INDEX

Table of Contents

1. Experiment Name: Bresenham's Circle Drawing Algorithm Implementation........................2
Objectives:........................2
Theory:........................2
Algorithm:........................2
Source Code:........................2
2. Experiment Name: Bresenham's Line Drawing Algorithm Implementation........................4
Objectives:........................4
Theory:........................4
Algorithm:........................4
Source Code:........................4
3. Experiment Name: DDA (Digital Differential Analyzer) Line Drawing Algorithm Implementation........................6
Objectives:........................6
Theory:........................6
Algorithm:........................6
Source Code:........................6
4. Experiment Name: Midpoint Line Drawing Algorithm Implementation........................8
Objectives:........................8
Theory:........................8
Algorithm:........................8
Source Code:........................8
5. Experiment Name: Polygon (Rectangle) Filling Algorithm Implementation........................10
Objectives:........................10
Theory:........................10
Algorithm:........................10
Source Code:........................10
6. Experiment Name: Midpoint Ellipse Drawing Algorithm Implementation........................12
Objectives:........................12
Theory:........................12
Algorithm:........................12
Source Code:........................12
7. Experiment Name: 2D Transformation Implementation (Scaling, Rotation)........................14
Objectives:........................14
Theory:........................14
Algorithm:........................14
Source Code:........................14

1. Experiment Name: Bresenham's Circle Drawing Algorithm Implementation

Objectives:

- To understand the concept of Bresenham's Circle Drawing Algorithm
- To implement the algorithm in C programming language
- To draw a circle using the Bresenham's Circle Drawing Algorithm

Theory:

Bresenham's Circle Drawing Algorithm is an efficient algorithm to draw a circle in a raster graphics display. The algorithm uses integer arithmetic to avoid floating-point arithmetic and uses only addition, subtraction, multiplication and division operations. The algorithm starts at the origin of a coordinate system and proceeds by selecting a point on the circumference of the circle using decision parameters. The algorithm generates the circle by plotting symmetric points in each octant.

Algorithm:

1. Initialize the center and radius of the circle
2. Set the initial decision parameter value as p = 3 - 2 * radius
3. Initialize the values of x and y as x = 0 and y = radius
4. Repeat the following steps until x is greater than or equal to y
    - Plot the eight-way symmetric points on the circumference of the circle
    - If the decision parameter is less than 0, update its value as p = p + 4 * x + 6
    - Otherwise, update the decision parameter as p = p + 4 * (x - y) + 10 and decrement the value of y by 1
    - Increment the value of x by 1
5. Exit

Source Code:

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

void bresenham_circle(int x0, int y0, int radius)
{
    int x = 0, y = radius;
    int p = 3 - 2 * radius;

    while (x <= y)
    {
        // plot eight-way symmetric points
        putpixel(x0 + x, y0 + y, WHITE);
        putpixel(x0 - x, y0 + y, WHITE);
        putpixel(x0 + x, y0 - y, WHITE);
        putpixel(x0 - x, y0 - y, WHITE);
        putpixel(x0 + y, y0 + x, WHITE);
        putpixel(x0 - y, y0 + x, WHITE);
        putpixel(x0 + y, y0 - x, WHITE);
        putpixel(x0 - y, y0 - x, WHITE);

        // update the decision parameter
        if (p < 0)
        {
            p += 4 * x + 6;
        }
        else
        {
            p += 4 * (x - y) + 10;
            y--;
        }
        x++;
    }
}

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");

    int x0 = 250, y0 = 250, radius = 100;
    bresenham_circle(x0, y0, radius);

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

2. Experiment Name: Bresenham's Line Drawing Algorithm Implementation

Objectives:

- To understand the concept of Bresenham's Line Drawing Algorithm
- To implement the algorithm in C programming language
- To draw a line using the Bresenham's Line Drawing Algorithm

Theory:

Bresenham's Line Drawing Algorithm is an efficient algorithm to draw a line in a raster graphics display. The algorithm uses integer arithmetic to avoid floating-point arithmetic and uses only addition, subtraction, and multiplication operations. The algorithm starts at one endpoint of the line and proceeds by selecting the next pixel in the line using decision parameters. The algorithm generates the line by plotting symmetric points in each octant.

