Scan Conversion Algorithms for 2D Output Primitives

Types of Primitives to be Scan Converted

- Straight Lines
- Polygons
- Circles
- Ellipses and Other 2-D Curves
- Text (Characters)

Scan Conversion Algorithms for Drawing Straight Lines

- Task
  - Given pixel coordinates of endpoints
    - P1 (x1,y1) and P2 (x2,y2)
  - Determine which pixels need to be painted
- Criteria
  - Straight as possible between endpoints
  - Constant density (no gaps or bunching)
  - Density independent of orientation
  - <u>Must be fas</u>t

Line Equations

- <u>Differential equation:</u>

  dy/dx = m (m=constant: the slope)
- Integrate (indefinite)

  y = m*x + constant

  The constant (b) is called y intercept

  (value of y when x=0)
- y = m*x + b
- “slope-intercept” form

- Integrate between endpoints (definite)-->

  (y2-y1) = m*(x2-x1)

  m = (y2-y1)/(x2-x1)

  (an operational definition of slope)

- Integrate between endpoint (x1,y1) and arbitrary point to be plotted (x,y) -->

  y - y1 = m*(x-x1)

  y = m*(x-x1) + y1

  This is the “point-slope” form

  – Compute points (x,y) given a point (x1,y1) and the slope of the line

Parametric Form

Express x and y linearly in terms of a paramter, t

x = ax*t + bx

y = ay*t + by

ax, bx, ay, by are constants to be determined

Let t range between t=0, endpoint (x1,y1) and t=1, endpoint (x2,y2)

Determining the constants: Use endpoint values

x1 = ax*0 + bx ==> bx = x1

x2 = ax*1 + bx ==> ax = x2-x1

So x = (x2-x1)*t + x1, 0<=t<=1

And y = (y2-y1)*t + y1

Brute Force Line-Drawing Algorithm

Use "point-slope" form

<u>Step in x direction,</u> assume x2 > x1

(if x1 > x2, swap the points)

Compute m = (y2-y1)/(x2-x1)

num-pts = x2-x1+1

x = x1

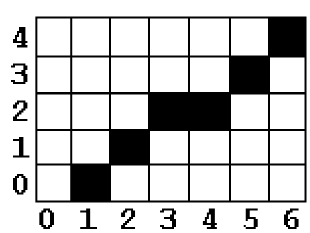
Repeat num-pts times

    y = m*(x-x1) + y1
    SetPixel(x, round(y))
    x = x+1

Problem if |y2-y1| > |x2-x1| --> gaps

(1,0) to (6,4)

n = 6-1+1 = 6



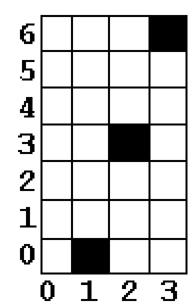
x2-x1 = 5

y2-y1 = 4

no gaps!

(1,0) to (3,6)

n = 3-1+1 = 3



x2-x1=2

y2-y1=6

gaps!

Solution: Step in y direction

Stepping in y direction

If |y2-y1| > |x2-x1|, <u>step in y,</u> assume y2 > y1
(if y1 > y2, swap the points):

```
Compute inv_m = (x2-x1)/(y2-y1)
num-pts = y2-y1+1
y = y1
Repeat num-pts times
    x = inv_m*(y-y1) + x1
    SetPixel(round(x), y)
    y = y+1
```

Brute Force line algorithm, continued

- Vertical lines (x2 = x1)
  - y = y+1 for each new pixel
  - x doesn't change
- Horizontal lines (y2 = y1)
  - x = x + 1
  - y doesn't change

Brute Force Method is **Too Slow**

- Each iteration has:
  - floating point multiply
  - floating point add
  - round() operations

Incremental Methods--The Digital Differential Analyzer (DDA)

- Idea: get new point from previous point
- $dy/dx = m \Rightarrow \Delta y/\Delta x = m \Rightarrow \Delta y = m*\Delta x$
- But $\Delta y = ynew - yold$
- And $\Delta x = xnew-xold$
  - So $xnew = xold + \Delta x$
  - and $ynew = yold + \Delta y$
  - i.e., $ynew = yold + m*\Delta x$

