Now Playing:

The Loneliness of a Middle Distance Runner
Belle & Sebastian
from *Sing Jonathan David EP*
Released June 18, 2001

Rasterization: Line Drawing

Rick Skarbez, Instructor
COMP 575
October 4, 2007
Some slides and images courtesy Jeremy Wendt (2005) and Eric Bennett (2006)

Announcements

- Assignment 2 is out
  - Due next Tuesday by the end of class

Last Time

- Discussed clipping
  - Points
  - Lines
  - Polygons
- Introduced Assignment 2

Today

- Introduce rasterization
- Talk about some line drawing algorithms
- Discuss line anti-aliasing

Rendering Pipeline

- OpenGL rendering works like an assembly line
  - Each stage of the pipeline performs a distinct function on the data flowing by
  - Each stage is applied to every vertex to determine its contribution to output pixels

Geometry (Vertices) → Vertex Processing → Rasterizer → Fragment Processing → Pixels

Rasterization

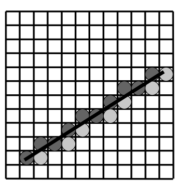
- In the rasterization step, geometry in device coordinates is converted into fragments in screen coordinates
- After this step, there are no longer any “polygons”

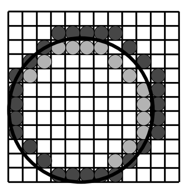
Rasterization

- All geometry that makes it to rasterization is within the normalized viewing region
- All the rasterizer cares about is (x, y)
  - z is only used for z-buffering later on
- Need to convert continuous (floating point) geometry to discrete (integer) pixels

Mapping Continuous to Discrete

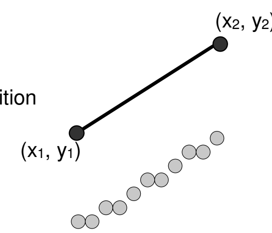
- Note that there isn’t only one “right” way to do this





Line Drawing

- A classic part of the computer graphics curriculum
- Input:
  - Line segment definition
    - (x_1, y_1), (x_2, y_2)
- Output:
  - List of pixels



Line Representation

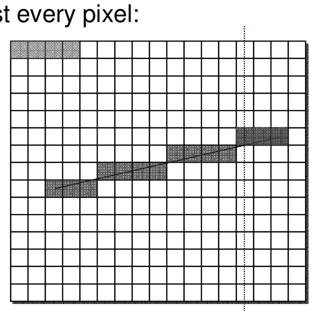
- We usually think about lines in slope-intercept form:
  - $y = mx + b$
- There are some problems with this

Problems with Slope-Intercept Form

- We have the wrong variables
  - (x_0, y_0), (x_1, y_1), not (m, b)
- m is the slope of the line
  - $m = (y_1 - y_0) / (x_1 - x_0)$
  - What happens if the line is vertical?
    - $m = \infty$

Line Algorithm #1: Brute Force

- Test every pixel:



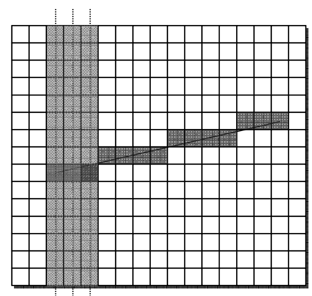
Line Algorithm #1: Brute Force

```
function DrawLine(LineColor, x0, y0, x1, y1)

float m = (y1 - y0) / (x1 - x0);
float b = -x0 * m + y0;

ForEach y = 0:ImageHeight-1
{
ForEach x = 0:ImageWidth-1
{
     if (y == round( m*x + b ))
       if((x>=x0) && (x<=x1))
         Output[x,y] = LineColor;
}
}
```

Brute Force: Pros and Cons

- Pros:
  - Very simple to implement
- Cons:
  - Very slow
    - Need to traverse every screen pixel for every line
  - Can't handle vertical lines properly
  - Requires floating point ops, including round()

