

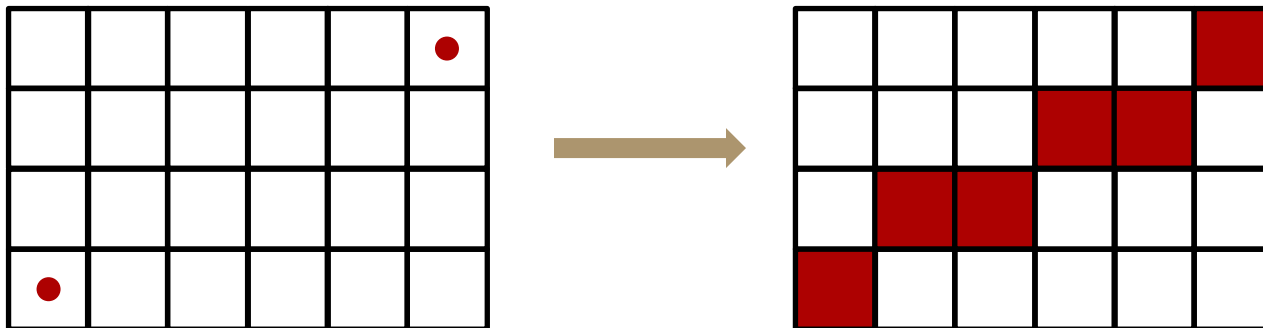
CENG – 477

Introduction to Computer Graphics

Rasterization

Rasterization

- Rasterization is concerned with creating **fragments** from **vertices**
- It works in screen coordinates – thus it is the next step after the **viewport transform**
- **Goal:** Given a set of vertices which fragments must be “turned on” to create the primitive:



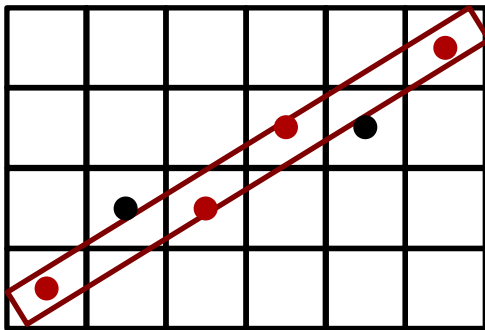
Rasterization

- **Subproblems:**
 - How to deal with different primitives
 - How to make it fast
 - How to interpolate color and other attributes
- We'll start with line rasterization

Line Rasterization

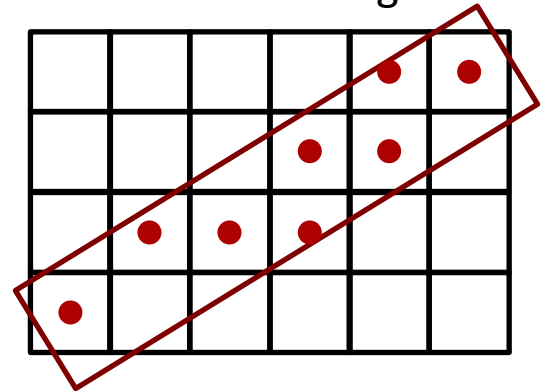
- Several methods exist
- **Option:** Treat the line as a thin rectangle and turn on pixels that are inside this rectangle

Too thin rectangle



- May cause gaps in the line

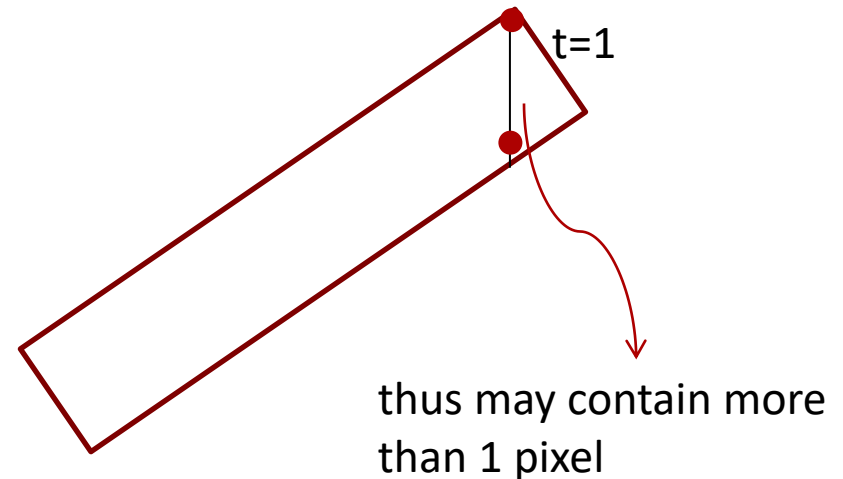
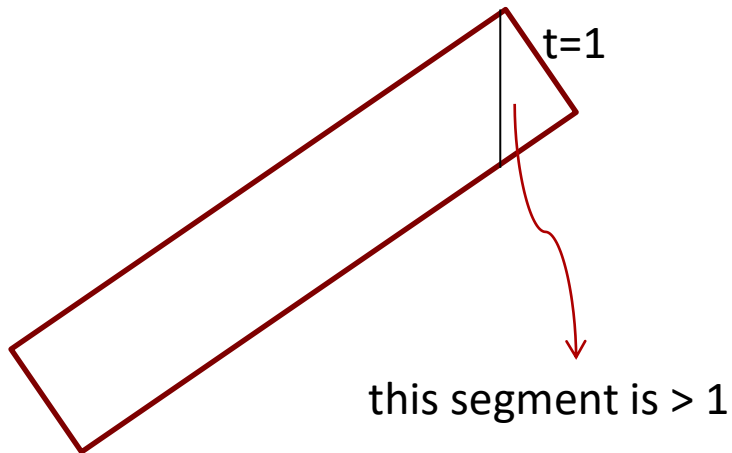
Too thick rectangle



