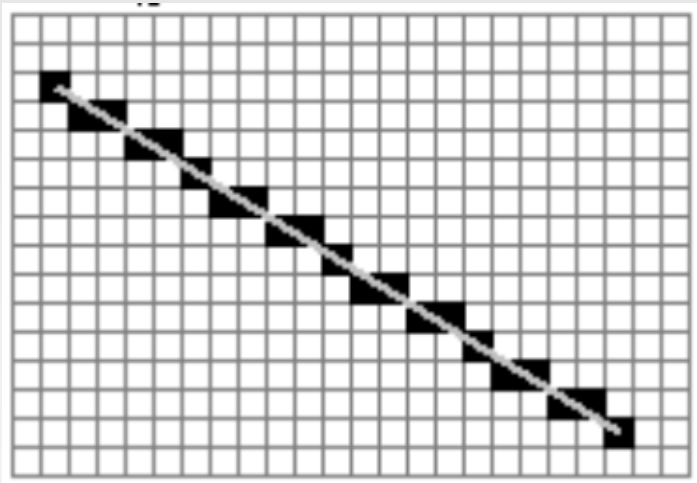
UNIP

UNIVERSIDADE PAULISTA

Computação Gráfica



Rasterização de Linhas

Professora Sheila Cáceres

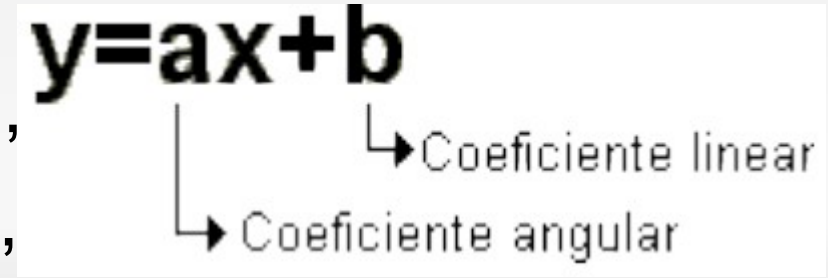
Equação da Reta

- Apresenta a seguinte lei de formação:

$$y=f(x) = ax + b$$

sendo a e b números reais e a diferente de zero.

- **Observação**: a e b são chamados de coeficientes e x é a variável independente.
- Exemplos:
  - f(x) = x + 2,
  - y = -2x + 6,



Raiz de uma função de primeiro grau

- A raiz de uma função do primeiro grau é o valor que, substituído no lugar de x, faz com que f(x) seja igual a zero (f(x)=0).
- Encontramos a raiz dessa função igualando ax + b a zero.
- Exemplos:

$f(x) = 2x - 4$

$2x - 4 = 0$

$2x = 4$

$x = 2$ (raiz)

Gráfico de uma função de 1° grau

Gráfico

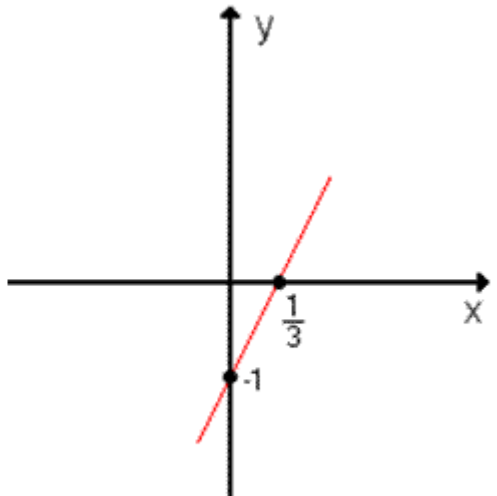
O gráfico de uma função polinomial do 1º grau, $y = ax + b$, com $a \neq 0$, é uma reta oblíqua aos eixos Ox e Oy.

Vamos construir o gráfico da função $y = 3x - 1$. Como o gráfico é uma reta, basta obtermos dois de seus pontos e "ligá-los" com o auxílio de uma régua.

- Para $x = 0$, temos que $y = 3.0 - 1 = -1$. Logo, um ponto da reta é $(0, -1)$.
- Para $y = 0$, temos que $0 = 3x - 1$. Logo, outro ponto da reta é $\left(\frac{1}{3}, 0\right)$.

Marcamos os pontos $(0, -1)$ e $\left(\frac{1}{3}, 0\right)$ no plano cartesiano e os "ligamos" com uma reta, conforme imagem abaixo.

x	y
0	-1
$\frac{1}{3}$	0



- Podemos atribuir valores arbitrários para x e obteremos os valores respectivos para y.
- Exemplo

