Binghamton University

EngiNet™

State University of New York

Thomas J. Watson

School of Engineering
and Applied Science

EngiNet™

WARNING

CS 560

Computer Graphics

Professor Richard Eckert

Lecture # 7

February 12, 2001

Scan Conversion Algorithms for 2D Output Primitives

- 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) & P2 (x2,y2)
  - Determine which pixels need to be painted
- Criteria
  - Straight as possible
  - Constant density (no gaps or bunching)
  - Density independent of orientation
  - Must be fast

Line Equations

- Differential equation:
  dy/dx = m (m=constant--the slope)
- Integrate (indefinite)
  y = m*x + constant
  The constant (b) 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

Parametric Form

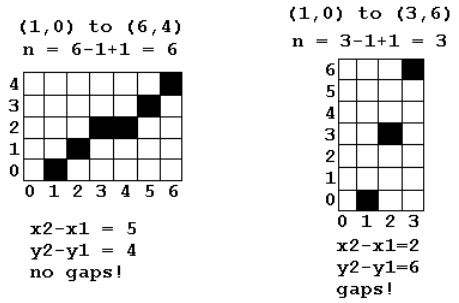
Express x and y linearly in terms of a paramter, t

x = ax*t + bx
y = ay*t + by
ax, bx, ay, by are constants to be determined
Let t range between endpoints 0 (x1,y1) and 1 (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

Step in x direction (x2>x1)
Compute m = (y2-y1)/(x2-x1)
num-pts = x2-x1+1
x = x1
Repeat num-pts times
    y = m*(x-x1) + y1
    SetPixel(round(x),round(y))
    x = x+1

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



Solution: Step in y direction

Solution: Step in y direction

If |y2-y1| > |x2-x1|, step in y (assume y2>y1):

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),round(y))
  y = y+1

Brute Force line algorithm, continued

Vertical lines (x2=x1)
Horizontal lines (y2=y1)

- Handle as separate cases

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 ==> $\Delta y/\Delta x = m$ ==> $\Delta y = m*\Delta x$
- But Δy = ynew - yold
- And Δx = xnew-xold
  - So xnew = xold + Δx
  - and ynew = yold + Δy
  - i.e., ynew = yold + m*Δx

DDA, continued

- Can choose $\Delta x = 1$
  - stepping in x direction
- Then compute each new y value
  ynew = yold + m

DDA Algorithm (stepping in x, x2>x1)

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, we can step in y
- DDA Algorithm (stepping in y, y2>y1):

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

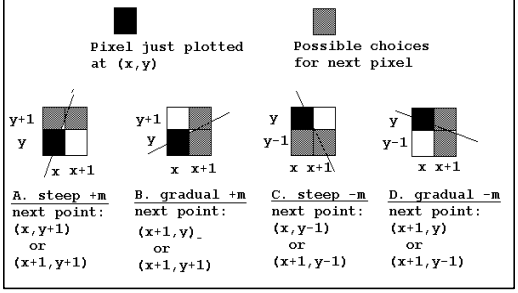
- Floating point multiply gone from loop
- But loop still has a floating point add
- And a round()
- WE CAN DO BETTER!
- Best performance:
  - Only integer adds/subtracts inside loop

Bresenham's Line-drawing Algorithm

- Used in most graphics packages
- Often implemented in hardware
- Also 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



- We need a "Predictor" function to choose which pixel is next
- Should use only simple integer math
- 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

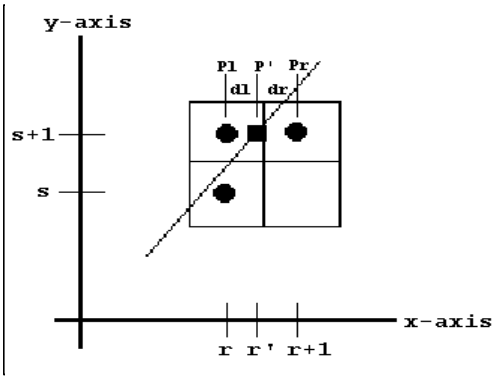
- Assume last pixel plotted (r,s)
  - Next pixel must be at (r,s+1)--"left" pixel
  - or at (r+1,s+1)--"right" pixel
- Mathematical point at (r', s+1)
- Define two horizontal distances

  dl=distance from math. Pt. to "left" pixel

  dr=distance from math. Pt. and "right" pixel

  dl = r'-r

  dr = r+1-r'



- 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: 2*r' - 2*r -1 < 0

Equation for line:

y = m*x + b

define $\Delta y=y2-y1$, $\Delta x=x2-x1$

Mathematical point lies on line, so:

$s+1 = (\Delta y/\Delta x)*r' + b$

$r' = (s+1-b)*\Delta x/\Delta y$

$dl-dr = 2*r' - 2*r -1 < 0$

So:

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

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

But this "predictor" is too complex

Multiply by Δy

(always positive for Case-A lines)

Call result the "predictor", P

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

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

Bresenham's Contribution

- We're trying to get new point from old point, so:
  - Maybe we can get a new predictor value from its old value in a simple way
  - Need 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

Change in Predictor:

ΔP = Pn - Po, so:

Pn = Po + ΔP

Two cases:

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

Left case:

Pn = 2*Δx*((so+1)+1-b) - 2*ro*Δy - Δy

Po = 2*Δx*(so+1-b) - 2*ro*Δy - Δy

Subtracting Po from Pn gives ΔP

Result:

ΔP = 2*Δx

Right case:

Pn = 2*Δx*((so+1)+1-b) - 2*(ro+1)*Δy - Δy

Po = 2*Δx*(so+1-b) - 2*ro*Δy - Δy

Again subtracting Po from Pn gives ΔP:

ΔP = 2*(Δx - Δy)

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

```
If P < 0    // left case
     P = P + 2*Δx
     x = x
     y = y + 1
If P>0      // right case
     P = P + 2*(Δx-Δy)
     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:

  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

```
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 WxH screen, Address = W*y + x
  - Multiply involved
- Bresenham: We know next pixel is one of two choices
- Faster to access frame buffer with addresses

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

Case A Line (gradual +m)