DDA, continued

- Choose $\Delta x = 1$
  - stepping in x direction
  - Pixel by pixel
- Then compute each new y value

  ynew = yold + m

DDA Algorithm stepping in x, x2 > x1

(If x1 > x2, swap the points)

```
Compute m = (y2-y1)/(x2-x1)
num-pts = x2-x1+1
x = x1
y = y1
Repeat num-pts times
    SetPixel(x,round(y))
    x = x+1
    y = y+m
```

- As for the Brute force method, if |m|>1 and we step in x, we get gaps
  – So we can step in y
- DDA Algorithm, stepping in y, y2 > y1
  – (if y1 > y2, swap the points):

```
Compute inv_m = (x2-x1)/(y2-y1)
num-pts = y2-y1+1
x = x1
y = y1
Repeat num-pts times
    SetPixel(round(x),y)
    y = y+1
    x = x+inv_m
```

DDA is Better, but Still Not Fast Enough

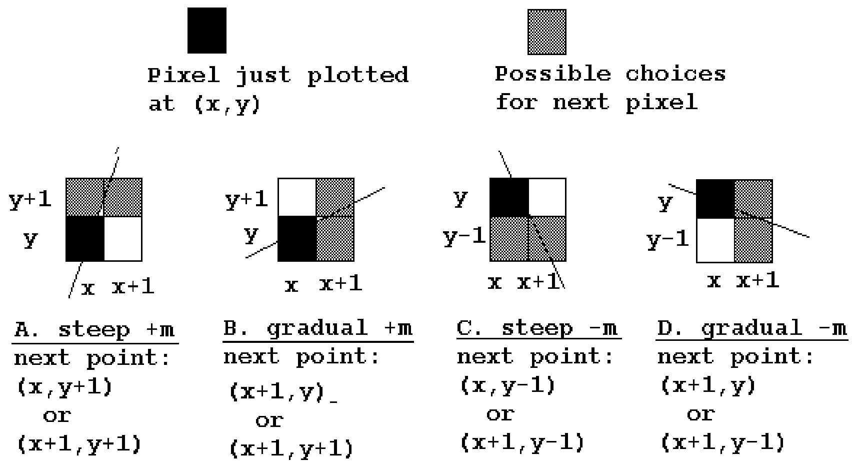
- Floating point multiply gone from loop
- But loop still has a floating point add
- And a round()
- WE CAN DO BETTER!
- Best performance:
  – Only integer adds/subtracts inside loop

Bresenham's Line-drawing Algorithm

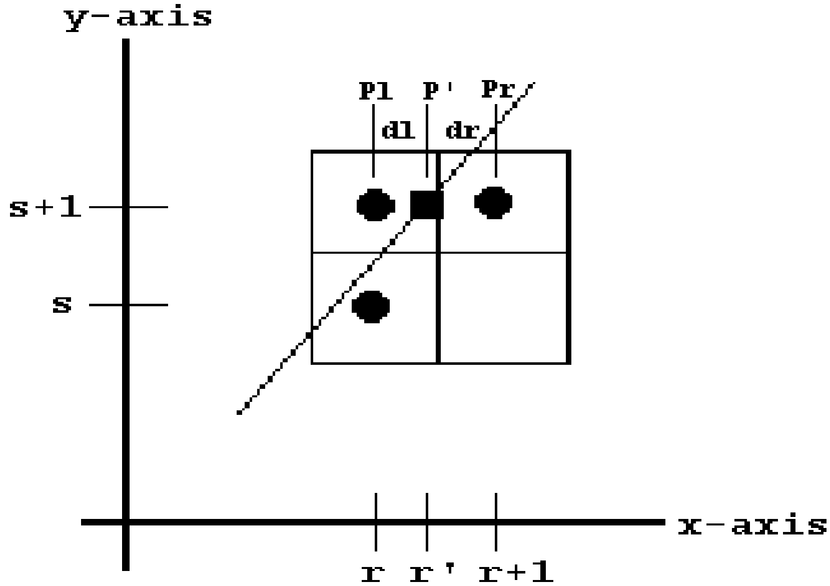
- Used in most graphics packages
- Often implemented in hardware
- Incremental (new pixel from old)
- Uses only integer operations

