Line Drawing

CS150 Fall 2010

Objective

- Given endpoints (x_0, y_0), (x_1, y_1) we want to shade the pixels that come closest to satisfying:
  - $y - y_0 = (y_1 - y_0)/(x_1 - x_0) * (x - x_0)$
- This is called rasterization

Towards a Line Drawing Algorithm

- For now, we'll deal with the simple case where x0 < x1, y0 < y1, 0 <= slope <= 1. IE a (< 45 degree) line going up and to the right.
- **Idea**: For each x, compute y according to our line equation, and round to the nearest integer.
- This works, but it's slow.

Towards a Line Drawing Algorithm

- **Slightly better Idea**: Track error (deviation from the ideal line) and a "current" y rather than recomputing y each time.
  - Error starts out at 0
  - For every step in the x direction, add the slope to the error
  - When do we know we need to change the y value? When the error is greater than half a pixel.
- The error-tracking version of this algorithm is typically called **Bresenham's Algorithm**.

Bresenham's Line Drawing Algorithm

```
1  void draw(int x0, int y0, int x1, int y1) {
2    float slope = ((float) y1 - y0) / ((float) x1 - x0);
3    float err = 0.0;
4    int y = y0;
5    for (int x = x0; x <= x1; x++) {
6      draw(x,y);
7      err += slope;
8      if (err >= 0.5) {
9        y++;
10       err -= 1.0;
11     }
12   }
13 }
```

Bresenham's Line Drawing Algorithm

- Pros:
  - Simple
  - Gives us the correct (minimal error) answer.
  - Seems fast
  - No "seek" operations on x or y.
- Cons (right now):
  - Only handles lines in the first quadrant that aren't "steep".
  - Uses floating point arithmetic!
    - Slow, expensive, accumulates error.
  - No anti-aliasing.. but this is outside of the scope of our problem.

Can we do better?

- We'd like to be able to draw any kind of line.
- Right now we are restricted to lines with with slope < 1 that increase to the right.
  -

Coordinate Ordering

- Before we jump into the main loop, sort our points by x-coordinate.
- Now we have satisfied x0 < x1.
- To deal with the case where y0 > y1 (the line is sloping down), we just need to decrease rather than increase y when our error gets too large.

Can we do better?

```
1  void draw(int x0, int y0, int x1, int y1) {
2    if (x0 > x1) {
3      swap(x0, x1);
4      swap(y0, y1);
5    }
6
7    float slope = ((float) y1 - y0) / ((float) x1 - x0);
8    float dErr = abs(slope);
9
10   int yStep = y0 > y1 ? -1 : 1;
11
12   float err = 0.0;
13   int y = y0;
14
15   for (int x = x0; x <= x1; x++) {
16     draw(x,y);
17     err += dErr;
18     if (err >= 0.5) {
19       y += yStep;
20       err -= 1.0;
21     }
22   }
23 }
```

Steep Lines

- Right now, if the line is steeper than 45 degrees, we get gaps because we only draw one pixel for every x coordinate.
- This one is an easy fix: If we detect we are drawing a "steep" line, then we just switch all of the x's and y's.

Steep Lines

```
1  void draw(int x0, int y0, int x1, int y1) {
2    char steep = (ABS(y1 - y0) > abs(x1 - x0)) ? 1 : 0;
3    if (steep) {
4      SWAP(x0, y0);
5      SWAP(x1, y1);
6    }
7    if (x0 > x1) {
8      SWAP(x0, x1);
9      SWAP(y0, y1);
10   }
11
12   float slope = ((float) y1 - y0) / ((float) x1 - x0);
13   float dErr = ABS(slope);
14
15   int yStep = y0 > y1 ? -1 : 1;
16
17   float err = 0.0;
18   int y = y0;
19
20   for (int x = x0; x <= x1; x++) {
21     if (steep)
22       draw(y,x);
23     else
24       draw(x,y);
25     err += dErr;
26     if (err >= 0.5) {
27       y += yStep;
28       err -= 1.0;
29     }
30   }
31 }
```

Floating Point

- Doing floating point arithmetic is *really hard*
  - It is even harder to do it correctly
- It would be best if we could use integers everywhere.
- With a little bit of cleverness we can eliminate all of the floating points.
- Let's start by multiplying all of our floating point values by (x1 - x0)

