CSC 4356
Interactive Computer Graphics
Lecture 7: Rasterization

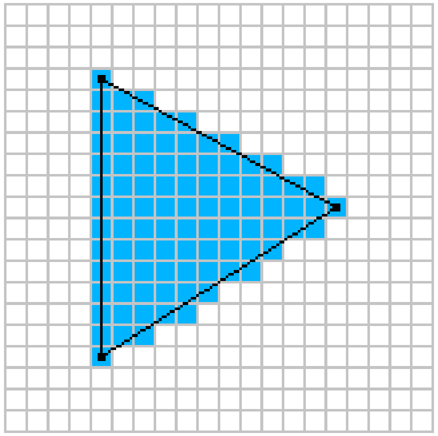
Jinwei Ye

http://www.csc.lsu.edu/~jye/CSC4356/

Tue & Thu: 10:30 - 11:50am
218 Tureaud Hall

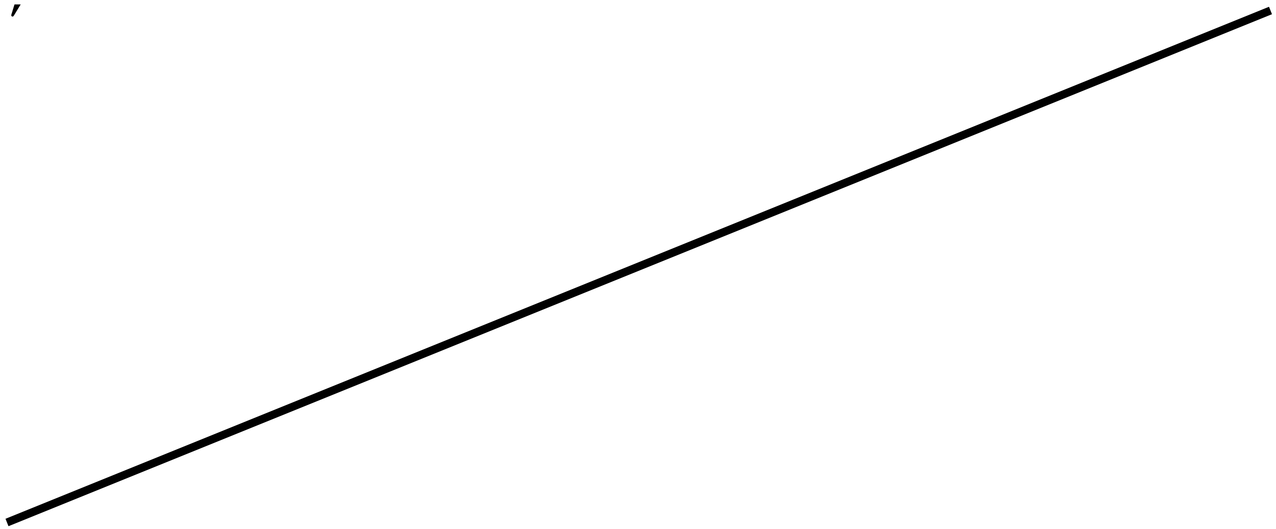
Rasterization

- Rasterization is the process that converts *continuous* primitives into *discontinuous* pixel representation
- Determine coverage
  - Which pixels belong to the primitive?
- Determine pixel parameters
  - Such as color, depth, etc.
  - How to interpolate?

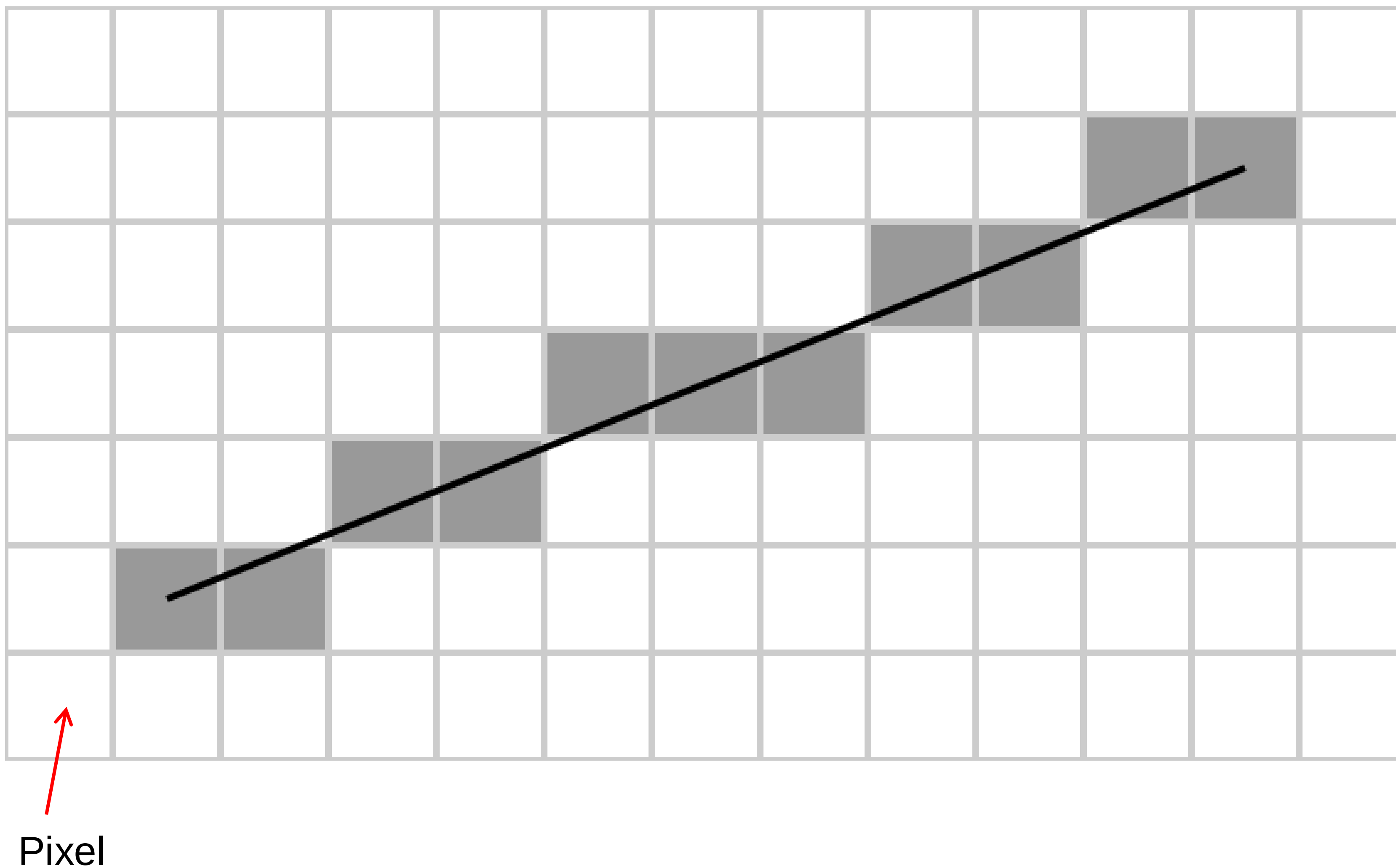


How does OpenGL draw a line?

```
glBegin(GL_LINES);
        glVertex3f (x1, y1, z1);
        glVertex3f (x2, y2, z2);
glEnd();
```

