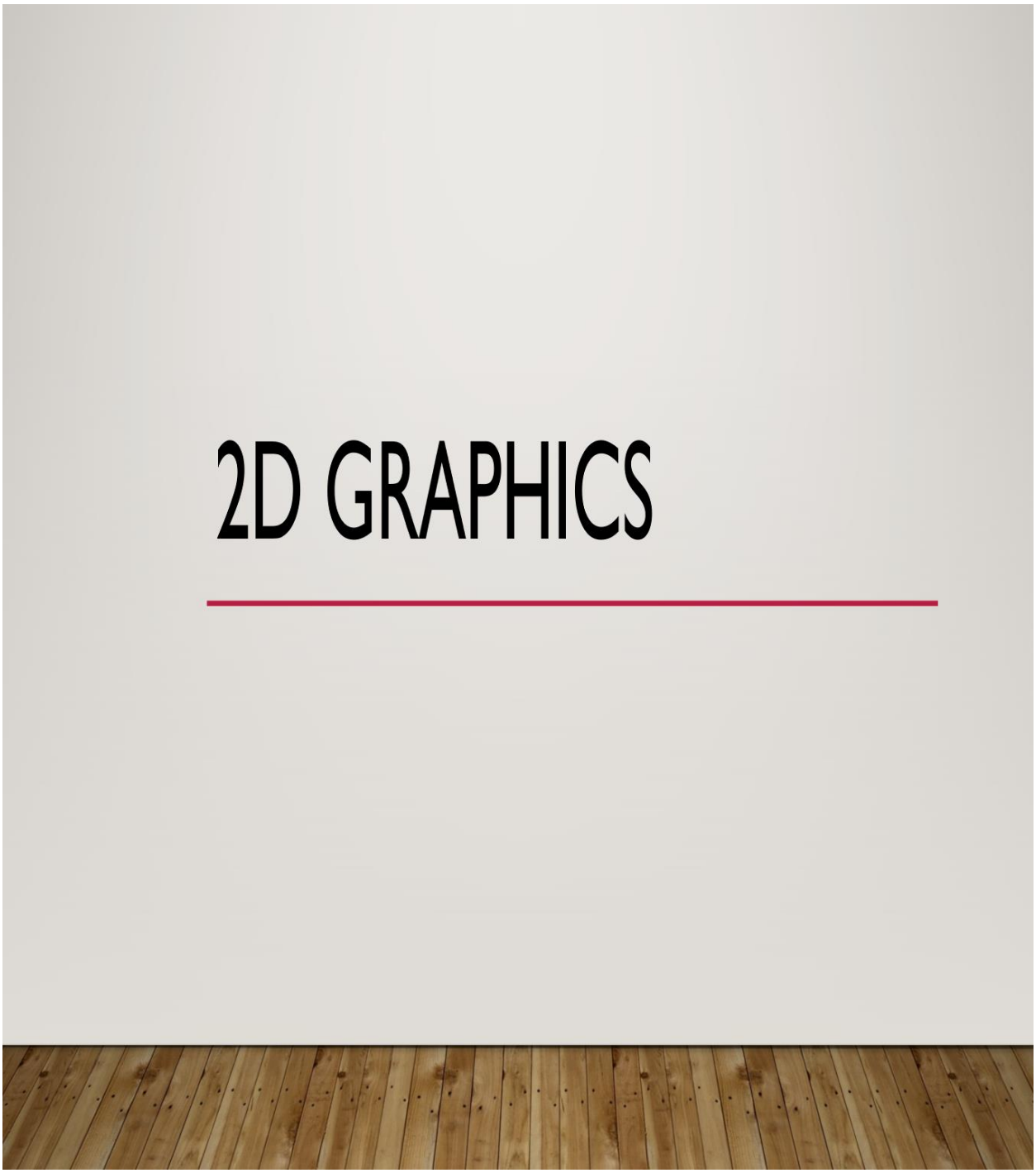
COMPUTER GRAPHICS LAB MANUAL JAVA

M SUMANA

ASSISTANT PROFESSOR,CSE

VELS UNIVERSITY(VISTAS)


2D GRAPHICS

2D COMPUTER GRAPHICS

2D computer graphics is the creation, manipulation, and representation of images in a two-dimensional plane using computers. It involves the use of digital images, geometric shapes, text, and colors on a flat surface defined by height and width, without depth. This type of graphics is widely used in fields like desktop publishing, technical drawing, typography, cartography, and user interfaces.

Key Concepts of 2D Computer Graphics:

- Pixels and Coordinates: Images are made up of pixels arranged in a grid. The coordinate system typically starts at the top-left corner with x increasing to the right and y increasing downward.
- Geometric Primitives: Basic shapes such as points, lines, curves, and polygons are used as building blocks.
- Transformations: Operations like translation (moving), rotation, scaling (resizing), reflection, and shearing manipulate objects within the 2D space.
- Color Representation: Colors are assigned to pixels or shapes using different color models like RGB.
- Vector vs Raster Graphics: Vector graphics use mathematical descriptions of shapes, allowing infinite scaling without loss of quality, while raster graphics are pixel-based images.

Applications and Characteristics:

- 2D graphics provide more direct control over images than 3D graphics, making them ideal for static images, diagrams, typographic design, and simple animations.
- Digitally stored 2D images can be more compact, flexible for different resolutions, and easily edited or transmitted.
- Common output devices include monitors and printers, often using coordinate transformations to place and draw objects correctly on screen.

Overall, 2D computer graphics is a foundational aspect of visual computing, enabling everything from simple image editing to complex graphical user interfaces and animations

3D COMPUTER GRAPHICS

3D computer graphics refers to the creation, manipulation, and representation of objects in three-dimensional space using computers. Unlike 2D graphics which operate only on height and width, 3D graphics add depth as the third dimension, enabling more realistic and immersive images and animations.

Key Aspects of 3D Computer Graphics:

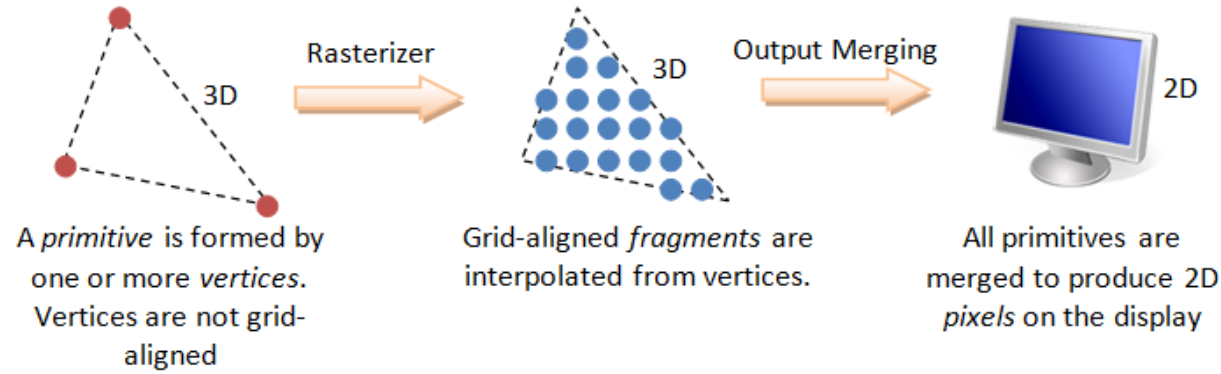
- 3D Modeling: The process of creating a mathematical representation of an object's shape using vertices (points), edges (lines), and faces (polygons). This creates a 3D model that can be manipulated.
- Layout and Animation: Placing 3D models in a scene while defining their positions, orientations, and movements over time through techniques like keyframing and motion capture.
- Rendering: The computation that converts 3D models into 2D images by simulating lighting, shading, textures, and camera perspective, producing photorealistic or stylized images.
- Coordinate System: 3D graphics use Cartesian coordinates (X, Y, Z) where X and Y define the horizontal and vertical dimensions, and Z defines depth.
- Application Areas: Used extensively in video games, movies, virtual reality, architecture, engineering, and simulation.