a) **f(x) = 2x + 4**

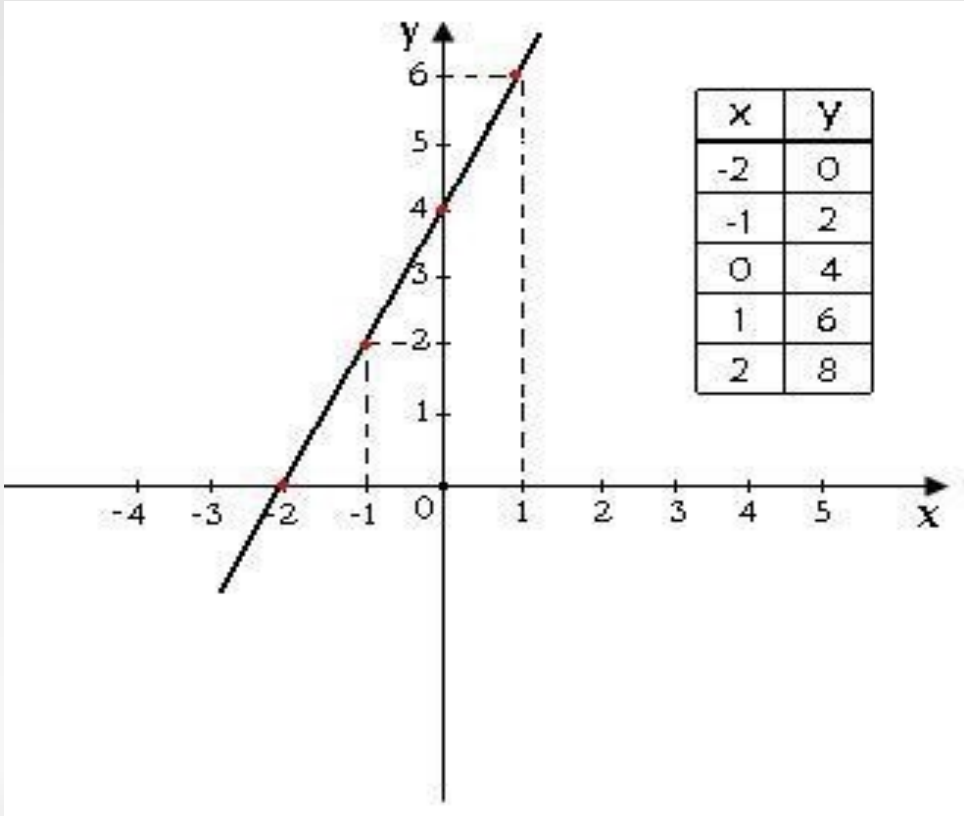
f(x) = 2.(-2) + 4 = 0

f(x) = 2.(-1) + 4 = 2

f(x) = 2.(0) + 4 = 4

f(x) = 2.(1) + 4 = 6

f(x) = 2.(2) + 4 = 8



x	y
-2	0
-1	2
0	4
1	6
2	8

b) **f(x) = - x + 3**

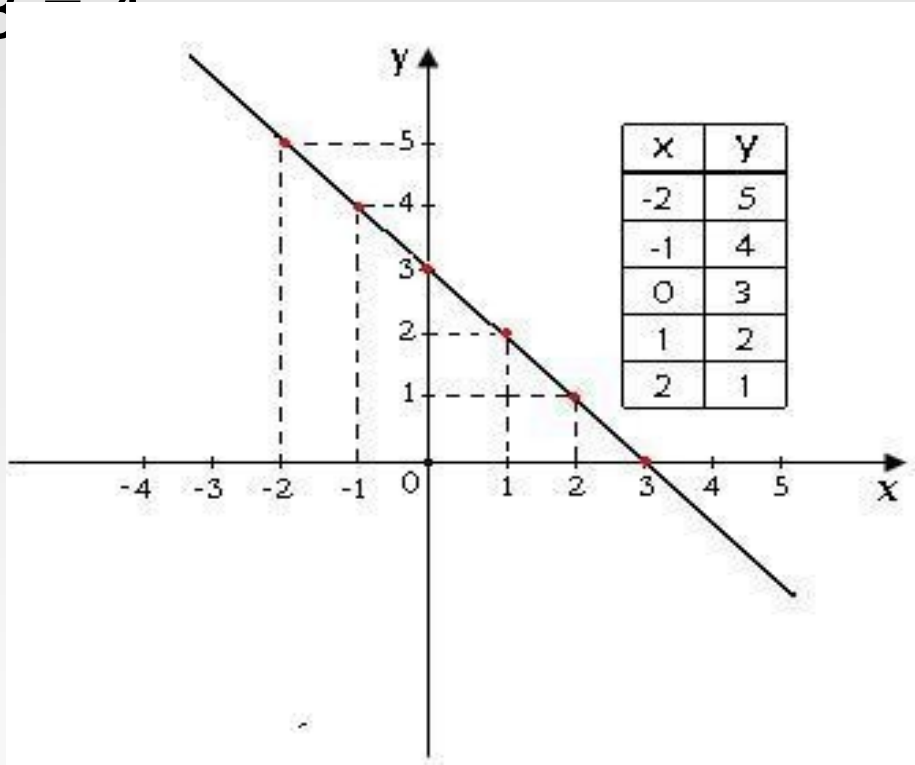
f(x) = - (-2) + 3 = 2 + 3 = 5

f(x) = - (-1) + 3 = 1 + 3 = 4

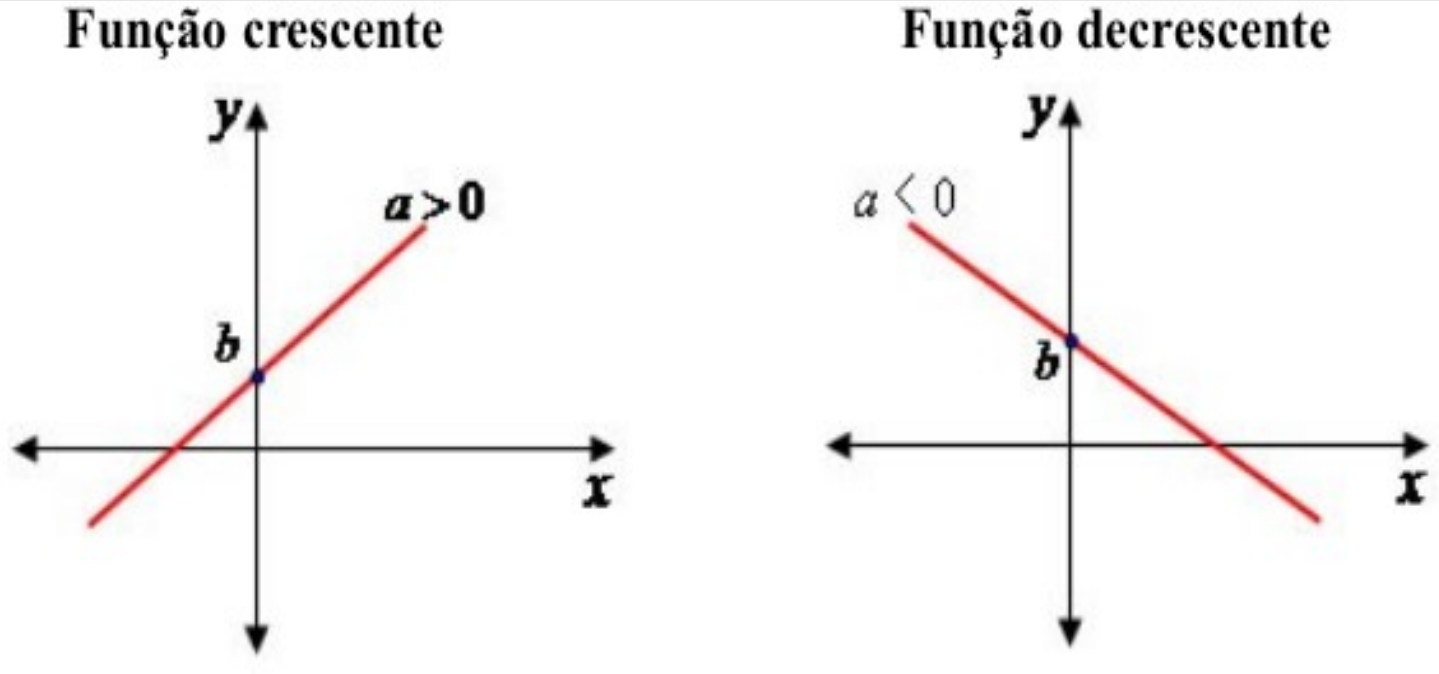
f(x) = - (0) + 3 = 3

f(x) = - (1) + 3 = 2

f(x) = - (2) + 3 = 1



Coeficiente Angular (a)



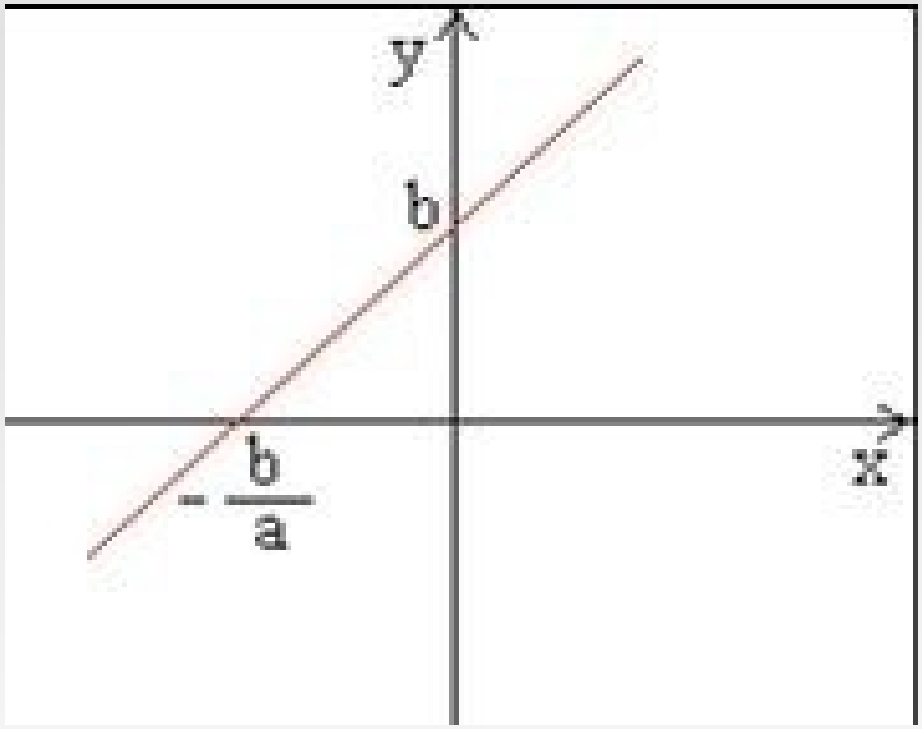
Declividade

- Chamamos declividade da reta à tangente do ângulo que a reta forma com o eixo dos x.
- Na função de primeiro grau, esta tangente tem valor igual ao coeficiente a, que é denominado coeficiente angular da reta.
- O coeficiente b é chamado de coeficiente linear.
- A partir do gráfico podemos determinar o valor do

$$a = \frac{y_1 - y_1}{x_1 - x_1}, \text{ onde } A(x_1, y_1) \text{ e } B(x_1, y_1)$$

ntos A

Interceptos



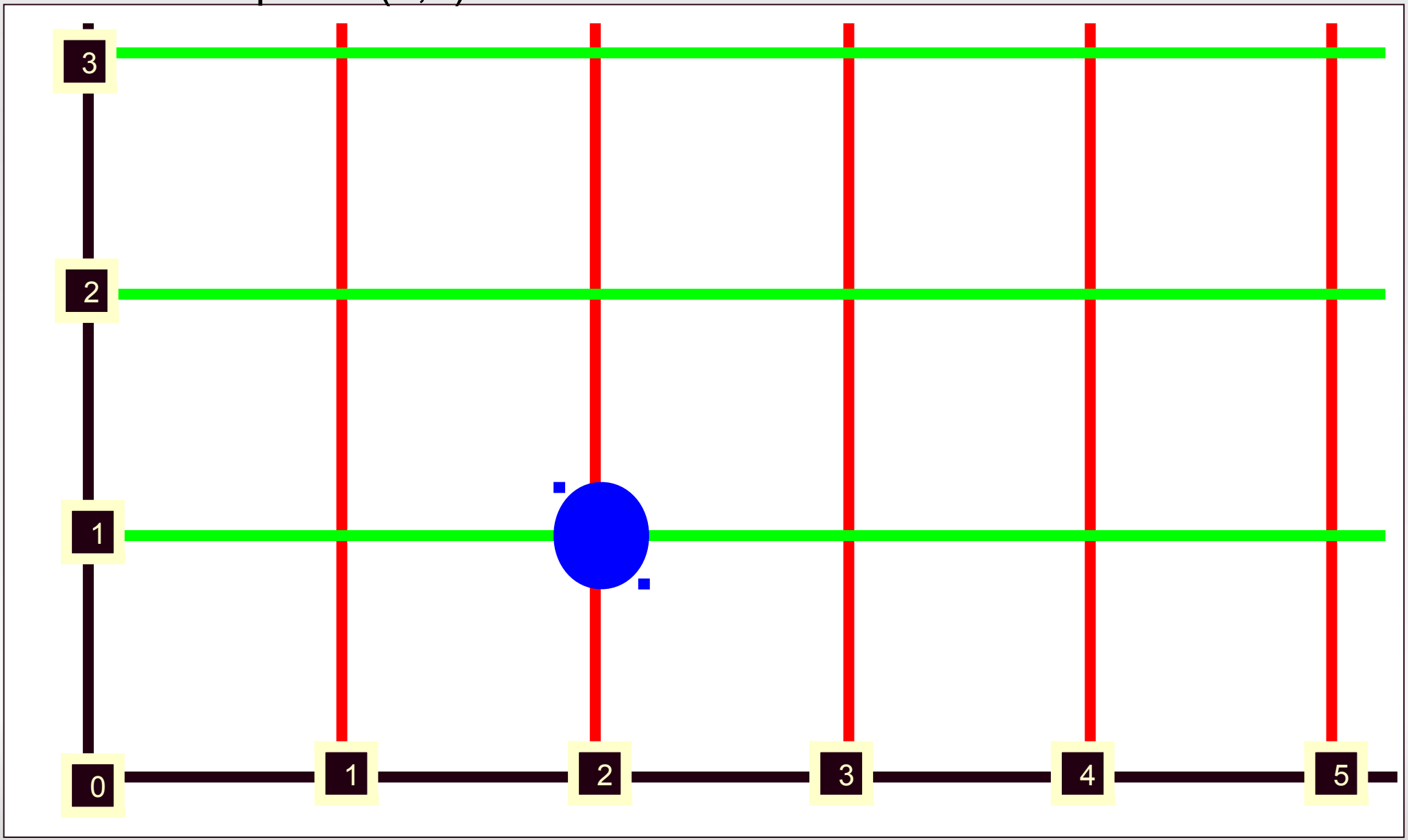
Rasterização

- Rasterização (rasterization or scan conversion) Converte informação de vertices / arestas em informação de pixels a serem mostrados.

  Ou seja, desenha as formas baseado somente nos vértices.
- **Utilização**
- Poderiamos obter cada ponto numa reta por exemplo mapeando desde o SRU para o SRD, mas:
- Em vez de mapearmos do SRU para o SRD cada ponto de uma aresta que liga dois

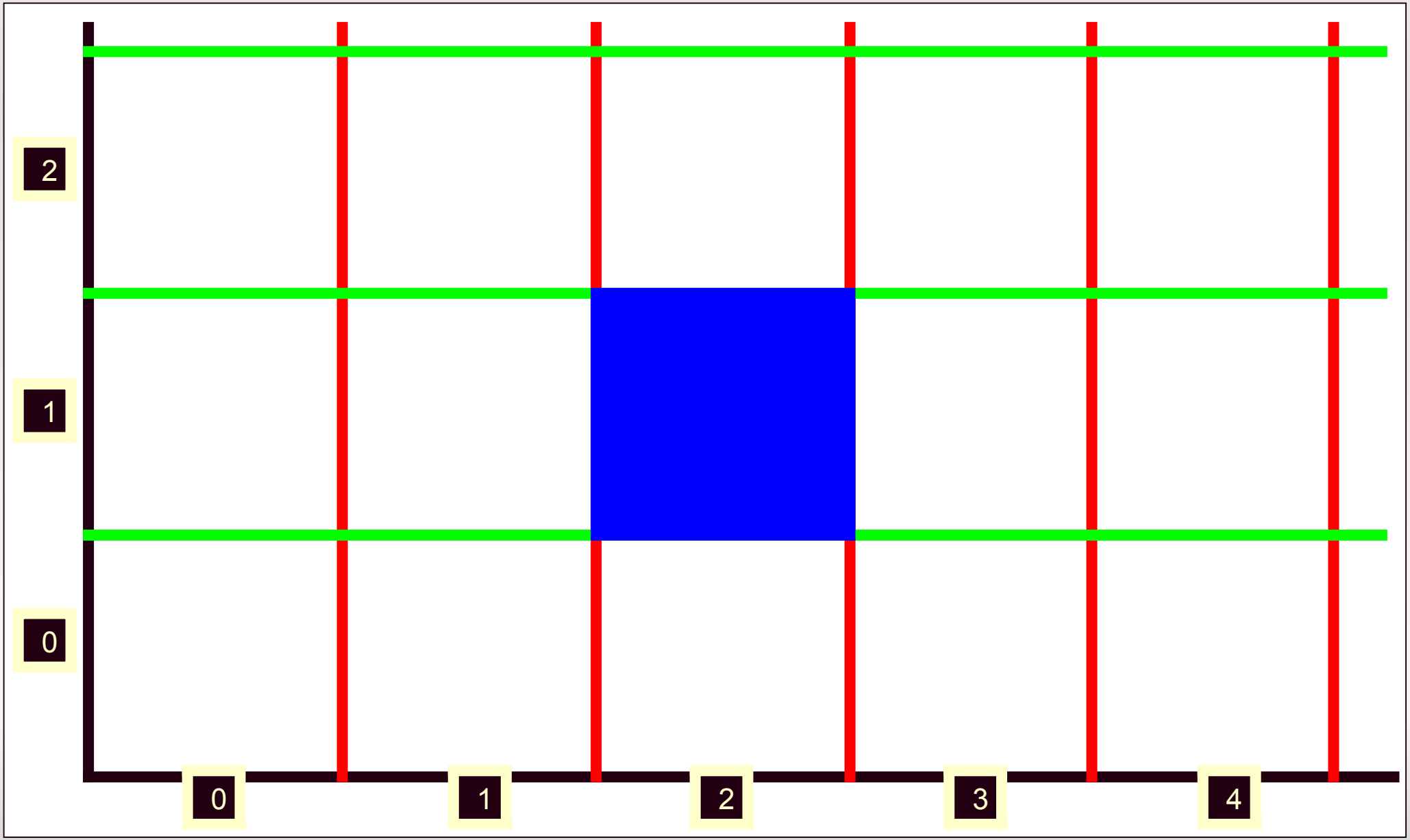
No SRU

Onde está o ponto (2,1)?



