Desenho de Segmentos de Recta

Sistemas Gráficos/

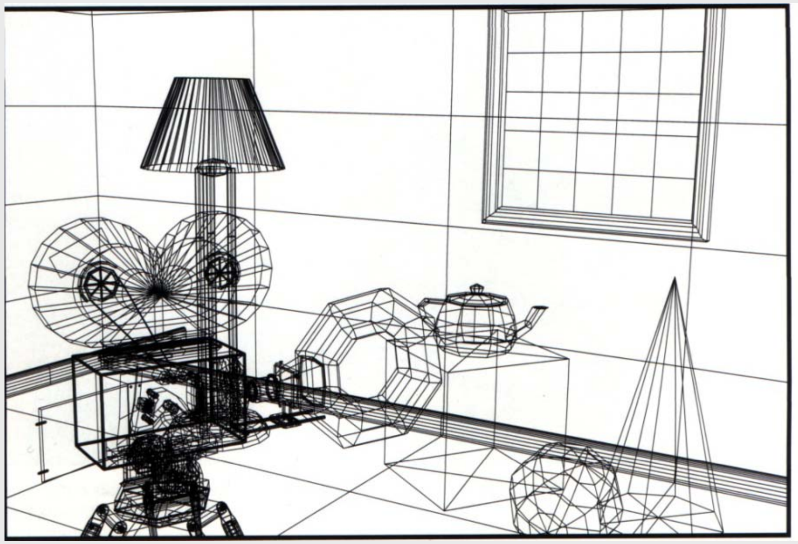
Computação Gráfica e Interfaces

Alg. para desenho de Segmentos de Recta - Motivação

• A maior parte das primitivas 2D, desenhadas centenas ou mesmo milhares de vezes por frame são obtidas pelo desenho de segmentos de recta.

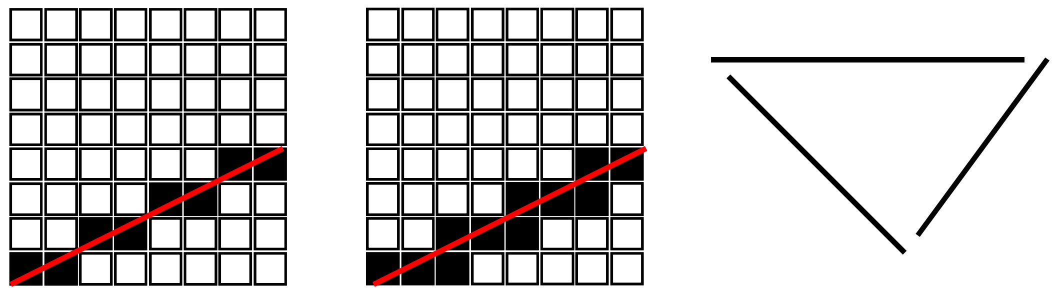
• Mesmo o desenho 3D em *wiredframe* é obtido por segmentos de recta 2D.

• A optimização destes algoritmos resulta num aumento de eficiência da aplicação.

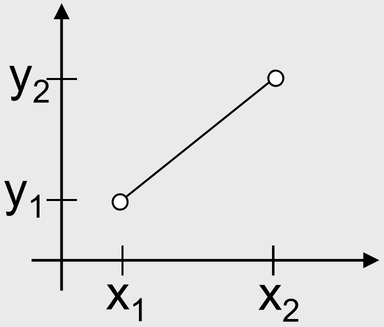


Alg. para desenho de Segmentos de Recta - Requisitos

- O algoritmo tem de obter coordenadas inteiras, porque só pode endereçar coordenadas (x,y) inteiras no *raster display.*
- Quando é criada uma imagem, os algoritmos que trabalham ao nível do *pixel* são chamados centenas ou milhares de vezes -> têm de ser eficientes.
- Os algoritmos devem criar linhas com aspecto visual satisfatório:
    - Devem parecer “rectas”
    - Terminar com precisão
    - Brilho constante



Alg. para desenho de Segmentos de Recta



Equação da recta:

$$y = m.x + b$$

Declive:

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

Podemos observar que:

Se m<1 então x avança sempre de uma unidade; y repete, ou não, o valor anterior.

