Scan Conversion Algorithms
- Straight Lines (Bresenham)
- Antialiasing Straight Lines
- Polygons
- Circles

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 ==&gt; 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 $Pl$ is:
  
  $dl - dr = r' - r - (r+1 - r') < 0$  
  or:
  $dl - dr = 2r' - 2r - 1 < 0$

But from the equation for a straight line:

$y = mx + b$  
$s+1 = (?y/\bar{x}) \cdot r' + b$  
$r' = (s+1-b) \cdot \bar{x}/\bar{y}$

So:

$dl - dr = 2r' - 2r - 1 < 0$ 
$dl - dr = 2(s+1-b) \cdot \bar{x}/\bar{y} - 2r - 1 < 0$

Result: 

$dl - dr = 2(s' + ?y\bar{b}) \cdot \bar{x}/\bar{y} - 2r - 1 < 0$

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

Multiply by $\bar{y}$

(always positive for Case-A lines)  
Call result the "predictor", $P$ 

$P = ?y(\bar{dl} - \bar{dr})$

Result:

$P = 2x(s+1-b) - 2y*?y - r - 1$

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 $Pn = Po + \Delta P$
- Call $sn$ & $so$ the new & old values of $s$
- Call $rn$ & $ro$ the new & old values of $r$

Change in Predictor:

$?P = Pn - Po$, so:

$Pn = Po + ?P$

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:

$Po = 2x(s+1-b) - 2ro*?y - ?y$
New Point Left Case \((r_0, s_0+1)\):
\[
P_n = 2^s x^*(s_0+1) - 2^{r_0} y - ? y
\]
\[
P_0 = 2^s x^*(s_0+1) - 2^{r_0} y - ? y
\]
Subtracting \(P_0\) from \(P_n\) gives \(P\):
\[
P = 2^s x - ? y
\]

New Point Right Case \((r_0+1, s_0+1)\):
\[
P_n = 2^s x^*((s_0+1)+1) - 2^{r_0+1} y - ? y
\]
\[
P_0 = 2^s x^*(s_0+1) - 2^{r_0} y - ? y
\]
Again subtracting \(P_0\) from \(P_n\) gives \(P\):
\[
P = 2^s x - ? y
\]

Both results are very simple (Integers!!)

Look at current value of the predictor:
If \(P < 0\)  // left case
\[
\begin{align*}
P &= P + 2^s x \\
x &= x \\
y &= y + 1
\end{align*}
\]
If \(P > 0\)  // right case
\[
\begin{align*}
P &= P + 2^s (x - y) \\
x &= x + 1 \\
y &= y + 1
\end{align*}
\]

Case-A Bresenham Algorithm
If \((x_1, y_1) = (x_2, y_2)\) swap endpoints;
\[
\begin{align*}
del_x &= x_2 - x_1; \\
del_y &= y_2 - y_1; \\
P &= 2^{del_x} - del_y; \\
cleft &= 2^{del_x}; \\
cright &= 2^{del_x} - 2^{del_y}; \\
x &= x_1; \\
y &= y_1; \\
num_pts &= |del_y| + 1;
\end{align*}
\]
Repeat \(num_pts\) times
\[
\begin{align*}
&\text{SetPixel}(x, y); y = y + 1; \\
&\text{If } (P < 0) \\
&\quad P = P + cleft; \\
&\text{Else} \\
&\quad P = P + cright; x = x + 1;
\end{align*}
\]

More Info on Bresenham Line-drawing Algorithm
Can be generalized to handle Case-C (steep negative slope) lines
\[
\begin{align*}
\text{Compute } sdy &= \text{sign}(y) \\
&= 1 \text{ if } y > y_{\text{ref}} \\
&= -1 \text{ if not}
\end{align*}
\]
Then, in definition of \(P\) and \(cright\):
\[
\begin{align*}
&\text{Replace } ? y \text{ with } sdy??y \\
&\text{Replace } y = y + 1 \text{ with } y = y + sdy
\end{align*}
\]
Then both Case-A and Case-C lines are handled

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 WxH 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 with addresses -- not values of x and y

Assume Row major order
- Take advantage of symmetry
- Store addresses instead of (x,y)
- Example: WxHx256 mode
  - Byte Address = W*y + x
  - Look at Case A (gradual +m)
  - Only integer add needed

Case A Line (gradual +m)

```
Case 1. address = address + 1
Case 2. address = address + W + 1
```

Aliasing (Jaggies)

- Inherent in Raster Scan systems
- Anti-aliasing technique for grayscale:
  - Consider broad line covering several pixels
  - Border pixels
    - Set intensity 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:
\[x^2 + y^2 = r^2\]

Brute Force Circle Algorithm

Suppose we have a Set8pixel() routine
\[x_{fin} = 0.707r\]
For \((x=0; x<x_{fin}; x++)\)
\{
  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 \((\text{theta}=90; \text{theta}>=45; \text{theta}-\))
\{
  x = r*cos(\text{theta});
  y = r*sin(\text{theta});
  Set8Pixel(round(x), round(y));
\}
EVEN SLOWER!
**DDA Circle Approximation**

\[ x^2 + y^2 = r^2 \]

Take Derivative:

\[ 2x + 2y(dy/dx) = 0 \]

\[ dy = (-x/y)dx \]

Step in x direction (dx=1)

\[ dy = -x/y \]

\[ y = y + dy \] (approximation)

---

**DDA Circle Algorithm**

\[ x=0; y=r; \]

\[ x_{\text{fin}}=0.707\times r; \]

while (\(x<=x_{\text{fin}}\))

\[
\begin{align*}
&\{ \\
&\text{Set8Pixel(round(x), round(y));} \\
&\text{y = y - (x/y);} \\
&\text{x = x + 1;} \\
&\}
\end{align*}
\]

Floating Pt. Divide--STILL TOO SLOW!