CS130 : Computer Graphics
Lecture 4: Rasterizing 2D Lines

Tamar Shinar
Computer Science & Engineering
UC Riverside
1. **object-oriented**
   foreach object ...

2. **image-oriented**
   foreach pixel ...

there’s more than one way to do **object-oriented rendering** – e.g., OpenGL graphics pipeline vs. Renderman
rasterization - make fragments from clipped objects
clipping - clip objects to viewing volume
hidden surface removal - determine visible fragments
What is rasterization?

Rasterization is the process of determining which pixels are “covered” by the primitive
What is rasterization?

- input: primitives, output: fragments
- enumerate the pixels covered by a primitive
- interpolate attributes across the primitive

- output 1 fragment per pixel covered by the primitive

Figure 1. Block diagram of OpenGL.
Rasterization

Compute integer coordinates for pixels near the 2D primitives

Algorithms are invoked many, many times and so must be efficient

Output should be visually pleasing, for example, lines should have constant density

Obviously, they should be able to draw all possible 2D primitives
we’ll assume stuff has been converted to normalized device coordinates
Line Representation
Implicit Line Equation

\[ f(X) = N \cdot (X - X_0) = 0 \]

<whiteboard>: work out the implicit line equation in terms of X0 and X1
Line Drawing
Which pixels should be used to approximate a line?

Draw the thinnest possible line that has no gaps
Line drawing algorithm
(case: $0 < m \leq 1$)

\[ y = y_0 \]
\[ \text{for } x = x_0 \text{ to } x_1 \text{ do} \]
\[ \quad \text{draw}(x,y) \]
\[ \quad \text{if } (<\text{condition}> \text{) then} \]
\[ \quad y = y + 1 \]

• move from left to right
• choose between
  \( (x+1,y) \) and \( (x+1,y+1) \)

draw pixels from left to right, occasionally move up
Line drawing algorithm

(case: $0 < m \leq 1$)

\[
\begin{align*}
y &= y_0 \\
\text{for } x = x_0 \text{ to } x_1 \text{ do} & \quad \text{draw}(x, y) \\
& \quad \text{if } (<\text{condition}> \text{) then} \\
& \quad \quad y = y + 1
\end{align*}
\]

• move from left to right
• choose between \((x+1, y)\) and \((x+1, y+1)\)

draw pixels from left to right, occasionally move up
Use the midpoint between the two pixels to choose

If the line falls **below** the midpoint, use the bottom pixel
if the line falls **above** the midpoint, use the top pixel
Use the midpoint between the two pixels to choose

If the line falls **below** the midpoint, use the bottom pixel
If the line falls **above** the midpoint, use the top pixel
Use the midpoint between the two pixels to choose

If the line falls **below** the midpoint, use the bottom pixel

If the line falls **above** the midpoint, use the top pixel
Use the midpoint between the two pixels to choose

Implicit line equation:

\[ f(X) = N \cdot (X - X_0) = 0 \]

<whiteboard>
evaluate \( f \) at midpoint:

\[ f(x, y + \frac{1}{2}) \ ? 0 \]

<whiteboard>: work out the implicit line equation in terms of \( X_0 \) and \( X_1 \)

Question: will \( f(x, y + 1/2) \) be \( > 0 \) or \( < 0 \)?
Use the midpoint between the two pixels to choose

Implicit line equation:

\[ f(X) = N \cdot (X - X_0) = 0 \]

Evaluate \( f \) at midpoint:

\[ f(x, y + \frac{1}{2}) > 0 \]

This means midpoint is above the line \( \rightarrow \) line is closer to bottom pixel
Line drawing algorithm

(case: \(0 < m \leq 1\))

\[
y = y_0 \\
\text{for } x = x_0 \text{ to } x_1 \text{ do} \\
\quad \text{draw}(x,y) \\
\quad \text{if } (f(x+1, y + \frac{1}{2}) < 0) \text{ then} \\
\quad \quad y = y + 1
\]
We can make the Midpoint Algorithm more efficient

\[
y = y_0 \\
\text{for } x = x_0 \text{ to } x_1 \text{ do} \\
\quad \text{draw}(x,y) \\
\quad \text{if } (f(x + 1, y + \frac{1}{2}) < 0) \text{ then} \\
\quad \quad y = y + 1
\]

In each iteration we draw the current pixel and we evaluate the line equation at the next midpoint halfway above the current pixel.
We can make the Midpoint Algorithm more efficient by making it incremental!

Assume we have drawn the last red pixel and evaluated the line equation at the next (Red) midpoint. There are two possible outcomes:

1. we will choose the bottom pixel. In this case the next midpoint will be at the same level \((x +1,y)\)
2. we will choose the top pixel. In this case the next midpoint will be one level up \((x+1, y+1)\)

The line equation at these next midpoints can be evaluated incrementally using the update formulas shown:

\[
f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0
\]

\[
f(x + 1, y) = f(x, y) + (y_0 - y_1)
\]

\[
f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0)
\]
We can make the Midpoint Algorithm more efficient

As we move over one pixel to the right, we will choose either 
$(x+1,y)$ (yellow) or $(x+1,y+1)$ (green) and the next midpoint we will evaluate will be either:

$$f(x+1,y) = f(x,y) + (y_0 - y_1)$$

$$f(x+1,y+1) = f(x,y) + (y_0 - y_1) + (x_1 - x_0)$$

$$f(x+1, y + \frac{1}{2}) > 0$$

$$f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0$$
We can make the Midpoint Algorithm more efficient

As we move over one pixel to the right, we will choose either \((x+1,y)\) (yellow) or \((x+1,y+1)\) (green) and the next midpoint we will evaluate will be either:

\[
f(x + 1, y + \frac{1}{2}) < 0
\]

\[
f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0
\]

\[
f(x + 1, y) = f(x, y) + (y_0 - y_1)
\]

\[
f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0)
\]
We can make the Midpoint Algorithm more efficient

\[
y = y_0 \\
d = f(x_0+1, y_0+1/2) \\
\text{for } x = x_0 \text{ to } x_1 \text{ do} \\
\quad \text{draw}(x, y) \\
\quad \text{if } (d < 0) \text{ then} \\
\quad \quad y = y + 1 \\
\quad \quad d = d + (y_0 - y_1) + (x_1 - x_0) \\
\quad \text{else} \\
\quad \quad d = d + (y_0 - y_1)
\]

algorithm is \textit{incremental} and uses only \textit{integer arithmetic}.
Adapt Midpoint Algorithm for other cases

case: 0 < m <= 1
Adapt Midpoint Algorithm for other cases

case: $l \leq m$
Adapt Midpoint Algorithm for other cases

case: \(-1 \leq m < 0\)
Line drawing references

- the algorithm we just described is the *Midpoint Algorithm* (Pitteway, 1967), (van Aken and Novak, 1985)

- draws the same lines as the *Bresenham Line Algorithm* (Bresenham, 1965)