CS 548: COMPUTER GRAPHICS
DRAWING LINES AND CIRCLES

SPRING 2015
DR. MICHAEL J. REALE
OPENGL POINTS AND LINES
In OpenGL, there are different constants used to indicate what kind of primitive we are trying to draw.

- For points, we have GL_POINTS.
- For lines, we have GL_LINES, GL_LINE_STRIP, and GL_LINE_LOOP.
DRAWING POINTS: LEGACY VS. NEW

• To draw points in legacy OpenGL:

  • `glBegin(GL_POINTS);`
  • `glVertex3f(1,5,0);`
  • `glVertex3f(2,3,1);`
  • ...
  • `glEnd();`

• To draw points in OpenGL 3.0+, we set up our buffers and then call:

  • `glDrawArrays(GL_POINTS, 0, pointCnt);`

• For other primitives (like lines), the procedure would be the same → just replace GL_POINTS with what you want instead.
DRAWING LINES

- Depending on which constant we choose, the lines will be drawn differently
  - **GL_LINES** = draw individual lines with every 2 vertices
  - **GL_LINE_STRIP** = draw a polyline connecting all vertices in sequence
  - **GL_LINE_LOOP** = draw a polyline, and then also connect the first and last vertices
DRAWING LINES

• `glBegin(GL_LINES);`
  • `glVertex2iv(p1)`
  • `glVertex2iv(p2);`
  • `glVertex2iv(p3);`
  • `glVertex2iv(p4);`
  • `glVertex2iv(p5);`
• `glEnd();`

• NOTE: p5 is ignored, since there isn’t another vertex to pair it with

• ALSO NOTE: the “v” part means passing in an array
DRAWING LINES

- `glBegin(GL_LINE_STRIP);`
  - `glVertex2iv(p1)`
  - `glVertex2iv(p2)`
  - `glVertex2iv(p3)`
  - `glVertex2iv(p4)`
  - `glVertex2iv(p5)`
- `glEnd();`

  Creates a **polyline**
DRAWING LINES

- `glBegin(GL_LINE_LOOP);`
  - `glVertex2iv(p1)`
  - `glVertex2iv(p2)`
  - `glVertex2iv(p3)`
  - `glVertex2iv(p4)`
  - `glVertex2iv(p5)`
- `glEnd();`

- Creates a **closed polyline**
DDA LINE DRAWING ALGORITHM
THE PROBLEM WITH PIXELS

• Pretty much all modern displays use a grid of pixels to display images
  • Discrete (digital) locations

• **Problem**: when drawing lines that aren’t perfectly horizontal, vertical, or diagonal, the points in the middle of the line do not fall perfectly into the pixel locations
  • I.e., have to round line coordinates to integers
  • Lines end up having this stair-step effect ("jaggies") → **aliasing**
LINE EQUATIONS

• 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:

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

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

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

• For a given $x$ interval $\delta x$ along a line, we can compute the corresponding $y$ interval $\delta y$:

$$\delta y = m \cdot \delta x$$

• 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 dy or dx
  • Sample one coordinate at unit intervals → 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
DDA ALGORITHM: M > 1.0

- **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:
  - Round \(x\) value to nearest integer

\[
x_{k+1} = x_k + \frac{1}{m}
\]
So, to summarize so far, which coordinate should be incremented?

Remember:

If \( |dx| > |dy| \):
- Step in X

Otherwise:
- Step in Y

We’ll see later in another algorithm an easy way to do this is to swap the roles of \( x \) and \( y \)
- So “\( x \)” is really \( y \), and “\( y \)” is really \( x \)
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

- **Disadvantages:**
  - Accumulation of round-off error can cause the line to drift off true path
  - Rounding procedure still time-consuming

- **Question:** can we do this with nothing but integers?
YES

- Yes, we can.
BRESENHAM’S LINE DRAWING ALGORITHM
BRESENHAM’S LINE ALGORITHM: INTRODUCTION

• Takes advantage of fact that slope m is really a fraction of integers
• Say we have a positive slope and \( dx > dy \) (so we’re incrementing in x)
• We have a pixel plotted at \((x, y)\)
• Given \( x + 1 \), the next y value is going to be either:
  • \( y \)
  • \( y + 1 \)
• The question is: which one is closer to the real line? \((x+1, y)\) or \((x+1, y+1)\)?
BRESENMHAM’S LINE ALGORITHM: THE IDEA

• The basic idea is to look at a decision variable to help us make the choice at each step

• Our previous position: \((x_k, y_k)\)

