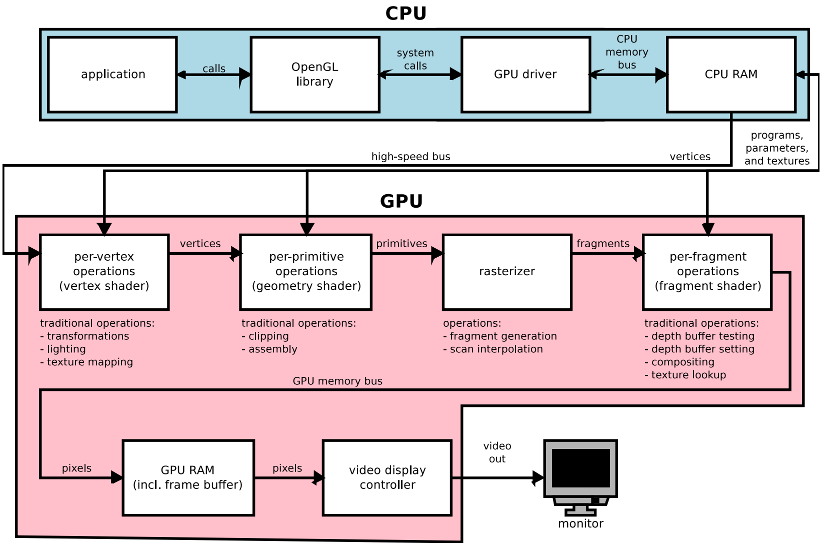
CptS 442/542 (Computer Graphics) Unit 3: Line Drawing

Bob Lewis

School of Engineering and Applied Sciences
Washington State University

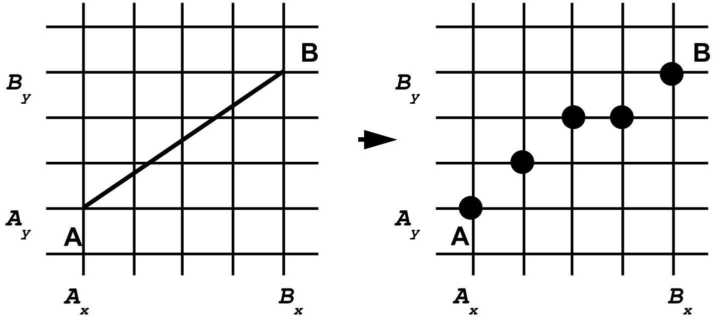
Fall, 2017

Review: The OpenGL Rendering Pipeline



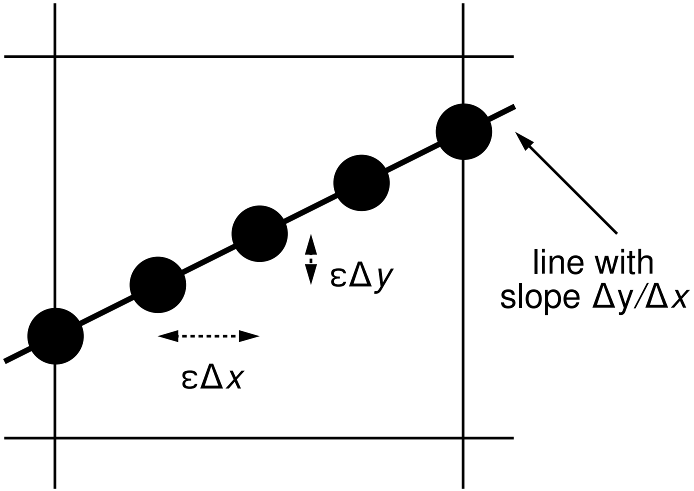
Abstracting the Line Rasterization Problem

Starting with two endpoints on an integer grid, find a representation that consists of “lit” pixels:

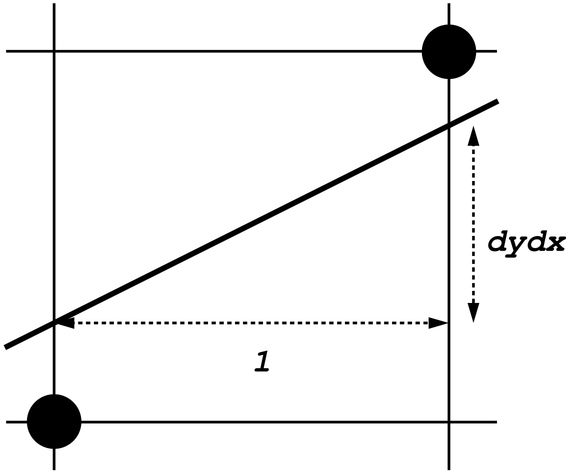


The only graphics primitive used is `setPixel(x,y)`.

The Ideal Digital Differential Analyzer (DDA)



The Practical DDA



A Very Simple DDA Implementation

Suppose our only graphics primitive was `setPixel(x, y)` Let's take the naive approach to draw a line from A to B:

```
void line(int Ax, int Ay, int Bx, int By)
{
    double y, dydx;
    int x;

    y = Ay;
    dydx = ((double) (By - Ay))/(Bx - Ax);
    for (x = Ax; x <= Bx; x++) {
        setPixel(x, round(y));
        y += dydx;
    }
}
```

What's wrong with this?

A Better DDA

Here's Newman & Sproull's "simple DDA" (in C):

```
void line(int Ax, int Ay, int Bx, int By) {
    double x, y, incrX, incrY;
    int length, i;

    length = abs(Bx - Ax);
    if (abs(By - Ay) > length)
        length = abs(By - Ay);
    incrX = ((double) (Bx - Ax)) / length;
    incrY = ((double) (By - Ay)) / length;
    x = Ax; y = Ay;
    for (i = 0; i <= length; i++) {
        setPixel(round(x), round(y));
        x += incrX; y += incrY;
    }
}
```

The Simple DDA

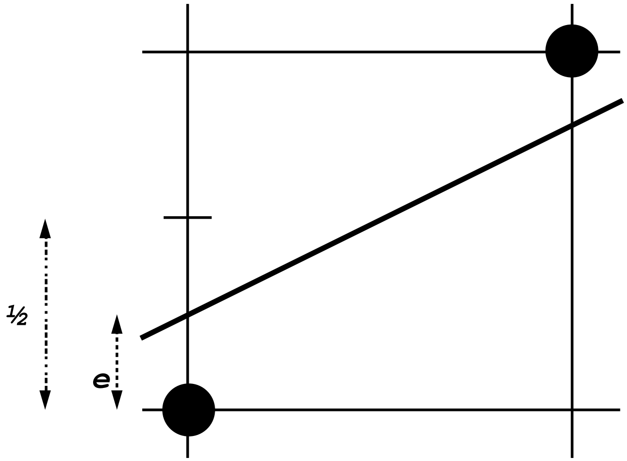
Good Points

- easy to remember
- works in all quadrants

Not-So-Good Points

- floating point
- wastes time adding 1.0 to one value

Bresenham (1965)



Bresenham's Algorithm I

(This is the original.)

```
void line(int Ax, int Ay, int Bx, int By) {
    double e, dydx;
    int x, y;

    dydx = ((double) (By - Ay)) / (Bx - Ax);
    e = dydx;
    y = Ay;
    for (x = Ax; x <= Bx; x++) {
        setPixel(x, y);
        if (e > 0.5) {
            y++;
            e -= 1.0;
        }
        e += dydx;
    }
}
```

Bresenham's Algorithm II

What's wrong with this?

- uses floating point
- only works in one octant

Even though we can remove the floating point (as Bresenham did), an alternative to the original Bresenham that sets the same pixels for lines (and circles) but is of more general use is...

