CS 450: COMPUTER GRAPHICS

RASTERIZING LINES

SPRING 2016

DR. MICHAEL J. REALE
OBJECT-ORDER RENDERING

• We going to start on how we will perform object-order rendering

• Object-order rendering
  • Go through each OBJECT \rightarrow find out what pixels are affected
  • Advantages: Much more efficient for large scenes with lots of objects
    • although how you access data will make this more or less efficient
  • Also called rendering by rasterization or scanline rendering
Rasterization = finding all pixels in an image that are occupied by a geometric primitive

- Also called “scan conversion”
- Example: have nice, continuous formula for a line $\Rightarrow$ pick “reasonable” pixels (discrete positions) to fill in to represent line

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

- **Graphics pipeline**
  - Generates (or renders) a 2D image given a 3D scene of objects
    - I.e., whole sequence of operations that takes us from (3D) OBJECTS $\rightarrow$ PIXELS IN IMAGE
  - Also called “graphics rendering pipeline” or just “the pipeline”
  - Composed of both hardware and software

- We’ll talk about the other steps in the pipeline in more detail later
  - For now, let’s assume that we have 2D representations of the primitives we want to draw (lines, circles, triangles, etc.)
    - Basically working in screen space
  - Picking the right pixels $\rightarrow$ job of the rasterization stage
• **Rasterization stage**
  • *Enumerates* pixels covered by primitive
  • *Interpolates* values (called **attributes**) across the primitive
    • Color would be an attribute
  • **Outputs fragments**
    • One fragment for each pixel covered by primitive
    • Each fragment “lives” at a particular pixel
    • Each fragment has its own set of attribute values

• Fragments then need to be processed (and possibly blended) to get each pixel’s final color
  • E.g., given two fragments from different objects but in the same location → which one do you draw? Do you blend them together?
  • Some include the fragment processing and blending stages as part of the rasterization stage
RASTERIZING LINES OVERVIEW
A POINT OF CLARITY

• In this case, we are talking about drawing/rasterizing **line segments**
  • That is, the lines have a defined beginning point and end point
To rasterize lines, we’ll discuss several algorithms:

- DDA (Digital differential analyzer)
- Midpoint
- Bresenham
A NOTE ABOUT EFFICIENCY

- As we discuss these algorithms, we will talk about their relative efficiency
- HOWEVER, hardware has changed → what is most efficient has changed

- Still true:
  - Should do fewer operations if you can
  - Multiply/divide more complex than add/subtract

- USED to be true:
  - Integer calculations were much faster than floating-point operations on older hardware
    - Bresenham → most efficient in this respect because it only uses integers
  - However, modern hardware doesn’t really have this problem anymore

- End of the day: need to actual profile your code on your hardware to see where the bottlenecks are
DDA LINE DRAWING ALGORITHM
SLOPE-INTERCEPT FORM

- As we’ve seen before, a straight line can be mathematically defined using the Cartesian **slope-intercept equation**:

  \[ y = mx + b \]

- We’re dealing with line segments, so these have specified starting and ending points:

  \[ (x_0, y_0) \]
  \[ (x_{\text{end}}, y_{\text{end}}) \]

- So, we can compute the slope \( m \) and the \( y \) intercept \( b \) as follows:

  \[ m = \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} \]
  \[ b = y_0 - mx_0 \]
INTERVALS

• Let’s say we have:

\[ \delta x = \text{change in } x = \text{interval in } x = "\text{run}" \]
\[ \delta y = \text{change in } y = \text{interval in } y = "\text{rise}" \]

• That means we can get \( \delta y \) from the slope \( m \) and \( \delta x \):

\[ \delta y = m \times \delta x = \frac{\text{rise}}{\text{run}} \times \text{run} = \text{rise} \]

• Similarly, we can get \( \delta x \) from \( \delta y \):

\[ \delta x = \frac{\delta y}{m} \]
DDA ALGORITHM

- Digital differential analyzer (DDA)
  - Scan-conversion line algorithm based on calculating either $\delta y$ or $\delta x$
  - Sample one coordinate at unit intervals $\rightarrow$ find nearest integer value for other coordinate

