligned}d &= d + 4(x-y) + 10 (\because d > 0) \\ &= 13 + 4(3-9) + 10 \\ &= 13 + 4(-6) + 10 \\ &= 23 - 24 = -1\end{aligned}$$

**Step5:** Plot (4, 9)

$$\begin{aligned}d &= -1 + 4x + 6 \\ &= -1 + 4(4) + 6 \\ &= 21\end{aligned}$$

**Step6:** Plot (5, 8)

$$\begin{aligned}d &= d + 4(x-y) + 10 (\because d > 0) \\ &= 21 + 4(5-8) + 10 \\ &= 21 - 12 + 10 = 19\end{aligned}$$

So  $P_1(0,0) \Rightarrow (50,50)$

$$P_2(1,10) \Rightarrow (51,60)$$

$$P_3(2,10) \Rightarrow (52,60)$$

$$P_4(3,9) \Rightarrow (53,59)$$

$$P_5(4,9) \Rightarrow (54,59)$$

$$P_6(5,8) \Rightarrow (55,58)$$

#### **Advantages of Bresenham's Circle Drawing Algorithm:**

- The entire algorithm is based on the simple equation of circle  $X^2 + Y^2 = R^2$ .
- It is easy to implement.

#### **Disadvantages of Bresenham's Circle Drawing Algorithm:**

- Accuracy of the generating points is an issue in this algorithm.
- This algorithm suffers when used to generate complex and high graphical images.
- There is no significant enhancement with respect to performance.

**Conclusion:**

Upon completion Students will be able to understand concept of DDA line & Bresenham's circle drawing algorithm

**Questions:**

1. Explain the derivation of decision parameters in Bresenham's circle drawing algorithm.
2. Explain the concept of encapsulation with example.

### Group B: Experiment No. 04

**Problem Statement:** Write C++ program to draw 2-D object and perform following basic transformations, a)Scaling b) Translation c) Rotation. Use operator overloading.

**Objective:** To understand 2-D object drawing with basic transformations.

**Outcome:**

Implement basic transformations Scaling, Translation and Rotation to draw 2-D object using Operator Overloading.

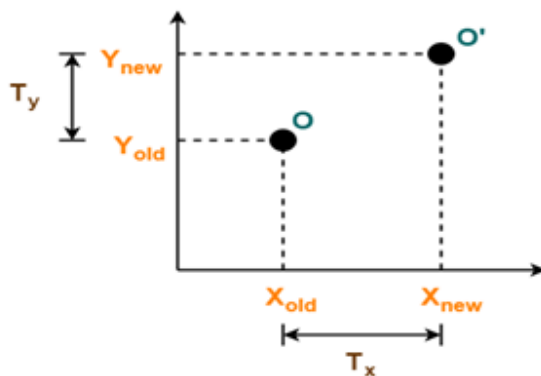
#### Theory Concepts:

Transformation means changing some graphics into something else by applying rules. We can have various types of transformations such as translation, scaling up or down, rotation, shearing, reflection etc. When a transformation takes place on a 2D plane, it is called 2D transformation. Transformations play an important role in computer graphics to reposition the graphics on the screen and change their size or orientation. Translation, Scaling and Rotation are basic transformations.

#### 1. Translation:

A translation moves an object to a different position on the screen. You can translate a point in 2D by adding translation coordinate or translation vector ( $T_x, T_y$ ) to the original coordinates. Consider

- Initial coordinates of the object  $O = (X_{old}, Y_{old})$
- New coordinates of the object  $O$  after translation =  $(X_{new}, Y_{new})$
- Translation vector or Shift vector =  $(T_x, T_y)$



This translation is achieved by adding the translation coordinates to the old coordinates of the object as-

$$X_{new} = X_{old} + T_x \quad (\text{This denotes translation towards X axis})$$

$$Y_{new} = Y_{old} + T_y \quad (\text{This denotes translation towards Y axis})$$

