SILVER OAK UNIVERSITY
EDUCATION TO INNOVATION

School of Technology, Design & Computer Application
Silver Oak College of Engineering and Technology
Bachelor of Technology
Information Technology

Semester:	IV	Academic Year:	2024-25
Course Name:	Computer Graphics with AR/VR and Metaverse	Course Code:	1010043227

Mid Semester Question Bank

Sr. No.	Questions With Answer	Marks
Unit-1 Basic of Computer Graphics Primitives		
1	What is Computer Graphics?	4

Answer 1:

Computer graphics is an art of drawing pictures, lines, charts, etc. using computers with the help of programming. Computer graphics image is made up of a number of pixels.

Pixel: Pixel is the smallest addressable graphical unit represented on the computer screen.

- Computer is an information processing machine. User needs to communicate with the computer and the computer graphics is one of the most effective and commonly used ways of communication with the user.
- It displays the information in the form of graphical objects such as pictures, charts, diagrams and graphs.
- Graphical objects convey more information in less time and easily understandable formats, for example statically graphs shown in stock exchange.
- In computer graphics, pictures or graphics objects are presented as a collection of discrete pixels.
- We can control the intensity and color of pixels which decide how a picture looks like.

- The special procedure determines which pixel will provide the best approximation to the desired picture or graphics object; this process is known as Rasterization.
- The process of representing a continuous picture or graphics object as a collection of discrete pixels is called Scan Conversion.

2	**Applications of Computer Graphics?**	**4**

Answer 2:

User interface: Visual objects which we observe on screen which communicate with the user are one of the most useful applications of the computer graphics. Ex.App Icon

Plotting of graphics and charts: in industry, business, government and educational organizations drawing like bars, pie-charts, and histograms are very useful for quick and good decision making. Ex. Stock Market

Office automation and desktop publishing: It is used for creation and dissemination of information. It is used in in-house creation and printing of documents which contain text, tables, graphs and other forms of drawn or scanned images or pictures.

Computer aided drafting and design: It uses graphics to design components and systems such as automobile bodies, structures of building etc. Ex. Automobile Parts Designing

Simulation and animation: Use of graphics in simulation makes mathematical models and mechanical systems more realistic and easy to study. Ex. Cartoon and Animation Movies

Art and commerce: There are many tools provided by graphics which allow users to make their picture animated and attractive which are used in advertising. Ex. Creative Pictures

Process control: Now a day's automation is used which is graphically displayed on the screen.

Cartography: Computer graphics are also used to represent geographic maps, weather maps, oceanographic charts etc. Ex. Geographic maps

Education and training: Computer graphics can be used to generate models of physical, financial and economic systems. These models can be used as educational aids.

Ex .Models of Physics

Image processing: It is used to process images by changing the property of the image. Ex. Photo Editing

3	**Explain points, lines, circles and ellipses as primitives basics.**	**6**

Answer 3:

1. Points

A point is the most basic geometric entity. It has a location but no size, length, or area. It is defined by a set of coordinates in a given space (e.g., 2D or 3D).

- Representation in 2D: P(x,y)
- Representation in 3D:P(x,y,z)

In computer graphics, points are often used as vertices to define polygons and other complex structures.

---

2. Lines

- A line is an infinite set of points extending in both directions. It has length but no thickness.
- **Equation of a line in 2D (slope-intercept form): Y = mx + b** where is the slope and is the y-intercept.
- **Parametric equation of a line:P(t)=P0+t⋅ d** ,where P0 is a starting point, d is the direction vector, and t is a scalar parameter
- Line segment: A line segment is a finite portion of a line between two points.
- In computer graphics, lines are used for wireframe models, ray tracing, and edge detection.

---

3. Circles

- A circle is a set of points that are all at the same distance (radius) from a given center point.
- Equation of a circle (centered at (h,k)): $(x-h)^2 + (y-k)^2 = r^2$ where r is the radius.
- Circles are commonly used in graphics for rendering objects, UI elements, collision detection, and more.

---

4. Ellipses

- An ellipse is a generalization of a circle, where the sum of distances from any point on the ellipse to two fixed points (foci) is constant.
- Standard equation of an ellipse (centered at):

$$\frac{(x-h)^2}{a^2} + \frac{(y-k)^2}{b^2} = 1$$

  where a is the semi-major axis and b is the semi-minor axis.
- Ellipses are used in computer graphics for rendering, physics simulations, and even in planetary orbits.

4.	**Explain Boundary Fill.**	**5**

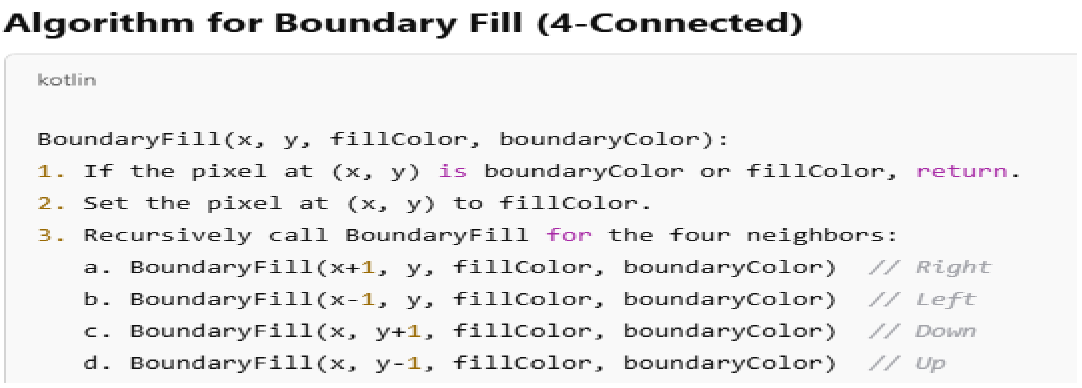
Answer 4:

Definition:

The Boundary Fill Algorithm is a recursive method used in computer graphics to fill a closed region with a specific color. It starts from a seed point inside the boundary and spreads outward until it reaches the boundary color.

Working Principle:

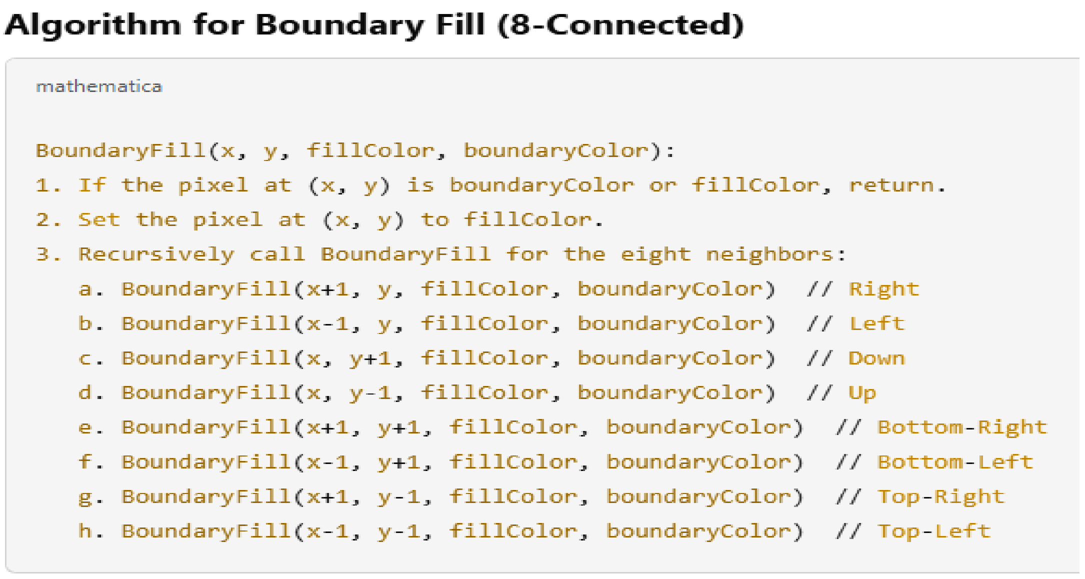
1. Choose a seed point (x, y) inside the region.
2. Check if the current pixel is the boundary color or the fill color.
3. If it is neither, color the pixel with the desired fill color.
4. Recursively apply the process to neighboring pixels (either 4-connected or 8-connected).



Algorithm for Boundary Fill (4-Connected)

kotlin

```
BoundaryFill(x, y, fillColor, boundaryColor):
1. If the pixel at (x, y) is boundaryColor or fillColor, return.
2. Set the pixel at (x, y) to fillColor.
3. Recursively call BoundaryFill for the four neighbors:
   a. BoundaryFill(x+1, y, fillColor, boundaryColor)  // Right
   b. BoundaryFill(x-1, y, fillColor, boundaryColor)  // Left
   c. BoundaryFill(x, y+1, fillColor, boundaryColor)  // Down
   d. BoundaryFill(x, y-1, fillColor, boundaryColor)  // Up
```



Algorithm for Boundary Fill (8-Connected)

mathematica

```
BoundaryFill(x, y, fillColor, boundaryColor):
1. If the pixel at (x, y) is boundaryColor or fillColor, return.
2. Set the pixel at (x, y) to fillColor.
3. Recursively call BoundaryFill for the eight neighbors:
   a. BoundaryFill(x+1, y, fillColor, boundaryColor)  // Right
   b. BoundaryFill(x-1, y, fillColor, boundaryColor)  // Left
   c. BoundaryFill(x, y+1, fillColor, boundaryColor)  // Down
   d. BoundaryFill(x, y-1, fillColor, boundaryColor)  // Up
   e. BoundaryFill(x+1, y+1, fillColor, boundaryColor)  // Bottom-Right
   f. BoundaryFill(x-1, y+1, fillColor, boundaryColor)  // Bottom-Left
   g. BoundaryFill(x+1, y-1, fillColor, boundaryColor)  // Top-Right
   h. BoundaryFill(x-1, y-1, fillColor, boundaryColor)  // Top-Left
```

Advantages:

- Simple and easy to implement.
- Works well for filling enclosed regions.

Disadvantages:

- Uses recursion, which can cause stack overflow.
- Can be slow for large areas.

5	**Explain Flood Fill**	**5**

Answer 5 :

The flood fill technique is used to fill an area of connected pixels bounded by different colors. It is called "flood fill" because it behaves like water flooding over a surface, filling all the connected areas until it reaches a boundary.

Key Features of Flood Fill Algorithm

Let us see some of the components of flood-fill algorithm −

Seed Point − This is the starting point inside the polygon where the filling process begins, it can be any point inside the polygon.
Boundary Condition − The algorithm stops when it reaches the edges or boundaries of the polygon.
Recursion or Stack-Based − The algorithm can be implemented recursively or using an explicit stack to avoid deep recursion issues.

Types of Flood Fill Algorithms

There are two types of filling depending on the specific requirement −

1. 4-Connected Flood Fill

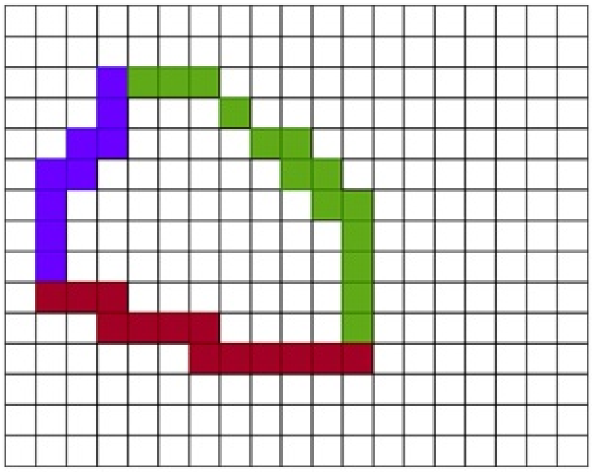
In this approach, each pixel has four neighbours: right, left, above, and below. The algorithm checks these four neighbouring pixels to decide whether to fill them.

