INTRODUCTION TO COMPUTER GRAPHICS

Computer graphics is the field of computing that deals with generating images, animations, and visual representations of data. It is widely used in various fields such as entertainment, education, engineering, medical imaging, and scientific research.

Computer graphics can be classified into two types:

1. **Interactive Graphics** – Allows real-time user interaction, e.g., video games, simulations, and virtual reality.
2. **Non-Interactive Graphics** – Generates images without real-time user interaction, e.g., 3D rendering, animated movies, and visual effects.

1.1 APPLICATIONS OF COMPUTER GRAPHICS

Computer graphics finds applications in various domains, including:

- **Entertainment**: Used in video games, animated movies, and special effects in films.
- **Education and Training**: Simulations for medical, military, and aviation training.
- **Engineering and Design**: CAD (Computer-Aided Design) for architectural and mechanical designs.
- **Medical Imaging**: CT scans, MRI, and 3D reconstructions of organs.
- **Scientific Visualization**: Graphical representation of complex data in research.
- **Virtual and Augmented Reality**: Creating immersive environments for simulations and real-world interactions.

1.2 DIFFERENT I/O DEVICES IN COMPUTER GRAPHICS

1.2.1 INPUT DEVICES

Input devices enable users to interact with computer graphics by providing data and control signals. Common input devices include **graphics tablets** for digital drawing, **joysticks** for gaming and simulations, **trackballs** for precise control in CAD applications, and **light pens** for direct screen interaction. **Touch screens** offer intuitive control, while **scanners** digitize physical documents. **Mice and keyboards** are standard tools for graphic design, and **motion capture devices** are used in animation and game development to record real-world movements.

1.2.2 OUTPUT DEVICES

Output devices display, print, or render graphical content. Traditional **CRT monitors** have been replaced by **LCD and LED displays** for higher resolution and better color accuracy. **Printers** like **plotters** and **laser printers** are used for vector and high-quality image printing. **Projectors** display visuals on large screens, while **3D printers** create physical models from digital designs. **Head-mounted displays (HMDs)**, such as VR headsets, provide immersive experiences in virtual and augmented reality.

1.3 ELEMENTS OF GRAPHICS

The essential elements in graphics include:

- **Pixels**: Smallest unit of a digital image.
- **Resolution**: Number of pixels in an image, affecting clarity.
- **Color Models**: RGB, CMY, HSV used for color representation.
- **Rendering**: Process of generating images from models.
- **Transformation**: Operations like scaling, rotation, and translation of objects.

1.4 GRAPHIC SYSTEMS

A **graphic system** consists of hardware and software components used to generate, manipulate, and display graphical content. The key components include:

- **Graphics Processing Unit (GPU)** – A specialized processor for rendering images and animations.
- **Display Devices** – Such as **LCD, LED, and OLED monitors** for visual output.
- **Input Devices** – Tools like **graphic tablets, joysticks, and touchscreens** for user interaction.
- **Software & APIs** – Graphics libraries such as **OpenGL, DirectX, and WebGL** that facilitate rendering and processing.

1.5 VIDEO DISPLAY DEVICES

Video display devices are used to visualize graphics and images generated by a computer. These include:

- **CRT (Cathode Ray Tube) Monitors** – Used in early computer graphics, now obsolete.
- **LCD (Liquid Crystal Display) and LED Displays** – Modern screens offering higher resolution and better color accuracy.

- **Projectors** – Used for large-screen displays in presentations and cinemas.
- **Head-Mounted Displays (HMDs)** – Such as VR headsets for immersive experiences.

1.6 SCAN SYSTEMS

1.6.1 RASTER SCAN SYSTEMS

In **raster scan systems**, the screen is divided into tiny pixels, and images are formed by illuminating these pixels line by line from top to bottom. This method is widely used in modern monitors and TVs. **Advantages** include smooth shading and support for high-resolution images. **Disadvantages** include higher memory requirements and potential flickering in low-refresh-rate screens.