- Example: $0 < m \leq 1.0$ (slope positive, with $\delta x > \delta y$)
  - Increment $x$ in unit intervals ($\delta x = 1$)
  - Compute successive $y$ values as follows:
    $$ y_{k+1} = y_k + m $$
  - Round $y$ value to nearest integer
Problem: If slope is positive AND greater than 1.0 ($m > 1.0$), then we increment by $x \rightarrow$ skip pixels in $y$!

Solution: swap roles of $x$ and $y$!

- Increment $y$ in unit intervals ($\delta y = 1$)
- Compute successive $x$ values as follows:

$$x_{k+1} = x_k + \frac{1}{m}$$

Round $x$ value to nearest integer
DDA ALGORITHM: WHICH DO WE STEP IN?

• So, to summarize so far, which coordinate should be increment?

• Remember:

\[
m = \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} = \frac{\delta y}{\delta x}
\]

• If \(\text{abs}(\delta x) > \text{abs}(\delta y)\):
  • Step in \(X\)

• Otherwise:
  • Step in \(Y\)
DDA ALGORITHM: LINES IN REVERSE

• We’ve been assuming that the ending point has a coordinate value greater than the starting point:
  • Left to right, if incrementing \( x \)
  • Bottom to top, if incrementing \( y \)
• However, we could be going in reverse. If so, then:
  • If right to left, \( \delta x = -1 \)
  • If top to bottom, \( \delta y = -1 \)
DDA ALGORITHM: CODE

// Get dx and dy
int dx = x1 - x0;
int dy = y1 - y0;

int steps, k;
float xIncrement, yIncrement;

// Set starting point
float x = x0;
float y = y0;
DDA ALGORITHM: CODE

// Determine which coordinate we should step in
if (abs(dx) > abs(dy))
    steps = abs(dx);
else
    steps = abs(dy);

// Compute increments
xIncrement = float(dx) / float(steps);
yIncrement = float(dy) / float(steps);
// Let’s assume we have a magic function called setPixel(x,y) that sets a pixel at (x,y) to the appropriate color.
// Set value of pixel at starting point
setPixel(round(x), round(y));

// For each step...
for (k = 0; k < steps; k++) {
    // Increment both x and y
    x += xIncrement;
    y += yIncrement;

    // Set pixel to correct color
    // NOTE: we need to round off the values to integer locations
    setPixel(round(x), round(y));
}
DDA ALGORITHM: PROS AND CONS

• **Advantage:**
  • Faster than using the slope-intercept form directly ➔ no multiplication, only addition
    • *Caveat*: initial division necessary to get increments
  • Works for lines in all directions by default

• **Disadvantages:**
  • Accumulation of round-off error ➔ line can drift off true path
  • Rounding procedure time-consuming
MIDPOINT ALGORITHM
MIDPOINT ALGORITHM

• **DDA** → uses explicit form

• **Midpoint algorithm** → uses implicit form \( f(x,y) \)
  • Avoids rounding
  • *Basic idea:* test whether the point \( f(x+1, y + \frac{1}{2}) \) is inside or outside → choose closest point
  • Can be extended to other kinds of curves beyond lines
    • We’ll talk about some of these later...
CHOOSING THE NEXT PIXEL

• For starters, let’s assume our slope $m$ is (0, 1]
  • We’ll talk about the other three cases later...
• Thus, we’re incrementing in $x$
• Basically, if we’ve plotted a pixel $(x,y)$, we need to choose between:
  • $(x+1, y)$
  • $(x+1, y+1)$
• Question: which one is closer to the true line?
CHOOSE WISELY...