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

• \(d_{upper} = \text{distance of } (x_k + 1, y_k + 1) \text{ from the true line coordinate } (x_k + 1, y)\)
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:

$$d_{lower} - d_{upper} = (m(x_k + 1) + b - y_k) - (y_k + 1 - m(x_k + 1) - b)$$

$$= m(x_k + 1) + b - y_k - y_k - 1 + m(x_k + 1) + b$$

$$= 2m(x_k + 1) - 2y_k + 2b - 1$$

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

• Remember that:

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

• Substituting with our current equation:

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

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

\[
p_k = \Delta x (d_{lower} - d_{upper})
\]

\[
= \Delta x \left( 2 \frac{\Delta y}{\Delta x} (x_k + 1) - 2y_k + 2b - 1 \right)
\]

\[
= 2\Delta y (x_k + 1) - 2\Delta x \cdot y_k + \Delta x (2b - 1)
\]

\[
= 2\Delta y \cdot x_k + 2\Delta y - 2\Delta x \cdot y_k + \Delta x (2b - 1)
\]

\[
= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + 2\Delta y + \Delta x (2b - 1)
\]

\[
= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c
\]
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 \((x_0, y_0)\)
2. Plot first point \((x_0, y_0)\)
3. Compute constants \(\Delta x\), \(\Delta y\), \(2\Delta y\), 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 \(p_k < 0\) \(\rightarrow\) plot \((x_k + 1, y_k)\) \(\rightarrow\) \(p_{k+1} = p_k + 2\Delta y\)
   - Otherwise \(\rightarrow\) plot \((x_k + 1, y_k + 1)\) \(\rightarrow\) \(p_{k+1} = p_k + 2\Delta y - 2\Delta x\)
5. Perform step 4 \((\Delta x - 1)\) times

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

\[p_k = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + 2\Delta y + \Delta x(2b - 1)\]
\[= 2\Delta y(0) - 2\Delta x(0) + 2\Delta y + \Delta x(2(0) - 1)\]
\[= 2\Delta y + \Delta x(-1)\]
\[= 2\Delta y - \Delta x\]
BRESENHAM’S LINE ALGORITHM: CODE

// 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;
// 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);
while(x < x1) {
    x++;

    if(p < 0)
        p += twoDy;
    else {
        y++;
        p += twoDyMinusDx;
    }
    setPixel(x,y);
}
BRESENHAM’S LINE ALGORITHM: GENERALIZED

• What we’re talked about only works with $0 < |m| < 1.0$
• For other slopes, we take advantage of symmetry:
  • If $dy > dx \rightarrow$ swap x and y
    • WARNING: Would then need to call setPixel(y, x)
  • After swapping endpoints and potentially swapping x and y, if “y0” > “y1” $\rightarrow$ decrement “y” rather than increment
    • NOTE: “y” may actually be x if you swapped them
• Two more warnings:
  • 1) In the sample code, dx and dy are ABSOLUTE VALUES
  • 2) In the next image, when I say x and y, I mean the actual x and y
abs(dx) < abs(dy) \Rightarrow \text{swap x and y}

- Increment y
- Subtract from x

- Increment x
- Add to y

abs(dx) > abs(dy)

- Swap endpoints, then:
  - Increment x
  - Subtract from y

- Swap endpoints, then:
  - Increment y
  - Add to x

- Increment x
- Subtract from y

abs(dx) > abs(dy) \Rightarrow \text{swap x and y}
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.
A more general way of viewing Bresenham’s Algorithm that can be applied to other conics is called the **Midpoint Algorithm**:

- Uses the implicit representation \( F(x, y) = Ax + By + C = 0 \) → e.g., implicit line equation
  - If a point is on the “inside”, \( F(x, y) < 0 \)
  - If a point is on the “outside”, \( F(x, y) > 0 \)
  - If a point is exactly on the boundary, \( F(x, y) = 0 \)
- Test whether the point \( F(x+1, y + \frac{1}{2}) \) is inside or outside → choose closest point
MIDPOINT CIRCLE ALGORITHM
INTRODUCTION

• A circle can be defined by its implicit form:
  \[ F(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2 = 0 \]

  \[ (x_c, y_c) \rightarrow \text{center of circle} \]
  \[ r = \text{radius} \]

• Since a circle is symmetric in all 8 octants \( \rightarrow \)
  just compute one octant and replicate in others
DECISION VARIABLE

- Assume the circle is centered at (0,0)
- We’re going to start by:
  - Incrementing x by 1
  - Choose whether to go down or not in y
- To determine our next choice, we will look at our decision variable based on the midpoint \((x_k + 1, y_k - \frac{1}{2})\):

\[
p_k = F\left(x_k + 1, y_k - \frac{1}{2}\right) = (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2
\]