1.6.2 RANDOM SCAN SYSTEMS

In **random scan systems** (also known as vector displays), images are drawn directly using lines instead of pixels. The system moves the electron beam to specific points on the screen to create images. **Advantages** include sharp line drawings and reduced memory usage. However, they are less effective for rendering complex scenes and were mainly used in early CAD applications.

1.7 VIDEO BASICS

- **Video Controller** – A dedicated hardware component that controls how images are displayed on the screen. It manages the transfer of image data from the computer's memory to the display device.
- **Raster-Scan Display Processor** – A processor specifically designed to handle raster graphics. It assists in rendering images efficiently by managing pixel-based operations, including color mapping, shading, and anti-aliasing.

PRACTICAL NO. 1

AIM: Write a program for creating a simple two-dimensional shape of any objects using lines, circle, etc.

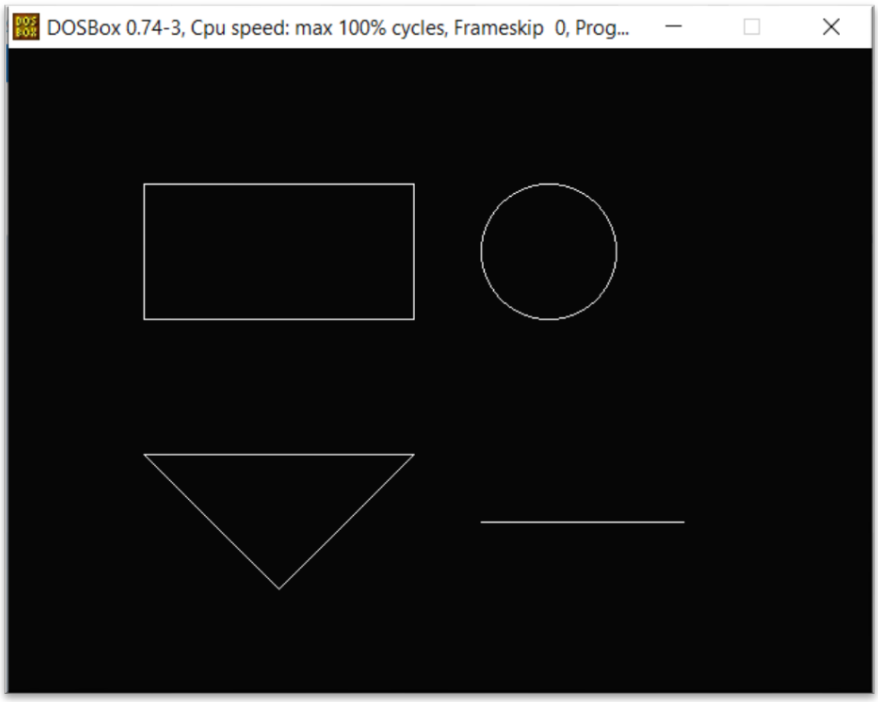
ALGORITHM:

1. Start the program.
2. Include required header files (graphics.h and conio.h).
3. Initialize graphics mode:
    - Declare variables gd and gm for graphics.
    - Use DETECT to automatically detect the graphics driver.
    - Call initgraph(&gd, &gm, "C:\\TurboC3\\BGI") to start graphics mode.

4. Draw shapes:
    - Rectangle: Use rectangle(100, 100, 300, 200).
    - Circle: Use circle(400, 150, 50).
    - Line**:** Use line(500, 350, 350, 350).

Triangle: Draw three lines to form a triangle.
5. Wait for user input using getch(); to keep the output visible.
6. Close graphics mode using closegraph();.
7. End the program

OUTPUT:



PRACTICAL NO. 1

AIM: Write a program for creating a simple two-dimensional shape of any objects using lines, circle, etc.

```
#include <graphics>

#include<conio.h>


int main()

int gd = DETECT, gm;

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


// Draw a rectangle

rectangle(100, 100, 300, 200);


// Draw a circle

circle(400, 150, 50);


//Draw a line

line(500, 350, 350, 350):


// Draw a triangle

line(100, 300, 200, 400);

line(200, 400, 300, 300);

line(300, 300, 100, 300);

getch();

closegraph():

return 0;

}
```

PRACTICAL NO. 2

AIM: Write a program to Draw a color cube and spin it using transformation matrices.

