Title: Implementation of DDA Line Drawing Algorithm

Introduction:

The Digital Differential Analyzer (DDA) Line Drawing Algorithm is used in computer graphics to draw straight lines between two points. The intermediate points along the line are calculated using equal increments in the x or y direction. Simple arithmetic operations and rounding are used to determine the pixel positions. Although floating-point calculations are involved, the line is generated step by step, making the process easy to understand. This algorithm is considered a fundamental concept in computer graphics and is used as a basis for more advanced line-drawing techniques.

Objectives:

1. To explore the Digital Differential Analyzer (DDA) Line Drawing Algorithm and understand its role in computer graphics.
2. To study the step-by-step calculations used to generate a line between two points.
3. To demonstrate how DDA calculates intermediate pixels for line plotting.
4. To provide a clear explanation of the algorithm's pseudo-code.
5. To implement the algorithm in a specific programming language.

Necessary Tools:

1. Programming Language: Python (for implementing the DDA Line Drawing Algorithm)
2. Software/IDE: Visual Studio Code (VS Code)
3. Hardware: A basic computer or laptop
4. Libraries: Pure Python implementation, no external libraries required

Algorithm:

1. Input the line endpoints:
   Get the coordinates of the start point (x1,y1)(x_1, y_1)(x1,y1) and the end point (x2,y2)(x_2, y_2)(x2,y2).
2. Calculate differences:
   - dx=x2−x1dx = x_2 - x_1dx=x2−x1
   - dy=y2−y1dy = y_2 - y_1dy=y2−y1
3. Find the number of steps:
   - `steps = max(|dx|, |dy|)`
   - This decides how many points we need to plot.
4. Calculate increments for each step:
   - `x_inc = dx / steps` → How much x increases each step
   - `y_inc = dy / steps` → How much y increases each step
5. Initialize starting point:
   - `x = x1`
   - `y = y1`
6. Plot the line:

    - Repeat `steps + 1` times:
        - Round the current `x` and `y` and plot the pixel `(round(x), round(y))`
        - Update the coordinates:
            - `x = x + x_inc`
            - `y = y + y_inc`
7. Stop:
    - When the loop finishes, the line from `(x1, y1)` to `(x2, y2)` is drawn.

Pseudocode:

1. Start the function with inputs: `x1, y1, x2, y2` (the line endpoints).
2. Calculate differences:
    - `dx = x2 - x1`
    - `dy = y2 - y1`
3. Find the number of steps:
    - `steps = max(abs(dx), abs(dy))`
4. Calculate increments for each step:
    - `x_inc = dx / steps` → amount to increase x each step
    - `y_inc = dy / steps` → amount to increase y each step
5. Initialize starting point:
    - `x = x1`
    - `y = y1`
6. Loop for each step (from 0 to steps):
    - Plot the current pixel at `(round(x), round(y))`
    - Update the coordinates:
        - `x = x + x_inc`
        - `y = y + y_inc`
7. End the loop when all steps are completed.

Activity Diagram:

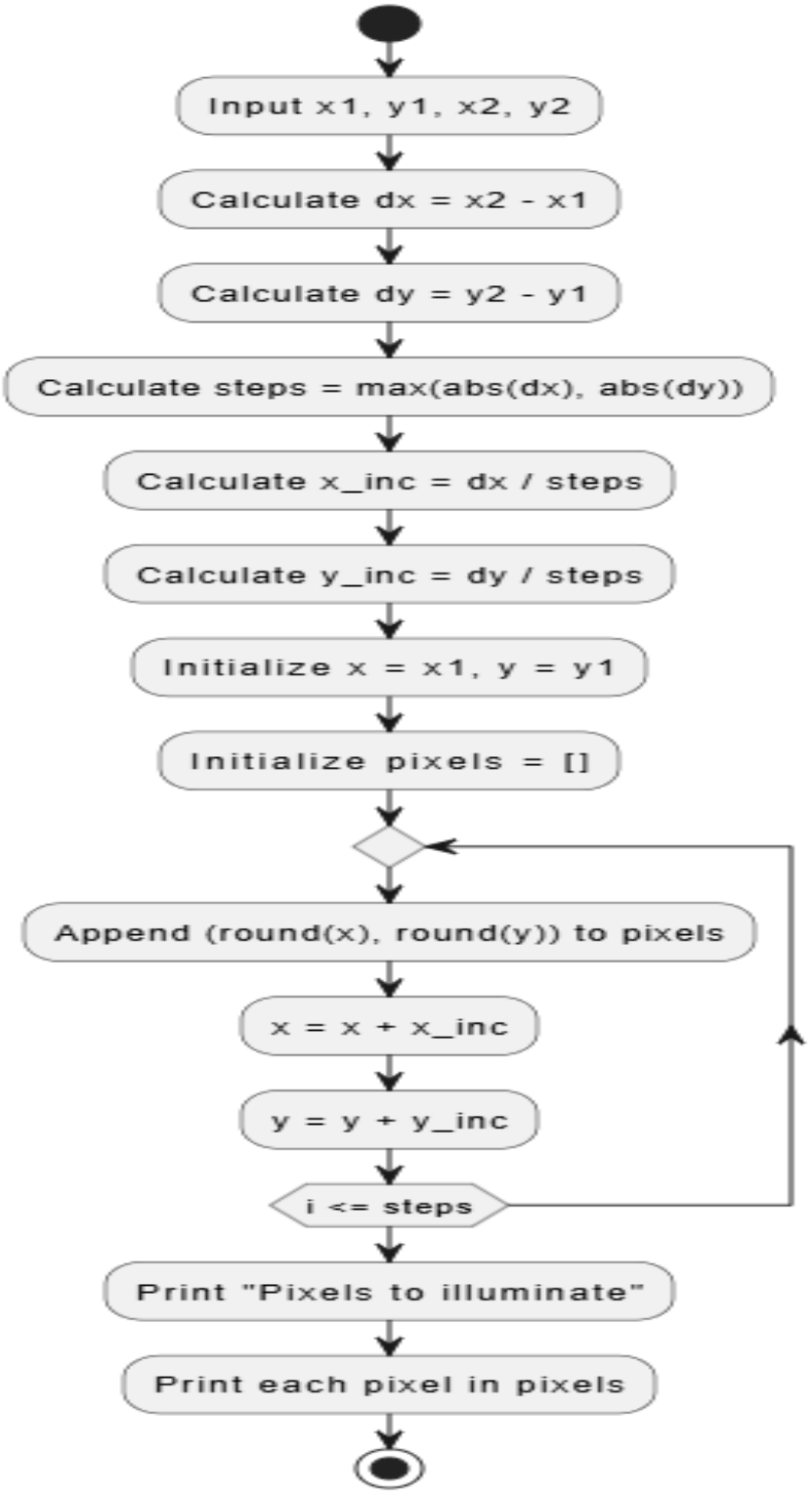


Figure-1:Activity diagram of DDA Line Drawing Algorithm

Source Code:

```python
def dda(x1, y1, x2, y2):
    dx, dy = x2 - x1, y2 - y1
    steps = max(abs(dx), abs(dy)) or 1  # avoid division by zero
    x_inc, y_inc = dx / steps, dy / steps
    x, y = x1, y1
    pixels = []

    for _ in range(steps + 1):
        pixels.append((round(x), round(y)))
        x += x_inc
        y += y_inc

    return pixels

def main():
    vals = input("Enter x1 y1 x2 y2: ").split()
    x1, y1, x2, y2 = map(int, vals)
    pixels = dda(x1, y1, x2, y2)

    print("Pixels to illuminate:")
    for p in pixels:
        print(p)

if __name__ == "__main__":
    main()
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                Python -
PS F:\1st Skills\Python> & C:/Users/baizid/AppData/Local/Programs/Python/Python313/python.exe '
/bresenham.py"
Enter x1 y1 x2 y2: 2 3 12 8
Pixels to illuminate:
(2, 3)
(3, 4)
(4, 4)
(5, 4)
(6, 5)
(7, 6)
(8, 6)
(9, 6)
(10, 7)
(11, 8)
(12, 8)
PS F:\1st Skills\Python>
```

Limitations of the Code:

The DDA Line Drawing Algorithm uses floating-point calculations, which can make it slower when large-scale graphics are involved. Minor errors in pixel plotting may occur due to rounding, especially for steep lines. The algorithm is not as efficient as Bresenham's method, which relies only on integer calculations.

Conclusion:

The DDA Line Drawing Algorithm is simple and can be easily implemented. Pixel coordinates for a straight line between two points are generated effectively. Although it is less efficient than Bresenham's algorithm, it helps in understanding the process of line generation clearly. This algorithm serves as a fundamental concept in computer graphics and forms the basis for more advanced techniques.

Problems Faced:

Division by zero had to be handled when both endpoints were the same. Accurate pixel plotting required careful rounding of coordinates. The program had to be ensured to work correctly for all types of line slopes, including steep, shallow, vertical, and horizontal lines.

References:

1. D. F. Rogers and J. A. Adams, *Mathematical Elements for Computer Graphics*, 2nd ed., McGraw-Hill, 1990.
2. E. Angel, *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*, 6th ed., Pearson, 2011.
3. K. Hearn and M. Baker, *Computer Graphics with OpenGL*, 4th ed., Pearson, 2014.

