International Journal of Scientific Research & Engineering Trends
Volume 10, Issue 6, Nov-Dec-2024, ISSN (Online): 2395-566X

Comparative Study of Dda Algarthem, Bresenham's Line-Drawing Algorithm, Midpoint Circle Algorithm Using Python

Professor N.Tamiloli, T.Gowtham
Department of Mechanical Engineering,
PERI Institute of Technology

Abstract- Efficient algorithms for rendering geometric shapes are fundamental in computer graphics. This study presents a comparative analysis of the Digital Differential Analyzer (DDA), Bresenham's line-drawing, and Midpoint circle algorithms. We evaluate their performance in terms of computational efficiency, accuracy, and ease of implementation. Python is used as the platform to implement and test the algorithms. Experimental results demonstrate that while DDA offers simplicity in implementation, Bresenham's algorithm is computationally more efficient for line drawing. The Midpoint circle algorithm proves robust for circular shapes but is relatively complex. This paper provides insights into the algorithms' suitability for various real-world applications, backed by runtime performance and output quality metrics.

Index Terms- algorithms, geometric, Analyzer, Python, circular. etc

I. INTRODUCTION

1. DDA Algorithm

The Digital Differential Analyzer (DDA) algorithm is a fundamental technique in computer graphics used for rendering straight lines. Introduced as one of the earliest algorithms for rasterization, DDA employs an incremental approach to determine intermediate points between two endpoints of a line. By leveraging the line equation y=mx+c, the algorithm calculates successive pixel coordinates through stepwise additions, ensuring a smooth approximation of the line.

DDA is based on the principle of linear interpolation. For a given line, it determines the number of steps required based on the larger difference in the x or y coordinates, and then computes increments in the respective direction. The use of floating-point arithmetic allows DDA to maintain accuracy in slopes, especially for non-integer endpoints, but it can also lead to performance inefficiencies due to rounding and computational overhead.

Despite its simplicity and straightforward implementation, the DDA algorithm is less efficient compared to integer-based algorithms like Bresenham's line-drawing algorithm. However, it remains an important pedagogical tool and foundational concept in computer graphics education. DDA is particularly useful in scenarios where simplicity and ease of understanding outweigh the need for high computational efficiency, making it a staple for introductory graphics programming. The output shows in figure 1.

Some references shown in below

Hearn, D., & Baker, M. P	Detailed explanation of line and circle-drawing algorithms and their applications
https://www.tutorialspoint.com	Explains DDA, Bresenham, and Midpoint algorithms with examples.
https://www.geeksforgeeks.org	Step-by-step breakdown of DDA and Bresenham algorithms.
Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F.	Comprehensive resource on graphics algorithms, including DDA, Bresenham, and Midpoint methods)
Python-based Graphics Algorithms - GitHub Repositories	GitHub - Line Drawing Algorithms
https://matplotlib.org/stablo/contents.html	https://matplotlib.org/stable/contents.html

DDA algorithms code

```
def DDA(x1, y1, x2, y2):
dx = x2 - x1
dy = y2 - y1
steps = abs(dx) if abs(dx) > abs(dy) else abs(dy)
#steps = max(abs(dx),abs(dy))
x_increment = dx / steps
y_increment = dy / steps
  x = x1
    y = y1
    points = [ ]
    for i in range(steps):
        x += x_increment
        y += y_increment
```

© 2024 IJSRET

International Journal of Scientific Research & Engineering Trends
Volume 10, Issue 6, Nov-Dec-2024, ISSN (Online): 2395-566X

```
        points.append((round(x), round(y)))
    return points, steps
def draw_line(points):
    x_values = [point[0] for point in points]
    y_values = [point[1] for point in points]
    plt.plot(x_values, y_values, marker='o',mec='hotpink',mfc='hotpink')
    plt.xlabel('X_axis')
    plt.ylabel('Y_axis')
    plt.title('DDA Algorithm Line Drawing')
    plt.grid(True)
    plt.show()
x1, y1 = 2, 4
x2, y2 =3, 6
points, steps = DDA(x1, y1, x2, y2)
print("Number of iterations:", steps)
print("Points between the two  points:", points)
draw_line(points)
```

Output

Number of iterations: 2

Points between the two points: [(2, 5), (3, 6)]

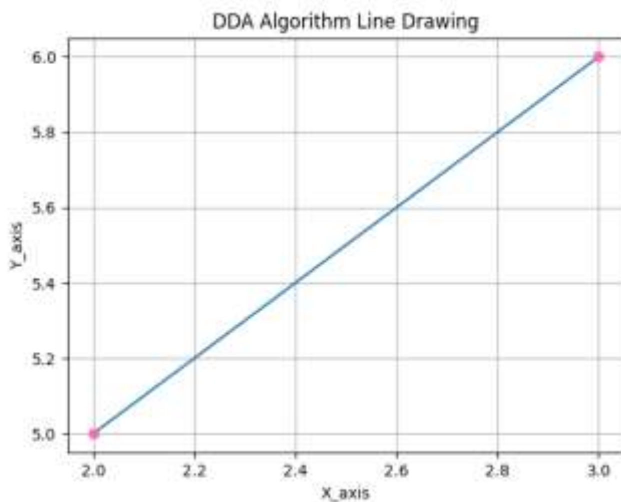

Figure 1: Line drawing DDA algorithm

2. Bresenham's Line-Drawing Algorithm

Bresenham's line-drawing algorithm is a highly efficient rasterization technique used to draw straight lines on a grid-based display, such as a computer screen. Introduced by Jack Bresenham in 1965, this algorithm eliminates the need for floating-point operations by using integer arithmetic, making it faster and more suitable for real-time graphics rendering.