ALGORITHM:

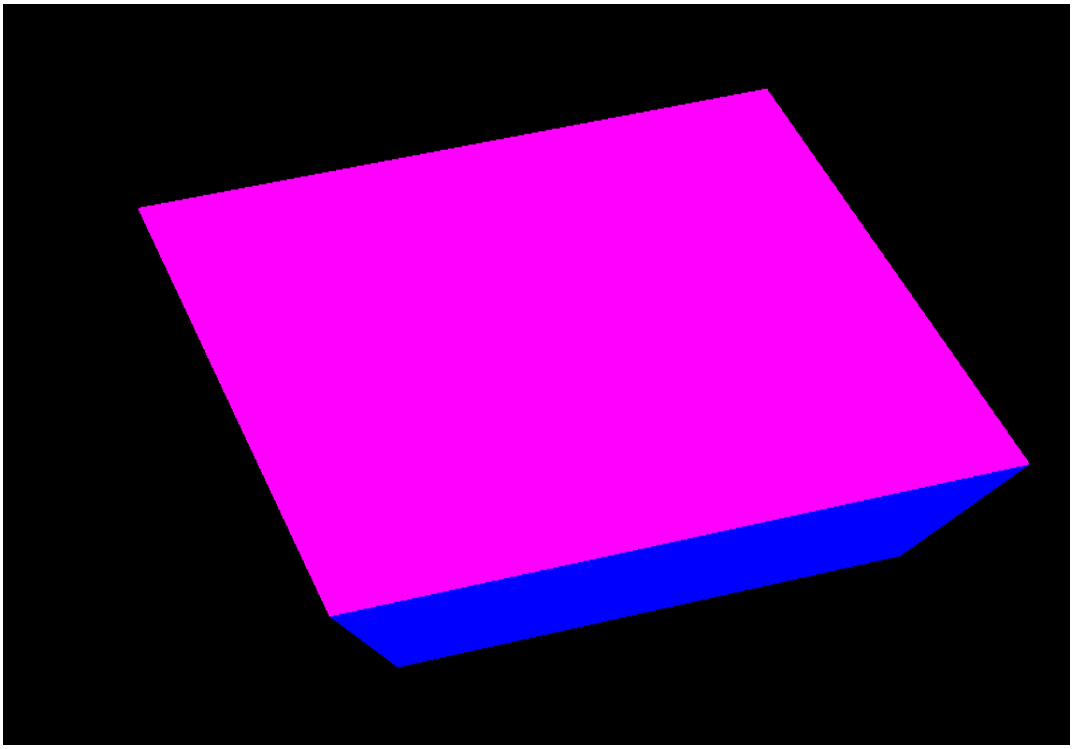
1. Initialize OpenGL and set up the display mode.
2. Create a window with a specified size and title.
3. Enable depth testing for proper 3D rendering.
4. Set the projection matrix using a perspective view.
5. Define a function to draw a colored cube using glBegin(GL_QUADS).
6. Define a function to display the scene, clear buffers, apply transformations, and render the cube.
7. Define an animation update function to rotate the cube and refresh the display.
8. Set a timer function to update animation at regular intervals.
9. Set up display and timer functions, then start the OpenGL main loop.