Workflow Phases in 3D Graphics:

1. Modeling: Design the object's shape.
2. Texturing: Apply surface colors and patterns.
3. Lighting: Define how lights affect the model.
4. Rendering: Generate the final 2D image or animation from the 3D scene.

3D graphics combine complex algorithms for realistic representation, but the images are mostly displayed on 2D screens. Techniques like perspective projection help simulate depth on flat displays.

Overall, 3D computer graphics enable realistic visualization by representing objects with depth and spatial relationships, enriching digital content beyond flat images.



Vertex, Primitives, Fragment and Pixel

Course Objectives

- To make the students understand graphics concepts and develop, design and implement two and three-dimensional graphical structures using OpenGL.
- To understand multimedia compression techniques and applications of multimedia.

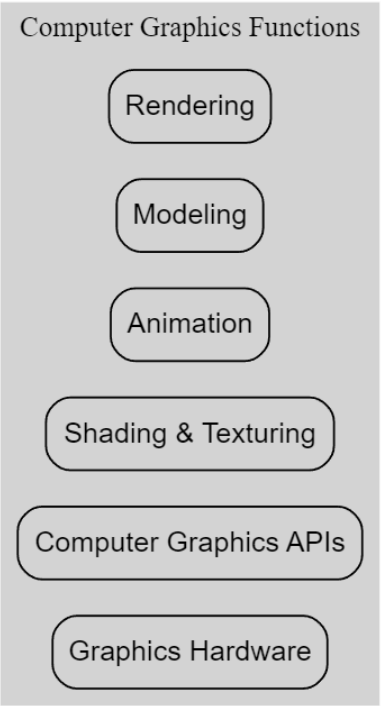
Experiments:

1. Study of Fundamental Graphics Functions.
2. Implementation of Line drawing algorithms: DDA Algorithm, Bresenham's Algorithm
3. Implementation of Circle drawing algorithms: Bresenham's Algorithm, Mid-Point Algorithm.
4. Programs on 2D and 3D transformations
5. Write a program to implement Cohen Sutherland line clipping algorithm
6. Write a program to draw Bezier curve.
7. Using Image Editing Software Tool-GIMP3.0 perform different operations (rotation, scaling move etc..) on objects
8. To perform animation using any Animation software (Create Frame by Frame Animations using multimedia authoring tools-GIMP3.0)
9. To perform basic operations on image using any image editing software-GIMP3.0

Ex No 1 Study of Fundamental Graphics Functions

Functions of Computer Graphics

Computer Graphics Functions have lots of techniques and tools used to create, manipulate & display visual content on digital platforms. These functions perform the generation of images, animations & interactive experiences through specialized software and hardware.
Example: Consider a scenario where a game developer made computer graphics functions to create a 3D environment. They use modeling techniques to craft detailed characters & environments apply textures and shaders to simulate materials realistically & use rendering techniques to display the final scene on a screen.



Techniques

- Making Pictures Look Real: There are different ways to make computer images look super realistic. Some methods include turning shapes into images (rasterization), tracing light beams to create images (ray tracing) & using fancy techniques for lifelike lighting.
- Creating 3D Stuff: When people make 3D things on computers they use different methods. Some use shapes made of lots of sides (polygon Modeling) & some use mathematical curves called NURBS and others Mold things like clay known as sculpting and some use rules to make things automatically such as procedural Modeling.
- Making Things Move: To make things in these images move there are tricks too. Some involve setting specific points in time for movement called keyframing, creating skeletons for things to move along (rigging), recording real movement such as motion capture & using rules for movement called animation.
- Painting and Texturing: To make these 3D things look more real designers painted and textured. Think of it like applying paint and patterns to give them the look of real materials like wood or metal.
- Teaching Computers to 'See': Computers can learn to understand what they 'see' too. There are programs that help them recognize objects out what is in a picture & even guess how far away things are.

Computer graphics has a wide range of applications, impacting fields like entertainment, design, education, and scientific visualization. It's used in creating digital art, animations, and video games, as well as in designing products, buildings, and simulations. Furthermore, computer graphics aids in data visualization, creating user interfaces, and even in medical imaging.
Here's a more detailed breakdown:

1. Design and Manufacturing:

- **Computer-Aided Design (CAD):**

  CAD software uses computer graphics to design products, buildings, and other complex structures with precision and detail.

- **Machine Drawing:**
  Computer graphics allows for clear and precise machine drawings, which are crucial for manufacturing processes.

2. Entertainment:

- **Animation:**

  Computer graphics is essential for creating animated movies, television shows, and cartoons.

- **Video Games:**

  Interactive computer graphics are the foundation of modern video games, providing immersive and engaging experiences.

- **Special Effects:**
  Computer graphics are used to create stunning visual effects in movies and other media.

3. Education and Training:
   - **Educational Models:**

     Computer graphics can create interactive models to explain complex scientific or engineering concepts.

   - **Simulations:**
     Simulations using computer graphics can train individuals in various fields, such as flight simulation for pilots.

4. Scientific Visualization:
   - **Data Visualization:**

     Computer graphics helps in visualizing complex scientific data, making it easier to understand trends and patterns.

   - **Medical Imaging:**
     Computer graphics plays a role in medical imaging, aiding in diagnosis and treatment planning.

5. Other Applications:
   - **Presentation Graphics:**

     Computer graphics is used to create charts, graphs, and other visual aids for presentations.

   - **Graphical User Interfaces (GUIs):**

     GUIs, with their icons and menus, rely on computer graphics for intuitive user interaction.

   - **Computer Art:**

     Digital painting and other forms of computer art are created using computer graphics software.

   - **Image Processing:**
     Techniques in computer graphics are used to enhance, edit, and analyze images.

