Drawing Line Algorithms

Computer Graphics

Dr. Akram Alsubari

Email: Akram.alsubari87@gmail.com

Introduction

to render a line on a pixel-based display, it is crucial to translate mathematical equations into pixel positions.

The DDA algorithm is particularly suitable when we need to draw lines on raster displays

Raster graphics represent images using pixels, so each point on the screen corresponds to a pixel.

The DDA algorithm helps in drawing straight lines between two endpoints by calculating intermediate pixel values.

DDA STEPS

- Compute $\Delta x = x_2 - x_1$, $\Delta y = y_2 - y_1$.
- Determine steps = max(|Δx|, |Δy|).
- Increment x by Δx/steps, y by Δy/steps.
- Plot rounded coordinates

Digital Differential Analyzer (DDA)

1

Step 1 – Input of two end points: x1, y1, x2, y2

2

Step 2 – Calculate the difference between two end points: dx = x2 - x1;dy = y2 - y1

3

Step 3 – Identify the number of steps. If dx > dy, then you need more steps in x coordinate; otherwise in y coordinate. steps = max(abs(dx), abs(dy))

4

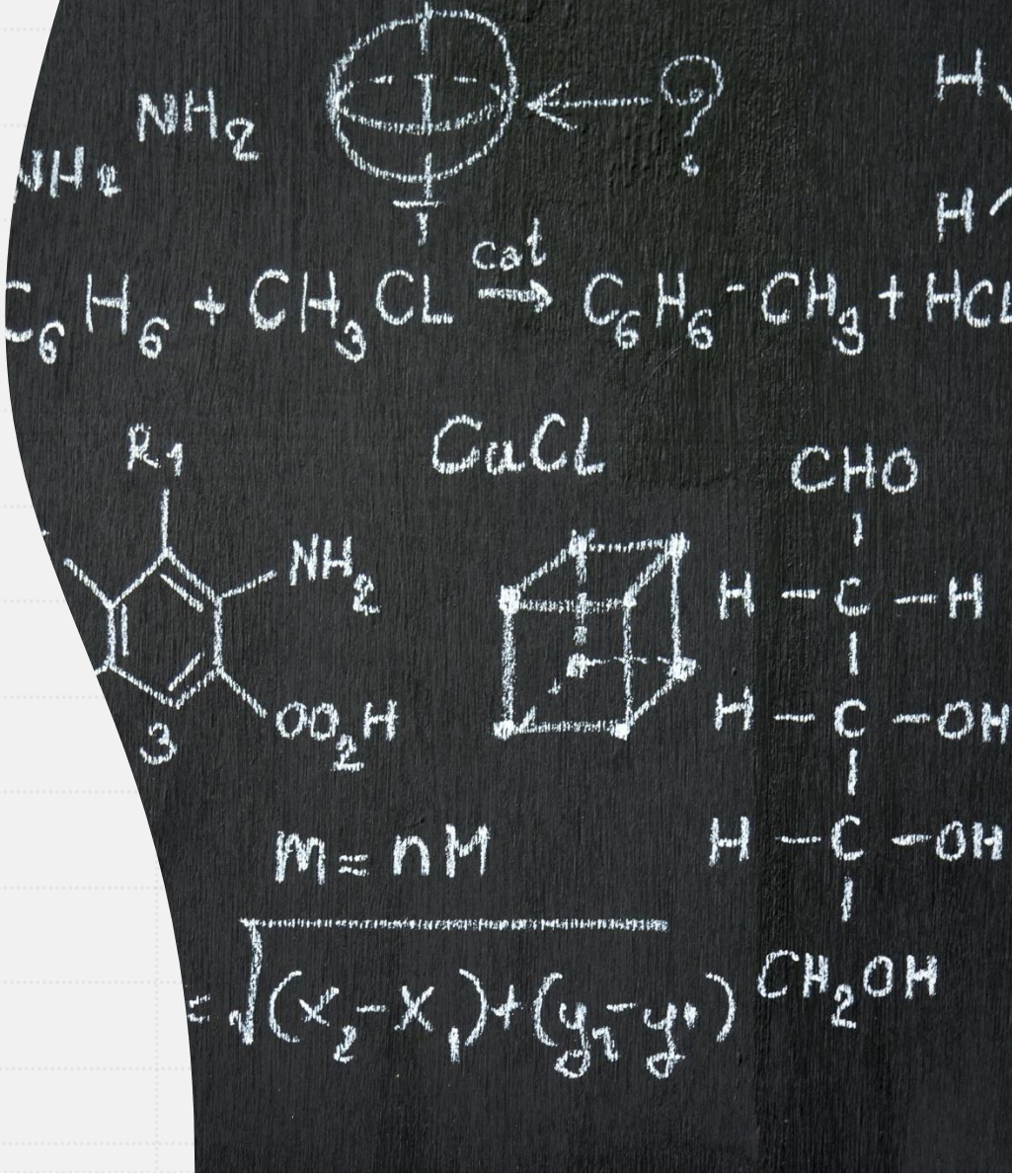
Step 4 – Calculate the increment in x coordinate and y coordinate. x_inc = dx / steps ; y_inc = dy / steps

