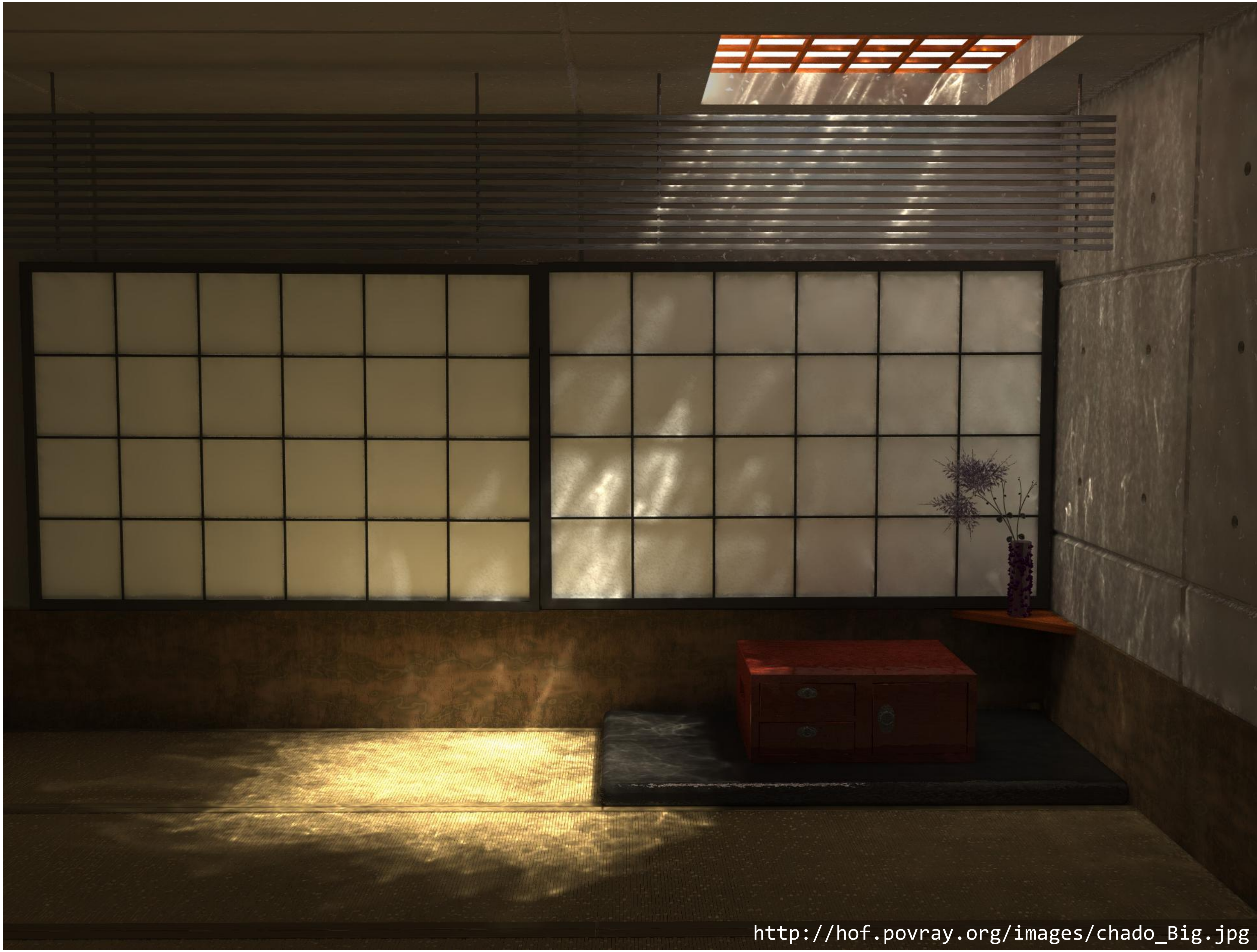
Rasterização de primitivas 2D

Computação Gráfica

Inverno 2011/2012


http://hof.povray.org/images/chado_Big.jpg

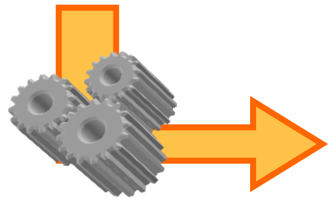
Sumário

- Enquadramento
- *Viewport vs window*
- Rasterização de primitivas gráficas
  - Linhas
  - Círculos

Enquadramentos (1)

Mundo virtual

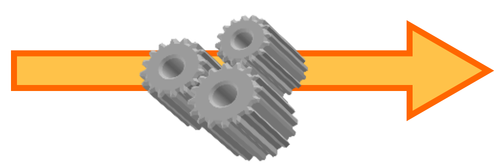
- Para que o modelo do mundo virtual seja mostrado ao utilizador, é necessário calcular, sobre ele, a vista a apresentar
- Quais são as unidades no mundo?
  - Metros, polegadas, etc. – **Coordenadas do mundo**



as coordenadas têm que ser mapeadas para coordenadas do ecrã (ou melhor, **coordenadas do dispositivo**)

- Como vimos, este processamento é apenas um dos muitos feitos no *pipeline* de processamento
- Muitas vezes, ao mesmo tempo que se realiza esta transformação, também se efectua o *clipping*

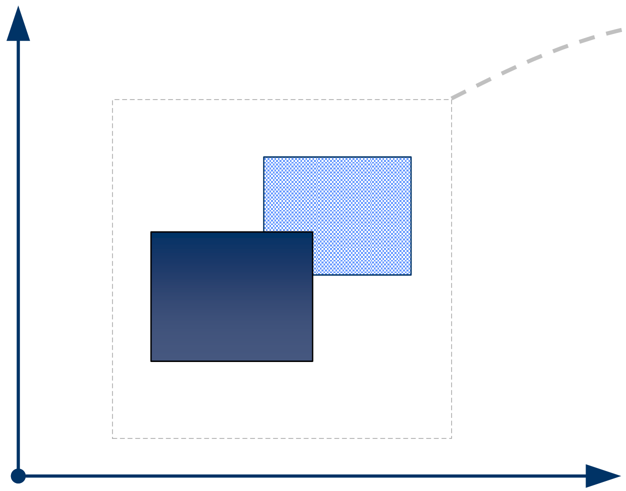
Definir a janela **no mundo** (origem)



Definir a janela **no dispositivo de saída** (destino)

Enquadramentos (2)

Window vs Viewport

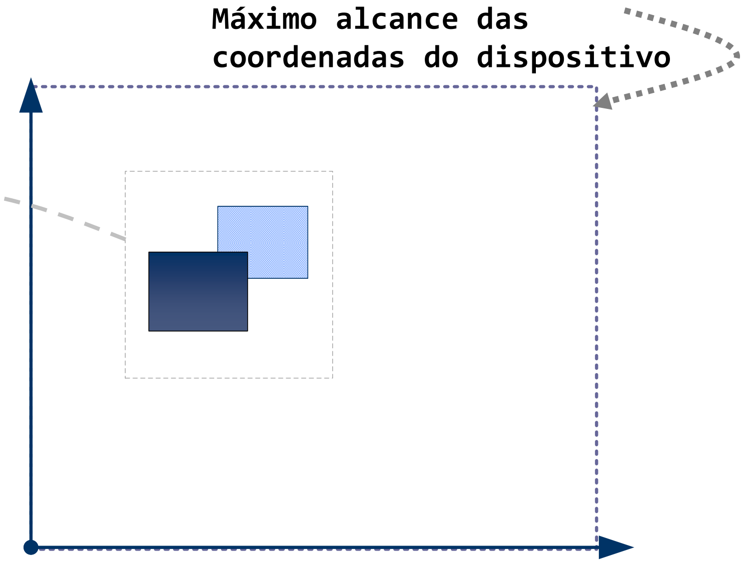


Coordenadas do mundo - WC

Window

A região rectangular, <u>definida sobre o mundo</u>, que representa a zona a apresentar ao utilizador