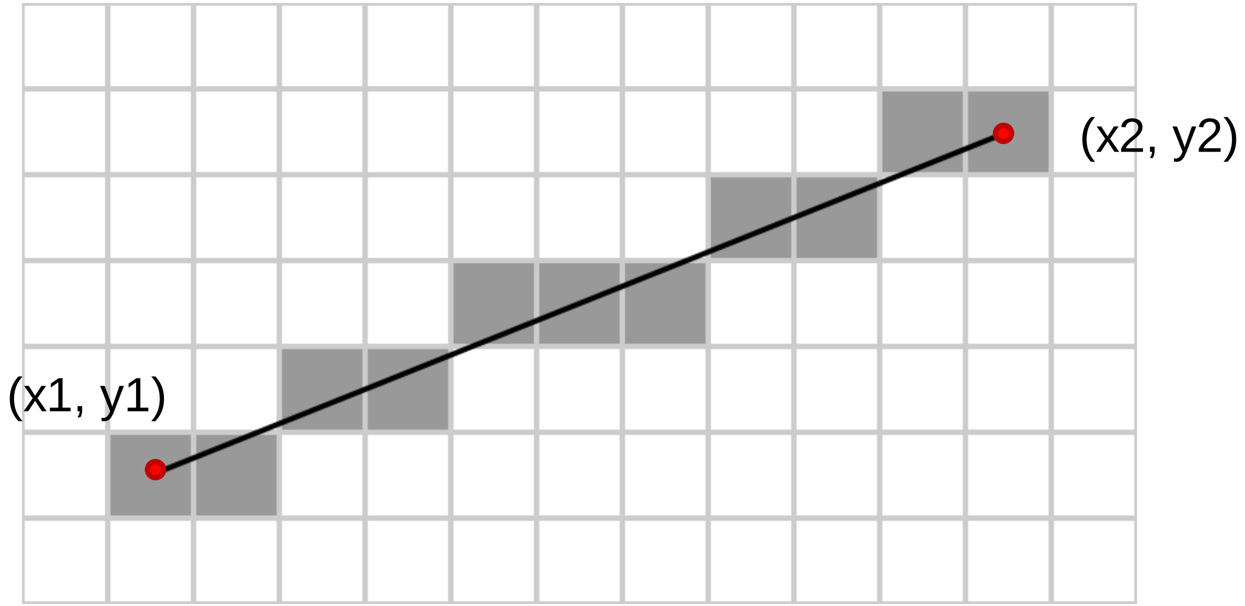


Everything is rasterized!



Line Rasterization Problem

- Given:
  - Two endpoints: integers (x1, y1) & (x2, y2)
- Identify:
  - Which pixels (x, y) to display for the line?



Requirements

- Transform **continuous** primitive into **discrete** samples
- Uniform thickness & brightness
- Continuous appearance
- No gaps
- Accuracy
- Speed

DDA Line Drawing

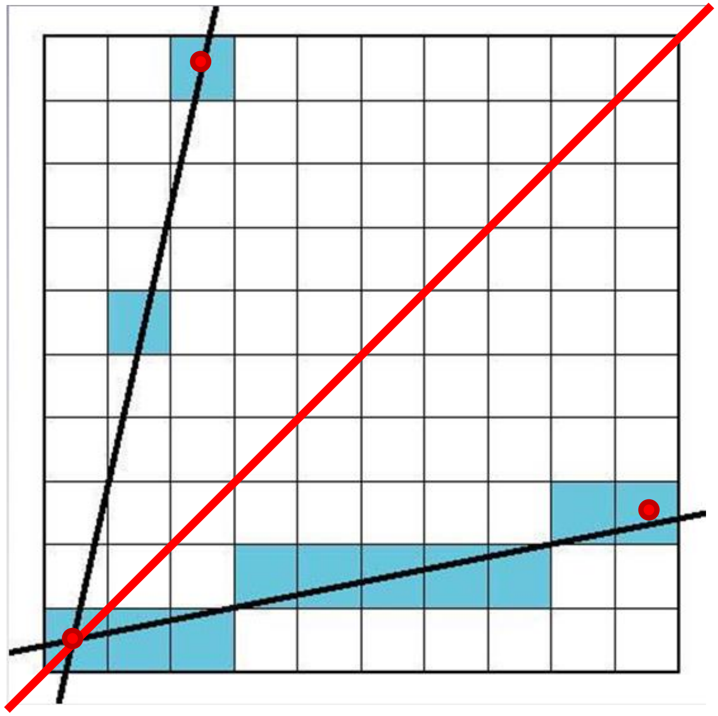
- DDA stands for Digital Differential Analyzer, the name of a class of old machines used for plotting functions

- Slope-intercept form of a line:

  $y = mx + b$

  slope: $m = dy/dx$

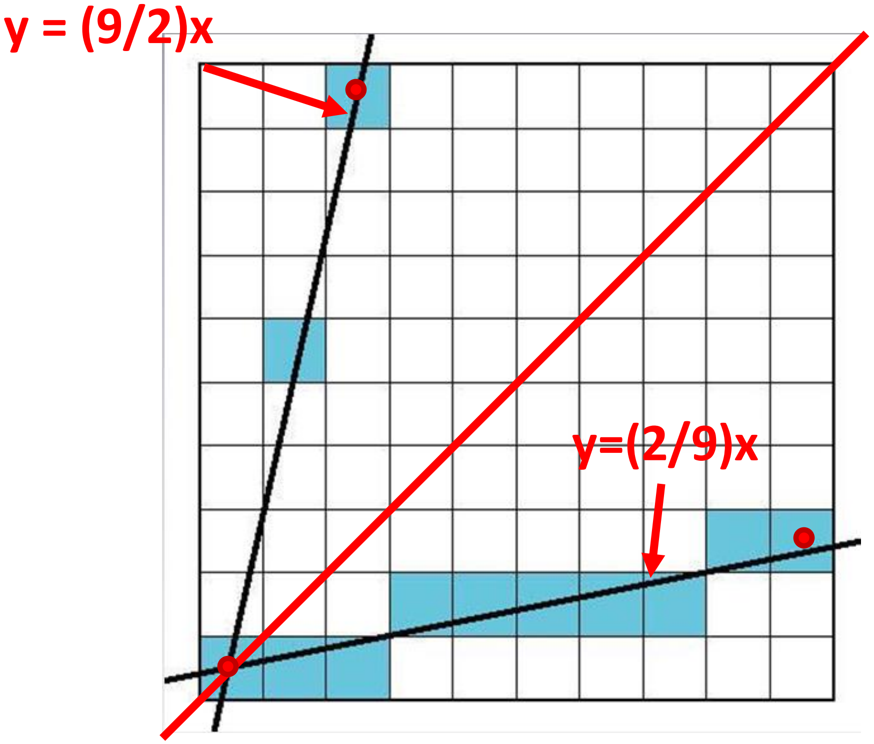
  intercept: b is where the line intersects the y-axis