Procedure for Java program in Eclipse

The general procedure for creating and running a Java program in Eclipse involves the following steps:

- **Launch Eclipse and Select Workspace:**
    - Open the Eclipse IDE.
    - Choose a suitable directory for your workspace, which is where your projects and files will be stored.
- **Create a New Java Project:**
    - Navigate to File > New > Java Project.
    - Provide a name for your project (e.g., "MyJavaProject").
    - Ensure the appropriate JRE (Java Runtime Environment) is selected.
    - It is generally recommended to uncheck "Create module-info.java file" for basic projects.
    - Click "Finish".
- **Create a Java Class:**
    - In the "Package Explorer" (usually on the left), expand your newly created project.
    - Right-click on the src folder.
    - Select New > Class.
    - Enter a package name (e.g., JavaProgram) and a class name (e.g., HelloWorld).
    - Check the option "public static void main(String[] args)" if you want Eclipse to automatically generate the main method for your executable class.
    - Click "Finish".
- **Write Java Code:**
    - The newly created Java class file will open in the editor.
    - Write your Java code within the class, including your main method if you created an executable class.
    - For example, a simple "Hello World" program would look like this:

Simple Java Program

```
package JavaProgram;

    public class HelloWorld {
       public static void main(String[] args) {
          System.out.println("Hello, World!");
       }
    }
```

- **Run the Java Application:**
    - Right-click anywhere within the Java class editor.
    - Select Run As > Java Application.
    - The output of your program will appear in the "Console" view at the bottom of the Eclipse window.

Ex No 2 a). Implementation of Line drawing algorithms: DDA Algorithm

DDA Line Generation

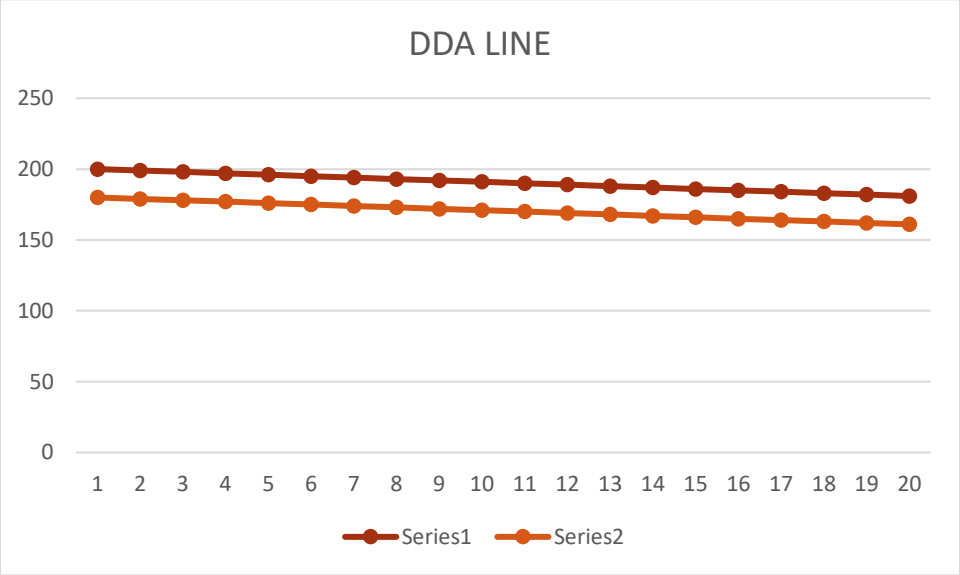
```
// Java Code for DDA line generation
public class Solution {

  // function for rounding off the pixels
  public static int round(float n) {
    if (n - (int) n < 0.5)
      return (int) n;
    return (int) (n + 1);
  }

  // Function for line generation
  public static void DDALine(int x0, int y0, int x1, int y1) {

    // Calculate dx and dy
    int dx = x1 - x0;
    int dy = y1 - y0;

    int step;

    // If dx > dy we will take step as dx
    // else we will take step as dy to draw the complete
    // line
    if (Math.abs(dx) > Math.abs(dy))
      step = Math.abs(dx);
    else
      step = Math.abs(dy);

    // Calculate x-increment and y-increment for each step
    float x_incr = (float) dx / step;
    float y_incr = (float) dy / step;

    // Take the initial points as x and y
    float x = x0;
    float y = y0;

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

      // putpixel(round(x), round(y), WHITE);
      System.out.println(round(x) + " " + round(y));
```

```
    x += x_incr;
    y += y_incr;
    // delay(10);
   }
  }

  // Driver code
  public static void main(String[] args) {

   int x0 = 200, y0 = 180, x1 = 180, y1 = 160;

   // Function call
   DDALine(x0, y0, x1, y1);

  }
}
```

Output:

```
200 180
199 179
198 178
197 177
196 176
195 175
194 174
193 173
192 172
191 171
190 170
189 169
188 168
187 167
186 166
185 165
184 164
183 163
182 162
181 161
```

DDA Line Generation Graph



ExNo 2 b). Implementation of Line drawing algorithms: Bresenham's Algorithm

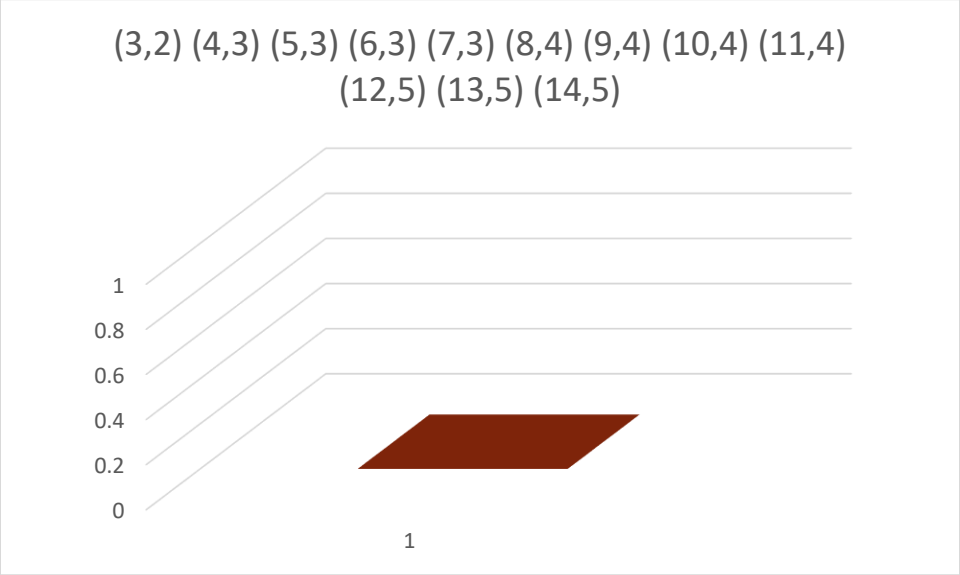
```
// Java program for Bresenhams Line Generation
// Assumptions :
// 1) Line is drawn from left to right.
// 2) x1 < x2 and y1 < y2
// 3) Slope of the line is between 0 and 1.
// We draw a line from lower left to upper
// right.
class GFG {
   // function for line generation

   static void bresenham(int x1, int y1, int x2, int y2)
   {
      int m_new = 2 * (y2 - y1);
      int slope_error_new = m_new - (x2 - x1);

      for (int x = x1, y = y1; x < = x2; x++) {
         System.out.print(
            "
            (" + x + ", " + y + ")\n
            & quot;);

         // Add slope to increment angle formed
         slope_error_new += m_new;

         // Slope error reached limit, time to
         // increment y and update slope error.
         if (slope_error_new& gt; = 0) {
            y++;
            slope_error_new -= 2 * (x2 - x1);
         }
      }
   }

   // Driver code
   public static void main(String[] args)
   {
      int x1 = 3, y1 = 2, x2 = 15, y2 = 5;
```

```
    // Function call
    bresenham(x1, y1, x2, y2); }}
```

