

Elementary raster algorithms for fast rendering

Elementary Rendering

- Geometric Primitives
 - Line processing
 - Polygon processing
- Managing OpenGL State
- OpenGL Buffers

OpenGL Geometric Primitives

- All geometric primitives are specified by vertices

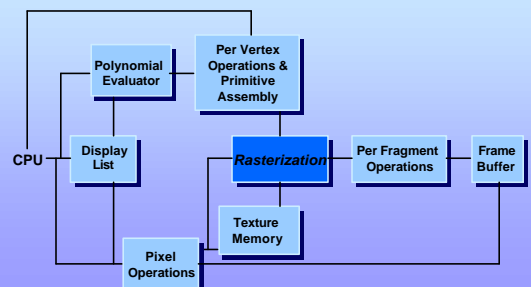


Design of Line Algorithms

Why Lines?

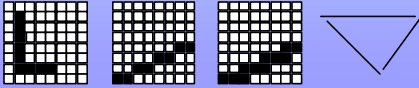
- Lines:
 - Most common 2D primitive - done 100s or 1000s of times each frame, even 3D wireframes are eventually 2D lines!
 - Lines are *compatible* with vector displays but nowadays most displays are raster displays. Any render stage before viz might need discretization.
 - Optimized algorithms contain numerous tricks/techniques that help in designing more advanced algorithms for line processing.

Line Algorithms in the OpenGL Architecture



Line Requirements

- Must compute integer coordinates of pixels which lie on or near a line or circle.
- Pixel level algorithms are invoked hundreds or thousands of times when an image is created or modified – must be fast!
- Lines must create visually satisfactory images.
 - Lines should appear straight
 - Lines should terminate accurately
 - Lines should have constant density
- Line algorithm should always be defined.



Basic Math Review

Point-slope Formula For a Line

Given two points (X_1, Y_1) , (X_2, Y_2)

Consider a third point on the line:

$$P = (X, Y)$$

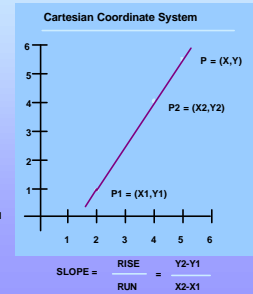
$$\text{Slope} = (Y_2 - Y_1)/(X_2 - X_1) \\ = (Y - Y_1)/(X - X_1)$$

Solving For Y

$$Y = [(Y_2 - Y_1)/(X_2 - X_1)] * (X - X_1) + Y_1$$

or, plug in the point $(0, b)$ to get the **Slope-intercept form**:

$$Y = mx + b$$



Other Helpful Formulas

- **Length of line segment between P_1 and P_2 :**

$$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- **Midpoint of a line segment between P_1 and P_2 :**

$$P_3 = ((X_1 + X_2)/2, (Y_1 + Y_2)/2)$$

- Two lines are **perpendicular** iff

- 1) $M_1 = -1/M_2$
- 2) Cosine of the angle between them is 0.

Using this information, what are some possible algorithms for line drawing?

Parametric Form

Given points $P_1 = (X_1, Y_1)$ and $P_2 = (X_2, Y_2)$

$$X = X_1 + t(X_2 - X_1)$$

$$Y = Y_1 + t(Y_2 - Y_1)$$

t is called the parameter. When

$$t = 0 \text{ we get } (X_1, Y_1)$$

$$t = 1 \text{ we get } (X_2, Y_2)$$

As $0 < t < 1$ we get all the other points on the line segment between (X_1, Y_1) and (X_2, Y_2) .

New algorithm ideas based on parametric form?

Simple DDA* Line Algorithm

```

void DDA(int X1,Y1,X2,Y2)
{
    int Length, I;
    float X,Y,Xinc,Yinc;

    Length = ABS(X2 - X1);
    if (ABS(Y2 - Y1) > Length)
        Length = ABS(Y2-Y1);
    Xinc = (X2 - X1)/Length;
    Yinc = (Y2 - Y1)/Length;

    X = X1;
    Y = Y1;
    while(X<X2){
        Plot(Round(X),Round(Y));
        X = X + Xinc;
        Y = Y + Yinc;
    }
}
    
```

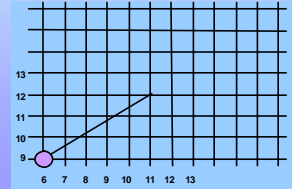
DDA creates good lines but it is too time consuming due to the round function and long operations on real values.