- If \(p_k < 0\) \(\rightarrow\) midpoint inside circle \(\rightarrow\) choose \(y_k\)
- If \(p_k > 0\) \(\rightarrow\) midpoint outside circle \(\rightarrow\) choose \(y_k - 1\)
To figure out how to update the decision variable, let’s look at the next value:

\[ p_{k+1} = F\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right) \]

\[ = (x_{k+1} + 1)^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2 \]
HOLD ON TO YOUR MATHEMATICAL HATS...

\[ p_{k+1} = [(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2 \]

\[ = [x_k + 2]^2 + y_{k+1}^2 - y_{k+1} + \frac{1}{4} - r^2 \]

\[ = x_k^2 + 4x_k + 4 + y_{k+1}^2 - y_{k+1} + \frac{1}{4} - r^2 \]

\[ = \left(x_k^2 + 2x_k + 1\right) + 2x_k + 3 + \left(y_{k+1}^2 - y_{k+1} + \frac{1}{4} + \left(y_k^2 - y_k + \frac{1}{4}\right) - \left(y_k^2 - y_k + \frac{1}{4}\right)\right) - r^2 \]

\[ = (x_k + 1)^2 + 2x_k + 2 + 1 + \left(y_{k+1}^2 - y_{k+1} + \frac{1}{4} + \left(y_k^2 - y_k + \frac{1}{2}\right)^2 - y_k^2 + y_k - \frac{1}{4}\right) - r^2 \]

\[ = (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 + 2(x_k + 1) + 1 + \left(y_{k+1}^2 - y_{k+1} + \frac{1}{4} - y_k^2 + y_k - \frac{1}{4}\right) \]

\[ = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \]

Remember:

\[ p_k = F \left(x_k + 1, y_k - \frac{1}{2}\right) \]

\[ = (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 \]
**NEXT DECISION VARIABLE**

\[ p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \]

- \( y_{k+1} \) is:
  - \( y_k \) if \( p_k < 0 \) \( \rightarrow \)

\[
p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1
  = p_k + 2(x_k + 1) + 1
  = p_k + 2x_{k+1} + 1
\]

- \((y_k - 1)\) if \( p_k > 0 \) \( \rightarrow \)

\[
p_{k+1} = p_k + 2(x_k + 1) + ((y_k - 1)^2 - y_k^2) - (y_k - 1 - y_k) + 1
  = p_k + 2(x_k + 1) + (y_k^2 - 2y_k + 1 - y_k^2) - (y_k - 1 - y_k) + 1
  = p_k + 2x_{k+1} + 1 - 2y_k + 2
  = p_k + 2x_{k+1} + 1 - 2(y_k - 1)
  = p_k + 2x_{k+1} + 1 - 2y_{k+1}
\]
UPDATING THE DECISION VARIABLE

- If $p_k < 0 \rightarrow$ add $2x_{k+1} + 1$
- If $p_k > 0 \rightarrow$ add $2x_{k+1} + 1 - 2y_{k+1}$

- We can also update $2x_{k+1}$ and $2y_{k+1}$ incrementally:

\[
2x_{k+1} = 2(x_k + 1) = 2x_k + 2
\]
\[
2y_{k+1} = 2(y_k - 1) = 2y_k - 2
\]
INITIAL VALUES

• We will start at (0,r)
• Initial decision variable value:

\[
p_0 = F\left(1, r - \frac{1}{2}\right) = (1)^2 + \left(r - \frac{1}{2}\right)^2 - r^2 = 1 + \left(r^2 - r + \frac{1}{4}\right) - r^2
\]

\[
= \frac{5}{4} - r
\]

• However, if our radius r is an integer, we can round \(p_0\) to \(p_0 = 1 - r\), since all increments are integers
MIDPOINT CIRCLE ALGORITHM SUMMARIZED

1. Input radius $r$ and circle center $(x_c, y_c)$; first point = $(x_0, y_0) = (0, r)$

2. Calculate initial decision variable value:

3. At each $x_k$, test $p_k$:
   - If $p_k < 0 \rightarrow$ next point is $(x_{k+1}, y_k)$ and:
   - Otherwise $\rightarrow$ next point is $(x_{k+1}, y_k - 1)$ and:

4. For each calculated position $(x, y)$, plot $(x + x_c, y + y_c)$

5. Plot corresponding symmetric points in other seven octants

6. Repeat steps 3 through 5 UNTIL $x >= r$

Where:

- $2x_{k+1} = 2x_k + 2$
- $2y_{k+1} = 2y_k - 2$

$p_0 = \frac{5}{4} - r$

$p_{k+1} = p_k + 2x_{k+1} + 1$

$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$