DDA Line Drawing

- Basic idea: If we increment the x coordinate by one pixel at each step, the slope of the line tells us how much to increment y per step
  - i.e., dx = 1, dy = m

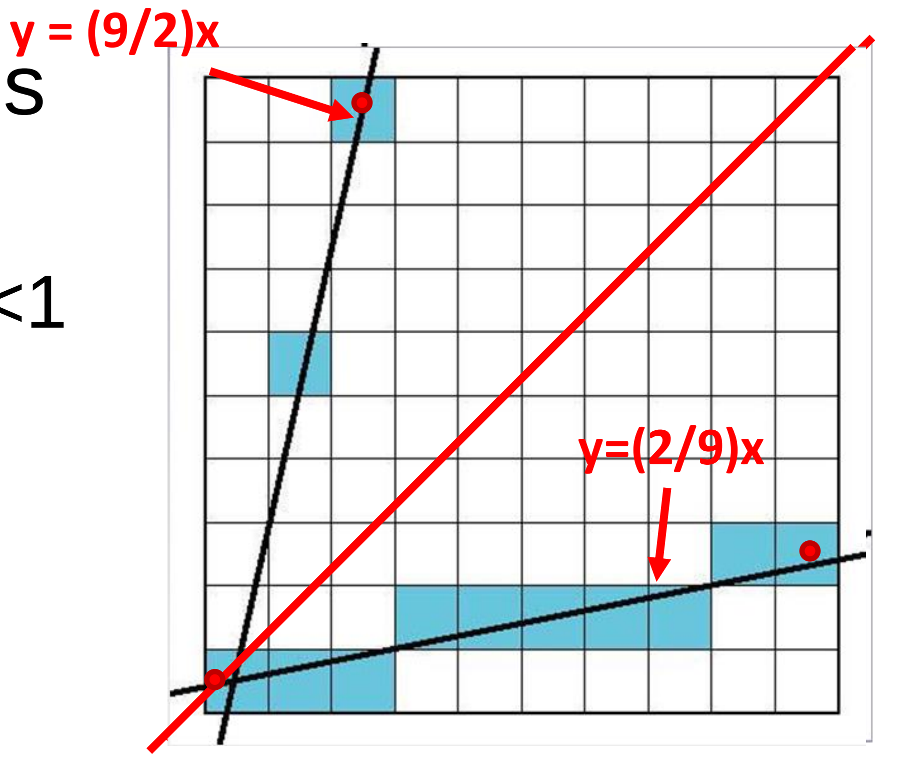
  (because m = dy/dx)



DDA Line Drawing

- This only works if m <= 1
  - otherwise there are gaps

- Solution: Reverse axes and step in y direction
  - Now dy = 1, dx = 1/m <1



DDA: Algorithm

- Given two endpoints (x1, y1), (x2, y2)
  - Integer coordinates: Round if endpoints were originally real-valued
  - Assume (x1, y1) is to the left of (x2, y2)
  - Swap if they aren't
- Then we can compute slope:

$$m = dy/dx = (y2 – y1) / (x2 – x1)$$

- Iteratively find the next pixel to display starting from (x1,y1)

DDA: Algorithm

- How to Iterate?
  - If |m| <= 1: Iterate integer x from x1 to x2, incrementing (or decrementing) by one pixel each step (x = x + 1)
    - Initialize real y = y1
    - At each step, y = y + m, and plot pixel (x, round(y))
  - Else |m| > 1: Iterate integer y from y1 to y2, incrementing (or decrementing) by one pixel each step (y = y + 1)
    - Initialize real x = x0
    - At each step, x = x + 1/m, and plot pixel (round(x), y)

Any Improvement?

- DDA is slow
  - Floating-point calculations, rounding is relatively expensive
- Idea: avoid rounding, do everything with integer arithmetic for speedup

Revisit Line Equation

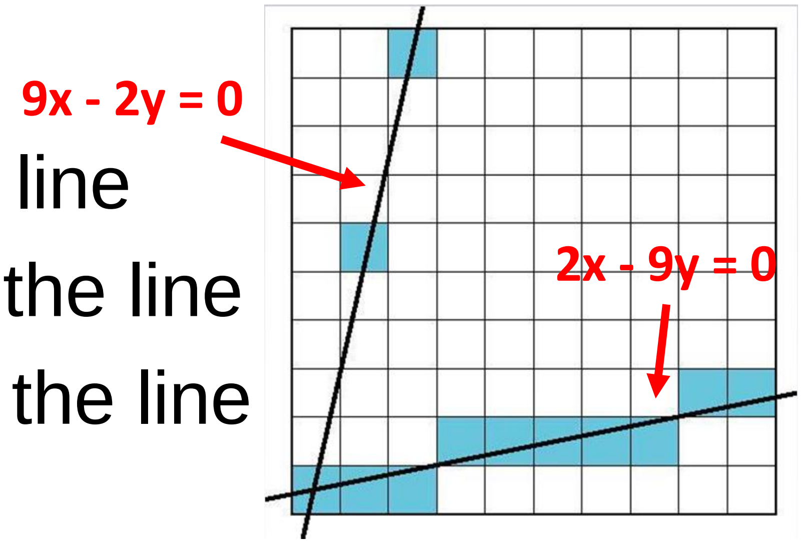
- Recall the slope-intercept form of a line is

$$y = (dy/dx)x + b$$

- Implicit form of a line is

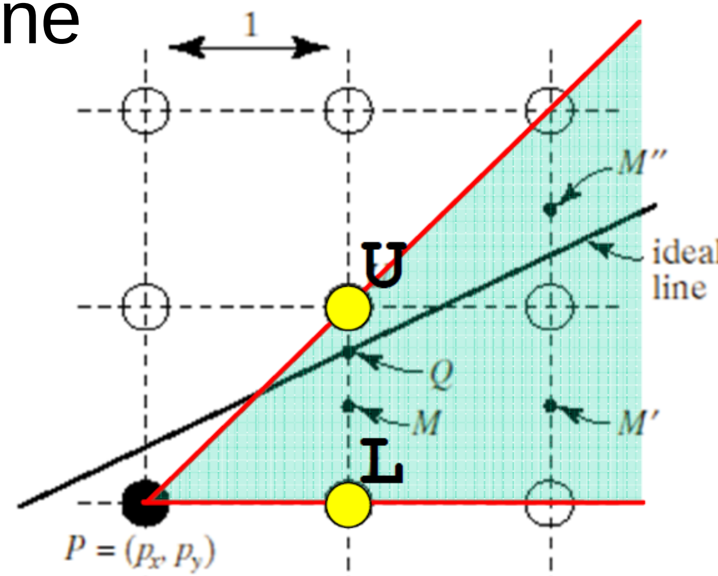
$$F(x, y) = dy \cdot x - dx \cdot y + dx \cdot b = 0$$