*DDA: Digital Differential Analyzer

DDA Example

Compute which pixels should be turned on to represent the line from (6,9) to (11,12).

Length = ?
Xinc = ?
Yinc = ?



DDA Example

Line from (6,9) to (11,12).

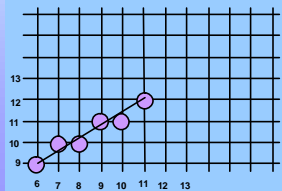
Length := Max of (ABS(11-6), ABS(12-9)) = 5

Xinc := 1

Yinc := 0.6

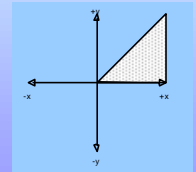
Values computed are:

(6,9), (7,9.6),
(8,10.2), (9,10.8),
(10,11.4), (11,12)



Fast Lines – Midpoint Method

- Simplifying assumptions: Assume we wish to draw a line between points (0,0) and (a,b) with slope m between 0 and 1 (i.e. line lies in first octant).

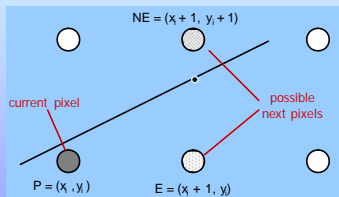


- The general formula for a line is $y = mx + B$ where m is the slope of the line and B is the y-intercept. From our assumptions $m = b/a$ and $B = 0$.

- $y = (b/a)x + 0 \rightarrow f(x,y) = bx - ay = 0$ is an equation for the line.

Fast Lines (cont.)

For lines in the first octant, given one pixel on the line, the next pixel is to the right (E) or to the right and up (NE).

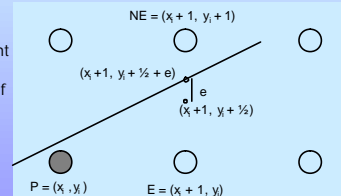


Having turned on pixel P at (x_i, y_i) , the next pixel is NE at (x_i+1, y_i+1) or E at (x_i+1, y_i) . Choose the pixel closer to the line $f(x, y) = bx - ay = 0$.

Fast Lines (cont.)

The midpoint between pixels E and NE is $(x_i + 1, y_i + \frac{1}{2})$.

Let e be the "upward" distance between the midpoint and where the line actually crosses between E and NE. If e is positive the line crosses above the midpoint and is closer to NE. If e is negative, the line crosses below the midpoint and is closer to E. To pick the correct point we only need to know the sign of e .



The Decision Variable

$$\begin{aligned}
 f(x_i+1, y_i + \frac{1}{2} + e) &= 0 \text{ (point on line)} \\
 &= b(x_i + 1) - a(y_i + \frac{1}{2} + e) \\
 &= b(x_i + 1) - a(y_i + \frac{1}{2}) - ae \\
 &= f(x_i + 1, y_i + \frac{1}{2}) - ae
 \end{aligned}
 \Rightarrow f(x_i + 1, y_i + \frac{1}{2}) = ae$$

Let $d_i = f(x_i + 1, y_i + \frac{1}{2}) = ae$; d_i is known as the **decision variable**.
 Since $a = 0$, d_i has the same sign as e .

Therefore, we only need to know the value of d_i to choose between pixels E and NE. If $d_i = 0$ choose NE, else choose E.

But, calculating d_i directly each time requires at least two adds, a subtract, and two multiplies -> too slow!

Decision Variable calculation

Algorithm:

Calculate d_0 directly, then for each $i \geq 0$:

if $d_i = 0$ Then

Choose NE = $(x_i + 1, y_i + 1)$ as next point

$$\begin{aligned}
 d_{i+1} &= f(x_{i+1} + 1, y_{i+1} + \frac{1}{2}) = f(x_i + 1 + 1, y_i + 1 + \frac{1}{2}) \\
 &= b(x_i + 1 + 1) - a(y_i + 1 + \frac{1}{2}) = f(x_i + 1, y_i + \frac{1}{2}) + b - a \\
 &= d_i + b - a
 \end{aligned}$$

else

Choose E = $(x_i + 1, y_i)$ as next point

$$\begin{aligned}
 d_{i+1} &= f(x_{i+1} + 1, y_{i+1} + \frac{1}{2}) = f(x_i + 1 + 1, y_i + \frac{1}{2}) \\
 &= b(x_i + 1 + 1) - a(y_i + \frac{1}{2}) = f(x_i + 1, y_i + \frac{1}{2}) + b \\
 &= d_i + b
 \end{aligned}$$

→ Knowing d_i , we need only add a **constant** term to find d_{i+1} !