No SRD

Onde está (2,1)?



In java:

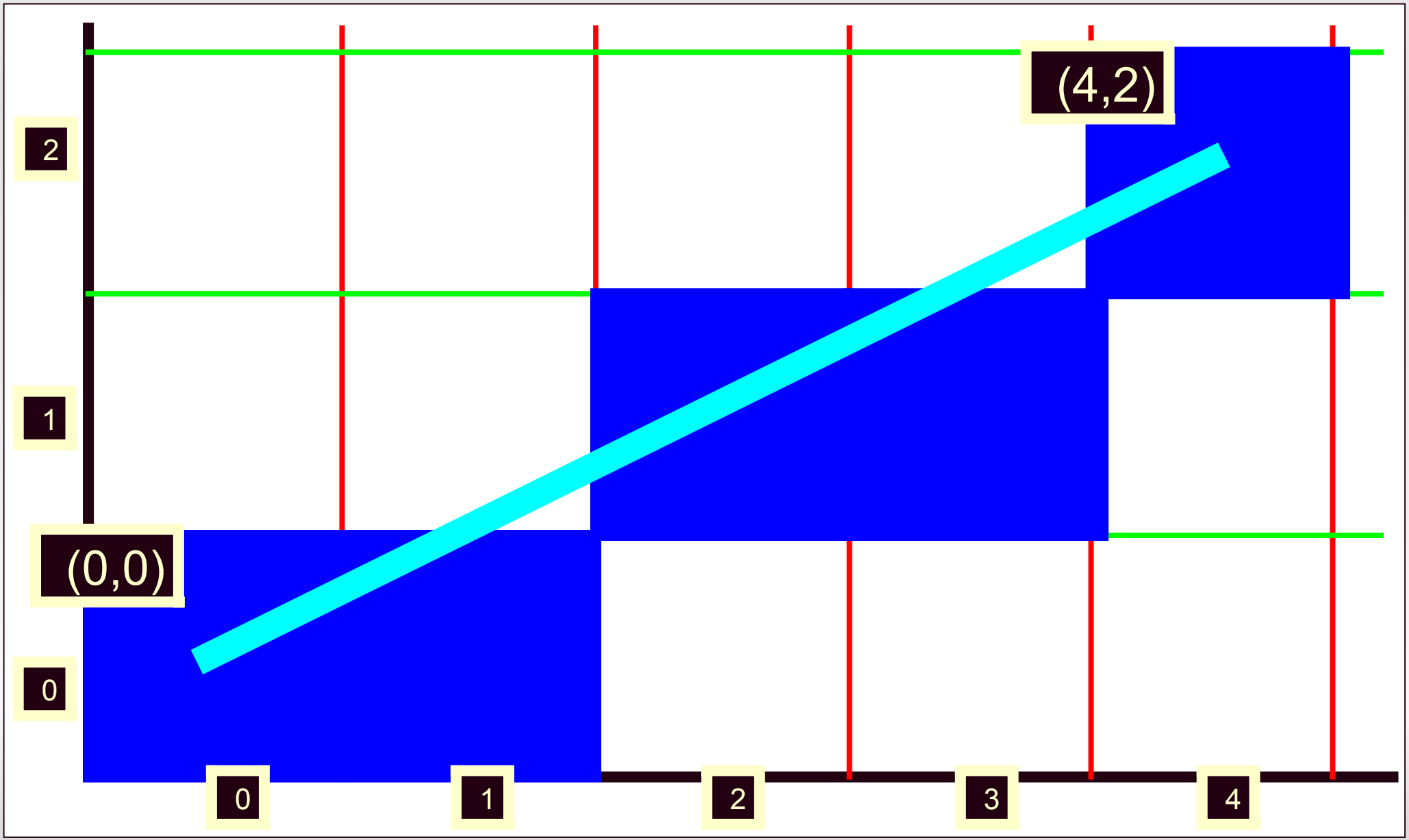
- Java não possui um método com o único propósito de colocar um pixel na tela, portanto, podemos definir um método para isso:

```
public void putPixel(Graphics g, int x, int y){
        g.drawLine(x,y,x,y);
   }
```

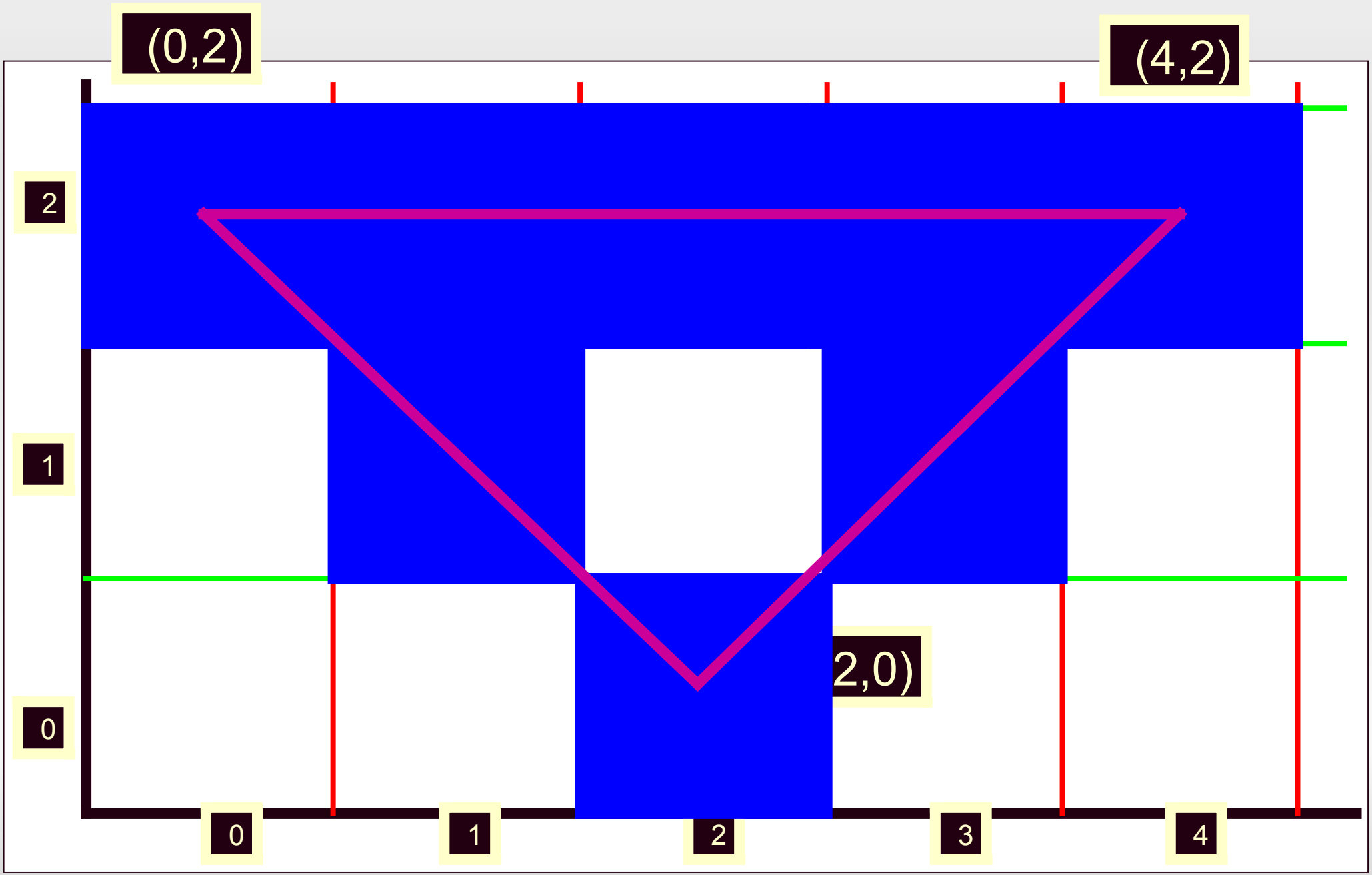
Rasterização de Linhas

Draw a line from 0,0 to 4,2

How do we choose between 1,0 and 1,1? What would be a good heuristic?