  - F = 0: point (x,y) is on the line
  - F > 0: point (x,y) is below the line
  - F < 0: point (x,y) is above the line



Decision Making

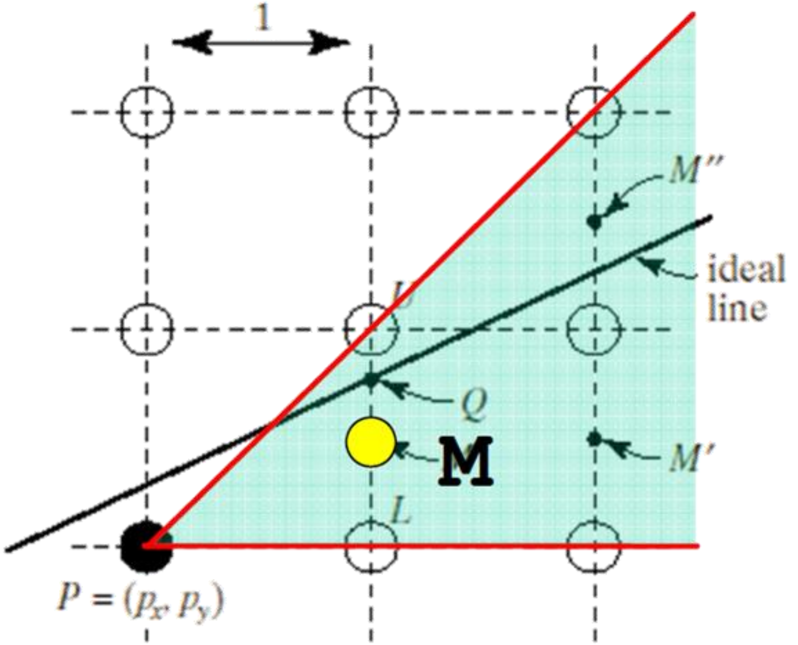
- Given our assumptions about the slope (|m|<1), after drawing (x, y) the only choice for the next pixel is between the upper pixel U = (x+1, y+1) and the lower one L = (x+1, y)
- We want to draw the pixel (U or L) that is closer to the "ideal" line



How to Make The Decision?

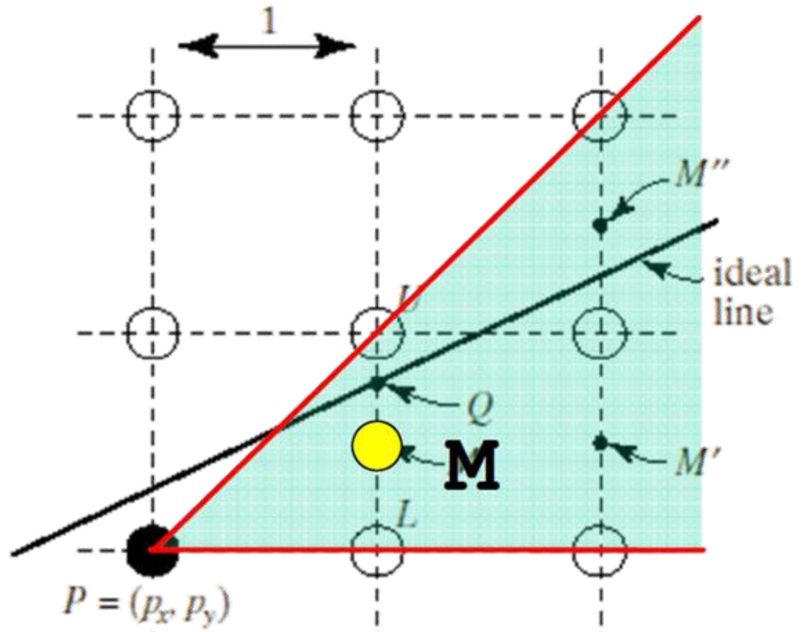
- After drawing (x, y), in order to choose the next pixel to draw we consider the midpoint M = (x+1, y+0.5)
  - If M is on the line, then U and L are equally distant from the ideal line
  - If M is below the line, then U is closer to the line
  - If M is above the line,

    then L is closer to the line



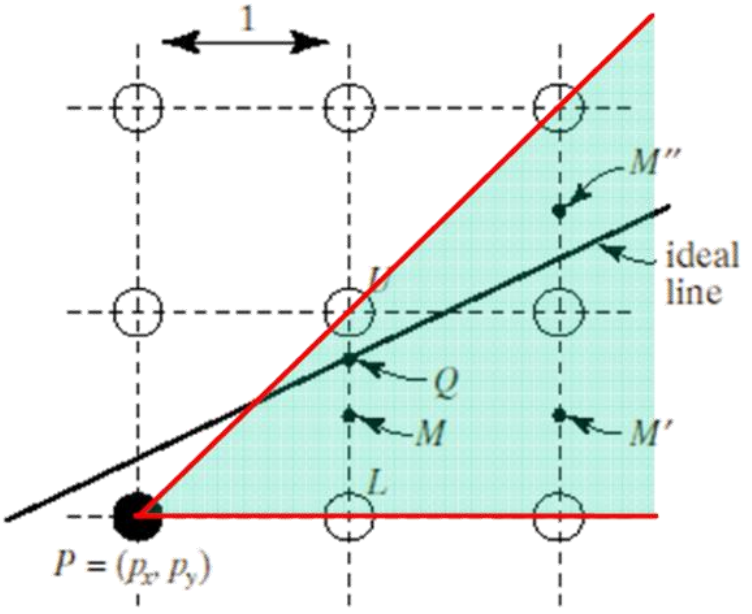
Decision Function

- Therefore F is a decision function to determine which pixel to draw:
  – If F(M) = F(x+1, y+0.5) > 0 (M below the line), pick U
  – If F(M) = F(x+1, y+0.5) <= 0 (M above or on line), pick L



Midpoint Algorithm (Bresenham's)

- Why is it faster?
  - F does not have to be fully evaluated everytime
- Suppose we do the full evaluation once and get F(x+1, y+0.5) for the first pixel to decide
- Then for the second pixel:
  - If we choose L, the next midpoint M' is (x+2, y+0.5)
  - If we choose U, the next midpoint M'' is (x+2, y+1.5)



Midpoint Algorithm (Bresenham's)