In Matrix form, the above translation equations may be represented as-



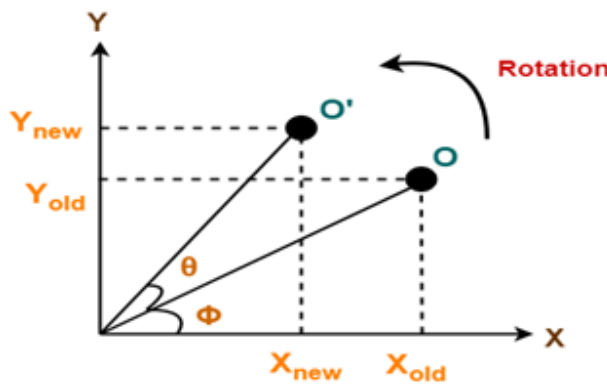
$$\begin{bmatrix} X_{\text{new}} \\ Y_{\text{new}} \end{bmatrix} = \begin{bmatrix} X_{\text{old}} \\ Y_{\text{old}} \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

Translation Matrix

## 2. Rotation:

In rotation, we rotate the object at particular angle  $\theta$  (theta) from its original position. Consider

- Initial coordinates of the object  $O = (X_{\text{old}}, Y_{\text{old}})$
- Initial angle of the object  $O$  with respect to origin  $= \Phi$
- Rotation angle  $= \theta$
- New coordinates of the object  $O$  after rotation  $= (X_{\text{new}}, Y_{\text{new}})$



This anti-clockwise rotation is achieved by using the following rotation equations-

$$X_{\text{new}} = X_{\text{old}} \times \cos\theta - Y_{\text{old}} \times \sin\theta$$

$$Y_{\text{new}} = X_{\text{old}} \times \sin\theta + Y_{\text{old}} \times \cos\theta$$

In Matrix form, the above rotation equations may be represented as-

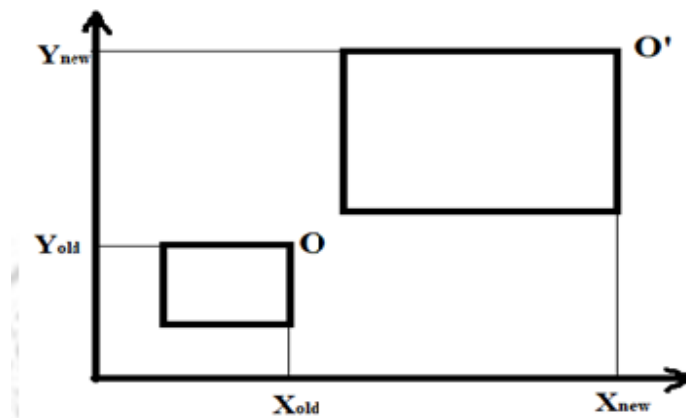
$$\begin{bmatrix} X_{\text{new}} \\ Y_{\text{new}} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} X_{\text{old}} \\ Y_{\text{old}} \end{bmatrix}$$

Rotation Matrix

### 3. Scaling:

Scaling transformation is used to change the size of an object. 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 ( $S_x, S_y$ ). If scaling factor  $> 1$ , then the object size is increased. If scaling factor  $< 1$ , then the object size is reduced. Consider

- Initial coordinates of the object  $O = (X_{old}, Y_{old})$
- Scaling factor for X-axis =  $S_x$
- Scaling factor for Y-axis =  $S_y$
- New coordinates of the object  $O$  after scaling =  $(X_{new}, Y_{new})$



This scaling is achieved by using the following scaling equations-  $X_{new} = X_{old} \times S_x$   
 $Y_{new} = Y_{old} \times S_y$

In Matrix form, the above scaling equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \end{bmatrix}$$

**Scaling Matrix**

**Homogeneous Coordinates:**

Matrix multiplication is easier to implement in hardware and software as compared to matrix addition. Hence we want to replace matrix addition by multiplication while performing transformation operations. So the solution is **homogeneous coordinates**, which allows us to express all transformations (including translation) as matrix multiplications.