Output

```
(3,2)
(4,3)
(5,3)
(6,3)
(7,3)
(8,4)
(9,4)
(10,4)
(11,4)
(12,5)
(13,5)
(14,5)
(15,5)
```

Bresenhams Line Graph



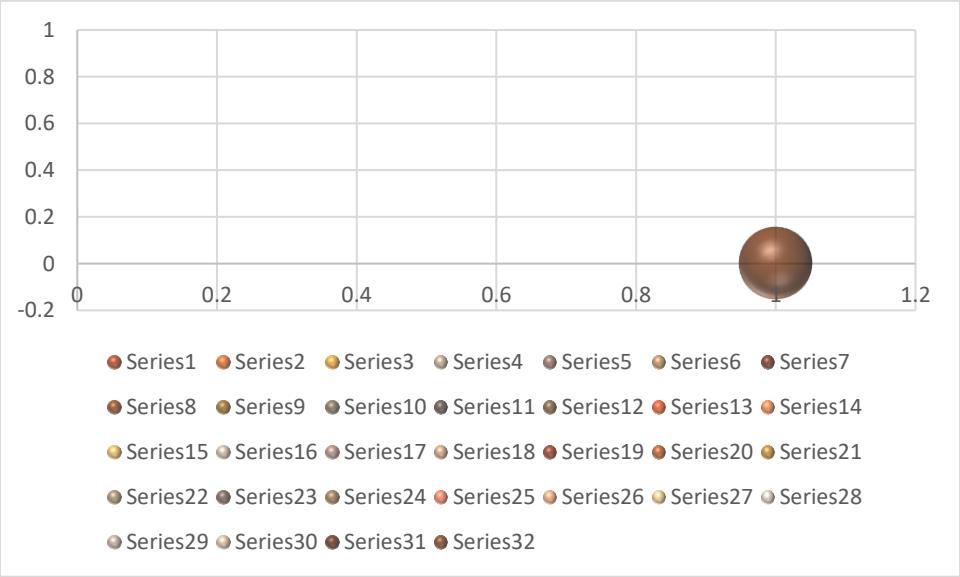
Ex No 3a). Implementation of Circle drawing algorithms: Bresenham's Algorithm.

```
public class BresenhamCircle {

    public static void drawCircle(int xc, int yc, int r) {
        int x = 0;
        int y = r;
        int d = 3 - 2 * r; // Initial decision parameter
        // Plot the initial point and its symmetric counterparts
        plotPoints(xc, yc, x, y);

        while (y >= x) {
            x++; // Increment x

            if (d > 0) { // If the midpoint is outside or on the circle
                y--; // Decrement y
                d = d + 4 * (x - y) + 10; // Update decision parameter
            } else { // If the midpoint is inside the circle
                d = d + 4 * x + 6; // Update decision parameter
            }

            // Plot the points and their symmetric counterparts
            plotPoints(xc, yc, x, y);
        } }
    // Helper function to plot all 8 symmetric points
    private static void plotPoints(int xc, int yc, int x, int y) {
        System.out.println("(" + (xc + x) + ", " + (yc + y) + ")");
        System.out.println("(" + (xc - x) + ", " + (yc + y) + ")");
        System.out.println("(" + (xc + x) + ", " + (yc - y) + ")");
        System.out.println("(" + (xc - x) + ", " + (yc - y) + ")");
        System.out.println("(" + (xc + y) + ", " + (yc + x) + ")");
        System.out.println("(" + (xc - y) + ", " + (yc + x) + ")");
        System.out.println("(" + (xc + y) + ", " + (yc - x) + ")");
        System.out.println("(" + (xc - y) + ", " + (yc - x) + ")");
    }
    public static void main(String[] args) {
        int centerX = 50;
        int centerY = 50;
        int radius = 10;

        System.out.println("Drawing a circle with center (" + centerX + ", " + centerY + ") and
radius " + radius + ":");drawCircle(centerX, centerY, radius);}}
```

output:

```
Drawing a circle with center (50, 50) and radius 10:
(50, 60)(50, 60)(50, 40)(50, 40)(60, 50)(40, 50)(60, 50)(40, 50)(51, 60)(49, 60)
(51, 40)(49, 40)(60, 51)(40, 51)(60, 49)(40, 49)(52, 60)(48, 60)(52, 40)(48, 40)
(60, 52)(40, 52)(60, 48)(40, 48)(53, 59)(47, 59)(53, 41)(47, 41)(59, 53)(41, 53)
(59, 47)(41, 47)(54, 59)(46, 59)(54, 41)(46, 41)(59, 54)(41, 54)(59, 46)(41, 46)
(55, 58)(45, 58)(55, 42)(45, 42)(58, 55)(42, 55)(58, 45)(42, 45)(56, 57)(44, 57)
(56, 43)(44, 43)(57, 56)(43, 56)(57, 44)(43, 44)(57, 56)(43, 56)(57, 44)(43, 44)
(56, 57)(44, 57)(56, 43)(44, 43)
```

BresenhamCircle Graph:



Ex no3 b). Implementation of Circle drawing algorithms: Mid-Point Algorithm.

// Java program for implementing

// Mid-Point Circle Drawing Algorithm

```
class GFG {
    // Implementing Mid-Point Circle
    // Drawing Algorithm
    static void midPointCircleDraw(int x_centre,
                        int y_centre, int r)
    {
        int x = r, y = 0;
        // Printing the initial point
        // on the axes after translation
        System.out.print("(" + (x + x_centre)
              + ", " + (y + y_centre) + ")");
        // When radius is zero only a single
        // point will be printed
        if (r > 0) {
            System.out.print("(" + (x + x_centre)
                + ", " + (-y + y_centre) + ")");
            System.out.print("(" + (y + x_centre)
                + ", " + (x + y_centre) + ")");
            System.out.println("(" + (-y + x_centre)
                + ", " + (x + y_centre) + ")");
        }
        // Initialising the value of P
        int P = 1 - r;
        while (x > y) {
            y++;
            // Mid-point is inside or on the perimeter
            if (P <= 0)
                P = P + 2 * y + 1;
            // Mid-point is outside the perimeter
            else {
                x--;
                P = P + 2 * y - 2 * x + 1;
            }
            // All the perimeter points have already
            // been printed
            if (x < y)
                break;
            // Printing the generated point and its
            // reflection in the other octants after
            // translation
```

```
            System.out.print("(" + (x + x_centre)
                    + ", " + (y + y_centre) + ")");

            System.out.print("(" + (-x + x_centre)
                    + ", " + (y + y_centre) + ")");

            System.out.print("(" + (x + x_centre) +
                    ", " + (-y + y_centre) + ")");

            System.out.println("(" + (-x + x_centre)
                    + ", " + (-y + y_centre) + ")");
            // If the generated point is on the
            // line x = y then the perimeter points
            // have already been printed
            if (x != y) {

                System.out.print("(" + (y + x_centre)
                        + ", " + (x + y_centre) + ")");

                System.out.print("(" + (-y + x_centre)
                        + ", " + (x + y_centre) + ")");

                System.out.print("(" + (y + x_centre)
                        + ", " + (-x + y_centre) + ")");

                System.out.println("(" + (-y + x_centre)
                    + ", " + (-x + y_centre) +")");
            }
        }
    }

    // Driver code
    public static void main(String[] args) {

        // To draw a circle of radius
        // 3 centered at (0, 0)
        midPointCircleDraw(0, 0, 3);
    }
}
```