- Pseudocode-wise, this becomes:

\[
\begin{align*}
Y &= Y_0 \\
\text{For } x &= x_0 \text{ to } x_1 \\
\quad \text{draw}(x, y) \\
\quad \text{if (some condition)}: \\
\quad \quad y &= y + 1
\end{align*}
\]

- So, what is “some condition”?

https://prbydesignblog.files.wordpress.com/2014/11/last_crusade_choose_wisely.jpg
THE “MIDPOINT” PART...

• If we have an implicit equation for the line $f(x,y)$:

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

• Then:
  • $f(x,y) > 0 \rightarrow$ ABOVE line
  • $f(x,y) = 0 \rightarrow$ ON line
  • $f(x,y) < 0 \rightarrow$ BELOW line

• SO, we take a look at the \textit{midpoint} between our two choices: $f(x + 1, y + 0.5)$
  • $f(x + 1, y + 0.5) > 0 \rightarrow$ midpoint ABOVE line $\rightarrow$ BOTTOM point closer $\rightarrow$ choose $(x + 1, y)$
  • $f(x + 1, y + 0.5) < 0 \rightarrow$ midpoint BELOW line $\rightarrow$ TOP point closer $\rightarrow$ choose $(x + 1, y + 1)$
“SOME CONDITION”

• Thus, the pseudocode becomes:

\[
Y = Y_0 \\
\text{For } x = x_0 \text{ to } x_1 \\
\quad \text{draw}(x,y) \\
\quad \text{if } (f(x+1, y+0.5) < 0): \\
\quad \quad y = y + 1
\]
INCREMENTAL VERSION

- One problem: we are evaluating the function \( f(x,y) \) every iteration

- We can make an incremental version of the approach that will make it more efficient

- Let’s say our next point is going to be \((x,y)\) or \((x,y+1)\) \(\to\) need to evaluate \( f(x, y+0.5) \)

- Observation: we must have evaluated ONE of the following in the previous iteration:
  - \( f(x-1, y-0.5) \)
  - OR
  - \( f(x-1, y+0.5) \)
  - ...depending on whether we chose to increment \( y \) last time or not
Two interesting things to note about our implicit line equation:

\[
f(x+1, y) = (y_0 - y_1)(x+1) + (x_1 - x_0)y + x_0y_1 - x_1y_0
= (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 + (y_0 - y_1)
= f(x, y) + (y_0 - y_1)
\]

\[
f(x+1, y+1) = (y_0 - y_1)(x+1) + (x_1 - x_0)(y+1) + x_0y_1 - x_1y_0
= (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 + (y_0 - y_1) + (x_1 - x_0)
= f(x, y) + (y_0 - y_1) + (x_1 - x_0)
\]
INCREMENTAL VERSION

• This means that, if we already have $f(x,y)$, we can compute both $f(x+1,y)$ and $f(x+1,y+1)$ by adding on certain constant values:

\[
\begin{align*}
  f(x+1, y) &= f(x, y) + (y_0 - y_1) \\
  f(x+1, y+1) &= f(x, y) + (y_0 - y_1) + (x_1 - x_0)
\end{align*}
\]

• SO, that means, if we need $f(x, y+0.5)$:
  - DIDN’T increment $y$ last time $\Rightarrow$ have $f(x - 1, y + 0.5) \Rightarrow f(x, y+0.5) = f(x-1, y+0.5) + (y_0 - y_1)$
  - DID increment $y$ last time $\Rightarrow$ have $f(x - 1, y - 0.5) \Rightarrow f(x, y+0.5) = f(x-1, y-0.5) + (y_0 - y_1) + (x_1 - x_0)$
INCREMENT VERSION: ADDITIONAL EFFICIENCY

\[ f(x, y + 0.5) = f(x - 1, y + 0.5) + (y_0 - y_1) \]

\[ f(x, y + 0.5) = f(x - 1, y - 0.5) + (y_0 - y_1) + (x_1 - x_0) \]

