Rasterization

Foley, Ch: 3

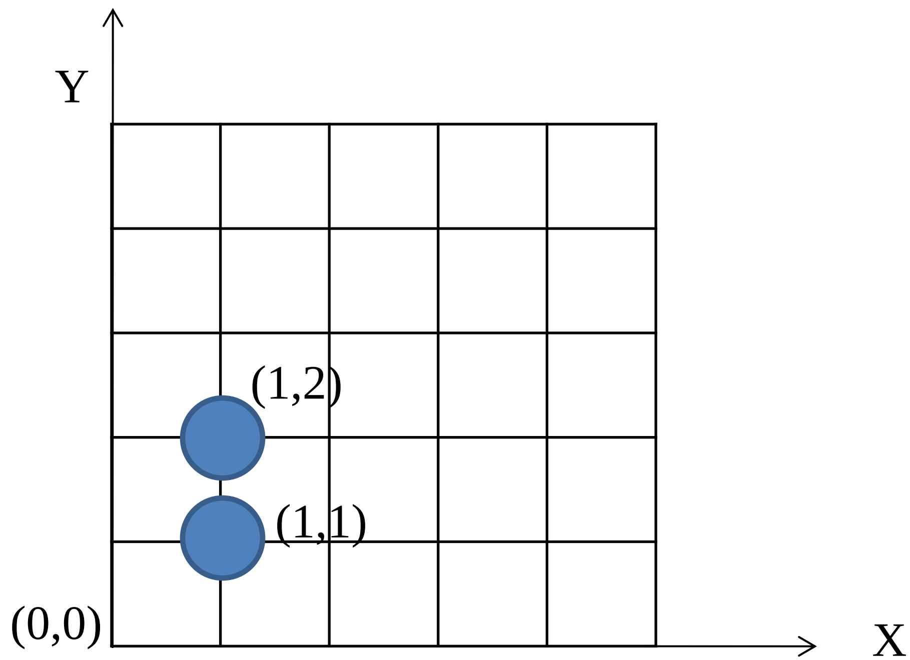
Upto 3.1, 3.2 (upto 3.2.3), 3.3, 3.6 (upto 3.6.3)

Overview

- Scan conversion : mapping objects/shapes to pixels
- Fast image generation (e.g. in computer games)
- Simple Raster Graphics Package (SRGP)
- Scan converting lines
- Scan converting circles
- Scan converting polygons

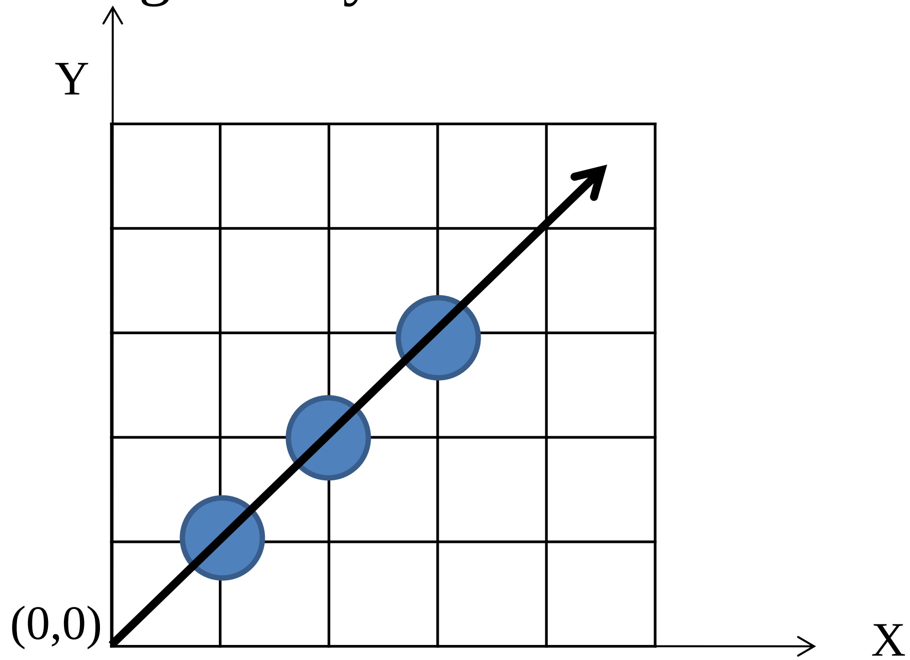
Pixels in SRGP

- Pixels are represented as circles centered on uniform grid
- Each (x,y) of the grid has a pixel



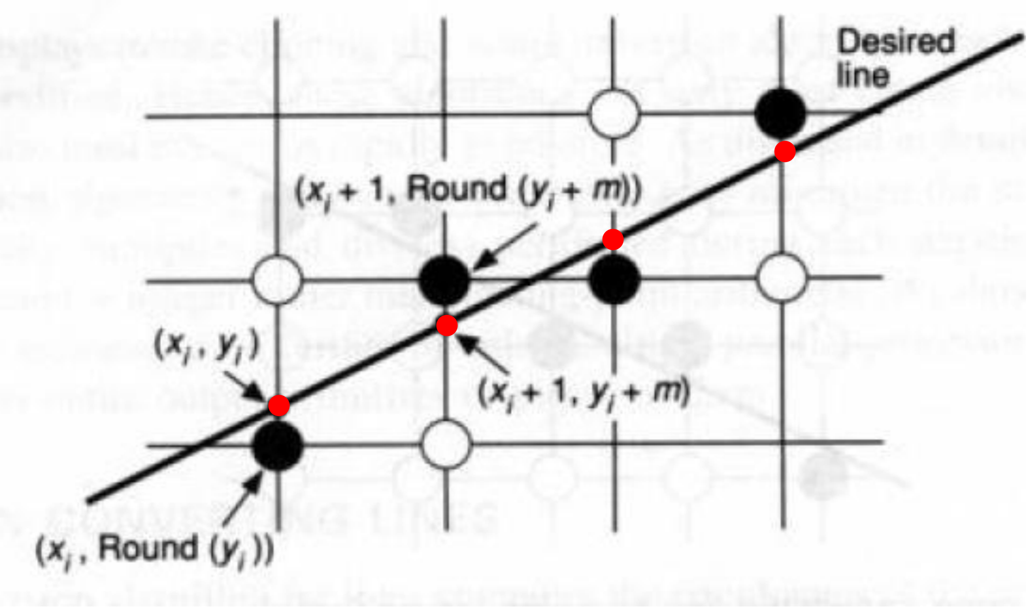
Scan Converting Lines

- If the line has slope, m = 1
- Incremental along x and y axis is 1



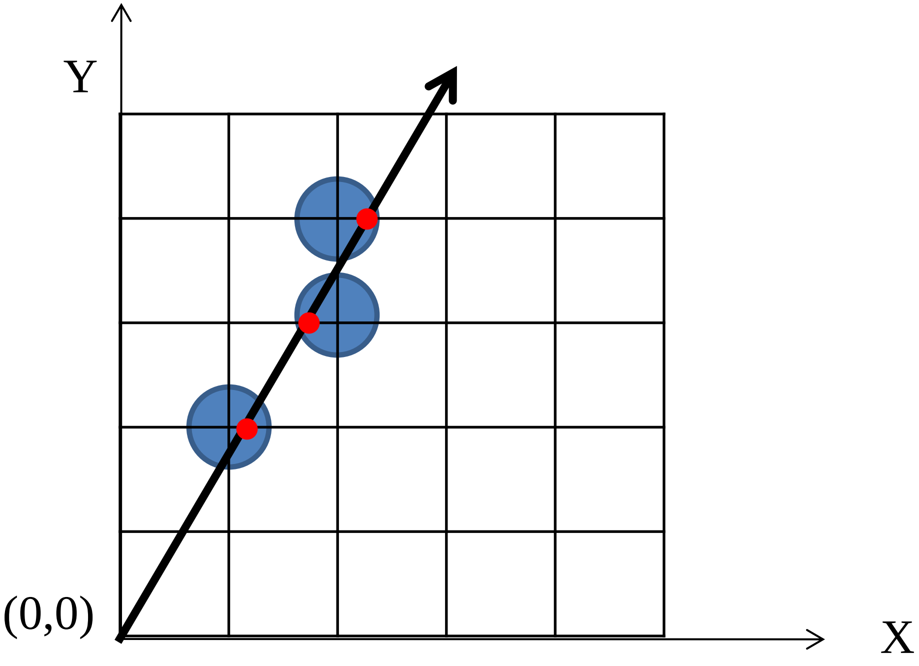
Scan Converting Lines

- If the line has slope < 1
- Increment along x is 1, increment along y is fractional
  - Round the value along y axis



Scan Converting Lines

- If the line has slope > 1
- Increment along y is 1, increment along x is fractional
  - Round the value along y axis



Scan Converting Lines : Basic Incremental Algorithm

Two end points (x_o,y_o) and (x_1,y_1)