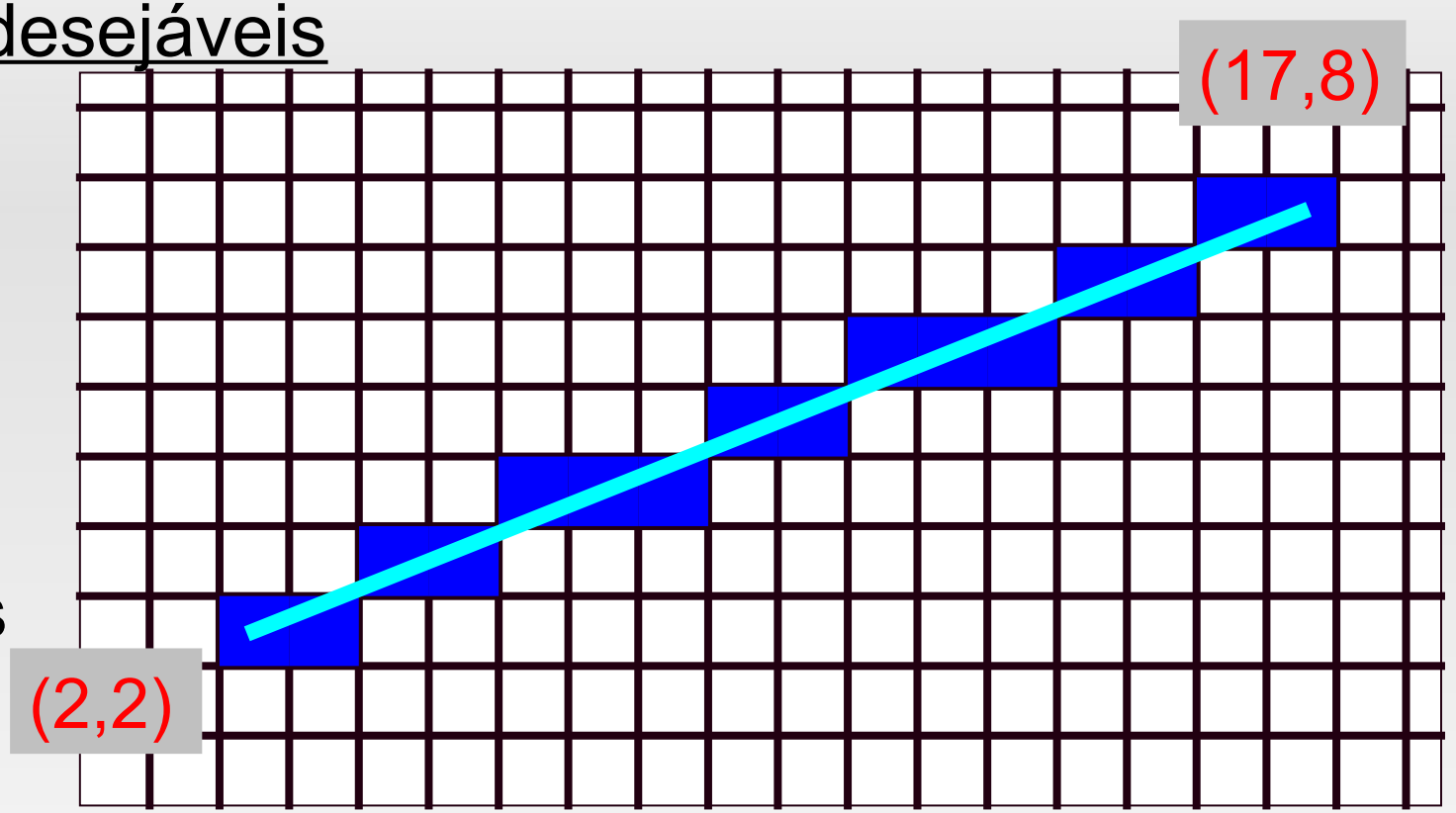
Let’s draw a triangle



A linha ideal

Características desejáveis

- Aparência continua
- Uniformidade de espessura e intensidade
- Pixels próximos a línea ideal estão “on”
- Rapidez de rasterização



Rasterização de linhas

- XRi e YRi: coordenadas de um ponto inicial Pi
- XRf e YRf: coordenadas de um ponto Pf, final
- Equação da reta que passa por Pi e Pf é dada por:

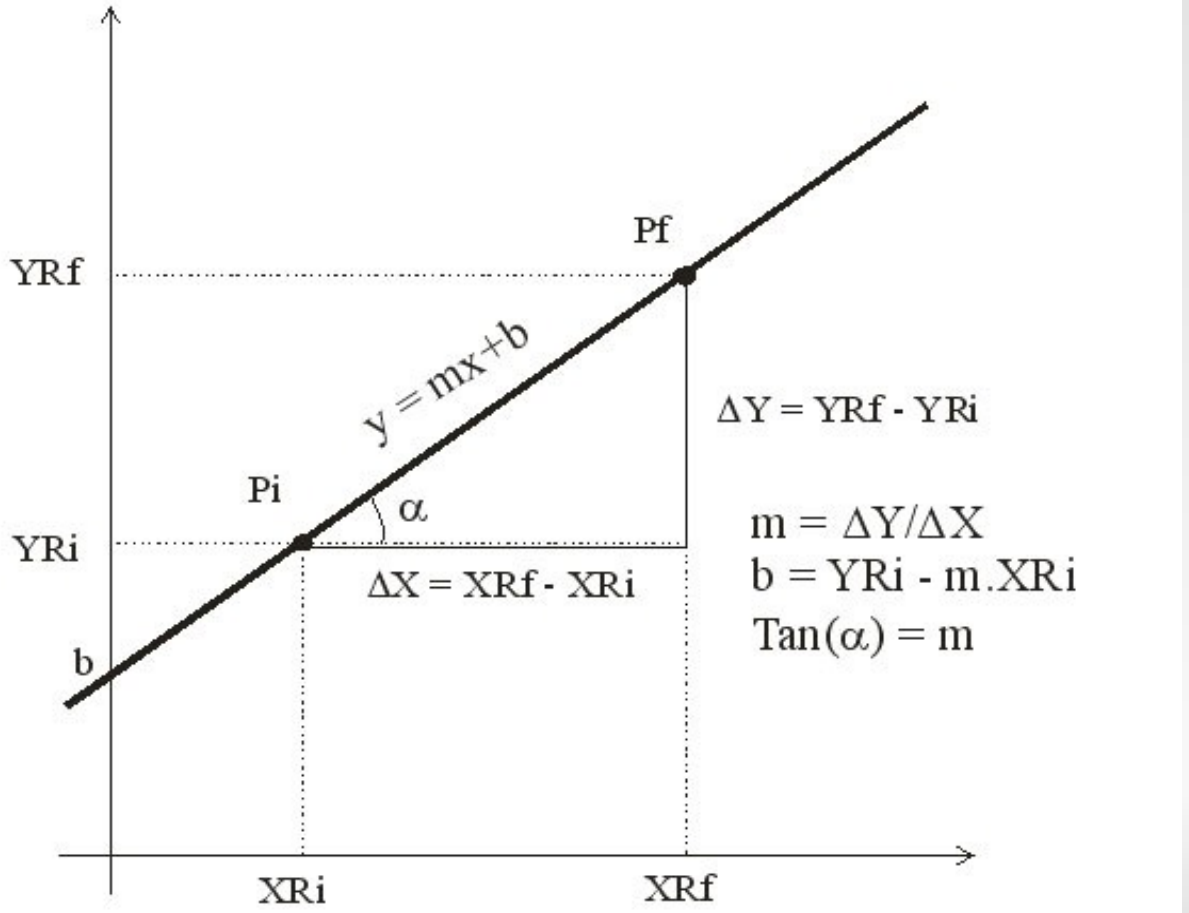


figura 1. Elementos da Reta que passa por dois pontos dados (Pi, Pf)

Rasterização de Linhas

- Java possui um método denominado

```
g.drawLine(int xi, int yi, int xf, int yf);
```

- Porém devemos aprender como isso acontece por dentro: portanto

```
drawLine(Graphics g, int xi, int yi, int xf, int yf){

}
```

Método Slope-Intercept

- De algebra: y = *a*x + *b*
  - *a* = slope *b* = y intercept Codificando:

```
……..
    for(int x= xi;x<xf;x++){
          y= (int) Math.round(a*x+b);
          putPixel(g,x,y);
    }
……..
```

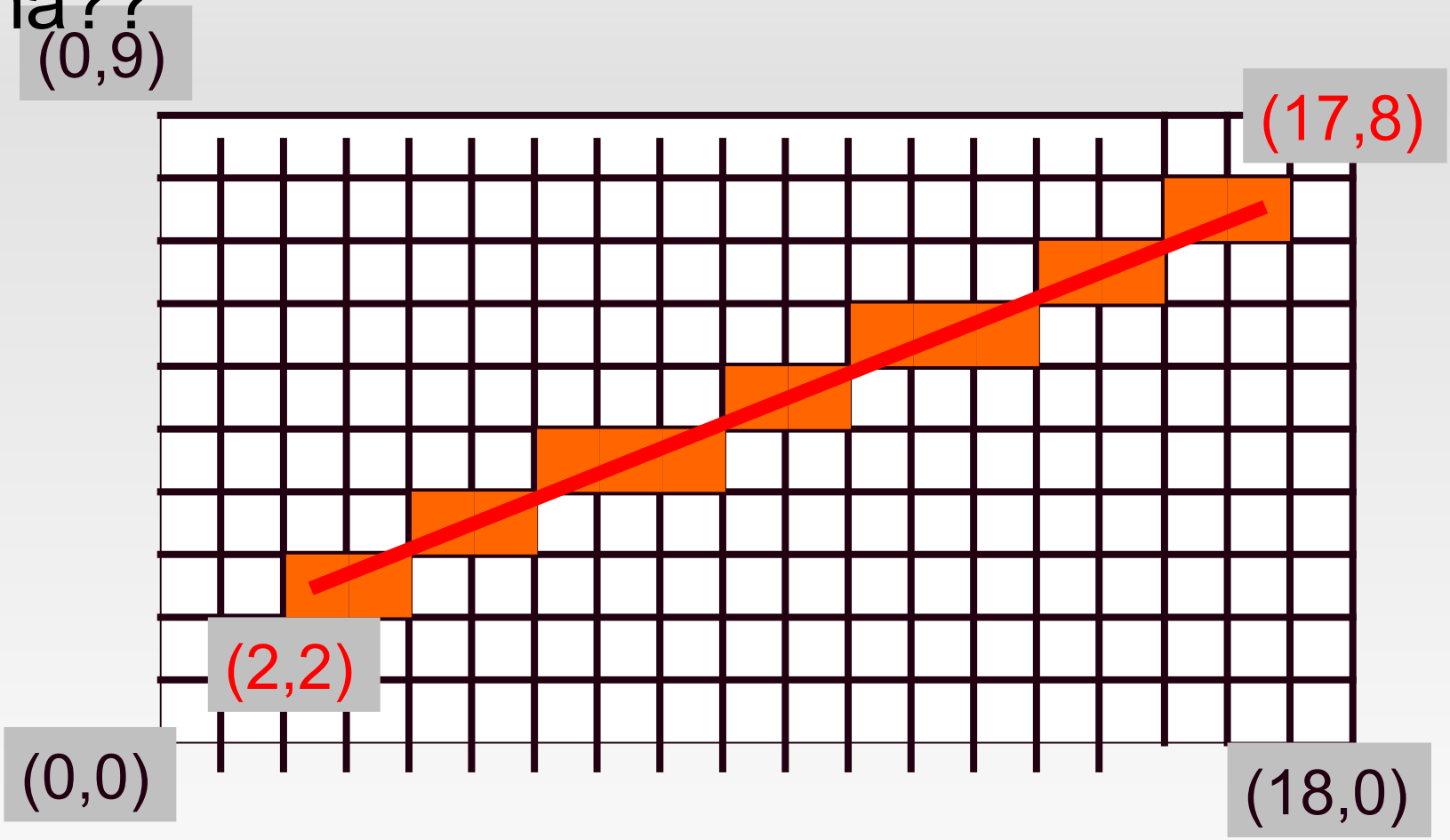
- Funciona??