• Can make faster by precomputing and storing:

\[
\begin{align*}
& (y_0 - y_1) \\
& (x_1 - x_0)
\end{align*}
\]
To keep track of this, we have a decision variable $d$:

\[
Y = Y_0 \\
d = f(x_0+1, y_0+0.5) \\
For \ x = x_0 \ to \ x_1 \\
\quad \text{draw}(x,y) \\
\quad \quad \text{if} \ (d < 0): \\
\quad \quad \quad y = y + 1 \\
\quad \quad \quad d = d + (x_1 - x_0) + (y_0 - y_1) \\
\quad \quad else \\
\quad \quad \quad d = d + (y_0 - y_1)
\]
MIDPOINT ALGORITHM: PROS AND CONS

• **Advantages:**
  - Like DDA → only using addition in main loop
  - **UNLIKE DDA:**
    - No rounding
    - No initial division needed
  - Extendable to other kinds of curves (circles, ellipses, etc.)

• **Disadvantages:**
  - If lines are VERY long → can accumulate floating point errors over time
    - Decision variable = still floating-point number
    - Usually not a big problem in practice
  - Still using floating point calculations → slower on older hardware
BRESENHAM’S ALGORITHM
INTRODUCTION

• The *Midpoint algorithm* still uses floating-point calculations for its decision variable

• *Bresenham’s algorithm* uses all integers
  • More efficient in days of yore
  • Slightly more involved derivation
  • Uses $y$ distances between candidate points and true line to make decision
BRESENHAM’S LINE ALGORITHM: THE IDEA

• We’ll start by assuming the slope is (0, 1] like before

• Similar to midpoint $\rightarrow$ look at a decision variable to help us make the choice at each step
  • However, derivation of variable is different...

• Our previous position: $(x_k, y_k)$

$$d_{lower} = y \text{ distance of } (x_k + 1, y_k) \text{ from true line coordinate } (x_k + 1, y)$$

$$d_{upper} = y \text{ distance of } (x_k + 1, y_k + 1) \text{ from true line coordinate } (x_k + 1, y)$$
BRESENHAM’S LINE ALGORITHM: MATH

• The value of y for the mathematical line at \(x_k + 1\) is given by:

\[
y = m(x_k + 1) + b
\]

• Ergo:

\[
d_{\text{lower}} = y - y_k
\]
\[
= m(x_k + 1) + b - y_k
\]

\[
d_{\text{upper}} = (y_k + 1) - y
\]
\[
= y_k + 1 - m(x_k + 1) - b
\]
To determine which of the two pixels is closer to the true line path, we can look at the sign of the following:

\[
d_{\text{lower}} - d_{\text{upper}} = (m(x_k + 1) + b - y_k) - (y_k + 1 - m(x_k + 1) - b)
\]

\[
= m(x_k + 1) + b - y_k - y_k - 1 + m(x_k + 1) + b
\]

\[
= 2m(x_k + 1) - 2y_k + 2b - 1
\]

- Positive \(\rightarrow\) \(d_{\text{upper}}\) is smaller \(\rightarrow\) choose \((y_k + 1)\)
- Negative \(\rightarrow\) \(d_{\text{lower}}\) is smaller \(\rightarrow\) choose \(y_k\)
BRESENHAM’S LINE ALGORITHM: EVEN MORE MATH

• Remember that:

\[ m = \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} = \frac{\delta y}{\delta x} = \frac{\Delta y}{\Delta x} \]

• Substituting with our current equation:

\[ d_{\text{lower}} - d_{\text{upper}} = 2 \frac{\Delta y}{\Delta x} (x_k + 1) - 2y_k + 2b - 1 \]

• We will let our decision variable \( p_k \) be the following:

\[
p_k = \Delta x (d_{\text{lower}} - d_{\text{upper}}) \\
= \Delta x \left( 2 \frac{\Delta y}{\Delta x} (x_k + 1) - 2y_k + 2b - 1 \right) \\
= 2\Delta y (x_k + 1) - 2\Delta x \cdot y_k + \Delta x (2b - 1) \\
= 2\Delta y \cdot x_k + 2\Delta y - 2\Delta x \cdot y_k + \Delta x (2b - 1) \\
= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + 2\Delta y + \Delta x (2b - 1) \\
= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c
\]
BRESENHAM’S LINE ALGORITHM: DECISION VARIABLE

\[ p_k = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \]