Calculate slope $m = (y_1-y_0) / (x_1-x_0)$

<u>If (m<1):</u>

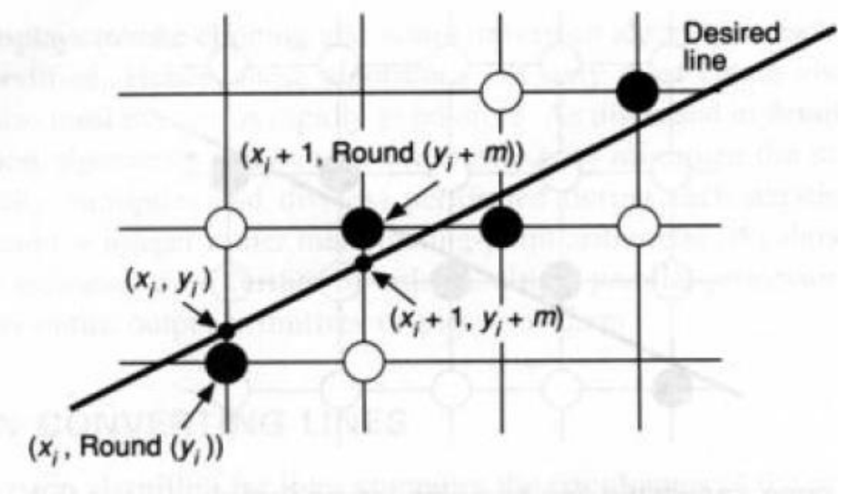
$y_{i+1} = mx_{i+1} + B$

$= m(x_i + \Delta x) + B$

$= (mx_i + B) + m\Delta x$

$= y_i + m\Delta x$

$= y_i + m$



So next point to intensify is $(x_{i+1}, \text{round}(y_{i+1}))$ where $x_{i+1}= x_i+1$

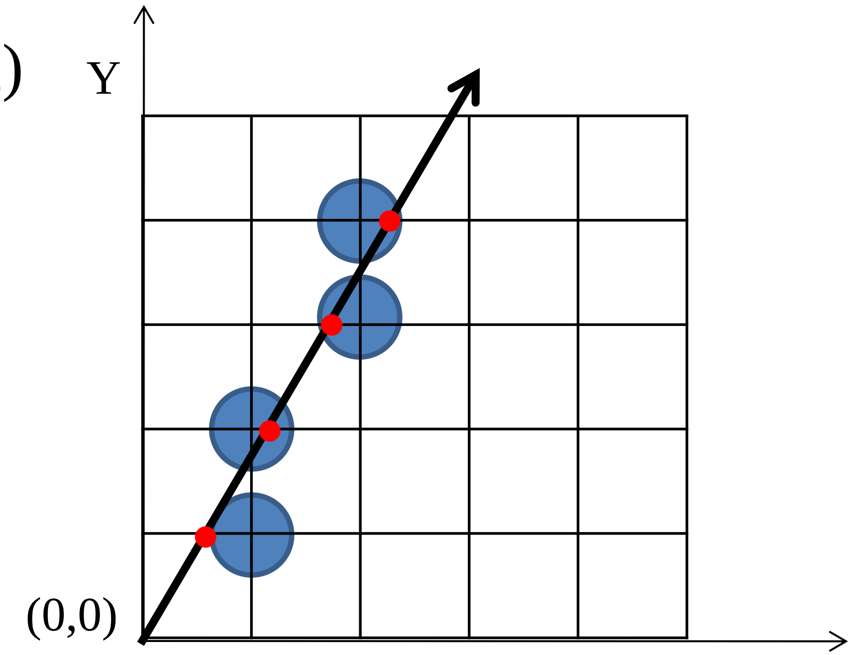
Scan Converting Lines : Basic Incremental Algorithm

Two end points (x_o,y_o) and (x_1,y_1)

Calculate slope $m = (y_1-y_0) / (x_1-x_0)$

<u>If(m>1) (Steeper):</u>

$$x_{i+1} = (y_{i+1} – B)/m$$
$$= ((y_i + \Delta y) – B)/m$$
$$= (y_i - B)/m + \Delta y/m$$
$$= x_i + 1/m$$



So next point to intensify is $(round(x_{i+1}), y_{i+1})$ where $y_{i+1}= y_i+1$

Scan Converting Lines : Basic Incremental Algorithm

Basic Incremental Algorithm Problems:

1. Handle fractional values → Rounding takes time
2. Working with real numbers (e.g. y, m)

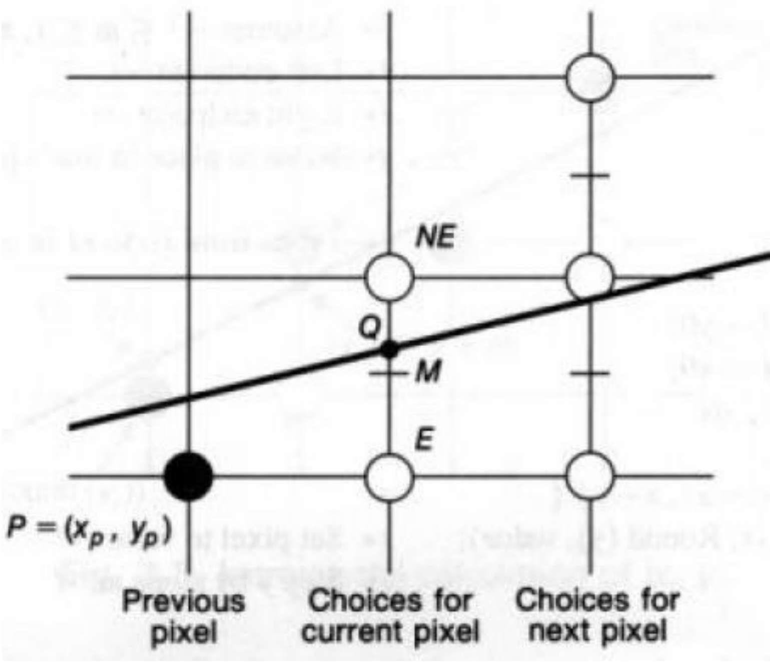
… etc.

Solution: **Midpoint Line Algorithm**

1. Only integer arithmetic
2. Avoid rounding

Scan Converting Lines : Midpoint Line Algorithm

- Assumption : Slope is between 0 and 1
- Lower end point (x_o,y_o) and upper point (x_1,y_1)
- Initially we are at $P(x_p,y_p)$
- Choose between **NE** and **E**
- Midpoint **M** is $\mathbf{(x_p+1, y_p+1/2)}$
- **M lies on which side of the line?**
  - M is on the line → any pixel NE / **E**
  - M is below the line → choose NE
  - M is above the line → choose E

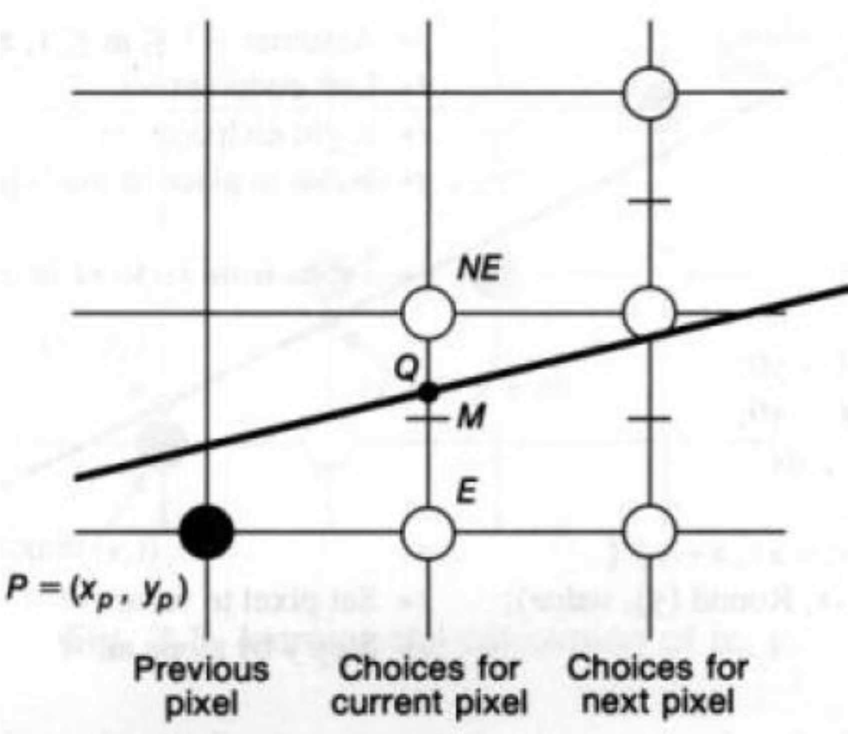


- **Find the equation of the line f (x,y)**
  - f (x,y) = ax + by + c
    a = dy; b = -dx; c = dx.B (B is the y-intercept in the slope-intercept form)
  - Let d = f (M) = f $(x_p+1, y_p+1/2)$
  - d = 0 → M is on the line
    d > 0 → M is below the line
    d < 0 → M is above the line

Scan Converting Lines : Midpoint Line Algorithm

- Find the next value of the decision variable d?
  - Need to apply incremental approach