- Line thickness varies

Line Rasterization

- Adjusting the line thickness could be problematic; different slopes may require different thicknesses



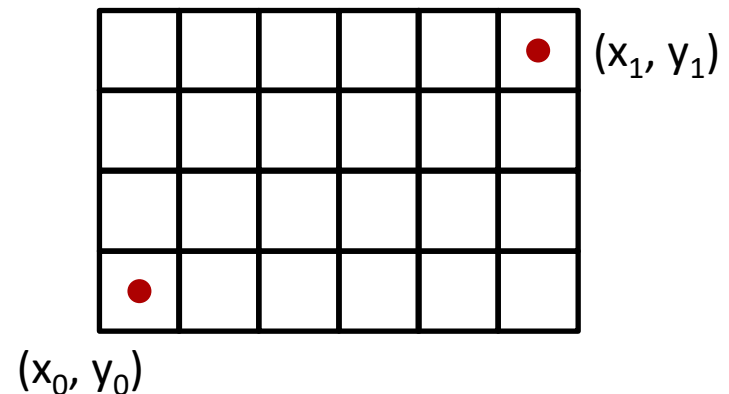
- Converting the line to a rectangle may be inefficient as well

Line Rasterization

- Therefore we need another solution
- We can use line equations to decide which pixels belong the line
- Assume that we want to draw a line between two screen coordinates: (x_0, y_0) to (x_1, y_1)
- Assume that the slope of the line, m , is in range $(0, 1]$

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

Which pixels should we draw?

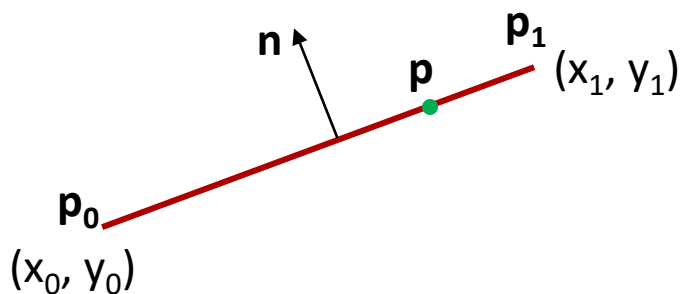


Line Equation

- Let's first remember the implicit line equation:

$$f(x, y) = x(y_0 - y_1) + y(x_1 - x_0) + x_0y_1 - y_0x_1$$

- We can derive it from geometry:



$$\mathbf{n} \cdot [x_1 - x_0, y_1 - y_0]^T = 0$$

$$\mathbf{n} = [y_0 - y_1, x_1 - x_0]$$

$$f(\mathbf{p}) = (\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n} = 0$$

What is the meaning of $f(\mathbf{p}) = 0$, $f(\mathbf{p}) > 0$ and $f(\mathbf{p}) < 0$

The Basic Algorithm

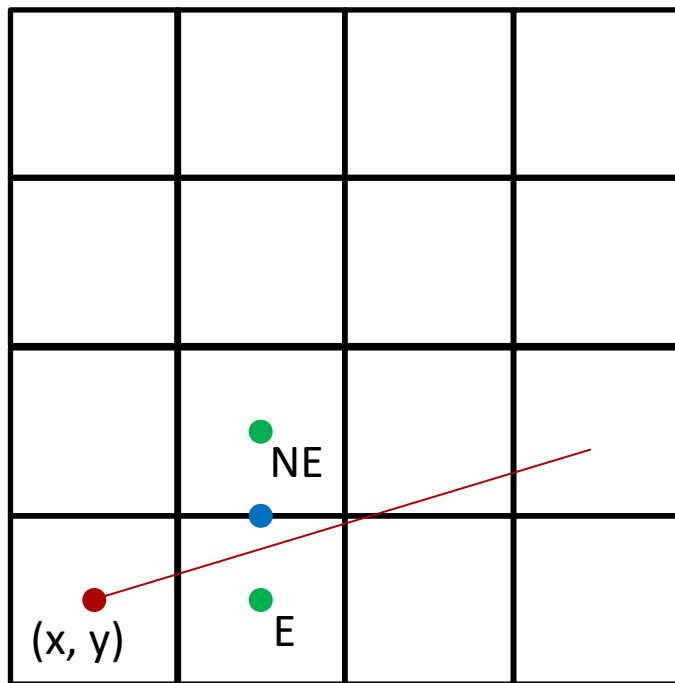
- The basic algorithm is as follows:

```
y = y0  
for x = x0 to x1 do:  
    draw(x, y)  
    if (some condition) then:  
        y = y + 1
```

- Because the slope is in $(0, 1]$, we always go right and sometimes go up (assuming that $x_0 < x_1$ and $y_0 < y_1$)

The Midpoint Algorithm

- Assume we have just drawn (x, y) . Which pixel to draw next?