Máximo alcance das coordenadas do dispositivo



Viewport

A região rectangular, <u>definida sobre o dispositivo</u>, onde será visualizada a janela do mundo

Enquadramentos (3)

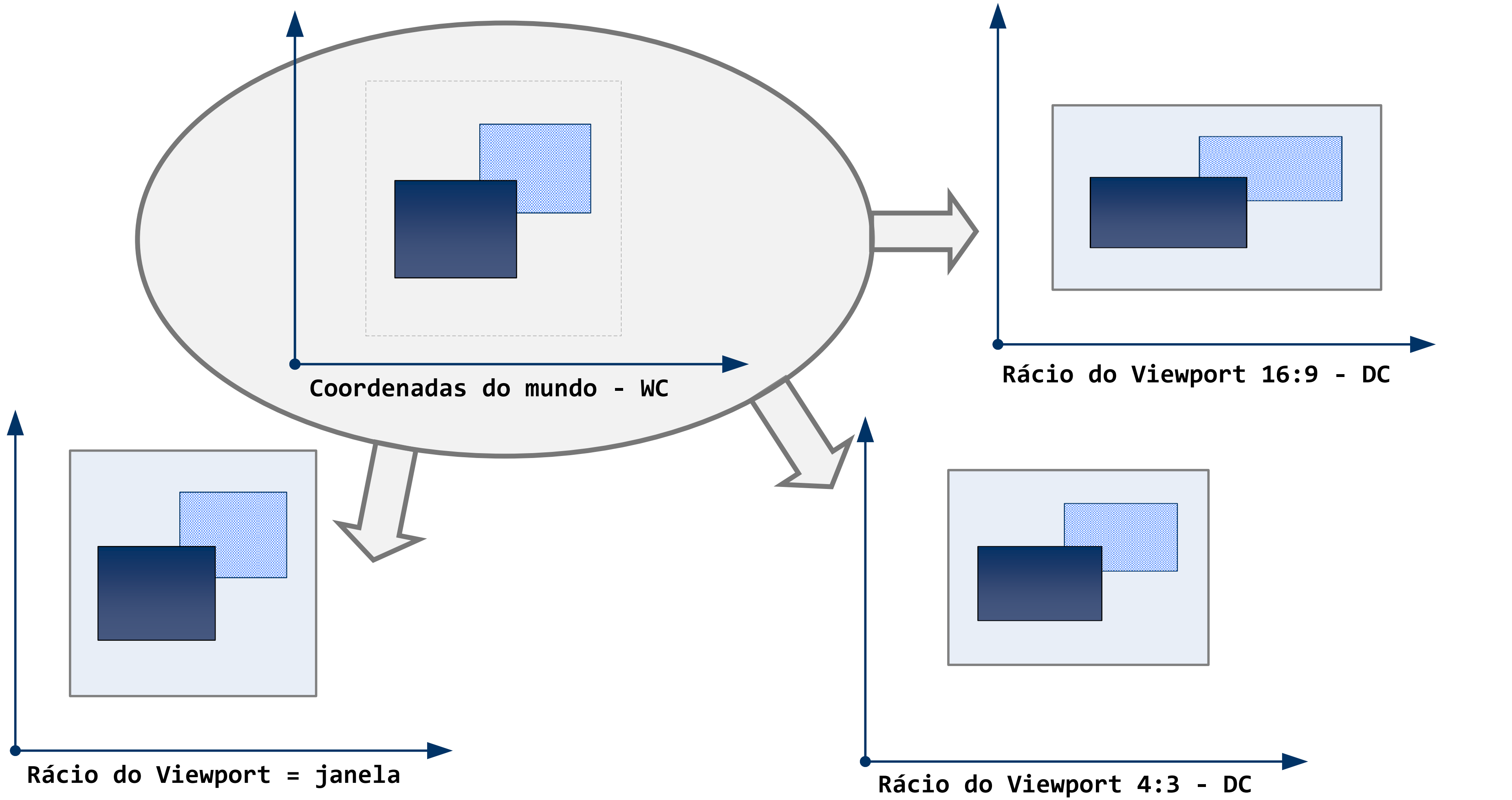
- Note-se que o conceito de janela não é o aquele a que estamos habituados

- Não existe afinidade entre a **janela** rectangular que define uma área no mundo e a **janela** utilizada pelos gestores de janelas popularizados pelo *Windows OS* e *Mac OS*

- Poder-se-á dizer que, uma vez que o conceito de gestor de janelas é mais recente, e dado que o conceito de janela tem já um semântica bem definida em computação gráfica, o nome deveria ter sido outro - *viewports*

Enquadramentos (4)

- Note-se que os *viewports* podem não ter o mesmo rácio que a janela



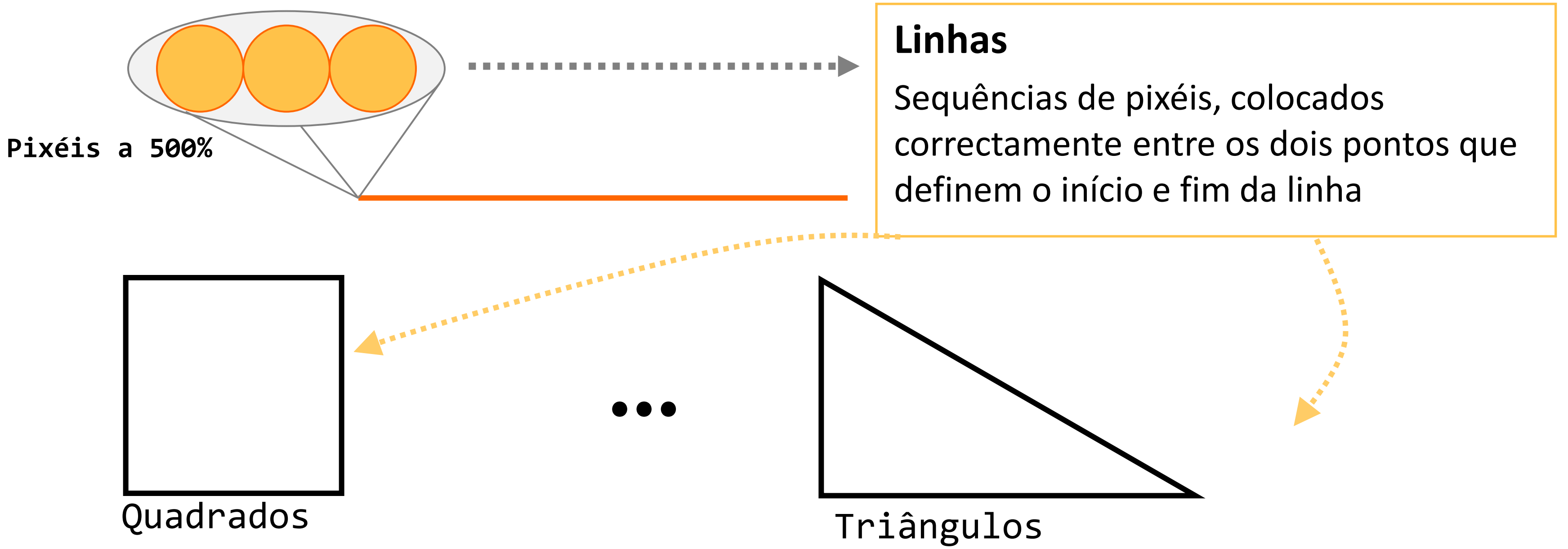
Sumário

- Enquadramento
- *Viewport vs window*
- Rasterização de primitivas gráficas
  - Linhas
  - Círculos

Primitivas de rasterização

- Já foi referido que todos os objectos têm que ser representados numa matriz de pixéis, de forma a serem mostrados pelo dispositivo *raster* (neste caso num monitor)

- **Q:** Como fazer isto? Como representar objectos complexos? Quais as formas envolvidas?