Example 1:

Point1 V:(2,2)

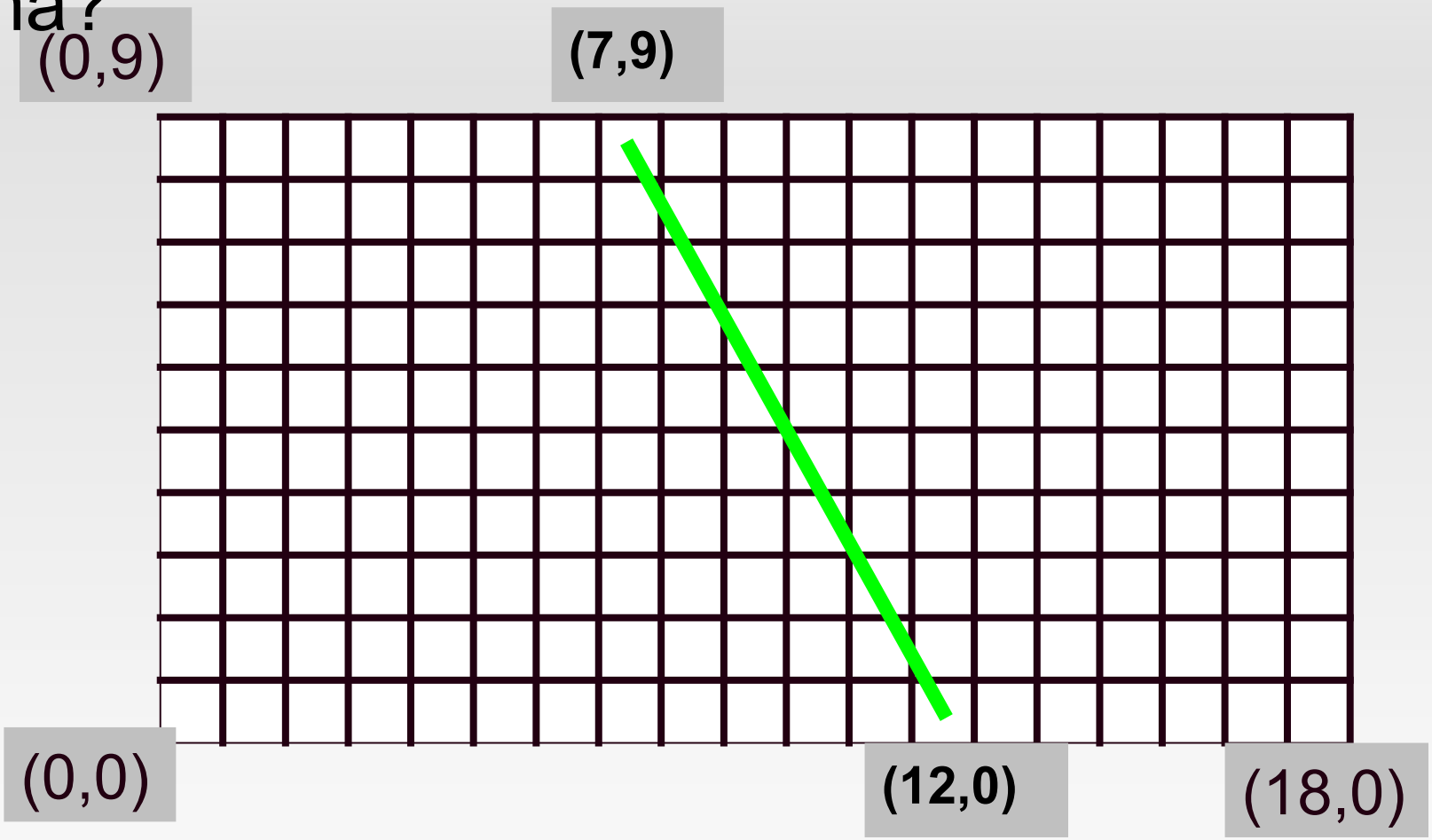
Point2 V:(17,8)

•Slope = 6/15 < 1



Example:

- Funciona?



Example 2:

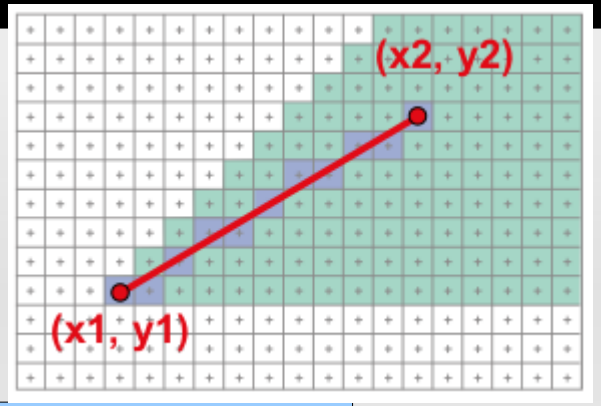
Point1 V:(7,9)

Point2 V:(12,0)

Problemas com pendente>1?

Método Slope-Intercept

- Podemos tratar os dois casos:
  - a > 1
  - a < 1

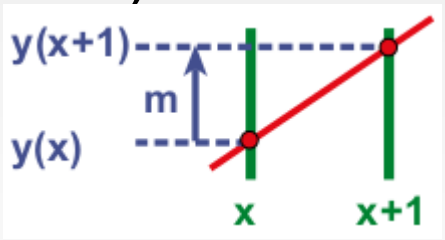



```
if(a<1){
        for(x= xi;x<xf;x++){
            y= (int) Math.round(a*x+b);
            putPixel(g,x,y);
        }
    }
    else{
        for(y = yi; y<yf; y++){
            x= (int) Math.round((y-b)/a);
            putPixel(g,x,y);
        }
    }
```

Método DDA

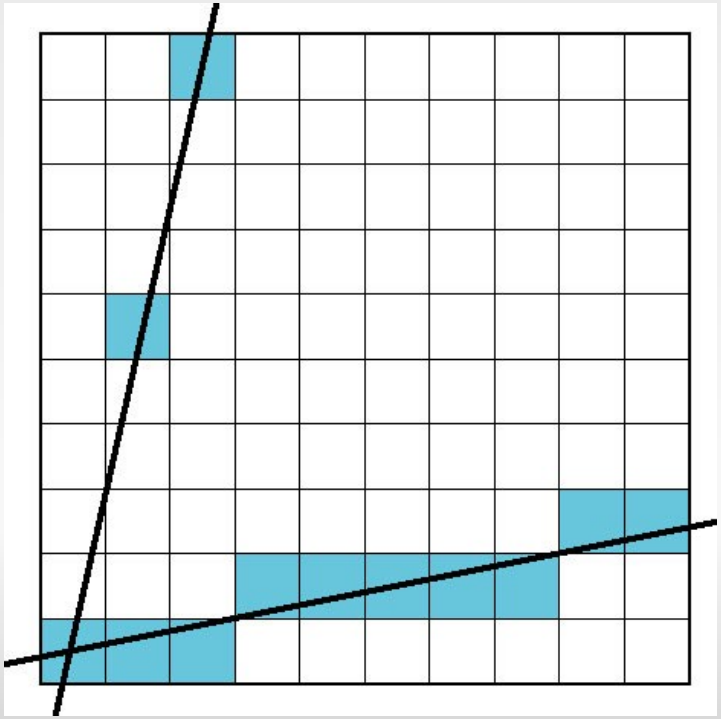
- Melhora o método anterior.
- DDA vem do ingles digital differential analyzer.
- Foca-se na equação: m = dy / dx
- É um algoritmo incremental. Ex: Para uma reta com inclinação<1, incrementamos y pela mesma quantidade a cada passo (incremento) de x.
- Podemos fazer uma operação a menos dentro do ciclo: reduzimos um produto em relação ao met. anterior (no lugar de fazer y=m*x+b fariamos y=yanterior+a).

```
for(int x= xi;x<xf;x++){
            y= y+a;
            putPixel(g,x,(int) Math.round(y));
}
```



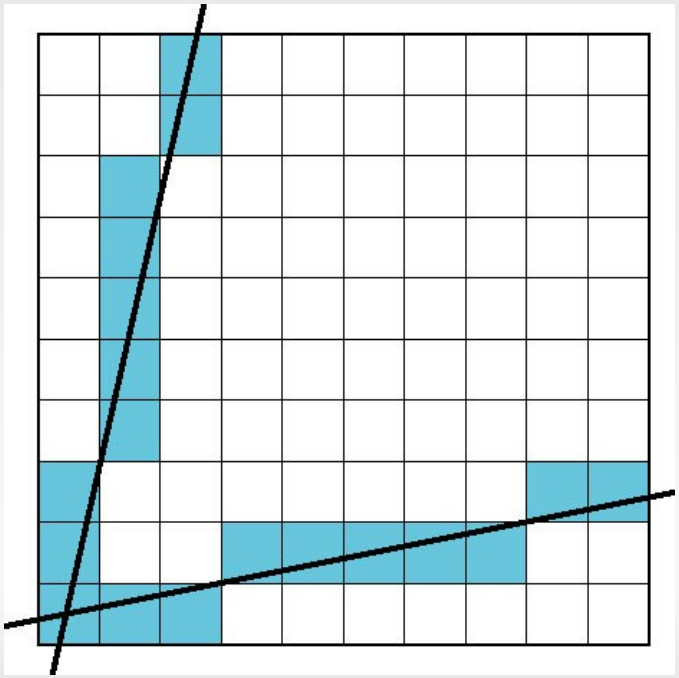
Problema

- DDA = Para cada x desenhar, um pixel no melhor y (mesmo problema que o método anterior).
- Problema com linhas que tem declividade elevada.



Usando simetria

- Para $1 \geq a \geq 0$, usar o código anterior
- Para a > 1, trocar de rol entre x e y:
  - Para cada y, desenhar o x adequado.



```
if(a<1){
        double y=yi;
        for(int x= xi;x<xf;x++){
            y= y+a;
            putPixel(g,x,(int) Math.round(y));
        }
    }else{
        double x=xi;
        for(int y= yi;y<yf;y++){
            x= x+(1/a);
            putPixel(g,(int)Math.round(x),y);
        }
    }
```

Método DDA

- Podemos geralizar o algoritmo para que funcione para os diversos casos em diversos quadrantes (só um bucle).

```
dx = x2 - x1
dy = y2 – y1

if (|dx| > |dy|)
        steps = |dx|
else
        steps = |dy|

xinc = dx/steps
yinc = dy/steps

x=x1
y=y1
for(k=0; k< steps; k++){
        putPixel(x, y)
        x += xinc;
        y += yinc;
}
```

