State Institute of Engineering & Technology, Nilokheri
Department of Computer Science & Engineering (AI &ML)

Lab Manual

Interactive Computer Graphics
(B23-CAM-311)

INDEX

S. No.	Name of the Practical
1.	Write a program to implement DDA line drawing algorithm.
2.	Write a program to implement Bresenham's line drawing algorithm.
3.	Write a program to implement Bresenham's circle drawing algorithm.
4.	Write a program to implement the Midpoint circle drawing algorithm.
5.	Write a program to perform line clipping.
6.	Write a program to implement polygon filling.
7.	Write a program to move an object using the concept of 2-D translation transformation.
8.	Write a program to move an object using the concept of 2-D rotation transformation.
9.	Write a program to move an object using the concept of 2-D scaling transformation
10.	Write a program to move an object using the concept of 2-D shearing transformation.

ROGRAM NO. 1

Aim: - Write a program to implement DDA line drawing algorithm.

Description: DDA algorithm is an incremental scan conversion method. Here we perform calculations at each step using the results from the preceding step. The characteristic of the DDA algorithm is to take unit steps along one coordinate and compute the corresponding values along the other coordinate. The unit steps are always along the coordinate of greatest change, e.g. if dx = 10 and dy = 5, then we would take unit steps along x and compute the steps along y.

In DDA we need to consider two cases;

One is slope of the line less than or equal to one (|m| ≤1) and slope of the line greater than one (m| > 1).

1)When |m| ≤ 1 means y2-y1 = x2-x1 or y2-y1 <x2-x1. In both these cases we assume x to be the major axis. Therefore, we sample x axis at unit intervals and find the y values corresponding to each x value. We have the slope equation as

$$\Delta y = m \Delta x$$
$$y2-y1 = m (x2-x1)$$

In general terms we can say that y i+1 - yi = m (xi+1 - xi). But here Δ x = 1; therefore, the equation reduces to y i+1= yi + m = yi + dy/dx.

2) When | m| > 1 means y2-y1> x2-x1 and therefore we assume y to be the major axis. Here we sample y axis at unit intervals and find the x values corresponding to each y value. We have the slope equation as

$$\Delta y = m \Delta x$$
$$y2-y1 = m (x2-x1)$$

Algorithm:

1. Start.
2. Declare variables x, y, x1, y1, x2, y2, k, dx, dy, s, xi, yi and also declare

   gdriver=DETECT, mode.

3. Initialize the graphic mode with the path location in TurboC3 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), do s=abs(dx).
8. Otherwise, s= abs(dy).
9. Then xi=dx/s and yi=dy/s.
10. Start from k=0 and continuing till k<s, the points will be

i. x=x+xi.

ii. Y=y+yi.

11. Plot pixels using putpixel at points (x,y) in specified colour.

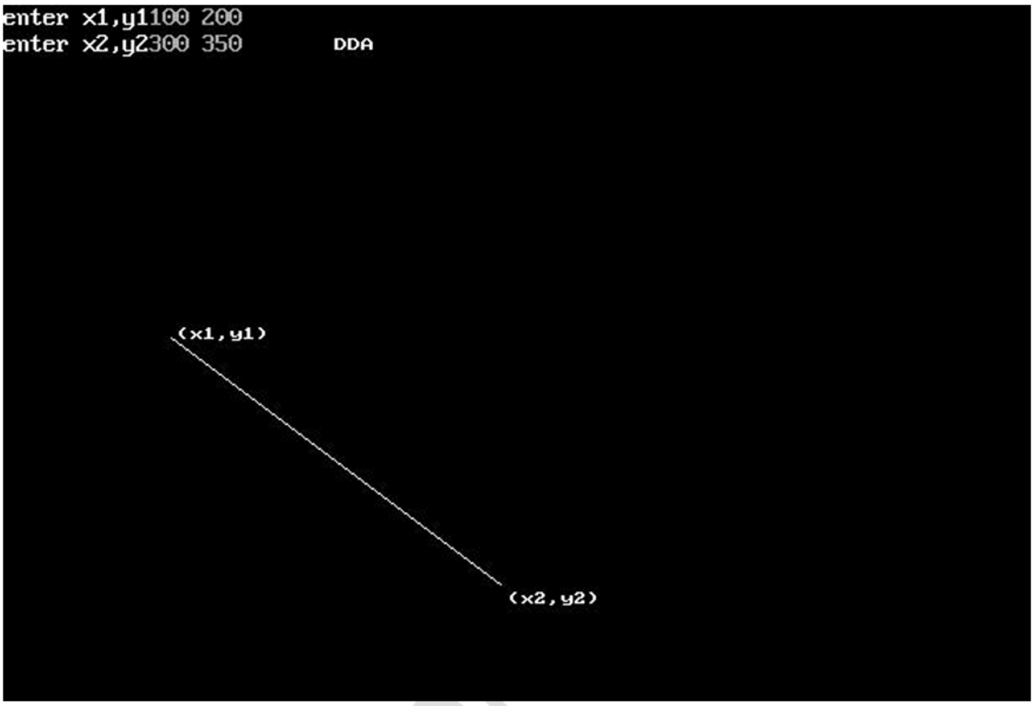
12. Close Graph and stop.

Program:

```
#include<stdio.h>
#include<graphics.h>
#include<math.h>
float round(float a);
void main()
{
int gd=DETECT,gm;
// gd=graphics driver (detects best graphics driver and assigns it as default, gm=graphics mode.

int x1, y1, x2, y2, steps, k;
float xincr, yincr, x, y, dx, dy;
printf("enter x1, y1");
scanf("%d%d", &x1, &y1);
printf("enter x2, y2");
scanf("%d %d", &x2, &y2);
initgraph(&gd, &gm,"c:\\turboc3\\BGI"); //initializes the graph
dx=x2-x1;
dy=y2-y1;
if(abs(dx)>abs(dy))
steps=abs(dx);
else
steps=abs(dy);
xincr=dx/steps;
yincr=dy/steps;
x=x1;
y=y1;
for(k=1;k<=steps;k++)
```

```
{
delay(100); //for seeing the line drawing process slowly.
x+=xincr;
y+=yincr;
putpixel(round(x), round(y), WHITE);
}
outtextxy(200,20,"DDA"); // for printing text at desired screen location.
outtextxy(x1+5,y1-5,"(x1,y1)");
outtextxy(x2+5,y2+5,"(x2,y2)");
getch();
closegraph();   // closes the graph and comes back to previous graphic mode.
}
float round(float a)
{
int b=a+0.5;
return b;
}
```

Output:-



Applications of DDA

The DDA algorithm is used to draw straight lines by calculating intermediate pixel positions between the start and end points. Its applications include:

1. Computer Graphics
   - Drawing straight lines in graphics applications and CAD software.
2. Computer-Aided Design (CAD)
   - Creating engineering drawings, blueprints, and architectural designs.
3. Computer Games
   - Drawing roads, walls, boundaries, and other straight objects in 2D games.
4. Graphic Editors
   - Used in drawing tools such as Paint and other image editing software.
5. Educational Purpose
   - Teaching the fundamentals of scan conversion and raster graphics.
6. GUI Development
   - Drawing windows, menus, buttons, and borders in graphical user interfaces.
7. Plotters and Printers
   - Converting vector graphics into raster images for printing.
8. Simulation and Animation
   - Drawing paths and trajectories in simulation and animation software.

Viva Questions with Answers

Q1. What is the DDA algorithm?
Answer: DDA (Digital Differential Analyzer) is a scan conversion algorithm used to draw a straight line by calculating intermediate pixel positions using incremental floating-point calculations.

Q2. What is the basic principle of the DDA algorithm?
Answer: The algorithm divides the line into equal steps and increments the x-coordinate or y-coordinate by a fixed amount while calculating the corresponding coordinate using the line slope.

Q3. What is the major disadvantage of the DDA algorithm?
Answer: The DDA algorithm uses floating-point arithmetic and rounding operations, which make it slower and can introduce cumulative rounding errors.

Q4. Which algorithm is generally preferred over DDA for line drawing and why?
Answer: Bresenham's Line Drawing Algorithm is preferred because it uses only integer arithmetic, making it faster, more efficient, and more accurate than DDA.

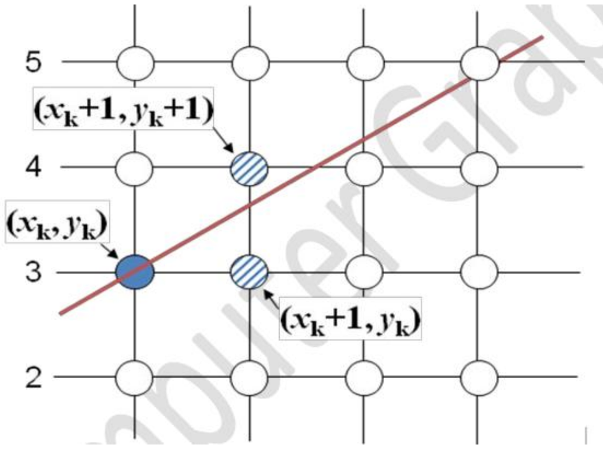
Q5. How is the number of steps determined in the DDA algorithm?
Answer: The number of steps is calculated as:
Steps = $\max(|x_2 - x_1|, |y_2 - y_1|)$
The larger difference between the x-coordinates and y-coordinates determines the number of iterations required to draw the line.