- Two cases:
  1. If E is chosen
  2. If NE is chosen

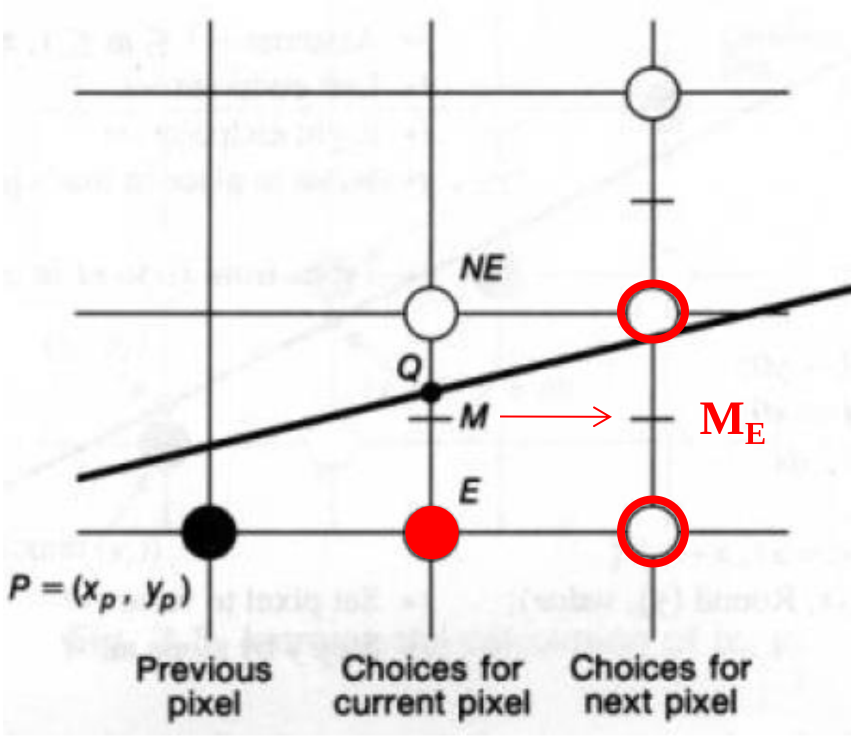


Scan Converting Lines : Midpoint Line Algorithm

- Find the next value of the decision variable d?
  - Need to apply incremental approach

If E is chosen :

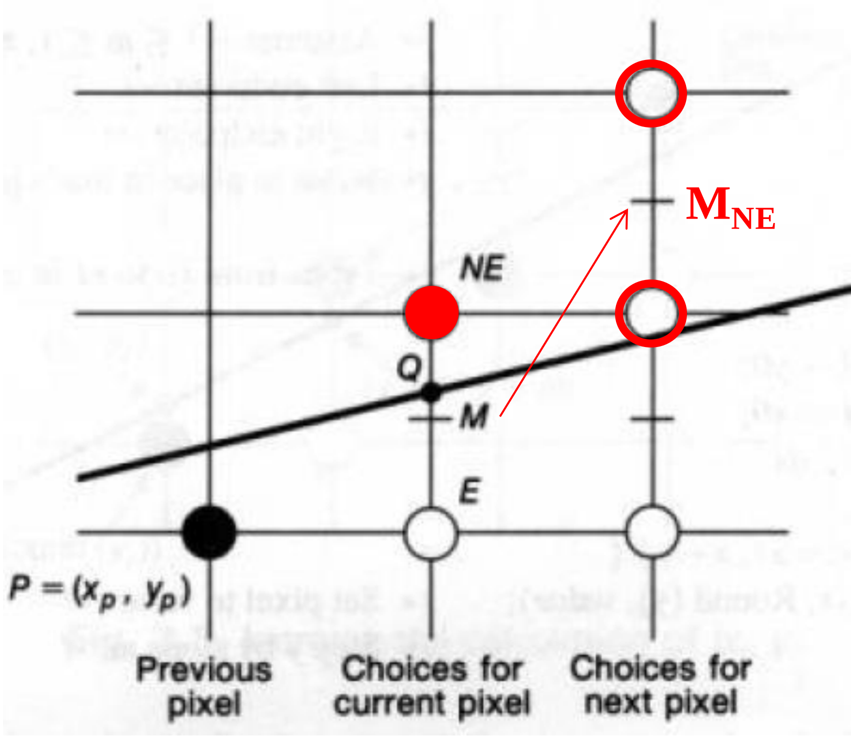
- New Midpoint:
  $M_E\ (x_p+2,\ y_p+1/2)$
- New decision variable, d_{new}
  $= f\ (M_E) = f\ (x_p+2,\ y_p+1/2)$
  $= f\ (x_p+1,\ y_p+1/2) + dy.1$
  $= d_{old} + dy$



So, $d_E = dy$

Scan Converting Lines : Midpoint Line Algorithm

- Find the next value of the decision variable d?
  - Need to apply incremental approach



If NE is chosen :

- New Midpoint:
  M_{NE} $(x_p+2, y_p+ 3/2)$
- New decision variable d_{new}
  $= f\ (M_{NE}) = f(x_p+2, y_p+ 3/2)$
  $= f\ (x_p+1, y_p+ 1/2) + dy.1- dx.1$
  $= d_{old} + dy – dx$

So, $d_{NE} = dy - dx$

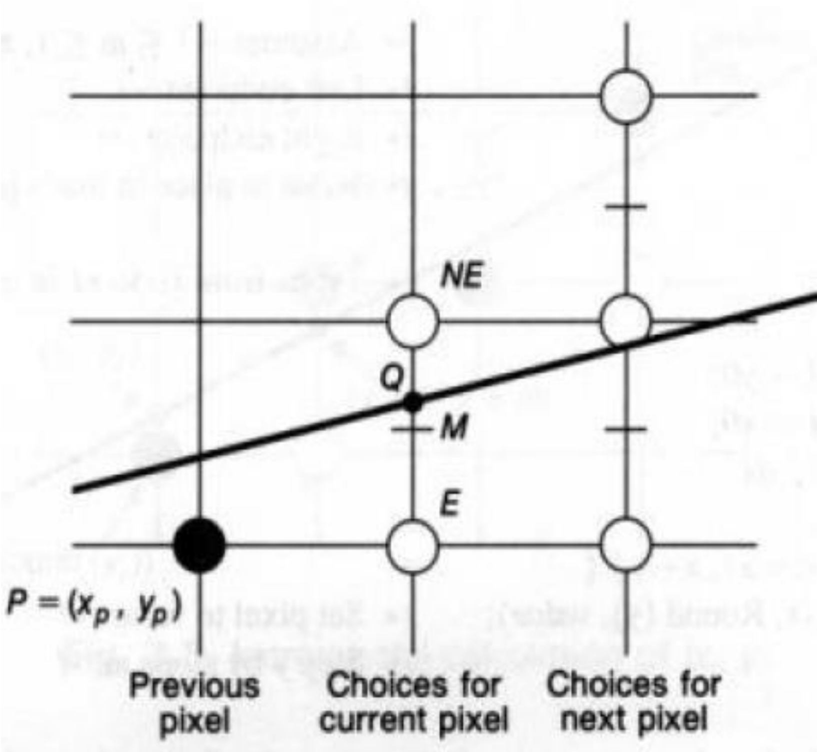
Scan Converting Lines : Midpoint Line Algorithm

- Initial value of decision variable $\mathbf{d_{start}}$
- Start point (x_o, y_o)
- $d_{start} = f(x_o+1,\ y_o + 1/2)$

$$= f(x_o, y_o) + dy - dx/2$$

$$= dy - dx/2$$

Here the fraction in **dx/2** can be avoided considering **f(x,y) = 2(ax+by+c)**



```
void MidpointLine (int x0, int y0, int x1, int y1, int value)
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx;            /* Initial value of d */
    int incrE = 2 * dy;             /* Increment used for move to E */
    int incrNE = 2 * (dy - dx);     /* Increment used for move to NE */
    int x = x0;
    int y = y0;
    WritePixel (x, y, value);       /* The start pixel */

    while (x < x1) {
        if (d <= 0) {               /* Choose E */
            d += incrE;
            x++;
        } else {                    /* Choose NE */
            d += incrNE;
            x++;
            y++;
        }
        WritePixel (x, y, value);   /* The selected pixel closest to the line */
    } /* while */

} /* MidpointLine */
```

Simple addition and comparison

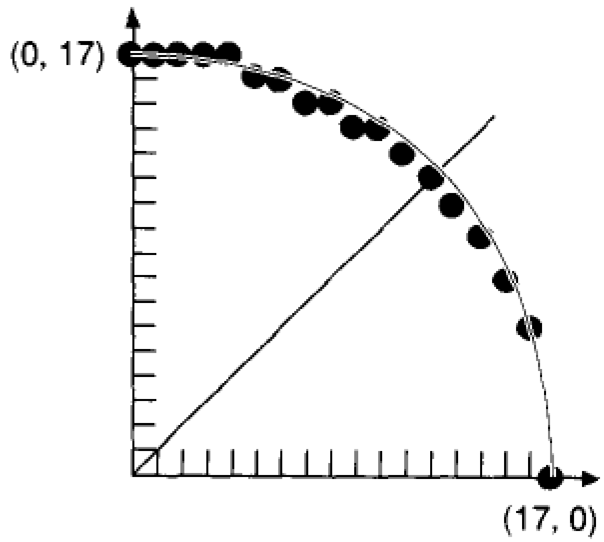
Fig. 3.8 The midpoint line scan-conversion algorithm.

Scan Converting Lines : Midpoint Line Algorithm

- The algorithm works for 0<=m<=1
- What about other slopes?
  - Think!