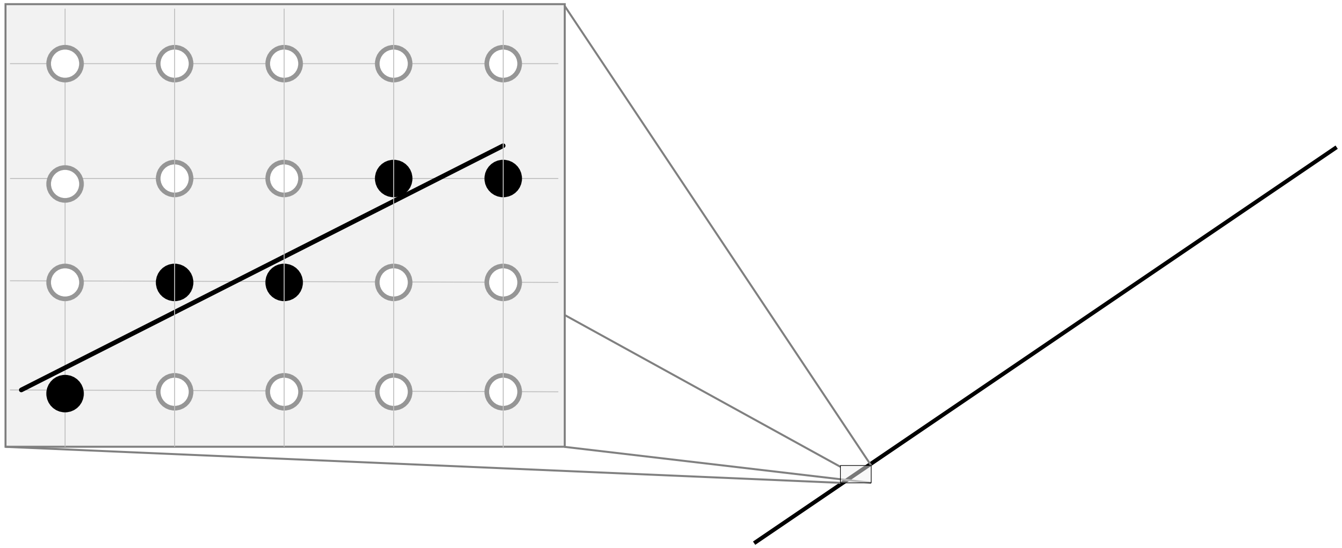


Sumário

- Enquadramento
- *Viewport vs window*
- Rasterização de primitivas gráficas
  - Linhas
  - Círculos

Rasterização de linhas

- Os dispositivos raster não conseguem representar uma recta perfeita – apenas uma aproximação
  - **Motivo:** É necessário passar de um **meio contínuo** para um **meio discreto**



- É então necessário encontrar um algoritmo que consiga uma boa aproximação a um segmento de recta entre dois pontos

Recta – um pouco de matemática (1)

- Matematicamente falando, uma linha (em português) designa-se por **recta**.
- Definições:

  “A geometrical object that is straight, infinitely long and infinitely thin.”

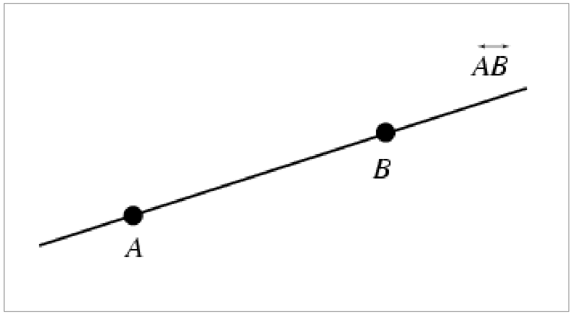
  http://mathopenref.com/line.html

  “The infinite extension in both directions of a line segment, which is the path of shortest distance between two points.”

  http://mathworld.wolfram.com/Line.html

- Equações da recta

Forma explícita	y = mx + B
Forma implícita	Ax + By + C = 0
Forma paramétrica	r(t) = o + td

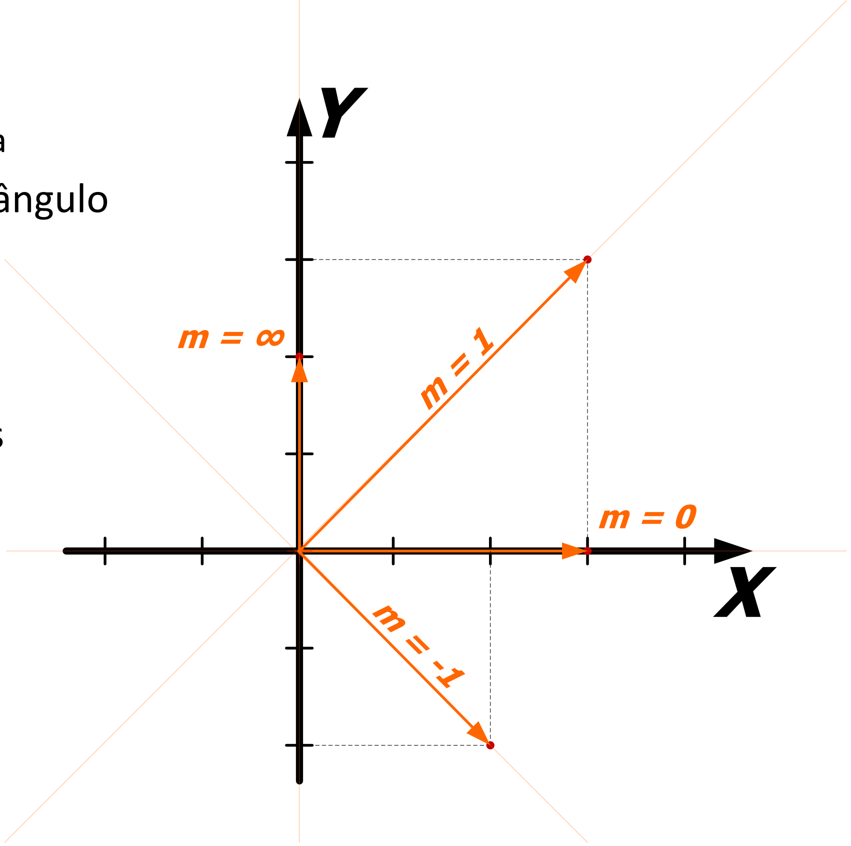


Ver: http://www.cut-the-knot.org/Curriculum/Calculus/StraightLine.shtml

Recta – um pouco de matemática (2)

$$y = mx + B$$

- Qual é o significado do declive (m) e do valor B na **forma explícita** da equação da recta?
  - m
    - representa a inclinação
      - (ângulo em relação ao eixo do XX) da recta
    - Não cresce de forma linear em relação ao ângulo que faz com o eixo do XX
    - Toma valor 0 para rectas horizontais
    - Toma valor 1 para inclinações de 45º
    - Toma valor infinito (∞) para rectas verticais
  - B
    - Local onde a recta intercepta o eixo do YY