- Compute the midpoint of the next pixel
- If the midpoint is on or above the line choose the right pixel (E: east)
- If the midpoint is below the line, choose the top right pixel (NE: north-east)

$$y = y_0$$

for $x = x_0$ to x_1 **do**:

draw(x, y)

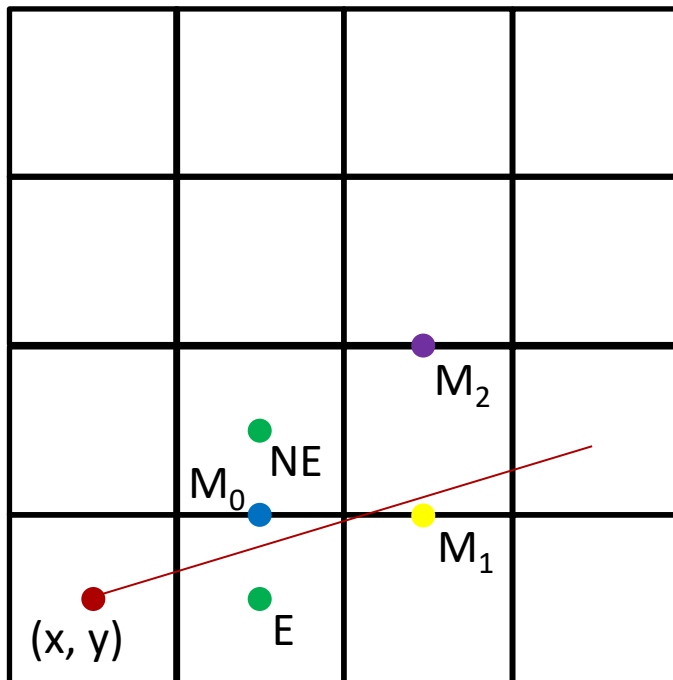
if $f(x+1, y+0.5) < 0$ **then**:

$$y = y + 1$$

We call these pixels E and NE

The Midpoint Algorithm

- This algorithm works well but requires evaluating the line equation at every iteration
- Can be optimized with an **incremental** algorithm:



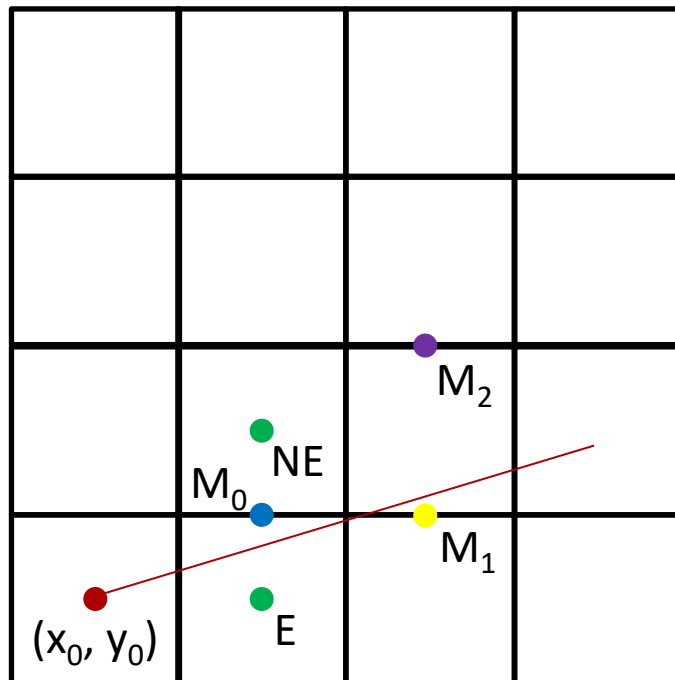
- If we selected E, in the next iteration we need the value of $f(\mathbf{M}_1)$
- If we selected NE, in the next iteration we need $f(\mathbf{M}_2)$
- Both can be computed from $f(\mathbf{M}_0)$

$$f(M_1) - f(M_0) = y_0 - y_1$$

$$f(M_2) - f(M_0) = (y_0 - y_1) + (x_1 - x_0)$$

The Midpoint Algorithm

- This algorithm works well but requires evaluating the line equation at every iteration
- Can be optimized with an **incremental** algorithm:



- In other words, if we know $f(\mathbf{M})$, we can compute the next $f(\mathbf{M})$ by simple integer arithmetic
- What is the first $f(\mathbf{M})$?
- Note that $f(x_0, y_0) = 0$ as it is the starting point of the line

$$f(M_0) = f(x_0 + 1, y_0 + 0.5)$$

$$= (y_0 - y_1) + 0.5(x_1 - x_0)$$

The Midpoint Algorithm

- So the overall algorithm is:

```
y = y0
d = (y0 - y1) + 0.5(x1 - x0)
for x = x0 to x1 do:
    draw(x, y)
    if d < 0 then: // choose NE
        y = y + 1
        d += (y0 - y1) + (x1 - x0)
    else: // choose E
        d += (y0 - y1)
```

The Midpoint Algorithm

- For max efficiency, $f(x, y) = 0$ is written as $2f(x, y) = 0$. This entirely eliminates floating point operations:

```
y = y0
d = 2(y0 - y1) + (x1 - x0)
for x = x0 to x1 do:
    draw(x, y)
    if d < 0 then: // choose NE
        y = y + 1
        d += 2[(y0 - y1) + (x1 - x0)]
    else: // choose E
        d += 2(y0 - y1)
```

The Midpoint Algorithm

- The presented algorithm works when the slope of the line, m , is in range $(0, 1]$
- For other slope ranges, minor modifications to the algorithm is required
- For instance if $m \in (1, \infty)$, we need to swap the roles of x and y
- Other cases must be adapted similarly

Float vs Integer

- The midpoint algorithm was originally developed by Pitteway in 1967
- Floating point arithmetic was very expensive at the time
- Does it still matter?
- We implemented both algorithms and drew one million lines each between 1000 and 1400 pixels long:
 - Test run on Intel Core i7 CPU at 3.2 GHz
 - Compiled with g++ and -O2 option
 - Basic algorithm: 7.2 seconds
 - Optimized algorithm: 3 seconds
 - So it still makes a difference!