5

Step 5 – Put the pixel by successfully incrementing x and y coordinates accordingly. Repeat step-5 till it complete the drawing of the line.

DDA Example

- Draw line between points (0,0) to (5,3) using DDA Algorithm
- Δx=5, Δy=3 → steps=5.
- x_increment=1, y_increment=0.6.
- Points: (0,0), (1,1), (2,1), (3,2), (4,2), (5,3).

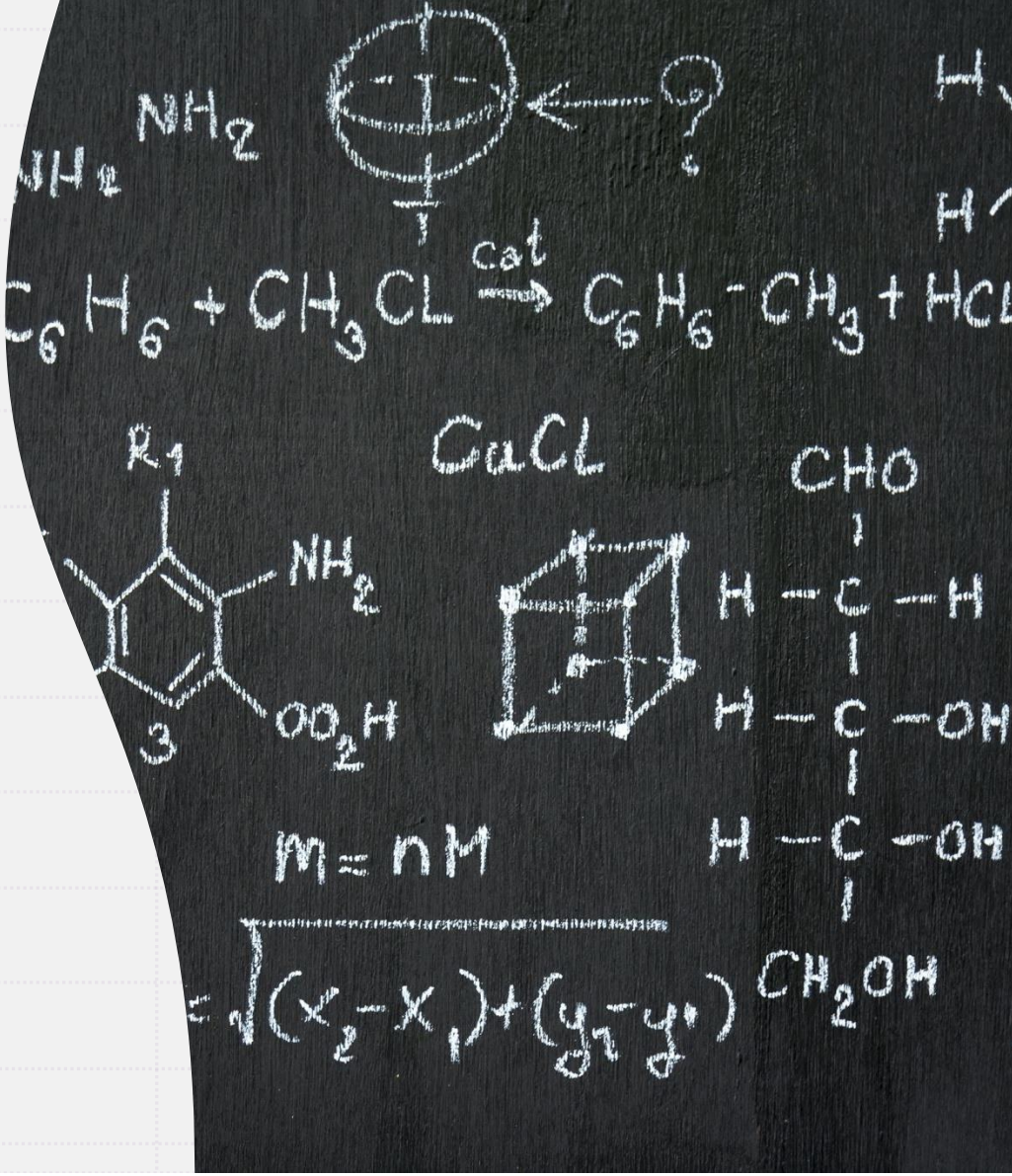


Disadvantages of the DDA Algorithm

- Slow: Floating-point operations slow for large lines.
- computationally expensive: due to floating-point operations tend to be more computationally expensive than integer operations.
- Rounding operations can introduce small errors

DDA Algorithm Python Code

```python
def dda(x1, y1, x2, y2):
    points = []
    dx = x2 - x1
    dy = y2 - y1
    steps = max(abs(dx), abs(dy))
    if steps == 0:  # Same start and end poi
        return [(x1, y1)]
    x_inc = dx / steps
    y_inc = dy / steps
    x, y = x1, y1
    for _ in range(steps + 1):
        points.append((round(x), round(y)))
        x += x_inc
        y += y_inc
    return points
```



Bresenham's Algorithm

Calculate Δx, Δy.

Initialize decision parameter: ($p = 2\Delta y - \Delta x$).

Update (p): - If ($p \geq 0$): ($p \mathrel{+}= 2(\Delta y - \Delta x)$), increment y. - Else: ($p \mathrel{+}= 2\Delta y$).

Increment x, repeat until $x = x_2$.

