Midpoint Line Algorithm Examples

Like DDA, we will be running a loop to start from (x1, y1) to reach (x2, y2) while calculating the intermediary pixel coordinates on the way. We start at the starting point, and continuously jump to the current pixel's neighbor and go on like this. Which neighbor to choose is determined by the decision variable called "d". In the basic version of this algorithm (the version that we derived - derivation given in slides), we can only jump to the E (east), and north-east (NE) neighbor. This limits our ability to draw lines that only go upwards and rightwards and is less slopey (m < 1).

Midpoint Line Drawing Algorithm (Basic Version)

```
Procedure MidpointLine(x1, y1, x2, y2)
    dx ← x2 - x1
    dy ← y2 - y1

    incE  ← 2 * dy         // increment when moving to east pixel
    incNE ← 2 * (dy - dx) // increment when moving to north-east pixel

    d ← 2 * dy - dx        // decision variable

    x ← x1
    y ← y1

    SetPixel(x, y)

    While x < x2 Do
        If d <= 0 Then
            // Move to east pixel
            d ← d + incE
            x ← x + 1
        Else
            // Move to north-east pixel
            d ← d + incNE
            x ← x + 1
            y ← y + 1
        End If

        SetPixel(x, y)
    End While
End Procedure
```

Example: Calculate the pixels of a line segment that goes from (10, 10) to (20, 18)
`x1 = 10, y1 = 10,`
`x2 = 20, y2 = 18`

Calculate the initial values needed (apply formulas):

`dx = (x2 - x1) = (20 - 10) = 10`
`dy = (y2 - y1) = (18 - 10) = 8`

$d_{init} = 6$
$\Delta NE = -4$
$\Delta E = 16$

Now, make the table:

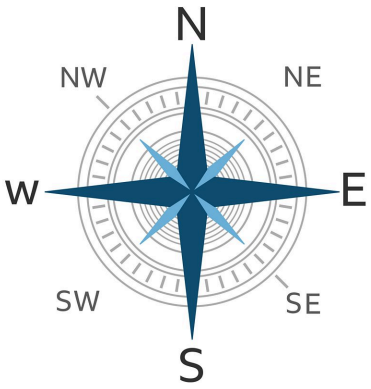
x	y	d	neighbor	d (new)
10	10	6 (dinit)	NE	2
11	11	2	NE	-2
12	12	-2	E	14
13	12	14	NE	10
14	13	10	NE	6
15	14	6	NE	2
16	15	2	NE	-2
17	16	-2	E	14
18	16	14	NE	10
19	17	10	NE	6
20	18	-	-	-

That's the answer.

But what if the line might go in some other direction, like a line that goes leftwards and/or downwards, what if the line has slope that's greater than 1? In this case, our basic MPL algo won't draw the correct line. We, rather, enhance that algorithm with **8-way symmetry**.

Understanding Zones:

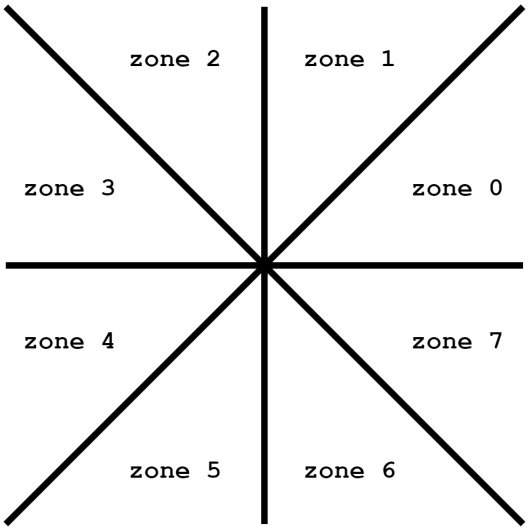
So, a line's direction is the direction where the line goes from the starting point to the ending point. This follows much like a compass. There are 8 directions in a compass.



In the basic version of MPL, we could go only towards between N and NE (thus choosing either E neighbor or NE neighbor). But a line can go at any direction.

We group all possible directions into 8 groups, called zones. These are:

Zone	Direction
0	Line goes at a direction between E and NE (our algorithm works in this zone)
1	Line goes at a direction between NE and N
2	Line goes at a direction between N and NW
3	Line goes at a direction between NW and W
4	Line goes at a direction between W and SW
5	Line goes at a direction between SW and S
6	Line goes at a direction between S and SE
7	Line goes at a direction between SE and E



⚠️ Note: Zones are direction specific, and won't necessarily align with quadrants/octants. In fact, a line go encompass multiple quadrants/octants, but it will go in a single direction (for example between S and SW).

Anyway, back to 8-way symmetry.

8-way symmetry work on the principle of **finding** the zone of a line segment, then **converting** the endpoints (start and end coordinates) from the found zone to zone 0 (because our algo only works in zone 0), then **calculate** the pixels in zone 0 using our existing algo. But before plotting the pixels on screen, we need to **convert back** the calculated points back to their original zone.

So, the steps are:

1. Find the zone of the line, we label the zone as m.
2. Convert the endpoints (start coordinate and end coordinate) from zone-m to zone-0.
3. Using the converted coordinates, run the existing midpoint algorithm. And generate a set of points.
4. Convert the set of points back to their original zone (zone-0 to zone-m).