The Midpoint Algorithm: The Implicit Function

Start with an *implicit function* $f(\mathbf{P})$ ($\equiv f(P_x, P_y)$) that evaluates a point $\mathbf{P}$

- $f(\mathbf{P}) > 0$ inside the object
- $f(\mathbf{P}) = 0$ on the boundary of the object
- $f(\mathbf{P}) < 0$ outside the object

(Implicit functions are *very* useful critters in graphics.) In the case of a line, the implicit function we want is:

$$f(\mathbf{P}) = aP_x + bP_y + c$$

The Midpoint Algorithm: Finding the Implicit Equation

Given two endpoints **A** and **B**, we can determine a and b as follows:

$$f(\mathbf{A}) = aA_x + bA_y + c \qquad f(\mathbf{B}) = aB_x + bB_y + c$$

(Both equations are 0 since **A** and **B** are on the line.) Define

$$W = B_x - A_x \qquad H = B_y - A_y$$

Subtract the left equation the from the right equation to get

$$aW + bH = 0$$

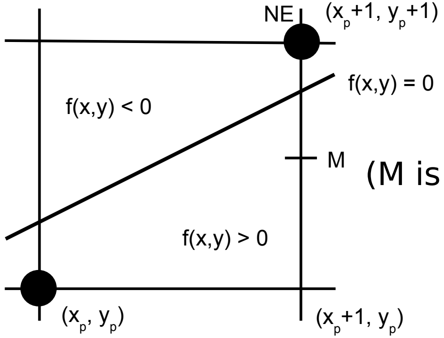
This has a solution (infinitely many, in fact):

$$a = H \qquad b = -W$$

Since **A**'s and **B**'s coordinates are integers, so are a and b.
We chose $b < 0$ so that $f(\mathbf{P}) < 0$ for **P** above the line.
What happened to c?

The Midpoint Algorithm: Reducing the Problem

By symmetry, we're going to only deal with one octant.



(M is the midpoint)

The essential observation:

"Move E or move NE?" $\equiv$ "Is **M** above the line or below it?"

The Midpoint Algorithm: The Decision Variable

F is a *decision variable*:

$$\begin{aligned} F &= 2f(\mathbf{M}) \\ &= 2f(x_p + 1, y_p + 1/2) \\ &= 2a(x_p + 1) + 2b(y_p + 1/2) + 2c \end{aligned}$$

(Note the factor of 2. It doesn't change the sign of **F**, but why is it there? We'll see below.)
So algorithm becomes:

If $F > 0$, move NE. Otherwise, move E.

The Midpoint Algorithm: Initialization

The initial value of F is:

$$\begin{aligned} F_0 &= 2f(A_x+1, A_y+1/2) \\ &= 2f(A_x+1, A_y+1/2) \\ &= 2a(A_x+1)+2b(A_y+1/2)+2c \\ &= 2f(\mathbf{A})+2a+b \end{aligned}$$

Since a and b's are integers, so is F. Multiplying $f(\mathbf{M})$ by 2 has allowed us to eliminate a division-by-2 which would have required us to use floating point. Clever, eh?

The Midpoint Algorithm: Optimization

Recall the algorithm:

If $F > 0$, move NE. Otherwise, move E.

Optimization: Don't recalculate F from scratch. Two cases:

1. If we're going to move NE ($F > 0$):

$$\begin{aligned} F' &= 2f(x_p + 2, y_p + 3/2) \\ &= 2a(x_p + 2) + 2b(y_p + 3/2) + 2c \\ &= F + 2a + 2b \end{aligned}$$

2. If we're going to move E ($F <= 0$):

$$\begin{aligned} F' &= 2f(x_p + 2, y_p + 1/2) \\ &= 2a(x_p + 2) + 2b(y_p + 1/2) + 2c \\ &= F + 2a \end{aligned}$$

So on each step, we only have to increment F, x, and sometimes y.

The Midpoint Algorithm: Implementation

```
void drawLine(int Ax, int Ay, int Bx, int By)
{ // works in first octant (0 to 45 deg) only
    int y = Ay, W = Bx - Ax, H = By - Ay, x;
    int F = 2 * H - W; // current error term

    for (x = Ax; x <= Bx; x++) {
        setPixel(x, y);
        // set up for next pixel
        if (F < 0) // move east
            F += 2 * H;
        else {     // move northeast
            y++;
            F += 2 * (H - W);
        }
    }
}
```

Elegant, yes?