- Basic Idea of Bresenham Algorithm:
  - All lines can be placed in one of four categories:
    - A. Steep positive slope ($m > 1$)
    - B. Gradual positive slope ($0 < m <= 1$)
    - C. Steep negative slope ($m < -1$)
    - D. Gradual negative slope ($0 >= m >= -1$)
  - In each case, there are only 2 choices for the next pixel to be plotted!

The Four Bresenham Cases



- Look at Case-A (Steep positive slope)
- Also assume P1 is to the left of P2 (x1<x2)
  - If not true, points can be swapped
- delta_y > delta_x ==> stepping in y

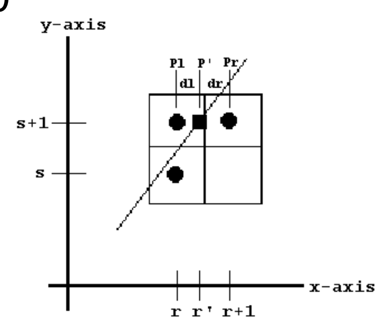


- If dl<dr,
  - Pl is closer to actual point than Pr
- i.e., if dl-dr<0, choose "left" pixel
- Criterion for choosing “left” pixel (Pl) is:

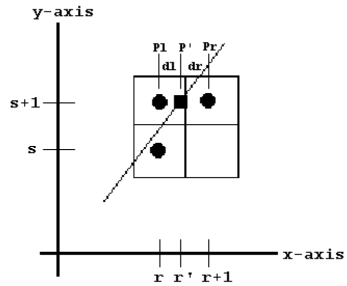
  dl-dr = r’-r - (r+1-r’) < 0

  or:

  dl-dr = 2*r’ - 2*r -1 < 0



But from the equation for a straight line:



$y = m*x + b$
New $y = s+1$
$s+1 = (\Delta y/\Delta x)*r' + b$
$r' = (s+1-b)*\Delta x/\Delta y$
So:
Criterion for choosing Pl:
$dl-dr = 2*r' - 2*r -1 < 0$
$dl-dr = 2*(s+1-b)*\Delta x/\Delta y - 2*r -1 <0$

Result:

$dl-dr = 2*(s* + 1 - b)*\Delta x /\Delta y -2*r -1 < 0$

If dl-dr is negative, choose "left" pixel

Multiply by Δy to get rid of divide operation

(always positive for Case-A lines)

Call result the "predictor", P

$P = \Delta y*(dl-dr)$

Result:

$\underline{P=2*\Delta x*(s+1-b) - 2*r*\Delta y - \Delta y}$

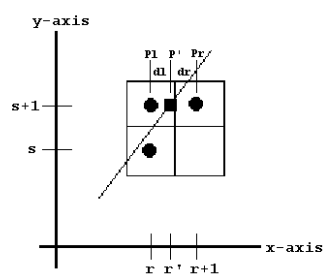
Divide is gone--but it's still too complex

Bresenham's Contribution

– Try to find a recurrence relation for P
– Call Pn the new value, and Po the old value
  • Then Pn = Po + ΔP
– Call sn & so the new & old values of s
– Call rn & ro the new & old values of r

Predictor P:

<u>P=2*Δx*(s+1-b) - 2*r*Δy - Δy</u>



Change in Predictor:

ΔP = Pn - Po, so:

Pn = Po + ΔP

Point just plotted: (ro,so)

Two cases for new point:

Left case (rn=ro and sn=so+1)

Right case (rn=ro+1 and sn=so+1)

For both cases:

Po = 2*Δx*(so+1-b) - 2*ro*Δy - Δy

Predictor P: <u>$P=2*\Delta x*(s+1-b) - 2*r*\Delta y - \Delta y$</u>

New Point Left Case (ro,so+1):