- Now let's plug the current midpoint M and the next midpoints M' and M'' into the decision function $F(x, y) = dy \cdot x - dx \cdot y + dx \cdot b = 0$

$$F_M = F(x + 1, y + 0.5) = dy(x + 1) - dx(y + 0.5) + dx \cdot b$$

$$F_{M'} = F(x + 2, y + 0.5) = dy(x + 2) - dx(y + 0.5) + dx \cdot b$$

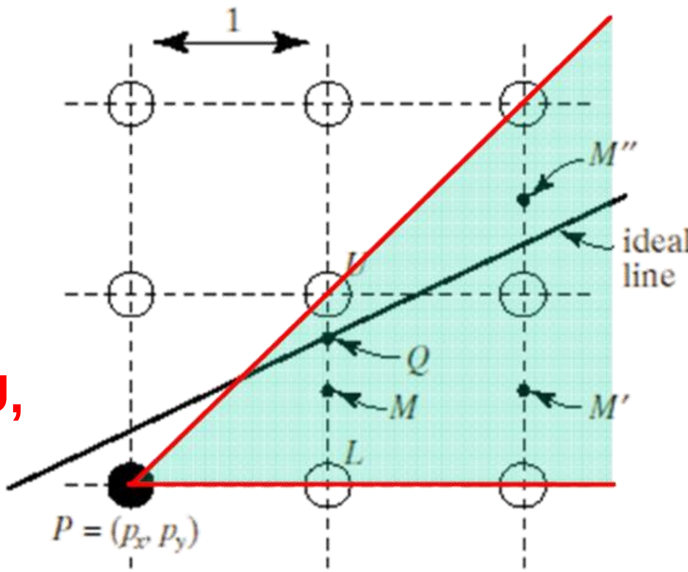
$$F_{M''} = F(x + 2, y + 1.5) = dy(x + 2) - dx(y + 1.5) + dx \cdot b$$

- So we have

$$F_{M'} - F_M = dy$$

$$F_{M''} - F_M = dy - dx$$

Depending on whether we choose L or U, we just have to add dy or dy – dx to the old value of F to get the new value



Midpoint Algorithm (Bresenham's)

- To initialize, we do a full calculation of F at the first midpoint next to the left line endpoint (x1,y1)

$$F(x1 + 1, y1 + 0.5)$$
$$= dy(x1 + 1) - dx(y1 + 0.5) + dx \cdot b$$
$$= F(x1, y1) + dy - 0.5\ dx$$

- F(x1, y1) = 0 because the end point is on the line, so

$$F = dy - 0.5\ dx$$

- Only the sign matters for the decision, so to make it an integer value we multiply by 2 to get $2F = 2\ dy - dx$
- To update, keep current values for x and y and evaluate F by its increment:
  - When L is chosen: F += 2dy and x++
  - When U is chosen: F += 2(dy - dx) and x++ , y++

Algorithm Summary

- Decision Function: $F = 2(dy \cdot x - dx \cdot y + dx \cdot b)$
- Initialization:
  - $dx = x\_end - x\_start$
  - $dy = y\_end - y\_start$
  - $F = 2dy - dx$
- Iterate:
  - if $F \le 0$, choose the lower point and $F=F+2dy$
  - if $F > 0$, choose the upper point and $F=F+2(dy-dx)$
- All integer operations!

Line Parameters

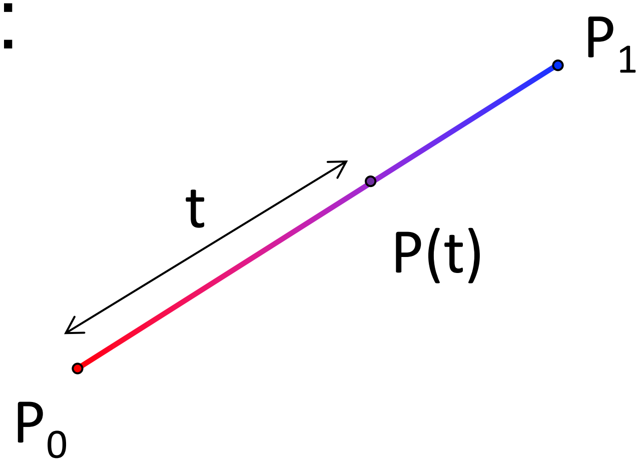
- Now we know how to determine the line pixels
- How to determine the line parameters, such as color?
  - If the two vertices have the same color, the line will be in uniform color.
  - If the two vertices have different colors, what would be the color for the line?

Blending by Linear Interpolation

- If the two vertices have different colors, the line color would be blended by linear interpolation
- Colors vary with distance fraction
- Parametric representation:

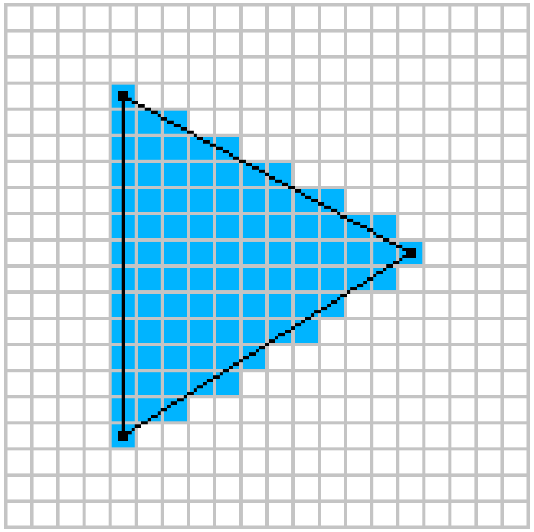
$$P(t) = P_0 + t(P_1 - P_0)$$
$$= P_0 + tP_1 - tP_0$$
$$= (1 - t)P_0 + tP_1$$

where $t \in [0,1]$



What About Triangle?

- Given three vertices of a triangle
- How to fill in the area?
- How to determine the pixel properties?
  - color, depth, etc.



Why Triangle?

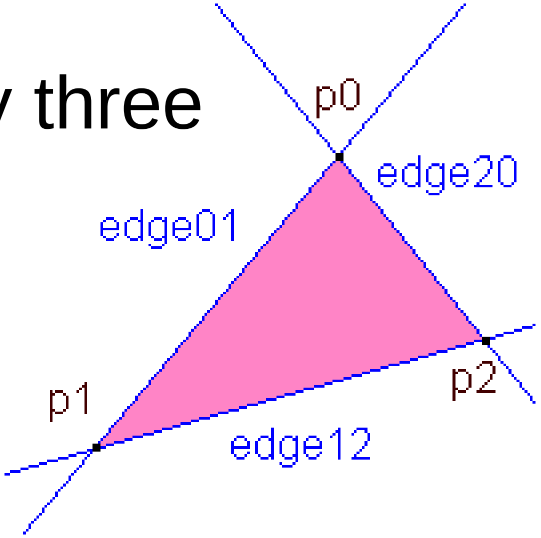
- Triangle is simple
  - A triangle can be defined by three vertices (x_0,y_0), (x_1,y_1), and (x_2,y_2)
  - A triangle can also be defined by three edges