PROGRAM NO. 2

Aim: Write a program to implement Bresenham's line drawing algorithm.

Description: Bresenham's Line Drawing Algorithm is an efficient scan conversion algorithm used to draw a straight line on a raster display. Unlike the DDA algorithm, it uses only **integer arithmetic**, making it faster and more accurate.
The algorithm determines the nearest pixel to the ideal line at each step using a **decision parameter (p)**. It avoids floating-point calculations, reducing computation time and improving performance.
It is mainly used for drawing lines where the slope lies between **0 and 1**. Modified versions handle other slopes.

Move across the x axis in unit intervals and at each step choose between two different y coordinates



For example, from position (2, 3) we have to choose between (3, 3) and (3, 4). We would like the point that is closer to the original line
So we have to take decision to choose next point. So next pixels are selected based on the value of decision parameter p. The equations are given in below algorithm.

Algorithm:

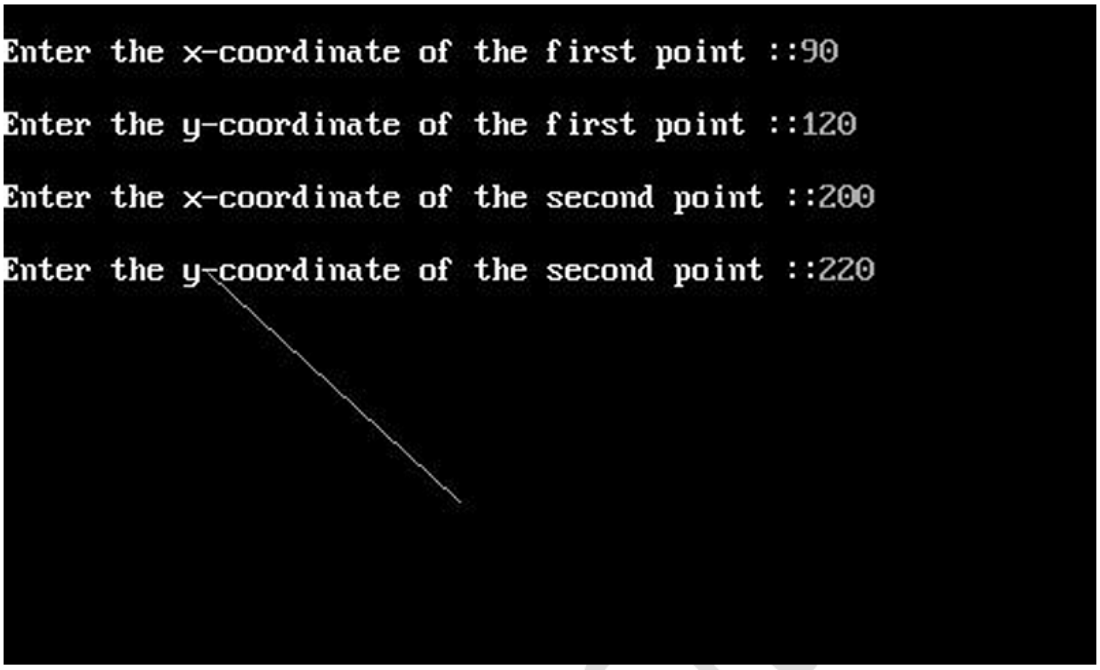
1. Read the starting point **(x_1, y_1)** and ending point **(x_2, y_2)**.
2. Calculate:
   - $dx = x_2 - x_1$
   - $dy = y_2 - y_1$
3. Initialize the decision parameter:
   - $\mathbf{p = 2dy - dx}$
4. Set the initial point as (x_1, y_1).
5. Plot the first pixel.
6. Repeat until $x = x_2$:
   - If $\mathbf{p < 0}$
     - $x = x + 1$
     - $p = p + 2dy$
   - Else
     - $x = x + 1$

        - y = y + 1
        - p = p + 2dy − 2dx
    - Plot the new point.
7. Stop.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
	int x,y,x1,y1,x2,y2,p,dx,dy;
	int gd=DETECT,gm;
	initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
	printf("\nEnter the x-coordinate of the first point ::");
	scanf("%d",&x1);
	printf("\nEnter the y-coordinate of the first point ::");
	scanf("%d",&y1);
	printf("\nEnter the x-coordinate of the second point ::");
	scanf("%d",&x2);
	printf("\nEnter the y-coordinate of the second point ::");
	scanf("%d",&y2);
	x=x1; y=y1; dx=x2-x1; dy=y2-y1;
	putpixel(x,y,2);
	p=(2*dy-dx);
	while(x<=x2)
	{
		if(p<0)
		{
			x=x+1;
			p=p+2*dy;
		}
		else
		{
			x=x+1;
			y=y+1;
			p=p+(2*dy)-(2*dx);
		}
		putpixel(x,y,7);
	}
	getch();
	closegraph();
}
```

Output:



Applications:

1. Drawing straight lines in Computer Graphics.
2. CAD (Computer-Aided Design) software.
3. Engineering drawing applications.
4. Video games for rendering 2D objects.
5. Plotters and printers.
6. GIS (Geographical Information Systems).
7. Robot path visualization.
8. Graph plotting software.
9. Graphical User Interfaces (GUI).
10. Image processing applications.

Viva Questions with Answers

Q1. What is Bresenham's Line Drawing Algorithm?
Answer: It is an efficient raster scan algorithm used to draw a straight line using integer arithmetic instead of floating-point calculations.

Q2. Why is Bresenham's algorithm faster than the DDA algorithm?
Answer: Because it uses only integer operations and avoids floating-point arithmetic and rounding.

Q3. What is the decision parameter in Bresenham's algorithm?
Answer: The decision parameter determines which pixel is closer to the ideal line. It is initialized as: $\mathbf{p = 2dy - dx}$

Q4. What is the initial value of the decision parameter?
Answer: $p = 2dy - dx$
where:

- $dx = x_2 - x_1$
- $dy = y_2 - y_1$

Q5. Which arithmetic operations are mainly used in Bresenham's algorithm?
Answer: Integer addition and subtraction.

Q6. Which is better: DDA or Bresenham?
Answer: Bresenham's algorithm is generally better because it is faster, more efficient, and more accurate due to the use of integer arithmetic.

Q7. What type of display uses Bresenham's algorithm?
Answer: Raster scan displays.

Q8. Why is the algorithm called an incremental algorithm?
Answer: Because it calculates the next pixel using the previous pixel's information and updates the decision parameter incrementally.

Q9. What is the time complexity of Bresenham's algorithm?
Answer: **O(n)**, where *n* is the number of pixels along the line.

Q10. What is the major advantage of Bresenham's algorithm over DDA?
Answer: It eliminates floating-point operations, making it faster and more suitable for real-time graphics applications.

PROGRAM NO. 3

Aim: Write a program to implement Bresenham's circle drawing algorithm.

Description: Bresenham's Circle Drawing Algorithm is an efficient raster graphics algorithm used to draw a circle by determining the nearest pixel to the actual circle path using integer arithmetic. Instead of using the circle equation

$$x^2 + y^2 = r^2$$

or trigonometric functions, the algorithm calculates a decision parameter to determine whether the next pixel should be chosen in the East (E) direction or the South-East (SE) direction.
The algorithm starts from the top of the circle at (0, r) and uses the eight-way symmetry of a circle. After calculating one point, the remaining seven symmetric points are plotted automatically, making the algorithm very efficient.

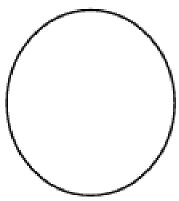
Algorithm:
1. Start the program.
2. Initialize the variables.
3. Call the initgraph() function.
4. Input radius r and circle centre (xc , yc), then set the coordinates for the first point on the circumference of a circle centred on the origin as:
5. Calculate the initial value of the decision parameter as:
6. Starting with k = 0 at each position xk , perform the following test. If pk < 0, the next point along the circle centred on (0, 0) is (xk +1, yk) and rp−=450),
7. Otherwise, the next point along the circle is (xk +1, yk-1) and
8. Determine symmetry points in the other seven octants
9. Move each calculated pixel position (x, y) onto the circular path centred at (xc , yc) to plot the coordinate values:
10. Repeat steps 3 to 5 until x >= y
11. Stop the graphics driver.
12. Stop the program.

Program:
```
#include<stdio.h>
#include<conio.h>
 #include<graphics.h>
void circlepoints(int,int);
void main()
{
        int x,y,p,r;
        int gdriver=DETECT, gmode;
        initgraph(&gdriver,&gmode,"C:\\tc\\bgi:");
        clrscr();
        printf("Enter the radius");
        scanf("%d",&r);
        x=0;y=r;p=1-r;
```