- Multiplying by \( \Delta x \) won’t affect the sign of \( p_k \), since \( \Delta x > 0 \)
- Note that constant \( c \) does not depend on the current position at all, so can compute it ahead of time:
  \[ c = 2\Delta y + \Delta x(2b - 1) \]
- So, as before:
  - \( p_k \) positive \( \Rightarrow \) \( d_{\text{upper}} \) is smaller \( \Rightarrow \) choose \( (y_k + 1) \)
  - \( p_k \) negative \( \Rightarrow \) \( d_{\text{lower}} \) is smaller \( \Rightarrow \) choose \( y_k \)
BRESENHAM’S LINE ALGORITHM: UPDATING $P_k$

- We can get the next value of the decision variable (i.e., $p_{k+1}$) using $p_k$

\[
p_k = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c
\]
\[
p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c
\]
\[
p_{k+1} - p_k = (2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c) - (2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c)
\]
\[
= 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c - 2\Delta y \cdot x_k + 2\Delta x \cdot y_k - c
\]
\[
= 2\Delta y \cdot x_{k+1} - 2\Delta y \cdot x_k - 2\Delta x \cdot y_{k+1} + 2\Delta x \cdot y_k
\]
\[
= 2\Delta y (x_{k+1} - x_k) - 2\Delta x (y_{k+1} - y_k)
\]
BRESENHAM’S LINE ALGORITHM: UPDATING $P_K$

• However, we know:  
  $$x_{k+1} = x_k + 1$$

• Therefore:  
  $$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

  $$= 2\Delta y(x_k + 1 - x_k) - 2\Delta x(y_{k+1} - y_k)$$

  $$= 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

• So, we need to determine what $(y_{k+1} - y_k)$ was:
  
  • If $p_k$ was positive $\rightarrow (y_{k+1} - y_k) = 1 \rightarrow$
    $$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

  • If $p_k$ was negative $\rightarrow (y_{k+1} - y_k) = 0 \rightarrow$
    $$p_{k+1} = p_k + 2\Delta y$$
BRESENHAM’S LINE ALGORITHM: SUMMARIZED

1. Input two line endpoints and store LEFT endpoint in \((x_0, y_0)\)
   - Allows us to deal with lines going in the OPPOSITE direction!
2. Plot first point \((x_0, y_0)\)
3. Compute constants \(\Delta x, \Delta y, 2\Delta y, \text{ and } 2\Delta y - 2\Delta x\).
   Also compute first value of decision variable: \(p_0 = 2\Delta y - \Delta x\)
4. At each \(x_k\), test \(p_k\):
   - If \(p_k < 0\):
     plot \((x_k + 1, y_k)\)
     \(p_{k+1} = p_k + 2\Delta y\)
   - Otherwise:
     plot \((x_k + 1, y_k + 1)\)
     \(p_{k+1} = p_k + 2\Delta y - 2\Delta x\)
5. Perform step 4 \((\Delta x - 1)\) times

NOTE: Effectively assuming line starts at \((0,0)\) and thus \(b = 0\)

\[
p_0 = 2\Delta y \cdot x_0 - 2\Delta x \cdot y_0 + 2\Delta y + \Delta x(2b - 1) \\
= 2\Delta y(0) - 2\Delta x(0) + 2\Delta y + \Delta x(2(0) - 1) \\
= 2\Delta y + \Delta x(-1) \\
= 2\Delta y - \Delta x
\]

NOTE: This version ONLY works with \(0 < |m| < 1.0!!!\)
Ergo, \(\Delta x\) and \(\Delta y\) are positive here!
COMPARISON TO MIDPOINT