Se m>1 então y avança sempre de uma unidade; x repete, ou não, o valor anterior.

A equação pode ser simplificada para:

$y_{i+1} = m.x_{i+1} + b = m(x_i + \Delta x) + b = y_i + m.\Delta x$ Fazendo $\Delta x = 1$, $\boldsymbol{y_{i+1} = y_i + m}$

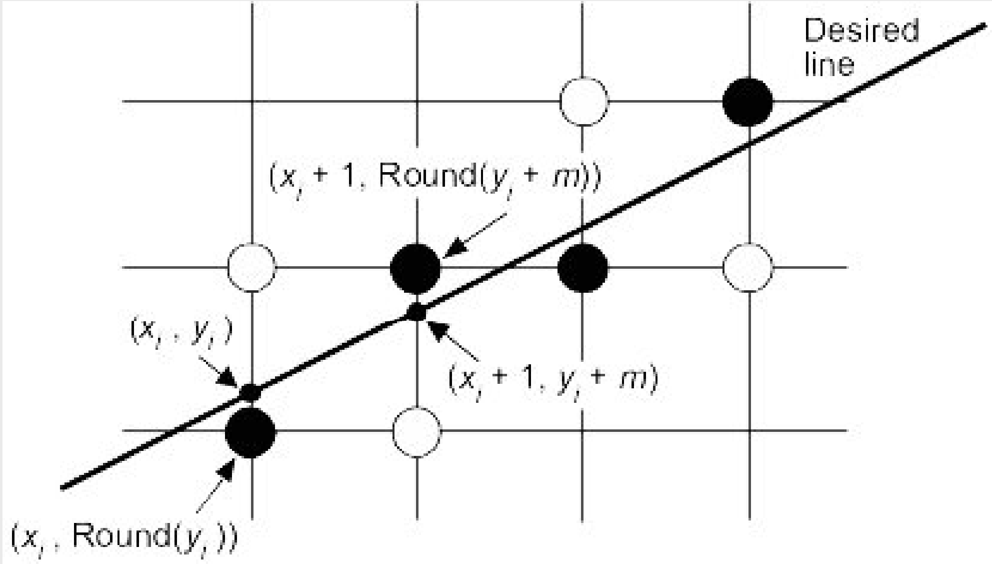
Algoritmo Básico para desenhar o segmento de recta (m<1)

1. Incrementar *x* de 1 em cada passo, partindo do ponto mais à esquerda.
2. $y_{i+1} = y_i + m$
3. O ponto da recta será: $(x_{i+1}, round(y_{i+1}))$ $\Big\{$ O pixel mais próximo da recta real, i.e. cuja distância é a menor.

DDA – Digital Differential Analyser

```
void DDA(int X0, int Y0, int X1, int Y1)
{   //considerando -1 <=m <=1  e  X0<X1
    int  x;
    float dy, dx, y, m;

    dy = Y1 – Y0;
    dx = X1 – X0;
    m = dy/dx;
    y = Y0;
    for (x=X0; x<=X1;x++) {
       WritePixel(x, (int)(y + 0.5));
       y += m;
    }
}
```



Problemas do algoritmo:

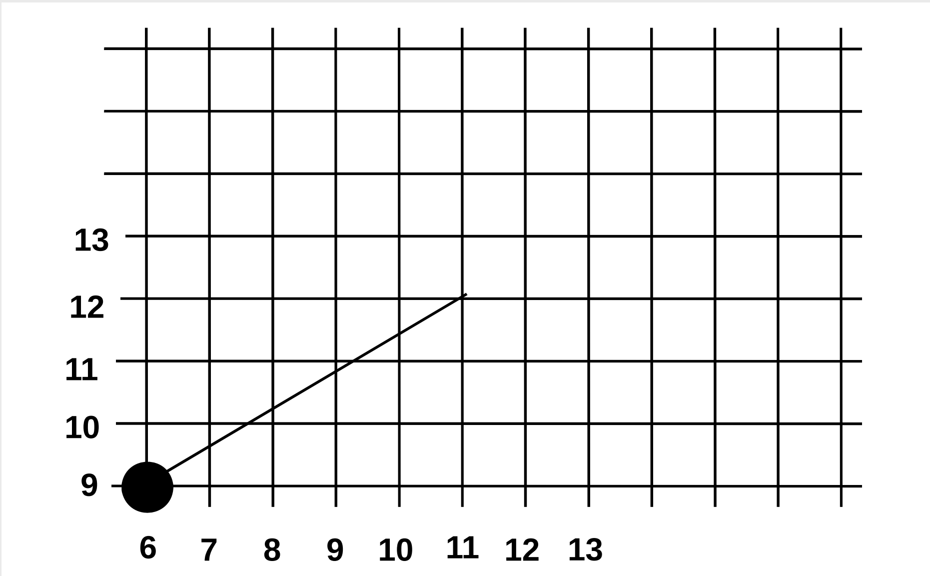
1. Operações em vírgula flutuante -> menor eficiência do que com inteiros

2. O valor de ***y*** evolui pelo incremento sucessivo de ***m***; variáveis reais têm precisão limitada -> soma acumulada de um valor inexacto pode originar um desvio do valor real pretendido *round*(y_i).

DDA – Digital Differential Analyser

Exercício:

Quais os pontos que vão pertencer ao segmento de recta entre (6,9) e (11,12) ?

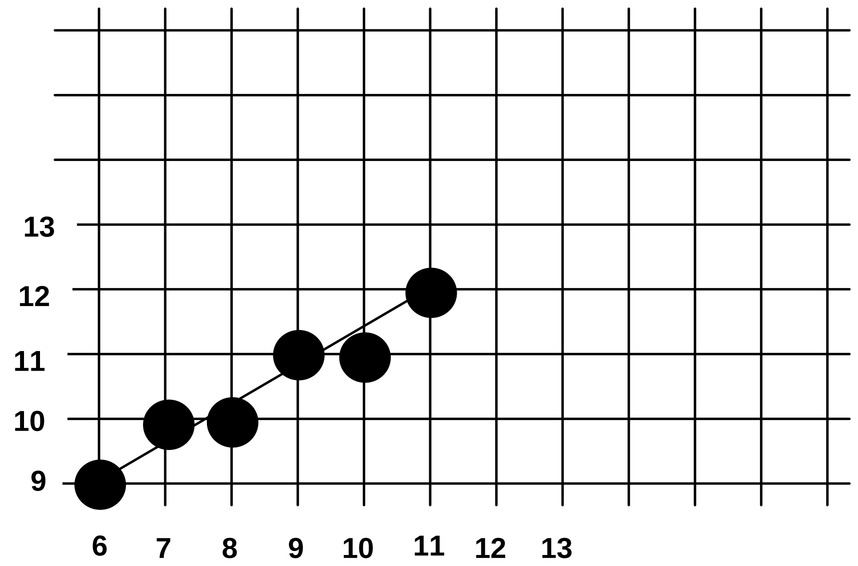


m = ?

DDA – Digital Differential Analyser

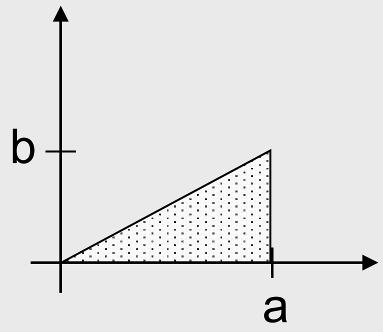
Os pontos calculados são:

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



Algoritmo Midpoint

- Supor que se pretende desenhar um segmento de recta entre os pontos (0,0) e (a,b)
  - $0 <= m <= 1$



- A equação da recta fica em
  $y = m.x$ sendo $m = b/a$

- $y = (b/a)x + 0 \rightarrow f(x,y) = bx - ay = 0$
  é também uma equação da recta.

