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 \((x_1, y_1)\) and P2 \((x_2, y_2)\)
  - Determine which pixels need to be painted

- **Criteria**
  - Straight as possible between endpoints
  - Constant density (no gaps or bunching)
  - Density independent of orientation
  - **Must be fast**

Line Equations

- **Differential equation:**
  \[
  \frac{dy}{dx} = m \quad (m=\text{constant: the slope})
  \]
- **Integrate (indefinite)**
  \[
  y = mx + \text{constant}
  \]
  The constant \((b)\) is called y intercept
  (value of \(y\) when \(x=0\))
- \(y = mx + b\)
- “slope-intercept” form
Integrate between endpoints (definite)-->  
\[(y_2 - y_1) = m*(x_2 - x_1)\]
\[m = (y_2 - y_1)/(x_2 - x_1)\]
(an operational definition of slope)

Integrate between endpoint \((x_1, y_1)\) and arbitrary point to be plotted \((x, y)\) -->  
\[y - y_1 = m*(x-x_1)\]
\[y = m*(x-x_1) + y_1\]
This is the “point-slope” form
– Compute points \((x,y)\) given a point \((x_1,y_1)\) and the slope of the line

Parametric Form
Express \(x\) and \(y\) linearly in terms of a parameter, \(t\)
\[x = ax*t + bx\]
\[y = ay*t + by\]
ax, bx, ay, by are constants to be determined
Let \(t\) range between \(t=0\), endpoint \((x_1,y_1)\) and \(t=1\), endpoint \((x_2,y_2)\)
Determining the constants: Use endpoint values
\[x_1 = ax*0 + bx \implies bx = x1\]
\[x_2 = ax*1 + bx \implies ax = x2-x1\]
So \[x = (x_2-x_1)*t + x_1, \quad 0<=t<=1\]
And \[y = (y_2-y_1)*t + y_1\]
Brute Force Line-Drawing Algorithm

Use “point-slope” form

Step in x direction, assume x2 > x1
(if x2 > x1, swap the points)

Compute \( m = \frac{y2-y1}{x2-x1} \)

\( \text{num-pts} = x2-x1+1 \)

\( x = x1 \)

Repeat num-pts times

\( y = m(x-x1) + y1 \)

SetPixel(x, round(y))

\( x = x+1 \)

Problem if \(|y2-y1| > |x2-x1|\)  -->  gaps

Solution: Step in y direction

\((1,0)\) 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!
Stepping in y direction

If \(|y_2-y_1| > |x_2-x_1|\), step in y, assume \(y_2 > y_1\)
(if \(y_1 > y_2\), swap the points):
Compute \(\text{inv}_m = (x_2-x_1)/(y_2-y_1)\)
\(\text{num-pts} = y_2-y_1+1\)
\(y = y_1\)
Repeat \(\text{num-pts}\) times
\(x = \text{inv}_m(y-y_1) + x_1\)
SetPixel(round(x), y)
\(y = y+1\)

Brute Force line algorithm, continued

- Vertical lines \((x_2 = x_1)\)
  \(y = y+1\) for each new pixel
  \(x\) doesn’t change
- Horizontal lines \((y_2 = y_1)\)
  \(x = x + 1\)
  \(y\) 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
- \( \frac{dy}{dx} = m \Rightarrow \Delta y/\Delta x = m \Rightarrow \Delta y = m \times \Delta x \)
- But \( \Delta y = y_{\text{new}} - y_{\text{old}} \)
- And \( \Delta x = x_{\text{new}} - x_{\text{old}} \)
  - So \( x_{\text{new}} = x_{\text{old}} + \Delta x \)
  - and \( y_{\text{new}} = y_{\text{old}} + \Delta y \)
  - i.e., \( y_{\text{new}} = y_{\text{old}} + m \times \Delta x \)
DDA, continued

- Choose $\Delta x = 1$
  - stepping in x direction
  - Pixel by pixel
- Then compute each new $y$ value
  $$y_{new} = y_{old} + m$$

DDA Algorithm

stepping in x, $x_2 > x_1$
(If $x_1 > x_2$, swap the points)

Compute $m = (y_2 - y_1)/(x_2 - x_1)$

num-pts = $x_2 - x_1 + 1$

x = $x_1$
y = $y_1$

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 $y$, $y_2 > y_1$
  – (if $y_1 > y_2$, swap the points):
    Compute $inv_m = (x_2-x_1)/(y_2-y_1)$
    $num-pts = y_2-y_1+1$
    $x = x_1$
    $y = y_1$
    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
- \( \text{delta}_y > \text{delta}_x \Rightarrow \) 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:

\[ y = m \times x + b \]
New \( y = s+1 \)
\[ s+1 = (\Delta y/\Delta x) \times r' + b \]
\[ r' = (s+1-b) \times \Delta x/\Delta y \]

So:
Criterion for choosing Pl:
\[ dl-dr = 2 \times r' - 2 \times r - 1 < 0 \]
\[ dl-dr = 2 \times (s+1-b) \times \Delta x/\Delta y - 2 \times r - 1 < 0 \]

Result:
\[ dl-dr = 2 \times (s^* + 1 - b) \times \Delta x/\Delta y - 2 \times r - 1 < 0 \]
If \( dl-dr \) is negative, choose "left" pixel
Multiply by \( \Delta y \) to get rid of divide operation
(always positive for Case-A lines)
Call result the "predictor", \( P \)
\[ P = \Delta y \times (dl-dr) \]
Result:
\[ P = 2 \times \Delta x \times (s+1-b) - 2 \times r \times \Delta y - \Delta y \]
Divide is gone--but it's still too complex
Bresenham's Contribution