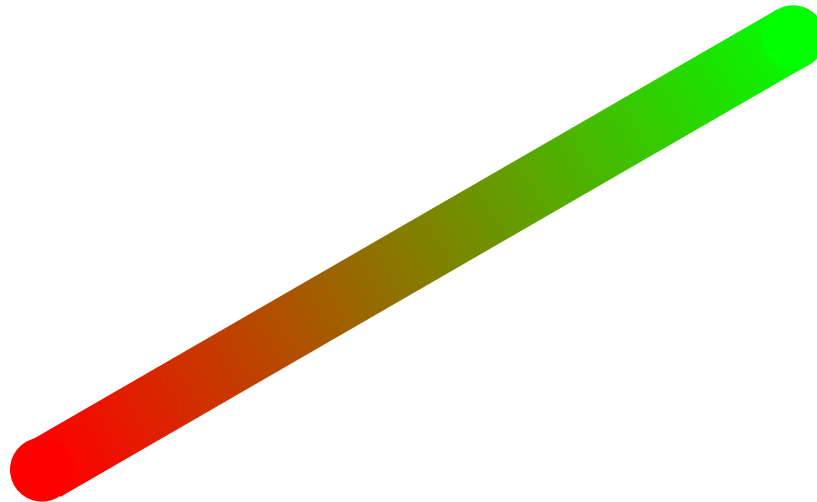
Interpolating Attributes

- What if the two end points of the line have a different color?
- The color across the line must smoothly change:

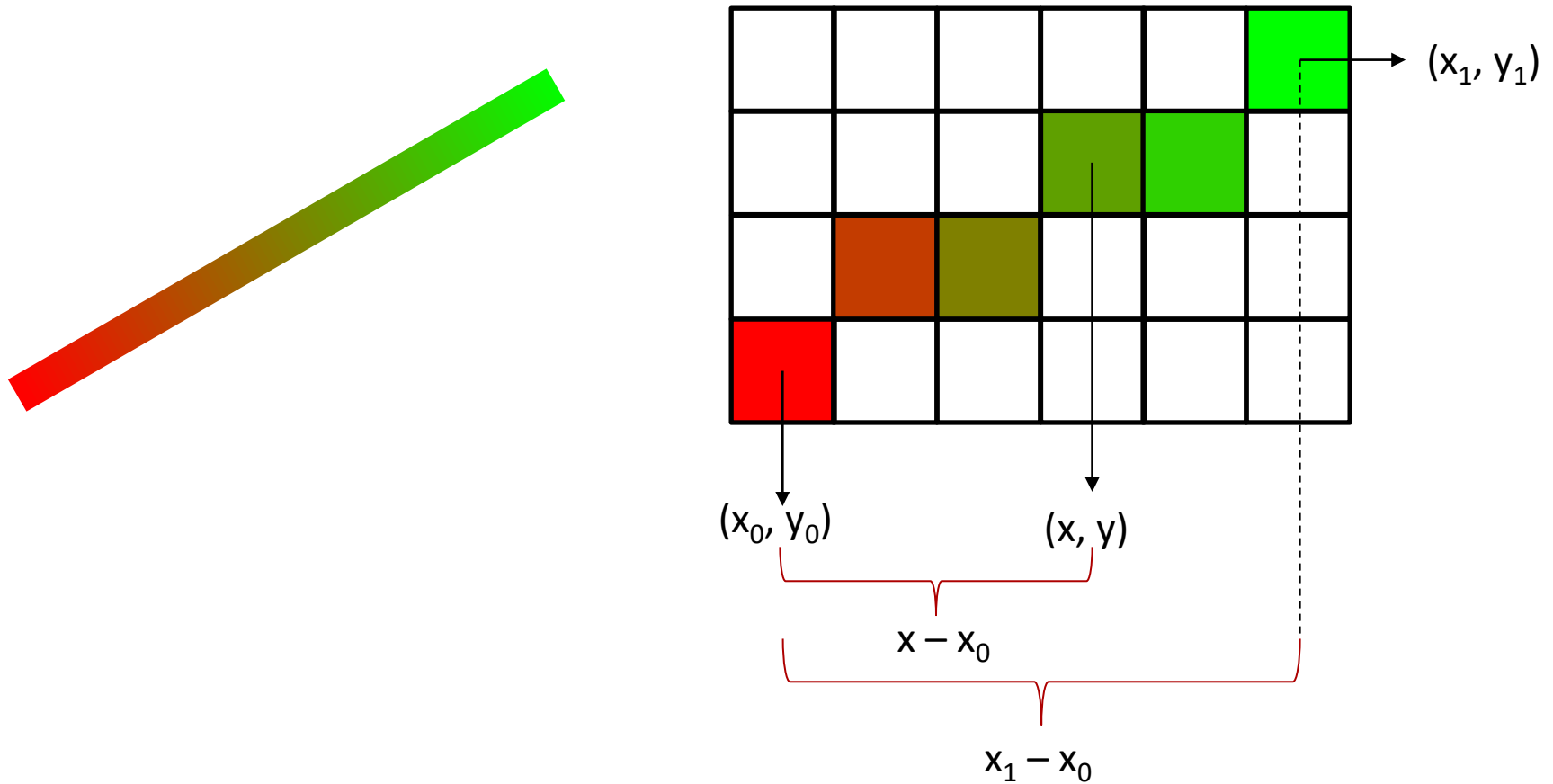


Interpolating Attributes

- What if the two end points of the line have a different color?
- The color across the line must smoothly change:



Interpolating Attributes



Interpolating Attributes

- Assume that the color of the endpoints are c_0 and c_1
- The color of an intermediate point should be:

$$c = (1 - \alpha)c_0 + \alpha c_1$$

where α is the interpolation variable

- At any pixel (x, y) , we can compute it based on the horizontal or vertical distance of the pixel to the first endpoint:

$$\alpha = \frac{x - x_0}{x_1 - x_0}$$

For $m \in [0,1]$ it is better to interpolate in the x direction as as it will produce a smoother variation

Interpolating Attributes

- Assume that the color of the endpoints are c_0 and c_1
- The color of an intermediate point should be:

$$c = (1 - \alpha)c_0 + \alpha c_1$$

where α is the interpolation variable

- It can also be computed incrementally

$$\alpha(x_0 + 1) = \alpha_1 = \frac{1}{x_1 - x_0}$$

$$\alpha(x_0 + 2) = \alpha_2 = \alpha_1 + \frac{1}{x_1 - x_0}$$

Algorithm with Interpolation

- Algorithm with color interpolation:

$y = y_0$

$d = (y_0 - y_1) + 0.5(x_1 - x_0)$

$c = c_0$

$dc = (c_1 - c_0) / (x_1 - x_0)$ // skip α ; directly compute color increment

for $x = x_0$ to x_1 **do**:

draw(x , y , round(c))

if $d < 0$ **then**: // choose NE

$y = y + 1$

$d += (y_0 - y_1) + (x_1 - x_0)$

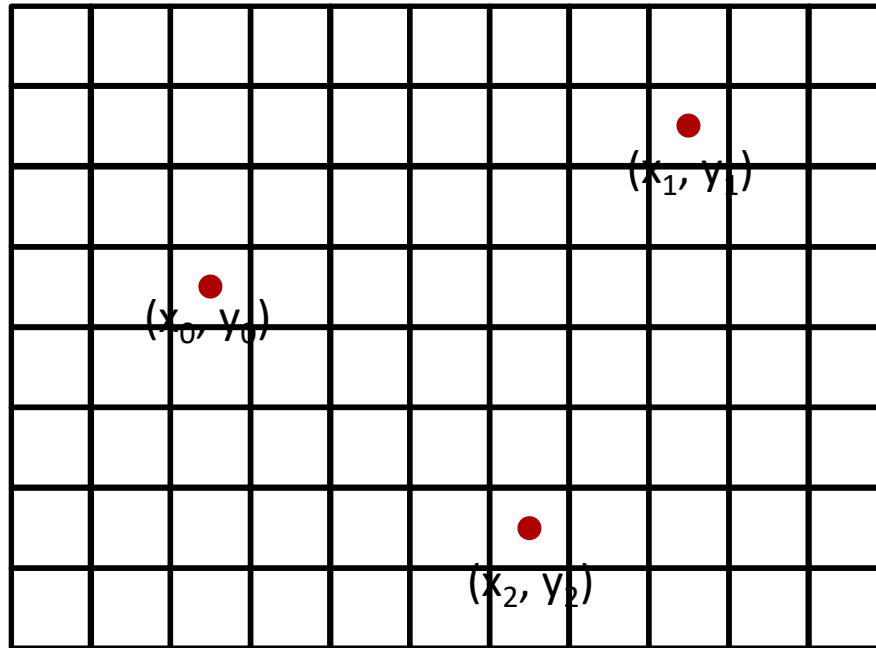
else: // choose E

$d += (y_0 - y_1)$

$c += dc$

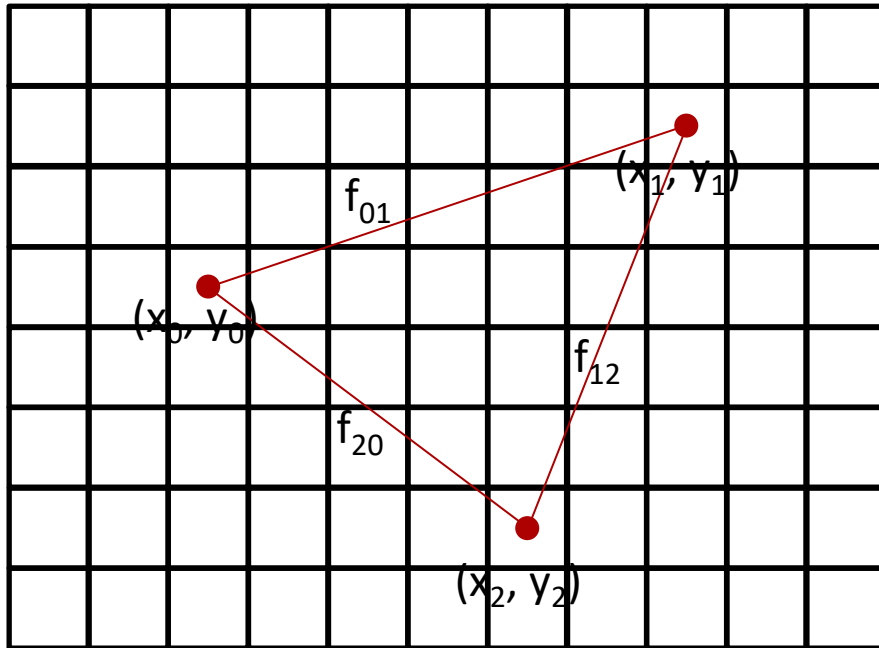
Triangle Rasterization

- Initially all we have is the screen coordinates of three vertices



Triangle Rasterization

- From them we compute line equations for the edges:



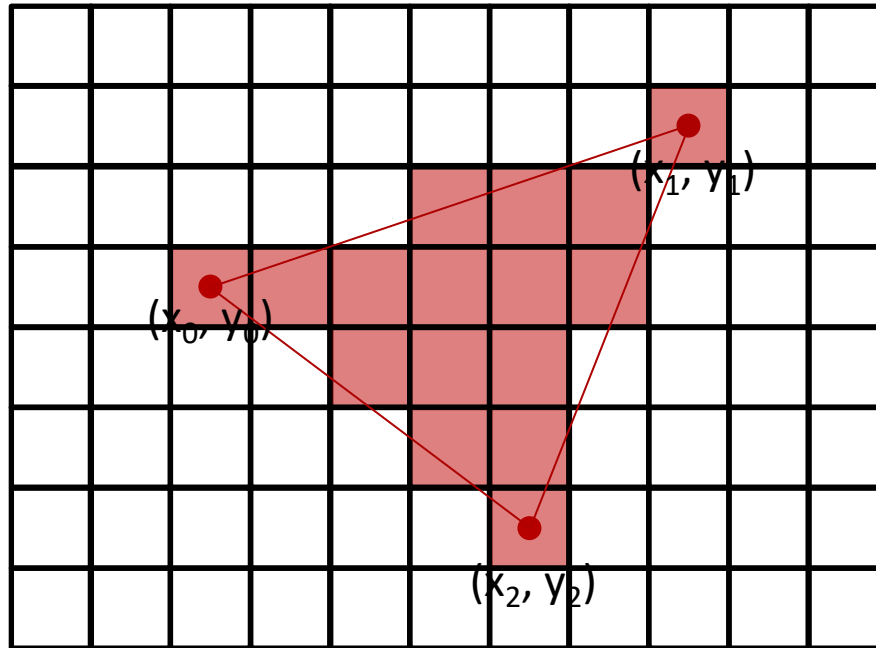
$$f_{01}(x, y) = x(y_0 - y_1) + y(x_1 - x_0) + x_0y_1 - y_0x_1$$

$$f_{12}(x, y) = x(y_1 - y_2) + y(x_2 - x_1) + x_1y_2 - y_1x_2$$

$$f_{20}(x, y) = x(y_2 - y_0) + y(x_0 - x_2) + x_2y_0 - y_2x_0$$

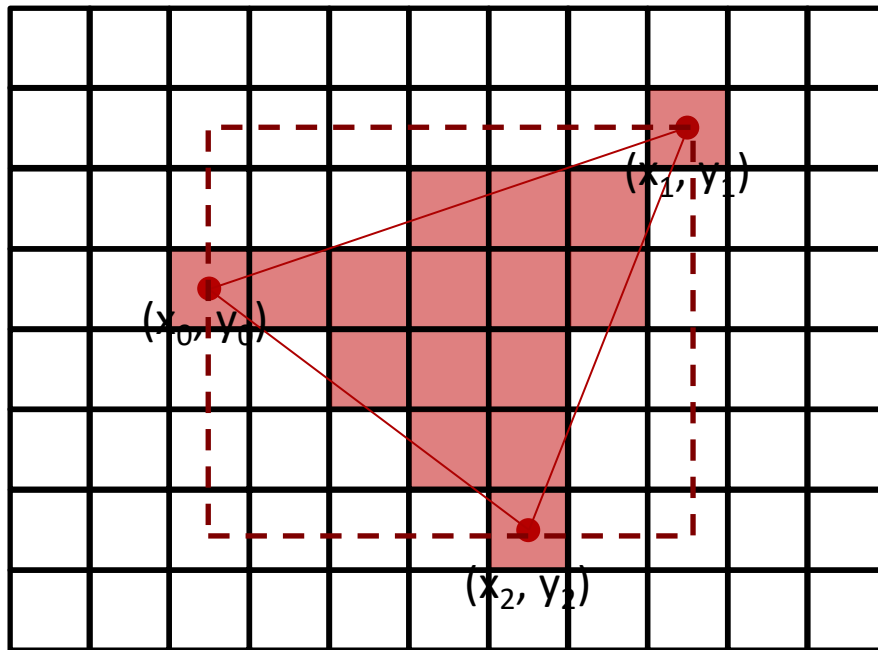
Triangle Rasterization

- We then walk across the viewport to determine inside pixels:



Triangle Rasterization

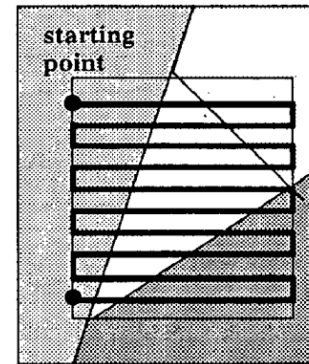
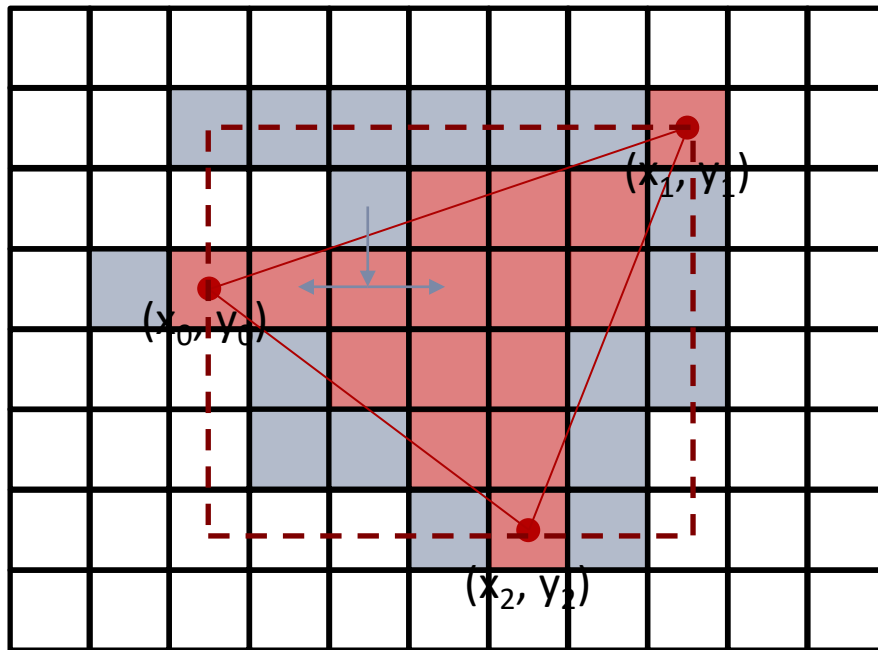
- For efficiency we may only walk within the bounding box of the triangle:



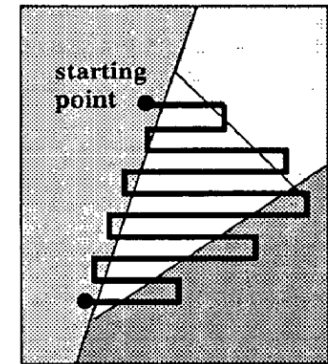
- For each pixel we visit, we must make an inside test with respect to all three edges
- We can simply plug-in the (x, y) value of the visited pixel to each line equation
- If all are negative, the pixel is inside the triangle

Triangle Rasterization

- We can also go to the next scanline if we exit the triangle
- This is known as Pineda rasterization



Traversing the Bounding Box

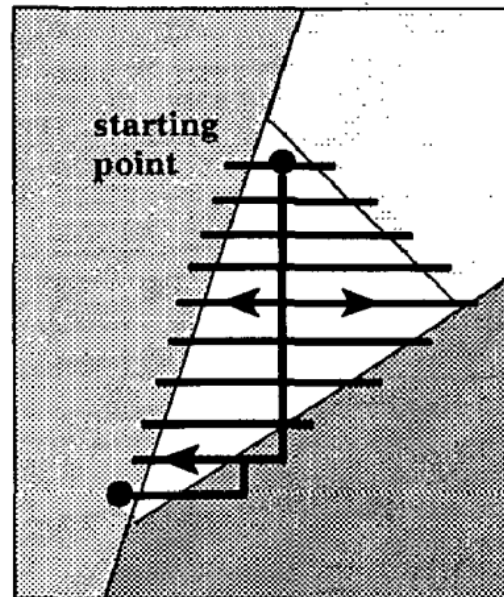


A More Efficient Traversal Algorithm

Pineda, J. (1988, June). A parallel algorithm for polygon rasterization. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (pp. 17-20).

Triangle Rasterization

- We can also go from the center to outward



See also:

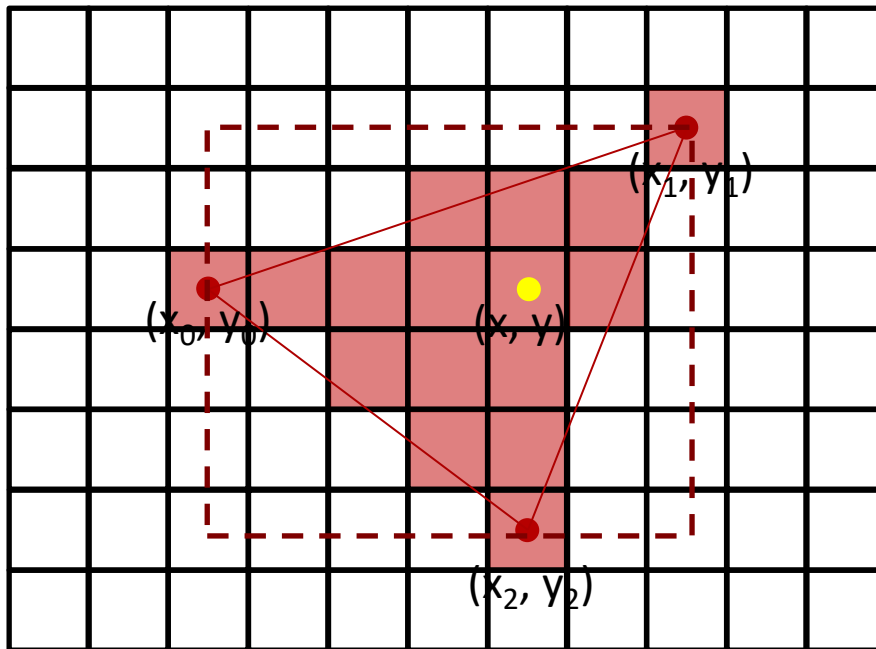
https://www.mathematik.uni-marburg.de/~thormae/lectures/graphics1/code_v2/RasterPoly/index.html