Algoritmo Midpoint

Para rectas no primeiro octante, o ponto seguinte a ***P*** será ***E*** ou ***NE***.

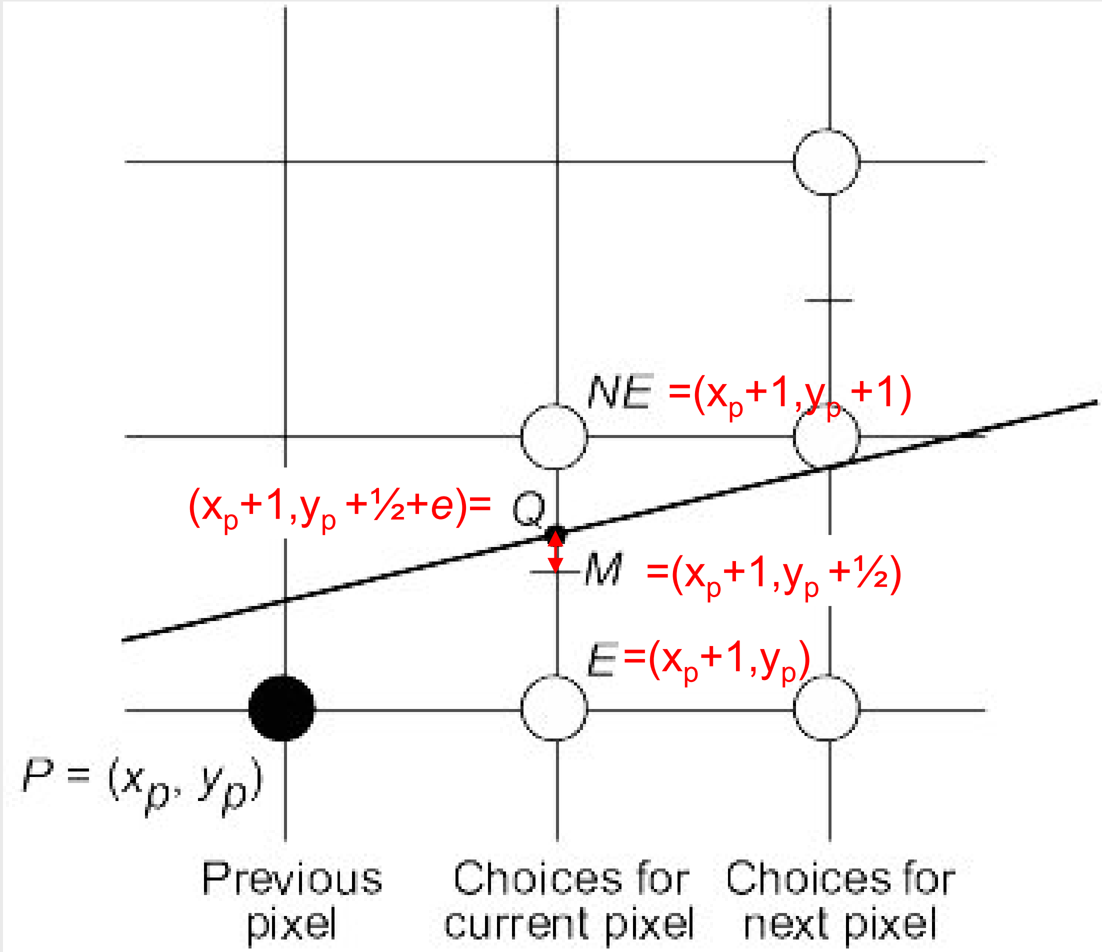
Escolher o ponto mais próximo da recta real:

$$f(x,y) = bx - ay = 0$$

Estratégia do algoritmo MidPoint:

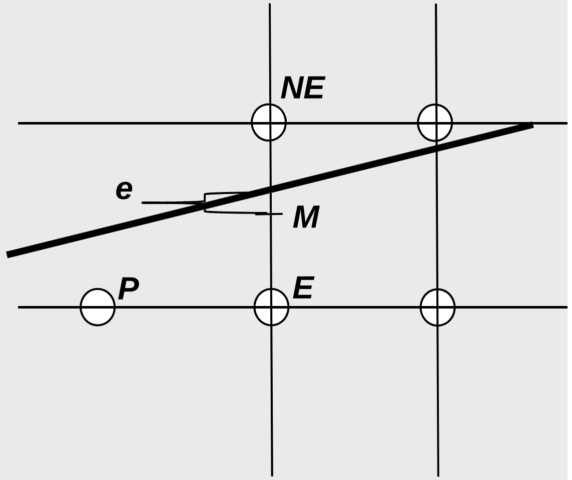
1. Verificar de que lado fica *M*
2. Se ***M*** acima da recta → escolhe ***E***
3. Se ***M*** abaixo da recta → escolhe ***NE***

O erro será sempre inferior a ½.



e=Q-M

Algoritmo Midpoint



O ponto médio entre ***E*** e ***NE*** é $(x_p + 1, y_p + ½)$.

Façamos ***e*** a distância entre o ponto onde a recta intersecta entre ***E*** e ***NE*** e o ponto médio.

Se ***e*** for positivo -> escolhe-se ***NE***

Se ***e*** for negativo -> escolhe-se ***E***

Conclui-se que para escolher o ponto correcto apenas é necessário saber o sinal de ***e***.

$f(x_p + 1, y_p + ½ + e) = 0 \quad \Leftrightarrow$ (ponto pertence à recta)

$b(x_p + 1) - a(y_p + ½ + e) = 0 \Leftrightarrow$

$b(x_p + 1) - a(y_p + ½) – a.e = 0 \Leftrightarrow$

$f(x_p + 1, y_p + ½) – a.e = 0 \Leftrightarrow$

$f(x_p + 1, y_p + ½) = a.e$

Designemos uma variável de decisão d_p como:

$d_p = f(x_p + 1, y_p + ½)$

Sendo $a > 0$

sign(**e**) =
sign(a.**e**) =
sign($f(x_p + 1, y_p + ½)$) =
sign(d_p)

→ apenas é necessário calcular o sinal de d_p para escolher o próximo ponto.

Algoritmo Midpoint

Calcular $d_p = f(x_p + 1, y_p + ½)$ em cada etapa requer pelo menos duas adições, uma subtracção e duas multiplicações → ineficiente

Para optimizar esse cálculo, podemos calcular o valor da variável de decisão de uma iteração $\mathbf{d_{i+1}}$ com base no seu valor anterior $\mathbf{d_i}$, e no “caminho” (NE ou E) seguido.

Genericamente,

$$d_{i+1} = f(x_{i+1} + 1, y_{i+1} + ½)$$

Sendo que, para $\mathbf{d_i >= 0}$ (movimento para **NE**),

$$\mathbf{x_{i+1} = x_i + 1} \text{ e } \mathbf{y_{i+1} = y_i + 1}$$

Enquanto que, para $\mathbf{d_i < 0}$ (movimento para **E**),

$$\mathbf{x_{i+1} = x_i + 1} \text{ e } \mathbf{y_{i+1} = y_i}$$

Algoritmo Midpoint

O algoritmo pode ser então composto da seguinte forma:

// Calcular d_0 directamente. Para cada i >= 0:

if $d_i \geq 0$ then

Plot $(x_i + 1, y_i + 1)$ // Escolhe ***NE*** como próximo ponto

$d_{i+1} = f(x_{i+1} + 1, y_{i+1} + ½) = f((x_i + 1) + 1, (y_i + 1) + ½)$

$= b(x_i + 1 + 1) - a(y_i + 1 + ½) = f(x_i + 1, y_i + ½) + b - a$

$= d_i + b - a$

else

Plot$(x_i + 1, y_i)$ // Escolhe ***E*** como próximo ponto

$d_{i+1} = f(x_{i+1} + 1, y_{i+1} + ½) = f((x_i + 1) + 1, y_i + ½)$

$= b(x_i + 1 + 1) - a(y_i + ½) = f(x_i + 1, y_i + ½) + b$

$= d_i + b$

Conclusão: Sabendo $\mathbf{d_i}$, apenas temos de somar um valor constante para saber d_{i+1}; o valor a somar pode ser **(d_i + b – a)** ou **(d_i + b)**, dependendo de se ter avançado para **NE** ou para **E**.