Noções a tomar em conta

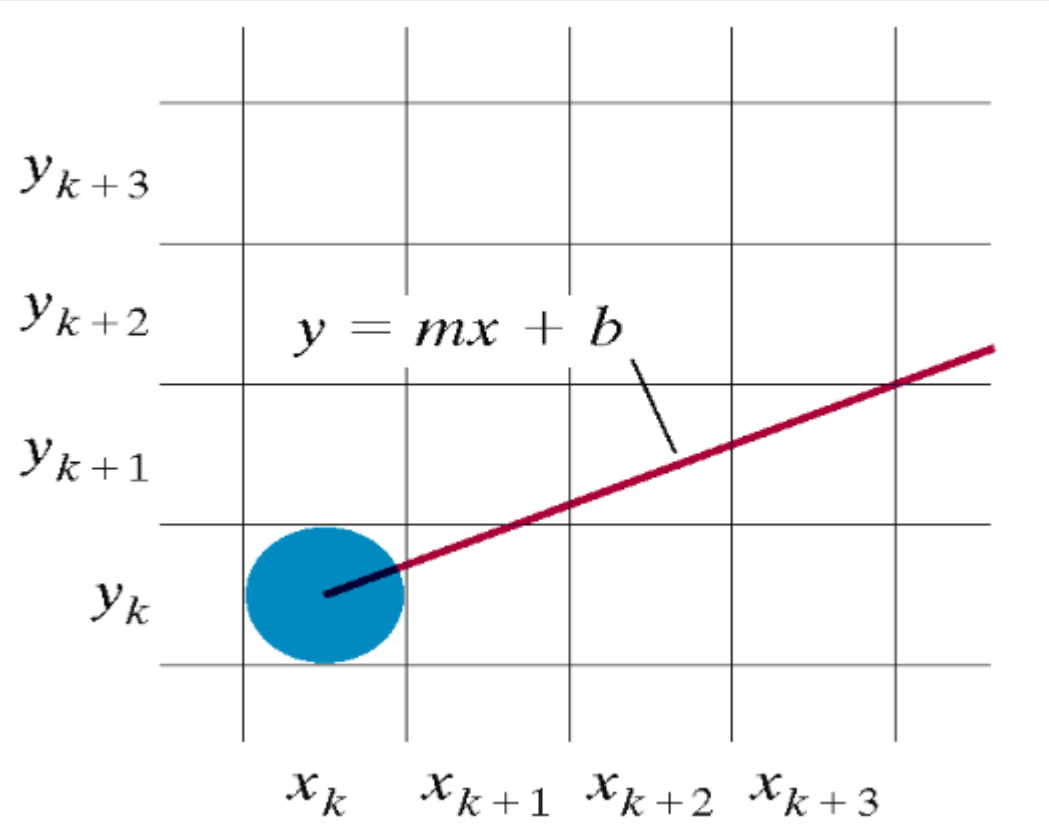
- *Em geral*:
  - *Adição e Subtração* são mais **rápidos** que a Multiplicação que é mais **rápida** que a Divisão.
  - Cálculos entre inteiros são mais rápidos que Floating point.
- Portanto, deseja-se converter todas as operações a operações entre inteiros.
- Como?

Brensenham

Algoritmo de Brensenham

- É um algoritmo eficiente (rápido) e preciso.
- Melhora o algoritmo DDA para usar somente **aritmética inteira**.
- Estende-se também para a rasterização de círculos.
- A cada passo determina o próximo melhor pixel baseado em quão próxima esta a verdadeira reta dos pixels possíveis.

Algoritmo de Brensenham

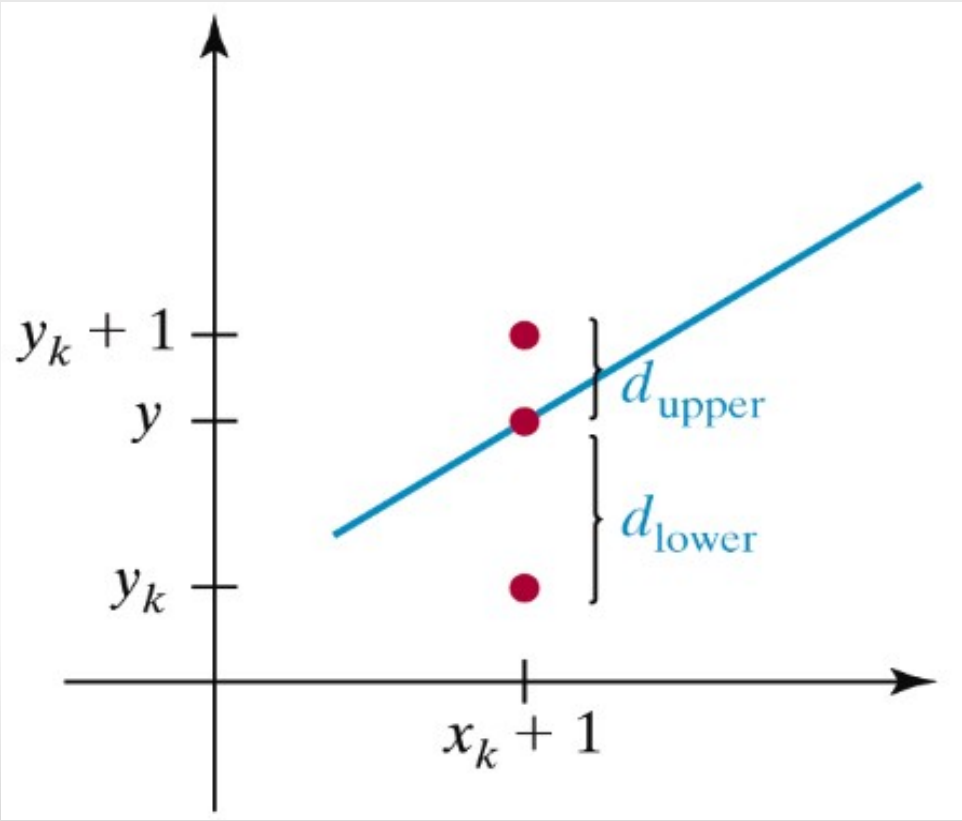


Qual é o próximo pixel ?

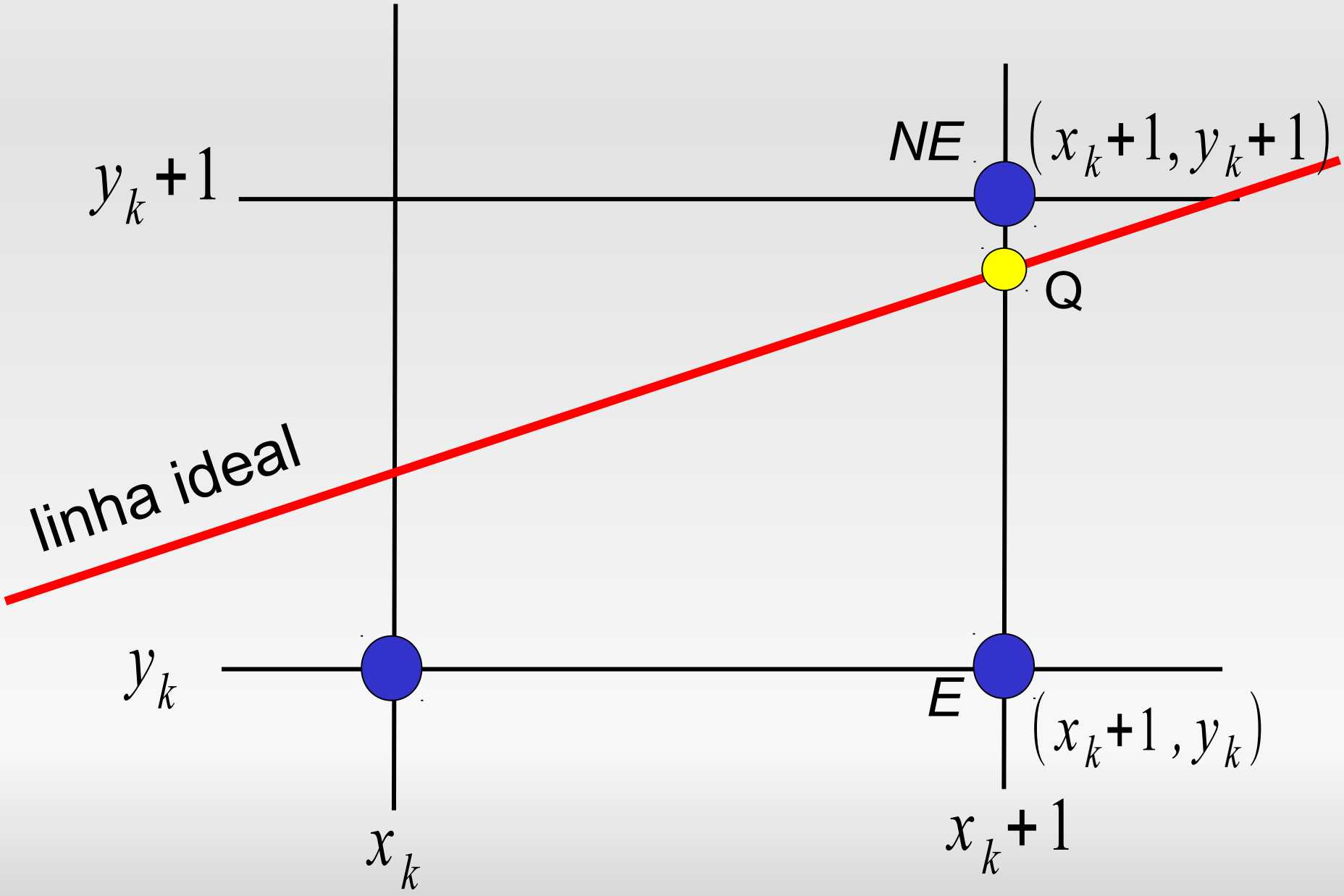
(x_{k+1}, y_k) ou (x_{k+1}, y_{k+1})

Algoritmo de Brensenham

- d (upper e lower) representam as distancias verticais entre as posições dos pixels e a reta y na posição x_k+1 do eixo x.
- Considerando que y_k, y_k+1, x_k+1 estão localizados no centro do pixels y_k, y_k+1, x_k+1 respectivamente.



Algoritmo de Brensenham



Algoritmo de Bresenham

- Consideraremos os casos base no primeiro octante com declividade m>0 e m<1.
- A desição pode se basear no signo (+ ou -) de:.

$$\mathbf{d_{lower} - d_{upper}}$$

$$\mathbf{d_{lower} - d_{upper}} \begin{cases} + \Rightarrow y_k + 1 \\ - \Rightarrow y_k \end{cases}$$

- **Objetivo:** Achar uma forma não cara computacionalmente para determinar o signo da expressão anterior.

Algoritmo de Bresenham

- $y = m(x_k + 1) + b$ (cálculo da coordenada y na linha x_k+1)
- $\mathbf{d_{lower} = y - y_k}$ $= m(x_k + 1) + b - y_k$
- $\mathbf{d_{upper} = (y_k + 1) - y}$ $= y_{k+1} - m(x_k + 1) - b$
- $\mathbf{d_{lower} - d_{upper}} = \mathbf{2m(x_k + 1) - 2y_k + 2b - 1}$

- Como m < 1 (no 1st octante) não é inteiro.
- **Substituimos m por dy/dx**, onde:
  - dy é va variação em y
  - dx é a variação em x
  - $\mathbf{d_{lower} - d_{upper} = 2(dy/dx)(x_k + 1) - 2y_k + 2b - 1}$
- Temos divisão mas podemos livrarmos dela.