```
		while(x<y)
		{
		   x++;
		   if(p>0)
		   {
			p=p+2*(x-y)+1;
			y--;
		   }
		   else  p=p+2*x+1;
		   circlepoints(x,y);
		}
		getch();
		closegraph();
}
void circlepoints(int x,int y)
{
		putpixel(x+300,y+300,8);
		putpixel(x+300,-y+300,8);
		putpixel(-x+300,y+300,8);
		putpixel(-x+300,-y+300,8);
		putpixel(y+300,x+300,8);
		putpixel(y+300,-x+300,8);
		putpixel(-y+300,x+300,8);
		putpixel(-y+300,-x+300,8);
}
```

Output:

Enter the radius 50



Applications:

1. **Computer Graphics**: Drawing circles efficiently on raster displays.
2. **Computer-Aided Design (CAD)**: Designing gears, wheels, pipes, and circular mechanical components.
3. **Video Games**: Drawing balls, targets, wheels, coins, and circular obstacles.
4. **Graphical User Interfaces (GUI)**: Creating circular buttons, icons, clocks, and progress indicators.
5. **Animation**: Drawing circular paths and rotating objects.
6. **Image Processing**: Rendering and detecting circular shapes in digital images.

Viva Based Questions:

Q1. What is Bresenham's Circle Drawing Algorithm?
Answer: It is an efficient raster graphics algorithm used to draw circles using only integer arithmetic and decision parameters.

Q2. Why is Bresenham's Circle Algorithm preferred over the direct circle equation?
Answer: Because it avoids floating-point calculations and square root operations, making it faster and more efficient.

Q3. What is the initial decision parameter in Bresenham's Circle Algorithm?
Answer: The initial decision parameter is:

$$d = 3 - 2r$$

where r is the radius of the circle.

Q4. What is the starting point of the algorithm?
Answer: The algorithm starts at:

$$(x, y) = (0, r)$$

Q5. What is the role of the decision parameter?
Answer: The decision parameter determines whether the next pixel should be selected in the East (E) direction or the South-East (SE) direction to best approximate the circle.

Q6. Which property of a circle is used in this algorithm?
Answer: The algorithm uses the eight-way symmetry of a circle, allowing one calculated point to generate seven additional symmetric points.

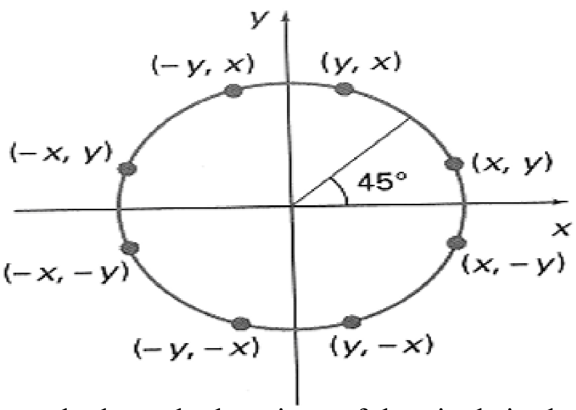
Q7. Why is Bresenham's Circle Algorithm efficient?
Answer: Because it uses only integer addition and subtraction, reducing computation time and making it suitable for real-time graphics.

PROGRAM NO. 4

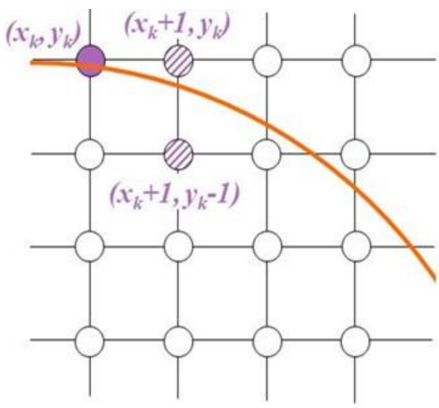
Aim: Write a program to implement the Midpoint circle drawing algorithm

Description: Circles have the property of being highly symmetrical, which is handy when it comes to drawing them on a display screen.

- We know that there are 360 degrees in a circle. First, we see that a circle is symmetrical about the x axis, so only the first 180 degrees need to be calculated.
- Next, we see that it's also symmetrical about the y axis, so now we only need to calculate the first 90 degrees.
- Finally, we see that the circle is also symmetrical about the 45-degree diagonal axis, so we only need to calculate the first 45 degrees.
- We only need to calculate the values on the border of the circle in the first octant. The other values may be determined by symmetry.



Bresenham's circle algorithm calculates the locations of the pixels in the first 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 eight octants of the circle. This is done till when the value of the y coordinate equals the x coordinate. The pixel positions for determining symmetry are given in the below algorithm.



- Assume that we have just plotted point (x_k, y_k)
- The next point is a choice between (x_k+1, y_k) and (x_k+1, y_k-1)
- We would like to choose the point that is nearest to the actual circle
- So we use decision parameter here to decide.

Algorithm:

1. Input radius r and circle centre (xc, yc), then set the coordinates for the first point on the circumference of a circle centred on the origin $(x_0, y_0) = (0, r)$

2. Calculate the initial value of the decision parameter as: $p_0 = \frac{5}{4} - r$

3. Starting with k = 0 at each position xk, perform the following test. If pk < 0, the next point along the circle centred on (0, 0) is (xk+1, yk) and: Otherwise the next point along the circle is (xk+1, yk-1) and: $p_{k+1} = p_k + 2x_{k+1} + 1$

4. Determine symmetry points in the other seven octants
$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

5. Move each calculated pixel position (x, y) onto the circular path centred at (xc, yc) to plot the coordinate values:

6. Repeat steps 3 to 5 until x >= $x = x + x_c$ $y = y + y_c$ y

Symmetric pixel positions:

- putpixel(xc+x,yc-y,GREEN); //For pixel (x,y)
- putpixel(xc+y,yc-x, GREEN); //For pixel (y,x)
- putpixel(xc+y,yc+x, GREEN); //For pixel (y,-x)
- putpixel(xc+x,yc+y, GREEN); //For pixel (x,-y)
- putpixel(xc-x,yc+y, GREEN); //For pixel (-x,-y)
- putpixel(xc-y,yc+x, GREEN); //For pixel (-y,-x)
- putpixel(xc-y,yc-x, GREEN); //For pixel (-y,x)
- putpixel(xc-x,yc-y, GREEN); //For pixel (-x,y)

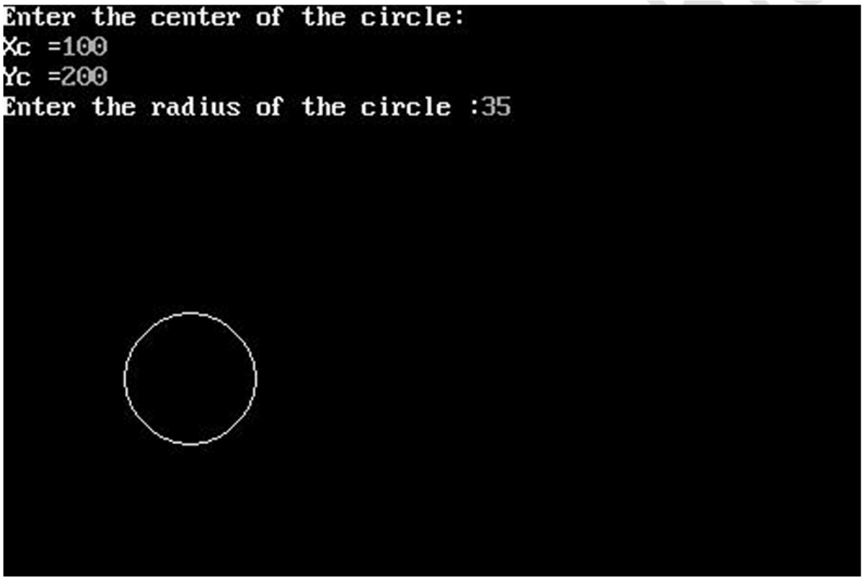
Program:

```
#include<dos.h>

#include<stdio.h>

#include<conio.h>

#include<graphics.h>

void draw_circle(int,int,int);

void symmetry(int,int,int,int);

void main()

{

 int xc,yc,R;
```

```
 int gd=DETECT,gm;
 initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
 printf("Enter the center of the circle:\n");
 printf("Xc =");
 scanf("%d",&xc);
 printf("Yc =");
 scanf("%d",&yc);
 printf("Enter the radius of the circle :");
 scanf("%d",&R);
 draw_circle(xc,yc,R);
 getch();
 closegraph();
}

void draw_circle(int xc,int yc,int rad)
{
 int x = 0;
 int y = rad;
 int p = 1-rad;
 symmetry(x,y,xc,yc);
 for(x=0;y>x;x++)
{
  if(p<0)
                p += 2*x + 3;
                else
                 {
                        p += 2*(x-y) + 5;
                           y--;
               }
            symmetry(x,y,xc,yc);
```