    $A_1x + B_1y + C_1 = 0$

    $A_2x + B_2y + C_2 = 0$

    $A_3x + B_3y + C_3 = 0$

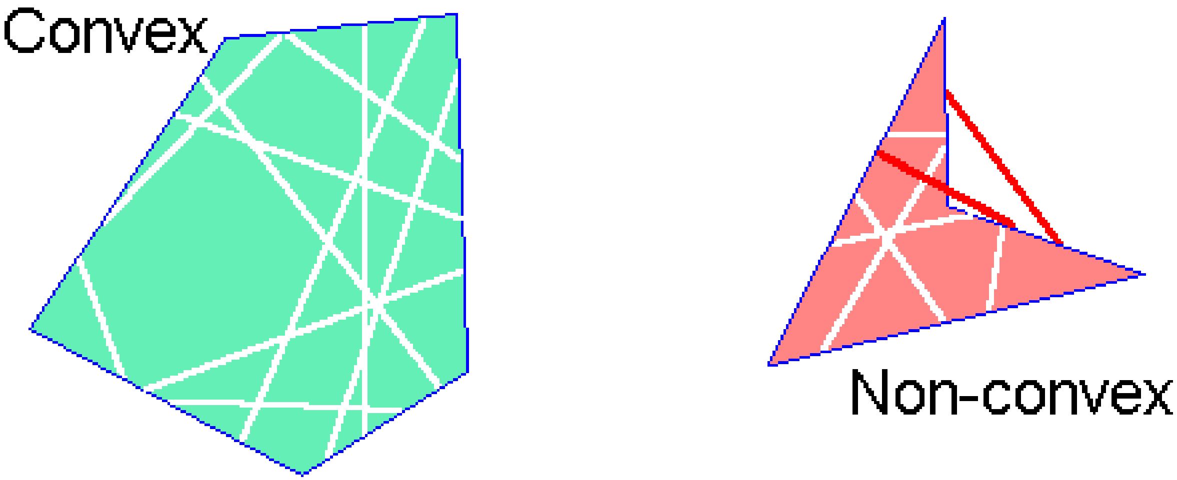
  - Why numbers of unknowns are different?



- As a result, scan converting triangles only involve linear equations

Why Triangle?

- What is convex?

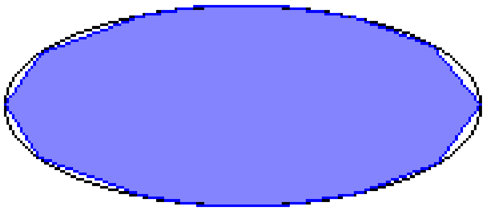


- Triangle is always convex
  - No matter how a triangle is oriented on the screen, a given scan line will contain only a single segment or span of the triangle

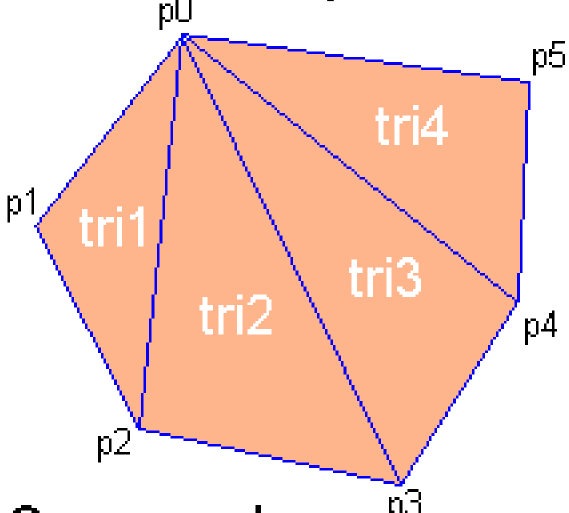
Why Triangle?

- Triangles can approximate any shape
  - Any 2D shape can be approximated by a polygon using locally linear approximation
  - Any 3D surfaces can be approximated by polygons
  - Polygons can be decomposed into triangles

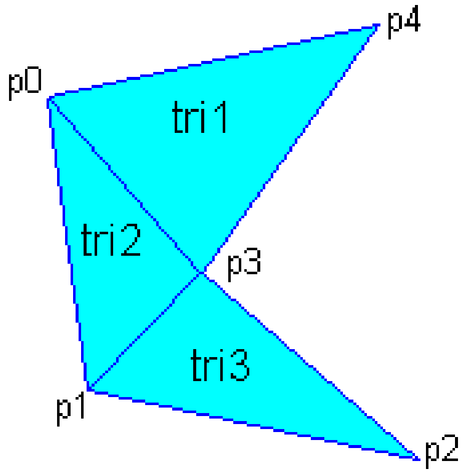
Polygonal Approximation



to a curve



Convex polygon



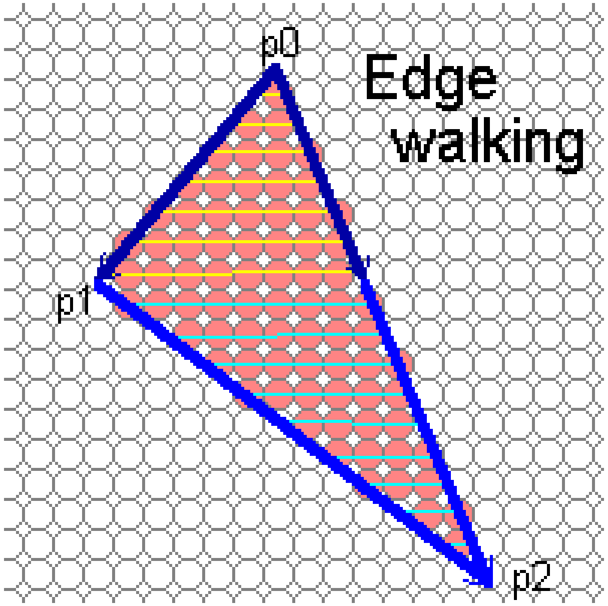
Non-convex

Triangle Rasterization

- Common triangle rasterization algorithms:
  - – Edge walking
  - – Edge equations
  - – Recursive subdivision (primitive or screen)

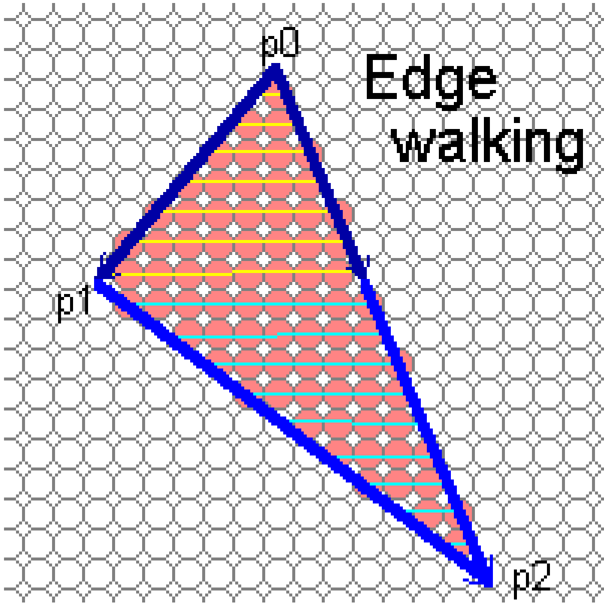
Edge Walking Algorithm

- Basic idea:
  - Draw edges vertically
  - Fill in horizontal spans for each scanline
  - Interpolate colors down edges
  - At each scanline, interpolate edge colors across span



Algorithm Overview

- Sort the vertices in both x and y
- Determine if the middle vertex, or *breakpoint* lies on the left or right side of the polygon
  - If the trianlge has an edge parallel to the scanline direction then there is no breakpoint
- Determines the left and right edge for each scanline (called *spans*)
- Walk down the left and right edges filling the pixels in-between until
  - A breakpoint is reached: switch edge
  - The bottom vertex is reached: exit