To obtain homogeneous coordinates we have to represent transformation matrices in 3x3 matrices instead of 2x2. For this we add dummy coordinate. Each 2 dimensional position (x,y) can be represented by homogeneous coordinate as (x,y,1).

**Translation Matrix (Homogeneous Coordinates representation)**

$$\begin{bmatrix} X_{\text{new}} \\ Y_{\text{new}} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{\text{old}} \\ Y_{\text{old}} \\ 1 \end{bmatrix}$$

**Rotation Matrix (Homogeneous Coordinates representation)**

$$\begin{bmatrix} X_{\text{new}} \\ Y_{\text{new}} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{\text{old}} \\ Y_{\text{old}} \\ 1 \end{bmatrix}$$

**Scaling Matrix (Homogeneous Coordinates representation)**

$$\begin{bmatrix} X_{\text{new}} \\ Y_{\text{new}} \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{\text{old}} \\ Y_{\text{old}} \\ 1 \end{bmatrix}$$

**Applying transformations on equilateral triangle:**

Consider that coordinates of vertices of equilateral triangle are (X1,Y1), (X2,Y2) and (X3,Y3). After applying basic transformations, we will get corresponding coordinates as (X1',Y1'), (X2',Y2') and (X3',Y3') respectively. Following multiplication will give the translation on this equilateral triangle:

$$\begin{bmatrix} X1' & X2' & X3' \\ Y1' & Y2' & Y3' \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X1 & X2 & X3 \\ Y1 & Y2 & Y3 \\ 1 & 1 & 1 \end{bmatrix}$$

### Applying transformations on rhombus:

Consider that coordinates of vertices of rhombus are (X1,Y1), (X2,Y2), (X3,Y3) and (X4,Y4) applying basic transformations, we will get corresponding coordinates as (X1',Y1'), (X2',Y2'), (X3',Y3') and (X4',Y4') respectively. Following multiplication will give the translation on this rhombus:

$$\begin{bmatrix} X1' & X2' & X3' & X4' \\ Y1' & Y2' & Y3' & Y4' \\ 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X1 & X2 & X3 & X4 \\ Y1 & Y2 & Y3 & Y4 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Similarly we can apply rotation and scaling on rhombus.

### Conclusion:

After completion of this experiment students will be able to use basic transformations like translation, scaling, rotation in 2D objects.

### Questions:

1. How to rotate any 2-D object about an arbitrary point? Explain in brief.
2. Explain the concept of operator overloading with example.

**Group B: Experiment No. 05**

**Problem Statement:** Write C++ program to generate Hilbert curve using concept of fractals .

**Objective:** To understand Hilbert Curve using Fractals.

**Outcome:**

Implement Hilbert Curve using concept of Fractals and Constructor.

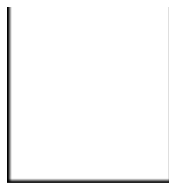
**Theory Concept:**

**The Hilbert curve**

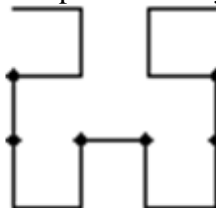
The Hilbert curve is a space filling curve that visits every point in a square grid with a size of  $2 \times 2$ ,  $4 \times 4$ ,  $8 \times 8$ ,  $16 \times 16$ , or any other power of 2. It was first described by David Hilbert in 1892. Applications of the Hilbert curve are in image processing: especially image compression and dithering. It has advantages in those operations where the coherence between neighboring pixels is important. The Hilbert curve is also a special version of a quadtree; any image processing function that benefits from the use of quadtrees may also use a Hilbert curve.