Please consult the pseudocode given below (also given in slides).

Midpoint Line Drawing Algorithm (8-Way Symmetry Version)

```
Procedure FindZone(x1, y1, x2, y2)
    dx ← x2 - x1
    dy ← y2 - y1

    If |dx| > |dy| Then
        If dx > 0 And dy > 0 Then
            Return 0
        Else If dx < 0 And dy > 0 Then
            Return 3
        Else If ??? Then
            Return ?
        Else If ??? Then
            Return ?
    Else
        If dx > 0 And dy > 0 Then
            Return 1
        Else If dx < 0 And dy > 0 Then
            Return 2
        Else If ??? Then
            Return ?
        Else If ??? Then
            Return ?
    End If
End Procedure


Procedure ConvertMtoZero(x, y, zone)
    If zone == 0 Then
        Return (x, y)
    Else If zone == 1 Then
        Return (y, x)
    Else If zone == 2 Then
        Return (y, -x)
    Else If zone == 3 Then
        Return (-x, y)
    Else If zone == 4 Then
        Return (?, ?)
    Else If zone == 5 Then
        Return (?, ?)
    Else If zone == 6 Then
        Return (?, ?)
```

```
    Else If zone == 7 Then
        Return (?, ?)
End Procedure


Procedure ConvertZeroToM(x, y, zone)
    If zone == 0 Then
        Return (x, y)
    Else If zone == 1 Then
        Return (y, x)
    Else If zone == 2 Then
        Return (-y, x)
    Else If zone == 3 Then
        Return (-x, y)
    Else If zone == 4 Then
        Return (?, ?)
    Else If zone == 5 Then
        Return (?, ?)
    Else If zone == 6 Then
        Return (?, ?)
    Else If zone == 7 Then
        Return (?, ?)
End Procedure


// Notice that a 5-th parameter got added, we need to pass the zone
Procedure MidpointLine(x1', y1', x2', y2', zone)
    // Note: you cannot reuse the dx and dy from zone finding
    // You need to recalculate since the points got changed/converted
    dx' ← x2' - x1'
    dy' ← y2' - y1'

    incE  ← 2 * dy'         // increment for moving to east pixel
    incNE ← 2 * (dy' - dx') // increment for moving to north-east pixel

    d ← 2 * dy' - dx'       // decision variable

    x ← x1'
    y ← y1'

    cx, cy ← ConvertZeroToM(x, y, zone) // convert pixel to original zone
    SetPixel(cx, cy) // draw the converted pixel
```

```
    While x < x2' Do
        If d <= 0 Then
            d ← d + incE
            x ← x + 1
        Else
            d ← d + incNE
            x ← x + 1
            y ← y + 1
        End If

        cx, cy ← ConvertZeroToM(x, y, zone) // convert pixel to original zone
        SetPixel(cx, cy) // draw the converted pixel
    End While
End Procedure


Procedure MidpointLineEightway(x1, y1, x2, y2)
    zone ← FindZone(x1, y1, x2, y2)          // find the zone
    x1', y1' ← ConvertMtoZero(x1, y1, zone) // convert start point to zone 0
    x2', y2' ← ConvertMtoZero(x2, y2, zone) // convert end point to zone 0
    MidpointLine(x1', y1', x2', y2', zone)  // draw the line in zone 0 and
convert each point to original zone
End Procedure
```

Example: Calculate the pixels of a line segment that goes from (5, 0) to (-3, 10)

x1 = 5, y1 = 0,
x2 = -3, y2 = 10

Step 1: Find the zone

Using the **`FindZone`** function, we find that the line belongs in zone 2. Because:

dx = -8
dy = 10

|dx| < |dy| and **dx < 0** and **dy > 0**. So, zone = **2**

Step 2: Convert the end points (zone-m to zone-0)

Using the **`ConvertMtoZero`** function, we can do it for starting coordinate and ending coordinate. These give us new coordinates.

x1' = 0, y1' = -5,
x2' = 10, y2' = 3

Step 3 & 4: Using the new coordinates, run the MPL algorithm. And convert back (zone-0 to zone-m)

dx' = (x2' - x1') = (10 - 0)   = 10
dy' = (y2' - y1') = (3 - (-5)) = 8

Very Important!!

Note that we cannot reuse the dx and dy from the zone finding step, as the points got changed. We therefore recalculated them. Using the new dx' and dy' values, calculate the following:

d_{init} = 6
ΔNE = -4
ΔE   = 16

Now, make the table:

x'	y'	d	neighbor	d (new)	Converted Back (Step-4)	
					x	y
0	-5	6 (dinit)	NE	2	5	0
1	-4	2	NE	-2	4	1
2	-3	-2	E	14	3	2
3	-3	14	NE	10	3	3
4	-2	10	NE	6	2	4
5	-1	6	NE	2	1	5
6	0	2	NE	-2	0	6
7	1	-2	E	14	-1	7
8	1	14	NE	10	-1	8
9	2	10	NE	6	-2	9
10	3	-	-	-	-3	10

Notice how we performed step-4 (converting zone-0 to zone-m) in the same table. This is convenient. For step-4, use the **`ConvertZeroToM`** function.

That's the answer.