Algoritmo Midpoint

O valor d_0 pode ser obtido por:

$d_0 = f(x_0 + 1, y_0 + 1/2) = f(0 + 1, 0 + 1/2) = b.1 – a.1/2 = b - a/2$

Quando ***a*** é um número ímpar d_0 assume valores não inteiros. Uma vez que só nos interessa conhecer o sinal de d_i em cada etapa, podemos multiplicar toda a equação por 2 que não alteramos em nada o funcionamento do algoritmo:

Inicialização de d:

$$\mathbf{D_0 = 2.(b - a/2) = 2b - a}$$

Actualização de D quando movimento é para NE:

$$D_{i+1} = D_i + 2.(b - a)$$

Actualização de D quando movimento é para E:

$$D_{i+1} = D_i + 2.b$$

Algoritmo Midpoint

```
MidPoint(int X0, int Y0, int X1, int Y1)
{  int a, b, d, inc1, inc2, x, y;
   a = X1 – X0;
   b = Y1 – Y0;
   inc2 = 2*b;
   d = inc2 – a;        //   d  = 2*b – a;
   inc1 = d - a;        // inc1 = 2*(b-a);
   x = X0; y=Y0;
   for(i=0; i<a; i++)
   {   plot(x,y);
       x = x+1;
       if (d >= 0)
       {    y=y+1;   d=d+inc1; }
       else{d=d+inc2; }
   }
}
// Para rectas no primeiro octante e 0<=m<=1
```

Vantagens:
- Apenas aritmética inteira (**+** e ***2**).
- Permite o cálculo incremental dos pontos, i.e. obter (x_{i+1},y_{i+1}) a partir de (x_i,y_i).

Algoritmo Midpoint

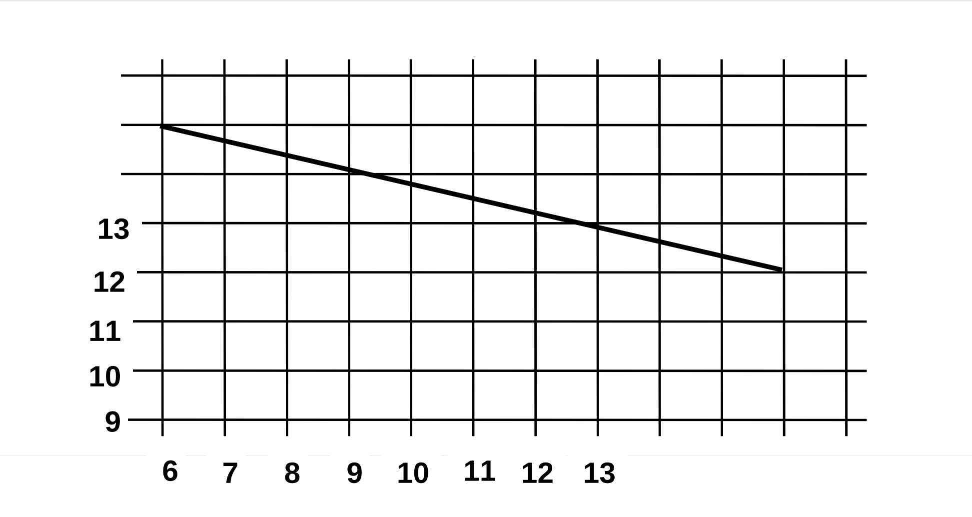
Vantagens:

Apenas aritmética inteira.

Permite o cálculo incremental dos pontos, i.e. obter (x_{i+1},y_{i+1}) a partir de (x_i,y_i).