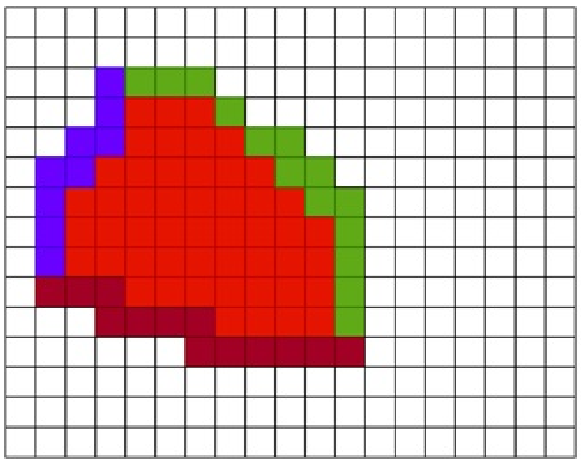
```
FloodFill(x, y, targetColor, fillColor)
   If the pixel at (x, y) is not targetColor or is already fillColor
      Return
   Set the pixel at (x, y)  to fillColor
   FloodFill(x+1, y, targetColor, fillColor)  // Right
   FloodFill(x-1, y, targetColor, fillColor)  // Left
   FloodFill(x, y+1, targetColor, fillColor)  // Down
   FloodFill(x, y-1, targetColor, fillColor)  // Up
```

2. 8-Connected Flood Fill

This approach is more comprehensive, allowing each pixel to have eight neighbours. In addition to the four neighbours from the 4-connected method, it also checks the diagonal neighbours (top-right, top-left, bottom-right, and bottom-left).

The flood-fill algorithm operates by selecting a seed point and checking the color of all the neighbouring pixels. If a neighbouring pixel has the same color as the selected one, it gets filled with a new color. This process continues, spreading outward until it hits a boundary.
It generally uses a random color to fill the internal region then after filling, replace the colors with specified color given as input to the algorithm.
Let us see the example of how 4-connected flood filling is working. Initially we have a polygon with blue green and brown boundary colors.





Advantages and Disadvantages of Flood Fill Algorithm

The following table highlights the advantages and disadvantages of using the Flood Fill algorithm −

Advantages	Disadvantages
Simple to implement − The algorithm is easy to code and understand.	**Memory intensive** − Flood-fill requires significant memory, especially when filling large areas or using a recursive approach.
Efficient for irregular regions − It can fill irregular shapes where the boundaries are not clear-cut.	**Slow performance** − The algorithm can be slower compared to other filling algorithms, especially if the region is large.

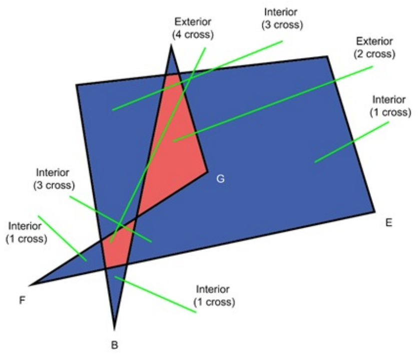
6	Explain inside-outside test with example.	4

Answer 6 :

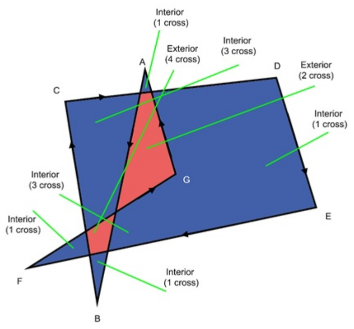
The Inside-Outside Test is a method used in computational geometry to determine whether a given point lies inside, outside, or on the boundary of a polygon.

Methods:

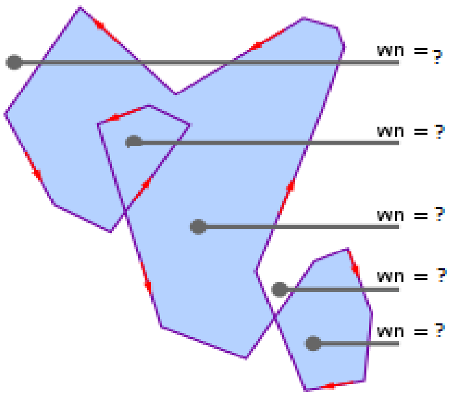
1. Ray-Casting Algorithm (Even-Odd Rule):
    - A horizontal ray is extended from the point in question.
    - Count how many times the ray intersects the polygon's edges.
    - If the count is odd, the point is inside; if even, it is outside.



2. Winding Number Algorithm:
    - Measures how many times the polygon winds around the point.
    - If the winding number is zero, the point is outside; otherwise, it is inside.



Non-zero Winding Number − The point lies inside the polygon. This means the winding number can be any positive or negative number, but not zero.
Zero Winding Number − The point lies outside the polygon.

7	Calculate the Winding number for a given point with respect to the  polygon..	

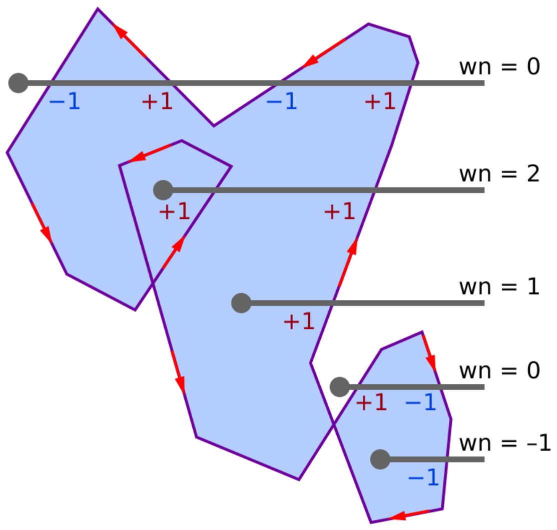
Answer 7:

- The Winding Number indicates how many times a polygon wraps around a given point.

- It is calculated by summing up the angles formed by the edges of the polygon at the given point.

- Formula:

W=∑(angle between consecutive edges
If W≠0, the point is inside.



8	Draw line AB with coordinates A(2,2) and B(6,6). Digital Differential Analyzer)	3

ANSWER 8:

The DDA algorithm calculates intermediate points along a line:

- Calculate dx = 6 - 2 = 4, dy = 6 - 2 = 4.
- Steps = max(dx, dy) = 4.
- x_increment = dx / steps = 1, y_increment = dy / steps = 1.
- Points: (2, 2), (3, 3), (4, 4), (5, 5), (6, 6).

- *Image description:* A line from (2,2) to (6,6) with points marked.

9	Draw line AB with coordinates A(2,2) and B(6,6). (Bresenham’s Line Algorithm)	4

Answer 9:

Bresenham's algorithm is an efficient method to draw lines using only integer arithmetic. It avoids floating-point calculations, making it faster than algorithms like the DDA algorithm.

Steps:

1. Calculate dx and dy:
    - dx = x2 - x1 = 6 - 2 = 4
    - dy = y2 - y1 = 6 - 2 = 4
2. Initialize:
    - x = x1 = 2
    - y = y1 = 2
3. Determine the decision parameter:
    - p0 = 2dy - dx = 2 * 4 - 4 = 4
4. Iterate and calculate points:
    - Since dx > dy, we increment x by 1 in each step and decide whether to increment y or not.
    - For each x, do the following:
        - Plot the point (x, y).
        - If p_k < 0, then the next point is (x_k + 1, y_k), and p_(k+1) = p_k + 2dy.
        - Otherwise (p_k >= 0), the next point is (x_k + 1, y_k + 1), and p_(k+1) = p_k + 2dy - 2dx.

k	p_k	(x, y)
0	4	(2, 2)
1	4 + 2*4 - 2*4 = 4	(3, 3)
2	4 + 2*4 - 2*4 = 4	(4, 4)
3	4 + 2*4 - 2*4 = 4	(5, 5)
4	4 + 2*4 - 2*4 = 4	(6, 6)

10	Draw circle with radius r=10, and center of circle is (1,1) (Only one octant x=0 to x=y) (Midpoint Circle Algorithm).	6

Answer 10

Midpoint Circle Algorithm

The Midpoint Circle Algorithm is an efficient way to generate a circle by making decisions about which pixel to plot based on a decision parameter. It exploits the circle's symmetry to calculate only a fraction of the points.

Steps:

1. **Initialization:**
   - Start with the first point in the first octant: (x, y) = (0, r) = (0, 10).
   - Initialize the decision parameter: p0 = 1 - r = 1 - 10 = -9.
2. **Iterative Calculation:**
   - Iterate while x ≤ y.
   - For each step:
     - If p_k < 0, the next point is (x_k + 1, y_k), and p_(k+1) = p_k + 2x_k + 3.
     - Otherwise (p_k ≥ 0), the next point is (x_k + 1, y_k - 1), and p_(k+1) = p_k + 2x_k - 2y_k + 5.
     - Plot the point (x, y).
     - Plot the other seven symmetric points in the other octants.
     - Increment x.
3. **Translation:**
   - Translate the generated points to the circle's center (1, 1) by adding 1 to the x-coordinates and 1 to the y-coordinates.

k	(x_k, y_k)	p_k	Next Point (x, y)
0	(0, 10)	-9	(1, 10)
1	(1, 10)	-6	(2, 10)
2	(2, 10)	-1	(3, 10)
3	(3, 10)	6	(4, 9)
4	(4, 9)	-3	(5, 9)
5	(5, 9)	8	(6, 8)
6	(6, 8)	3	(7, 7)
7	(7, 7)	-5	(8, 7)

11	Calculate intermediate pixel position (For first quadrant) for ellipse with rx=8, ry=6 and ellipse centre is at origin.	6

Answer:11

1. Region 1 Initialization:
   - Start with (x, y) = (0, ry) = (0, 6).
   - Initialize the decision parameter for Region 1: p1_0 = ry^2 - rx^2 * ry + (1/4) * rx^2 = 6^2 - 8^2 * 6 + (1/4) * 8^2 = 36 - 384 + 16 = -332

2. Iterative Calculation for Region 1:
    - Iterate until 2ry^2 * x >= 2rx^2 * y
    - For each step:
        - If p1_k < 0, the next point is (x_k + 1, y_k), and p1_(k+1) = p1_k + 2ry^2 * x_k + ry^2
        - Otherwise (p1_k >= 0), the next point is (x_k + 1, y_k - 1), and p1_(k+1) = p1_k + 2ry^2 * x_k - 2rx^2 * y_k + ry^2
        - Plot the point (x, y)
3. Region 2 Initialization:
    - The initial point for Region 2 is the last point calculated in Region 1.
    - Calculate the decision parameter for Region 2: p2_0 (using the last point from Region 1).
4. Iterative Calculation for Region 2:
    - Iterate until y = 0
    - For each step:
        - If p2_k > 0, the next point is (x_k, y_k - 1), and p2_(k+1) = p2_k - 2rx^2 * y_k + rx^2
        - Otherwise (p2_k <= 0), the next point is (x_k + 1, y_k - 1), and p2_(k+1) = p2_k + 2ry^2 * x_k - 2rx^2 * y_k + rx^2
        - Plot the point (x, y)

Calculations for Region 1:

k	(x_k, y_k)	p1_k	Next Point (x, y)
0	(0, 6)	-332	(1, 6)
1	(1, 6)	-260	(2, 6)
2	(2, 6)	-152	(3, 6)
3	(3, 6)	-8	(4, 6)
4	(4, 6)	160	(5, 5)
5	(5, 5)	-104	(6, 5)
6	(6, 5)	32	(7, 4)
7	(7, 4)	-76	(8, 4)

12	DDA ALGORITHM a. Calculate the points between the starting point (5, 6) and ending point (8, 12). b. Calculate the points between the starting point (5, 6) and ending point (13, 10). c. Calculate the points between the starting point (1, 7) and ending point (11, 17).	6

Answer 12:

a. Calculate the points between the starting point (5, 6) and ending point (8, 12).