Bresenham’s decision variable:
- Starting value:
- Increment if $p > 0$
- Increment cases:
  - Incremented $y$ last time:
  - Did NOT increment $y$ last time:

\[ p_k = 2\Delta y - \Delta x \]

\[ p_{k+1} = p_k + 2\Delta y - 2\Delta x \]

\[ p_{k+1} = p_k + 2\Delta y \]

Midpoint decision variable:
- Starting value:
- Increment if $d < 0$
- Increment cases:
  - Incremented $y$ last time:
  - Did NOT increment $y$ last time:

\[ f(x+1, y + 0.5) = f(x, y) + (y_0 - y_1) + (x_1 - x_0) \times 0.5 \]

\[ = -\Delta y + \Delta x \times 0.5 \]

\[ d_{k+1} = d_k + (y_0 - y_1) + (x_1 - x_0) = -\Delta y + \Delta x \]

\[ d_{k+1} = d_k + (y_0 - y_1) = d_k - \Delta y \]

Bresenham:
- Everything multiplied by 2 $\rightarrow$ removes floating point
- Decision variable (and increments) flipped
// NOTE: dx and dy are ABSOLUTE VALUES in this code

int dx = fabs(x1 - x0);
int dy = fabs(y1 - x0);

int p = 2*dy - dx;
int twoDy = 2*dy;
int twoDyMinusDx = 2*(dy - dx);

int x,y;
BRESENHAM’S LINE ALGORITHM: CODE

// Determine which endpoint to use as start position
if(x0 > x1) {
    x = x1;
    y = y1;
    x1 = x0;
} else {
    x = x0;
    y = y0;
}

// Plot first pixel
setPixel(x,y);
BRESENHAM’S LINE ALGORITHM: CODE

while(x < x1) {
    x++;
    if(p < 0)
        p += twoDy;
    else {
        y++;
        p += twoDyMinusDx;
    }
    setPixel(x,y);
}
DRAWING LINES IN ALL DIRECTIONS
• For Midpoint and Bresenham, we have thus far assumed slope $m$ is within $(0, 1]$
  • *Minor exception*: our current version of Bresenham allows us to draw lines in reverse in $x$, but the slope is still positive
• To draw lines in ALL directions, we’ll have to employ a few tricks...
  • We will be discussing the tricks with Bresenham in mind, but you can use similar strategies for Midpoint as well...
LINE DRAWING GENERALIZED

- Abs(change in Y) > abs(change in X)?
  - Swap roles of X and Y
- Is X going in negative direction?
  - Swap endpoints
- Set starting Y coordinate
- Is Y going in negative direction?
  - Set Y increment to -1
  - Swap Y coordinates of endpoints
  - Basic idea: We will start at the original Y0 and we will decrement Y as necessary. HOWEVER, we will PRETEND like we are drawing a line that is going up → decisions will be the same (e.g., keep current Y, change Y)
- Calculate first decision variable and decision variable increments
abs(dx) < abs(dy) $\Rightarrow$ swap x and y

Increment y
Subtract from x

Increment y
Add to x

Increment x
Add to y

Increment x
Subtract from y

Swap endpoints, then:
Increment x
Add to y

Swap endpoints, then:
Increment y
Add to x

Swap endpoints, then:
Increment y
Subtract from x

Swap endpoints, then:
Increment x
Subtract from y
SPECIAL CASES IN LINE DRAWING

• To save time, if you have a line that is:
  • \( \Delta x = 0 \) (vertical)
  • \( \Delta y = 0 \) (horizontal)
  • \( |\Delta x| = |\Delta y| \) (diagonal)
  • ...you can just draw it directly without going through the entire algorithm.
ALGORITHM COMPARISON

- **DDA (Digital differential analyzer)**
  - Not very efficient → rounding
  - Very straightforward
  - Can drift because of round-off error
- **Midpoint**
  - Efficient
    - No rounding, no initial division like in DDA
  - Mostly straightforward
  - Uses minimal floating-point calculations
  - Can drift, but only for very long lines
- **Bresenham**
  - Efficient
  - More involved to derive
  - Uses all integers