**Cups and joins**

The basic elements of the Hilbert curves are what I call "cups" (a square with one open side) and "joins" (a vector that joins two cups). The "open" side of a cup can be top, bottom, left or right. In addition, every cup has two end-points, and each of these can be the "entry" point or the "exit" point. So, there are eight possible varieties of cups. In practice, a Hilbert curve uses only four types of cups. In a similar vein, a join has a direction: up, down, left or right.



A first order Hilbert curve is just a single cup (see the figure on the left). It fills a  $2 \times 2$  space. The second order Hilbert curve replaces that cup by four (smaller) cups, which are linked together by three joins (see the figure on the right; the link between a cup and a join has been marked with a fat dot in the figure). Every next order repeats the process or replacing each cup by four smaller cups and three joins.



The function presented below (in the "C" language) computes the Hilbert curve. Note that the curve is symmetrical around the vertical axis. It would therefore be sufficient to draw half of the Hilbert curve.

**Conclusion:**

After completion of this experiment students will be able to use Hilbert Curve using the concept of Fractals.

**Questions:**

1. What is the importance of curves and fractals in computer graphics?
2. What are applications of curves and fractals?

**Group C: Experiment No. 06**

**Problem Statement:** Write OpenGL program to draw Sun Rise and Sunset.

**Objective:** To understand OpenGL concepts implement to draw Sun Rise and Sunset.

**Outcome:**

Implement OpenGL concept.

**Theory Concept:**

**OpenGL Basics:**

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 is a low-level, widely supported modeling and rendering software package, available across all platforms. It can be used in a range of graphics applications, such as games, CAD design, or modeling. 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 freglut. Later included library were GLEE, GLEW and glbinding.
- **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.

**Installation of OpenGL on Ubuntu**

We need the following sets of libraries in programming OpenGL:

1. **Core OpenGL (GL):** consists of hundreds of functions, which begin with a prefix "gl" (e.g., glColor, glVertex, glTranslate, glRotate). The Core OpenGL models an object via a set of geometric primitives, such as point, line, and polygon.

2. **OpenGL Utility Library (GLU):** built on-top of the core OpenGL to provide important utilities and more building models (such as quadric surfaces). GLU functions start with a prefix "glu" (e.g., gluLookAt, gluPerspective)

3. **OpenGL Utilities Toolkit (GLUT):** provides support to interact with the Operating System (such as creating a window, handling key and mouse inputs); and more building models (such as sphere and torus). GLUT functions start with a prefix of "glut" (e.g., glutCreateWindow, glutMouseFunc). GLUT is designed for constructing small to medium sized OpenGL programs. While GLUT is well-suited to learning OpenGL and developing simple OpenGL applications, GLUT is not a full-featured toolkit so large

applications requiring sophisticated user interfaces are better off using native window system toolkits. GLUT is simple, easy, and small.

Alternative of GLUT includes SDL.

**4. OpenGL Extension Wrangler Library (GLEW):** "GLEW is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

For installing OpenGL on ubuntu, just execute the following command (like installing any other thing) in terminal:

- `sudo apt-get install freeglut3-dev` For

working on Ubuntu operating system:

- `gcc filename.c -lGL -lGLU -lglut`

where filename.c is the name of the file with which this program is saved.

### Prerequisites for OpenGL

Since OpenGL is a graphics API and not a platform of its own, it requires a language to operate in and the language of choice is C++.

### Getting started with OpenGL

#### Overview of OpenGL program:

- Main
  - Open window and configure frame buffer (using GLUT for example)
  - Initialize GL states and display (Double buffer, color mode, etc.)
- Loop
  - Check for events

if window event (resize, unhide, maximize etc.)

modify the viewport

and Redraw

else if input event (keyboard and mouse etc.)

handle the event (such as move the camera or change the state)

*and usually draw the scene*

- Redraw
  - Clear the screen (and buffers e.g., z-buffer)
  - Change states (if desired)
  - Render
  - Swap buffers (if double buffer)