Algorithm:

1. Initialize the endpoints of the line
2. Calculate the differences between the endpoint coordinates dx and dy
3. Calculate the decision parameter as d = 2 * dy - dx
4. Initialize the values of x and y as the x-coordinate of the starting point and the y-coordinate of the starting point
5. Repeat the following steps until x is equal to the x-coordinate of the ending point
    - Plot the pixel at (x, y)
    - If the decision parameter is greater than or equal to 0, increment y and update the decision parameter as d = d + 2 * (dy - dx)
    - Otherwise, update the decision parameter as d = d + 2 * dy
    - Increment x by 1
6. Exit

Source Code:

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

void bresenham_line(int x0, int y0, int x1, int y1)
{
    int dx = abs(x1 - x0);
    int dy = abs(y1 - y0);
    int x = x0, y = y0;
    int d = 2 * dy - dx;
    int x_increment = x1 > x0 ? 1 : -1;
    int y_increment = y1 > y0 ? 1 : -1;

    while (x != x1)
    {
        // plot the pixel at (x, y)
        putpixel(x, y, WHITE);

        // update the decision parameter
        if (d >= 0)
        {
            y += y_increment;
            d += 2 * (dy - dx);
        }
        else
        {
            d += 2 * dy;
        }
        x += x_increment;
    }
}

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");

    int x0 = 100, y0 = 100, x1 = 300, y1 = 250;
    bresenham_line(x0, y0, x1, y1);

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

3. Experiment Name: DDA (Digital Differential Analyzer) Line Drawing Algorithm Implementation

Objectives:

- To understand the concept of DDA (Digital Differential Analyzer) Line Drawing Algorithm
- To implement the algorithm in C programming language
- To draw a line using the DDA Line Drawing Algorithm

Theory:

DDA (Digital Differential Analyzer) Line Drawing Algorithm is an incremental method for generating a straight line. It is based on the simple idea of interpolation between two points. The algorithm calculates the slope of the line, and then increments the coordinate with the maximum change (either x or y) by 1, and computes the other coordinate based on the slope. This process is repeated until the end point is reached.

Algorithm:

1. Initialize the endpoints of the line
2. Calculate the differences between the endpoint coordinates dx and dy
3. Calculate the slope of the line as m = dy / dx
4. Initialize the values of x and y as the x-coordinate of the starting point and the y-coordinate of the starting point
5. Repeat the following steps until x is equal to the x-coordinate of the ending point
    - Plot the pixel at (x, y)
    - Increment x by 1
    - Calculate the value of y as y = y + m
6. Exit

Source Code:

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

void dda_line(int x0, int y0, int x1, int y1)
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    float m = (float)dy / dx;
    float x = x0, y = y0;

    for (int i = 0; i < dx; i++)
    {
        // plot the pixel at (x, y)
        putpixel((int)x, (int)y, WHITE);

        // update the value of y
        y += m;
        x++;
    }
}

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");

    int x0 = 100, y0 = 100, x1 = 300, y1 = 250;
    dda_line(x0, y0, x1, y1);

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

4. Experiment Name: Midpoint Line Drawing Algorithm Implementation

Objectives:

- To understand the concept of Midpoint Line Drawing Algorithm
- To implement the algorithm in C programming language
- To draw a line using the Midpoint Line Drawing Algorithm

Theory:

The Midpoint Line Drawing Algorithm is a faster method of generating a straight line compared to the DDA algorithm. It uses the Bresenham's algorithm and avoids the use of floating point arithmetic, making it more efficient for line drawing.

The algorithm works by starting at one endpoint and incrementing the x coordinate while simultaneously incrementing the y coordinate by either 1 or 0 depending on the decision parameter (p) value. The decision parameter is calculated as the midpoint between two candidate pixels, which are the pixels that lie on the line and are closest to the theoretical line.