Scan Converting Circles

- Naive Approach (expensive computation)
  - For a circle centered at the origin, $y = \sqrt{(R^2 - x^2)}$
  - Draw a quarter by incrementing x from 0 to R in unit steps



Large gaps for values of x close to R

  - Circles not centered at the origin, can be translated for computation and pixels can be written with appropriate offsets

Scan Converting Circles

- Another Inefficient Method
  - For a circle centered at the origin, $y = R\cos\Theta$, $x = R\sin\Theta$
  - Vary Θ from 0 to 90 degree uniformly and plot x, y
  - Avoids large gaps but still computationally inefficient

- Eight Way Symmetry

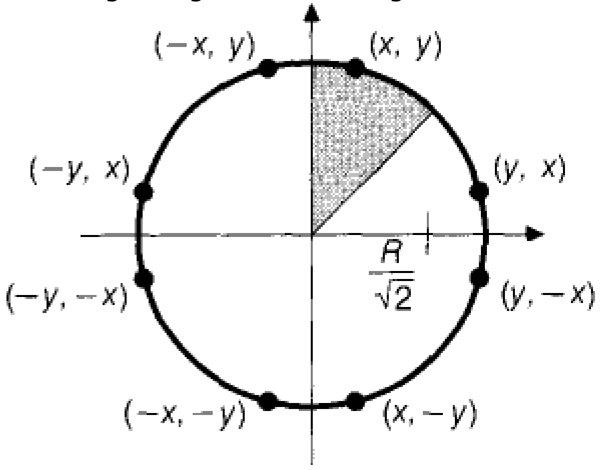
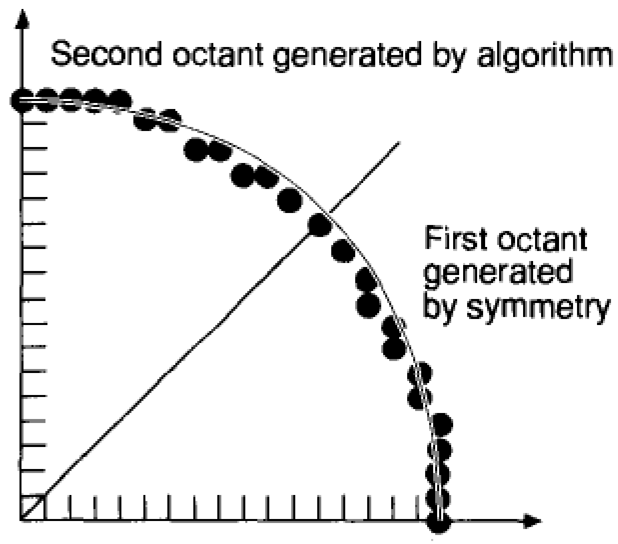
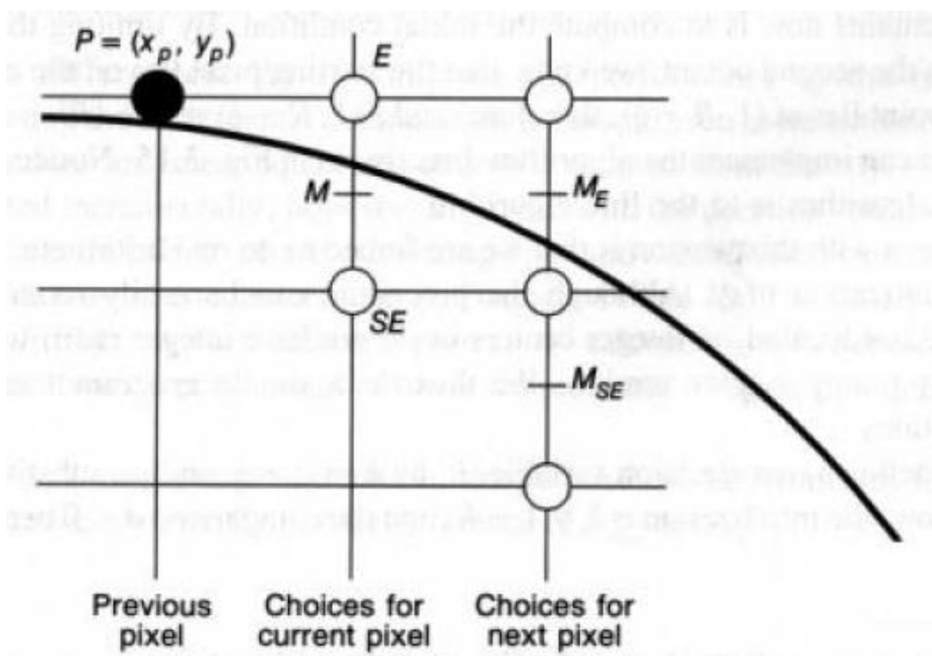


Fig. 3.13 Eight symmetrical points on a circle.



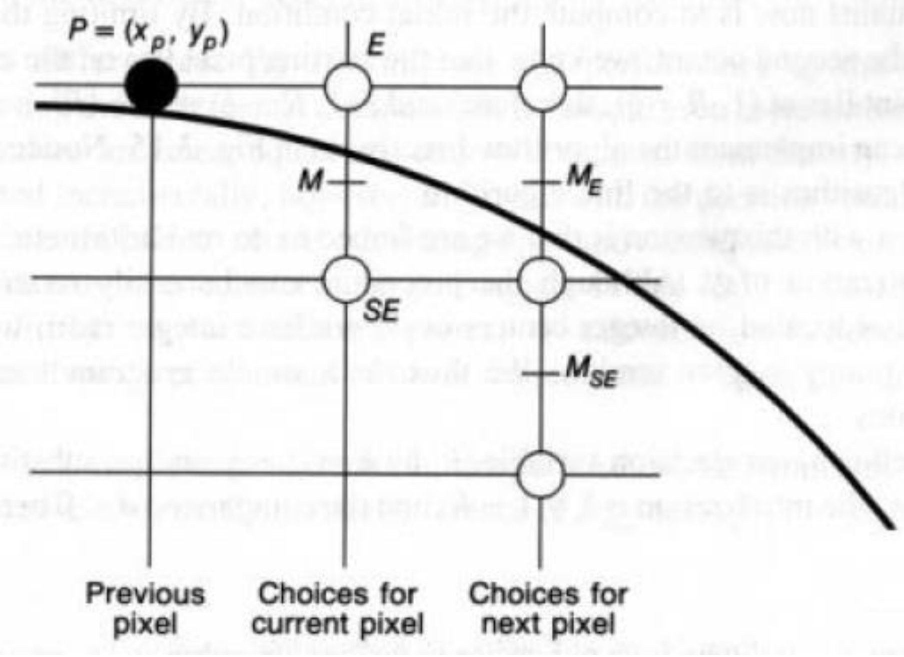
Scan Converting Circles : Midpoint Circle Algorithm

- Compute for the second octant only i.e. 45^0 of the circle, from $x = 0$ to $x = R/\sqrt{2}$
- Say, currently we are at $P(x_p, y_p)$.
- Choose between E or SE based on midpoint M$\mathbf{(x_p+1, y_p - 1/2)}$
- **M lies inside or outside the circle?**
  - M is outside the circle → choose SE
  - M is inside the circle → choose E
  - M is on the circle → any pixel SE /E



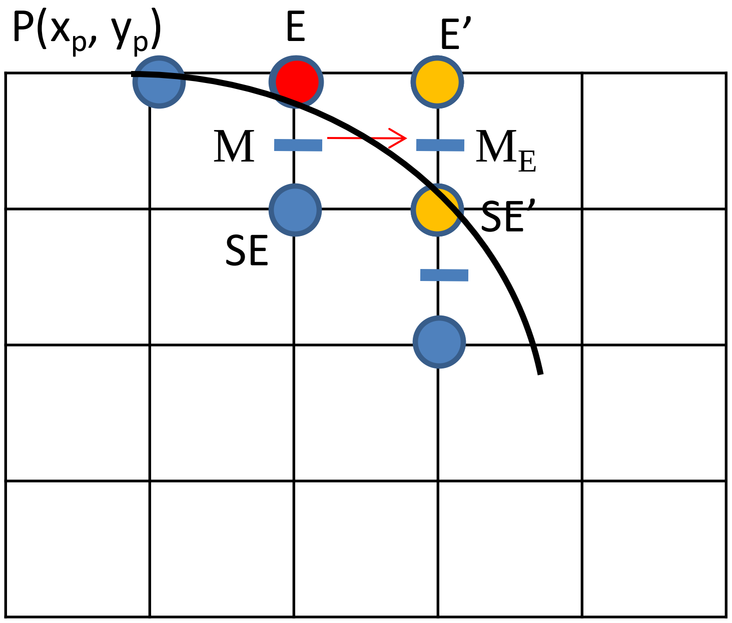
Scan Converting Circles : Midpoint Circle Algorithm

- Use $\mathbf{f(x,y) = x^2 + y^2 - R^2}$
- $d = f(M) = f(\mathbf{x_p+1, y_p - 1/2})$
- $d = 0$: M is on the circle
  → any pixel SE/E
- $d > 0$: M is outside the circle
  → choose SE
- $d < 0$: M is inside the circle
  → choose E



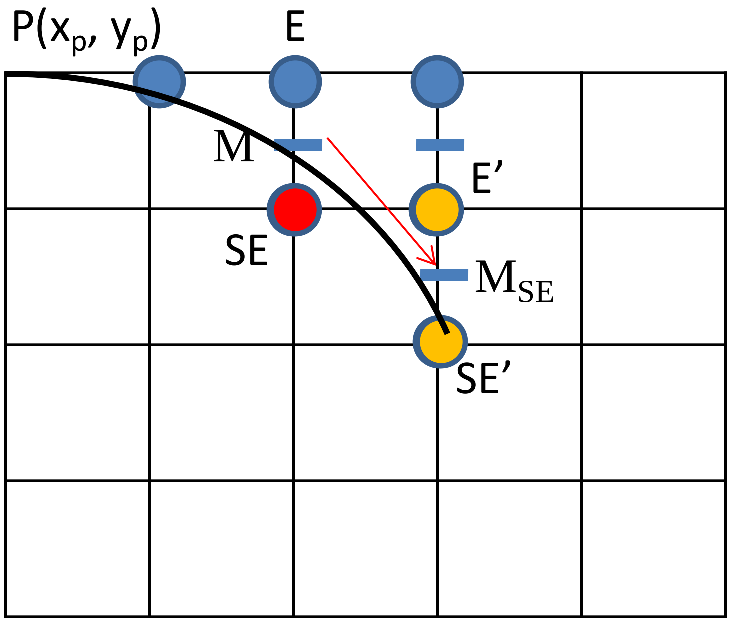
Scan Converting Circles : Midpoint Circle Algorithm