Output:

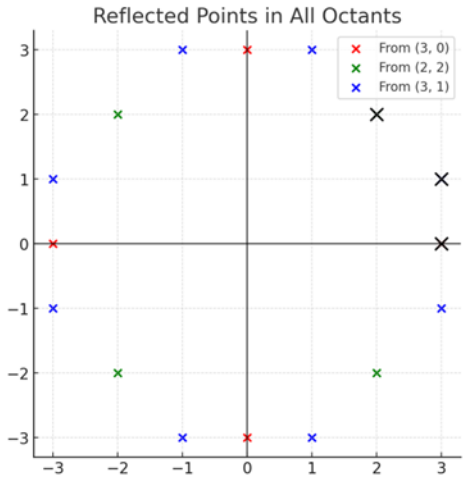
(3, 0) (3, 0) (0, 3) (0, 3)

(3, 1) (-3, 1) (3, -1) (-3, -1)

(1, 3) (-1, 3) (1, -3) (-1, -3)

(2, 2) (-2, 2) (2, -2) (-2, -2)

Mid-Point Circle Graph:



Ex no:4 a). Programs on 2D Transformations

2D Transformation | Rotation of objects

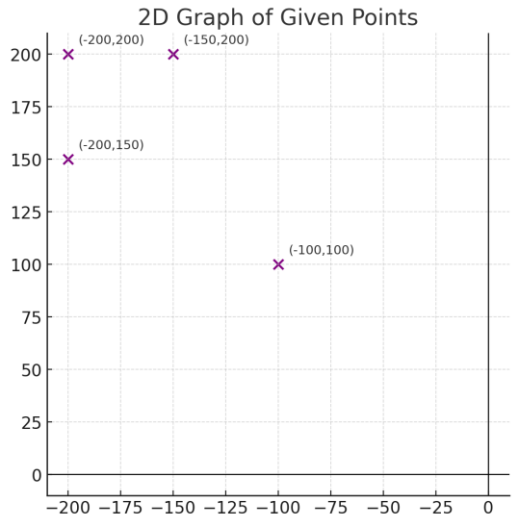
```
// Java program to rotate an object by
// a given angle about a given point
public class rotation {

  static void rotate(double a[][], int n, int x_pivot,
                     int y_pivot, int angle)
  {
    int i = 0;
    while (i < n)
    {

      // Shifting the pivot point to the origin
      // and the given points accordingly
      int x_shifted = (int)a[i][0] - x_pivot;
      int y_shifted = (int)a[i][1] - y_pivot;

      // Calculating the rotated point co-ordinates
      // and shifting it back
      double x = Math.toRadians(angle);
      a[i][0] = x_pivot
        + (x_shifted * Math.cos(x)
          - y_shifted * Math.sin(x));
      a[i][1] = y_pivot
        + (x_shifted * Math.sin(x)
          + y_shifted * Math.cos(x));
      System.out.printf("(%f, %f) ", a[i][0],
                        a[i][1]);
      i++;
    }
  }

  // Driver Code
  public static void main(String[] args)
  {
    // 1st Example
    // The following figure is to be
    // rotated about (0, 0) by 90 degrees
    int size1 = 4; // No. of vertices

    // Vertex co-ordinates must be in order
    double points_list1[][] = { { 100, 100 },
```

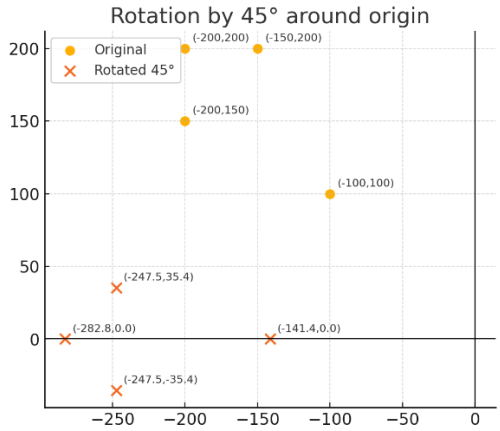
```
                        { 150, 200 },
                        { 200, 200 },
                        { 200, 150 } };
    rotate(points_list1, size1, 0, 0, 90);

    // 2nd Example
    // The following figure is to be
    // rotated about (50, -50) by -45 degrees
    /*int size2 = 3;//No. of vertices
        double points_list2[][2] = {{100, 100}, {100, 200},
                                    {200, 200}};
        rotate(points_list2, size2, 50, -50, -45);*/
  }
}
```

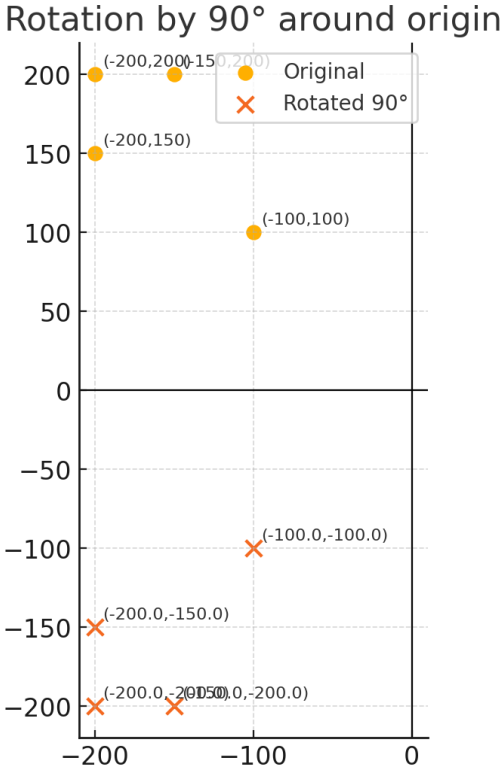
Output:

```
(-100, 100), (-200, 150), (-200, 200), (-150, 200)
```

2D Transformation Graph




Rotation by 45° around origin
Original
Rotated 45°
(-200,200)
(-150,200)
(-200,150)
(-100,100)
(-247.5,35.4)
(-282.8,0.0)
(-141.4,0.0)
(-247.5,-35.4)
200
150
100
50
0
−250
−200
−150
−100
−50
0