Title: Implementation of Bresenham Line Drawing Algorithm

Introduction:

The Bresenham Line Drawing Algorithm is used in computer graphics to draw straight lines efficiently between two points. Only integer calculations are used, which makes the algorithm faster than floating-point methods. The pixels along the line are chosen step by step based on the closest approximation to the ideal line. This method is considered accurate, efficient, and widely used in computer graphics applications.

Objectives:

1. To explore the Bresenham Line Drawing Algorithm and understand its purpose in computer graphics.
2. To study the underlying principles and logic behind the algorithm.
3. To demonstrate the advantages of Bresenham's algorithm over other line drawing techniques.
4. To provide a clear explanation of the algorithm's pseudo-code.
5. To implement the algorithm in a specific programming language.

Necessary Tools:

1. Programming Language: Python (for implementing the Bresenham Line Drawing Algorithm)
2. Software/IDE: Visual Studio Code (VS Code)
3. Hardware: A basic computer or laptop
4. Libraries: pure Python implementation, no external libraries used

Algorithm:

1. Take starting point `(x1, y1)` and ending point `(x2, y2)`.
2. Calculate differences: `dx = x2 - x1`, `dy = y2 - y1`.
3. Initialize the decision parameter: `p = 2*dy - dx`.
4. Start at `(x1, y1)` and print the point.
5. Repeat while `x < x2`:
    - If `p < 0`, move right → `x = x + 1`, update `p = p + 2*dy`.
    - Else, move diagonal → `x = x + 1`, `y = y + 1`, update `p = p + 2*dy - 2*dx`.
6. Print each new point until the end point `(x2, y2)` is reached.

Pseudocode:

1. Start with two points: `(x1, y1)` as the starting point and `(x2, y2)` as the ending point.
2. Calculate differences:
    - `dx = x2 - x1`
    - `dy = y2 - y1`
3. Initialize the decision parameter:
    - `p = 2 * dy - dx`
4. Set the starting point:

    - `x = x1`
    - `y = y1`
    - Print `(x, y)`
5. Loop until `x` reaches `x2`:

1. If `p < 0` → move horizontally:
    - `x = x + 1`
    - `p = p + 2 * dy`
2. Else → move diagonally:
    - `x = x + 1`
    - `y = y + 1`
    - `p = p + 2 * dy - 2 * dx`
3. Print the current point `(x, y)`

6. End Loop when the line reaches `(x2, y2)`.
7. End Function

Activity Diagram:

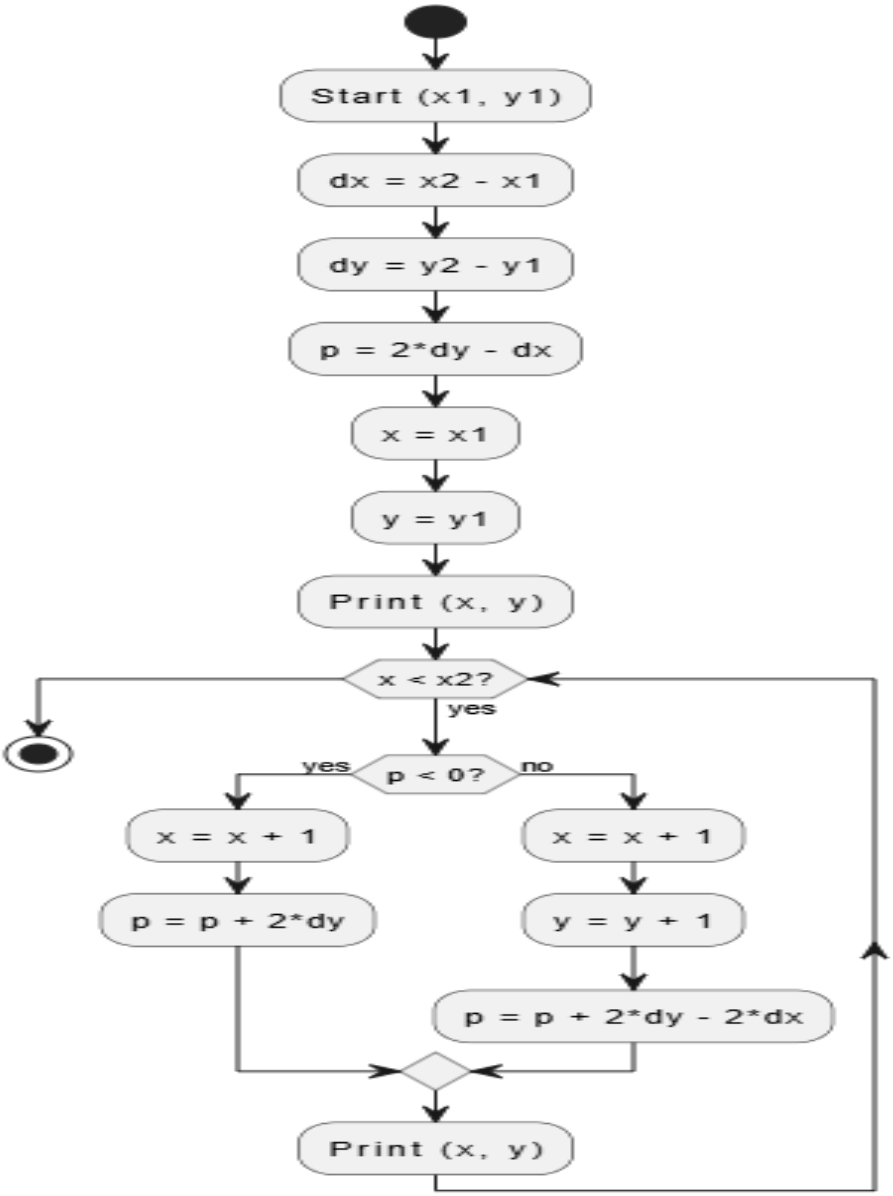


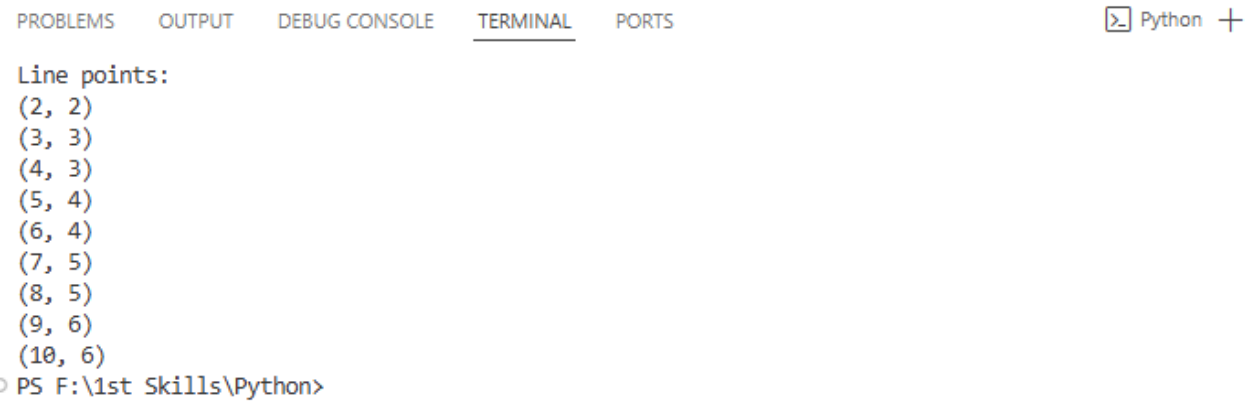
Figure-1: Activity diagram of Bresenham Line Drawing Algorithm

Source Code:

bresenham.py > ...

```python
def bresenham(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    p = 2*dy - dx
    x, y = x1, y1

    print("Line points:")
    print(f"({x}, {y})")

    while x < x2:
        if p < 0:
            x += 1
            p += 2*dy
        else:
            x += 1
            y += 1
            p += 2*dy - 2*dx
        print(f"({x}, {y})")

bresenham(2, 2, 10, 6)
```

Output:



Limitations of the Code:

The code works only for lines with slopes between 0 and 1. Lines with negative slopes or vertical orientation cannot be handled directly. Steep lines require additional logic to be drawn correctly.

Conclusion:

Bresenham's algorithm is a fast and efficient way to draw lines using integer arithmetic. The implementation prints each line pixel, demonstrating how a computer decides pixel positions. Using simple human logic makes the algorithm easy to understand and implement.

Problems Faced:

It was challenging to understand how the decision parameter p determines the next pixel. Handling steep or negative slopes required modifications. The step-by-step logic of plotting each point had to be carefully checked, and debugging the process was sometimes difficult.

References:

[1] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1996.
[2] R. J. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.
[3] D. Hearn and M. P. Baker, *Computer Graphics with OpenGL*, 3rd ed., Pearson, 2003.