#### OpenGL order of operations

- Construct shapes (geometric descriptions of objects – vertices, edges, polygons etc.)
- Use OpenGL to
  - Arrange shape in 3D (using transformations)
  - Select your vantage point (and perhaps lights)
  - Calculate color and texture properties of each object



- Convert shapes into pixels onscreen

### OpenGL Syntax

- All functions have the form: `gl*`
  - `glVertex3f()` – **3** means that this function take three arguments, and **f** means that the type of those arguments is float.
  - `glVertex2i()` – **2** means that this function take two arguments, and **i** means that the type of those arguments is integer
- All variable types have the form: `GL*`
  - In OpenGL program it is better to use OpenGL variable types (portability)
    - `GLfloat` instead of `float`
    - `GLint` instead of `int`

### OpenGL primitives

Drawing two lines

```
glBegin(GL_LINES);
glVertex3f(, , ); // start point of line 1
glVertex3f(, , ); // end point of line 1
glVertex3f(, , ); // start point of line 2
glVertex3f(, , ); // end point of line 2
glEnd();
```

We can replace `GL_LINES` with `GL_POINTS`, `GL_LINELOOP`, `GL_POLYGON` etc.

### OpenGL states

- On/off (e.g., depth buffertest)
  - `glEnable( GLenum )`
  - `glDisable( GLenum )`
  - Examples:
    - `glEnable(GL_DEPTH_TEST);`
    - `glDisable(GL_LIGHTING);`
- Mode States
  - Once the mode is set the effect stays until reset
  - Examples:
    - `glShadeModel(GL_FLAT)` or `glShadeModel(GL_SMOOTH)`
    - `glLightModel(...)` etc.

OpenGL provides a consistent interface to the underlying graphics hardware. This abstraction allows a single program to run on different graphics hardware easily. A program written with OpenGL can even be run in software (slowly) on machines with no graphics acceleration. OpenGL function names always begin with `gl`, such as `glClear()`, and they may end with characters that indicate the types of the parameters, for example `glColor3f(GLfloat red, GLfloat green, GLfloat blue)` takes three floating-point color parameters and `glColor4dv(const GLdouble *v)` takes a pointer to an array that contains 4 double-precision floating-point values. OpenGL constants begin with `GL`, such as `GL_DEPTH`. OpenGL also uses special names for types that are passed to its functions, such as `GLfloat` or `GLint`, the corresponding C types are compatible, that is `float` and `int` respectively.

GLU is the OpenGL utility library. It contains useful functions at a higher level than those provided by

OpenGL, for example, to draw complex shapes or set up cameras. All GLU functions are written on top of OpenGL. Like OpenGL, GLU function names begin with glu, and constants begin with GLU. GLUT, the OpenGL Utility Toolkit, provides a system for setting up callbacks for interacting with the user and functions for dealing with the windowing system. This abstraction allows a program to run on different operating systems with only a recompile. Glut follows the convention of prepending function names with glut and constants with GLUT.

### Writing an OpenGL Program with GLUT

An OpenGL program using the three libraries listed above must include the appropriate headers. This requires the following three lines:

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

Before OpenGL rendering calls can be made, some initialization has to be done. With GLUT, this consists of initializing the GLUT library, initializing the display mode, creating the window, and setting up callback functions. The following lines initialize a full color, double buffered display: *glutInit(&argc, argv);*  
*glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB);*

Double buffering means that there are two buffers, a front buffer and a back buffer. The front buffer is displayed to the user, while the back buffer is used for rendering operations. This prevents flickering that would occur if we rendered directly to the front buffer.

Next, a window is created with GLUT that will contain the viewport which displays the OpenGL front buffer with the following three lines:

```
glutInitWindowPosition(px, py);  
glutInitWindowSize(sx, sy);  
glutCreateWindow(name);
```

To register callback functions, we simply pass the name of the function that handles the event to the appropriate GLUT function.

```
glutReshapeFunc(reshape);  
glutDisplayFunc(display);
```

Here, the functions should have the following prototypes:

```
void reshape(int width, int height); void  
display();
```

In this example, when the user resizes the window, reshape is called by GLUT, and when the display needs to be refreshed, the display function is called. For animation, an idle event handler that takes no arguments can be created to call the display function to constantly redraw the scene with *glutIdleFunc*. Once all the callbacks have been set up, a call to *glutMainLoop* allows the program to run.