The Midpoint Algorithm: Analysis

How does this differ from Bresenham's original?

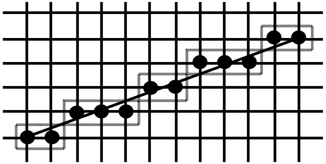
- same pixels are "lit"
- about the same computational efficiency (in integer form)
- *but* more easily generalizes to any implicit function:
  - circles
  - ellipses
  - parabolas
  - ...

It still only works in the first octant. (This is easy to fix.)

Wanna See my Fast Draw?

In his book, *Zen of Graphics Programming*, Michael Abrash goes *way* beyond Bresenham and midpoint algorithms:

- rewrite in PC assembler
- use registers wherever possible (and bizarrely)
- uses "run-length slice" algorithm:



With hardware support, nowadays most of Abrash's work is moot, but his dedication to the clever use of available hardware is still praiseworthy. Go ye and do likewise!

So what's the best line drawing algorithm?

Criteria:

- target hardware
- performance
- reliability (including coding)
- styles required:
  - dashing
  - thickness

Thick Lines

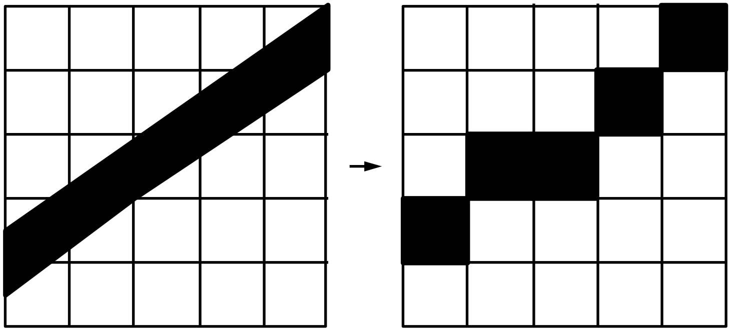
Several ways to do it:

- replicate pixels
  - need to account for varying thickness according to angle
  - sharp corners a problem
- move a "pen" polygon
  - polygon should approximate circle
  - may be expensive
- draw two primitives and fill space in between
- replace all thick line primitives with filled polygons

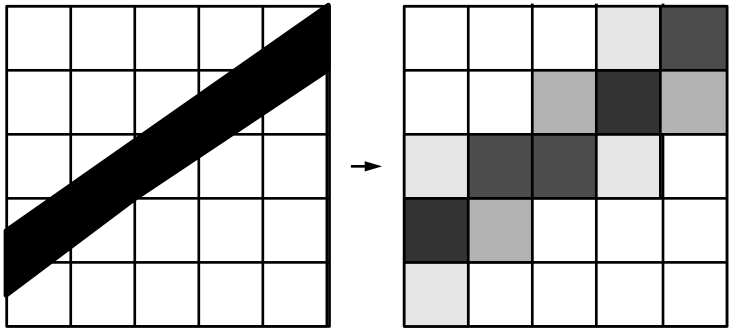
Aliasing and Antialiasing

- otherwise known as "the jaggies"
- toggle antialiasing in *pa01_circles* or *pa02_wire_track* output and view it with *xmag(1)*
- aliasing:
  - high frequency (in space or time) signal sampled at insufficient rate appears (under an "alias") as a lower-frequency signal
  - usual example: "wagon wheel" (originally seen in westerns) effect `https://www.youtube.com/watch?v=jHS9JGkEOmA`
- jaggies are technically not aliasing, but *reconstruction anisotropy*
  - everybody calls the jaggies "aliasing", anyway
  - getting rid if the jaggies is, therefore, "antialiasing"

Without Antialiasing



With Antialiasing



Antialiasing Algorithms

In the past, there were a *lot* of approaches to antialiasing. Algorithms looked at:

- line thickness
- distance of line from pixel center (How is this measured?)
- pixel response function (e.g. unweighted area sampling)
- What is the "shape" of a pixel?
  - (Turns out to be a very zen question.)

Now, it's supported pretty well in hardware.