....

- Decision Parameter: Tracks the error between the ideal line and the chosen pixel
- Uses integer arithmetic to avoid floating-point operations
- Use a decision parameter to choose between `E` (horizontal) or `NE` (diagonal) moves

Bresenham Implementation Code

```
36  def bresenham_simple(x1, y1, x2, y2):
37      points = []
38      dx = x2 - x1
39      dy = y2 - y1
40      p = (2 * dy) - dx
41      y = y1
42      print(' bresenham_simple\n')
43      print('X\tY\tdecision\tNext')
44      for x in range(x1, x2 + 1):
45          # Append current (x, y) FIRST
46          points.append((x, y))
47          print(x,'\t', y,'\t', p, end='\t')
48          # Update p and y for NEXT iteration
49          if p < 0:
50              print('p=p+(2*dy)')
51              p += 2 * dy
52          else:
53              print('p=p+(2*dy)-(2*dx)')
54              p += (2 * dy) - (2 * dx)
55              y += 1  # Increment AFTER appending
56      return points
```

Mid-point Line Generation Algorithm

- The main idea of the Mid-Point Line Drawing Algorithm is to find the best pixels to make a straight line.
- Decision Parameter: Checks if the midpoint between two candidate pixels is above/below the line.

  d = dy - dx/2 (midpoint evaluation)
- Update Rule: d += dy` or `(dy - dx)`
- Same output as Bresenham’s