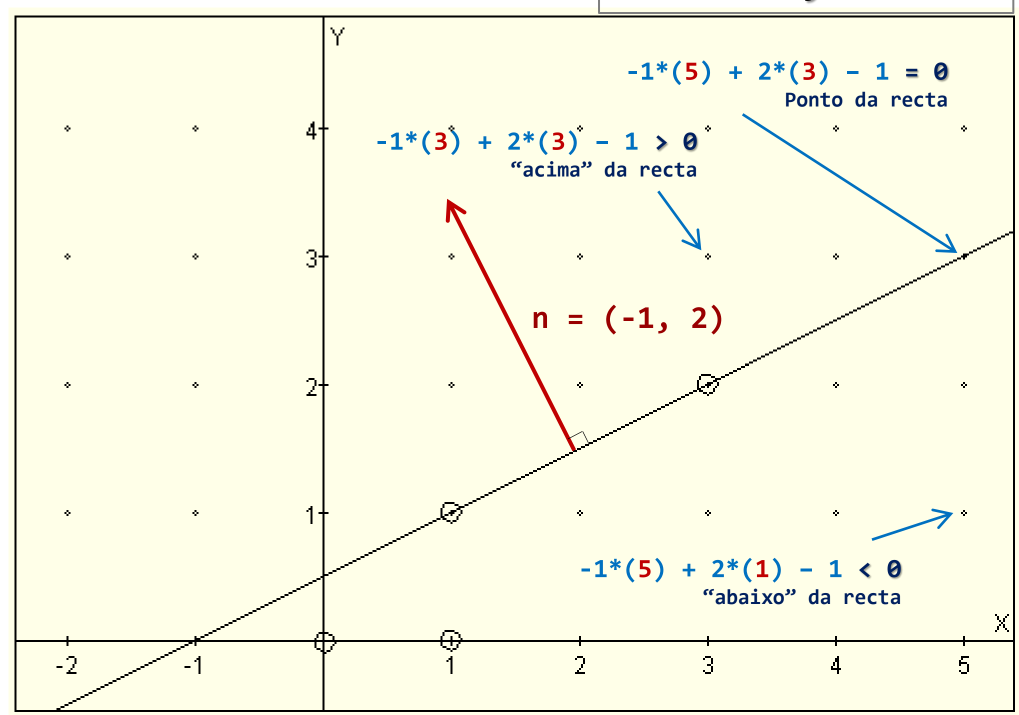
Nota: Esta equação **não suporta** a definição de rectas verticais

Recta – um pouco de matemática (3)

$$Ax + By + C = 0$$

- Para que serve a **forma implícita** da equação da recta? Que significado têm os valores **A**, **B** e **C**?
  - A e B
    - Componentes do vector **n=(A,B)** que é normal à recta
  - C
    - Menor distância à origem (escala em relação à normal **n**)

$$-1x + 2y - 1 = 0$$



- Esta forma é útil para verificar se determinado ponto
  - **pertence** à recta;
  - se está **acima** da recta (*direcção da normal*);
  - ou **abaixo** da recta.

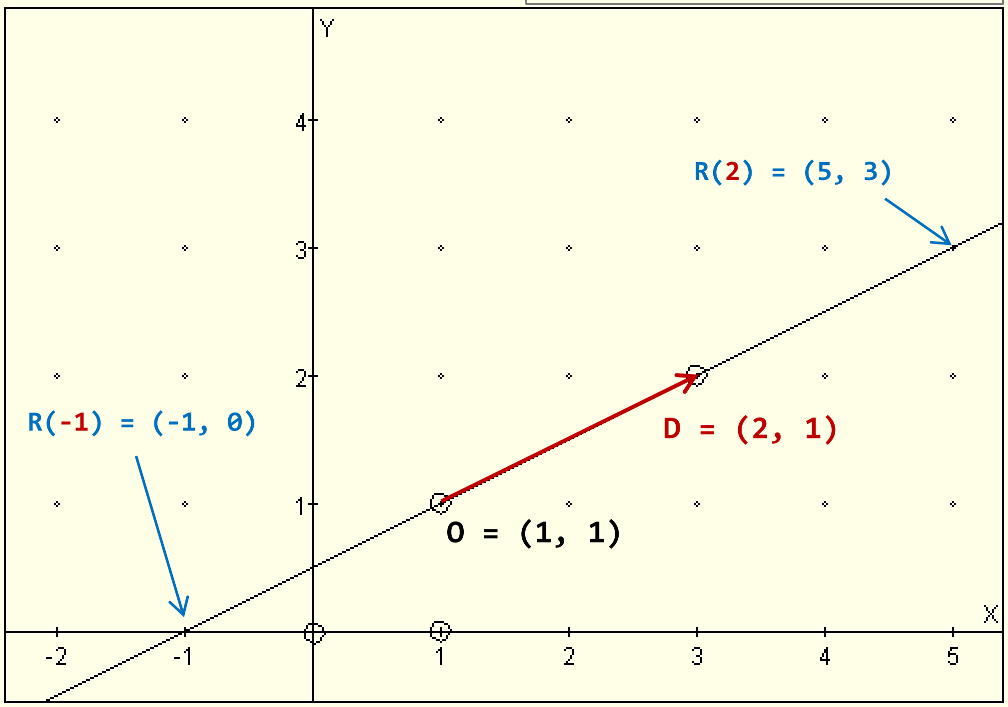
Nota: Esta equação **suporta** a definição de rectas verticais

Recta – um pouco de matemática (4)

`R(t) = O + tD`

- Para que serve a **forma paramétrica** da equação da recta? Que significado têm os valores **O**, **D** e **`t`**?
  - `O:` Ponto (O de origem) da recta
  - `D:` Direcção da recta
  - `t:` Variável "geradora" de pontos na recta que passa pelo ponto O e tem direcção D.

`R(t) = (1,1) + t(1,2)`



- Esta forma é útil para (p.ex):
  - Cálculo de intersecções
  - Cálculo da posição de um jogador que anda em linha recta a determinada velocidade.

Nota: Esta equação **suporta** a definição de **todas** as rectas

Primitivas de rasterização – linhas (1)

Algoritmo incremental

- Este algoritmo calcula, de forma incremental, os pixéis mais próximos da recta real
  - Recebe como entrada as coordenadas `(x,y)` de dois pontos da recta
  - Apenas desenha o segmento de recta definido pelos dois pontos (finito)

- Primeiro, calcula-se o declive da recta

$$m = \frac{\Delta y}{\Delta x}$$

- Partindo do ponto inicial, (x_i, y_i), onde `i=0`, incrementa-se a coordenada em X, um pixel de cada vez

$$x_{i+1} = x_i + 1$$

- O valor de Y calcula-se através da forma explícita da equação da recta

```
y = mx + B
```

Primitivas de rasterização – linhas (2)

Algoritmo incremental

- Dado que a coordenada em X é incrementada de uma unidade, não existe problema em lidar com ela na matriz *raster*

- Já Y, depende do **resultado da multiplicação entre dois reais** – `m` e x_i.
- É necessário proceder a um acerto no valor de Y de forma a que passe para o pixel que melhor representa o seu valor real

$$P_i = \left(x_i, round(y_i)\right) \qquad round(y_i) = floor(0.5 + y_i)$$

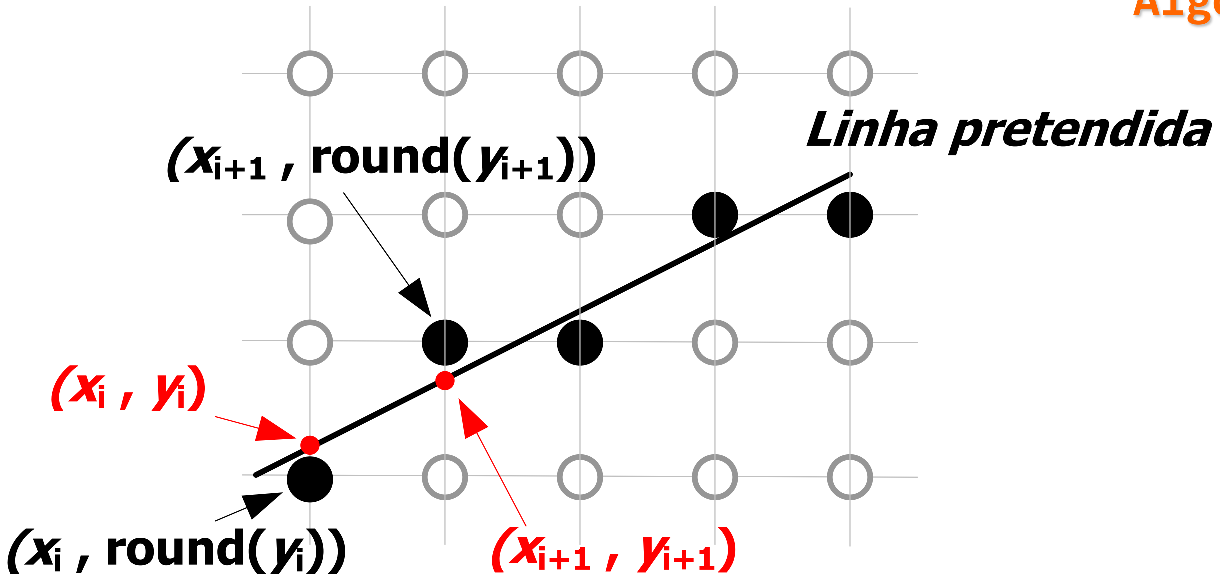
- De forma a optimizar os cálculos, pode ser feita uma simplificação tendo em atenção que

$$y_{i+1} = mx_{i+1} + B = m(x_i + \Delta x) + B = y_i + m\Delta x$$

- Sabendo que $\Delta x = 1$ e $y = mx + B$, então $y_{i+1} = y_i + m$

Primitivas de rasterização – linhas (3)