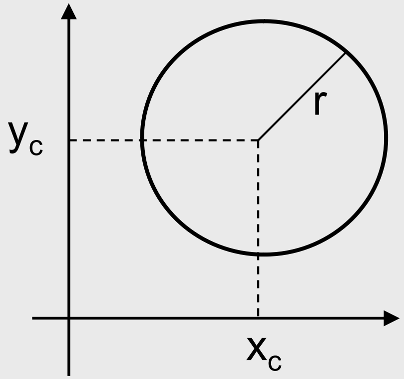
Exercícios:

1. Generalize o algoritmo para funcionar com qualquer declive ***m****.*
2. Implemente o código no programa do trabalho prático.



3. Utilize o algoritmo de Midpoint para obter a tabela de pontos e o valor de $\boldsymbol{d_i}$ em cada etapa para o caso da figura.

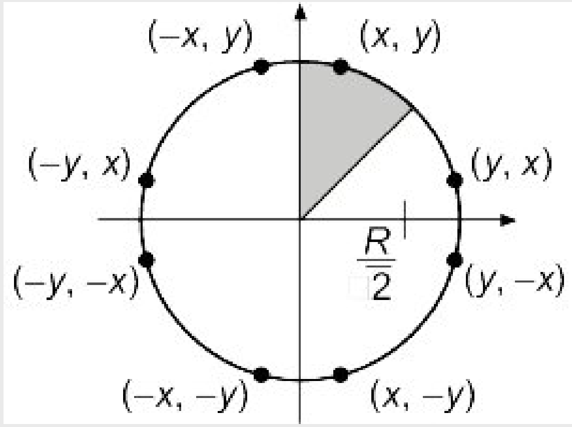
Algoritmo Midpoint para desenho de circunferências



Algumas propriedades das circunferências:

1. Calcular a circunferência pela sua equação $(x-x_c)^2+(y-y_c)^2 = r^2$ não é eficiente.

2. A simetria da circunferência pode ser explorada:



Obtendo ***(x,y)*** obtém-se também:

(-x,y) (-x,-y) (x, -y)

(y,x) (-y,x) (-y,-x) (y, -x)

→ Calcula-se apenas o segundo octante

x=0 até x=y=R/sqrt(2)

3. Se centro em (0,0) → $f(x,y)=x^2+y^2-r^2$

f(x,y) < 0 então (x,y) está **dentro** da circunferência

= 0 então (x,y) está **sobre** a circunferência

> 0 então (x,y) está **fora** da circunferência

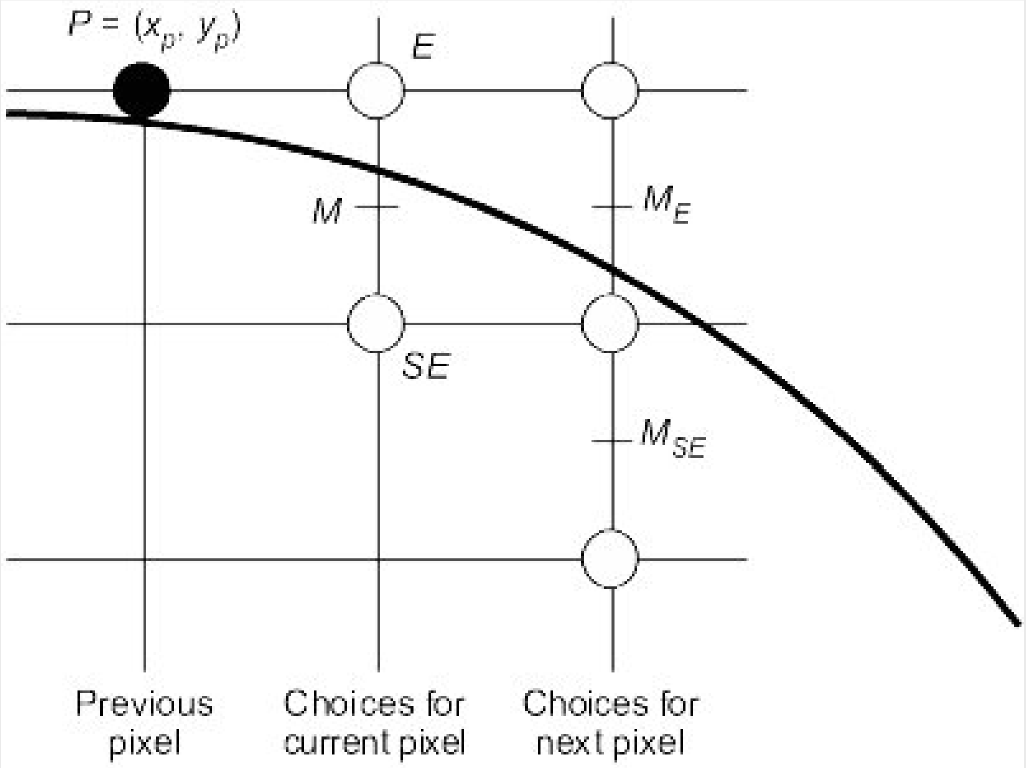
Algoritmo Midpoint para desenho de circunferências