Calculate dx and dy:

- dx = 8 - 5 = 3
- dy = 12 - 6 = 6

Determine the number of steps:

- steps = max(abs(3), abs(6)) = 6

Calculate increments:

- x_increment = 3 / 6 = 0.5
- y_increment = 6 / 6 = 1

Iterate and calculate points:

Step	x	y	round(x),round(y)
1	5	6	(5, 6)
2	5.5	7	(6, 7)
3	6	8	(6, 8)
4	6.5	9	(7,9)
5	7	10	(7,10)
6	7.5	11	(8,11)
7	8	12	(8, 12)

Points: (5, 6), (6, 7), (6, 8), (7, 9), (7, 10), (8, 11), (8, 12)

b. Calculate the points between the starting point (5, 6) and ending point

(13, 10).

Calculate dx and dy:

- dx = 13 - 5 = 8
- dy = 10 - 6 = 4

Determine the number of steps:

- steps = max(abs(8), abs(4)) = 8

Calculate increments:

x_increment = 8 / 8 = 1

y_increment = 4 / 8 = 0.5

Step	x	y	round(x),round(y)
1	5	6	(5, 6)
2	6	6.5	(6, 7)
3	7	7	(7,7)
4	8	7.5	(8,8)
5	9	8	(9,8)
6	10	8.5	(10,9)
7	11	9	(11,9)
8	12	9.5	(12,10)
9	13	10	(13,10)

Points: (5, 6), (6, 7), (7, 7), (8, 8), (9, 8), (10, 9), (11, 9), (12, 10), (13, 10)

C. Calculate the points between the starting point (1, 7) and ending point (11, 17).

Calculate dx and dy:

- dx = 11 - 1 = 10
- dy = 17 - 7 = 10

Determine the number of steps:

- steps = max(abs(10), abs(10)) = 10

Calculate increments:

- x_increment = 10 / 10 = 1
- y_increment = 10 / 10 = 1

Iterate and calculate points:

Step	x	y	round(x),round(y)
1	1	7	(1,7)
2	2	8	(2,8)
3	3	9	(3,9)
4	4	10	(4,10)
5	5	11	(5,11)
6	6	12	(6,12)
7	7	13	(7,13)
8	8	14	(8,14)
9	9	15	(9,15)
10	10	16	(10,16)
11	11	17	(11,17)

Points: (1, 7), (2, 8), (3, 9), (4, 10), (5, 11), (6, 12), (7, 13), (8, 14), (9, 15), (10, 16), (11, 17)

13	Bresenham Line Drawing Algorithm a. Calculate the points between the starting coordinates (9, 18) and ending coordinates (14, 22). b. Calculate the points between the starting coordinates (20, 10) and ending coordinates (30, 18).	6

Answer :13

a. Calculate the points between the starting coordinates (9, 18) and ending coordinates (14, 22).

1.Calculate Differences:

- dx = |14 - 9| = 5
- dy = |22 - 18| = 4

2. Determine Slope Direction:

- x2 > x1 (incrementing x)
- y2 > y1 (incrementing y)

3. Decide Major Axis:

- dx > dy (5 > 4), so the x-axis is the major axis.

4. Initialize Decision Parameter:

- p0 = 2 * dy - dx = 2 * 4 - 5 = 8 - 5 = 3

Step	(x, y)	p	p < 0	Next (x, y)
0	(9, 18)	3	FALSE	(10, 19)
1	(10, 19)	3 + 2*4 - 2*5 = 1	FALSE	(11, 20)
2	(11, 20)	1 + 2*4 - 2*5 = -1	TRUE	(12, 20)
3	(12, 20)	-1 + 2*4 = 7	FALSE	(13, 21)
4	(13, 21)	7 + 2*4 - 2*5 = 5	FALSE	(14, 22)

Points:**(9, 18), (10, 19), (11, 20), (12, 20), (13, 21), (14, 22)**

b. Calculating points between (20, 10) and (30, 18)

- **Start (x1, y1):** (20, 10)
- **End (x2, y2):** (30, 18)

1. **Calculate Differences:**
    - dx = |30 - 20| = 10
    - dy = |18 - 10| = 8
2. **Determine Slope Direction:**
    - x2 > x1 (incrementing x)
    - y2 > y1 (incrementing y)
3. **Decide Major Axis:**
    - dx > dy (10 > 8), so the x-axis is the major axis.
4. **Initialize Decision Parameter:**
    - p0 = 2 * dy - dx = 2 * 8 - 10 = 16 - 10 = 6
5. **Iterate and Plot:**

Step	(x, y)	p	Condition (p < 0?)	Next (x, y)
0	(20, 10)	6	FALSE	(21, 11)
1	(21, 11)	6 + 2*8 - 2*10 = 2	FALSE	(22, 12)
2	(22, 12)	2 + 2*8 - 2*10 = -2	TRUE	(23, 12)
3	(23, 12)	-2 + 2*8 = 14	FALSE	(24, 13)
4	(24, 13)	14 + 2*8 - 2*10 = 10	FALSE	(25, 14)
5	(25, 14)	10 + 2*8 - 2*10 = 6	FALSE	(26, 15)
6	(26, 15)	6 + 2*8 - 2*10 = 2	FALSE	(27, 16)
7	(27, 16)	2 + 2*8 - 2*10 = -2	TRUE	(28, 16)
8	(28, 16)	-2 + 2*8 = 14	FALSE	(29, 17)
9	(29, 17)	14 + 2*8 - 2*10 = 10	FALSE	(30, 18)

14	Mid Point Circle Drawing Algorithm a. Given the center point coordinates (0, 0) and radius as 10, generate all the points to form a circle. b. Given the center point coordinates (4, -4) and radius as 10, generate all the points to form a circle	6

Answer :14

a. Given the center point coordinates (0, 0) and radius as 10, generate all the points to form a circle.

Iteration	(xk, yk)	Decision Parameter (pk)	Condition (pk < 0?)	Next Pixel (xk+1, yk+1)	Update Decision Parameter (pk+1)
Start	(0, 10)	p0 = 1 - 10 = -9	Yes	(1, 10)	p1 = -9 + 2*0 + 3 = -6
1	(1, 10)	p1 = -6	Yes	(2, 10)	p2 = -6 + 2*1 + 3 = -1
2	(2, 10)	p2 = -1	Yes	(3, 10)	p3 = -1 + 2*2 + 3 = 6
3	(3, 10)	p3 = 6	No	(4, 9)	p4 = 6 + 2*3 - 2*10 + 5 = -3
4	(4, 9)	p4 = -3	Yes	(5, 9)	p5 = -3 + 2*4 + 3 = 8
5	(5, 9)	p5 = 8	No	(6, 8)	p6 = 8 + 2*5 - 2*9 + 5 = -5
6	(6, 8)	p6 = -5	Yes	(7, 8)	p7 = -5 + 2*6 + 3 = 10
7	(7, 8)	p7 = 10	No	(8, 7)	p8 = 10 + 2*7 - 2*8 + 5 = 3
8	(8, 7)	p8 = 3	No	(9, 6)	p9 = 3 + 2*8 - 2*7 + 5 = 7
9	(9, 6)	p9 = 7	No	(10, 5)	p10 = 7 + 2*9 - 2*6 + 5 = 12
10	(10, 5)	p10 = 12	No	(11, 4)	p11 = 12 + 2*10 - 2*5 + 5 = 27

b. Given the center point coordinates (4, -4) and radius as 10, generate all the points to form a circle

Iteration	(xk, yk)	Decision Parameter (pk)	Condition (pk < 0?)	Next Pixel (xk+1, yk+1)	Update Decision Parameter (pk+1)
Start	(0, 10)	p0 = 1 - 10 = -9	Yes	(1, 10)	p1 = -9 + 2*0 + 3 = -6
1	(1, 10)	p1 = -6	Yes	(2, 10)	p2 = -6 + 2*1 + 3 = -1
2	(2, 10)	p2 = -1	Yes	(3, 10)	p3 = -1 + 2*2 + 3 = 6
3	(3, 10)	p3 = 6	No	(4, 9)	p4 = 6 + 2*3 - 2*10 + 5 = -3
4	(4, 9)	p4 = -3	Yes	(5, 9)	p5 = -3 + 2*4 + 3 = 8
5	(5, 9)	p5 = 8	No	(6, 8)	p6 = 8 + 2*5 - 2*9 + 5 = -5
6	(6, 8)	p6 = -5	Yes	(7, 8)	p7 = -5 + 2*6 + 3 = 10
7	(7, 8)	p7 = 10	No	(8, 7)	p8 = 10 + 2*7 - 2*8 + 5 = 3

15	Bresenham Circle Drawing Algorithm a. Given the center point coordinates (0, 0) and radius as 8, generate all the points to form a circle. b. Given the center point coordinates (10, 10) and radius as 10, generate all the points to form a circle.	6

Answer:15

Center (0, 0), Radius = 8

Initialization:

- Center (xc, yc) = (0, 0)
- Radius (r) = 8
- Initial point: (x, y) = (0, 8)
- Initial decision parameter: `p = 3 - 2 * r = 3 - 2 * 8 = 3 - 16 = -13`

Iteration	(xk, yk)	Decision Parameter (pk)	Condition (pk < 0?)	Next Pixel (xk+1, yk+1)	Update Decision Parameter (pk+1)	8 Symmetric Points (±x, ±y), (±y, ±x)
Start	(0, 8)	-13	Yes	(1, 8)	-13 + 4*0 + 6 = -7	(0, 8), (0, -8), (8, 0), (-8, 0)
1	(1, 8)	-7	Yes	(2, 8)	-7 + 4*1 + 6 = 3	(1, 8), (-1, 8), (1, -8), (-1, -8), (8, 1), (-8, 1), (8, -1), (-8, -1)
2	(2, 8)	3	No	(3, 7)	3 + 4*(2 - 8) + 10 = -11	(2, 8), (-2, 8), (2, -8), (-2, -8), (8, 2), (-8, 2), (8, -2), (-8, -2)
3	(3, 7)	-11	Yes	(4, 7)	-11 + 4*3 + 6 = 7	(3, 7), (-3, 7), (3, -7), (-3, -7), (7, 3), (-7, 3), (7, -3), (-7, -3)
4	(4, 7)	7	No	(5, 6)	7 + 4*(4 - 7) + 10 = 5	(4, 7), (-4, 7), (4, -7), (-4, -7), (7, 4), (-7, 4), (7, -4), (-7, -4)
5	(5, 6)	5	No	(6, 5)	5 + 4*(5 - 6) + 10 = 11	(5, 6), (-5, 6), (5, -6), (-5, -6), (6, 5), (-6, 5), (6, -5), (-6, -5)
6	(6, 5)	11	No	(7, 4)	11 + 4*(6 - 5) + 10 = 25	(6, 5), (-6, 5), (6, -5), (-6, -5), (5, 6), (-5, 6), (5, -6), (-5, -6)
7	(7, 4)	25	No	(8, 3)	25 + 4*(7 - 4) + 10 = 37	(7, 4), (-7, 4), (7, -4), (-7, -4), (4, 7), (-4, 7), (4, -7), (-4, -7)

Unit-2 2D and 3D Transformation

1	Explain 2D & 3D Translation.	7

Answer 1:

2D Translation

Definition:

2D Translation is a rigid-body transformation that moves every point in a 2D object or scene by a fixed distance in a given direction. It shifts the object without changing its shape, size, or orientation.

Mathematical Representation:

A 2D point P(x, y) can be translated to a new position P'(x', y') by adding a translation vector T(tx, ty) to its coordinates:

- **x' = x + tx**
- **y' = y + ty**

Where:

- (x, y) are the original coordinates of the point.
- (x', y') are the new coordinates of the translated point.
- (tx, ty) is the translation vector, where:
  - tx specifies the horizontal translation distance.
  - ty specifies the vertical translation distance.