The algorithm determines which pixel to illuminate at each step based on the error term, which represents the deviation of the line from the ideal mathematical line. Starting from one endpoint, it calculates the difference in x and y coordinates (dx and dy) and decides the "steepness" of the line. Based on this steepness, Bresenham's algorithm either increments the x-coordinate or y-coordinate while adjusting the error term accordingly.

For lines with gentle slopes ($0 \le m \le 1$), the algorithm moves one pixel along the x-axis for each step and decides whether to move vertically based on the error term. For steeper lines ($m>1$), the roles of x and y are swapped. The decision-making is achieved through a simple comparison operation, and the error term is updated using addition or subtraction, ensuring computational efficiency.

Key advantages of Bresenham's algorithm include its ability to draw lines with high accuracy and its suitability for hardware implementation due to its reliance on integer calculations. It also handles edge cases like vertical and horizontal lines efficiently. Bresenham's method is widely used in applications requiring real-time rendering, making it a cornerstone of computer graphics.

```
def Bresenham3D(x1, y1, z1, x2, y2, z2):
    ListOfPoints = []
    ListOfPoints.append((x1, y1, z1))
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    dz = abs(z2 - z1)
    if (x2 > x1):
        xs = 1
    else:
        xs = -1
    if (y2 > y1):
        ys = 1
    else:
        ys = -1
    if (z2 > z1):
        zs = 1
    else:
        zs = -1

    # Driving axis is X-axis"
    if (dx >= dy and dx >= dz):
        p1 = 2 * dy - dx
        p2 = 2 * dz - dx
        while (x1 != x2):
            x1 += xs
            if (p1 >= 0):
                y1 += ys
                p1 -= 2 * dx
            if (p2 >= 0):
                z1 += zs
                p2 -= 2 * dx
            p1 += 2 * dy
            p2 += 2 * dz
            List Of Points. append((x1, y1, z1))

    # Driving axis is Y-axis"
```

© 2024 IJSRET

```
    elif (dy >= dx and dy >= dz):
        p1 = 2 * dx - dy
        p2 = 2 * dz - dy
        while (y1 != y2):
            y1 += ys
            if (p1 >= 0):
                x1 += xs
                p1 -= 2 * dy
            if (p2 >= 0):
                z1 += zs
                p2 -= 2 * dy
            p1 += 2 * dx
            p2 += 2 * dz
            ListOfPoints.append((x1, y1, z1))

    # Driving axis is Z-axis"
    else:
        p1 = 2 * dy - dz
        p2 = 2 * dx - dz
        while (z1 != z2):
            z1 += zs
            if (p1 >= 0):
                y1 += ys
                p1 -= 2 * dz
            if (p2 >= 0):
                x1 += xs
                p2 -= 2 * dz
            p1 += 2 * dy
            p2 += 2 * dx
            ListOfPoints.append((x1, y1, z1))
    return ListOfPoints


def main():
    (x1, y1, z1) = (-1, 1, 1)
    (x2, y2, z2) = (5, 3, -1)
    ListOfPoints = Bresenham3D(x1, y1, z1, x2, y2, z2)
    print(ListOfPoints)
```

Output
(-1, 1, 1), (0, 1, 1), (1, 2, 0), (2, 2, 0), (3, 2, 0), (4, 3, -1), (5, 3, -1)

3. Midpoint Circle Algorithm
The Midpoint Circle Algorithm is a widely used method in computer graphics for rasterizing circles. It is an incremental algorithm that leverages the symmetry of circles to minimize calculations, making it efficient and suitable for rendering circular shapes in pixel-based displays.

At its core, the algorithm works by determining the pixels closest to the ideal circle at each step. Starting at the top of the circle (x = 0, y = radius), it calculates the next pixel position by evaluating a decision parameter derived from the circle's implicit equation: x2+y2−r2=0. This decision parameter helps determine whether the next pixel should move horizontally, vertically, or diagonally.

The algorithm exploits the eight-way symmetry of a circle, which allows it to compute only one-eighth of the circle and mirror the results to the other seven octants. This significantly reduces computation. At each step, the decision parameter is updated using integer arithmetic, eliminating the need for floating-point calculations, which boosts performance.

As the algorithm progresses, the y-coordinate is decremented (as it moves towards the horizontal axis), and the x-coordinate is incremented. The decision parameter is updated based on whether the midpoint between two potential pixel choices lies inside or outside the circle.