PRACTICAL NO. 2

AIM: Write a program to Draw a color cube and spin it using transformation matrices.

```
#include <GL/glut.h>
#include <iostream>
float angle = 0.0f;
void displayText(float x, float y, const char* text) {
    glRasterPos2f(x, y);
    for (const char* c = text; *c != '\0'; c++) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, *c);
    }
}
void drawCube() {
    glBegin(GL_QUADS);
    // Front face (red)
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(-0.5, -0.5, 0.5);
    glVertex3f(0.5, -0.5, 0.5);
    glVertex3f(0.5, 0.5, 0.5);
    glVertex3f(-0.5, 0.5, 0.5);

    // Back face (green)
    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(-0.5, -0.5, -0.5);
    glVertex3f(-0.5, 0.5, -0.5);
```

OUTPUT:



```
glVertex3f(0.5, 0.5, -0.5);
glVertex3f(0.5, -0.5, -0.5);

// Left face (blue)
glColor3f(0.0, 0.0, 1.0);
glVertex3f(-0.5, -0.5, -0.5);
glVertex3f(-0.5, -0.5, 0.5);
glVertex3f(-0.5, 0.5, 0.5);
glVertex3f(-0.5, 0.5, -0.5);

// Right face (yellow)
glColor3f(1.0, 1.0, 0.0);
glVertex3f(0.5, -0.5, -0.5);
glVertex3f(0.5, 0.5, -0.5);
glVertex3f(0.5, 0.5, 0.5);
glVertex3f(0.5, -0.5, 0.5);

// Top face (cyan)
glColor3f(0.0, 1.0, 1.0);
glVertex3f(-0.5, 0.5, -0.5);
glVertex3f(-0.5, 0.5, 0.5);
glVertex3f(0.5, 0.5, 0.5);
glVertex3f(0.5, 0.5, -0.5);

// Bottom face (magenta)
```

```
glColor3f(1.0, 0.0, 1.0);
    glVertex3f(-0.5, -0.5, -0.5);
    glVertex3f(0.5, -0.5, -0.5);
    glVertex3f(0.5, -0.5, 0.5);
    glVertex3f(-0.5, -0.5, 0.5);

    glEnd();
}
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.0);
    glRotatef(angle, 1.0, 1.0, 1.0);
    drawCube();
    glLoadIdentity();
    glColor3f(1.0, 1.0, 1.0);
    glutSwapBuffers();
}
void update(int value) {
 angle += 1.0f;
    if (angle > 360) {
        angle -= 360;
    }
    glutPostRedisplay();
    glutTimerFunc(16, update, 0);
```

```
}
void init() {
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, 1.0, 1.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("2203443: Gursimran Kaur");
    init();
    glutDisplayFunc(display);
    glutTimerFunc(16, update, 0);
    glutMainLoop();
     return 0;
}
```

PRACTICAL NO. 3

AIM: Implement the DDA algorithm for drawing line (programmer is expected to shift the origin to the center of the screen and divide the screen into required quadrants).

ALGORITHM:

1. Start
2. Input the starting coordinates (x1, y1) and ending coordinates (x2, y2)
3. Compute the difference:

   - dx = x2 - x1
   - dy = y2 - y1

4. Determine the number of steps required:

   - steps = max(abs(dx), abs(dy))

5. Compute the increments for each step:

   - xInc = dx / steps
   - yInc = dy / steps

6. Initialize starting point:

   - x = x1, y = y1

7. Repeat for steps + 1 times:

   - Plot the pixel at (round(x), round(y))
   - Update x = x + xInc
   - Update y = y + yInc