Notes on Edge Walking

- Advantage:
  - Generally very fast
- Disadvantages:
  - Loaded with special cases (left and right breakpoints, no breakpoints)
  - Difficult to get right
  - Requires computing fractional offsets when interpolating parameters across the triangle

Edge Equations

- An edge equation is simply the equation of the line containing that edge
  - Line equation: $Ax + By + C = 0$
  - Given a point P(x,y):

    P is on the line:

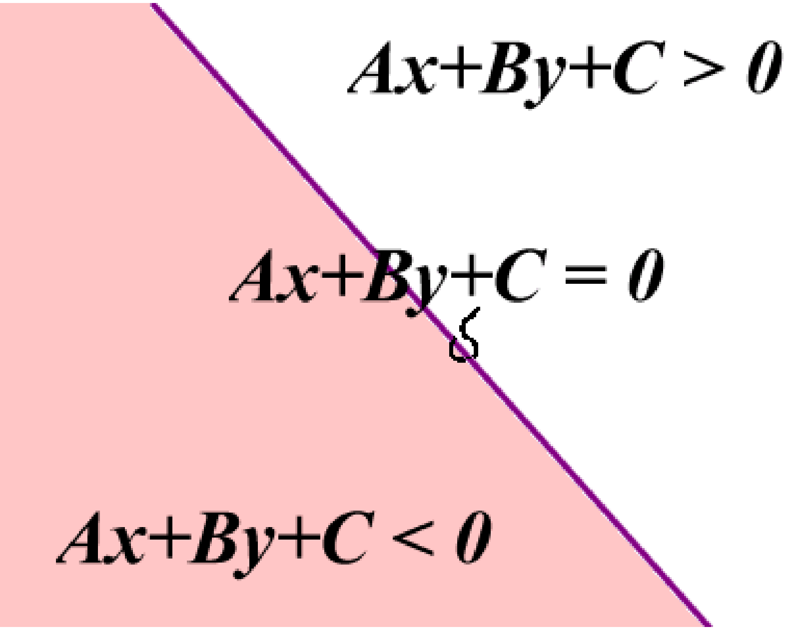
    $Ax + By + C = 0$

    P is above the line:

    $Ax + By + C > 0$

    P is below the line:

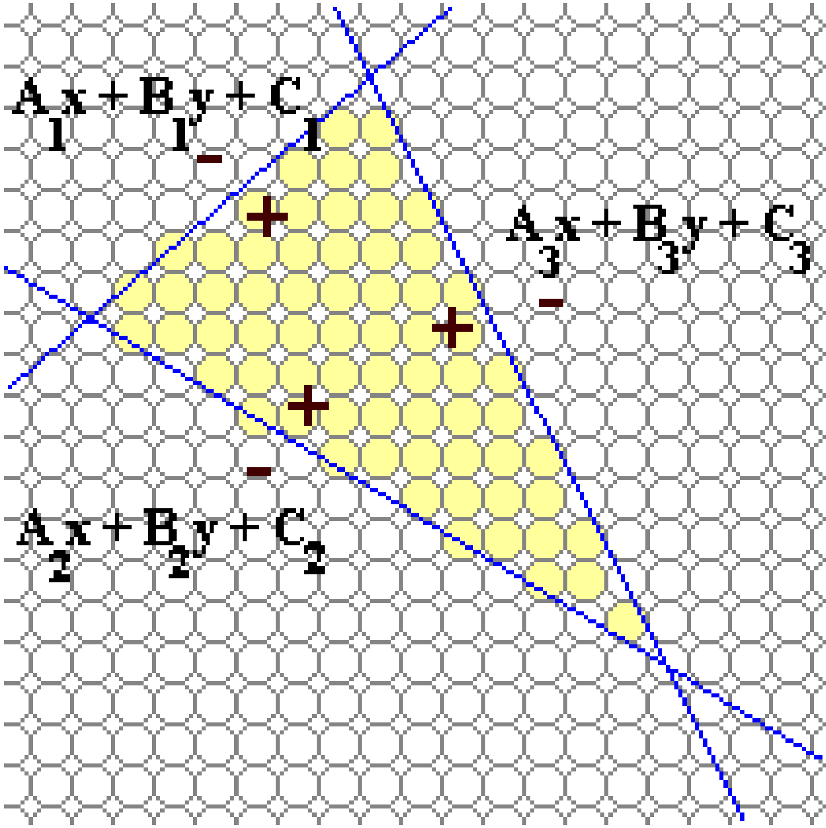
    $Ax + By + C < 0$



- An edge equation define two *half-spaces*

Triangle Rasterization by Edge Equations

- A triangle can be defined as the intersection of three positive half-spaces
  - We can choose which
  - half-space is positive by multiplying -1
  - Turn on those pixels for which all edge equations evaluate to > 0

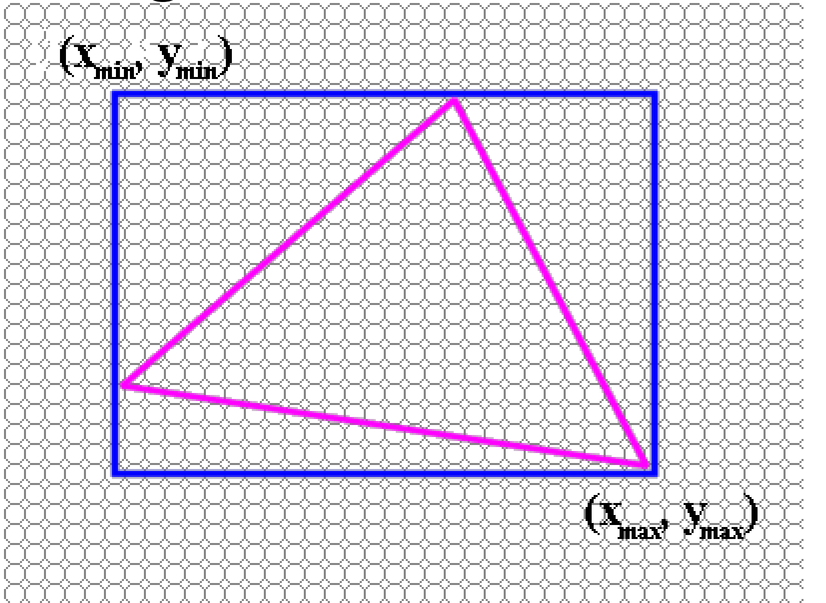


Edge-Equation Rasterizer: Implementation

- How to implement an edge-equation rasterizer in software?
  - Which pixels do you consider?
  - How do you compute the edge equations?
  - How do you orient the edges correctly?

Which pixels to consider?

- Screen space is large
  - Display resolution (HD): 1920 x 1080 (Megapixel)
  - It is in-efficient to test all pixels
- We can compute a bounding box
  - Only consider the pixels inside the bounding box



Compute Edge Equations?