Algoritmo incremental



- Note que se $|m| \geq 1$, o incremento em y será maior que 1. Neste caso baste inverter os papeis de x e y

$$\boxed{y_{i+1} = y_i + 1} \quad \text{e} \quad \boxed{x_{i+1} = x_i + 1/m}$$

Primitivas de rasterização – linhas (4)

Algoritmo incremental
pseudo algoritmo

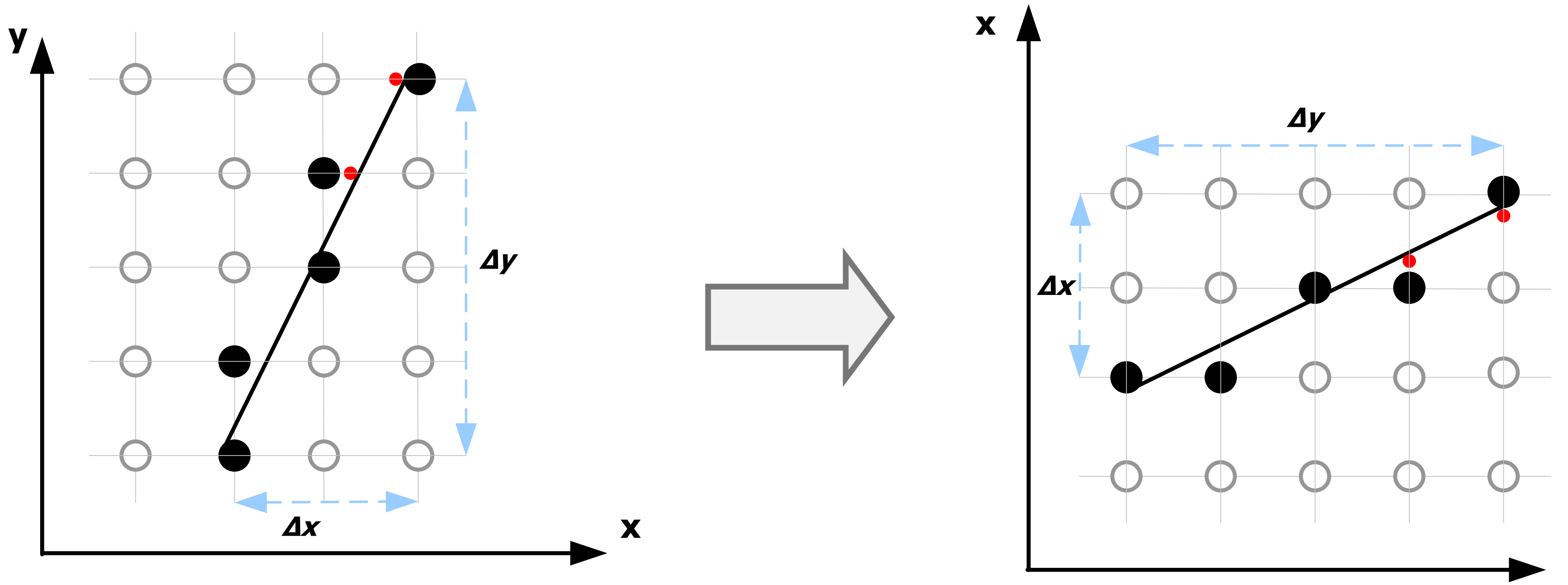
```
void drawline(Ponto ini, Ponto fim)
//admitindo |m| < 1 e sem casos especiais
{              //        Δy                 Δx
  double m  = (fim.Y - ini.Y) / (fim.X - ini.X);
  double xi = ini.X
  double yi = Y;
  while(xi < fim.X)
  {
     ++xi;
     yi = yi + m;
     setPixel(xi, Math.floor(0.5 + yi));
  }
}
```

Primitivas de rasterização – linhas (5)

Algoritmo incremental

- Este algoritmo apresenta alguns problemas, nomeadamente quando o declive é superior a 1

- A resolução deste problema passa por tirar partido de simetrias existentes



y é actualizado de **m**

x é actualizado de **1/m**

Primitivas de rasterização – linhas (6)

Algoritmo incremental

- Este algoritmo, bastante simples, tem alguns pontos menos bons, nomeadamente

  - Por cada cíclo, é necessário fazer uma **conversão de double para inteiro** – problemas de desempenho

  - Embora as coordenadas de um *pixel* sejam inteiras, Y necessita de ser um número de vírgula flutuante pois **`m` é uma fracção**

Objectivo

Encontrar um algoritmo que use apenas aritmética de inteiros

Motivação → Aumento de desempenho

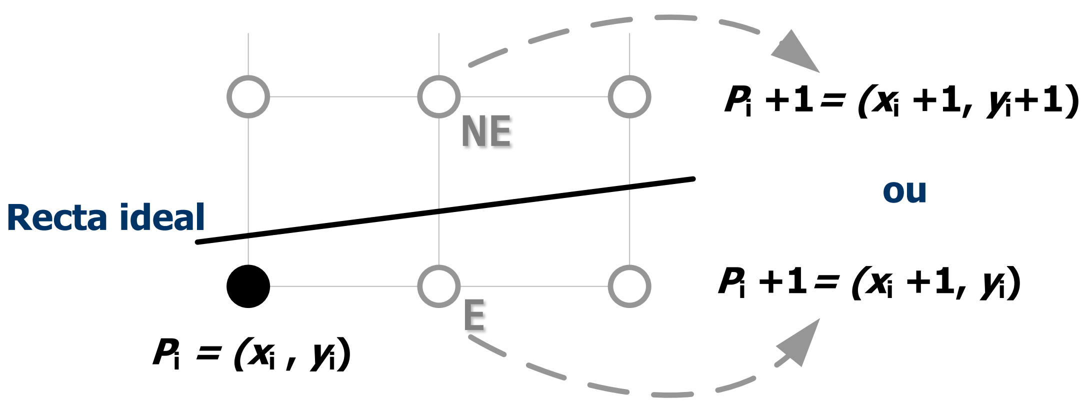
Primitivas de rasterização – linhas (7)

Algoritmo de Bresenham

- Este algoritmo foi proposto por Bresenham, **utiliza apenas aritmética de inteiros**, e vai incrementalmente calculando os *pixeis* que melhor se aproxima à recta ideal

- O autor demonstrou que o algoritmo minimiza o erro relativamente à recta ideal, cujo valor absoluto é sempre ≤ ½