- After we reach E or SE, what is the new value of d?
- **<u>Case 1-We are at E:</u>**
- New midpoint $M_E(\mathbf{x_p+2, y_p - 1/2})$
- $d_{new} = f(M_E) = f(\mathbf{x_p+2, y_p - 1/2})$
  $= d_{old} + (\mathbf{2x_p + 3})$
  $= d_{old} + \mathbf{\Delta E}$
- $d_{new} > 0 \rightarrow$ M is outside the circle
  $\rightarrow$ Choose SE'
- $d_{new} < 0 \rightarrow$ M is inside the circle
  $\rightarrow$ Choose E'
- $d_{new} = 0 \rightarrow$ M is on the circle
  $\rightarrow$ Choose E' / SE'



Scan Converting Circles : Midpoint Circle Algorithm

- After we reach E or SE, what is the new value of d?
- **<u>Case 2-We are at SE:</u>**
- New midpoint $M_{SE}(\mathbf{x_p+2, y_p - 3/2})$
- $d_{new} = f\,(M_{SE}) = f(\mathbf{x_p+2, y_p - 3/2})$
  $= d_{old} + (\mathbf{2x_p - 2y_p + 5})$
  $= d_{old} + \mathbf{\Delta SE}$
- $d_{new} > 0$ → M is outside the circle
  → Choose SE'
- $d_{new} < 0$ → M is inside the circle
  → Choose E'
- $d_{new} = 0$ → M is on the circle
  → Choose E' / SE'



Scan Converting Circles : Midpoint Circle Algorithm

- Initial Value of d:

  $d_{start} = f(1, R-½)$
  $= 1^2 + (R-½)^2 - R^2$
  $= 1 + R^2 – 2.R. ½ + (½)^2$
  $– R^2$
  $= 1 + 1/4 – R$
  $= 5/4 – R$

```
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    double d = 5.0 / 4.0 – radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)            /* Select E */
            d += 2.0 * x + 3.0;
        else {                /* Select SE */
            d += 2.0 * (x – y) + 5.0;
            y––;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

```
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    double d = 5.0 / 4.0 - radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)              /* Select E */
            d += 2.0 * x + 3.0;
        else {                  /* Select SE */
            d += 2.0 * (x - y) + 5.0;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

```
void MidpointLine (int x0, int y0, int x1, int y1, int value)
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx;
    int incrE = 2 * dy;
    int incrNE = 2 * (dy - dx);
    int x = x0;
    int y = y0;
    WritePixel (x, y, value);

    while (x < x1) {
        if (d <= 0) {
            d += incrE;
            x++;
        } else {
            d += incrNE;
            x++;
            y++;
        }
        WritePixel (x, y, value);
    } /* while */

} /* MidpointLine */
```

Scan Converting Circles : Midpoint Circle Algorithm

- Initial Value of d:
- $d_{start} = 5/4 - R$
- Say, $h = d - 1/4$
  $d = h + 1/4$
- $h + 1/4 = 5/4 - R$
  $h = 1 - R$
- $h = -1/4 \rightarrow$ M is on the circle
  $\rightarrow$ any pixel SE/E
  $h > -1/4 \rightarrow$ M is outside the circle
  $\rightarrow$ choose SE
  $h < -1/4 \rightarrow$ M is inside the circle
  $\rightarrow$ choose E

If h starts out with integer value and gets incremented by integer value (ΔE or ΔSE) then the comparisons, $h = -1/4$, $h > -1/4$, $h < -1/4$ reduce to $h=0$, $h>0$, and $h<0$

Scan Converting Circles : Midpoint Circle Algorithm

```
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    double d = 5.0 / 4.0 – radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)            /* Select E */
            d += 2.0 * x + 3.0;
        else {                /* Select SE */
            d += 2.0 * (x – y) + 5.0;
            y−−;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

```
procedure MidpointCircle (radius, value: integer);
{Assumes center of circle is at origin. Integer arithmetic
var
    x, y, d : integer;
begin
    x := 0;                       {Initialization}
    y := radius;
    d := 1 – radius;
    CirclePoints (x, y, value);