Fast Line Algorithm

The initial value for the decision variable, d_0 , may be calculated directly from the formula at point (0,0).

$$d_0 = f(0 + 1, 0 + 1/2) = b(1) - a(1/2) = b - a/2$$

Therefore, the algorithm for a line from (0,0) to (a,b) in the first octant is:

```

x = 0;
y = 0;
d = b - a/2;
for(i = 0; i < a; i++) {
    Plot(x,y);
    if (d = 0) {
        x = x + 1;
        y = y + 1;
        d = d + b - a;
    }
    else {
        x = x + 1;
        d = d + b;
    }
}
    
```

Note that the only non-integer value is a/2. If we then multiply by 2 to get $d' = 2d$, we can do all integer arithmetic. The algorithm still works since we only care about the sign, not the value of d .

Bresenham's Line Algorithm

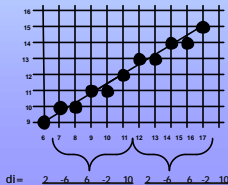
We can also generalize the algorithm to work for lines beginning at points other than (0,0) by giving x and y the proper initial values. This results in Bresenham's Line Algorithm.

```

{Bresenham for lines with slope between 0 and 1}
a = ABS(xend - xstart);
b = ABS(yend - ystart);
d = 2*b - a;
Incr1 = 2*(b-a);
Incr2 = 2*b;
if (xstart > xend) {
    x = xend;
    y = yend;
}
else {
    x = xstart;
    y = ystart;
}
for (i = 0; i < a; i++) {
    Plot(x,y);
    x = x + 1;
    if (d = 0) {
        y = y + 1;
        d = d + incr1;
    }
    else
        d = d + incr2;
}
    
```

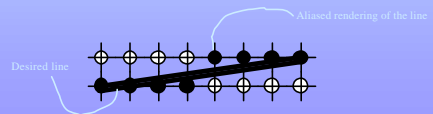
Optimizations

- Speed can be increased even more by detecting cycles in the decision variable. These cycles correspond to a repeated pattern of pixel choices.
- The pattern is saved and if a cycle is detected it is repeated without recalculating.



The aliasing problem

- Aliasing is caused by finite addressability of the display.
- Approximation of lines and circles with discrete points often gives a staircase appearance or "Jaggies".



Antialiasing - solutions

- Aliasing can be smoothed out by using higher addressability.
- If addressability is fixed but intensity is variable, use the intensity to control the address of a "virtual pixel". Two adjacent pixels can be used to give the impression of a point part way between them. The perceived location of the point is dependent upon the ratio of the intensities used at each. The impression of a pixel located halfway between two addressable points can be given by having two adjacent pixels at half intensity.
- An antialiased line has a series of virtual pixels each located at the proper address.



Aliasing / Antialiasing Examples

"Jaggies"

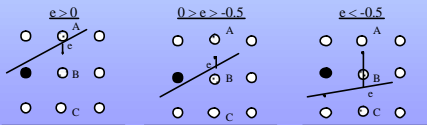


"Jaggies"



Antialiased Bresenham Lines

- Line drawing algorithms such as Bresenham's can easily be modified to implement virtual pixels. We use the distance ($e = d/a$) value to determine pixel intensities.
- Three possible cases which occur during the Bresenham algorithm:



$$e > 0$$

$$A = 0.5 + e$$

$$B = 1 - \text{abs}(e+0.5)$$

$$C = 0$$

$$0 > e > -0.5$$

$$A = 0.5 + e$$

$$B = 1 - \text{abs}(e+0.5)$$

$$C = 0$$

$$e < -0.5$$

$$A = 0$$

$$B = 1 - \text{abs}(e+0.5)$$

$$C = -0.5 - e$$

Line Rendering References

Bresenham, J.E., "Ambiguities In Incremental Line Rastering," IEEE Computer Graphics And Applications, Vol. 7, No. 5, May 1987.

Eckland, Eric, "Improved Techniques For Optimising Iterative Decision-Variable Algorithms, Drawing Anti-Aliased Lines Quickly And Creating Easy To Use Color Charts," CSC 462 Project Report, Department of Computer Science, North Carolina State University (Spring 1987).

Foley, J.D. and A. Van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley 1982.

Newman, W.M and R.F. Sproull, Principles Of Interactive Computer Graphics, McGraw-Hill, 1979.