Floating Point

```
1  void draw(int x0, int y0, int x1, int y1) {
2    char steep = (ABS(y1 - y0) > abs(x1 - x0)) ? 1 : 0;
3    if (steep) {
4      SWAP(x0, y0);
5      SWAP(x1, y1);
6    }
7    if (x0 > x1) {
8      SWAP(x0, x1);
9      SWAP(y0, y1);
10   }
11
12   int dErr = ABS(y1 - y0);
13   int yStep = y0 > y1 ? -1 : 1;
14   int dX = x1 - x0;
15
16   float err = 0.0;
17   int y = y0;
18
19   for (int x = x0; x <= x1; x++) {
20     if (steep)
21       draw(y,x);
22     else
23       draw(x,y);
24     err += dErr;
25     if (err >= 0.5 * dX) {
26       y += yStep;
27       err -= dX;
28     }
29   }
30 }
```

Floating Point

- This is really close. The only problem remaining is that pesky multiply by 0.5.
- We can closely approximate a multiply by 0.5 with a right shift by 1.
- However, if we add error on every iteration of the algorithm we will probably get a wrong answer. Instead, we can choose to only pay this penalty once, when we initialize **err**.
- Let me show you what I mean:

Floating Point

```
1  void draw(int x0, int y0, int x1, int y1) {
2    char steep = (ABS(y1 - y0) > abs(x1 - x0)) ? 1 : 0;
3    if (steep) {
4      SWAP(x0, y0);
5      SWAP(x1, y1);
6    }
7    if (x0 > x1) {
8      SWAP(x0, x1);
9      SWAP(y0, y1);
10   }
11
12   int dErr = ABS(y1 - y0);
13   int yStep = y0 > y1 ? -1 : 1;
14   int dX = x1 - x0;
15
16   int err = -(dX >> 1); // divide by 2
17   int y = y0;
18
19   for (int x = x0; x <= x1; x++) {
20     if (steep)
21       draw(y,x);
22     else
23       draw(x,y);
24     err += dErr;
25     if (err > 0) {
26       y += yStep;
27       err -= dX;
28     }
29   }
30 }
```

Floating Point

- Finally, the canonical optimized version of this algorithm makes one more tiny optimization by inverting □**err** everywhere.

Floating Point

```
1  void draw(int x0, int y0, int x1, int y1) {
2    char steep = (ABS(y1 - y0) > abs(x1 - x0)) ? 1 : 0;
3    if (steep) {
4      SWAP(x0, y0);
5      SWAP(x1, y1);
6    }
7    if (x0 > x1) {
8      SWAP(x0, x1);
9      SWAP(y0, y1);
10   }
11
12   int dErr = ABS(y1 - y0);
13   int yStep = y0 > y1 ? -1 : 1;
14   int dX = x1 - x0;
15
16   int err = dX >> 1; // divide by 2
17   int y = y0;
18
19   for (int x = x0; x <= x1; x++) {
20     if (steep)
21       draw(y,x);
22     else
23       draw(x,y);
24     err -= dErr;
25     if (err < 0) {
26       y += yStep;
27       err += dX;
28     }
29   }
30 }
```

More than just lines

- A concept that comes up frequently in computer graphics algorithm is that of a Digital Differential Analyzer. A DDA is a device or algorithm that linearly interpolates one or more variables across some range.
- The algorithm we just wrote implements a DDA that interpolates one value (y) over a certain range (the x values).
- It is relatively straightforward to interpolate more values over the same range when we look at Bresenham's in this way. Some interesting additional values to interpolate might be red, green, blue, and even alpha channels for drawing gradients.
- With a little bit of cleverness DDAs can be used to draw triangles and even arbitrary polygons. Ask the TAs if you're curious.

Hardware Implementation Notes

- See the lab guide for suggestions for interfacing with the CPU.
- Make sure you can stall your line drawing engine!
- For your design doc, considering dividing your design into a data-path and some sort of control (just like a processor!).
- Your control logic will probably look like a simple FSM.

Acknowledgements

- This lecture partially adapted from CS150 Spring 2010 Lecture 16.
  - Check out these slides for examples of running the algorithm.