Matrix Representation (Homogeneous Coordinates):

In computer graphics, transformations are often represented using matrices, especially when dealing with a sequence of transformations. To represent 2D translation as a matrix multiplication, we use homogeneous coordinates. A 2D point (x, y) is represented as a 3D vector (x, y, 1). The 2D translation matrix is a 3x3 matrix:

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

The translation operation in matrix form is:

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

Key Characteristics of 2D Translation:

- **Rigid Body Transformation:** The shape and size of the object remain unchanged.
- **Preserves Orientation:** The orientation of the object does not change.

- **Vector Addition:** Effectively, the translation vector is added to the position vector of each point in the object.
- **Inverse Translation:** To move the object back to its original position, a translation with the vector (-tx, -ty) is applied.

Applications of 2D Translation:

- Moving objects across the screen in games and animations.
- Positioning elements in user interfaces (UI).
- Implementing scrolling functionality.
- As a component of more complex transformations (e.g., rotation around an arbitrary point).

3D Translation

Definition:

3D Translation is a rigid-body transformation that moves every point in a 3D object or scene by a fixed distance along each of the three coordinate axes (x, y, and z). Similar to 2D translation, it shifts the object without altering its shape, size, or orientation.

Mathematical Representation:

A 3D point P(x, y, z) can be translated to a new position P'(x', y', z') by adding a 3D translation vector T(tx, ty, tz) to its coordinates:

- **x' = x + tx**
- **y' = y + ty**
- **z' = z + tz**

Where:

- (x, y, z) are the original coordinates of the point.
- (x', y', z') are the new coordinates of the translated point.
- (tx, ty, tz) is the translation vector, where:
    - tx specifies the translation distance along the x-axis.
    - ty specifies the translation distance along the y-axis.
    - tz specifies the translation distance along the z-axis.

Matrix Representation (Homogeneous Coordinates):

In 3D graphics, homogeneous coordinates represent a 3D point (x, y, z) as a 4D vector (x, y, z, 1). The 3D translation matrix is a 4x4 matrix:

$$\begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The translation operation in matrix form is:

[x'] [1 0 0 tx] [x]
[y'] = [0 1 0 ty] [y]
[z'] = [0 0 1 tz] [z]
[1] [0 0 0   1] [1]

Key Characteristics of 3D Translation:

- **Rigid Body Transformation:** The shape and size of the 3D object remain unchanged.
- **Preserves Orientation:** The orientation of the 3D object does not change.
- **Vector Addition:** The 3D translation vector is added to the position vector of each vertex of the 3D object.
- **Inverse Translation:** To move the object back to its original position, a translation with the vector (-tx, -ty, -tz) is applied.

Applications of 3D Translation:

- Moving objects within a 3D virtual environment (e.g., in games, simulations, VR/AR).
- Positioning models and components in 3D modeling software.
- Implementing camera movements in 3D scenes.
- As a fundamental step in complex 3D transformations (e.g., rotation around an arbitrary axis, transformations in articulated models).

2	Explain 2D & 3D Rotation.	7

Answer 2 :

2D Rotation Definition:

2D Rotation is a rigid-body transformation that turns every point in a 2D object or scene around a fixed point, called the **center of rotation**, by a specific angle. It changes the orientation of the object but preserves its shape, size, and the relative positions of its parts.

Mathematical Representation (Rotation about the Origin):

Consider a point P(x, y) that needs to be rotated by an angle θ (theta) counter-clockwise around the origin (0, 0) to a new position P'(x', y'). The rotation equations are:

- **$x' = x * \cos(\theta) - y * \sin(\theta)$**
- **$y' = x * \sin(\theta) + y * \cos(\theta)$**

Where:

- (x, y) are the original coordinates of the point.
- (x', y') are the new coordinates of the rotated point.
- θ is the angle of rotation (in radians or degrees, but trigonometric functions

usually expect radians).

Matrix Representation (Homogeneous Coordinates, Rotation about the Origin):

Using homogeneous coordinates (x, y, 1), the 2D rotation matrix about the origin is:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation operation in matrix form is:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation about an Arbitrary Point (cx, cy):

To rotate a point around an arbitrary center of rotation (cx, cy), we typically perform the following steps:

1. **Translate:** Translate the object so that the center of rotation (cx, cy) coincides with the origin (by applying a translation of (-cx, -cy)).
2. **Rotate:** Rotate the translated object around the origin by the desired angle θ.
3. **Translate Back:** Translate the object back so that the origin coincides with the original center of rotation (by applying a translation of (cx, cy)).

The combined transformation matrix for rotation about an arbitrary point is obtained by multiplying the individual transformation matrices in the correct order:

$T(cx, cy) * R(\theta) * T(-cx, -cy)$

Where:

- T(tx, ty) is the translation matrix.
- R(θ) is the rotation matrix about the origin.

Key Characteristics of 2D Rotation:

- **Rigid Body Transformation:** Preserves shape and size.
- **Changes Orientation:** Alters the angular position of the object.
- **Defined by Angle and Center:** Requires a rotation angle and a center of rotation.
- **Inverse Rotation:** Rotating by -θ (or 360° - θ) reverses the rotation.

Applications of 2D Rotation:

- Rotating objects in animations and games (e.g., spinning wheels, turning characters).
- Creating circular or radial designs.

- Adjusting the orientation of UI elements.
- As part of complex transformations like orbiting motions.

3D Rotation Definition:

3D Rotation is a rigid-body transformation that turns every point in a 3D object or scene around a fixed line, called the **axis of rotation**, by a specific angle. Similar to 2D rotation, it changes the orientation of the object while preserving its shape, size, and the relative positions of its parts.

Mathematical Representation (Rotation about Coordinate Axes):

Rotations in 3D are often defined around one of the three principal coordinate axes (x, y, z).

Rotation about the X-axis (Rx(θ)):

$x' = x$
$y' = y * \cos(\theta) - z * \sin(\theta)$
$z' = y * \sin(\theta) + z * \cos(\theta)$
Matrix form (homogeneous coordinates):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the Y-axis (Ry(θ)):

$x' = x * \cos(\theta) + z * \sin(\theta)$
$y' = y$
$z' = -x * \sin(\theta) + z * \cos(\theta)$
Matrix form (homogeneous coordinates):

$$\begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the Z-axis (Rz(θ)):
$x' = x * \cos(\theta) - y * \sin(\theta)$
$y' = x * \sin(\theta) + y * \cos(\theta)$
$z' = z$
Matrix form (homogeneous coordinates):

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Key Characteristics of 3D Rotation:

- **Rigid Body Transformation:** Preserves shape and size.
- **Changes Orientation:** Alters the angular position of the object in 3D space.

- **Defined by Angle and Axis:** Requires a rotation angle and an axis of rotation.
- **Order Matters:** The order in which rotations around different axes are applied affects the final orientation (rotations are generally not commutative).
- **Inverse Rotation:** Rotating by -θ reverses the rotation around the same axis.

Applications of 3D Rotation:

- Orienting objects in 3D scenes.
- Creating complex movements and animations (e.g., rotating planets, turning vehicles).
- Implementing camera controls (e.g., panning, tilting, rolling).
- Manipulating 3D models in CAD/CAM software.
- As a crucial component in character animation and robotics.

3	Explain 2D & 3D Scaling.	7

Answer 3:

2D Scaling Definition:

2D Scaling is a transformation that changes the size of a 2D object by multiplying the x and y coordinates of each vertex by specific scaling factors. This can result in the object becoming larger (scaling up) or smaller (scaling down). Scaling can be **uniform** (same scaling factor for both x and y, preserving the aspect ratio) or **non-uniform** (different scaling factors for x and y, potentially distorting the aspect ratio).

Mathematical Representation:

For a point P(x, y), after scaling by factors Sx in the x-direction and Sy in the y-direction, the new coordinates P'(x', y') are:

- **x' = x * Sx**
- **y' = y * Sy**

Where:

- (x, y) are the original coordinates.
- (x', y') are the scaled coordinates.
- Sx is the scaling factor along the x-axis.
- Sy is the scaling factor along the y-axis.

Matrix Representation (Homogeneous Coordinates):

Using homogeneous coordinates (x, y, 1), the 2D scaling matrix is:

$$\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The scaling operation in matrix form is:

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

Types of 2D Scaling:

- **Uniform Scaling:** Sx = Sy. The object's overall size changes, but its proportions remain the same.
- **Non-Uniform Scaling:** Sx ≠ Sy. The object can be stretched or compressed along the x or y axis, changing its aspect ratio.

Scaling about a Fixed Point (fx, fy):

To scale an object about a fixed point (fx, fy) other than the origin, the following steps are performed:

1. **Translate:** Translate the object so that the fixed point (fx, fy) coincides with the origin (by applying a translation of (-fx, -fy)).
2. **Scale:** Scale the translated object by the desired scaling factors Sx and Sy.
3. **Translate Back:** Translate the object back so that the origin coincides with the original fixed point (by applying a translation of (fx, fy)).

The combined transformation matrix is:

T(fx, fy) * S(Sx, Sy) * T(-fx, -fy)

Key Characteristics of 2D Scaling:

- Changes the size of the object.
- Can be uniform (preserving aspect ratio) or non-uniform (distorting aspect ratio).
- Scaling factors greater than 1 enlarge the object.
- Scaling factors less than 1 (and greater than 0) shrink the object.
- Scaling factor of 1 leaves the size unchanged.
- Scaling factors can be negative, resulting in a reflection along with scaling.

Applications of 2D Scaling:

- Zooming in and out of images or scenes.
- Resizing UI elements.
- Creating special effects.
- Adjusting the size of objects to fit within a specific area.

3D Scaling

Definition:

3D Scaling is a transformation that changes the size of a 3D object by multiplying the x, y, and z coordinates of each vertex by specific scaling factors along the respective axes. Similar to 2D scaling, it can be **uniform** (same scaling factor for all three axes) or

non-uniform (different scaling factors for each axis).

Mathematical Representation:

For a point P(x, y, z), after scaling by factors Sx, Sy, and Sz along the x, y, and z axes respectively, the new coordinates P'(x', y', z') are:

- **x' = x * Sx**
- **y' = y * Sy**
- **z' = z * Sz**

Where:

- (x, y, z) are the original coordinates.
- (x', y', z') are the scaled coordinates.
- Sx is the scaling factor along the x-axis.
- Sy is the scaling factor along the y-axis.
- Sz is the scaling factor along the z-axis.

Matrix Representation (Homogeneous Coordinates):

Using homogeneous coordinates (x, y, z, 1), the 3D scaling matrix is:

```
[ Sx  0   0    0 ]
[ 0   Sy  0    0 ]
[ 0    0   Sz  0 ]
[ 0    0   0    1 ]
```

The scaling operation in matrix form is:

```
[ x' ] = [ Sx  0   0   0 ] [ x ]
[ y' ] = [ 0   Sy  0   0 ] [ y ]
[ z' ] = [ 0   0   Sz  0 ] [ z ]
[ 1  ] = [ 0   0   0   1 ] [ 1 ]
```

Types of 3D Scaling:

- **Uniform Scaling:** Sx = Sy = Sz. The object's overall size changes proportionally in all dimensions.
- **Non-Uniform Scaling:** Sx, Sy, and Sz are not all equal. The object can be stretched or compressed along individual axes, leading to changes in its proportions and potentially its shape.

Key Characteristics of 3D Scaling:

- Changes the size of the object in three dimensions.
- Can be uniform (preserving aspect ratio) or non-uniform (distorting aspect ratio).
- Scaling factors greater than 1 enlarge the object along the corresponding axis.
- Scaling factors less than 1 (and greater than 0) shrink the object along the corresponding axis.
- Scaling factor of 1 leaves the size unchanged along that axis.