Parametro de desição (p_k)

- Consideraremos um parametro de desição

$$\mathbf{p_k = dx\ (d_{lower} - d_{upper}\ )}$$

- O signo de p_k é o mesmo signo de $d_{lower} - d_{upper}$ posto que dx é positivo. Adicionamos dx com o intuito de tirar a divisão.

$$\mathbf{p_k} = dx\ (\ \ 2\ (\ dy / dx\ )(x_k + 1) - 2y_k + 2b\ - 1\ \ )$$

$$\mathbf{p_k = 2\ dy\ x_k - \ 2\ dx\ y_k\ +\ c}$$

Onde c é uma constante; $\mathbf{c = \ 2\ dy + 2\ dx\ b\ - dx}$

$$\mathbf{p_k} \begin{cases} + \ \Rightarrow \ y_k + 1 \\ - \ \Rightarrow \ y_k \end{cases}$$

Achando p_{k+1}

- $p_k = 2\ dy\ x_k - 2\ dx\ y_k + c$
- $\mathbf{p_{k+1}} = 2\ dy\ \mathbf{x_{k+1}} - 2\ dx\ \mathbf{y_{k+1}} + c$
- Subtraindo temos:
  - $\mathbf{p_{k+1} - p_k = 2dy\ (\ x_{k+1} - x_k\ ) - 2dx\ (\ y_{k+1} - y_k\ )}$
- Subtituindo $x_{k+1} = x_k + 1$ (no primeiro octante incrementamos a cada passo x em 1 unidade)
  - $\mathbf{p_{k+1} - p_k = 2dy - 2dx\ (\ y_{k+1} - y_k\ )}$
- $\mathbf{p_{k+1} = p_k + 2dy - 2dx\ (\ y_{k+1} - y_k\ )}$
- Onde ($y_{k+1} - y_k$) é 1 ou 0. (porque?) depende do signo de p_k .
- Concluindo:
  - $p_{k+1} = p_k + 2dy$ (when $p_k < 0 \rightarrow$ negativo)
  - $p_{k+1} = p_k + 2dy - 2dx$ (when $p_k >= 0 \rightarrow$ positivo)

Valor inicial da variable de desição

- O valor inicial de **p** é computado usando a equação para p usando (x_0, y_0)
- Obtendo o valor inicial:

  sabemos que $p_k = 2dyx_k - 2dxy_k + c$

  onde $c = 2dy + dx(2b-1)$

  e temos: $y_k = mx_k + b$, $b = y_k - mx_k$ e $m = dy/dx$

  $p_k = 2dyx_k - 2dxy_k + 2dy + dx(\ 2(y_k - (dy/dx)x_k) - 1\ )$

  Resolvendo:

  $p_k = 2dyx_k - 2dxy_k + 2dy + 2dxy_k - 2dyx_k - dx$

  $p_0 = 2dy - dx$

Algoritmo Bresenham

```
/*  Bresenham line-drawing procedure for |m| < 1.0.  */
   void lineBres (int x0, int y0, int xEnd, int yEnd)
   {
      int dx = fabs (xEnd - x0),  dy = fabs(yEnd - y0);
      int p = 2 * dy - dx;
      int twoDy = 2 * dy,  twoDyMinusDx = 2 * (dy - dx);
      int x, y;

      /* Determine which endpoint to use as start position */
      if (x0 > xEnd) {
         x = xEnd;
         y = yEnd;
         xEnd = x0;
      }
      else {
         x = x0;
         y = y0;
      }
```

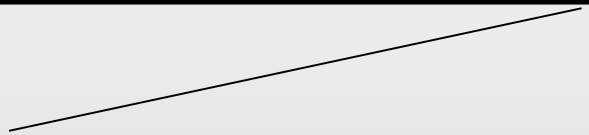
```
      setPixel (x, y);
      while (x < xEnd) {
            x++;
            if (p < 0)
               p += twoDy;
            else {
               y++;
               p += twoDyMinusDx;
            }
            setPixel (x, y);
         }
   }
/* From Hearn & Baker's Computer
Graphics with OpenGL, 3rd Edition */
```

Fabs → valor absoluto

Paradigma dos 4 Universos da Linha

Universo Físico

Uma linha consiste em ∞ points



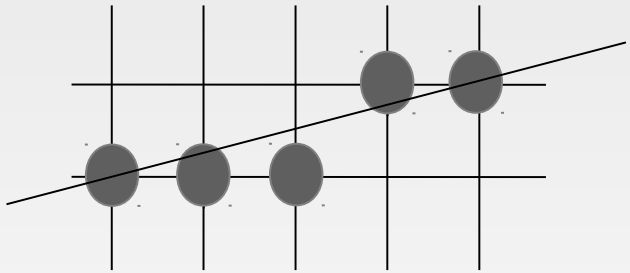
Universo Matemático

Existem diversas maneiras de se definir matematicamente uma linha!

- (x1,x2) (y1,y2)
- y = mx + b
- Equação implícita
- Equação paramétrica

Universo de Representação

A representação é feita com o uso de alguma definição matemática. Mas a discretização mais relevante ocorre mesmo no momento do raster.



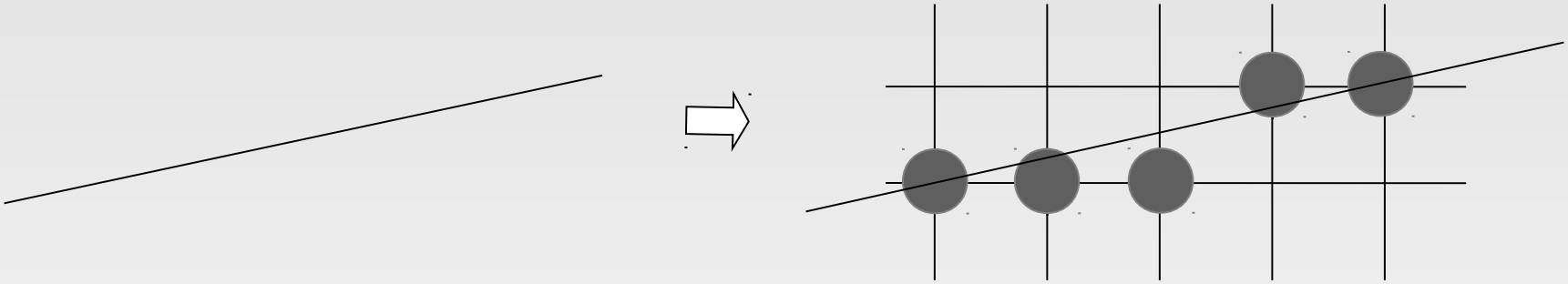
Universo de Implementação

- Variáveis para as definições matemáticas
- Algoritmos de rasterização

- int x1, x2, y1, y2;
- Bresenham

Amostragem, *Aliasing*, e *Anti-aliasing*

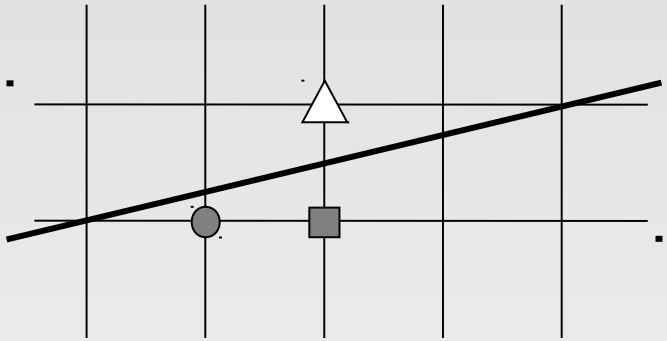
- A linha, que no universo físico é contínua, é amostrada em uma matriz finita 2D de pixels.
- Tal discretização pode causar distorções visuais como cisalhamento ou efeito de escada.



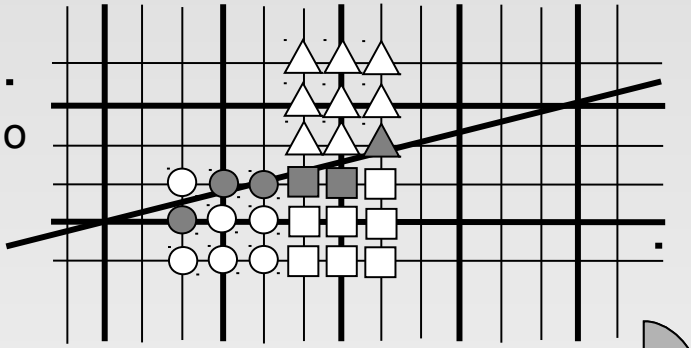
- Essas distorções são chamadas de *aliasing*.
- Para reduzir o problema de *aliasing*, usa-se uma técnica chamada *anti-aliasing*.
- A técnica consiste em uma superamostragem (uma vez que o aliasing é causada por uma subamostragem)

SUPERAMOSTRAGEM

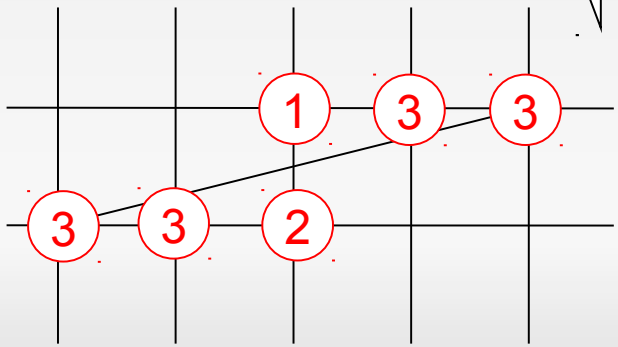
- Superamostragem = Amostrar um objeto numa resolução maior do que será reconstruído.



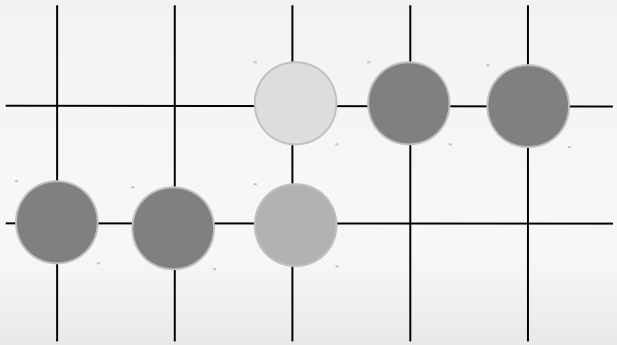
dividir os pixels em sub-pixels (i.e. 9), aplicar o algoritmo de Bresenham nesses sub-pixels



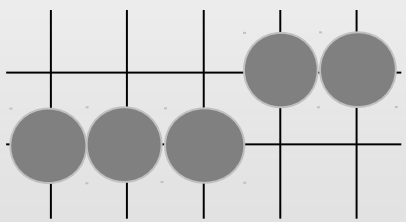
contar o número de sub-pixels “acesos” por pixel

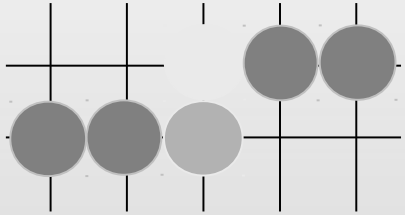


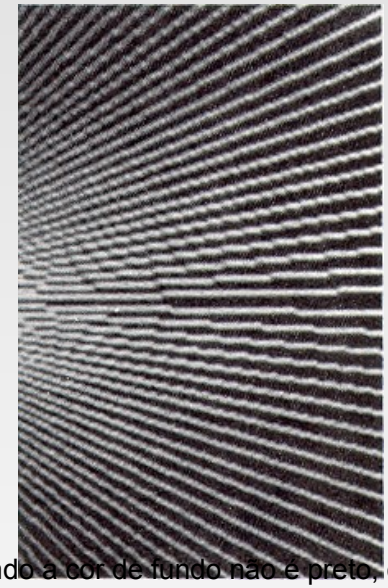
O pixel será aceso com intensidade proporcional ao número de sub-pixels acesos.