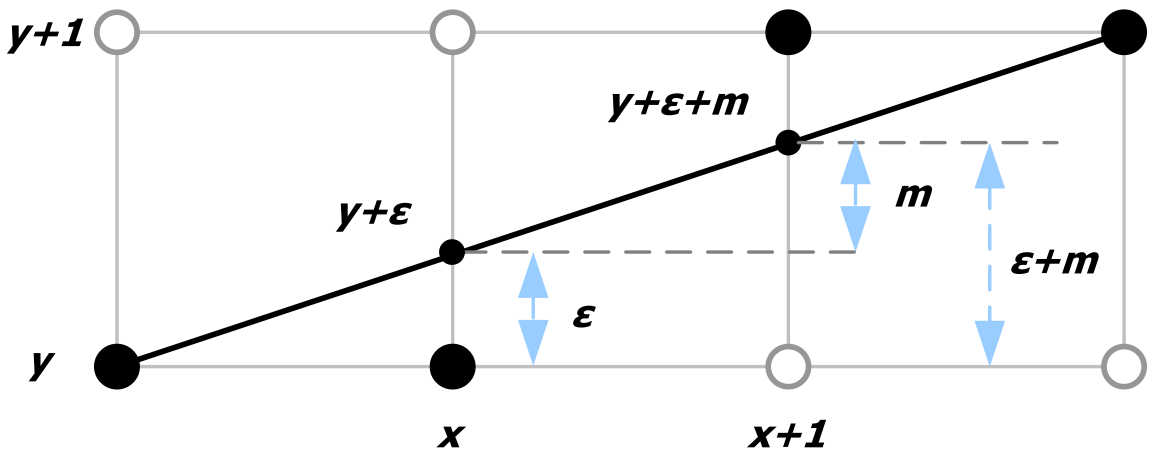
- A decisão de qual o *pixel* a iluminar resume-se a escolha um de dois possíveis **E** ou **NE**, **escolhendo-se aquele que minimizar o erro**



Primitivas de rasterização – linhas (8)

Algoritmo de Bresenham

- A figura seguinte mostra as várias variáveis em jogo e o seu significado



- Nas condições anteriores, escolhe-se o ponto **E** **(x+1,y)** se

$$\boxed{y+\varepsilon+m < y+0,5}$$

- Caso contrário, escolhe-se **NE** **(x+1,y+1)**

Primitivas de rasterização – linhas (9)

Algoritmo de Bresenham

- Para que o algoritmo seja incremental, é necessário **actualizar o valor do erro, de acordo com a decisão anterior**, ou seja

Se **E** **(x+1,y)**, então $\boxed{\varepsilon = (y + \varepsilon + m) - y}$

Se **NE** **(x+1,y+1)**, então $\boxed{\varepsilon = (y + \varepsilon + m) - (y + 1)}$

- No entanto o teste $\boxed{\varepsilon + m < 0{,}5}$ utiliza **aritmética de virgula flutuante**

- Podemos efectuar uma manipulação algébrica de forma a passar para **aritmética de inteiros**

Primitivas de rasterização – linhas (10)

Algoritmo de Bresenham

- Se multiplicarmos ambos os lados por Δx e depois por 2 teremos

$$\boxed{\begin{array}{l} \varepsilon + m < 0{,}5 \\ \Delta x \times (\varepsilon + m) < \Delta x \times 0{,}5 \\ 2\Delta x \times (\varepsilon + m) < \Delta x \end{array}}$$

- Sabendo que $\boxed{m = \dfrac{\Delta y}{\Delta x}}$ fica $\boxed{2(\varepsilon \Delta x + \Delta y) < \Delta x}$

- Note-se que uma multiplicação por 2 pode ser **implementada com um simples *shift* à esquerda**…

Primitivas de rasterização – linhas (11)

Algoritmo de Bresenham

- Continua a ser necessária aritmética de vírgula flutuante na actualização do erro

$$\boxed{\varepsilon = \varepsilon + m} \quad \text{e} \quad \boxed{\varepsilon = \varepsilon + m - 1}$$

- No entanto, multiplicando ambos os lados por Δx, temos

$$\begin{aligned} \varepsilon\Delta x &= \varepsilon\Delta x + \Delta y \\ \varepsilon\Delta x &= \varepsilon\Delta x + \Delta y - \Delta x \end{aligned}$$

- A notação pode ser simplificada, efectuando a substituição

$$\boxed{\varepsilon' = \varepsilon\Delta x} \quad \text{fica} \quad \boxed{\varepsilon' = \varepsilon' + \Delta y} \quad \text{e} \quad \boxed{\varepsilon' = \varepsilon' + \Delta y - \Delta x}$$

Primitivas de rasterização – linhas (12)

Algoritmo de Bresenham

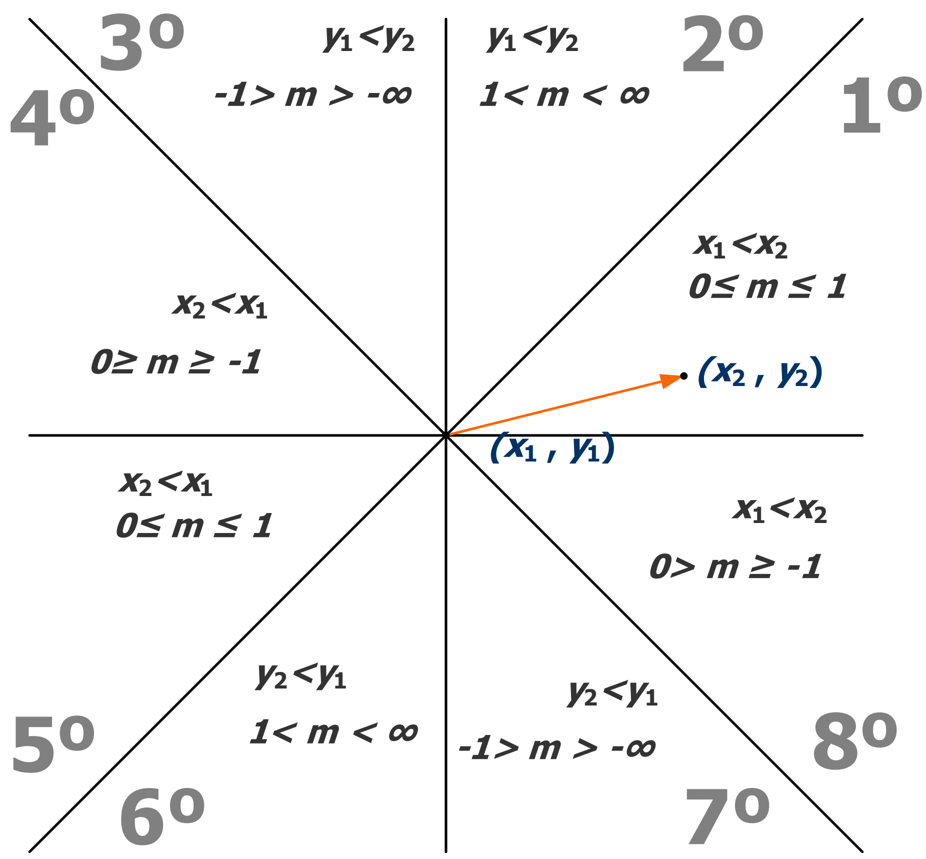
```
//Para o primeiro octante (0 <= m <= 1)
void drawline(Ponto ini,Ponto fim)
{
  int e  = 0;
  int dx = fim.x - ini.x,
  int dy = fim.y - ini.y;
  int y = ini.y;
  for(int x = ini.x; x < fim.x; ++x) {
    setPixel(x, y);
    e += dy;
    if(2*e >= dx) {
      ++y;
      e -= dx;
   }
  }
}
```

Primitivas de rasterização – linhas (13)