- Edge equation can be computed using the coordinates of its two vertices (x_0,y_0) & (x_1,y_1)
- Treat it as a linear system:

  $Ax_0 + By_0 + C = 0$

  $Ax_1 + By_1 + C = 0$
- Two Equations, three unknowns?
  - Line equations are up to a scalar
  - Solve A and B in terms of C

Compute Coefficients

- Setup the linear system:

$$\begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = -C \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

- Multiply both side by inverse matrix:

$$\begin{bmatrix} A \\ B \end{bmatrix} = \frac{-C}{x_0 y_1 - x_1 y_0} \begin{bmatrix} y_1 - y_0 \\ x_1 - x_0 \end{bmatrix}$$

- If we choose $C = x_0 y_1 - x_1 y_0$
  - Then we have $A = y_0 - y_1$ and $B = x_0 - x_1$

Numerical Issue

- Calculating $C = x_0 y_1 - x_1 y_0$ involves some numerical precision issues
  - Floating point number subtraction has numerical precision issue
  - For example:
    - $\underline{1.234}\text{x}10^4 - \underline{1.233}\text{x}10^4 = \underline{1}.000\text{x}10^1$
    - We lose most of the significant digits in result
- When two vertices are very close to each other, we have this problem
  - $x_0 \approx x_1$, $y_0 \approx y_1$, thus $C = x_0 y_1 - x_1 y_0 \approx 0$

Numerical Issue

- We can avoid the subtraction by using our line equation:

  $Ax_0 + By_0 + C = 0$

  $Ax_1 + By_1 + C = 0$

- *So given* $A = y_0 - y_1$ and $B = x_1 - x_0$
  - *We have* $C = -Ax_0 - By_0$ or $C = -Ax_1 - By_1$
- Why is this better? Which should we choose?
  - We average the two to avoid bias:

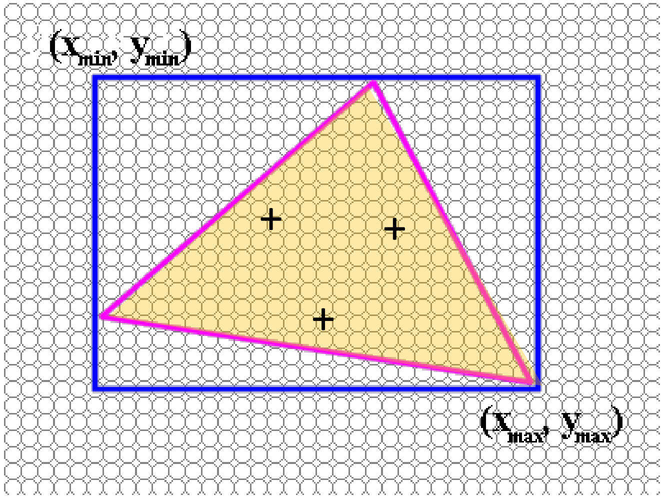
$$C = -[A(x_0+x_1) + B(y_0+y_1)] / 2$$

Edge Orientation?

- Now we know how to find edge equation from two vertices
- Given three vertices P_0, P_1, P_2 of a triangle, what would be the orientations of the three edge?
  - such that the half-spaces defined by the edge equations all share the same sign on the interior of the triangle
- Be consistent (e.g.: $[P_0 P_1]$, $[P_1 P_2]$, $[P_2 P_0]$)
- Test the sign for triangle interior on one edge
  - Flip if needed ($A = -A$, $B = -B$, $C = -C$)

Edge-Equation Rasterizer: Code

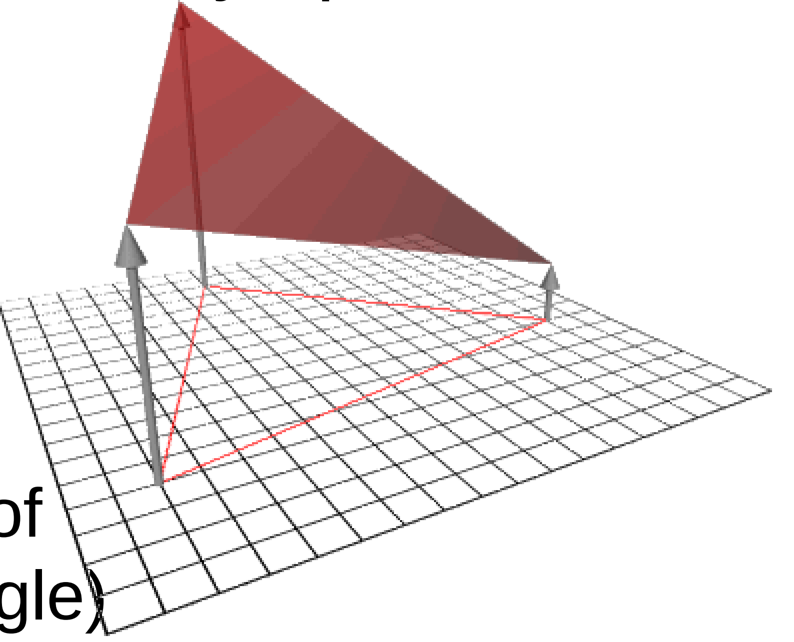
- Basic structure of code:
  - Setup: compute edge equations & bounding box
  - Outer loop: for each scanline in bounding box...
  - Inner loop: check each pixel on scanline, evaluating edge equations and drawing the pixel if all three are positive



Edge Equations: Interpolating Color

- Now we know how to draw a solid triangle (All vertices have the same color)
- What if they have different colors (or other parameters, e.g. depth)? How to interpolate?
- Idea: triangles are planar in any space:
  - This is the “redness” parameter space
  - Also need to do this for green and blue
  - Plane equation
    $z = A_r x + B_r y + C_r$

(here z stands for redness of a point (x,y) inside the triangle)



Edge Equations: Interpolating Color

- How to find the plane equation?
- Given redness values r_0, r_1, and r_2 at the 3 vertices, we can set up the linear system to for A_r ,B_r , and C_r

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$

Edge Equations: Interpolating Color

- Linear system:

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$

- The solution is

$$\frac{1}{2area} \begin{bmatrix} y_1 - y_2 & y_2 - y_0 & y_0 - y_1 \\ x_2 - x_1 & x_0 - x_2 & x_1 - x_0 \\ x_1 y_2 - x_2 y_1 & x_2 y_0 - x_0 y_2 & x_0 y_1 - x_1 y_0 \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$

Edge Equations: Interpolating Color

- Notice that the matrix elements are exactly the coefficients of the edge equations

$$\frac{1}{2area}\begin{bmatrix} A_2 & A_3 & A_1 \\ B_2 & B_3 & B_1 \\ C_2 & C_3 & C_1 \end{bmatrix}\begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$

$$2area = x_0y_1 - x_1y_0 + x_1y_2 - x_2y_1 + x_2y_0 - x_0y_2$$
$$= C_0 + C_1 + C_2$$

- So the setup of plane equation coefficients is easy and cost-effective
  - Simply take coefficients from the edge equation
  - Matrix multiplication