8. End

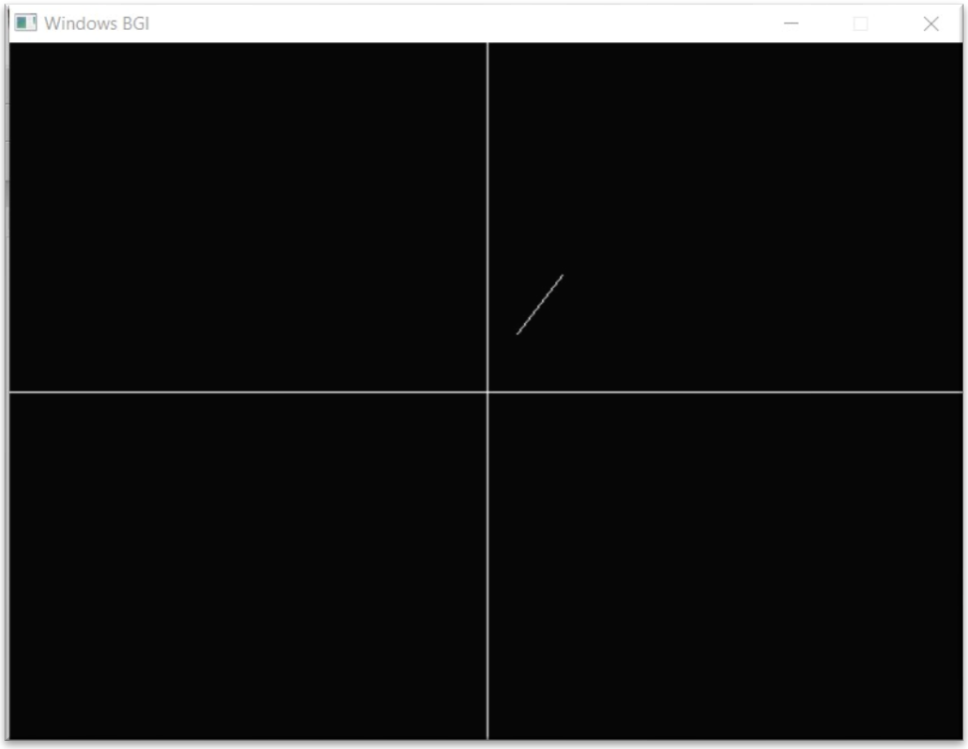
PRACTICAL NO. 3

AIM: Implement the DDA algorithm for drawing line (programmer is expected to shift the origin to the center of the screen and divide the screen into required quadrants).

```
#include <iostream>
#include <graphics.h>
#include <cmath>
using namespace std;
// Function to convert Cartesian coordinates to screen coordinates
void plotPoint(int x, int y) {
    int xc = getmaxx() / 2; // Center X
    int yc = getmaxy() / 2; // Center Y
    putpixel(xc + x, yc - y, WHITE); // Convert (x,y) to screen coordinates
}
// DDA Line Drawing Algorithm (Origin Shifted to Center)
void drawLineDDA(int x1, int y1, int x2, int y2) {
    float dx = x2 - x1;
    float dy = y2 - y1;
    float steps = max(abs(dx), abs(dy)); // Select greater value for iteration
    float xInc = dx / steps; // Increment in x
    float yInc = dy / steps; // Increment in y
    float x = x1, y = y1;
    for (int i = 0; i <= steps; i++) {
        plotPoint(round(x), round(y)); // Plot the calculated point
        x += xInc;
        y += yInc;
```

OUTPUT:

```
Line Drawing using DDA Algorithm

Enter the starting coordinates (x1, y1): 20 40

Enter the ending coordinates (x2, y2): 50 80
```



```
    }
}
int main() {
    int x1, y1, x2, y2;
    cout<< "\n Line Drawing using DDA Algorithm";
    cout << "\n Enter the starting coordinates (x1, y1): ";
    cin >> x1 >> y1;
    cout << "\n Enter the ending coordinates (x2, y2): ";
    cin >> x2 >> y2;

    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    // Draw X and Y axes for quadrant division
    int xc = getmaxx() / 2;
    int yc = getmaxy() / 2;
    line(xc, 0, xc, getmaxy()); // Y-Axis
    line(0, yc, getmaxx(), yc); // X-Axis
    // Draw the line with DDA Algorithm (Origin shifted)
    drawLineDDA(x1, y1, x2, y2);

    getch(); // Wait for user input
    closegraph();
    return 0;
}
```

PRACTICAL NO. 4

AIM: Write a program to input the line coordinates from the user to generate a line using Bresenham's Algorithm.