Algoritmo de Bresenham

- **Atenção:** o algoritmo apresentado apenas funciona para

$\boxed{0 \leq m \leq 1}$ e $\boxed{ini.x < fim.x}$



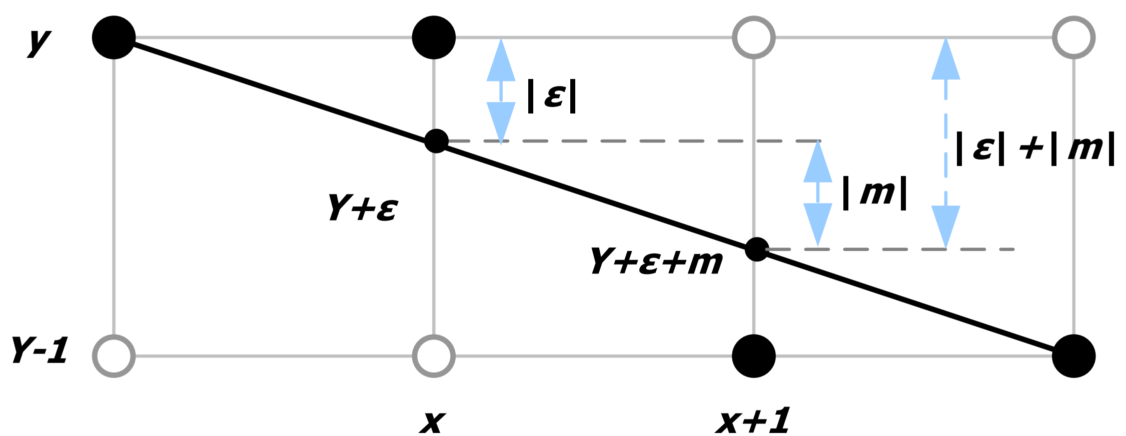
- Os restantes casos são apresentados na figura

Primitivas de rasterização – linhas (14)

- Quando a direcção se altera, para que o algoritmo funcione basta trocar os pontos

- Quando o declive é superior a 1, basta trocar as variáveis de iteração, por exemplo, iterar em y em vez de x

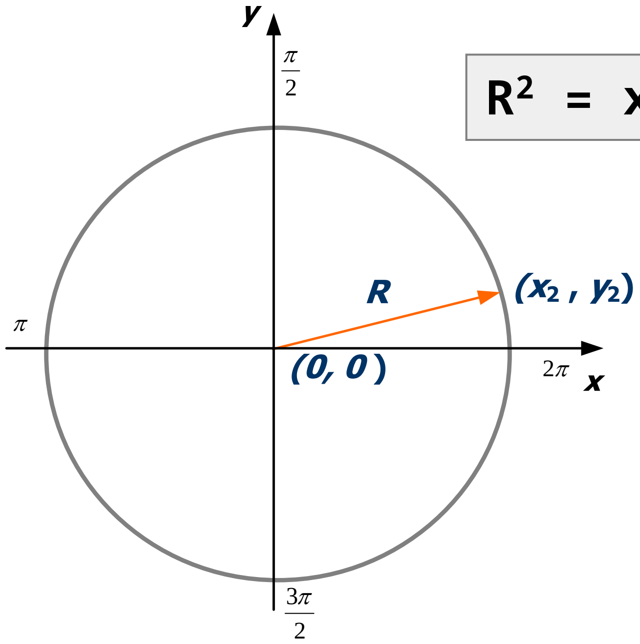
- Para declives negativos é necessário deduzir as fórmulas, de acordo com a figura seguinte



Sumário

- Enquadramento
- *Viewport vs window*
- Rasterização de primitivas gráficas
  - Linhas
  - Círculos

Primitivas de rasterização – círculos (1)



$$R^2 = x^2 + y^2$$

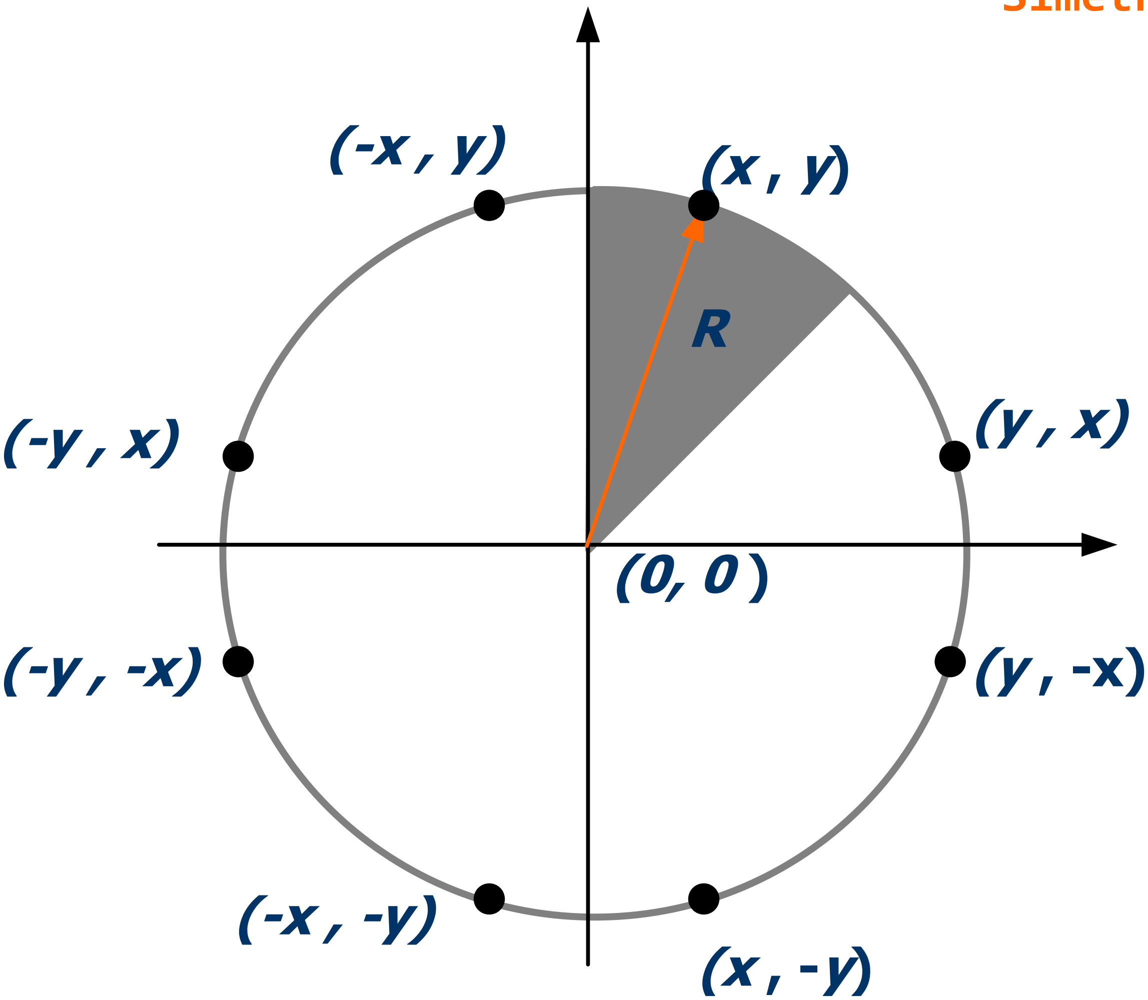
Com $y = f(x)$ fica $y = \pm\sqrt{R^2 - x^2}$

- Como o cálculo dos pontos na circunferência é ineficiente utilizando directamente a fórmula anterior, é necessário encontrar um algoritmo, que, à semelhança do que se fez para a recta, tente minimizar as operações com vírgula flutuante

- E claro, utilizar a natureza **simétrica** do círculo

Primitivas de rasterização – círculos (2)

Simetria do círculo



Primitivas de rasterização – círculos (3)

Simetria do círculo

- Tirando partido da simetria dois 8 octantes, traçar um círculo resume-se a calcular os pontos do primeiro octante e, para cada um evocar a seguinte função

```
//Para um círculo centrado na origem
void writeCirclePixel(int x, int y)
{
   writePixel( x,  y);
   writePixel( y,  x);
   writePixel( y, -x);
   writePixel( x, -y);
   writePixel(-x, -y);
   writePixel(-y, -x);
   writePixel(-y,  x);
   writePixel(-x,  y);
}
```

Primitivas de rasterização – círculos (4)