- Negative scaling factors result in reflection along with scaling.

Applications of 3D Scaling:

- Adjusting the size of 3D models.
- Creating effects of distance and perspective.
- Scaling individual components of a complex model.
- Implementing zoom functionality in 3D viewers.

4	Explain Composite Transformation. (Translation, Rotation & Scaling)	6

Answer 4:

A **composite transformation** occurs when two or more basic transformations (like translation, rotation, and scaling) are applied to an object in sequence. The order in which these transformations are applied is crucial, as matrix multiplication is generally not commutative (i.e., the order affects the final result).

The primary advantage of composite transformations is that a sequence of transformations can be represented by a single **composite transformation matrix**. This matrix is obtained by multiplying the individual transformation matrices together in the order they are applied (from right to left). Applying this single composite matrix to the vertices of an object achieves the same final transformation as applying the individual transformations sequentially, but with greater efficiency.

Mathematical Representation:

If we have a sequence of transformations T1, T2, T3 applied to a point P, the final transformed point P' can be represented as:

P' = T3 * T2 * T1 * P

Where T1, T2, and T3 are the transformation matrices for the individual transformations, and the multiplication is matrix multiplication. The composite transformation matrix **T_composite** is then:

T_composite = T3 * T2 * T1

And the final transformed point is:

P' = **T_composite** * P

Order of Operations:

Remember that when reading the sequence of transformations applied to a point (like T3 * T2 * T1 * P), the transformation closest to the point (T1 in this case) is applied first, then the next one (T2), and so on. When multiplying the matrices to get the composite matrix, the order is reversed.

Composite Transformations in 2D:

In 2D, using homogeneous coordinates (3x3 matrices), we can create composite transformations involving translation, rotation, and scaling.

- **Translation followed by Rotation:** R(θ) * T(tx, ty)

- **Rotation about an arbitrary point (cx, cy):** T(cx, cy) * R(θ) * T(-cx, -cy) (Translate to origin, rotate, translate back)

- **Scaling about an arbitrary point (fx, fy):** T(fx, fy) * S(sx, sy) * T(-fx, -fy) (Translate to origin, scale, translate back)

The resulting 3x3 composite matrix can then be applied to each vertex (represented as a 3x1 homogeneous coordinate vector [x, y, 1]<sup>T</sup>) of the 2D object to achieve the combined transformation.

Composite Transformations in 3D:

Similarly, in 3D, using homogeneous coordinates (4x4 matrices), we can combine translation, rotation (around x, y, or z axes), and scaling.

- **Translation followed by Rotation around Z-axis:** Rz(θ) * T(tx, ty, tz)

- **Rotation about an arbitrary axis:** This is a more complex composite transformation involving translations to move the axis to the origin, rotations to align the axis with a principal axis, the rotation itself, and then the inverse transformations to move everything back.
- **Scaling about an arbitrary point (fx, fy, fz):** T(fx, fy, fz) * S(sx, sy, sz) * T(-fx, -fy, -fz)
- The resulting 4x4 composite matrix is then applied to each vertex (represented as a 4x1 homogeneous coordinate vector [x, y, z, 1]<sup>T</sup>) of the 3D object.

Examples of Composite Transformations:

- **Rotating an object around its center:** This involves translating the object so that its center is at the origin, then rotating it, and finally translating it back to its original center.
- **Scaling an object about a specific corner:** This requires translating the object so that the corner is at the origin, then scaling, and then translating back.
-
- **Moving an object along a circular path:** This can be achieved by repeatedly applying small rotations around a center point and small translations along the tangent of the circle.

Benefits of Using Composite Transformations:

- **Efficiency:** Applying a single composite matrix is generally more efficient than applying a sequence of individual matrices, especially when dealing with a large number of vertices.
- **Organization:** It simplifies the representation of complex transformations as a single entity.
- **Flexibility:** Allows for the creation of intricate transformations by combining basic ones.

5	What is the homogeneous matrix representation? (Translation, Rotation & Scaling)	6

Answer 5:

The homogeneous matrix representation is a powerful technique used in computer graphics and linear algebra to unify different types of geometric transformations, such as translation, rotation, and scaling, into a single matrix format. This allows for efficient composition of multiple transformations through matrix multiplication.

Here's a breakdown of the homogeneous matrix representation for 2D and 3D transformations:

1. Homogeneous Coordinates:

The key idea behind homogeneous matrices is the use of **homogeneous coordinates**.

- **2D Homogeneous Coordinates:** A 2D point (x, y) is represented by a 3D vector (x, y, w), where 'w' is a non-zero weight factor. Usually, 'w' is set to 1, so the point becomes (x, y, 1).
- **3D Homogeneous Coordinates:** A 3D point (x, y, z) is represented by a 4D vector (x, y, z, w), where 'w' is a non-zero weight factor. Typically, 'w' is set to 1, resulting in (x, y, z, 1).

The extra dimension (w) allows us to represent affine transformations (including translation, which is not a linear transformation in standard Cartesian coordinates) as linear transformations in the higher-dimensional homogeneous space.

2. Transformation Matrices:

Each basic transformation (translation, rotation, scaling) can be represented by a specific homogeneous matrix.

a) 2D Homogeneous Transformation Matrices (3x3):

Translation by (tx, ty):

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation by an angle θ (counter-clockwise) about the origin:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scaling by (sx, sy) along the x and y axes:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

b) 3D Homogeneous Transformation Matrices (4x4):

Translation by (tx, ty, tz):

$$\begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation by an angle θ (counter-clockwise) about the X-axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation by an angle θ (counter-clockwise) about the Y-axis:

$$\begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation by an angle θ (counter-clockwise) about the Z-axis:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling by (sx, sy, sz) along the x, y, and z axes:

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. Applying Transformations:

To apply a transformation to a point (in homogeneous coordinates), you multiply the transformation matrix by the point's vector:

- **2D:** P' = T * P, where P' = [x', y', 1]<sup>T</sup>, T is the 3x3 transformation matrix, and P = [x, y, 1]<sup>T</sup>.
- **3D:** P' = T * P, where P' = [x', y', z', 1]<sup>T</sup>, T is the 4x4 transformation matrix, and P = [x, y, z, 1]<sup>T</sup>.

4. Composite Transformations:

The real power of homogeneous matrices lies in their ability to represent sequences of transformations as a single matrix. If you want to apply multiple transformations (e.g., translate, then rotate, then scale), you multiply their respective homogeneous matrices together in the order they are applied (from right to left):

T_composite = S * R * T

Where T is the translation matrix, R is the rotation matrix, and S is the scaling matrix. Applying T_composite to a point will yield the same result as applying the individual transformations sequentially.

Why Use Homogeneous Matrices?

- **Unified Representation:** All common affine transformations (translation, rotation, scaling, shear, reflection) can be represented using matrix multiplication.
- **Composition of Transformations:** Multiple transformations can be easily combined into a single matrix by matrix multiplication. This is crucial for complex object manipulations.
- **Efficiency:** Hardware and software in computer graphics are often optimized for matrix operations, making homogeneous transformations efficient to process.
- **Perspective Projections (3D):** In 3D graphics, homogeneous coordinates are essential for representing perspective projections, which make distant objects appear smaller. This involves manipulating the 'w' component of the homogeneous coordinates

6	Explain Reflection.	5

Answer 6:

Reflection in computer graphics is a transformation that produces a **mirror image** of an object relative to a **line of reflection** (in 2D) or a **plane of reflection** (in 3D). It essentially "flips" the object across this axis or plane.

Key Characteristics:

1. **Mirror Image:** The reflected object is a mirror replica of the original.
2. **Equal Distance:** Every point on the reflected object is the same perpendicular distance from the line/plane of reflection as its corresponding point on the original object, but on the opposite side.
3. **Reversal of Orientation:** The orientation (e.g., clockwise or counter-clockwise order of vertices) of the reflected object is reversed.
4. **Rigid Body Transformation:** While the orientation changes, the shape and size of the object remain the same.
5. **Inverse is Itself:** Applying the same reflection transformation twice returns the object to its original position.

Matrix Representation (Homogeneous Coordinates - Examples):

2D Reflection across the Y-axis:
[-1 0 0]
[0 1 0]
[0 0 1]
2D Reflection across the X-axis:
[1 0 0]
[0 -1 0]
[0 0 1]
3D Reflection across the XY-plane (z=0):
[1 0 0 0]
[0 1 0 0]

```
[ 0  0 -1  0]
[ 0  0  0  1]
```

Reflections about arbitrary lines or planes can be achieved by combining translation, rotation, and the basic axis/plane reflection matrices

7	What is Shear?	4

Answer 7:

Shear is a geometric transformation that distorts the shape of an object by shifting its points parallel to a fixed line (in 2D) or a fixed plane (in 3D), with the amount of shift proportional to their perpendicular distance from that line or plane.

Effect:

- **Changes the shape:** Squares can become parallelograms, and circles can become ellipses.
- **Preserves parallelism:** Lines that are parallel before the shear remain parallel after.
- **Alters angles:** The angles between lines within the object are generally changed.
- **Preserves area (in 2D) and volume (in 3D):** The overall area or volume of the object remains the same.
- It's a **non-rigid** transformation because the shape is altered.

Types of Shear:

- **2D Shear:**
  - **Horizontal Shear (Shear along the x-axis):** Shifts points horizontally by an amount proportional to their y-coordinate. Points with the same y-coordinate are shifted by the same amount. The x-coordinate changes, while the y-coordinate remains the same.
  - **Vertical Shear (Shear along the y-axis):** Shifts points vertically by an amount proportional to their x-coordinate. Points with the same x-coordinate are shifted by the same amount. The y-coordinate changes, while the x-coordinate remains the same.
  -
  - **X-Y Shear:** Both x and y coordinates are modified, resulting in a more complex distortion.
- **3D Shear:** Shear can occur along any of the three axes (x, y, or z), affecting the other two coordinates based on the distance from the fixed plane. For example:
  - **Shear in the x-direction:** The x-coordinate remains unchanged, while the y and z coordinates are altered based on the x-coordinate.
  - **Shear in the y-direction:** The y-coordinate remains unchanged, while the x and z coordinates are altered based on the y-coordinate.
  - **Shear in the z-direction:** The z-coordinate remains unchanged, while the x and y coordinates are altered based on the z-coordinate.

8	Translate the triangle [A(10,10), B(15,15), C(20,10)] 2 units in x direction and 1 unit in y direction.	4

Answer 8:

translate the triangle with vertices A(10, 10), B(15, 15), and C(20, 10) by 2 units in the x-direction and 1 unit in the y-direction.

Translation Vector: The translation vector is T(tx, ty) = T(2, 1).

To translate each vertex, we add the corresponding components of the translation vector to the vertex's coordinates:

Vertex A(10, 10): A'(x', y') = (Ax + tx, Ay + ty) A'(x', y') = (10 + 2, 10 + 1) **A' = (12, 11)**

Vertex B(15, 15): B'(x', y') = (Bx + tx, By + ty) B'(x', y') = (15 + 2, 15 + 1) **B' = (17, 16)**

Vertex C(20, 10): C'(x', y') = (Cx + tx, Cy + ty) C'(x', y') = (20 + 2, 10 + 1) **C' = (22, 11)**

Therefore, the translated triangle has the following vertices:

- **A' (12, 11)**
- **B' (17, 16)**
- **C' (22, 11)**

9	Locate the new position of the triangle [A(5,4), B(8,3), C(8,8)] after its rotation by 90 degree clockwise about the origin.	4

Answer 9:

To rotate a point (x, y) by 90 degrees clockwise about the origin, the new coordinates (x', y') are given by the rule:

x' = y y' = -x

Let's apply this rule to each vertex of the triangle:

Vertex A(5, 4): A'(x', y') = (4, -5) **A' = (4, -5)**

Vertex B(8, 3): B'(x', y') = (3, -8) **B' = (3, -8)**

Vertex C(8, 8): C'(x', y') = (8, -8) **C' = (8, -8)**

Therefore, the new position of the triangle after a 90-degree clockwise rotation about the origin has the following vertices:

- **A' (4, -5)**
- **B' (3, -8)**
- **C' (8, -8)**

10	Obtain the final coordinates after two rotations on point p(6,9) with rotation angles 30 degree and 60 degree respectively.	5

Answer 10:

Single Rotation by the Sum of Angles

The total rotation angle is 30 degrees + 60 degrees = 90 degrees. We can rotate the original point P(6, 9) by 90 degrees counter-clockwise.