Rotation by 90° around origin
Original
Rotated 90°
(-200,200)
(-200,150)
(-100,100)
(-100.0,-100.0)
(-200.0,-150.0)
(-200.0,-200.0)
200
150
100
50
0
−50
−100
−150
−200
−200
−100
0

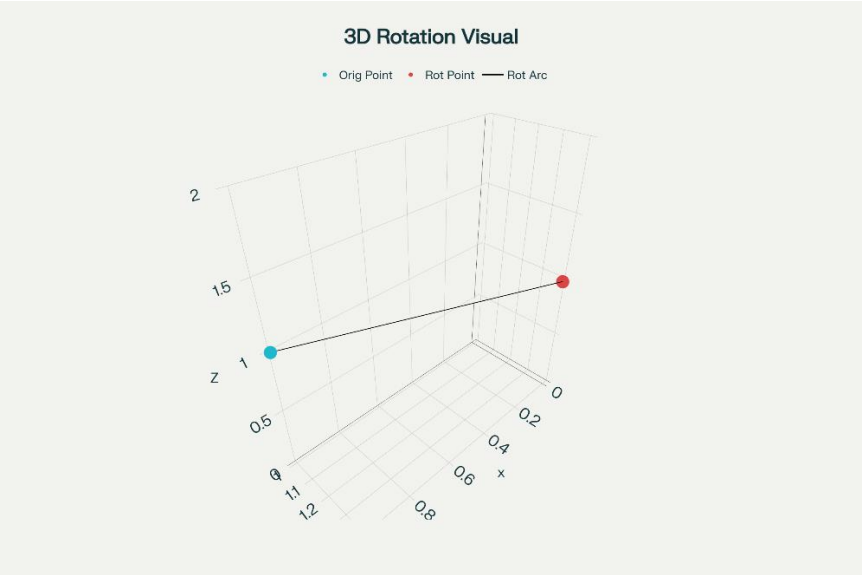
Ex no 4 b). Programs on 3D transformations

```
public class Simple3DRotation {
   public static void main(String[] args) {
      // Point coordinates
      double x = 1, y = 1, z = 1;
      // Rotation angle in degrees
      double theta = 45;
      // Convert to radians
      double radians = Math.toRadians(theta);

      // Rotation around Z-axis
      double[][] zRotation = {
         {Math.cos(radians), -Math.sin(radians), 0},
         {Math.sin(radians),  Math.cos(radians), 0},
         {0,                  0,                 1}
      };

      // Multiply matrix and vector
      double newX = zRotation[0][0]*x + zRotation[0][1]*y + zRotation[0][2]*z;
      double newY = zRotation[1][0]*x + zRotation[1][1]*y + zRotation[1][2]*z;
      double newZ = zRotation[2][0]*x + zRotation[2][1]*y + zRotation[2][2]*z;

      System.out.printf("Original point: (%.2f, %.2f, %.2f)%n", x, y, z);
      System.out.printf("Rotated point around Z by %.0f degrees: (%.2f, %.2f, %.2f)%n", theta,
newX, newY, newZ);
   }
}
```

Output:

Original point: (1.00, 1.00, 1.00)
Rotated point around Z by 45 degrees: (0.00, 1.41, 1.00)

3D Transformation Graph



Ex no 5. Write a program to implement Cohen Sutherland line clipping algorithm

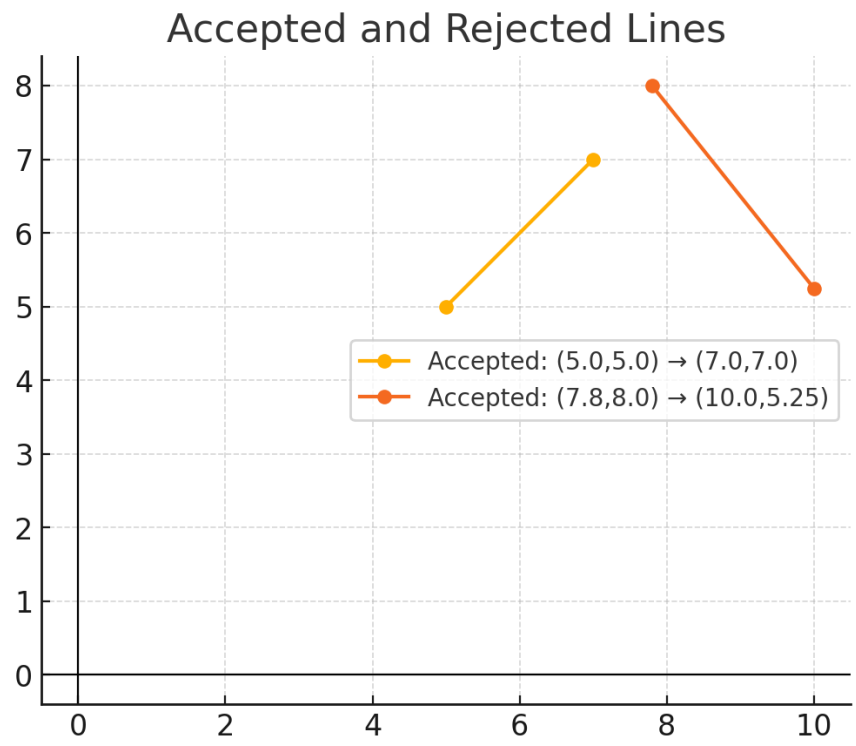
```
// Java program to implement Cohen Sutherland algorithm
// for line clipping.
import java.io.*;

class GFG {

    // Defining region codes
    static final int INSIDE = 0; // 0000
    static final int LEFT = 1; // 0001
    static final int RIGHT = 2; // 0010
    static final int BOTTOM = 4; // 0100
    static final int TOP = 8; // 1000

    // Defining x_max, y_max and x_min, y_min for
    // clipping rectangle. Since diagonal points are
    // enough to define a rectangle
    static final int x_max = 10;
    static final int y_max = 8;
    static final int x_min = 4;
    static final int y_min = 4;

    // Function to compute region code for a point(x, y)
    static int computeCode(double x, double y)
    {
        // initialized as being inside
        int code = INSIDE;

        if (x < x_min) // to the left of rectangle
            code |= LEFT;
        else if (x > x_max) // to the right of rectangle
            code |= RIGHT;
        if (y < y_min) // below the rectangle
            code |= BOTTOM;
        else if (y > y_max) // above the rectangle
            code |= TOP;

        return code;
    }

    // Implementing Cohen-Sutherland algorithm
    // Clipping a line from P1 = (x2, y2) to P2 = (x2, y2)
```

```
static void cohenSutherlandClip(double x1, double y1,
                                double x2, double y2)
{
    // Compute region codes for P1, P2
    int code1 = computeCode(x1, y1);
    int code2 = computeCode(x2, y2);

    // Initialize line as outside the rectangular window
    boolean accept = false;

    while (true) {
        if ((code1 == 0) && (code2 == 0)) {
            // If both endpoints lie within rectangle
            accept = true;
            break;
        }
        else if ((code1 & code2) != 0) {
            // If both endpoints are outside rectangle,
            // in same region
            break;
        }
        else {
            // Some segment of line lies within the
            // rectangle
            int code_out;
            double x = 0, y = 0;

            // At least one endpoint is outside the
            // rectangle, pick it.
            if (code1 != 0)
                code_out = code1;
            else
                code_out = code2;

            // Find intersection point;
            // using formulas y = y1 + slope * (x - x1),
            // x = x1 + (1 / slope) * (y - y1)
            if ((code_out & TOP) != 0) {
                // point is above the clip rectangle
                x = x1
                    + (x2 - x1) * (y_max - y1)
                          / (y2 - y1);
                y = y_max;
            }
            else if ((code_out & BOTTOM) != 0) {
                // point is below the rectangle
```

```
            x = x1
                + (x2 - x1) * (y_min - y1)
                    / (y2 - y1);
            y = y_min;
        }
        else if ((code_out & RIGHT) != 0) {
            // point is to the right of rectangle
            y = y1
                + (y2 - y1) * (x_max - x1)
                    / (x2 - x1);
            x = x_max;
        }
        else if ((code_out & LEFT) != 0) {
            // point is to the left of rectangle
            y = y1
                + (y2 - y1) * (x_min - x1)
                    / (x2 - x1);
            x = x_min;
        }

        // Now intersection point x, y is found
        // We replace point outside rectangle
        // by intersection point
        if (code_out == code1) {
            x1 = x;
            y1 = y;
            code1 = computeCode(x1, y1);
        }
        else {
            x2 = x;
            y2 = y;
            code2 = computeCode(x2, y2);
        }
      }
    }
    if (accept) {
        System.out.println("Line accepted from " + x1
                    + ", " + y1 + " to " + x2
                    + ", " + y2);
        // Here the user can add code to display the
        // rectangle along with the accepted (portion
        // of) lines
    }
    else
        System.out.println("Line rejected");
  }
```

```
  public static void main(String[] args)
  {
     // First Line segment
     // P11 = (5, 5), P12 = (7, 7)
     cohenSutherlandClip(5, 5, 7, 7);

     // Second Line segment
     // P21 = (7, 9), P22 = (11, 4)
     cohenSutherlandClip(7, 9, 11, 4);

     // Third Line segment
     // P31 = (1, 5), P32 = (4, 1)
     cohenSutherlandClip(1, 5, 4, 1);
  }
}
```