Introduction: The Midpoint Circle Algorithm is a fundamental and widely used rasterization technique in computer graphics for drawing circles on a digital screen. Rasterization refers to the process of converting geometric shapes into discrete pixels, enabling their display on a two-dimensional grid. Circles are essential primitives in graphics applications, and efficient algorithms are required to render them accurately and swiftly. The Midpoint Circle Algorithm provides a way to determine the pixels that approximate a circle centered at a specified coordinate with a given radius. Unlike other circle-drawing algorithms that rely on costly trigonometric calculations, the Midpoint Circle Algorithm employs integer arithmetic and bit-wise operations, making it highly efficient for real-time rendering applications. In this lab report, we explore the principles behind the Midpoint Circle Algorithm, implement it in a programming language, and analyze its performance and accuracy compared to other circle-drawing approaches. Through this exploration, we aim to gain a deeper understanding of the algorithm's underlying principles and practical applications in computer graphics.

Objective: The objective of the Midpoint Circle Algorithm lab report is to explore, implement, and analyze the efficiency and accuracy of the Midpoint Circle Drawing Algorithm in computer graphics.

Required Tools: This report will cover the essentials of implementing the Mid-point circle drawing Algorithm in C using the Code::Blocks IDE.

- Programming Language: **C**

- IDE: **Code::Blocks**

- Mathematical Tools: **Math Library.**

Algorithm:

Mid-point circle drawing algorithm

1. **Start**.

2. **Initialize variables**:

3. **Input radius**:

4. **Calculate initial decision parameter**:

   - p0 = 1 - r

5. **While loop** (while x0 < y0):

      - **If p0 < 0**:
        - Increment x0 by 1.
        - Update p0 by adding 2 * x0 + 1.
        - Print the point (x0, y0).
      - **Else**:

- Increment x0 by 1.
- Decrement y0 by 1.
- Update p0 by adding 2 * x0 - 2 * y0 + 1.
- Print the point (x0, y0).

6. **End**.

Source Code:

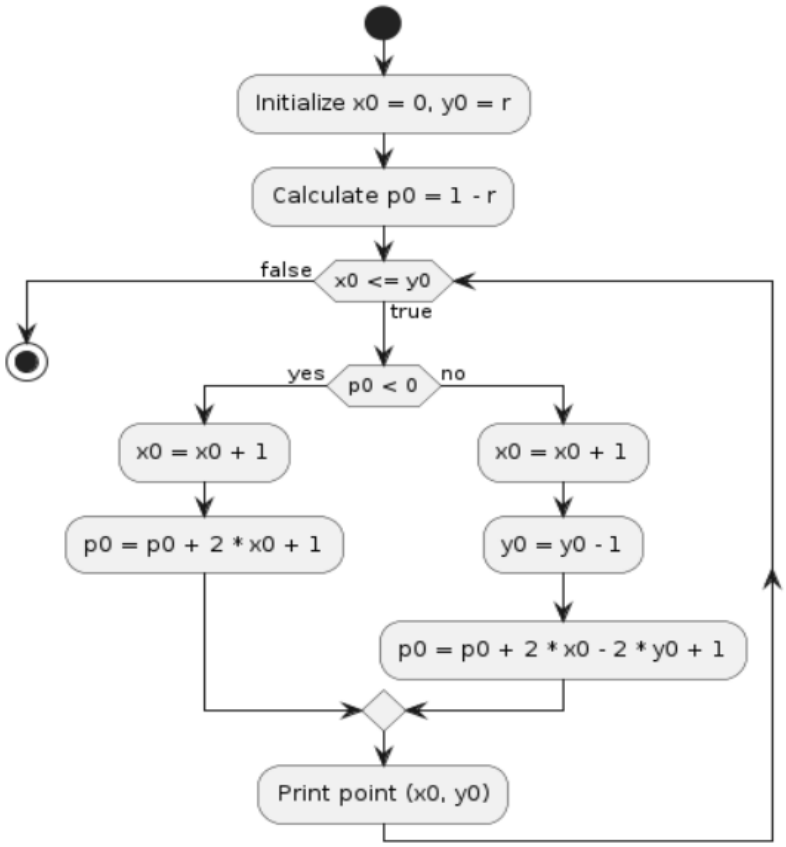
Start here | Mindpoint.c

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

int main()
{
    int x0=0,y0,r,p0,k=0;
    printf("Enter the value of Radius: ");
    scanf("%d", &r);
    p0= 1-r;
    y0= r;
    while(x0<y0)
    {
        if(p0<0)
        {
            x0= x0+1;
            p0= p0+2*x0+1;
            printf("%d %d\n", x0,y0);
        }
        else
        {
            x0= x0+1;
            y0= y0-1;
            p0= p0+2*x0-2*y0+1;
            printf("%d %d\n", x0,y0);
        }
    }
    return 0;
}
```

Output of Code:

```
"E:\Study 7th\CGIPL\Mindpoint.exe"
Enter the value of Radius: 10
1 10
2 10
3 10
4 9
5 9
6 8
7 7

Process returned 0 (0x0)   execution time : 4.253 s
Press any key to continue.
```

Flow Chart:



Limitation of Code:

- The code does not check for invalid input parameters, such as negative radii or non-integer center coordinates.
- The current implementation assumes a fixed center at (0, 0), which limits its general applicability.
- The visualization is basic and does not handle cases where parts of the circle extend beyond the plotting area.
- The code does not include anti-aliasing, leading to jagged or pixelated circle edges.

Limitations of Algorithm:

- The circle generated by this algorithm is not smooth.
- This algorithm is time consuming.
- The algorithm is specific to 2D circle rasterization and does not directly extend to ellipses or other curved shapes.
- As the circle radius increases, the algorithm's inefficiency becomes apparent, leading to slower rendering times compared to specialized algorithms like Bresenham's Circle Algorithm for large circles.

How to Prevent Limitation of the Code/Algorithm:

- Implement checks to ensure valid input parameters, such as positive radii and appropriate center coordinates.
- Modify the code to accept any arbitrary center point and update the plotting function accordingly.
- Implement boundary checks to ensure all points are within the drawable area and dynamically adjust the plotting area.
- Incorporate anti-aliasing techniques to smooth the edges of the circle and reduce pixelation.
- For large radii, optimize the algorithm to handle large datasets efficiently, possibly through spatial partitioning techniques.

Discussion:

In conclusion, the Midpoint Circle Algorithm proves to be a fundamental and efficient technique for rasterizing circles in computer graphics. The algorithm's ability to use integer arithmetic and bit-wise operations instead of costly trigonometric calculations makes it highly suitable for real-time rendering applications. However, it is limited to circles with integer radii and does not directly extend to other curved shapes or provide built-in anti-aliasing. Despite this, the Midpoint Circle Algorithm remains a valuable tool for rendering small to medium-sized circles with accuracy and speed. By comparing it with other circle-drawing approaches, we identified its advantages and limitations, facilitating a better understanding of its applications in computer graphics. This lab report serves as a stepping stone for further research and optimization of circle rasterization algorithms, contributing to advancements in real-time graphics rendering.

References:

1. https://www.geeksforgeeks.org/mid-point-circle-drawing-algorithm/
2. https://www.javatpoint.com/computer-graphics-midpoint-circle-algorithm

Title: RGB to Grayscale Conversion

Introduction:

This experiment focuses on converting a color image from the RGB color model to its equivalent grayscale representation. In computer graphics, color models define how colors are represented numerically. The grayscale transformation is performed using a weighted sum of the Red, Green, and Blue components, accounting for human visual sensitivity to each color channel.

Objectives:

1. Understand the concept of RGB and Grayscale color models.
2. Learn to apply the grayscale conversion formula programmatically.
3. Demonstrate the visual difference between color and grayscale representations.
4. Implement and verify the correctness of the grayscale conversion formula.

Necessary Tools:

- Programming Language: Python
- Software/IDE: Visual Studio Code or Google Colab
- Hardware: Standard computer or laptop
- Libraries: None (basic Python)

Algorithm:

```
1. Input RGB values (R, G, B) within [0,255].
2. Apply the grayscale formula: Gray = 0.299*R + 0.587*G + 0.114*B
3. Round the result to the nearest integer.
4. Output the grayscale color as (Gray, Gray, Gray).
```

Pseudocode:

```
BEGIN
Input R, G, B
Gray = 0.299*R + 0.587*G + 0.114*B
Gray = round(Gray)
Print (Gray, Gray, Gray)
END
```

Source Code:

```
# Python Program for RGB to Grayscale Conversion
R = int(input("Enter Red value (0-255): "))
G = int(input("Enter Green value (0-255): "))
B = int(input("Enter Blue value (0-255): "))

Gray = round(0.299 * R + 0.587 * G + 0.114 * B)
print("Grayscale Value:", (Gray, Gray, Gray))
```

Output:

```
Input:
R = 100, G = 150, B = 200

Output:
Grayscale Value: (141, 141, 141)
```

Limitations of the Code:

The grayscale conversion formula may slightly vary across systems due to rounding differences. The conversion assumes linear RGB, so gamma-corrected values might produce minor inaccuracies.

Conclusion:

The grayscale conversion successfully transforms RGB color into its brightness-equivalent form. It helps reduce image complexity while preserving essential visual intensity information.

Problems Faced:

Ensuring correct rounding of floating-point results and handling input validation for RGB values were required for accurate output.

References:

1. E. Angel, Interactive Computer Graphics: A Top-Down Approach Using OpenGL, 6th Ed., Pearson, 2011.
2. D. F. Rogers, Mathematical Elements for Computer Graphics, 2nd Ed., McGraw-Hill, 1990.
3. K. Hearn and M. P. Baker, Computer Graphics with OpenGL, 4th Ed., Pearson, 2014.

Title: Window to Viewport Mapping

Introduction:

This experiment demonstrates mapping coordinates from world/window space to screen/viewport space. The mapping is linear and preserves relative positions. Window-to-viewport transformation is a common step in rendering pipelines to convert logical coordinates into device coordinates.

Objectives:

1. Understand coordinate mapping between window and viewport.
2. Implement the mapping formula programmatically.
3. Verify correctness with sample inputs and outputs.
4. Learn rounding and handling of coordinate ranges.

Necessary Tools:

- Programming Language: Python
- Software/IDE: Visual Studio Code or Google Colab
- Hardware: Standard computer or laptop
- Libraries: None (basic Python)

Algorithm:

```
1. Input window coordinates: (window_min_x, window_max_x, window_min_y, window_max_y).
2. Input viewport coordinates: (viewport_min_x, viewport_max_x, viewport_min_y, viewport_max_y).
3. Input a point in window coordinates: (x_window, y_window).
4. Apply mapping formulas:
   x_viewport = ((x_window - window_min_x) / (window_max_x - window_min_x)) * (viewport_max_x - viewp
   y_viewport = ((y_window - window_min_y) / (window_max_y - window_min_y)) * (viewport_max_y - viewp
5. Round the results (optional) and output (x_viewport, y_viewport).
```

Pseudocode:

```
BEGIN
INPUT window_min_x, window_max_x, window_min_y, window_max_y
INPUT viewport_min_x, viewport_max_x, viewport_min_y, viewport_max_y
INPUT x_window, y_window
x_viewport = ((x_window - window_min_x) / (window_max_x - window_min_x)) * (viewport_max_x - viewport
y_viewport = ((y_window - window_min_y) / (window_max_y - window_min_y)) * (viewport_max_y - viewport
x_viewport = round(x_viewport)
y_viewport = round(y_viewport)
PRINT x_viewport, y_viewport
END
```

Source Code:

```
# Python Program for Window to Viewport Mapping
window_min_x, window_max_x, window_min_y, window_max_y = map(int, input("Enter window (minx maxx miny
viewport_min_x, viewport_max_x, viewport_min_y, viewport_max_y = map(int, input("Enter viewport (minx
x_window, y_window = map(float, input("Enter point in window (x y): ").split())

x_viewport = ((x_window - window_min_x) / (window_max_x - window_min_x)) * (viewport_max_x - viewport
y_viewport = ((y_window - window_min_y) / (window_max_y - window_min_y)) * (viewport_max_y - viewport
```

```
print("Point in viewport:", (round(x_viewport), round(y_viewport)))
```

Output:

```
Input:
Window: (0, 100, 0, 100)
Viewport: (0, 500, 0, 500)
Point in Window: (50, 50)

Output:
Point in viewport: (250, 250)
```

Limitations of the Code:

Mapping assumes non-zero window dimensions; division by zero must be handled. Floating-point precision may affect final rounded pixel coordinates.

Conclusion:

Window-to-viewport mapping correctly converts coordinates while preserving relative positions and scales. It is essential in graphics pipelines before rasterization.

Problems Faced:

Validating input ranges and preventing division by zero (when window_min == window_max or window_min_y == window_max_y) were necessary.

References:

1. E. Angel, Interactive Computer Graphics: A Top-Down Approach Using OpenGL, 6th Ed., Pearson, 2011.
2. D. F. Rogers, Mathematical Elements for Computer Graphics, 2nd Ed., McGraw-Hill, 1990.
3. K. Hearn and M. P. Baker, Computer Graphics with OpenGL, 4th Ed., Pearson, 2014.

Title: Binary Image Pixel Count

Introduction:

This experiment involves counting black and white pixels in a binary image represented as a 2D matrix of 0s and 1s. This simple analysis is a foundation for image processing tasks such as segmentation statistics and basic morphological operations.

Objectives:

1. Understand binary image representation.
2. Implement a program to count pixel occurrences.
3. Practice iterating through 2D arrays and maintaining counters.
4. Validate results with sample input.

Necessary Tools:

- Programming Language: Python
- Software/IDE: Visual Studio Code or Google Colab
- Hardware: Standard computer or laptop
- Libraries: None (basic Python)

Algorithm:

```
1. Input dimensions M (rows) and N (columns).
2. Initialize counters: black_pixels = 0, white_pixels = 0.
3. For each row, read N binary values.
4. For each value, if 0 increment black_pixels, else increment white_pixels.
5. After processing all rows, output counts.
```

Pseudocode:

```
BEGIN
INPUT M, N
black_pixels = 0
white_pixels = 0
FOR i in range(M):
    read row of N values
    FOR val in row:
        IF val == 0:
            black_pixels += 1
        ELSE:
            white_pixels += 1
PRINT black_pixels, white_pixels
END
```

Source Code:

```
# Python Program to Count Black and White Pixels
M, N = map(int, input("Enter rows and columns: ").split())
black_pixels = 0
white_pixels = 0
for _ in range(M):
    row = list(map(int, input().split()))
```

```
    for val in row:
        if val == 0:
            black_pixels += 1
        else:
            white_pixels += 1
print("Black Pixels:", black_pixels)
print("White Pixels:", white_pixels)
```

Output:

```
Input:
3 3
1 0 1
0 1 0
1 1 0

Output:
Black Pixels: 4
White Pixels: 5
```

Limitations of the Code:

Assumes input is well-formed (only 0s and 1s). No checks for invalid values were included; those should be added for robustness.

Conclusion:

Counting pixels provides quick statistics useful for simple image analysis tasks and preprocessing steps.

Problems Faced:

Ensuring correct parsing of input rows and validating that each row contains N values were necessary to avoid errors.

References:

1. R. C. Gonzalez and R. E. Woods, Digital Image Processing, 3rd Ed., Pearson, 2007.
2. A. K. Jain, Fundamentals of Digital Image Processing, Prentice Hall, 1989.

Title: RLE Image Compression and Restoration

Introduction:

This experiment demonstrates Run-Length Encoding (RLE), a simple lossless compression technique useful for binary images or images with long runs of identical pixels. The lab covers both compression (encoding) and restoration (decoding).

Objectives:

1. Understand run-length encoding for simple images.
2. Implement compression and decompression algorithms.
3. Verify correctness with sample input and restored output.
4. Observe compression efficiency for example patterns.

Necessary Tools:

- Programming Language: Python
- Software/IDE: Visual Studio Code or Google Colab
- Hardware: Standard computer or laptop
- Libraries: None (basic Python)

Algorithm:

```
Compression:
1. Input list of binary pixels.
2. Initialize compressed = [] and count = 1.
3. Loop from second pixel to end:
   if current == previous: increment count
   else: append (previous, count); reset count = 1
4. After loop append last (pixel, count)
5. Output compressed list

Restoration:
1. Input compressed list of tuples (pixel, count)
2. Initialize restored = []
3. For each (pixel, count): extend restored by [pixel]*count
4. Output restored list
```

Pseudocode:

```
BEGIN Compression
INPUT pixels_list
compressed = []
count = 1
FOR i from 1 to len(pixels_list)-1:
    IF pixels_list[i] == pixels_list[i-1]:
        count += 1
    ELSE:
        append (pixels_list[i-1], count) to compressed
        count = 1
append (pixels_list[-1], count) to compressed
PRINT compressed
END

BEGIN Restoration
INPUT compressed_list
```

```
restored = []
FOR (pixel, count) in compressed_list:
    extend restored by pixel repeated count times
PRINT restored
END
```

Source Code:

```
# Python Program for RLE Compression and Restoration
pixels = list(map(int, input("Enter binary pixels separated by space: ").split()))
# Compression
compressed = []
count = 1
for i in range(1, len(pixels)):
    if pixels[i] == pixels[i-1]:
        count += 1
    else:
        compressed.append((pixels[i-1], count))
        count = 1
compressed.append((pixels[-1], count))
print("Compressed Image:", compressed)

# Restoration
restored = []
for val, cnt in compressed:
    restored.extend([val] * cnt)
print("Restored Image:", restored)
```

Output:

```
Input:
1 1 0 0 0 1 1 1 0 0

Output:
Compressed Image: [(1, 2), (0, 3), (1, 3), (0, 2)]
Restored Image: [1, 1, 0, 0, 0, 1, 1, 1, 0, 0]
```

Limitations of the Code:

RLE works best for long runs of identical pixels; for noisy or high-entropy images it may increase size. Input validation is minimal.

Conclusion:

RLE demonstrated lossless compression and exact restoration for binary sequences. It is a good introductory compression technique for simple images.

Problems Faced:

Handling edge cases like empty input and validating binary values (only 0 or 1) were necessary to ensure robustness.

References:

1. R. M. Gray, Compression: Algorithms and Applications, IEEE Press, 1997.
2. R. C. Gonzalez and R. E. Woods, Digital Image Processing, 3rd Ed., Pearson, 2007.