The Midpoint Circle Algorithm is efficient, simple, and easy to implement, making it a popular choice in computer graphics. Its applications include rendering circles, arcs, and rounded shapes in 2D graphics, CAD software, and other graphical environments.

```
def midPointCircleDraw(x_centre, y_centre, r):
    x = r
    y = 0
        print("(", x + x_centre, ", ", y + y_centre, ")",
              sep = "", end = "")

         if (r > 0) :

        print("(", x + x_centre, ", ",-y + y_centre, ")",
                sep = "", end = "")
        print("(", y + x_centre, ", ",
                 x + y_centre, ")",
                 sep = "", end = "")
        print("(", -y + x_centre, ", ",
                  x + y_centre, ")", sep = "")
         P = 1 - r

    while x > y:

        y += 1

        if P <= 0:
            P = P + 2 * y + 1

        else:
            x -= 1
            P = P + 2 * y - 2 * x + 1

        if (x < y):
            break

        print("(", x + x_centre, ", ", y + y_centre,
                        ")", sep = "", end = "")
```

© 2024 IJSRET

International Journal of Scientific Research & Engineering Trends
Volume 10, Issue 6, Nov-Dec-2024, ISSN (Online): 2395-566X

```
        print("(", -x + x_centre, ", ", y + y_centre,
                        ")", sep = "", end = "")
        print("(", x + x_centre, ", ", -y + y_centre,
                        ")", sep = "", end = "")
        print("(", -x + x_centre, ", ", -y + y_centre,
                                ")", sep = "")
            if x != y:
                print("(", y + x_centre, ", ", x + y_centre,
                                ")", sep = "", end = "")
            print("(", -y + x_centre, ", ", x + y_centre,
                                ")", sep = "", end = "")
            print("(", y + x_centre, ", ", -x + y_centre,
                                ")", sep = "", end = "")
            print("(", -y + x_centre, ", ", -x + y_centre,
                                ")", sep = "")
if __name__ == '__main__':

    # 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)

II. RESULT AND DISCUSSION

DDA Algorithm (Digital Differential Analyzer)
Concept

- Utilizes floating-point arithmetic to calculate intermediate points between start and end points of a line.
- Steps are determined based on the greater absolute value of dx (change in x) or dy (change in y).

Advantages

- Simple and easy to implement.
- Handles diagonal and non-axis-aligned lines efficiently.

Disadvantages

- Computationally expensive due to floating-point calculations.
- Rounding errors can affect accuracy, leading to potential visual artifacts.

Bresenham's Line-Drawing Algorithm
Concept

- An improvement over the DDA algorithm, it uses integer arithmetic to decide the next pixel.
- Based on error calculation: the decision to plot a pixel depends on the cumulative error of the actual line versus the approximated path.

Advantages

- Faster than DDA due to integer operations.
- Highly efficient for raster displays as it avoids floating-point calculations.

Disadvantages

- Slightly more complex to implement.
- Primarily designed for straight lines; modifications are needed for other shapes.

Midpoint Circle Algorithm
Concept

- Utilizes symmetry of a circle to calculate one-eighth of the circle, then reflects the points to complete it.
- Relies on incremental calculations of decision parameters to determine the next point.

Advantages

- Efficient for circle drawing due to symmetry exploitation.
- Integer-only operations make it computationally inexpensive.

Disadvantages

- Limited to circular shapes.
- Modifications are required for ellipses or other curved geometries.

Comparisons of:DDA Algorithm, Bresenham's Algorithm, Midpoint Circle Algorithm

Aspect	DDA Algorithm	Bresenham's Algorithm	Midpoint Circle Algorithm
Purpose	Line drawing	Line drawing	Circle drawing
Arithmetic Used	Floating-point	Integer	Integer
Efficiency	Moderate	High	High
Applications	General lines	Raster graphics lines	Raster graphics circles

III. CONCLUSION

The comparative study of the DDA algorithm, Bresenham's Line-Drawing Algorithm, and the Midpoint Circle Algorithm highlights differences in efficiency, accuracy, and use cases. DDA, while simple and easy to implement, is slower due to floating-point operations and prone to rounding errors. Bresenham's algorithm is faster, leveraging integer arithmetic for efficient and precise line rendering, making it ideal for real-time applications. The Midpoint Circle Algorithm is highly efficient for circular shapes, using symmetry and integer operations to minimize computations. Overall, Bresenham's and Midpoint algorithms outperform DDA in speed and accuracy, with each being well-suited for their respective tasks in graphics.

© 2024 IJSRET

REFERENCES

1. Hearn, D., & Baker, M. P. (2014). Computer Graphics with OpenGL. Pearson Education.
2. https://www.tutorialspoint.com
3. https://www.geeksforgeeks.org
4. Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1996). Computer Graphics: Principles and Practice. Addison-Wesley.
5. Python-based Graphics Algorithms - GitHub Repositories:
6. GitHub - Line Drawing Algorithms
7. https://matplotlib.org/stable/contents.html

© 2024 IJSRET