Algoritmo Midpoint

- Bresenham desenvolveu um algoritmo incremental para traçar círculos, que, à semelhança do seu algoritmo para linhas, traça os pontos **garantindo a minimização do erro**

- O algoritmo aqui apresentado é baseado nesse, mas utiliza o critério do ponto médio

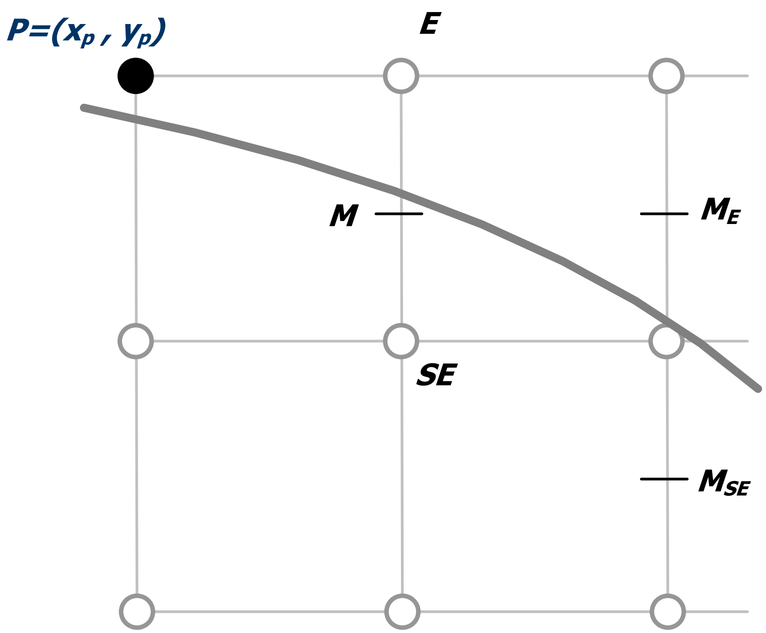
- Tem como vantagem ser mais facilmente generalizável para cónicas em geral

- O algoritmo considera apenas 45º do circulo, no 2º octante onde

$$\boxed{0 \leq x \leq R/\sqrt{2}}$$

Primitivas de rasterização – círculos (5)

Algoritmo Midpoint



- Tendo o ponto **P** sido escolhido, numa iteração anterior, como sendo o mais próximo do círculo, o próximo ponto a escolhe será **E** (*East*) ou o **SE** (*South-East*), avaliando a função no ponto médio entre os dois - **M**

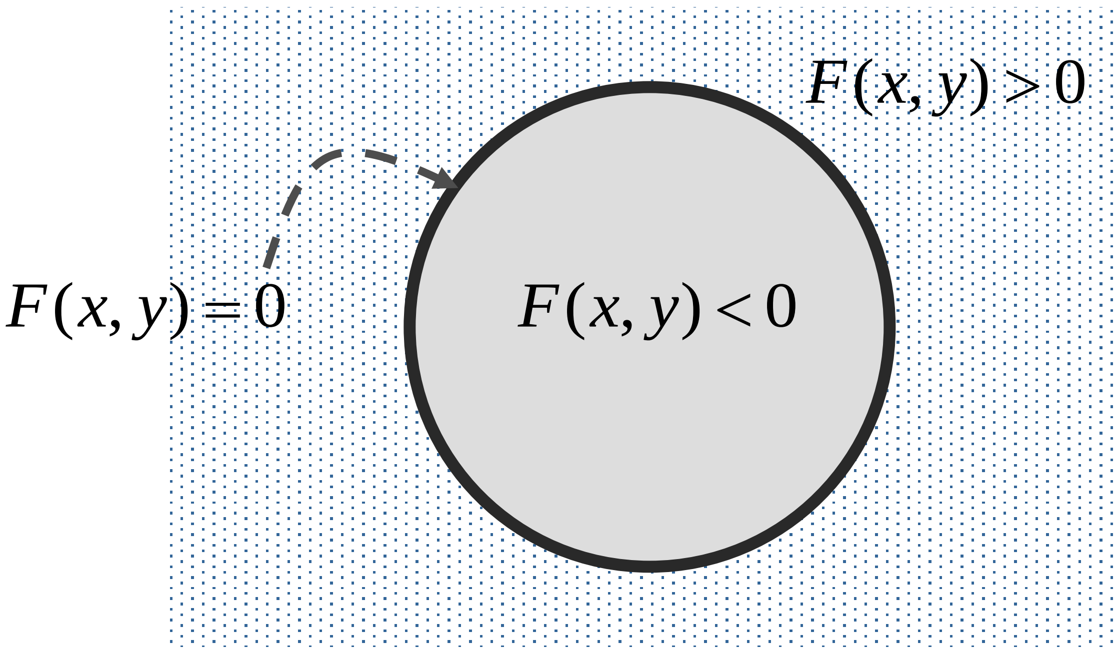
Primitivas de rasterização – círculos (6)

Algoritmo Midpoint

- Defina-se a função ***F(x,y)*** como

$$F(x,y) = x^2 + y^2 - R^2$$

- Esta função será



Primitivas de rasterização – círculos (7)

Algoritmo Midpoint

- É possível deduzir que o *pixel* mais próximo do circulo é o **SE** se o ponto **M** estiver fora do círculo

- Por analogia, **E** é o *pixel* mais próximo, se **M** estiver dentro do círculo

- Define-se então uma variável de decisão **d**, que representa o valor da função no ponto médio **M**

$$d_{old} = F\left(x_p + 1, y_p - \frac{1}{2}\right) = (x_p + 1)^2 + \left(y_p - \frac{1}{2}\right)^2 - R^2$$

Primitivas de rasterização – círculos (8)

Algoritmo Midpoint

- Vamos admitir que $d_{old} < 0$, tendo sido escolhido o *pixel* **E**.
- A próxima avaliação de **d** será

$$d_{new} = F\left(x_p + 2, y_p - \frac{1}{2}\right) = (x_p + 2)^2 + \left(y_p - \frac{1}{2}\right)^2 - R^2$$

- Verifica-se que $d_{new} = d_{old} + (2x_p + 3)$, ou seja $\Delta_E = (2x_p + 3)$

- Se $d_{old} \geq 0$ escolhe-se o *pixel* **SE**, ficando

$$d_{new} = F\left(x_p + 2, y_p - \frac{3}{2}\right)$$

donde $\Delta_{SE} = (2x_p - 2y_p + 5)$

Primitivas de rasterização – círculos (9)

- Resta definir a condição inicial para **d**

- Como o algoritmo se limita ao segundo quadrante, sabemos que o *pixel* inicial fica nas coordenadas **(0, R)**

- O próximo ficará em **(1, R-½)**, logo

$$F\left(1, R - \frac{1}{2}\right) = \frac{5}{4} - R$$

- Para eliminar as operações em vírgula flutuante, é possível efectuar um conjunto de manipulações algébricas

Primitivas de rasterização – círculos (10)

Algoritmo Midpoint

```
void drawCircle(int radius) {
   int x = 0;
   int y = radius;
   int d = 1 - radius;

   writeCirclePixel(x, y);
   while(y > x) {
     if(d < 0) {  // escolher E
       d += 2 * x + 3;
     } else {
       d += 2 * (x - y) + 5;
       --y;
     }
     ++x;
     writeCirclePixel(x, y);
   }
}
```

Referências

- **Computer Graphics: Principles and Practice in C,**
  James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes,
  Addison-Wesley Professional; 2nd edition (1995)

- **The Bresenham Line-Drawing Algorithm,**
  http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html, 2006

- **Equations of the Straight Line,**
  http://www.cut-the-knot.org/Curriculum/Calculus/StraightLine.shtml, 2006

- **JAVA 2D**
  API geral: http://java.sun.com/j2se/1.5.0/docs/api/
  Applets: *http://java.sun.com/docs/books/tutorial/deployment/applet/index.html*
  Graphics: *http://java.sun.com/j2se/1.5.0/docs/api/java/awt/Graphics.html*