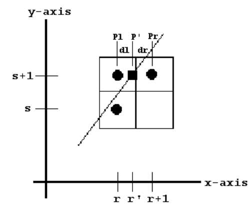
$Pn = 2*\Delta x*((so+1)+1-b) - 2*ro*\Delta y - \Delta y$

$Po = 2*\Delta x*(so+1-b) - 2*ro*\Delta y - \Delta y$

Subtracting Po from Pn gives ΔP

Result:

<u>$\Delta P = 2*\Delta x$</u>



New Point Right Case (ro+1,so+1):

$Pn = 2*\Delta x*((so+1)+1-b) - 2*(ro+1)*\Delta y - \Delta y$

$Po = 2*\Delta x*(so+1-b) - 2*ro*\Delta y - \Delta y$

Again subtracting Po from Pn gives ΔP:

<u>$\Delta P = 2*(\Delta x - \Delta y)$</u>

- Both results are very simple (Integers!!)
- Look at current value of the predictor:

If (P < 0) // left case

$P = P + 2*\Delta x$

$x = x$

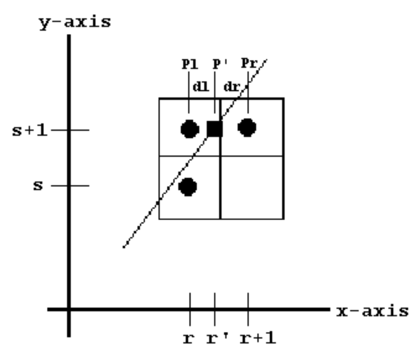
$y = y + 1$

If (P>0) // right case

$P = P + 2*(\Delta x-\Delta y)$

$x = x + 1$

$y = y + 1$



- But to start things off, we need an initial value P0 of the predictor
- Substitute left-hand endpoint (x1,y1) into predictor definition:

P = 2*Δx*(s+1-b) - 2*r*Δy - Δy ==>

P0 = 2*Δx*(y1+1-b) -2*x1*Δy - Δy

- And use fact that (x1,y1) is on line:

i.e., y1 = (Δy/Δx)*x1 + b

P0= 2*Δx*((Δy/Δx)*x1 + b +1 - b) -2*x1*Δy - Δy

P0=2*Δy*x1 + 2*Δx -2*x1*Δy - Δy

- Result: P0 = 2*Δx - Δy

Case-A Bresenham Algorithm (Steep positive slope)

```
If  (x1>x2) swap endpoints;
del_x = x2-x1;   del_y = y2-y1;
P = 2*del_x - del_y;
cleft = 2*delx;   cright = 2*del_x - 2*del_y;
x = x1;   y = y1;   num_pts = |del_y| + 1;
Repeat num_pts times
     SetPixel(x,y);  y = y + 1;
     If (P < 0)
        P = P + cleft;
    Else
        {P = P + cright;  x = x + 1;}
```

- Can be generalized to handle Case-C (steep negative slope) lines
- Compute sdy = sign(Δy)
  - = 1 if y2>y1
  - = -1 if not
- Then, in definition of P and cright:
  – Replace Δy with sdy*Δy
  – Replace y = y + 1 with y = y + sdy
- Then both Case-A and Case-C lines are handled

More Info on Bresenham Line-drawing Algorithm

- See Hearn & Baker Text Book
- Section 3-1 (pages 88-95)
- Specifically Case-B lines