    while y > x do
      begin
        if d < 0  then            {Select E}
          begin
            d := d + 2 * x + 3;
            x := x + 1
          end
        else
          begin                   {Select SE}
            d := d + 2 * (x – y) + 5;
            x := x + 1;
            y := y – 1
          end;
        CirclePoints (x, y, value)
      end  {while}
end;  {MidpointCircle}
```

Replacing h by d

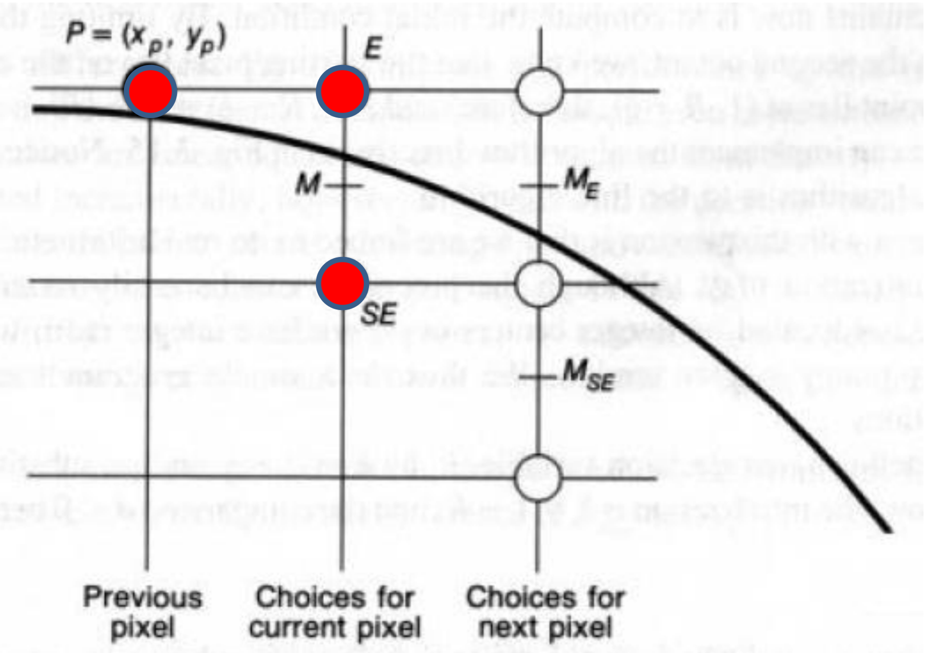
Scan Converting Circles : Midpoint Circle Algorithm

Improvements by second order difference:

- $\Delta E = 2x_p + 3$
- $\Delta SE = 2x_p - 2y_p + 5$

From the point of evaluation, $P(x_p, y_p)$

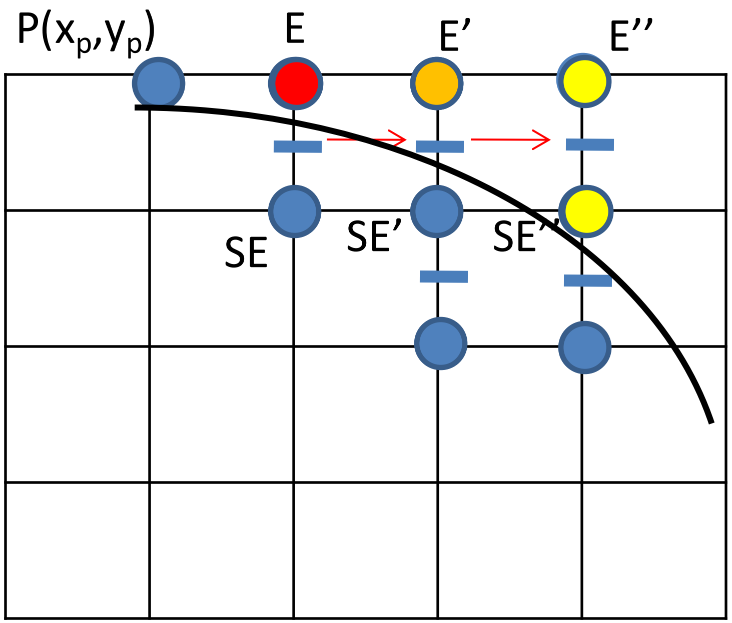
- E is chosen → new point of evaluation is $E(x_p+1, y_p)$
- SE is chosen→ new point of evaluation is $SE(x_p+1, y_p-1)$
- Find increments ΔE and ΔSE wrt new point of evaluation, $E(x_p+1, y_p)$, $SE(x_p+1, y_p-1)$



Second order difference wrt $E(x_p+1,y_p)$

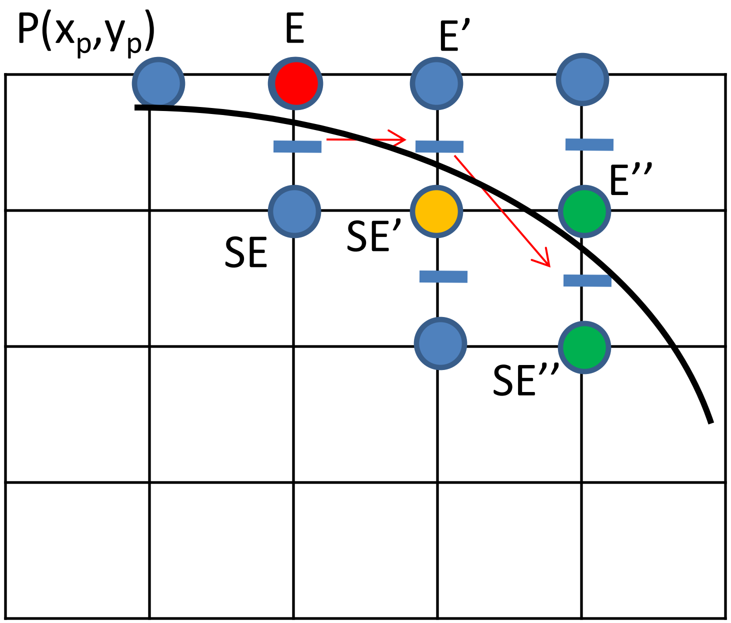
- $P(x_p,y_p)$ → $\mathbf{\Delta E= 2x_p + 3 \;\; = \Delta E_{old}}$ Increment in **d** when E is chosen
  → $\mathbf{\Delta SE= 2x_p - 2y_p + 5= \Delta SE_{old}}$ Increment in **d** when E is chosen

- $E(x_p+1,y_p)$ → $\mathbf{\Delta E= 2(x_p + 1) + 3 = (2x_p + 3) + 2 = \Delta E_{old} \;\; + 2}$



Second order difference wrt $E(x_p+1,y_p)$

- $P(x_p,y_p)$
  - $\Delta E = 2x_p + 3 = \Delta E_{old}$ Increment in **d** when E is chosen
  - $\Delta SE = 2x_p - 2y_p + 5 = \Delta SE_{old}$ Increment in **d** when E is chosen
- $E(x_p+1,y_p)$
  - $\Delta E = 2(x_p + 1) + 3 = (2x_p + 3) + 2 = \Delta E_{old} + 2$
  - $\Delta SE = 2(x_p + 1) - 2y_p + 5 = (2x_p - 2y_p + 5) + 2 = \Delta SE_{old} + 2$



Second order difference wrt $E(x_p+1,y_p)$

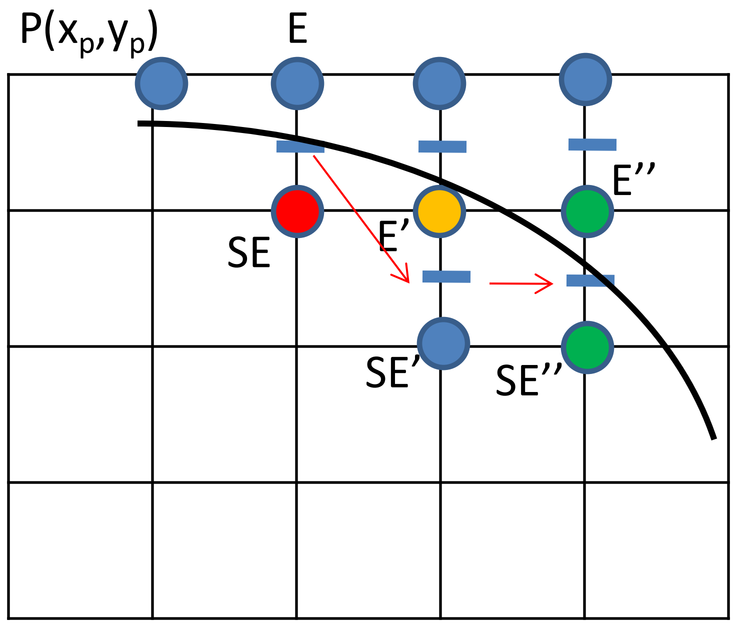
```
void MidpointCircle (int radius, int value)
/* This procedure uses second-order partial differences to compute increments */
/* in the decision variable. Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    int deltaE = 3;
    int deltaSE = -2 * radius + 5;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0) {            /* Select E */
            d += deltaE;
            deltaE += 2;
            deltaSE += 2;
        } else {
            d += deltaSE;       /* Select SE */
            deltaE += 2;
            deltaSE += 4;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

Fig. 3.18 Midpoint circle scan-conversion algorithm using second-order differences.

Second order difference wrt SE(x_p+1,y_p-1)

- P(x_p,y_p) → $\Delta E = 2x_p + 3 = \Delta E_{old}$ Increment in **d** when E is chosen
  → $\Delta SE = 2x_p - 2y_p + 5 = \Delta SE_{old}$ Increment in **d** when E is chosen

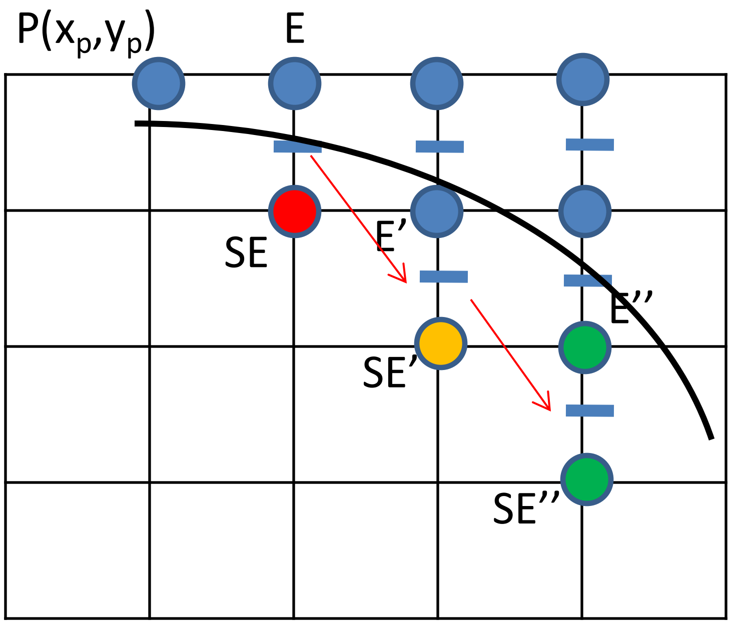
- SE(x_p+1,y_p-1) → $\Delta E = 2(x_p + 1) + 3 = (2x_p + 3) + 2 = \Delta E_{old} + 2$



Second order difference wrt SE(x_p+1,y_p-1)

- P(x_p,y_p)
  - $\Delta E = 2x_p + 3 = \Delta E_{old}$ Increment in **d** when E is chosen
  - $\Delta SE = 2x_p - 2y_p + 5 = \Delta SE_{old}$ Increment in **d** when E is chosen

- SE(x_p+1,y_p-1)
  - $\Delta E = 2(x_p + 1) + 3 = (2x_p + 3) + 2 = \Delta E_{old} + 2$
  - $\Delta SE = 2(x_p+1) - 2(y_p-1)+5 = (2x_p - 2y_p+5)+2+2 = \Delta SE_{old} + 4$



Second order difference wrt SE(x_p+1,y_p-1)