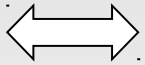


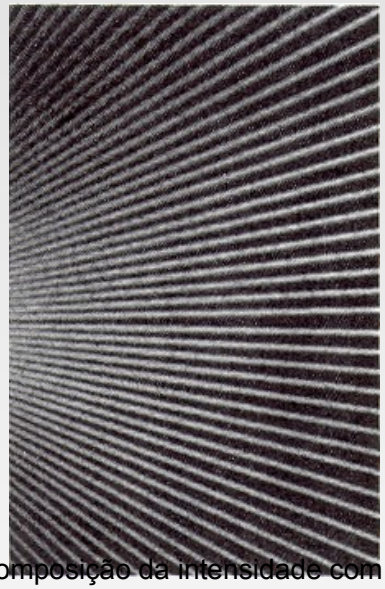
Exemplo de Anti-aliasing em Linhas









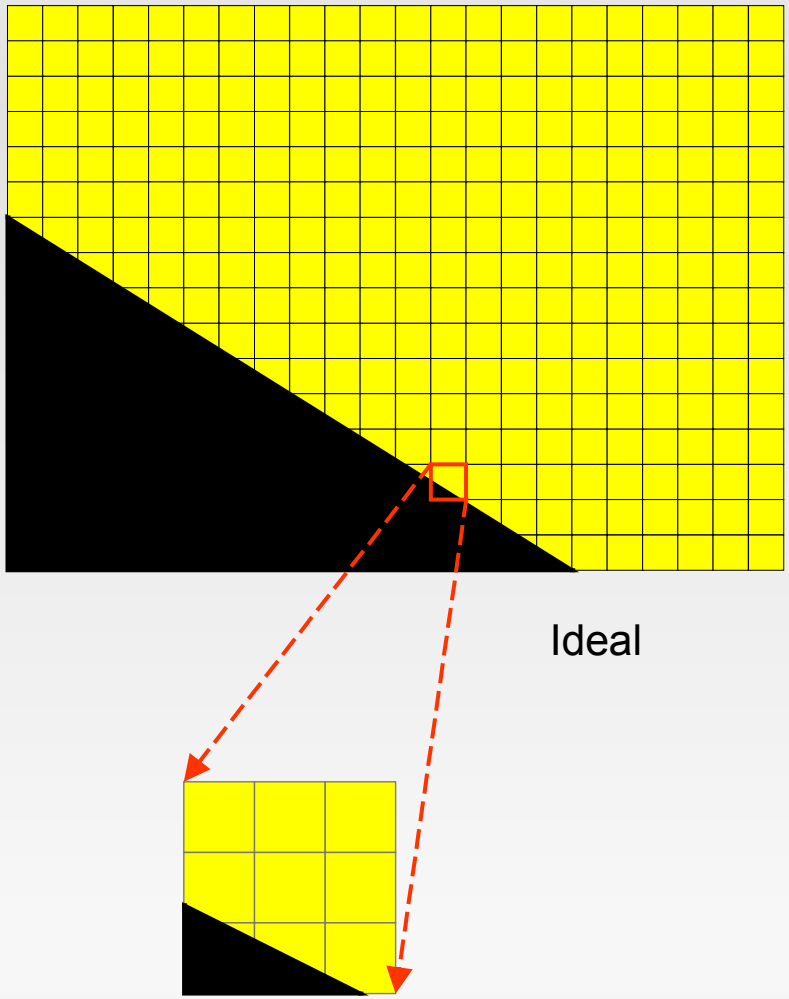


- Observe que quando a cor de fundo não é preto, o anti-aliasing deve fazer uma composição da intensidade com a cor de fundo.
- Anti-aliasing é necessário não só para linhas, mas também para polígonos e texturas (o que já é mais complicado)

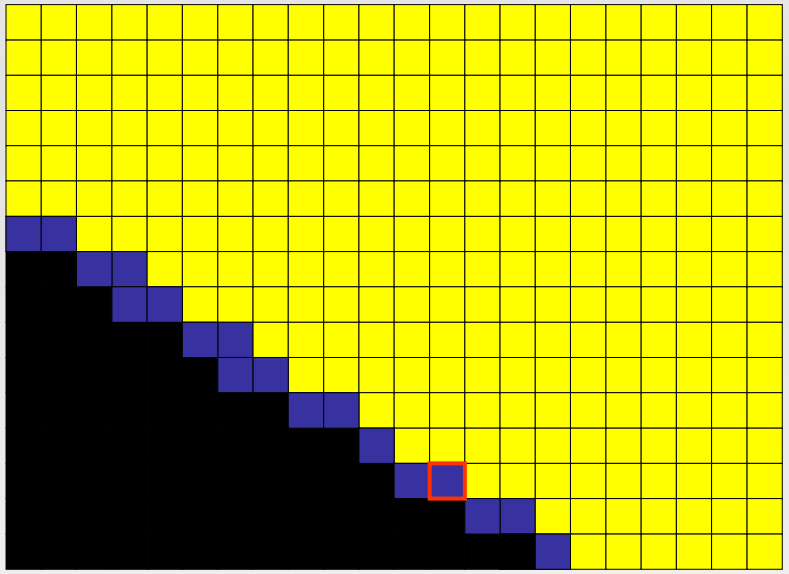
Observe que quando a cor de fundo não é preto, o anti-aliasing deve fazer uma composição da intensidade com a cor de fundo.

Anti-aliasing é necessário não só para linhas, mas também para polígonos e texturas (o que já é mais complicado)

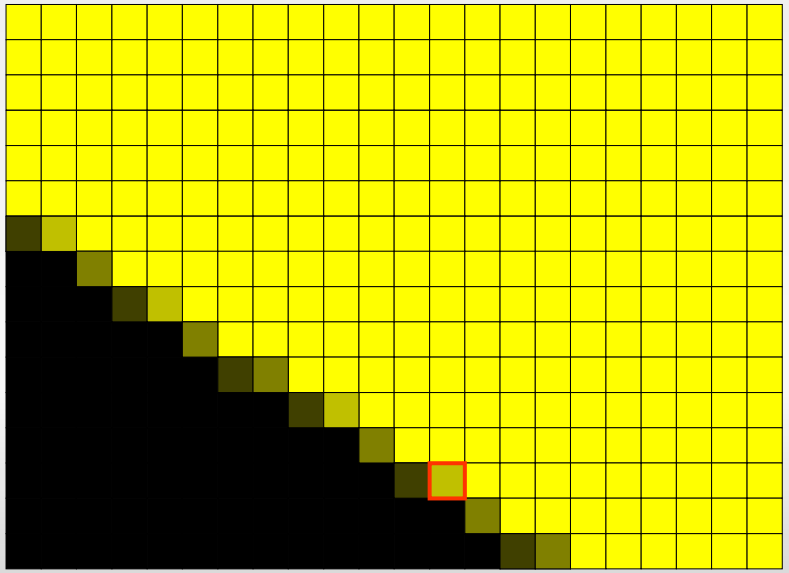
Antialiasing



Ideal



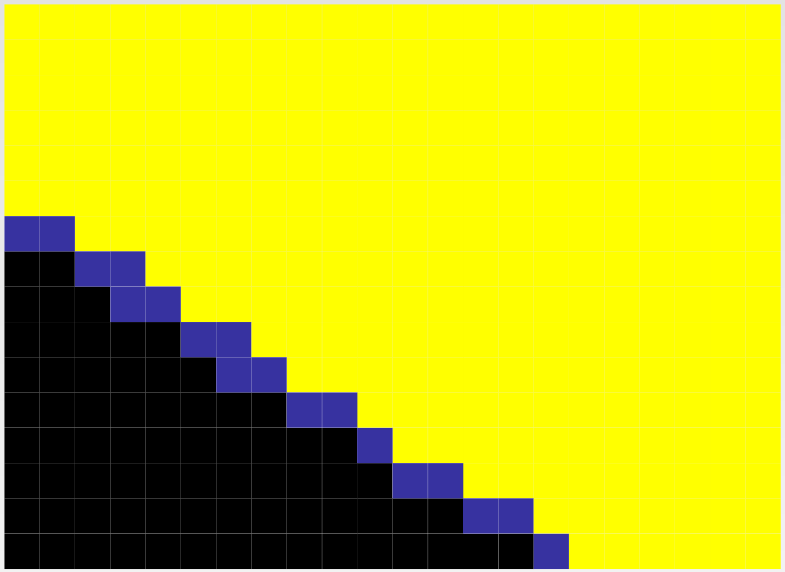
No Antialiasing



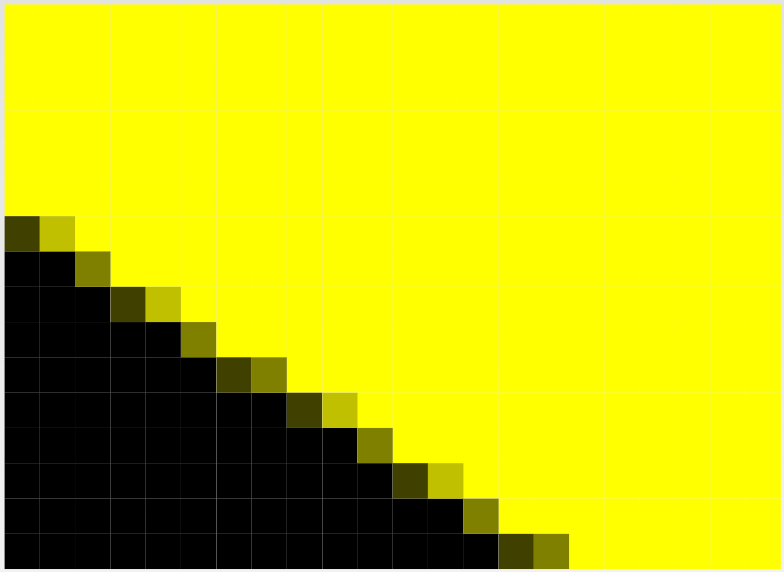
With Antialiasing

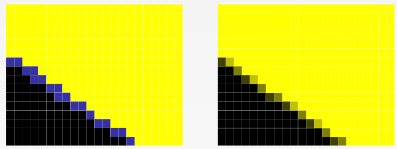
Antialiasing

No Antialiasing



With Antialiasing





Referências

- Computer Graphics with OpenGL, Hearn Baker, third edition
- Material do Prof. Ismael H F Santos, UniverCidade
- Computação gráfica para programadores Java, Ammeraal e Zhang, segunda edição.
- Computação Gráfica, Eduardo Azevedo e Aura Conci.
- Material Unip.
- Prof. Rodrigo Toledo. Computação Gráfica, PUC-Rio

- Nota: Algumas imagens e formulas da apresentação foram extraidas das fontes aqui apresentadas.