Chapter 8

Scan Conversion – Drawing on Raster Display (part 1 - Lines)

Midterm 1: Grade Distribution

Average: 84
Scan Conversion - Rasterization

- Convert continuous rendering primitives into discrete fragments/pixels
  - Lines
    - Bresenham (Midpoint)
  - Triangles
    - Implicit formulation
    - Scanline
    - Interpolation
- Discard geometry outside viewport window
Scan Conversion - Lines

Scan Conversion - Lines
Idea: Use Explicit Line Formula

Explicit - one coordinate as function of the others

\[ y = f(x) \]

\[ z = f(x, y) \]

line

\[ y = mx + b \]

\[ y = \frac{(y_2 - y_1)}{(x_2 - x_1)} (x - x_1) + y_1 \]

Typically separate into 4 (or 8) cases (why?)

---

Basic Line Drawing

Assume \( x_i < x_f \) & line slope absolute value is \( \leq 1 \)

```
Line ( x1, y1, x2, y2 )
begin
float dx, dy, x, y, slope ;
dx := x2 - x1 ;
dy := y2 - y1 ;
slope := dy/dx ;
y := y1 ;
for x from x1 to x2 do
begin
   PlotPixel ( x, Round ( y ) ) ;
   y := y + slope ;
end ;
end ;
```

Questions:
Can this algorithm use integer arithmetic?
Key Observation 1:
- At each step have ONLY 2 choices
  - East/North-East

Key Observation 2:
- Can decide based on whether midpoint is above/below line
- How?
  - Evaluate implicit line equation at 
    \((x+1, y+1/2)\)
Bresenham Algorithm

Implicit formulation = distance (up to scale)
\[
\tau = \{(x, y) | ax + by + c = xdy - ydx + c = 0\}
\]
\[
d(x, y) = 2(xdy - ydx + c)
\]
- Given point \( P = (x, y), d(x, y) \) is signed distance of \( P \) to \( \tau \) (up to scale)
- \( d \) is zero for \( P \in \tau \)

Bresenham (Midpoint) Algorithm

- Starting point satisfies \( d(x_1, y_1) = 0 \)
- Each step moves right (east) or upper right (northeast)
- Sign of \( d(x + 1; y + \frac{1}{2}) \) indicates if to move east or northeast
Bresenham (Midpoint) Algorithm

Line \((x_1, y_1, x_2, y_2)\)

begin
int \(x, y, dx, dy, d\);
\(x \leftarrow x_1;\) \(y \leftarrow y_1;\)
\(dx \leftarrow x_2 - x_1;\) \(dy \leftarrow y_2 - y_1;\)
PlotPixel \((x, y)\);
while \((x < x_2)\) do
    \(d = (2x + 2)dy - (2y + 1)dx + 2c;\) // \(2((x + 1)dy - (y + .5)dx + c)\)
    if \((d < 0)\) then
        \(x \leftarrow x + 1;\)
    else
        \(x \leftarrow x + 1;\)
        \(y \leftarrow y + 1;\)
    end;
PlotPixel \((x, y)\);
end;

Insanely efficient version (less computations inside the loop)
- compute \(d\) incrementally

At \((x_1, y_1)\)
\(d_{\text{start}} = d(x_1 + 1, y_1 + \frac{1}{2}) = 2dy - dx\)

Increment in \(d\) (after each step)
- If move east  \(\Delta_e = d(x + 2, y + \frac{1}{2}) - d(x + 1, y + \frac{1}{2}) -
                2((x + 2)dy - (y + \frac{1}{2})dx + c) - 2((x + 1)dy - (y + \frac{3}{2})dx + c) = 2dy\)
- If move northeast \(\Delta_{ne} = d(x_1 + 2, y_1 + \frac{3}{2}) - d(x_1 + 1, y_1 + \frac{1}{2}) =
                        2((x + 2)dy - (y + \frac{3}{2})dx + c) - 2((x + 1)dy - (y + \frac{1}{2})dx + c) = 2(dy - dx)\)
Bresenham (Midpoint) Algorithm

```plaintext
Line \((x_1, y_1, x_2, y_2)\)
begin
  int \(x, y, dx, dy, d, \Delta_x, \Delta_y\);
  \(x \leftarrow x_1;\) \(y \leftarrow y_1;\)
  \(dx \leftarrow x_2 - x_1;\) \(dy \leftarrow y_2 - y_1;\)
  \(d \leftarrow 2 \ast dy - dx;\)
  \(\Delta_x \leftarrow 2 \ast dy;\) \(\Delta_y \leftarrow 2 \ast (dy - dx);\)
PlotPixel \((x, y)\);
while \((x < x_2)\) do
  if \((d < 0)\) then
    begin
      \(d \leftarrow d + \Delta_x;\)
      \(x \leftarrow x + 1;\)
    end
  else begin
    \(d \leftarrow d + \Delta_y;\)
    \(x \leftarrow x + 1;\)
    \(y \leftarrow y + 1;\)
  end;
PlotPixel \((x, y)\);
end;
```

Bresenham Examples

- Intensity depends on angle

- Comment: extends to higher order curves – e.g. circles
Comparison: float/integer

Assume \( x_1 < x_2 \) & line slope is \( \leq 1 \)

```plaintext
Line ( x_1, y_1, x_2, y_2 )
begin
float dx, dy, x, y, slope ;
dx := x_2 - x_1 ;
dy := y_2 - y_1 ;
slope := dy / dx ;
y := y_1 ;
for x from x_1 to x_2 do
begin
  PlotPixel ( x, Round (y) ) ;
  y := y + slope ;
end ;
end ;
```

Implicit test

- Instead of clipping line in continuous space
- For each integer value of (x,y) test if inside window just before drawing
- Inefficient on CPU
- On a parallel (GPU) processor can be surprisingly fast

```plaintext
Line ( x_1, y_1, x_2, y_2 )
begin
float dx, dy, x, y, slope ;
dx := x_2 - x_1 ;
dy := y_2 - y_1 ;
slope := dy / dx ;
y := y_1 ;
for x from x_1 to x_2 do
begin
  if inside (x, y, int) then
  begin
    y := Round (y) ;
    PlotPixel ( x, y, int ) ;
    y := y + slope ;
  end ;
end ;
```
Scan Conversion of Lines

Discussion

- Integer: Bresenham
  - Good for hardware implementations (integer!)
- Floating Point
  - May be faster for software (depends on system)!
  - Easier to parallelize