The rotation formulas for a counter-clockwise rotation by 90 degrees are: $x' = x * \cos(90°) - y * \sin(90°)$ $y' = x * \sin(90°) + y * \cos(90°)$

$\cos(90°) = 0$ $\sin(90°) = 1$

$x'' = 6 * 0 - 9 * 1 = -9$ $y'' = 6 * 1 + 9 * 0 = 6$

So, after a single 90-degree counter-clockwise rotation, the point P becomes (-9, 6).

Comparison and Conclusion

There seems to be a discrepancy between the two methods due to rounding in the first method. Let's perform the first method with exact values:

- **Rotation 1: 30 degrees** $x' = 6 * (\sqrt{3} / 2) - 9 * (1 / 2) = 3\sqrt{3} - 4.5$ $y' = 6 * (1 / 2) + 9 * (\sqrt{3} / 2) = 3 + 4.5\sqrt{3}$
- **Rotation 2: 60 degrees** $x'' = (3\sqrt{3} - 4.5) * (1 / 2) - (3 + 4.5\sqrt{3}) * (\sqrt{3} / 2) = (3\sqrt{3} / 2) - 2.25 - (3\sqrt{3} / 2) - (4.5 * 3 / 2) = (3\sqrt{3} / 2) - 2.25 - (3\sqrt{3} / 2) - 6.75 = -9$

  $y'' = (3\sqrt{3} - 4.5) * (\sqrt{3} / 2) + (3 + 4.5\sqrt{3}) * (1 / 2) = (3 * 3 / 2) - (4.5\sqrt{3} / 2) + 1.5 + (4.5\sqrt{3} / 2) = 4.5 - (4.5\sqrt{3} / 2) + 1.5 + (4.5\sqrt{3} / 2) = 6$

As you can see, when using exact values, both methods yield the same final coordinates.

Final Coordinates:

The final coordinates of the point P(6, 9) after two rotations of 30 degrees and 60 degrees respectively are **(-9, 6)**.

11	Obtain the final coordinates after two scaling on line pq [p(2,2), q(8,8)] with scaling factors are (2,2) and (3,3) respectively.	5

Answer 11:

Method 1: Sequential Scaling

- **Scaling 1: (2, 2)** The scaling formulas are: $x' = x * sx1$ $y' = y * sy1$

◦ For point P(2, 2): P'(x', y') = (2 * 2, 2 * 2) = (4, 4) ◦ For point Q(8, 8): Q'(x', y') = (8 * 2, 8 * 2) = (16, 16) ● After the first scaling, the line segment becomes P'(4, 4) to Q'(16, 16). ● **Scaling 2: (3, 3)** Now, we scale the points P'(4, 4) and Q'(16, 16) using the scaling factors (3, 3). ◦ For point P'(4, 4): P''(x'', y'') = (4 * 3, 4 * 3) = (12, 12) ◦ For point Q'(16, 16): Q''(x'', y'') = (16 * 3, 16 * 3) = (48, 48) ● So, after the second scaling, the line segment becomes P''(12, 12) to Q''(48, 48).		
12	Find the coordinates after reflection of the triangle [A(10,10), B(15,15), C(20,10)] about x axis.	4

Answer 12:

To reflect a point (x, y) about the x-axis, the x-coordinate remains the same, and the y-coordinate changes its sign. The transformation rule is:

(x, y) becomes (x, -y)

Let's apply this rule to each vertex of the triangle:

Vertex A(10, 10): A'(x', y') = (10, -10) **A' = (10, -10)**

Vertex B(15, 15): B'(x', y') = (15, -15) **B' = (15, -15)**

Vertex C(20, 10): C'(x', y') = (20, -10) **C' = (20, -10)**

Therefore, the coordinates of the triangle after reflection about the x-axis are:

- **A' (10, -10)**
- **B' (15, -15)**
- **C' (20, -10)**

13	Shear the unit square in x direction with shear parameter ½ relative to line y=(-1).	5

Answer 13:

To shear the unit square in the x-direction with a shear parameter of ½ relative to the line y = -1, we need to follow these steps:

1. Define the Vertices of the Unit Square: The unit square has vertices at: A(0, 0) B(1, 0) C(1, 1) D(0, 1)

2. Understand the Shear Transformation Relative to a Line: A shear in the x-direction relative to a line y = y_ref is given by the following transformation equations: x' = x + shx * (y - y_ref) y' = y

where:

- (x, y) are the original coordinates.
- (x', y') are the new coordinates after shear.
- `shx` is the shear parameter (given as ½).
- `y_ref` is the y-coordinate of the reference line (given as -1).

3. Apply the Shear Transformation to Each Vertex:

- **Vertex A(0, 0):** x' = 0 + (1/2) * (0 - (-1)) = 0 + (1/2) * (1) = 0.5 y' = 0 **A' = (0.5, 0)**
- **Vertex B(1, 0):** x' = 1 + (1/2) * (0 - (-1)) = 1 + (1/2) * (1) = 1.5 y' = 0 **B' = (1.5, 0)**
- **Vertex C(1, 1):** x' = 1 + (1/2) * (1 - (-1)) = 1 + (1/2) * (2) = 1 + 1 = 2 y' = 1 **C' = (2, 1)**
- **Vertex D(0, 1):** x' = 0 + (1/2) * (1 - (-1)) = 0 + (1/2) * (2) = 0 + 1 = 1 y' = 1 **D' = (1, 1)**

4. The New Coordinates of the Sheared Unit Square: The sheared unit square has vertices at: A'(0.5, 0) B'(1.5, 0) C'(2, 1) D'(1, 1)

In summary, the unit square after being sheared in the x-direction with a shear parameter of ½ relative to the line y = -1 has the new coordinates A'(0.5, 0), B'(1.5, 0), C'(2, 1), and D'(1, 1).

14	Shear the unit square in y direction with shear parameter ½ relative to line x=(-1).	5

Answer 14:

To shear the unit square in the y-direction with a shear parameter of ½ relative to the line x = -1, we need to follow these steps:

1. Define the Vertices of the Unit Square: The unit square has vertices at: A(0, 0) B(1, 0) C(1, 1) D(0, 1)

2. Understand the Shear Transformation Relative to a Line: A shear in the y-direction relative to a line x = x_ref is given by the following transformation equations: x' = x y' = y + shy * (x - x_ref)

where:

- (x, y) are the original coordinates.
- (x', y') are the new coordinates after shear.
- shy is the shear parameter (given as ½).
- x_ref is the x-coordinate of the reference line (given as -1).

3. Apply the Shear Transformation to Each Vertex:

- **Vertex A(0, 0):** $x' = 0$ $y' = 0 + (1/2) * (0 - (-1)) = 0 + (1/2) * (1) = 0.5$ **A' = (0, 0.5)**
- **Vertex B(1, 0):** $x' = 1$ $y' = 0 + (1/2) * (1 - (-1)) = 0 + (1/2) * (2) = 0 + 1 = 1$ **B' = (1, 1)**
- **Vertex C(1, 1):** $x' = 1$ $y' = 1 + (1/2) * (1 - (-1)) = 1 + (1/2) * (2) = 1 + 1 = 2$ **C' = (1, 2)**
- **Vertex D(0, 1):** $x' = 0$ $y' = 1 + (1/2) * (0 - (-1)) = 1 + (1/2) * (1) = 1 + 0.5 = 1.5$ **D' = (0, 1.5)**

4. The New Coordinates of the Sheared Unit Square: The sheared unit square has vertices at: A'(0, 0.5) B'(1, 1) C'(1, 2) D'(0, 1.5)

In summary, the unit square after being sheared in the y-direction with a shear parameter of ½ relative to the line x = -1 has the new coordinates A'(0, 0.5), B'(1, 1), C'(1, 2), and D'(0, 1.5).

15	Translate the given point P(10,10,10) into 3D space with translation factor T(10,20,5).	4

Answer 15:

To translate a point P(x, y, z) in 3D space by a translation vector T(tx, ty, tz), you simply add the corresponding components of the translation vector to the coordinates of the point.

Given point P(10, 10, 10) and translation factor T(10, 20, 5).

The new coordinates P'(x', y', z') after translation will be:

$x' = x + tx$ $y' = y + ty$ $z' = z + tz$

Substituting the given values:

$x' = 10 + 10 = 20$ $y' = 10 + 20 = 30$ $z' = 10 + 5 = 15$

Therefore, the translated point P' has the coordinates **(20, 30, 15)**.

16	Rotate the point P(5,5,5) 90 degrees about Z-axis.	4

Answer :16

To rotate a point P(x, y, z) by an angle θ about the Z-axis, the transformation equations for the new coordinates P'(x', y', z') are:

$x' = x * \cos(\theta) - y * \sin(\theta)$ $y' = x * \sin(\theta) + y * \cos(\theta)$ $z' = z$

In this case, the point P is (5, 5, 5) and the rotation angle θ is 90 degrees.

First, convert the angle to radians if your trigonometric functions expect radians. However, most programming environments and calculators can handle degrees directly. $\cos(90°) = 0$ $\sin(90°) = 1$

Now, substitute the values into the rotation equations:

x' = 5 * cos(90°) - 5 * sin(90°) = 5 * 0 - 5 * 1 = 0 - 5 = -5

y' = 5 * sin(90°) + 5 * cos(90°) = 5 * 1 + 5 * 0 = 5 + 0 = 5

z' = z = 5

Therefore, the coordinates of the point P after a 90-degree rotation about the Z-axis are **(-5, 5, 5)**.

17	Scale the line AB with coordinates (10,20,10) and (20,30,30) respectively with scale factor S(3,2,4).	5

Answer 17:

To scale a line segment in 3D space, you need to scale each of its endpoints individually using the given scale factors. Let the endpoints of the line AB be A(x1, y1, z1) and B(x2, y2, z2), and the scale factor be S(sx, sy, sz).

The new coordinates of the scaled endpoints A'(x1', y1', z1') and B'(x2', y2', z2') will be:

For point A(10, 20, 10): x1' = x1 * sx = 10 * 3 = 30 y1' = y1 * sy = 20 * 2 = 40 z1' = z1 * sz = 10 * 4 = 40 **So, the new coordinates of A are A'(30, 40, 40).**

For point B(20, 30, 30): x2' = x2 * sx = 20 * 3 = 60 y2' = y2 * sy = 30 * 2 = 60 z2' = z2 * sz = 30 * 4 = 120 **So, the new coordinates of B are B'(60, 60, 120).**

Therefore, the scaled line segment A'B' has the coordinates **A'(30, 40, 40)** and **B'(60, 60, 120)**

18	Given a triangle with points (1,1), (0,0) and (1,0). Apply shear parameter 5 on X axis and 3 on Y axis and find out the new coordinates of the object.	5

Answer 18:

o apply a shear transformation with different parameters on the X and Y axes, we need to consider them as two separate shear transformations.

1. Shear along the X-axis with a shear parameter of 5:

The transformation matrix for shear along the X-axis is:

[1 shx 0]
[0   1   0]
[0   0   1]

where shx is the shear parameter in the X direction. In this case, shx = 5.