- Try to find a recurrence relation for $P$
- Call $P_n$ the new value, and $P_0$ the old value
  - Then $P_n = P_0 + \Delta P$
- Call $s_n$ & $s_0$ the new & old values of $s$
- Call $r_n$ & $r_0$ the new & old values of $r$

Predictor $P$:
$$P = 2^*\Delta x^*(s+1-b) - 2^*r^*\Delta y - \Delta y$$

Change in Predictor:
$$\Delta P = P_n - P_0$$
$$P_n = P_0 + \Delta P$$

Point just plotted: $(r_0,s_0)$

Two cases for new point:
- Left case ($r_n=r_0$ and $s_n=s_0+1$)
- Right case ($r_n=r_0+1$ and $s_n=s_0+1$)

For both cases:
$$P_0 = 2^*\Delta x^*(s_0+1-b) - 2^*r_0^*\Delta y - \Delta y$$
Predictor $P$: $P=2^* \Delta x^*(s+1-b) - 2^*r^*\Delta y - \Delta y$

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

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

Subtracting $P_o$ from $P_n$ gives $\Delta P$

Result:

$\Delta P = 2^*\Delta x$

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

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

Again subtracting $P_o$ from $P_n$ gives $\Delta P$:

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

- 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 $P_0$ of the predictor

Substitute left-hand endpoint $(x_1,y_1)$ into predictor definition:

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

$\Rightarrow$

$$P_0 = 2\Delta x(y_1+1-b) -2x_1\Delta y - \Delta y$$

And use fact that $(x_1,y_1)$ is on line:

i.e., $y_1 = (\Delta y/\Delta x)x_1 + b$

$$P_0= 2\Delta x^*(\Delta y/\Delta x)x_1 + b +1 - b) -2x_1\Delta y - \Delta y$$

$$P_0=2\Delta yx_1 + 2\Delta x -2x_1\Delta y - \Delta y$$

Result: $P_0 = 2\Delta x - \Delta y$

---

**Case-A Bresenham Algorithm**

*(Steep positive slope)*

If $(x_1>x_2)$ swap endpoints;

$\text{del}_x = x_2-x_1; \; \text{del}_y = y_2-y_1;$

$P = 2\text{del}_x - \text{del}_y;$

$c\left\text{left} = 2\text{del}_x; \; c\text{right} = 2\text{del}_x - 2\text{del}_y;$

$x = x_1; \; y = y_1; \; \text{num}_pts = |\text{del}_y| + 1;$

Repeat $\text{num}_pts$ times

SetPixel$(x,y)$; $y = y + 1$;

If $(P < 0)$

$P = P + c\left\text{left}$;

Else

${P = P + c\text{right}; \; x = x + 1;}$
• Can be generalized to handle Case-C (steep negative slope) lines
• Compute $sdy = \text{sign}(\Delta y)$
  $= 1 \quad \text{if } y_2 > y_1$
  $= -1 \quad \text{if not}$
• Then, in definition of $P$ and $cright$:
  \begin{itemize}
  \item Replace $\Delta y$ with $sdy^*\Delta y$
  \item Replace $y = y + 1$ with $y = y + sdy$
  \end{itemize}
• 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) is 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 (computation intensive).
    - Can use statistical sampling instead.
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
\[ x_{\text{fin}} = 0.707 \times r \]
For (x=0; x<=xfin ; x++)
{
    y = \sqrt{r^2 - x^2};
    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:
\[ 2x + 2y \frac{dy}{dx} = 0 \]
\[ dy = \frac{-x}{y} dx \]
Step in x direction (dx=1)
\[ dy = \frac{-x}{y} \]
\[ y = y + dy \text{ (approximation)} \]

DDA Circle Algorithm

\[ x=0; \ y=r; \]
\[ x_{\text{fin}}=0.707*r; \]

while \( x \leq x_{\text{fin}} \)

\{
    \text{Set8Pixel(} \text{round} (x), \text{ round} (y));
    y = y - \frac{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:
  \[ f = x^2 + y^2 - r^2 \]
- \( f=0 \implies (x,y) \text{ is on circle} \)
- \( f<0 \implies (x,y) \text{ is inside circle} \)
- \( f>0 \implies (x,y) \text{ is outside circle} \)

- 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

- 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 \(\Delta P_k = 2x_k + 1\)

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

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

\[ P_{k+1} = ((x_{k+1} + 1)^2 + ((y_k - 1) - 1/2)^2 - r^2 \]

\[ P_{k+1} = ((x_{k+1} + 1)^2 + ((y_k - 1) - 3/2)^2 - r^2 \]

\[ P_{k+1} = x_k^2 + 4x_k + 4 + y_k^2 - 3y_k + 9/4 - r^2 \]

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:

\[ P_0 (x_0=0, y_0=r) \]
\[ P_0 = (x_0 + 1)^2 + (y_0 - 1/2)^2 - r^2 \]
\[ P_0 = 5/4 - r \quad \rightarrow \quad 1-r \quad \text{(rounding to integer)} \]

---

**Midpoint Circle Algorithm**

```plaintext
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;}
}
```