```
            delay(50);
    }
}
void symmetry(int x,int y,int xc,int yc)
{
 putpixel(xc+x,yc-y,GREEN);  //For pixel (x,y)
 delay(50);
 putpixel(xc+y,yc-x, GREEN);  //For pixel (y,x)
 delay(50);
 putpixel(xc+y,yc+x, GREEN);  //For pixel (y,-x)
delay(50);
 putpixel(xc+x,yc+y, GREEN);  //For pixel (x,-y)
 delay(50);
 putpixel(xc-x,yc+y, GREEN);  //For pixel (-x,-y)
 delay(50);
 putpixel(xc-y,yc+x, GREEN);  //For pixel (-y,-x)
 delay(50);
 putpixel(xc-y,yc-x, GREEN);  //For pixel (-y,x)
 delay(50);
 putpixel(xc-x,yc-y, GREEN);  //For pixel (-x,y)
 delay(50);
}
```

Output:



Applications:

1. Computer Graphics
   - Drawing circles and circular objects efficiently on raster displays.
2. Computer-Aided Design (CAD)
   - Creating circular components such as gears, wheels, pipes, and mechanical parts.
3. Video Game Development
   - Drawing circular objects like balls, wheels, targets, and game characters.
4. Graphical User Interfaces (GUI)
   - Creating circular buttons, icons, loading indicators, and progress rings.
5. Animation
   - Drawing circular motion paths and animated circular objects.
6. Scientific Visualization
   - Representing atoms, planets, molecules, and circular diagrams.
7. Digital Mapping and GIS
   - Drawing circular zones, buffer regions, and coverage areas.
8. Image Processing
   - Detecting and rendering circular shapes in digital images.
9. Simulation Software
   - Simulating rotating objects such as gears, pulleys, and wheels.
10. Educational Purpose
    - Teaching raster graphics, scan conversion, and computer graphics algorithms.

Viva Questions with Answers

Q1. What is the Midpoint Circle Drawing Algorithm?
Answer: It is a raster graphics algorithm used to draw a circle efficiently using only integer arithmetic and decision parameters.

Q2. Why is the Midpoint Circle Algorithm preferred over the direct circle equation?
Answer: Because it avoids floating-point calculations and square root operations, making it faster and more efficient.

Q3. What is the initial decision parameter of the Midpoint Circle Algorithm?
Answer: The initial decision parameter is: $P_0 = 1 - r$
where r is the radius of the circle.

Q4. Which property of a circle is used in the Midpoint Circle Algorithm?
Answer: The algorithm uses the eight-way symmetry of a circle, where calculating one point allows the remaining seven symmetric points to be plotted.

Q5. What are the starting coordinates of the algorithm?
Answer: The algorithm starts from: $(x, y) = (0, r)$

Q6. Why is the algorithm called the "Midpoint" algorithm?
Answer: Because it uses the midpoint between two candidate pixels to decide which pixel is closer to the actual circle.

Q7. What type of arithmetic is used in the Midpoint Circle Algorithm?
Answer: It uses integer arithmetic only.

Q8. What is the major advantage of the Midpoint Circle Algorithm?
Answer: It is fast, accurate, and avoids floating-point computations, making it suitable for real-time graphics applications.

Q9. What is the stopping condition in the Midpoint Circle Algorithm?
Answer: The algorithm continues until: $x \geq y$. At this point, all octants of the circle have been completed.

Q10. What is the time complexity of the Midpoint Circle Drawing Algorithm?
Answer: The time complexity is $O(r)$, where r is the radius of the circle.

PROGRAM NO. 5

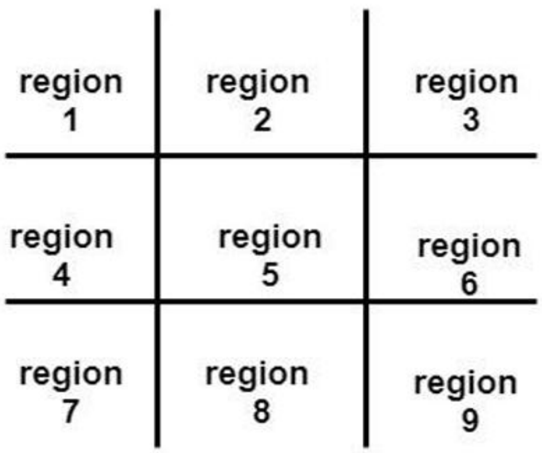
Aim: Write a program to perform Cohen Sutherland line clipping.

Description: 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

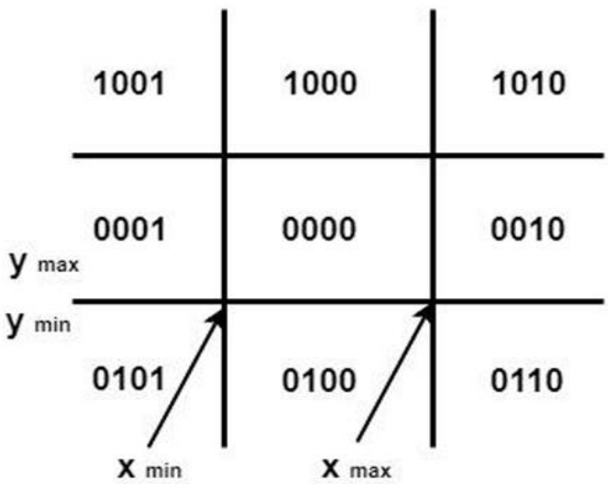
1. **Visible:** If a line lies within the window, i.e., both endpoints of the line lies within the window. A line is visible and will be displayed as it is.

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 (x1, y2) and B (x2, y2) are endpoints of line.
xmin, xmax are coordinates of the window.
ymin,ymax are also coordinates of the window.
x1>xmax
x2>xmax
y1>ymax
y2>ymax
x1<xmin
x2<xmin
y1<ymin
y2<ymin

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



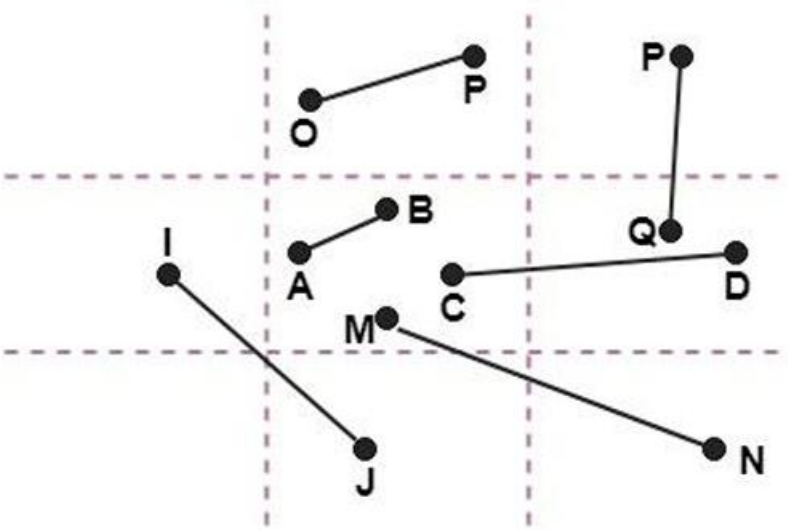
9 region



bits assigned to 9 regions

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

Following figure show lines of various types



Line PQ is an invisible line.
Line IJ are clipping candidates
Line MN are clipping candidate
Line CD are clipping candidate

Line AB is the visible case
Line OP is an invisible case

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:

1. Calculate positions of both endpoints of the line
2. Perform OR operation on both of these end-points
3. If the OR operation gives 0000
   Then
   line is considered to be visible
   else
   Perform AND operation on both endpoints
   If And ≠ 0000
   then the line is invisible
   else
   And=0000
   Line is considered the clipped case.
4. If a line is clipped case, find an intersection with boundaries of the window
   m=(y2-y1)(x2-x1)
   (a) If bit 1 is "1" line intersects with left boundary of rectangle window
   y3=y1+m(x-X1)
   where X = Xwmin
   where Xwminis the minimum value of X co-ordinate of window
   (b) If bit 2 is "1" line intersect with right boundary
   y3=y1+m(X-X1)
   where X = Xwmax
   where X more is maximum value of X co-ordinate of the window
   (c) If bit 3 is "1" line intersects with bottom boundary

X3=X1+(y-y1)/m
where y = ywmin
ywmin is the minimum value of Y co-ordinate of the window

(d) If bit 4 is "1" line intersects with the top boundary
X3=X1+(y-y1)/m
where y = ywmax
ywmax is the maximum value of Y co-ordinate of the window

Program:

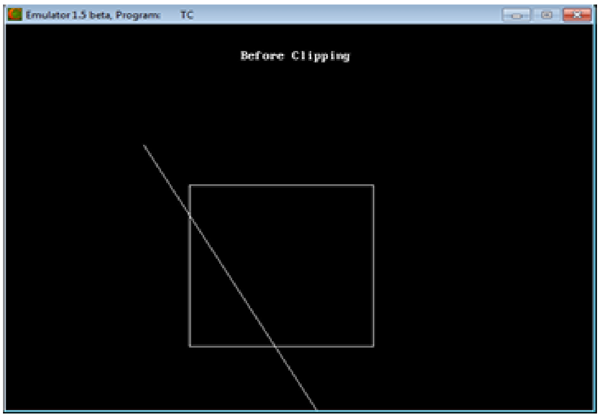
```
#include<stdio.h>
#include<graphics.h>
#include<conio.h>