Da mesma forma que foi feito para a recta define-se a variável de decisão ***d***:

$d_p = f(x_p + 1, y_p - ½) =$

$(x_p+1)^2+(y_p-½)^2 - r^2$

Subtracção



Algoritmo Midpoint para desenho de circunferências

Algoritmo:

// Calcular d_0 directamente. Para cada i >= 0:

if $d_i \geq 0$ then

Plot $(x_i + 1, y_i - 1)$ // Escolhe ***SE*** como próximo ponto

$d_{i+1} = f(x_{i+1} + 1, y_{i+1} - ½) = f(x_i + 1 + 1, y_i - 1 - ½)$

$= (x_i + 2)^2 + (y_i -3/2)^2 – r^2$

$= d_i + (2x_i - 2y_i + 5)$

else

Plot$(x_i + 1, y_i)$ // Escolhe ***E*** as next point

$d_{i+1} = f(x_{i+1} + 1, y_{i+1} - ½) = f(x_i + 1 + 1, y_i - ½)$

$= (x_i + 2)^2 + (y_i - ½)^2 – r^2$

$= d_i + (2x_i + 3)$

Conclusão: Podemos obter d_{i+1} a partir de d_i, mas é necessário calcular o incremento em cada etapa.

Algoritmo Midpoint para desenho de circunferências

O valor d_0 pode ser obtido considerando o primeiro ponto (0,R):

$$d_0 = f(0 + 1, R - 1/2) = 1 + (R^2 - R + ¼) - R^2 = 5/4 - R$$

```
MidPointCircle(int R)
{   int x, y;
    float d;

    x=0;  y=R;
    d = 5.0/4.0 – (float)R;
    plor(x,y);
    while(y > x)
    {     if (d >= 0)
          {    d=d+(x-y)*2+5;
               x++;  y--;  }
          else
          {    d=d+2.0*x+3;
               x++;        }
          plot(x,y);
    }
}
```

Observações:
- Utiliza aritmética em vírgula flutuante.
- Minimiza as operações efectuadas em vírgula flutuante

Algoritmo Midpoint para desenho de circunferências

Algoritmo optimizado:

```
MidPointCircle(int R)
{   int x, y, p, inc_E, inc_SE;

    x=0;  y=R;
    p=1-R;
    inc_E=3; inc_SE=5-2*R;
    plot(x,y);

    while(y > x)
    {     if (p<0)
          {    p=p+inc_E;
               inc_E=inc_E+2;
               inc_SE=inc_SE+2;
               x++;
          }
          else
          {    p=p+inc_SE;
               inc_E=inc_E+2;
               inc_SE=inc_SE+4;
               x++; y--;
          }
          plot(x,y);
    }
}
```

Observações:
- Utiliza aritmética de inteiros
- Faz uso de incrementos de segunda ordem