Graph line Clipping



Ex no 6. Write a program to draw Bezier curve.

```
public class BezierPoints {
   // Structure to hold a point
   static class Point {
      double x, y;
      Point(double x, double y) { this.x = x; this.y = y; }
   }

   // Calculate a point on cubic Bezier curve for parameter t [0..1]
   public static Point cubicBezier(double t, Point p0, Point p1, Point p2, Point p3) {
      double x = Math.pow(1 - t, 3) * p0.x +
               3 * t * Math.pow(1 - t, 2) * p1.x +
               3 * Math.pow(t, 2) * (1 - t) * p2.x +
               Math.pow(t, 3) * p3.x;

      double y = Math.pow(1 - t, 3) * p0.y +
               3 * t * Math.pow(1 - t, 2) * p1.y +
               3 * Math.pow(t, 2) * (1 - t) * p2.y +
               Math.pow(t, 3) * p3.y;

      return new Point(x, y);
   }

   public static void main(String[] args) {
      // Define control points
      Point p0 = new Point(100, 500);
      Point p1 = new Point(150, 100);
      Point p2 = new Point(650, 100);
      Point p3 = new Point(700, 500);

      System.out.println("Bezier Curve Points:");
      for (double t = 0; t <= 1.0; t += 0.05) {
         Point pt = cubicBezier(t, p0, p1, p2, p3);
         System.out.printf("t=%.2f: (%.2f, %.2f)%n", t, pt.x, pt.y);
      }
   }
}
```

Output:

Bezier Curve Points:
t=0.00: (100.00, 500.00)
t=0.05: (110.76, 443.00)
t=0.10: (127.60, 392.00)
t=0.15: (149.84, 347.00)
t=0.20: (176.80, 308.00)
t=0.25: (207.81, 275.00)
t=0.30: (242.20, 248.00)
t=0.35: (279.29, 227.00)
t=0.40: (318.40, 212.00)
t=0.45: (358.86, 203.00)
t=0.50: (400.00, 200.00)
t=0.55: (441.14, 203.00)
t=0.60: (481.60, 212.00)
t=0.65: (520.71, 227.00)
t=0.70: (557.80, 248.00)
t=0.75: (592.19, 275.00)
t=0.80: (623.20, 308.00)
t=0.85: (650.16, 347.00)
t=0.90: (672.40, 392.00)
t=0.95: (689.24, 443.00)

Bezier curve Graph

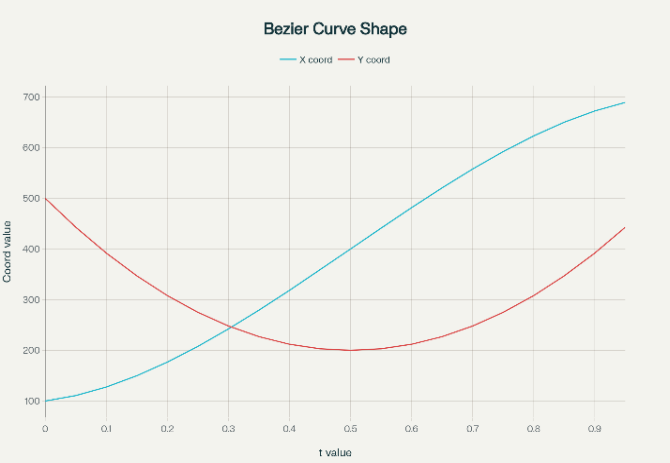


IMAGE EDITING SOFTWARE TOOL-GIMP3.0

7)Using Image Editing Software Tool-GIMP3.0 perform different operations (rotation, scaling move etc..) on objects

GIMP 3.0 basic drawing operations include using brush tools (like Pencil, Paintbrush, Airbrush), drawing paths for shapes, and managing colors and layers. To draw straight lines, you can use a brush tool, click a starting point on the canvas, hold Shift, drag to the desired endpoint, then click again to finish the line. For shapes, the Paths tool is useful: create anchor points linking them into paths, adjust curves as needed, then fill or stroke the path for outlines and color. You can select and customize brushes from the Brushes dialog and adjust colors for foreground/background before drawing. Text can also be added as independent layers and converted to paths for further editing.

GIMP 3.0 provides multiple transform tools to perform operations like **rotation**, **scaling**, and **moving** on objects, including layers, selections, and paths.

Rotation

- Use the Rotate Tool from the toolbox or by pressing Shift+R to rotate layers, selections, or paths.
- After making a selection or activating a layer, click within it using the Rotate Tool to open a rotation dialog, then adjust the angle and apply the transformation.
- The Unified Transform Tool also lets you combine rotate, scale, shear, and perspective actions in a single operation.