Applying this to each point (x, y) of the triangle, the new coordinates (x', y') will be: x' = x + shx * y y' = y

Let's apply this to the vertices:

- **Point (1, 1):** x' = 1 + 5 * 1 = 1 + 5 = 6 y' = 1 New coordinate: (6, 1)
- **Point (0, 0):** x' = 0 + 5 * 0 = 0 + 0 = 0 y' = 0 New coordinate: (0, 0)
- **Point (1, 0):** x' = 1 + 5 * 0 = 1 + 0 = 1 y' = 0 New coordinate: (1, 0)

After shearing along the X-axis with a parameter of 5, the new coordinates are (6, 1), (0, 0), and (1, 0).

2. Shear along the Y-axis with a shear parameter of 3:

The transformation matrix for shear along the Y-axis is:

```
[ 1   0   0 ]
[ shy  1   0 ]
[ 0   0   1 ]
```

where shy is the shear parameter in the Y direction. In this case, shy = 3.

Applying this to each point (x, y) *that resulted from the first shear*, the final new coordinates (x'', y'') will be: x'' = x' y'' = y' + shy * x'

Let's apply this to the coordinates obtained after the X-axis shear:

- **Point (6, 1):** x'' = 6 y'' = 1 + 3 * 6 = 1 + 18 = 19 Final coordinate: (6, 19)
- **Point (0, 0):** x'' = 0 y'' = 0 + 3 * 0 = 0 + 0 = 0 Final coordinate: (0, 0)
- **Point (1, 0):** x'' = 1 y'' = 0 + 3 * 1 = 0 + 3 = 3 Final coordinate: (1, 3)

Therefore, the final coordinates of the triangle after applying a shear parameter of 5 on the X-axis and then a shear parameter of 3 on the Y-axis are:

- **(6, 19)**
- **(0, 0)**
- **(1, 3)**

19	Given a triangle with coordinate points A(3,4), B(6,4), C(5,6). Apply the reflection on the XY axis and obtain the new coordinates of the object.	5

Answer 19:

To reflect a point (x, y) about the XY-axis (which is essentially a reflection across the origin in 2D, where both x and y coordinates change sign), the new coordinates (x', y') are given by the rule:

x' = -x y' = -y

Let's apply this rule to each vertex of the triangle:

Vertex A(3, 4): A'(x', y') = (-3, -4) **A' = (-3, -4)**

Vertex B(6, 4): B'(x', y') = (-6, -4) **B' = (-6, -4)**

Vertex C(5, 6): C'(x', y') = (-5, -6) **C' = (-5, -6)**

Therefore, the new coordinates of the triangle after reflection about the XY-axis (origin in 2D) are:

- **A' (-3, -4)**
- **B' (-6, -4)**
- **C' (-5, -6)**

Note: If the question intended a reflection across the X-axis only, the rule would be (x, y) becomes (x, -y). If it intended a reflection across the Y-axis only, the rule would be (x, y) becomes (-x, y). Since "XY axis" typically refers to the origin in 2D transformations, the above solution reflects across the origin.

20	Given a square object with coordinate points A(0,3), B(3,3), C(3,0), D(0,0). Apply the scaling parameter 4 towards X axis and 6 towards Y axis and obtain the new coordinates of the object.	5

Answer 20:

To apply scaling to a 2D object, we multiply the x-coordinate of each point by the scaling factor in the x-direction (Sx) and the y-coordinate by the scaling factor in the y-direction (Sy).

Given the square object with coordinates: A(0, 3) B(3, 3) C(3, 0) D(0, 0)

And the scaling parameters: Sx = 4 (towards the X-axis) Sy = 6 (towards the Y-axis)

Let's calculate the new coordinates for each point:

Point A(0, 3): A'(x', y') = (Ax * Sx, Ay * Sy) A'(x', y') = (0 * 4, 3 * 6) A' = (0, 18)

Point B(3, 3): B'(x', y') = (Bx * Sx, By * Sy) B'(x', y') = (3 * 4, 3 * 6) B' = (12, 18)

Point C(3, 0): C'(x', y') = (Cx * Sx, Cy * Sy) C'(x', y') = (3 * 4, 0 * 6) C' = (12, 0)

Point D(0, 0): D'(x', y') = (Dx * Sx, Dy * Sy) D'(x', y') = (0 * 4, 0 * 6) D' = (0, 0)

Therefore, the new coordinates of the scaled square object are:

- **A' (0, 18)**
- **B' (12, 18)**
- **C' (12, 0)**

- **D' (0, 0)**

21	Explain Composite Transformation for Translation, Rotation and Scaling.	5

Answer 21:

Composite transformation involves applying multiple geometric transformations sequentially to an object. For translation, rotation, and scaling, the order of application significantly impacts the final result due to the non-commutative nature of matrix multiplication.

To achieve a composite transformation, the individual transformation matrices are multiplied together. If we want to scale (S), then rotate (R), and finally translate (T) an object, the composite transformation matrix (M_composite) is calculated as:

M_composite = T * R * S

Applying this `M_composite` to a point (in homogeneous coordinates) performs all three transformations in the desired order.

Key Aspects:

1. **Order Matters:** The sequence of applying translation, rotation, and scaling yields different outcomes. For instance, scaling after translation affects the translated position, while scaling before translation affects the object's size before it's moved.
2.
3. **Matrix Multiplication:** Each basic transformation (translation, rotation, scaling) is represented by a specific homogeneous matrix (3x3 in 2D, 4x4 in 3D). The composite transformation matrix is obtained by multiplying these individual matrices in the reverse order of application.
4. **Efficiency:** Using a composite matrix is more efficient than applying each transformation matrix individually to every point of the object.
5. **Rotation/Scaling about Arbitrary Points:** Composite transformations are essential for performing rotations or scaling around points other than the origin. This involves translating the object so the arbitrary point is at the origin, performing the rotation/scaling, and then translating back.
6. **Unified Transformation:** The composite matrix encapsulates the entire sequence of transformations into a single matrix, simplifying the transformation process for complex operations.

	Unit-3 2D and 3D Viewing	
1	Explain Viewing pipeline.	4

Answer 1:

The **viewing pipeline** in computer graphics describes the sequence of transformations that convert a 3D scene description into a 2D image for display on a screen. It's a fundamental process that involves defining what to view, how to view it, and where to display it. Here's a breakdown of the key stages in a typical 3D viewing pipeline:

1. **Modeling Transformation (Object Coordinates to World Coordinates):**
   - Objects are initially defined in their own local coordinate systems (object coordinates).
   - Modeling transformations (translation, rotation, scaling) are applied to position and orient these objects within a common **world coordinate system**. This creates the overall 3D scene.
2. **Viewing Transformation (World Coordinates to Viewing/Camera Coordinates):**
   - To view the scene from a specific perspective, a **camera** or **viewing coordinate system** is defined. This involves specifying the camera's position, orientation (direction it's pointing), and the "up" direction.
   - The viewing transformation converts the coordinates of all objects from the world coordinate system to the camera's viewpoint. The camera is typically placed at the origin of this new coordinate system, looking down the negative Z-axis.
3. **Projection Transformation (Viewing Coordinates to Projection Coordinates):**
   - The 3D scene in viewing coordinates needs to be projected onto a 2D projection plane. This stage determines how the 3D objects will appear in 2D. Two main types of projections are:
     - **Perspective Projection:** Creates a realistic view with foreshortening (objects appear smaller as they are farther away). Defined by a view frustum (a truncated pyramid).
     - **Parallel Projection:** Preserves the relative sizes and shapes of objects, regardless of their distance. Defined by a view volume (a rectangular box or a cylinder).
   - This transformation maps the 3D viewing volume (frustum or box) into a normalized viewing volume (typically a cube from -1 to 1 in each dimension).
4. **Clipping (Projection Coordinates to Clipping Coordinates):**
   - Objects or parts of objects that lie outside the viewing volume (defined by the projection) are not visible and should be removed to improve rendering efficiency. This process is called **clipping**.
   - Clipping is performed against the boundaries of the normalized viewing volume.
5. **Viewport Transformation (Normalized Coordinates to Device Coordinates):**
   - The 2D projection of the visible scene (after clipping) is now in normalized device coordinates (typically ranging from -1 to 1 or 0 to 1).
   - The **viewport** is a rectangular region on the display screen where the final image will be rendered. The viewport transformation maps the normalized coordinates to the specific pixel coordinates of the viewport on the output device. This determines the size and position of the rendered image on the screen.

2	What is Coordinate Systems	4

Answer 2:

In computer graphics, **coordinate systems** are fundamental frameworks used to define the position and orientation of objects within a virtual space. They provide a structured way to use numerical values (coordinates) to uniquely identify points and describe geometric entities. Here's a breakdown of their importance and key aspects:

1. **Defining Position and Orientation:** Coordinate systems allow us to precisely specify where an object is located in space (its position) and how it is oriented (its rotation). This is essential for building and manipulating virtual scenes.

2. **Multiple Coordinate Spaces:** In a typical graphics pipeline, objects move through several different coordinate systems as they are transformed and prepared for rendering:

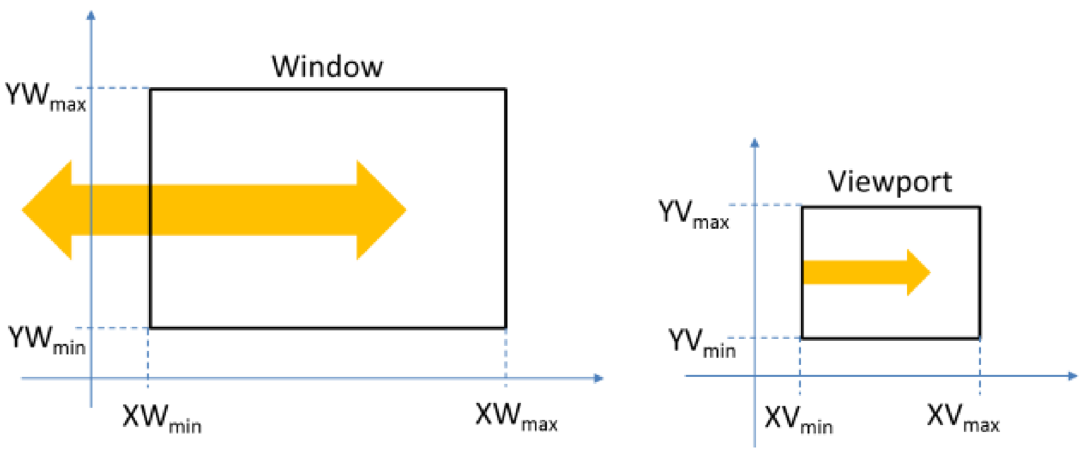
    - **Object/Local Space:** Each object has its own local coordinate system, making it easier to define its initial geometry.
    - **World Space:** All objects in the scene are placed within a common world coordinate system.
    - **View/Camera Space:** The scene is transformed relative to the camera's position and orientation.
    - **Clip Space:** Coordinates are transformed for perspective projection and clipping.
    - **Screen Space:** The final 2D coordinates are mapped to the pixels of the display screen.
3. **Types of Coordinate Systems:** Various types of coordinate systems are used in computer graphics, each with its advantages for specific tasks:

    - **Cartesian Coordinates (Rectangular):** Uses perpendicular axes (2D: x, y; 3D: x, y, z) to define positions as distances along these axes. Most commonly used.
    - **Polar Coordinates (2D):** Defines a point by its distance (radius) from an origin and the angle (theta) from a reference axis. Useful for circular or radial arrangements.
    - **Cylindrical Coordinates (3D):** Extends polar coordinates by adding a height (z-coordinate). Useful for objects with cylindrical symmetry.
    - **Spherical Coordinates (3D):** Defines a point by its distance (radius) from an origin and two angles (azimuth and elevation). Useful for objects with spherical symmetry or for representing directions.
    - **Homogeneous Coordinates:** Extends Cartesian coordinates by adding an extra dimension (w). This allows affine transformations (translation, rotation, scaling, shear) to be represented as matrix multiplications, simplifying complex transformations and perspective projection.
4. **Transformations Between Systems:** It's often necessary to convert coordinates between different coordinate systems. This is achieved using mathematical transformations (e.g., rotation matrices, translation vectors). These transformations ensure that objects are correctly positioned and oriented as they move through the viewing pipeline.