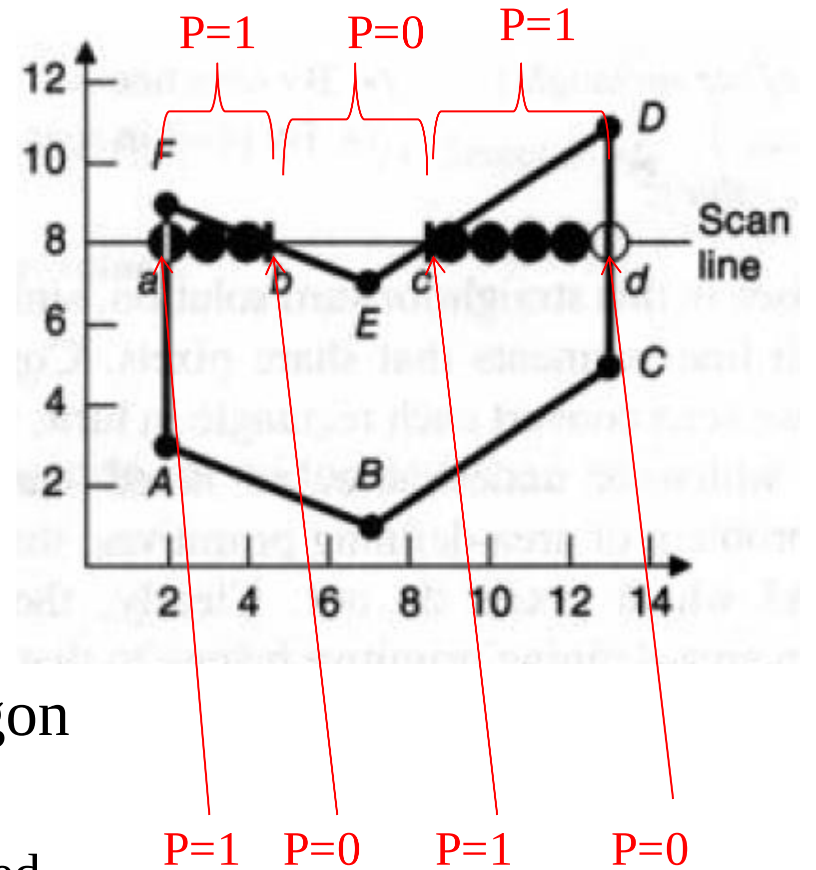
```
void MidpointCircle (int radius, int value)
/* This procedure uses second-order partial differences to compute increments */
/* in the decision variable. Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    int deltaE = 3;
    int deltaSE = -2 * radius + 5;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0) {            /* Select E */
            d += deltaE;
            deltaE += 2;
            deltaSE += 2;
        } else {
            d += deltaSE;       /* Select SE */
            deltaE += 2;
            deltaSE += 4;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

Fig. 3.18 Midpoint circle scan-conversion algorithm using second-order differences.

Polygon Scan Conversion

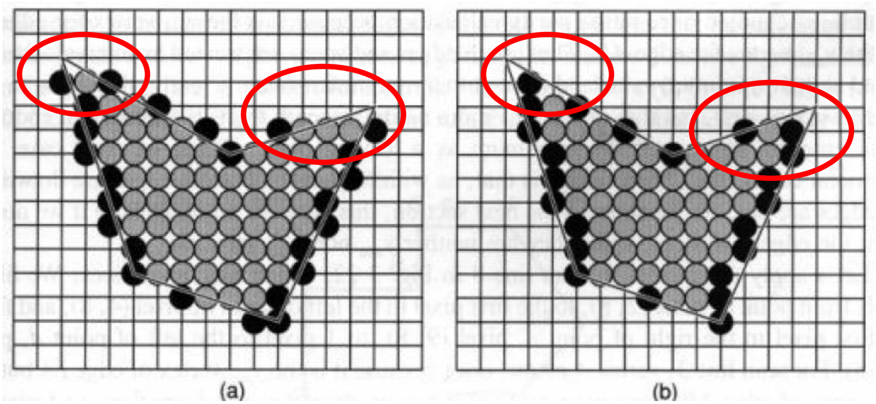
Steps:

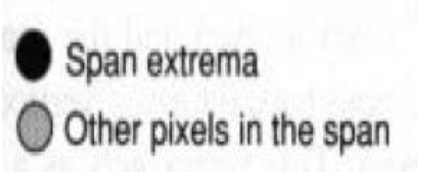
1. Find the intersections of the scan line with the polygon edges
2. Sort the intersection points in increasing order of x coordinate
3. For each pair of intersections, use odd parity rule to draw the pixels that are interior to the polygon
   - Initially parity is even, P = 0
   - For each intersection, parity is changed
   - **Draw the pixels only when the parity is odd**



Polygon Scan Conversion

- Draw only those pixels which are strictly interior to the polygon



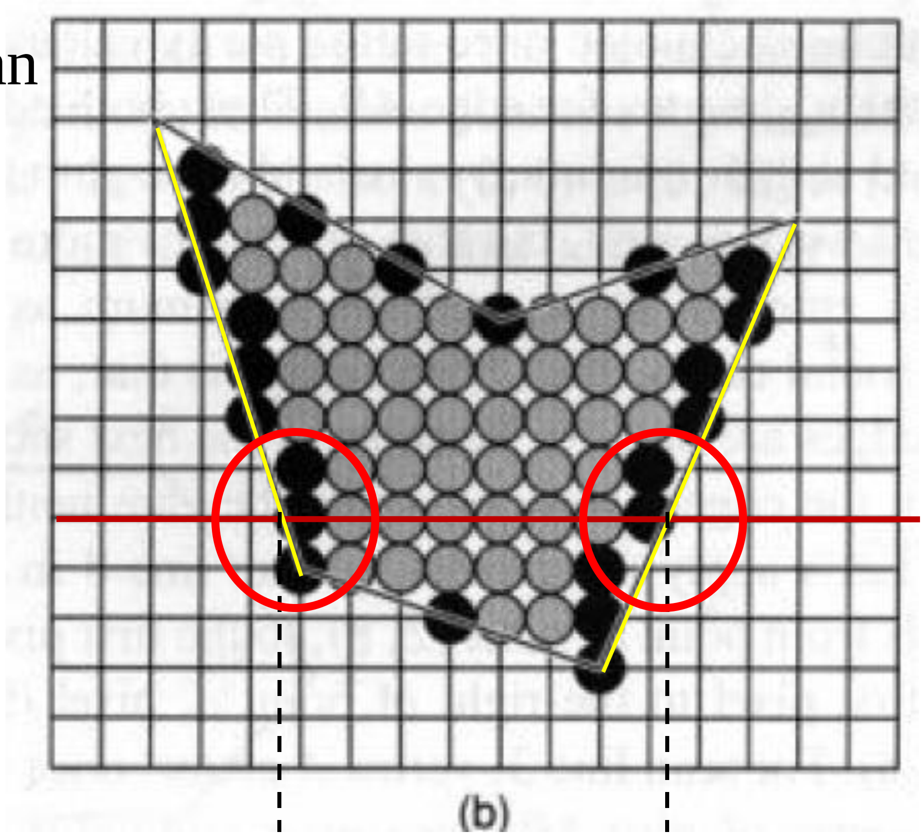


- Cases to consider:
  1. Intersection point is fractional
  2. Intersection point is integer
  3. Integer intersection point that is a shared vertex
  4. Integer intersection points that define horizontal edge

Polygon Scan Conversion

1. Intersection point is fractional

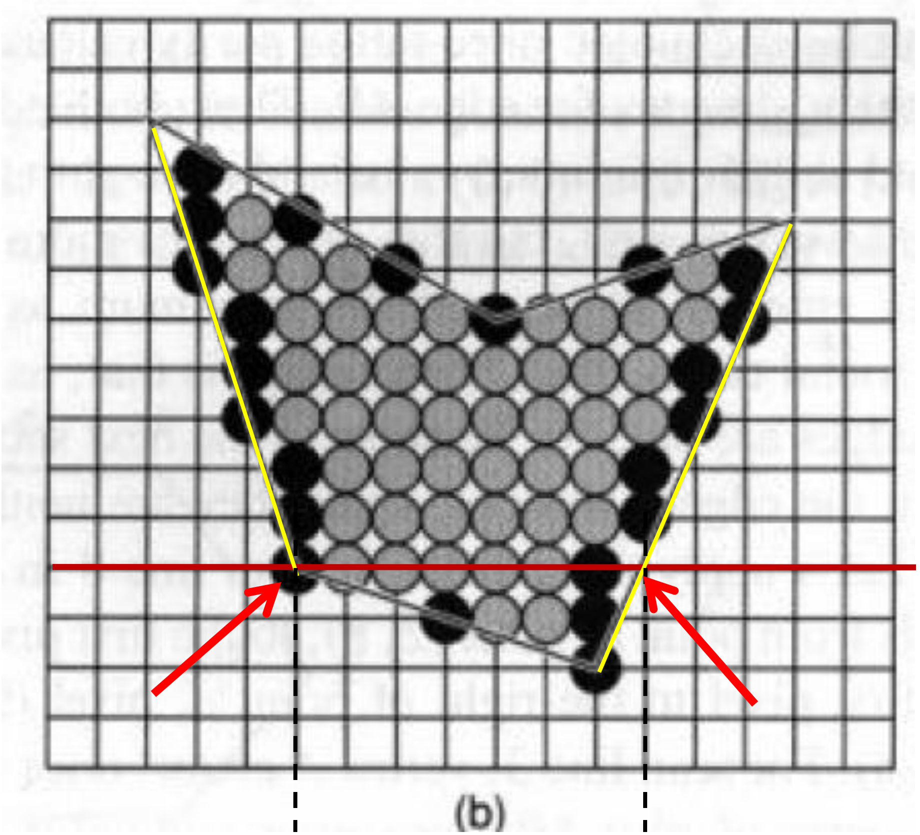
- If it is the left most of the span and before intersection we were outside the polygon, then round up
- If it is the right most of the span, and before intersection we were inside the polygon, then round down



Polygon Scan Conversion

2. Intersection point is integer

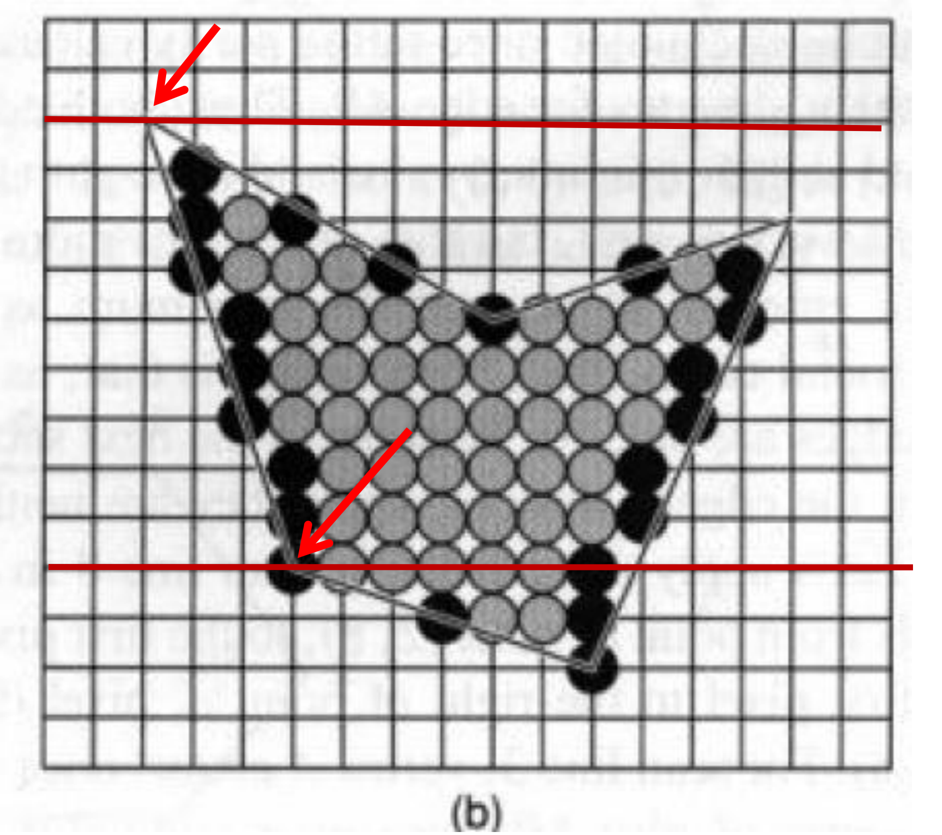
- If it is the left most of the span, draw it
- If it is the right most of the span, don't draw it