Scaling

- Activate the Scale Tool using Shift+S, the toolbox icon, or transform menu.
- Click on the layer, selection, or path to open scaling handles; drag these to resize as needed.
- The Scale dialog allows precise numerical adjustment and toggling of aspect ratio with the chain icon.
- Scaling can be performed on a selection's pixels, a layer, or the whole image, **Move**
- Select the Move Tool by pressing M or from the toolbox.
- You can move the active layer, a selection outline, or a path depending on the current mode.
- For layers: hold Ctrl+Alt or select layers via Ctrl+Click or Shift+Click in GIMP 3.0 to move multiple layers together.
- The Move tool allows precise position changes, snapping to guides or grids for easier alignment.

- To move selections, click and drag on the selection outline, or use arrow keys for pixel-level movement.

Unified Transform Tool

- The Unified Transform Tool combines rotate, scale, shear, and perspective into one interface for complex transformations.

GIMP 3.0's transform features allow efficient manipulation of objects for graphics editing and compositional flexibility.

8)To perform animation using any Animation software (Create Frame by Frame Animations using multimedia authoring tools-GIMP3.0)

GIMP 3.0 supports basic animation through frame-by-frame drawing using layers. The key concept is creating each frame of the animation as an individual layer, where you draw the elements for that frame. When played in sequence, these layers act as frames in the animation.

Basic Drawing Operations for Animation in GIMP 3.0:

- Create a new image with transparency enabled.
- Use separate layers for each animation frame. Each layer represents a single frame.
- Draw or paint your content on the first layer (frame).
- Duplicate the layer to create additional frames, modifying the drawing progressively to show action or movement.
- Use brush tools, pencil, eraser, and other drawing tools on each layer to create frame-by-frame changes.
- Adjust layer visibility to manage which frames are visible or hidden.
- Preview the animation via Filters → Animation → Playback to see the frames in motion.
- Use Filters → Animation → Optimize (for GIF) to reduce file size when exporting.
- Export the animation as a GIF file with animation and loop options enabled.

Basic Drawing Operations for Animation in GIMP 3.0 with Examples

Basic drawing operations for animation in GIMP 3.0 involve creating frame-by-frame animations using layers as frames. Here are the fundamental steps illustrated with examples:

1. Create New Image with Transparency

- Open GIMP and create a new file.
- Set the background to transparent for easier animation work.

2. Use Layers as Frames

- Each layer will act as a frame in the animation.
- Example: Create 3 layers named Frame 1, Frame 2, Frame 3.

3. Draw on the First Frame

- Select the Paintbrush tool.
- Example: Draw a simple circle in layer "Frame 1".

4. Duplicate the Frame for Next Step

- Right-click "Frame 1" layer → Duplicate Layer.
- Rename duplicated layer to "Frame 2".
- Make changes on Frame 2 layer for animation.
- Example: Move the circle slightly to the right.

5. Continue Frame Modifications

- Duplicate again for Frame 3.
- Example: Draw the circle further right to simulate movement.

6. Preview Animation

- Go to Filters → Animation → Playback.
- This shows layers as frames, animating your drawings.

7. Export as Animated GIF

- Export → Select GIF.
- Check “As animation” and set frame delay (e.g., 100ms per frame).
- Save the file.

Example Use Case

- Drawing a ball moving across the screen:
    - Frame 1: Ball at left side.
    - Frame 2: Ball moves slightly right.
    - Frame 3: Ball moves further right.
- Paintbrush tool is used to draw the ball on each layer with incremental position changes.
- Playback animates the frames giving the effect of motion.

Tools Used:

- Paintbrush/Pencil for drawing.
- Move tool to adjust objects on each frame.
- Eraser for corrections.
- Layer dialog for frame management.

9. To perform basic operations on image using any image editing software-GIMP3.0

To perform basic drawing operations in GIMP 3.0, select a paint tool like the Paintbrush or Pencil from the toolbox, then choose your desired brush settings and color. Click and drag on the canvas to draw, or hold Shift to draw straight lines. You can create shapes like circles using the Ellipse Select tool, then fill them with color using the Fill with foreground color tool.

1. Open GIMP and Create a New Image

1. Go to File > New from the main menu to create a new composition.
2. Define your image's dimensions, resolution, and background color.

2. Select a Drawing Tool

1. In the Toolbox on the left, select a paint tool.
    - **Paintbrush Tool:** Provides a balanced stroke.
    - **Pencil Tool:** Offers a hard, pixelated stroke.
    - **Airbrush Tool:** Creates a softer, more diffuse stroke.
2. To see more options or related tools, right-click on a tool icon to expand it.

3. Set Up Your Tools

1. **1. Tool Options:**

The Tool Options dialog allows you to adjust brush settings like size, opacity, angle, and hardness.

2. **2. Colors:**

Select your desired foreground and background colors from the color selection area on the toolbox.

4. Perform Drawing Operations

- **Freehand Drawing:**

Click and drag on the canvas with your chosen paint tool to draw a freehand stroke.

- **Straight Lines:**

Hold down the Shift key, click to set a starting point, and then click again at your desired endpoint to draw a straight line.

- **Drawing Circles and Ellipses:**
    1. Select the Ellipse Select tool from the toolbox.
    2. Click and drag on the canvas to draw an ellipse.

3. Hold Shift to draw a perfect circle.
4. With the circle selected, use the Fill with Foreground Color tool to fill it with your chosen color.

5. Working with Layers

- **New Layer:**

When you create a new image, you start with a background layer. You can add new, transparent layers using the Layers panel to draw on without affecting other parts of your image.

- **Save Your Work:**

Go to File > Save or File > Save As to save your project in GIMP's native format (XCF), which preserves layers. Use File > Export As to save in formats like JPEG or PNG.

Basic Drawing Steps in GIMP 3.0

- Open a new image canvas.
- Select a brush tool (Pencil, Paintbrush, etc.) from the Toolbox.
- Choose a foreground color (different from background).
- Click on the canvas, hold Shift, drag to draw straight lines.
- Use the Paths tool to create complex shapes with anchor points and curves.
- Fill or stroke paths to color shapes.
- Add text using the Text tool; texts are placed as separate layers.
- Undo mistakes with Ctrl+Z (Cmd+Z on Mac).

Drawing Shapes and Paths

- Use the Paths tool to create shapes by clicking to add points.
- Close the path by connecting back to the first point (Ctrl+click).
- Curve segments by dragging handles on path points.
- Fill shapes or stroke outlines from the Paths dialog options.
- Manage vectors separately from bitmap layers.

Brushes and Tools

- Use different brushes from the Brushes dialog to get varied effects.
- Customize brush size, angle, and shape.
- Use Pencil for hard edges, Paintbrush for softer strokes.
- Use Eraser, Clone, Smudge, and other brush tools for various effects.

These operations provide a foundation for basic drawing in GIMP 3.0, enabling freehand sketches, geometric shapes, and editable text with custom strokes and fills.