Figure 5. Smarter Algorithm Proceeds Outward From Center Line

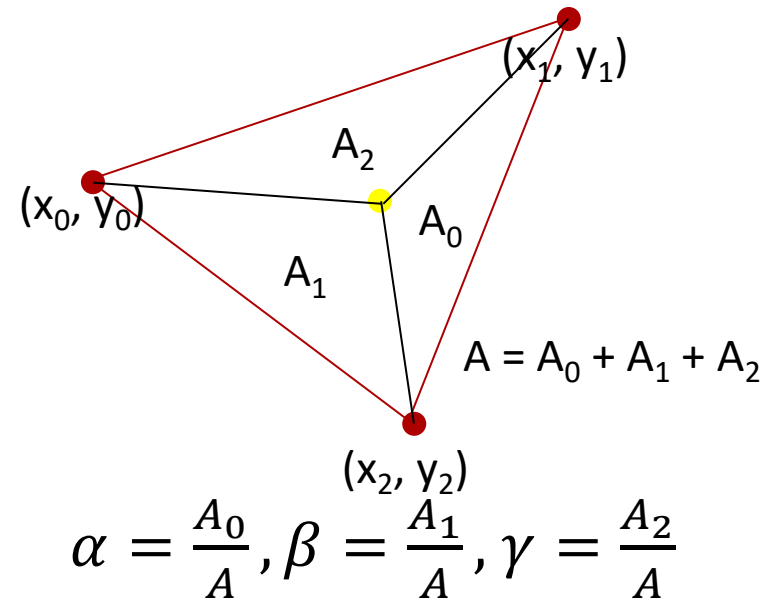
Pineda, J. (1988, June). A parallel algorithm for polygon rasterization. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (pp. 17-20).

Triangle Rasterization

- However, using barycentric coordinates will help us with interpolation of attributes



What are the barycentric coordinates of (x, y) ?



Triangle Rasterization

- We don't actually need to compute the areas as the bases are shared and will cancel out

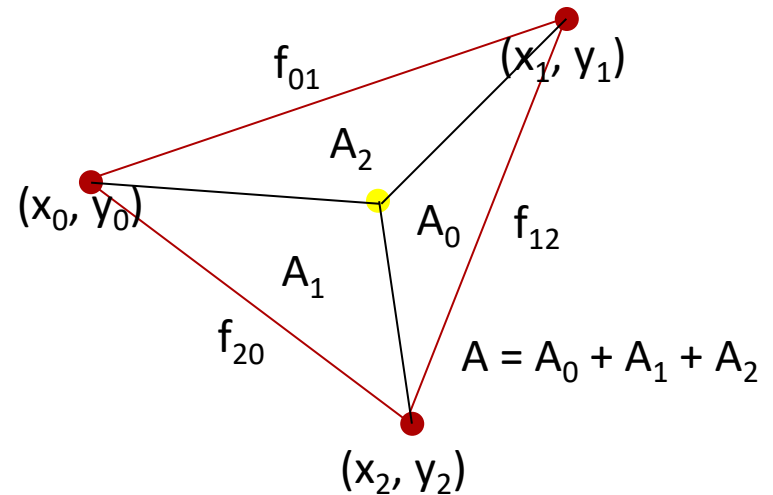
$$\alpha = \frac{A_0}{A}, \beta = \frac{A_1}{A}, \gamma = \frac{A_2}{A}$$

$$\alpha = \frac{f_{12}(x, y)}{f_{12}(x_0, y_0)}$$

$$\beta = \frac{f_{20}(x, y)}{f_{20}(x_1, y_1)}$$

$$\gamma = \frac{f_{01}(x, y)}{f_{01}(x_2, y_2)}$$

What are the barycentric coordinates of (x, y) ?



Overall Algorithm

```
for  $y = y_{\min}$  to  $y_{\max}$  do:  
  for  $x = x_{\min}$  to  $x_{\max}$  do:  
     $\alpha = f_{12}(x, y) / f_{12}(x_0, y_0)$   
     $\beta = f_{20}(x, y) / f_{20}(x_1, y_1)$   
     $\gamma = f_{01}(x, y) / f_{01}(x_2, y_2)$   
    if  $\alpha \geq 0$  and  $\beta \geq 0$  and  $\gamma \geq 0$  then:  
       $c = \alpha c_0 + \beta c_1 + \gamma c_2$   
      draw( $x, y, \text{round}(c)$ )
```

- Note that the computation of the barycentric coordinates can also be made incremental for greater efficiency

Summary

- The result of rasterization is a set of **fragments** (pixel-to-be) for each primitive
- Each fragment has interpolated values of attributes:
 - Color values
 - Texture coordinates
 - Depth value
 - Normals
 - Or any user-defined attribute for vertices
- The next stage in the life of a fragment is to go through the fragment pipeline