Polygon Scan Conversion

3. Integer intersection point that is a shared vertex

- Draw it only if it is y_{min} of an edge

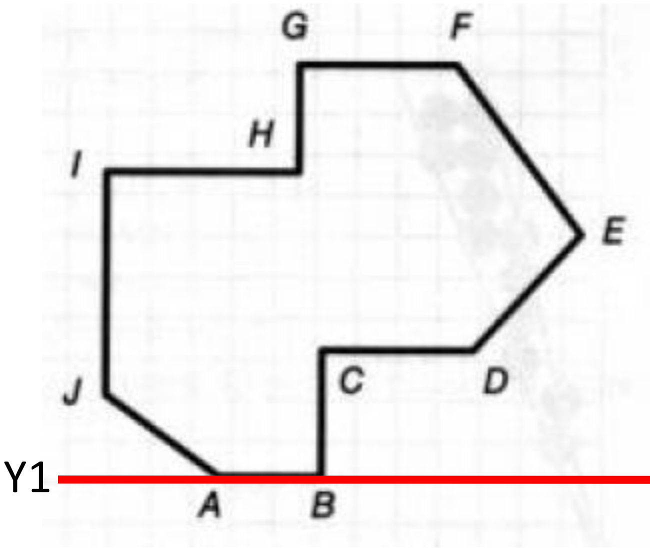


Polygon Scan Conversion

4. Integer intersection points that define horizontal edge

Scanline Y1:

- A is a Y_{min} of JA.
- A is drawn and parity becomes odd.
- Through out AB, parity remains odd, the span **AB is drawn.**
- B is Y_{min} of CB. Parity becomes even. Drawing stops.

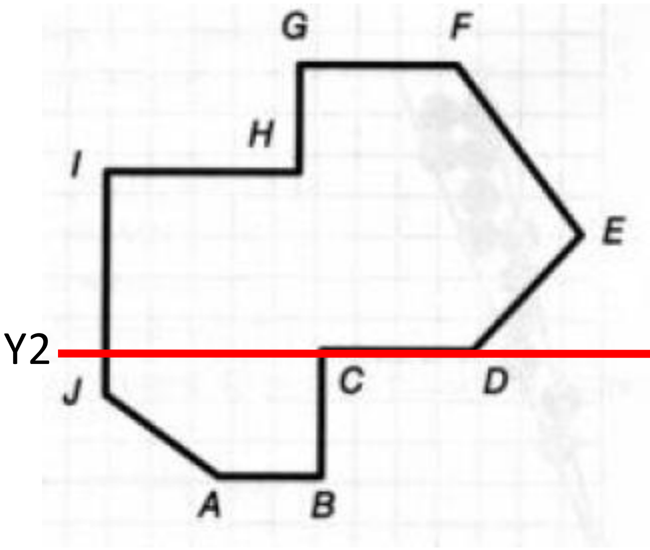


Polygon Scan Conversion

4. Integer intersection points that define horizontal edge

Scanline Y2:

- First Intersection point is drawn and parity becomes odd.
- Through out the span up to C, parity remains odd, the span is drawn.
- C is not Y_{min} of CD or BC. So C is not considered. Parity remains odd. Span **CD is drawn.**
- D is Y_{min} of DE. Parity becomes even. Drawing stops.

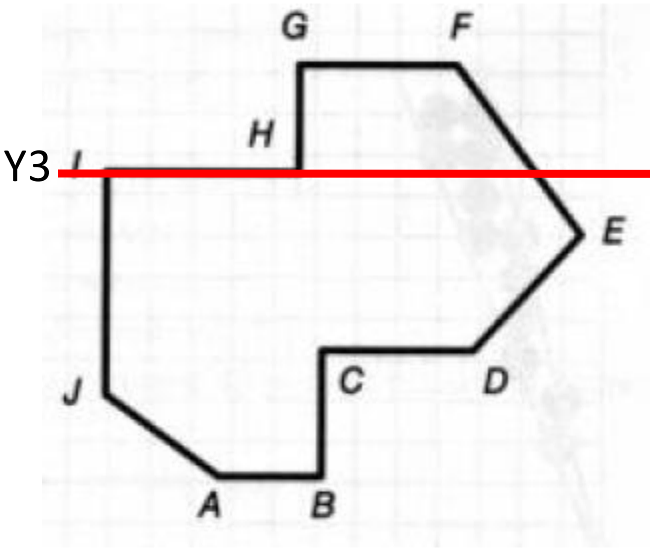


Polygon Scan Conversion

4. Integer intersection points that define horizontal edge

Scanline Y3:

- I is not Y_{min} of IH or IJ. So I is not considered. Parity remains even. Span **IH is not drawn.**
- H is Y_{min} of GH. So H is considered and Parity becomes odd. The span up to next intersection point is drawn.

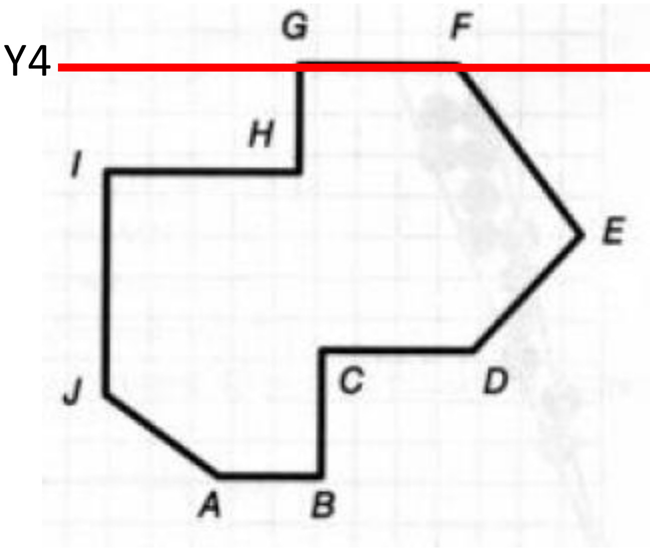


Polygon Scan Conversion

4. Integer intersection points that define horizontal edge

Scanline Y4:

- G is not Y_{min} of GF or GH. So G is not considered. Parity remains even. Span **GF is not drawn.**



Summary:

Top edges are not drawn

Bottom edges are drawn

Polygon Scan Conversion

- Silver : In case of thin polygon area, each scan line may not have a distinct span (e.g. a single pixel or no pixel)

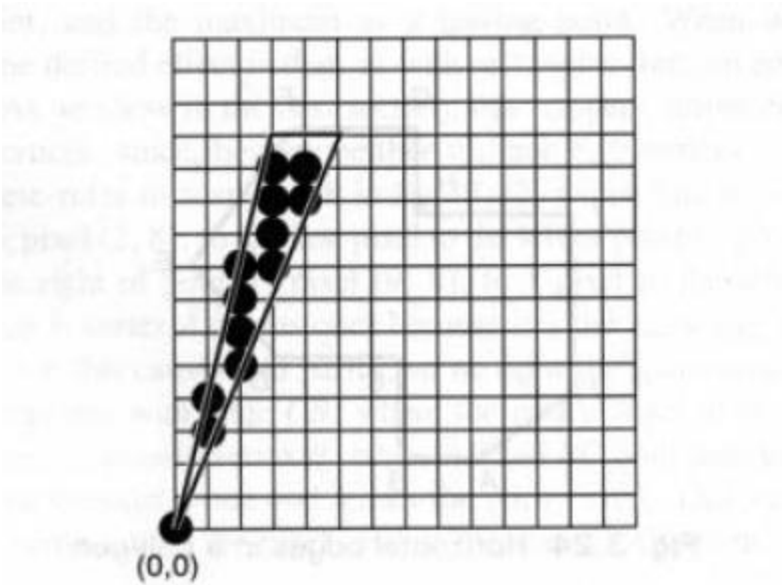


Fig. 3.25 Scan converting slivers of polygons.

Summary

- A Brief Idea of Rasterization

- Scan Conversion of Lines, Circles in Detail
  - Inefficient techniques and why they are inefficient
  - Efficient algorithms based on the position of midpoint calculation

- Scan Conversion of Filled Polygons
  - Challenges and some conventions

Thank you ☺