In the display function, typically the image buffer is cleared, primitives are rendered to it, and the results are presented to the user. The following line clears the image buffer, setting each pixel color to the clear color, which can be configured to be any color:

```
glClear(GL_COLOR_BUFFER_BIT);
```

---

The next line sets the current rendering color to blue. OpenGL behaves like a state machine, so certain state such as the rendering color is saved by OpenGL and used automatically later as it is needed.

```
glColor3f(0.0f, 0.0f, 1.0f);
```

To render a primitive, such as a point, line, or polygon, OpenGL requires that a call to *glBegin* is made to specify the type of primitive being rendered.

```
glBegin(GL_LINES);
```

Only a subset of OpenGL commands is available after a call to *glBegin*. The main command that is used is *glVertex*, which specifies a vertex position. In GL LINES mode, each pair of vertices define endpoints of a line segment. In this case, a line would be drawn from the point at (x0, y0) to (x1, y1).

```
glVertex2f(x0, y0); glVertex2f(x1, y1);
```

A call to *glEnd* completes rendering of the current primitive. *glEnd()*; Finally, the back buffer needs to be swapped to the front buffer that the user will see, which GLUT can handle for us:

```
glutSwapBuffers();
```

### Developer-Driven Advantages

- **Industry standard**

An independent consortium, the OpenGL Architecture Review Board, guides the OpenGL specification. With broad industry support, OpenGL is the only truly open, vendor-neutral, multiplatform graphics standard.

- **Stable**

OpenGL implementations have been available for more than seven years on a wide variety of platforms. Additions to the specification are well controlled, and proposed updates are announced in time for developers to adopt changes. Backward compatibility requirements ensure that existing applications do not become obsolete.

- **Reliable and portable**

All OpenGL applications produce consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or windowing system.

- **Evolving**

Because of its thorough and forward-looking design, OpenGL allows new hardware innovations to be accessible through the API via the OpenGL extension mechanism. In this way, innovations appear in the API in a timely fashion, letting application developers and hardware vendors incorporate new features into their normal product release cycles.

- **Scalable**

OpenGL API-based applications can run on systems ranging from consumer electronics to PCs, workstations, and supercomputers. As a result, applications can scale to any class of machine that

the developer chooses to target.

- **Easy to use**

OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers encapsulate information about the underlying hardware, freeing the application developer from having to design for specific hardware features.

- **Well-documented:**

Numerous books have been published about OpenGL, and a great deal of sample code is readily available, making information about OpenGL inexpensive and easy to obtain.

**Conclusion:**

After completion of this experiment students will be able to use OpenGL.

**Questions:**

1. What are the advantages of Open GL over other API's?
2. Explain rendering pipeline with reference to OpenGL.

**Group C: Experiment No. 07****Problem Statement:**

Write a C++ program to implement bouncing ball using sine wave form. Apply the concept of polymorphism.

**Objective:**

To understand and implement bouncing ball using sine wave form.

**Outcome:**

Implement concept of polymorphism.

**Theory Concept:****What is animation?**

**Animation** is the process of designing, drawing, making layouts and preparation of photographic sequences which are integrated in the multimedia and gaming products. Animation involves the exploitation and management of still images to generate the illusion of movement.

How to move an element to left, right, up and down using arrow keys?

To detect which arrow key is pressed, you can use `ncurses.h` header file. Arrow key's code is defined as: `KEY_UP`, `KEY_DOWN`, `KEY_LEFT`, `KEY_RIGHT`.

**Conclusion:**

After completion of this experiment students will be able to use Polymorphism.