#define INSIDE 0
#define LEFT   1
#define RIGHT  2
#define BOTTOM 4
#define TOP    8

int xmin, ymin, xmax, ymax;

int computeCode(double x, double y)
{
   int code = INSIDE;

   if (x < xmin)
      code |= LEFT;
   else if (x > xmax)
      code |= RIGHT;

   if (y < ymin)
      code |= TOP;
   else if (y > ymax)
      code |= BOTTOM;

   return code;
}

void cohenSutherlandClip(double x1,double y1,double x2,double y2)
{
   int code1 = computeCode(x1,y1);
   int code2 = computeCode(x2,y2);
   int accept = 0;

   while(1)
   {
```

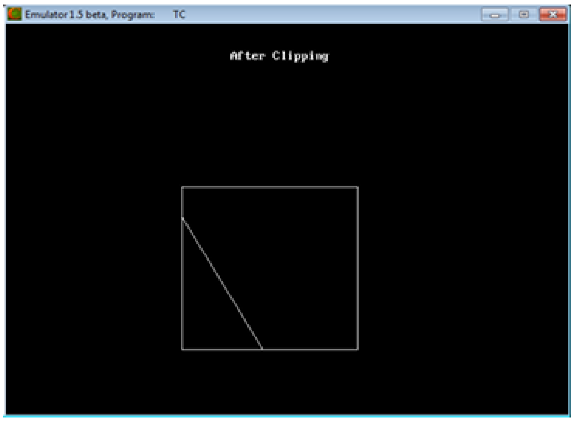
```
if((code1==0) && (code2==0))
{
    accept = 1;
    break;
}
else if(code1 & code2)
{
    break;
}
else
{
    double x,y;
    int code_out;

    if(code1 != 0)
        code_out = code1;
    else
        code_out = code2;

    if(code_out & TOP)
    {
        x = x1 + (x2-x1)*(ymin-y1)/(y2-y1);
        y = ymin;
    }
    else if(code_out & BOTTOM)
    {
        x = x1 + (x2-x1)*(ymax-y1)/(y2-y1);
        y = ymax;
    }
    else if(code_out & RIGHT)
    {
        y = y1 + (y2-y1)*(xmax-x1)/(x2-x1);
        x = xmax;
    }
    else if(code_out & LEFT)
    {
        y = y1 + (y2-y1)*(xmin-x1)/(x2-x1);
        x = xmin;
    }

    if(code_out == code1)
    {
        x1 = x;
        y1 = y;
        code1 = computeCode(x1,y1);
    }
```

```
            else
            {
               x2 = x;
               y2 = y;
               code2 = computeCode(x2,y2);
            }
        }
    }

    if(accept)
    {
        rectangle(xmin,ymin,xmax,ymax);
        line((int)x1,(int)y1,(int)x2,(int)y2);
    }
}

void main()
{
    int gd=DETECT,gm;
    double x1,y1,x2,y2;

    printf("Enter the clip window coordinates: ");
    scanf("%d%d%d%d",&xmin,&ymin,&xmax,&ymax);

    printf("Enter the line coordinates: ");
    scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2);

    initgraph(&gd,&gm,"C:\\TurboC3\\BGI");

    outtextxy(220,20,"Before Clipping");

    rectangle(xmin,ymin,xmax,ymax);
    line((int)x1,(int)y1,(int)x2,(int)y2);

    getch();

    cleardevice();

    outtextxy(225,20,"After Clipping");

    cohenSutherlandClip(x1,y1,x2,y2);

    getch();
    closegraph();
}
```

Output:
Enter the clip window coordinates:
200 200 400 400

Enter the line coordinates:
150 150 350 500





Applications

1. **Computer Graphics**
   - Displays only the visible portion of lines inside the viewing window.
2. **Computer-Aided Design (CAD)**
   - Clips engineering drawings and technical diagrams to the display area.
3. **GIS (Geographical Information Systems)**
   - Displays only the map features that lie within the selected region.
4. **Computer Games**
   - Removes objects or line segments outside the player's viewing area to improve rendering speed.
5. **GUI (Graphical User Interface)**
   - Clips windows, borders, and graphical elements that extend outside the display area.
6. **Animation**
   - Displays only the visible parts of moving objects within the screen.
7. **Plotters and Printers**
   - Prints only the portion of drawings that lies within the printable area.

Viva Questions with Answers

Q1. What is the Cohen–Sutherland Line Clipping Algorithm?
Answer: It is a line clipping algorithm used to determine and display only the visible portion of a line inside a rectangular clipping window.

Q2. What is clipping?
Answer: Clipping is the process of removing the portions of graphical objects that lie outside a specified viewing region (clipping window).

Q3. What is a clipping window?
Answer: A clipping window is a rectangular region that defines the visible area of the graphics screen.

Q4. What is the main idea behind the Cohen–Sutherland algorithm?
Answer: The algorithm assigns region codes (Outcodes) to the endpoints of a line and uses these codes to determine whether the line is completely visible, completely invisible, or partially visible.

Q5. What are Outcodes (Region Codes)?
Answer: Outcodes are 4-bit binary codes assigned to each endpoint of a line based on its position relative to the clipping window.
The four bits represent:

- Left
- Right
- Bottom
- Top

Q6. How many region codes are possible in the Cohen–Sutherland algorithm?
Answer: There are 9 regions (1 inside region and 8 outside regions). Each point is represented using a 4-bit region code.

Q7. What are the limitations of the Cohen–Sutherland algorithm?
Answer:

- Works primarily for rectangular clipping windows.
- Not suitable for arbitrary polygon clipping.
- Requires extensions for non-rectangular clipping regions.

Q8. Which clipping algorithm is considered more efficient than Cohen–Sutherland for many cases?
Answer: The Liang–Barsky Line Clipping Algorithm is often more efficient because it uses parametric equations and performs fewer intersection calculations.

Q9. What is the time complexity of the Cohen–Sutherland algorithm?
Answer: The average time complexity is O(1) for most practical cases because it quickly accepts or rejects lines using region codes.

Q10. Why is the Cohen–Sutherland algorithm widely used?
Answer: It is widely used because it is simple, efficient, uses bitwise operations, and quickly determines the visible portion of a line in a rectangular clipping window.

PROGRAM NO. 6

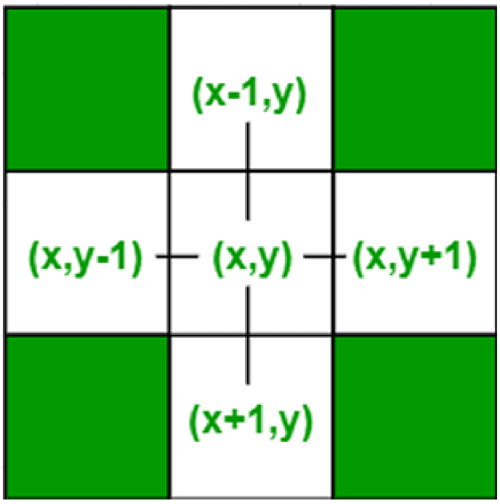
Aim: Write a program to implement polygon filling.

Description: The **Boundary Fill Algorithm** is a seed-fill algorithm used in computer graphics to fill the interior of a closed polygon with a specified fill color. The algorithm starts from a point (called the seed point) inside the polygon and recursively fills all neighboring pixels until it reaches the boundary color. The boundary of the polygon is drawn using a different color from the fill color.
The Boundary Fill Algorithm can be implemented using 4-connected or 8-connected pixels.

4-Connected Pixels
In the 4-connected Boundary Fill Algorithm, after filling a pixel, the algorithm recursively checks and fills its four neighbouring pixels:

- Left (x − 1, y)
- Right (x + 1, y)
- Top (x, y − 1)
- Bottom (x, y + 1)

Since only four neighboring pixels are considered, the filled region is called a **4-connected region**. This method is simple and efficient for polygons with well-defined boundaries but may leave gaps if the boundary is not completely connected.



Algorithm:

1. Start.
2. Initialize the graphics mode.
3. Draw a closed polygon using the line() function.
4. Select a seed point inside the polygon.
5. Read the color of the current pixel.
6. If the current pixel is neither the boundary color nor the fill color:
   - Paint the current pixel with the fill color.
   - Recursively call the Boundary Fill function for:
     - Right pixel (x+1, y)
     - Left pixel (x−1, y)
     - Upper pixel (x, y−1)
     - Lower pixel (x, y+1)

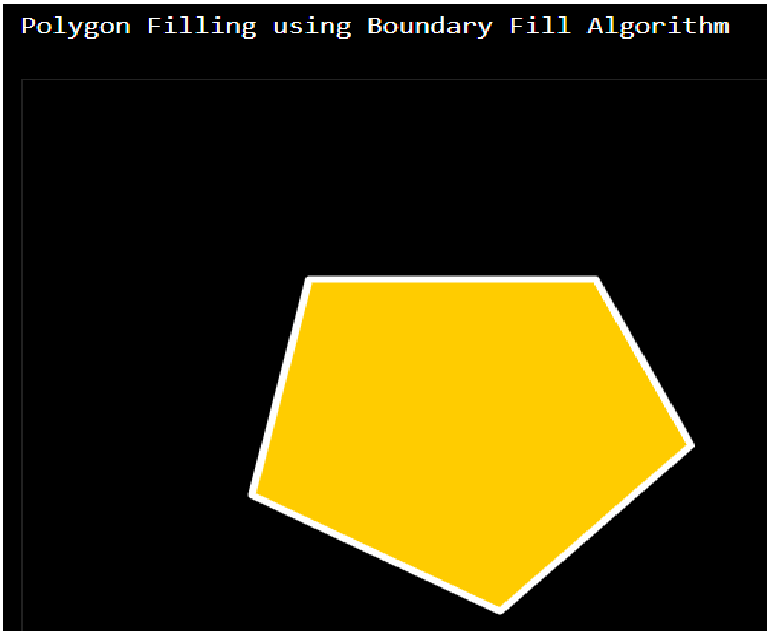
7. Continue until all interior pixels are filled.
8. Close the graphics window.
9. Stop.

Program:

```
#include<stdio.h>
#include<graphics.h>
#include<conio.h>

void boundaryfill(int x,int y,int fill,int boundary)
{
   int current;

   current = getpixel(x,y);

   if(current != boundary && current != fill)
   {
      putpixel(x,y,fill);

      boundaryfill(x+1,y,fill,boundary);
      boundaryfill(x-1,y,fill,boundary);
      boundaryfill(x,y+1,fill,boundary);
      boundaryfill(x,y-1,fill,boundary);
   }
}

void main()
{
   int gd=DETECT,gm;

   initgraph(&gd,&gm,"C:\\TurboC3\\BGI");

   outtextxy(20,20,"Polygon Filling using Boundary Fill Algorithm");

   /* Draw Polygon */
   setcolor(WHITE);

   line(150,120,300,120);
   line(300,120,350,220);
   line(350,220,250,320);
   line(250,320,120,250);
   line(120,250,150,120);

   /* Fill Polygon */
   boundaryfill(220,200,YELLOW,WHITE);
```

```
    getch();
    closegraph();
}
```

Output:



Applications

1. Paint and drawing software (MS Paint, Photoshop).
2. Computer-Aided Design (CAD).
3. Computer games for filling graphical objects.
4. Geographic Information Systems (GIS).
5. Animation and multimedia applications.

Viva Questions

Q1. What is the Boundary Fill Algorithm?
Answer: It is a seed-fill algorithm used to fill the interior of a closed polygon starting from a seed point until the boundary color is reached.

Q2. What is a seed point?
Answer: A seed point is a pixel selected inside the polygon from where the filling process begins.

Q3. What is the difference between 4-connected and 8-connected Boundary Fill?
Answer:

- 4-connected: Checks four neighboring pixels (left, right, top, bottom).
- 8-connected: Checks eight neighboring pixels, including the four diagonal neighbors.

Q5. Why must the boundary be closed?
Answer: If the boundary is not completely closed, the fill operation may leak outside the intended region.

PROGRAM NO. 7

Aim: Write a program to move an object using the concept of 2-D translation transformation.

Description: A 2-D Translation Transformation is a geometric transformation that moves an object from one position to another without changing its shape, size, or orientation.
In translation, every point of the object is shifted by a fixed distance along the x-axis and y-axis.
If a point has coordinates (x, y), then after translation its new coordinates become:

$$x' = x + t_x$$
$$y' = y + t_y$$

where:

- t_x= Translation distance along x-axis
- t_y= Translation distance along y-axis

Translation is represented using the following transformation matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Algorithm:

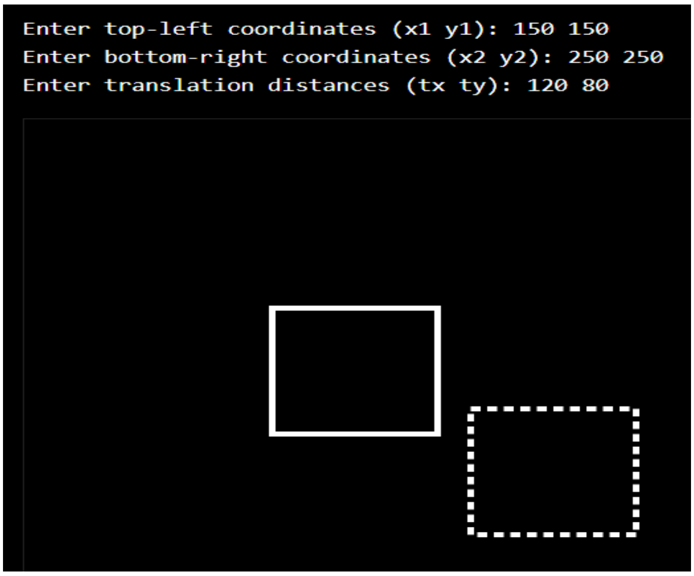
1. Start the program.
2. Initialize the graphics mode.
3. Input the coordinates of the rectangle.
4. Draw the original rectangle.
5. Input translation distances $t_x$and t_y.
6. Calculate the new coordinates:
   - $x_1 = x_1 + t_x$
   - $y_1 = y_1 + t_y$
   - $x_2 = x_2 + t_x$
   - $y_2 = y_2 + t_y$
7. Draw the translated rectangle.
8. Display both the original and translated objects.
9. Close the graphics mode.
10. Stop.

Program:

```
#include <stdio.h>
#include <graphics.h>
#include <conio.h>

int main() {
    int gd = DETECT, gm;
    int x1, y1, x2, y2, x3, y3;     // Original triangle vertices
    int tx, ty;                     // Translation factors
```

```
    // Initialize original triangle (example)
    x1 = 100; y1 = 100;
    x2 = 200; y2 = 100;
    x3 = 150; y3 = 200;

    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");  // Adjust path as per your setup

    // Draw Original Object (White)
    setcolor(WHITE);
    line(x1, y1, x2, y2);
    line(x2, y2, x3, y3);
    line(x3, y3, x1, y1);
    outtextxy(x1-20, y1-20, "Original");

    printf("Enter translation factors tx and ty: ");
    scanf("%d %d", &tx, &ty);

    // Translated coordinates
    int x1_new = x1 + tx;
    int y1_new = y1 + ty;
    int x2_new = x2 + tx;
    int y2_new = y2 + ty;
    int x3_new = x3 + tx;
    int y3_new = y3 + ty;

    // Draw Translated Object (Yellow)
    setcolor(YELLOW);
    line(x1_new, y1_new, x2_new, y2_new);
    line(x2_new, y2_new, x3_new, y3_new);
    line(x3_new, y3_new, x1_new, y1_new);
    outtextxy(x1_new-20, y1_new-30, "Translated");

    getch();
    closegraph();
    return 0;
}
```

Output:



Applications:
Computer Graphics

- Moving objects from one position to another on the screen.

Animation

- Moving characters, vehicles, and objects smoothly.

Computer Games

- Character movement, enemy movement, and scrolling backgrounds.

CAD (Computer-Aided Design)

- Repositioning engineering drawings and machine components.

GUI Development

- Moving windows, icons, and dialog boxes.

Robotics

- Simulating robot movement in two-dimensional space.

Image Processing

- Image shifting and alignment.

GIS (Geographical Information Systems)

- Shifting maps and geographical objects.

Viva Questions with Answers

Q1. What is a 2-D translation transformation?
Answer: A transformation that moves an object from one position to another without changing its shape, size, or orientation.

Q2. What is meant by translation?
Answer: Translation means shifting every point of an object by a fixed distance along the x-axis and y-axis.

Q3. What are t_x and t_y?
Answer:

- t_x is the translation distance along the x-axis.
- t_y is the translation distance along the y-axis.

Q4. Write the translation equations.
Answer:

$$x' = x + t_x$$
$$y' = y + t_y$$

Q5. Does translation change the size of an object?
Answer: No. Translation changes only the position of the object.

Q6. Does translation change the orientation of an object?
Answer: No. The object's orientation remains unchanged.

PROGRAM NO. 8

Aim: Write a program to move an object using the concept of 2-D rotation transformation.

Description: A 2-D Rotation Transformation is a geometric transformation used in computer graphics to rotate an object about a fixed point, usually the origin, by a specified angle. During rotation, the shape and size of the object remain unchanged, but its orientation changes. The direction of rotation may be clockwise or anticlockwise depending on the angle of rotation. If a point has coordinates (x, y) and is rotated by an angle θ about the origin, its new coordinates become (x', y'), where x' = x cos θ − y sin θ and y' = x sin θ + y cos θ. Rotation is represented using a transformation matrix and is widely used in computer graphics, animation, robotics, gaming, CAD software, image processing, and simulation applications. Since the distances between all points remain unchanged after rotation, it is considered a rigid-body transformation.

Rotation Transformation Matrix

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

where:

- θ = Angle of rotation
- x, y = Original coordinates
- x', y' = Rotated coordinates

Algorithm:

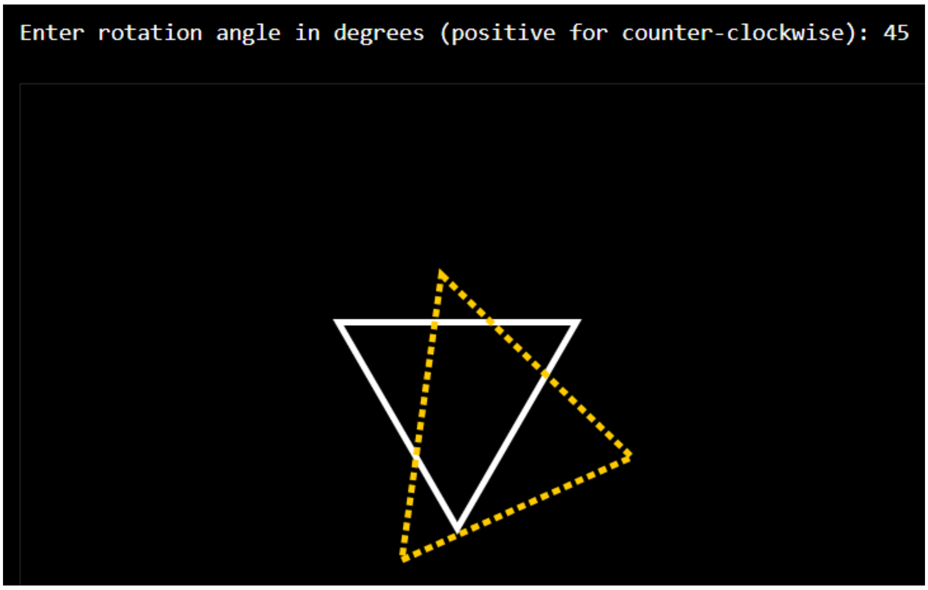
1. Start the program.
2. Initialize the graphics mode.
3. Input the coordinates of the rectangle.
4. Draw the original rectangle.
5. Input the angle of rotation.
6. Convert the angle into radians.
7. Apply the rotation equations to all four vertices.
8. Draw the rotated object.
9. Display both the original and rotated objects.
10. Close the graphics mode.
11. Stop.

Program:

```
#include <stdio.h>
#include <graphics.h>
#include <conio.h>
#include <math.h>

int main() {
    int gd = DETECT, gm;
    float x1, y1, x2, y2, x3, y3;   // Original
    float x1r, y1r, x2r, y2r, x3r, y3r; // Rotated
```

```c
    float theta, theta_rad;
    int xc = 300, yc = 250;         // Pivot point (center of screen)

    // Original Triangle
    x1 = 200; y1 = 150;
    x2 = 350; y2 = 150;
    x3 = 275; y3 = 280;

    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");

    // Draw Original (White)
    setcolor(WHITE);
    line(x1, y1, x2, y2);
    line(x2, y2, x3, y3);
    line(x3, y3, x1, y1);
    outtextxy(180, 120, "Original");

    printf("Enter rotation angle in degrees (positive for counter-clockwise): ");
    scanf("%f", &theta);

    theta_rad = theta * M_PI / 180.0;

    // Rotate around pivot (xc, yc)
    x1r = xc + (x1 - xc) * cos(theta_rad) - (y1 - yc) * sin(theta_rad);
    y1r = yc + (x1 - xc) * sin(theta_rad) + (y1 - yc) * cos(theta_rad);

    x2r = xc + (x2 - xc) * cos(theta_rad) - (y2 - yc) * sin(theta_rad);
    y2r = yc + (x2 - xc) * sin(theta_rad) + (y2 - yc) * cos(theta_rad);

    x3r = xc + (x3 - xc) * cos(theta_rad) - (y3 - yc) * sin(theta_rad);
    y3r = yc + (x3 - xc) * sin(theta_rad) + (y3 - yc) * cos(theta_rad);

    // Draw Rotated (Yellow)
    setcolor(YELLOW);
    line(x1r, y1r, x2r, y2r);
    line(x2r, y2r, x3r, y3r);
    line(x3r, y3r, x1r, y1r);
    outtextxy(x1r-30, y1r-30, "Rotated");

    getch();
    closegraph();
    return 0;
}
```

Output:



Applications:

1. Computer animation for rotating characters and objects.
2. Video game development for rotating vehicles, wheels, and sprites.
3. Computer-Aided Design (CAD) for rotating machine components and engineering drawings.
4. Robotics for changing the orientation of robotic arms.
5. Image processing for rotating images.
6. GIS (Geographical Information Systems) for map rotation.

Viva Based Questions:

Q1. What is a 2-D Rotation Transformation?
Answer: A transformation that rotates an object by a specified angle about a fixed point without changing its shape or size.

Q2. Does rotation change the size of an object?
Answer: No. Rotation changes only the orientation of the object.

Q3. What is the standard rotation point?
Answer: The origin (0,0).

Q4. Write the rotation equations.
Answer:

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

Q8. What is the formula to convert degrees into radians?
Answer:

$$\text{Radians} = \frac{\pi \times \text{Degrees}}{180}$$

PROGRAM NO. 9

Aim: Write a program to move an object using the concept of 2-D scaling transformation.

Description: A 2-D Scaling Transformation is a geometric transformation used in computer graphics to increase or decrease the size of an object. The scaling operation changes the dimensions of an object by multiplying its coordinates with scaling factors along the x-axis and y-axis. The shape of the object remains the same if both scaling factors are equal (uniform scaling); otherwise, the object is stretched or compressed in one direction (non-uniform scaling). If a point has coordinates (x, y), then after scaling, its new coordinates become (x', y'), where x' = x × Sx and y' = y × Sy. Here, Sx and Sy are the scaling factors along the x-axis and y-axis, respectively. If the scaling factor is greater than 1, the object enlarges, whereas if it lies between 0 and 1, the object shrinks. Scaling is widely used in computer graphics, image processing, CAD software, animation, gaming, and graphical user interfaces for resizing objects without altering their basic structure.

Scaling Transformation Matrix

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

where

- **Sx** = Scaling factor along x-axis
- **Sy** = Scaling factor along y-axis

Algorithm:

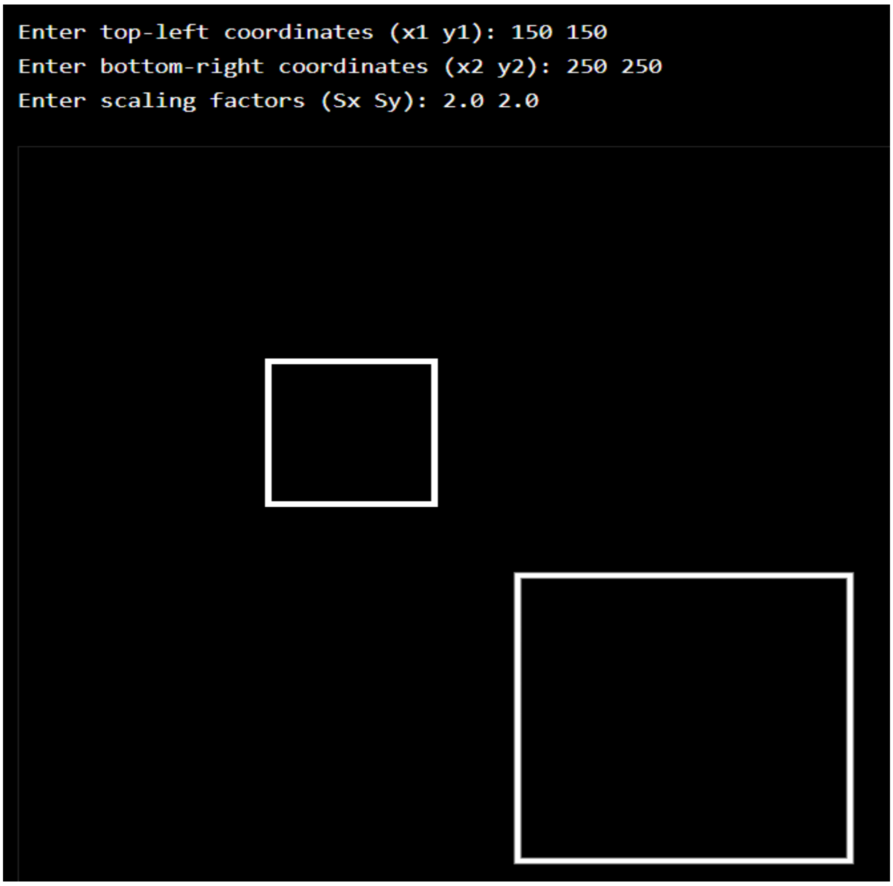
1. Start the program.
2. Initialize the graphics mode.
3. Input the coordinates of the rectangle.
4. Draw the original rectangle.
5. Input the scaling factors **Sx** and **Sy**.
6. Calculate the new coordinates:
    - x1 = x1 × Sx
    - y1 = y1 × Sy
    - x2 = x2 × Sx
    - y2 = y2 × Sy
7. Draw the scaled rectangle.
8. Display both the original and scaled objects.
9. Close the graphics mode.
10. Stop.

Program:

```
#include<stdio.h>
#include<graphics.h>
#include<conio.h>

void main()
{
```

```
    int gd=DETECT,gm;
    int x1,y1,x2,y2;
    float sx,sy;

    initgraph(&gd,&gm,"C:\\TurboC3\\BGI");

    printf("Enter top-left coordinates (x1 y1): ");
    scanf("%d%d",&x1,&y1);

    printf("Enter bottom-right coordinates (x2 y2): ");
    scanf("%d%d",&x2,&y2);

    printf("Enter scaling factors (Sx Sy): ");
    scanf("%f%f",&sx,&sy);