ALGORITHM:

1. Start the program and include necessary header files.
2. Define the drawLine() function using Bresenham's Line Algorithm.
3. Calculate the absolute difference between x and y coordinates.
4. Determine the step direction for x and y based on the start and end points.
5. Initialize the error term for decision-making in pixel plotting.
6. Use a loop to iterate through the line points until the end point is reached.
7. Plot each pixel using putpixel().
8. Display the coordinates near the start and end points using outtextxy().
9. Adjust the error term and update x and y coordinates accordingly.
10. In the main() function, take user input for start and end coordinates.
11. Initialize the graphics mode using initgraph().
12. Call the drawLine() function to draw the line.
13. Pause execution using getch() to display the result.
14. Close the graphics mode using closegraph().
15. End the program.

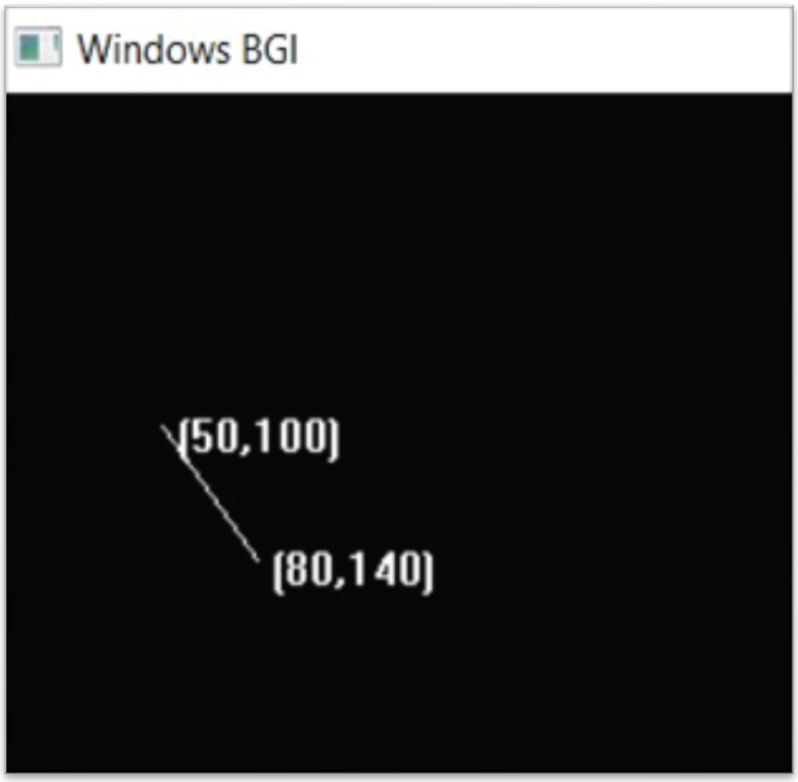
PRACTICAL NO. 4

AIM: Write a program to input the line coordinates from the user to generate a line using Bresenham's Algorithm.

```
#include <iostream>

#include <graphics.h>

#include <string>

using namespace std;

void drawLine(int x1, int y1, int x2, int y2) {

   int dx = abs(x2 - x1), dy = abs(y2 - y1);

   int sx = (x1 < x2) ? 1 : -1;

   int sy = (y1 < y2) ? 1 : -1;

   int err = dx - dy;

   while (true) {

      putpixel(x1, y1, WHITE);  // Plot the pixel



      // Display coordinates near the start and end points

      if ((x1 == x2 && y1 == y2) || (x1 == x1 && y1 == y1)) {

         string coordText = "(" + to_string(x1) + "," + to_string(y1) + ")";

         outtextxy(x1 + 5, y1 - 5, &coordText[0]);

      }

      if (x1 == x2 && y1 == y2)

         break;

          int e2 = 2 * err;

      if (e2 > -dy) { err -= dy; x1 += sx; }

      if (e2 < dx) { err += dx; y1 += sy; }
```

OUTPUT:

```
Enter the starting coordinates (x1, y1): 50 100

Enter the ending coordinates (x2, y2): 80 140
```



```
    }}
int main() {
    int x1, y1, x2, y2;
    cout<< "\n Line Drawing using Bresenham's Algorithm";
    cout << "Enter the starting coordinates (x1, y1): ";
    cin >> x1 >> y1;
    cout << "Enter the ending coordinates (x2, y2): ";
    cin >> x2 >> y2;
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    drawLine(x1, y1, x2, y2);
    // Pause to see the output
    getch();
    closegraph();
    return 0;
}
```

PRACTICAL NO. 5

AIM: Write a program to generate a complete moving wheel using Midpoint circle drawing algorithm and DDA line drawing algorithm.