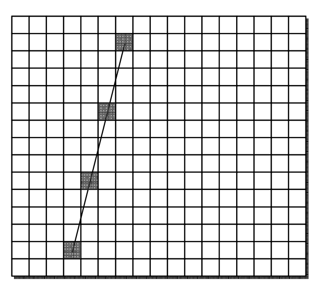
Line Algorithm #2: Line Traversal



Line Algorithm #2: Line Traversal

```
function DrawLine(LineColor, x0, y0, x1, y1)

if(x0>x1) flip ((x0,y0), (x1, y1));

float m = (y1 - y0) / (x1 - x0);
float b = -x0 * m + y0;

ForEach x = x0:x1
{
y = round( m*x + b );
Output[x,y] = LineColor;
}
```

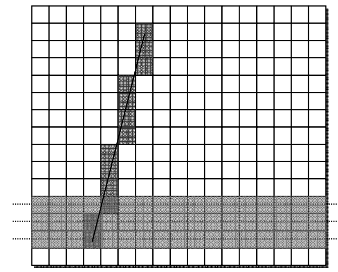
Line Traversal Problem



Line Algorithm #2a: Line Traversal++

- Check m right away
- If |m| > 1, need to step in y instead of x
- Even better, check whether |x1 - x0| or |y1 - y0| is bigger
  - Fixes the vertical line problem, too

Line Algorithm #2a: Line Traversal++



Line Traversal++: Pros and Cons

- Pros:
  - Still quite simple to implement
  - Much better performance
    - O(N) vs. O(N^2)
  - Can be pipelined
- Cons:
  - Still needs floating point round

Line Algorithm #3: Incremental Line Traversal

```
function DrawLine(LineColor, x0, y0, x1, y1)


if(x0>x1) flip ((x0,y0), (x1, y1));

float m = (y1 – y0) / (x1 – x0);
float b = -x0 * m + y0;

float y = y0;

for (x=x0; x<=x1; x++)
{
Output[x, round(y)] = LineColor;
y = y + m;
}
```

Note that we no longer need 'b'

Incremental Line Traversal: Pros and Cons

- Pros:
  - Moderate performance
    - Only max($|x_1 - x_0|$, $|y_1 - y_0|$) iterations
  - Handles vertical lines
- Cons:
  - Still needs floating point round
  - No longer able to easily pipeline

Line Algorithm #4: Y-Crossing Detection

```
if(x0>x1) flip ((x0,y0), (x1, y1));
float m = (y1 – y0) / (x1 – x0);
int y = y0;
float error = 0.0;

if (y1 > y0)
yStep = 1;
else
yStep = –1;

for (x=x0; x<=x1; x++)
{
Output[x,y] = LineColor;
error = error + fabs(m);
if(error > .5)
{
	y = y + yStep;
	error = error – 1.0;
}
}
```

Y-Crossing Detection: Pros and Cons

- Pros:
  - Pretty good performance
    - 2 fp adds, 1 fp sub, 1 compare in loop
    - No more rounding
- Cons:
  - Still floating point
  - Hard to pipeline
  - Needs special case for |m| > 1

The Need for Speed

- How can we do even better?
  - Need to get rid of floating point ops

The Need for Speed

```
float m = (y1 - y0) / (x1 - x0);

...

for (x=x0; x<=x1; x++)
{
Output[x,y] = LineColor;
error = error + fabs(m);
if(error > .5)
{
      y = y + yStep;
      error = error - 1.0;
}
}
```

- Still have floating point ‘m’
- Still using floating point error
  - Comparing to 0.5

Changes from #4

	Before	After	Finally...
m	$\frac{y_1 - y_0}{x_1 - x_0}$	$y_1 - y_0$	$2(y_1 - y_0)$
Test Value	.5	$.5(x_1 - x_0)$	$x_1 - x_0$
Subtracted Value	1.0	$x_1 - x_0$	$2(x_1 - x_0)$

Line Algorithm #5: Bresenham’s Algorithm

```
if(x0>x1) flip ((x0,y0), (x1, y1));
int y = y0;
int error = 0;

if (y1 > y0)
yStep = 1;
else
yStep = -1;

for (x=x0; x<=x1; x++)
{
Output[x,y] = LineColor;
error = error + abs(2 * (y1 - y0));
if(error > (x1 - x0))
{
      y = y + yStep;
      error = error - (2 * (x1 - x0));
}
}
```

Line Algorithm #5: Bresenham’s Algorithm

- So how do we do it?
  - Algorithm #4 stored the offset from the pixel center
  - Bresenham’s only stores a decision parameter:
    If > 0, go up, else, go across