Algorithm:

1. Initialize the endpoints of the line
2. Calculate the differences between the endpoint coordinates dx and dy
3. Calculate the decision parameter as p = 2 * dy - dx
4. Initialize the values of x and y as the x-coordinate of the starting point and the y-coordinate of the starting point
5. Repeat the following steps until x is equal to the x-coordinate of the ending point
    - Plot the pixel at (x, y)
    - If p < 0, increment p by 2 * dy
    - Else, increment p by 2 * (dy - dx) and increment y by 1
    - Increment x by 1
6. Exit

Source Code:

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

void midpoint_line(int x0, int y0, int x1, int y1)
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    int x = x0, y = y0;
    int p = 2 * dy - dx;

    while (x <= x1)
    {
        // plot the pixel at (x, y)
        putpixel(x, y, WHITE);

        if (p < 0)
            p += 2 * dy;
        else
        {
            p += 2 * (dy - dx);
            y++;
        }

        x++;
    }
}

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");

    int x0 = 100, y0 = 100, x1 = 300, y1 = 250;
    midpoint_line(x0, y0, x1, y1);

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

5. Experiment Name: Polygon (Rectangle) Filling Algorithm Implementation

Objectives:

- To understand the concept of polygon filling algorithm
- To implement the algorithm in C programming language
- To fill a polygon (rectangle) using the polygon filling algorithm

Theory:

The Polygon Filling Algorithm is used to fill the interior of a polygon with a specified color. One of the commonly used polygon filling algorithms is the Scanline algorithm. It works by scanning the lines in the polygon and filling the pixels between the edges. The algorithm first finds the top and bottom points of the polygon, then scans each line between the top and bottom points to fill the polygon.

In this experiment, we will implement the polygon filling algorithm to fill a rectangle. We will divide the rectangle into horizontal lines, and then we will fill the pixels between the edges of the rectangle for each line.

Algorithm:

1. Set the color of the polygon
2. Divide the polygon into horizontal lines
3. For each horizontal line, find the intersection points with the edges of the polygon
4. Sort the intersection points by their x-coordinates
5. Fill the pixels between the intersection points for each horizontal line
6. Exit

Source Code:

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

void rectangle_fill(int x1, int y1, int x2, int y2)
{
    int x, y;
    int color = WHITE;

    // find the top and bottom points of the rectangle
    int top = y1 < y2 ? y1 : y2;
    int bottom = y1 > y2 ? y1 : y2;

    // divide the rectangle into horizontal lines
    for (y = top; y <= bottom; y++)
    {
        // find the intersection points with the edges of the rectangle
        int intersections[2] = {0, 0};
        int num_intersections = 0;

        if (y >= y1 && y <= y2)
        {
            intersections[num_intersections++] = x1 + (y - y1) * (x2 -
x1) / (y2 - y1);
        }

        if (y >= y2 && y <= y1)
        {
            intersections[num_intersections++] = x2 + (y - y2) * (x1 -
x2) / (y1 - y2);
        }

        // sort the intersection points by their x-coordinates
        if (num_intersections == 2 && intersections[0] > intersections[1])
        {
            int temp = intersections[0];
            intersections[0] = intersections[1];
            intersections[1] = temp;
        }

        // fill the pixels between the intersection points
        if (num_intersections == 2)
        {
            for (x = intersections[0]; x <= intersections[1]; x++)
            {
                putpixel(x, y, color);
            }
        }
    }
}

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");

    int x1 = 100, y1 = 100, x2 = 300, y2 = 250;
    rectangle_fill(x1, y1, x2, y2);

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

6. Experiment Name: Midpoint Ellipse Drawing Algorithm Implementation

Objectives:

- To understand the Midpoint Ellipse Drawing Algorithm.
- To implement the algorithm using C language.
- To draw an ellipse on the screen using the implemented algorithm.

Theory:

The Midpoint Ellipse Drawing Algorithm is an efficient algorithm for drawing ellipses. It works by iteratively selecting points on the ellipse using the midpoint between the previously selected points. The algorithm uses the properties of the ellipse to determine the next point, and it uses symmetry to reduce the number of computations.