Speeding Up Bresenham

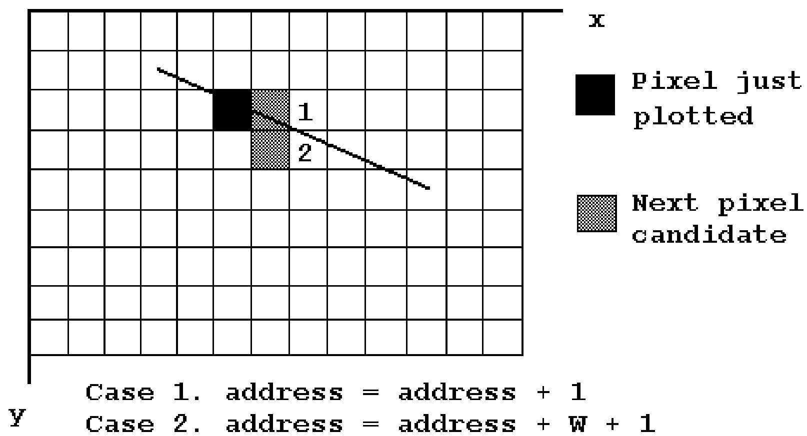
- Bresenham's algorithm calls SetPixel()
- Not optimized
  - SetPixel(x,y) must work for any pixel
  - For W x H screen, Address = W*y + x
  - Multiply involved (even though hidden)
- Bresenham: We know next pixel is one of two choices
- Faster to access frame buffer directly using addresses -- not values of x and y

- Assume Row major order
- Take advantage of symmetry
- Store addresses instead of coordinates (x,y)
- Example: W x H x 256 direct color mode
  - One byte per pixel

    Byte Address = W*y + x

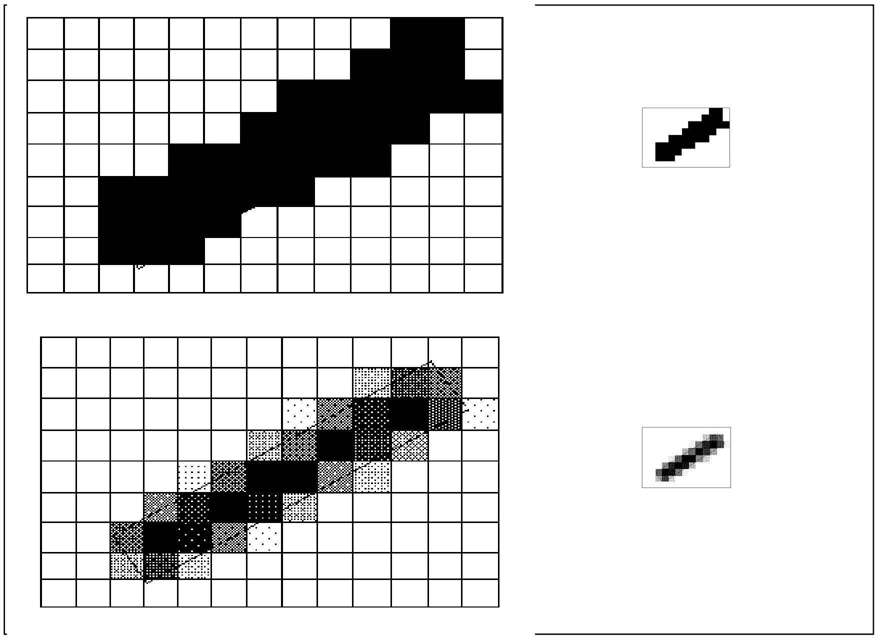
    Look at Case A (gradual +m)
  - Only integer add needed

Case A Line (gradual +m)



Aliasing (Jaggies)

- Inherent in Raster Scan systems
- Anti-aliasing technique for grayscale:
  - Consider broad line covering several pixels
  - Border pixels
    - Set intensity proportional to % of pixel inside line
    - Produces blurring
    - Looks less jagged
    - But must compute areas (compute intensive)
    - Can use statistical sampling instead



Polyline Algorithm

```
Polyline (POINT *p, int n)
{
int xo, yo, xn, yn;
if (n==0) return;
xo=p[0].x; yo=p[0].y;
if (n==1) {SetPixel(xo, yo); return;}
for (i=1; i<n; i++)
    {xn=p[i].x; yn=p[i].y;
    Line(xo,yo,xn,yn);
    xo=xn; yo=yn;}
}
```

Calling the Polyline Algorithm

```
POINT pt[3];
pt[0].x=50; pt[0].y=10;
pt[1].x=250; pt[1].y=50;
pt[2].x=125; pt[2].y=130;
Polyline(pt,3);
```