Bresenham's Algorithm: Pros and Cons

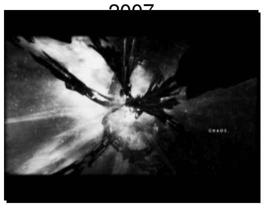
- Pros:
  - Great performance
    - Only integer arithmetic
- Cons:
  - Cannot be easily pipelined
  - Still needs special case for |m| > 1

Line Drawing Summary

- Talked about several line drawing algorithms
  - All produce the same output
  - Bresenham's algorithm is fastest in most cases
- I would suggest knowing how these work:
  - #2a: Line Traversal
  - #5: Bresenham's
- There are more that I did not discuss

Movie Break: Chaos Theory

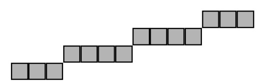
Conspiracy Group, Assembly 2006 / SIGGRAPH 2007



Available online:
http://xplsv.tv/movie.php?id=1942

How Do They Look?

- So now we know how to draw lines
- But they don't look very good:



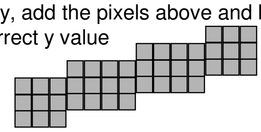
- Why not?
- Aliasing

Better Looking Lines

- There are ways to make lines look better:
  - Hacky: Just draw wider lines
  - Better: Anti-aliasing
- NOTE: This isn't really part of the rasterizer
  - Just a good place to talk about it

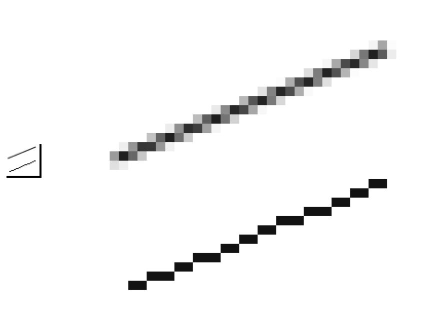
Quick Hack: Increase Line Width

- One quick fix may be to add nearby pixels:
  - Say, add the pixels above and below the correct y value



- Makes lines look a little bit better
- Does not increase computational complexity

Antialiasing



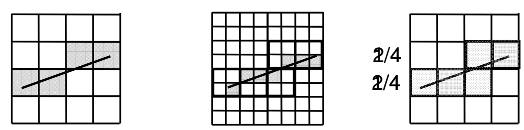
Antialiasing #1: Supersampling

- One technique that can be used for antialiasing is *supersampling*
  - Drawing at a higher resolution than will actually be used for the final output

Antialiasing #1: Supersampling

- Technique:
  1. Create an image 2x (or 4x, or 8x) bigger than the real image
  2. Scale the line endpoints accordingly
  3. Draw the line as before
     - No change to line drawing algorithm
  4. Average each 2x2 (or 4x4, or 8x8) block into a single pixel

Antialiasing #1: Supersampling



No antialiasing | 2x2 Supersampled | Downsampled to original size

Supersampling

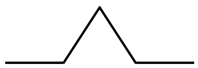
- So why is this a good idea?
  - Processing at a higher resolution produces more accurate data
    - Less *aliasing*
- However, it produces high frequency data that cannot be represented at the lower resolution
  - Need to *filter*
- Note: This usually makes lines appear fainter

Filtering Basics

- Filtering is, basically, removing some components from a signal
  - *i.e.* low frequencies (high-pass filter)
- We want to remove *high* frequencies
  - That is, we want a low-pass filter
- Since the high frequencies represent fine/sharp details, low-pass filtering is called *smoothing* or *blurring*

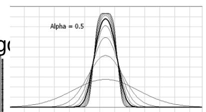
Low-Pass Filtering

- Want to smooth changes between neighboring pixels
- Many ways to do it
  - 2 Examples:
    - Tent: Fast, but not great



    - Gaussian: Slow, but very g

$$Gaussian(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left(\frac{(x-\mu)^2}{2\sigma^2}\right)}$$



Filtering

- Not going to go into any more details right now
- We'll talk about it more in the second half of the semester when we talk about real cameras
- For now, just accept that doing a better job filtering makes your antialiasing better

Resizing a high-resolution image

Incorrect

Gaussian (Correct)

Next Time

- Finish up anti-aliasing
  - Ratio method
- Continuing with rasterization
  - Shape and polygon drawing
- Assignment 2 due