ALGORITHM:

1. Initialize graphics mode using initgraph().
2. Define the drawCircle() function using the Midpoint Circle Algorithm.
3. Define the drawLineDDA() function using the DDA Algorithm.
4. Define the drawWheel() function to draw a circle and spokes.
5. In the main() function, set up graphics mode and define wheel properties.
6. Use a loop to animate the wheel moving across the screen.
7. Clear the screen in each iteration for smooth animation.
8. Display the name and URN using outtextxy().
9. Draw the ground as a horizontal line.
10. Draw the moving wheel using drawWheel().
11. Add a delay for smooth animation.
12. Wait for user input using getch().
13. Close the graphics mode using closegraph().

PRACTICAL NO. 5

AIM: Write a program to generate a complete moving wheel using Midpoint circle drawing algorithm and DDA line drawing algorithm.

```
#include <iostream>

#include <graphics.h>

#include <cmath>

#include <dos.h>

using namespace std;

// Function to draw a circle using Midpoint Circle Algorithm

void drawCircle(int xc, int yc, int r) {

    int x = 0, y = r;

    int p = 1 - r;


    while (x <= y) {

        putpixel(xc + x, yc + y, WHITE);

        putpixel(xc - x, yc + y, WHITE);

        putpixel(xc + x, yc - y, WHITE);

        putpixel(xc - x, yc - y, WHITE);

        putpixel(xc + y, yc + x, WHITE);

        putpixel(xc - y, yc + x, WHITE);

        putpixel(xc + y, yc - x, WHITE);

        putpixel(xc - y, yc - x, WHITE);

         x++;

        if (p < 0)

            p += 2 * x + 1;
```

OUTPUT:

```
        else {
            y--;
            p += 2 * (x - y) + 1;
        }}}
// Function to draw a line using DDA Algorithm
void drawLineDDA(int x1, int y1, int x2, int y2) {
    float dx = x2 - x1;
    float dy = y2 - y1;
    float steps = max(abs(dx), abs(dy));
    float xInc = dx / steps;
    float yInc = dy / steps;
    float x = x1, y = y1;
    for (int i = 0; i <= steps; i++) {
        putpixel(round(x), round(y), WHITE);
        x += xInc;
        y += yInc;
    }}
// Function to draw the wheel with spokes
void drawWheel(int xc, int yc, int r) {
    drawCircle(xc, yc, r);
    // Draw spokes using DDA
    drawLineDDA(xc, yc - r, xc, yc + r); // Vertical spoke
    drawLineDDA(xc - r, yc, xc + r, yc); // Horizontal spoke
    drawLineDDA(xc - r / 1.4, yc - r / 1.4, xc + r / 1.4, yc + r / 1.4); // Diagonal 1
    drawLineDDA(xc - r / 1.4, yc + r / 1.4, xc + r / 1.4, yc - r / 1.4); // Diagonal 2
```

```
}
int main() {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    int r = 50; // Radius of the wheel
    int y = getmaxy() - r - 20; // Ground level position
    int speed = 5; // Movement speed

    for (int x = 50; x < getmaxx() - 50; x += speed) {
        cleardevice(); // Clear screen for animation

        // Display name on the screen
        setcolor(WHITE);
        settextstyle(SANS_SERIF_FONT, HORIZ_DIR, 2);

        // Draw ground
        line(0, y + r + 10, getmaxx(), y + r + 10);
         // Draw moving wheel
        drawWheel(x, y, r);
        delay(50); // Delay for smooth animation
    }
    getch();
    closegraph();
    return 0;
}
```