Scan Converting Circles

Given:

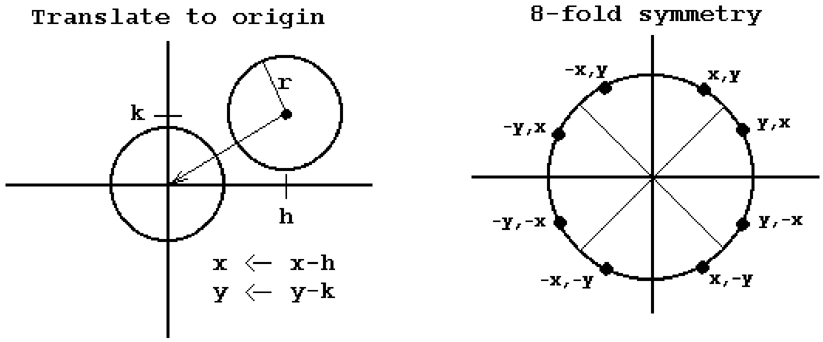
- Center: (h,k)
- Radius: r

Equation:

- $(x-h)^2 + (y-k)^2 = r^2$

To simplify we'll translate origin to center

- Simplified Equation:
  - $x^2 + y^2 = r^2$



Circle has 8-fold symmetry

So only need to plot points in 1st octant

$\Delta x > \Delta y$ so step in x direction

Brute Force Circle Algorithm

Suppose we have a Set8pixel() routine

```
xfin = 0.707*r
For (x=0; x<=xfin ; x++)
    {
    y = SQRT(r*r - x*x);
    Set8Pixel(round(x), round(y));
    }
    TOO SLOW!!
```

The Set8Pixel(x,y) routine

```
SetPixel(x,y);
SetPixel(x,-y);
SetPixel(-x,y);
SetPixel(-x,-y);
SetPixel(y,x);
SetPixel(y,-x);
SetPixel(-y,x);
SetPixel(-y,-x);
```

Could Use Parametric Equations

```
for (theta=90; theta>=45; theta- -)
    {
    x = r*cos(theta);
    y = r*sin(theta);
    Set8Pixel(round(x), round(y));
    }
```

EVEN SLOWER!

DDA Circle Approximation

$x^2 + y^2 = r^2$

Take Derivative:

2*x+2*y*(dy/dx) = 0

dy = (-x/y)*dx

Step in x direction (dx=1)

dy = -x/y

y = y + dy (approximation)

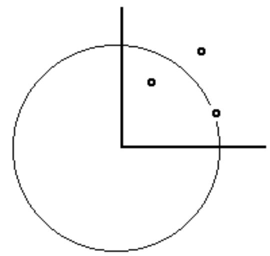
DDA Circle Algorithm

```
x=0; y=r;
xfin=0.707*r;
while (x<=xfin)
    {
    Set8Pixel(round(x), round(y));
    y = y - (x/y);
    x = x + 1;
    }
```

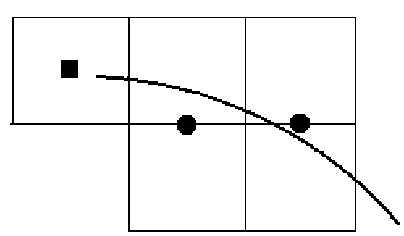
Floating Pt. Divide--STILL TOO SLOW!