Algorithm:

1. Compute the parameters of the ellipse (center coordinates, major and minor axis lengths).
2. Set the initial point (x0, y0) as the point on the ellipse where x = 0 and y = b.
3. Calculate the decision parameter p1 = b^2 - a^2b + (1/4)a^2, where a and b are the major and minor axis lengths, respectively.
4. At each step, determine whether to move in the x direction (increment x by 1) or the y direction (decrement y by 1) based on the value of the decision parameter.
5. Calculate the decision parameter for the next point (p2) based on the current point (xi, yi) and the direction of movement.
6. Repeat steps 4-5 until x >= a.

Source Code:

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

void midpointEllipse(int xc, int yc, int a, int b) {
    int x = 0, y = b;
    int p1 = b * b - a * a * b + (a * a) / 4;
    int d1 = 2 * b * b * x + b * b + a * a * (-2 * y + 1);

    while (2 * b * b * x <= 2 * a * a * 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);

        if (d1 < 0) {
            x++;
            p1 = p1 + 2 * b * b * x + b * b;
            d1 = d1 + 2 * b * b * x + b * b;
        } else {
            x++;
            y--;
            p1 = p1 + 2 * b * b * x + b * b - 2 * a * a * y;
            d1 = d1 + 2 * b * b * x - 2 * a * a * y + b * b;
        }
    }

    int p2 = b * b * (x + 0.5) * (x + 0.5) + a * a * (y - 1) * (y - 1) - a
* a * b * b;
    while (y >= 0) {
        putpixel(xc + x, yc + y, WHITE);
        putpixel(xc - x, yc + y, WHITE);
        putpixel(xc + x, yc - y, WHITE);
        putpixel(xc - x, yc - y, WHITE);

        if (p2 > 0) {
            y--;
            p2 = p2 - 2 * a * a * y + a * a;
        } else {
            x++;
            y--;
            p2 = p2 + 2 * b * b * x - 2 * a * a * y + a * a;
        }
    }
}

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

    int xc = 250, yc = 250;
    int a = 100, b = 50;

    midpointEllipse(xc, yc, a, b);

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

7. Experiment Name: 2D Transformation Implementation (Scaling, Rotation)

Objectives:

1. To understand the concept of 2D transformation in computer graphics.
2. To implement the scaling and rotation transformation using C programming language.
3. To visualize the transformed image on the output screen.

Theory:

The transformation is a process of converting one image into another image. In computer graphics, we use 2D transformation to modify or transform the image. There are various types of 2D transformation, such as scaling, rotation, translation, and shearing.

Scaling: Scaling is a process of resizing the object. We can increase or decrease the size of the object using scaling. There are two types of scaling, uniform scaling and non-uniform scaling. In uniform scaling, the size of the object increases or decreases equally in all directions. In non-uniform scaling, the size of the object increases or decreases differently in different directions.

Rotation: Rotation is a process of rotating the object around a fixed point or center. We can rotate the object clockwise or counterclockwise by an angle of theta degrees.

Algorithm:

1. Start the program.
2. Declare and initialize the required variables.
3. Load the input image.
4. Implement the scaling transformation on the input image.
5. Implement the rotation transformation on the scaled image.
6. Display the transformed image on the output screen.
7. End the program.

Source Code:

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

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");

    // Load the input image
    readimagefile("input.jpg", 0, 0, getmaxx(), getmaxy());
    // Scaling Transformation
    int sx = 2, sy = 2;
    for (int y = 0; y < getmaxy(); y++)
    {
        for (int x = 0; x < getmaxx(); x++)
        {
            int x1 = x / sx;
            int y1 = y / sy;
            putpixel(x, y, getpixel(x1, y1));
        }
    }

    // Rotation Transformation
    float angle = 30;
    int x0 = getmaxx() / 2;
    int y0 = getmaxy() / 2;
    for (int y = 0; y < getmaxy(); y++)
    {
        for (int x = 0; x < getmaxx(); x++)
        {
            int x1 = x0 + (x - x0) * cos(angle) - (y - y0) * sin(angle);
            int y1 = y0 + (x - x0) * sin(angle) + (y - y0) * cos(angle);
            putpixel(x, y, getpixel(x1, y1));
        }
    }

    // Display the transformed image
    delay(5000);
    cleardevice();
    readimagefile("output.jpg", 0, 0, getmaxx(), getmaxy());

    closegraph();
    return 0;
}
```