    /* Original Rectangle */
    setcolor(WHITE);
    rectangle(x1,y1,x2,y2);

    getch();

    /* Scaled Rectangle */
    setcolor(YELLOW);
    rectangle((int)(x1*sx),(int)(y1*sy),
          (int)(x2*sx),(int)(y2*sy));

    getch();
    closegraph();
}
```

Output:



Applications:

1. Resizing images in image editing software.
2. Zoom In and Zoom Out operations in graphics applications.
3. Computer-Aided Design (CAD) for resizing engineering drawings.
4. Animation for enlarging or shrinking objects.
5. Video games for resizing characters and game objects.

Viva Questions:

Q1. What is a 2-D Scaling Transformation?
Answer: A geometric transformation that changes the size of an object by multiplying its coordinates with scaling factors.

Q2. What are scaling factors?
Answer: Scaling factors (Sx and Sy) determine how much the object is enlarged or reduced along the x-axis and y-axis.

Q3. Write the scaling equations.
Answer:

$$x' = x \times S_x$$
$$y' = y \times S_y$$

Q4. What happens if Sx = Sy?
Answer: The object undergoes uniform scaling, where its proportions remain unchanged.

Q5. Does scaling change the orientation of an object?
Answer: No. Scaling changes only the size of the object, not its orientation.

Q6. What is the scaling transformation matrix?
Answer:

$$\begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

Q7. Name some applications of scaling transformation.
Answer: Image resizing, CAD, animation, gaming, GUI development, map zooming, medical imaging, scientific visualization, printing, and publishing.

Q8. What is the major difference between translation and scaling?
Answer: Translation changes the position of an object, whereas scaling changes its size.

Q9. Can scaling distort an object?
Answer: Yes. Non-uniform scaling (when Sx ≠ Sy) can distort the object's proportions.

PROGRAM NO. 10

Aim: Write a program to move an object using the concept of 2-D Shearing Transformation.

Description: A 2-D Shearing Transformation is a geometric transformation used in computer graphics to distort or slant the shape of an object along the x-axis or y-axis without changing its area. Unlike translation, rotation, or scaling, shearing changes the shape of an object while keeping one of its dimensions unchanged. During shearing, every point of the object is displaced by an amount proportional to its distance from a reference axis. There are two types of shearing: X-Shearing, in which the x-coordinate changes while the y-coordinate remains unchanged, and Y-Shearing, in which the y-coordinate changes while the x-coordinate remains unchanged. Shearing is represented using transformation matrices and is widely used in computer graphics, animation, image processing, CAD software, and digital design for creating slanted or inclined objects.
For X-Shearing, the transformed coordinates are:

$$x' = x + Sh_x \times y$$
$$y' = y$$

For Y-Shearing, the transformed coordinates are:

$$x' = x$$
$$y' = y + Sh_y \times x$$

where:

- Shx = Shearing factor along the x-axis
- Shy = Shearing factor along the y-axis

Shearing Transformation Matrix
X-Shearing

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & Sh_x \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Y-Shearing

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ Sh_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Algorithm:

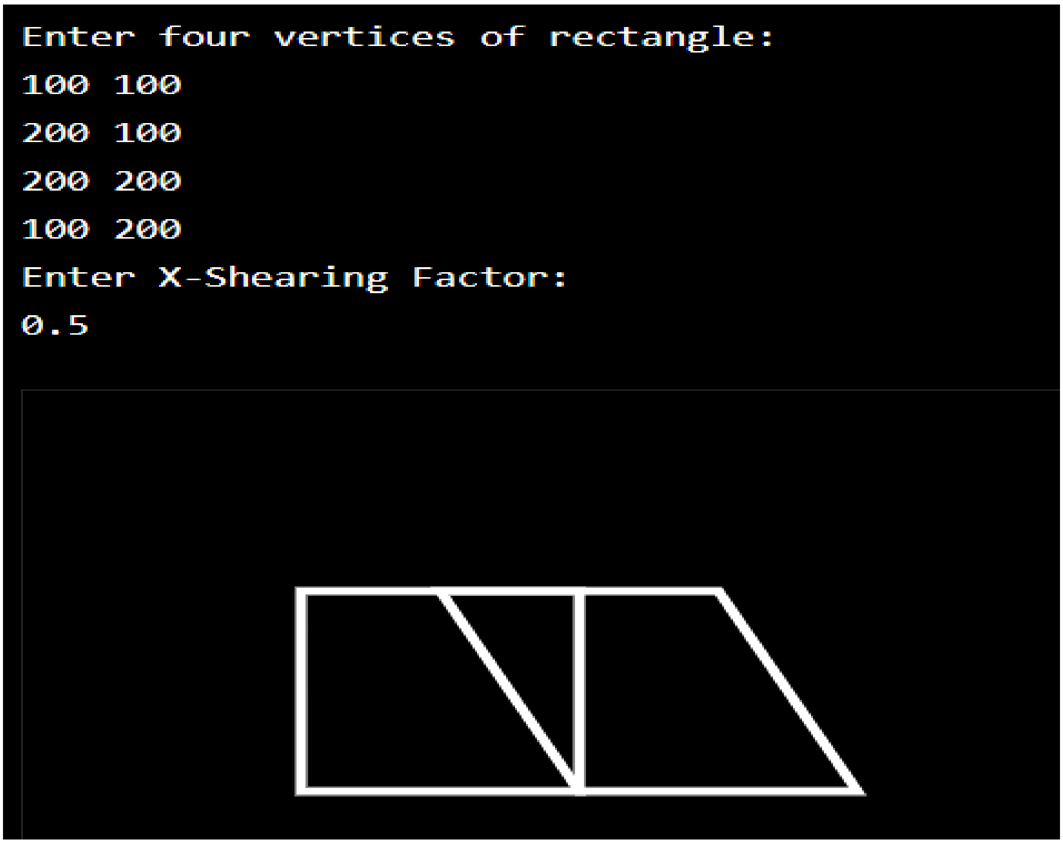
1. Start the program.
2. Initialize the graphics mode.
3. Input the four vertices of the rectangle.
4. Draw the original rectangle.
5. Input the shearing factor.
6. Apply the X-shearing transformation to all vertices.
7. Calculate the new coordinates using:
   - x' = x + Shx × y
   - y' = y
8. Draw the sheared rectangle.
9. Display both the original and transformed objects.
10. Close the graphics mode.
11. Stop.

Program:

```
#include<stdio.h>
#include<graphics.h>
#include<conio.h>

void main()
{
   int gd=DETECT,gm;
   int x[4],y[4];
   float shx;
   int sx[4],sy[4];
   int i;

   initgraph(&gd,&gm,"C:\\TurboC3\\BGI");

   printf("Enter four vertices of rectangle:\n");

   for(i=0;i<4;i++)
   {
      printf("Vertex %d (x y): ",i+1);
      scanf("%d%d",&x[i],&y[i]);
   }

   printf("Enter X-Shearing Factor: ");
   scanf("%f",&shx);

   setcolor(WHITE);

   for(i=0;i<3;i++)
      line(x[i],y[i],x[i+1],y[i+1]);

   line(x[3],y[3],x[0],y[0]);

   getch();

   for(i=0;i<4;i++)
   {
      sx[i]=x[i]+shx*y[i];
      sy[i]=y[i];
   }

   setcolor(YELLOW);

   for(i=0;i<3;i++)
      line(sx[i],sy[i],sx[i+1],sy[i+1]);
```

```
    line(sx[3],sy[3],sx[0],sy[0]);

    getch();

    closegraph();
}
```

Output:



Applications:

1. Creating slanted text and italic fonts.
2. Computer animation for simulating object deformation.
3. Image processing and image correction.
4. CAD software for modifying engineering drawings.
5. Architectural and civil engineering designs.

Viva Questions

Q1. What is a 2-D Shearing Transformation?
Answer: A transformation that slants or distorts an object by shifting its coordinates along one axis while keeping the other axis unchanged.

Q2. What are the two types of shearing?
Answer:

1. X-Shearing
2. Y-Shearing

Q3. Write the equation for X-Shearing.
Answer:

$$x' = x + Sh_x \times y$$
$$y' = y$$

Q4. Write the equation for Y-Shearing.
Answer:

$$x' = x$$
$$y' = y + Sh_y \times x$$

Q5. What is a shearing factor?
Answer: The shearing factor (Shx or Shy) determines the amount by which the object is slanted.

Q6. Does shearing change the size of an object?
Answer: It changes the shape of the object. The area may remain the same in ideal transformations, but the object becomes slanted.

Q7. Does shearing preserve angles?
Answer: No. Shearing changes the angles between the sides of an object.

Q8. What is the difference between X-shearing and Y-shearing?
Answer:

- In X-shearing, the x-coordinate changes while the y-coordinate remains unchanged.
- In Y-shearing, the y-coordinate changes while the x-coordinate remains unchanged.

Q9. Name some applications of shearing.
Answer: Animation, CAD, image processing, gaming, logo designing, digital art, and perspective effects.

Q10. What is the advantage of shearing?
Answer: It efficiently creates slanted and perspective-like effects and is widely used in graphics and design applications.