Midpoint Circle Algorithm

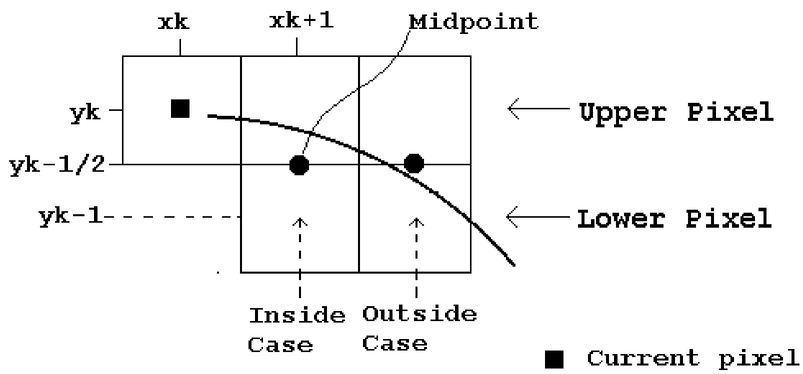
- Extension of Bresenham ideas
- Circle equation: $x^2 + y^2 = r^2$
- Define a circle function:
  $f = x^2 + y^2 - r^2$
- f=0 ==> (x,y) is on circle
- f<0 ==> (x,y) is inside circle
- f>0 ==> (x,y) is outside circle



- We've just plotted (x_k, y_k)
- $(\Delta x > \Delta y)$, so we're stepping in x
- Next pixel is either:
  $(x_k + 1, y_k)$ -- the "top" case or
  $(x_k + 1, y_k - 1)$ -- the "bottom" case
- Look at midpoint



Midpoint Circle Choices



Inside: f<0 ==> choose upper pixel
Outside: f>0 ==> choose lower pixel

- Evaluate f at midpoint
  $(x=x_k+1, y=y_k-1/2)$
- Define Predictor: $P_k= f(x_k+1,y_k-1/2)$
  $P_k<0$ ==> inside (choose top pixel)
  $P_k>0$ ==> outside (choose bottom pixel)
  $P_k = (x_k+1)^2 + (y_k-1/2)^2 - r^2$
- $P_k = x_k^2 + 2x_k +5/4 +y_k^2 -y_k - r^2$
- As for Bresenham, try to get a recurrence relation for P

- Top Case ($x_{k+1} = x_k + 1,\ y_{k+1} = y_k$):

$P_{k+1} = f(x_{k+1} + 1,\ y_{k+1} - 1/2)$
But $x_{k+1} = x_k + 1$ and $y_{k+1} = y_k$
So $P_{k+1} = ((x_k+1) + 1)^2 + (y_k - 1/2)^2 - r^2$
$P_{k+1} = (x_k+2)^2 + (y_k - 1/2)^2 - r^2$
$P_{k+1} = x_k^2 + 4x_k + 4 + y_k^2 - y_k + 1/4 - r^2$
But, $P_k = x_k^2 + 2x_k + 5/4 + y_k^2 - y_k - r^2$
$\Delta P_k = P_{k+1} - P_k$
So $\Delta P_k = 2x_k + 3$, But $x_{k+1} = x_k + 1$
So $\Delta P_k = 2x_{k+1} + 1$

- Bottom Case ($x_{k+1} = x_k + 1,\ y_{k+1} = y_k - 1$):

$P_{k+1} = f(x_{k+1} + 1,\ y_{k+1} - 1/2)$

- $P_{k+1} = ((x_k+1) + 1)^2 + ((y_k-1) - 1/2)^2 - r^2$
$= (x_k+2)^2 + ((y_k - 3/2)^2 - r^2$
$= x_k^2 + 4x_k + 4 + y_k^2 - 3x_k + 9/4 - r^2$
But $P_k = x_k^2 + 2x_k + 5/4 + y_k^2 - y_k - r^2$
$\Delta P_k = P_{k+1} - P_k$
So $\Delta P_k = 2x_k - 2y_k + 5$
$\Delta P_k = 2(x_{k+1} - y_{k+1}) + 1$

- Initial P:

  P0 ($x_0=0$, $y_0=r$)

  $P_0 = (x_0 + 1)^2 + (y_0 - 1/2)^2 - r^2$

  $P_0 = 5/4 - r$ --> $1-r$ (rounding to integer)

Midpoint Circle Algorithm

```
x=0; y=r;   P=1-r;
Set8Pixel(x,y);
while (x<y)
    {
    x = x + 1; Set8Pixel(x,y);
    if (P < 0)
        P = P + x<<1 + 1;
    else
        { y = y - 1; P = P + (x-y)<<1 + 1;}
    }
```