3	Explain window-to-viewport transformation.	4

Answer 3:

The **window-to-viewport transformation** is a crucial step in the 2D viewing pipeline of computer graphics. It's the process of mapping a rectangular region in world coordinates (the **window**) to a rectangular region on the display device (the **viewport**). This transformation ensures that the desired part of the 2D scene is displayed correctly on the screen, taking into account the size and aspect ratio of both the window and the viewport.



1. **Defining the Display:** The **viewport** defines the area on the screen (in device coordinates, usually pixels) where the image will be rendered. [1] It specifies the position (e.g., top-left corner) and dimensions (width and height) of this rectangular area.
2. **Selecting the Scene:** The **window** defines a rectangular area in the world coordinate system that the user wants to view. It specifies the minimum and maximum x and y world coordinates that should be mapped to the viewport.
3. **Maintaining Relative Positions:** The core of the transformation is to map a point (Xw, Yw) within the window to a corresponding point (Xv, Yv) within the viewport such that the relative position of the point within its respective rectangle is maintained. This means if a point is in the center of the window, it will be in the center of the viewport after transformation.
4. **Scaling and Translation:** The window-to-viewport transformation typically involves two main operations:

    ○ **Scaling:** The world coordinates within the window are scaled to fit the size of the viewport. The scaling factors in the x and y directions might be different to accommodate different aspect ratios.
    ○ **Translation:** After scaling, the scaled coordinates are translated to the correct position within the viewport on the display screen.

Mathematically, for a window defined by (Xw_min, Yw_min) and (Xw_max, Yw_max), and a viewport defined by (Xv_min, Yv_min) and (Xv_max, Yv_max), a point (Xw, Yw) in the window is mapped to (Xv, Yv) in the viewport using the following formulas:

$$\frac{x_v - x_{vmin}}{x_{vmax} - x_{vmin}} = \frac{x_w - x_{wmin}}{x_{wmax} - x_{wmin}}$$

$$\frac{y_v - y_{vmin}}{y_{vmax} - y_{vmin}} = \frac{y_w - y_{wmin}}{y_{wmax} - y_{wmin}}$$

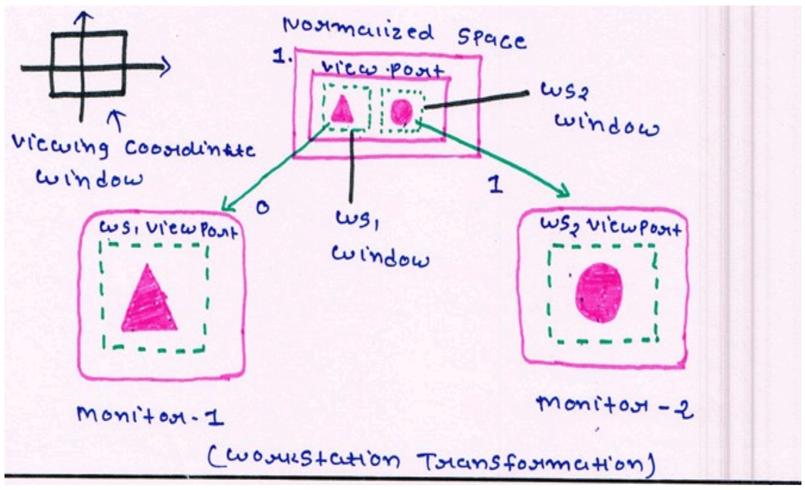
Solving by making viewport position as subject we obtain:

xv = xvmin + (xw − xwmin)sx

yv = yvmin + (yw − ywmin)sy

Where scaling factor are :

$$s_x = \frac{x_{vmax} - x_{vmin}}{x_{wmax} - x_{wmin}}$$
$$s_y = \frac{y_{vmax} - y_{vmin}}{y_{wmax} - y_{wmin}}$$

4	Explain workstation transformation monitor -2 	4

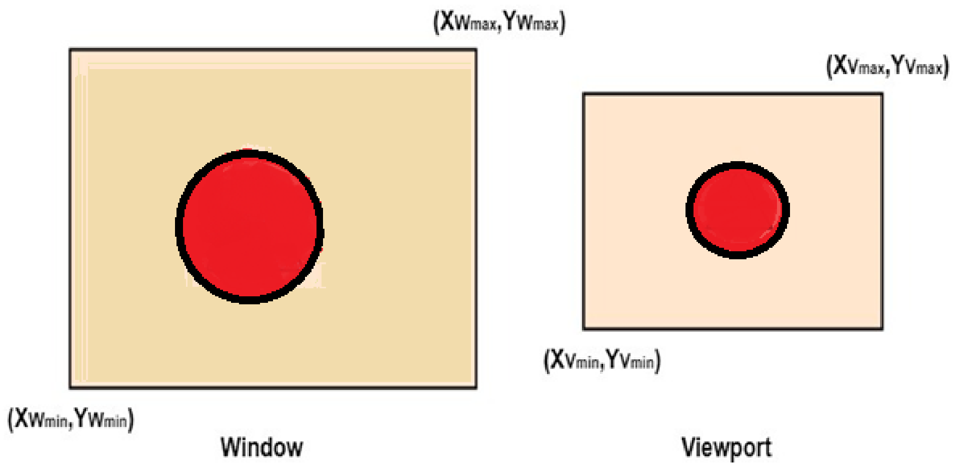
Answer 4:

1. **Normalized Space**:
    - The top rectangle represents the full scene in **normalized coordinates (0 to 1)**.
    - Different sections (windows) of this scene are selected for display on different monitors.
2. **Window Selection for Monitor 2 (WS2)**:
    - The **WS2 window** (dashed rectangle) selects a portion of the normalized space that contains a **black circle**.

- This portion is extracted and mapped onto **Monitor 2**.

3. **Viewport on Monitor 2**:
   - The selected WS2 window is displayed in the **WS2 viewport** on **Monitor 2**.
   - The black circle is now shown on Monitor 2, ensuring the mapping is correctly transformed.

Matrix Representation of the above three steps of Transformation:



Step1:Translate window to origin 1

$T_x=-Xw_{min}$ $T_y=-Yw_{min}$

Step2:Scaling of the window to match its size to the viewport

$S_x=(Xy_{max}-Xv_{min})/(Xw_{max}-Xw_{min})$

$S_y=(Yv_{max}-Yv_{min})/(Yw_{max}-Yw_{min})$

Step3:Again translate viewport to its correct position on screen.

$T_x=Xv_{min}$

$T_y=Yv_{min}$

Above three steps can be represented in matrix form:

$VT=T * S * T_1$

T = Translate window to the origin

S=Scaling of the window to viewport size

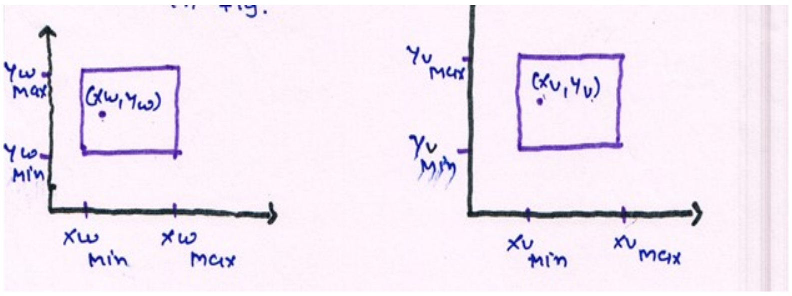
T_1=Translating viewport on screen.

$$T = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -Xw_{min} & -Yw_{min} & 1 \end{vmatrix}$$

$$S = \begin{vmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

$$T_1 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Xv_{min} & Yv_{min} & 1 \end{vmatrix}$$

Viewing Transformation= T * S * T1

5	Explain the window to view port transformation.write step by step formula  Yw max Yw min (xw,yw) xw min xw max Yv max (xv,yv) Yv min xv min xv max	5

Answer 5:

The window-to-viewport transformation is the process of mapping a rectangular region in world coordinates (the **window**) to a rectangular region on the display device (the **viewport**). This ensures the desired part of the 2D scene is displayed correctly on the screen.

Here's a step-by-step explanation with the formulas:

1. Define the Window: The window is defined in world coordinates by its minimum and maximum x and y values:

- `Xw_min`: Minimum x-coordinate of the window.
- `Yw_min`: Minimum y-coordinate of the window.
- `Xw_max`: Maximum x-coordinate of the window.
- `Yw_max`: Maximum y-coordinate of the window.

2. Define the Viewport: The viewport is defined in device coordinates (usually pixels) by its minimum and maximum x and y values:

- `Xv_min`: Minimum x-coordinate of the viewport (e.g., left edge of the screen area).
- `Yv_min`: Minimum y-coordinate of the viewport (e.g., bottom edge of the screen area, as screen coordinates often increase upwards).
- `Xv_max`: Maximum x-coordinate of the viewport (e.g., right edge of the screen area).
- `Yv_max`: Maximum y-coordinate of the viewport (e.g., top edge of the screen area).

3. Transformation Steps and Formulas:

For a point (Xw, Yw) in the window, its corresponding point (Xv, Yv) in the viewport is calculated in two main steps: scaling and translation.

Step 3.1: Calculate the Normalized Position within the Window: First, determine the relative position of the world coordinate point within the window, ranging from 0 to 1 in both x and y directions.

Normalized X (Nx):

Nx = (Xw - Xw_min) / (Xw_max - Xw_min)

- This formula calculates the fraction of the window's width that the x-coordinate Xw has traversed from the left edge.

Normalized Y (Ny):

Ny = (Yw - Yw_min) / (Yw_max - Yw_min)

- This formula calculates the fraction of the window's height that the y-coordinate Yw has traversed from the bottom edge.

Step 3.2: Map the Normalized Position to Viewport Coordinates: Next, map these normalized values to the range of the viewport coordinates.

Viewport X (Xv):

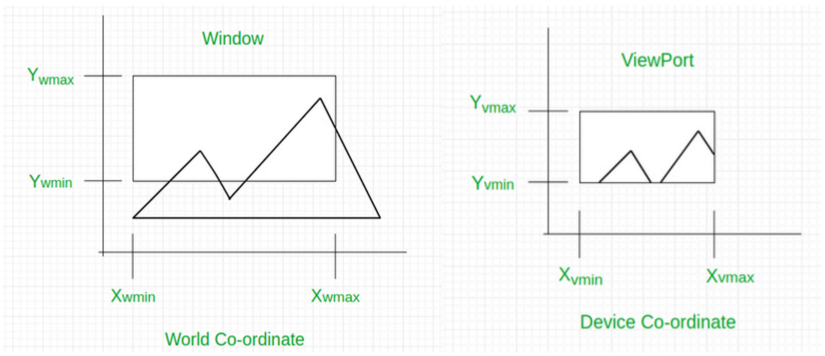
Xv = Xv_min + Nx * (Xv_max - Xv_min)

This formula scales the normalized x-value by the width of the viewport and adds the viewport's minimum x-coordinate to position it correctly.

Viewport Y (Yv):

Yv = Yv_min + Ny * (Yv_max - Yv_min)

This formula scales the normalized y-value by the height of the viewport and adds the viewport's minimum y-coordinate to position it correctly.


Window
Y_{wmax}
Y_{wmin}
X_{wmin}
X_{wmax}
World Co-ordinate
ViewPort
Y_{vmax}
Y_{vmin}
X_{vmin}
X_{vmax}
Device Co-ordinate