9 Técnicas de *Raster* – Algoritmos de Conversão Matricial

Algoritmos de Conversão Matricial são algoritmos capazes de determinar na matriz de *pixels* da superfície de exibição quais os *pixels* devem ser alterados de forma a simular-se a aparência do elemento gráfico desejado.

9.1 Conversão de Segmento de Reta

A conversão de segmentos de reta (ou simplesmente linhas) em um *grid* 2D de *pixels* consiste em determinar quais os *pixels* que fornecem a melhor aproximação do segmento (ver Figura 9.1). Este processo é chamado conversão por varredura.

As características essenciais de um algoritmo de conversão de linhas devem ser, em primeiro lugar, que o segmento convertido comece e termine exatamente sobre as extremidades P0(x_0, y_0) e P1(x_1, y_1). Esta característica implica que as coordenadas (x_0, y_0) e (x_1, y_1) devem ser definidas como inteiros. Em segundo, que os *pixels* selecionados quando o segmento tem orientação P_0 -> P_1 sejam os mesmos selecionados no caso em que o segmento tem orientação inversa, ou seja, os *pixels* ligados devem independer da orientação da reta.

A seguir são descritos dois algoritmos, o algoritmo básico incremental e o algoritmo do ponto médio, sendo este último equivalente ao algoritmo de Bresenham.

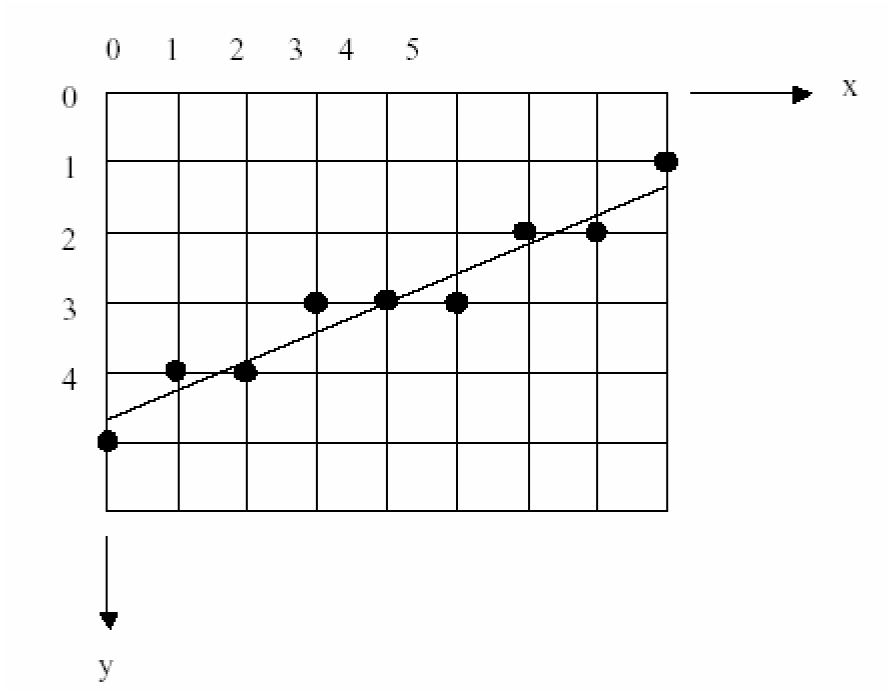

Figura 9.1 - Conversão de segmentos de reta em *pixels*.

9.1.1 Algoritmo básico incremental

O algoritmo básico incremental, também conhecido como DDA (*digital differencial analyzer*), assume as seguintes hipóteses:

a) O ponto extremo (x_0, y_0) encontra-se à esquerda de (x_1, y_1).

b) A inclinação $m = (y_1 - y_0) / (x_1 - x_0)$ é tal que : $|m| \leq 1$.

A Figura 9.2 mostra este algoritmo, escrito em linguagem C. A função *line* chama a função fictícia *writepixel*, que desempenha o papel de ligar um *pixel* com uma determinada cor. Esta função não faz parte do padrão C, e depende do compilador utilizado. As declarações de funções (protótipos de funções) e argumentos são feitas seguindo o padrão C++. Assume-se que o atributo cor possa ser representado por uma variável inteira *color*.

Para inclinações $|m| > 1$ os papéis desempenhados por x e y devem ser invertidos. O algoritmo DDA tem a desvantagem de tratar as variáveis y e m como reais.

}

```
include <math.h>

/*  Declaração de funções:  */

int round(float x);

/*  Rotina para a conversão de linhas pelo método DDA :        */

void line (int x0,int y0,int x1,int y1,int color)
{
            int x, round( );
            float dx, dy, y, m;

            dx = x1 - x0;
            dy = y1 - y0;
            m  = dy / dx;

            y = y0;
            for ( x = x0; x < = x1; x++)  {
                        writepixel(x,round(y),color);
                        y = y + m;
            }
}

/*  Rotina para o arredondamento de um número real:       */

int round(float x)
{
            return( (int)floor(x+0.5) );
}
```

Figura 9.2 - Algoritmo para conversão de linhas pelo método DDA.

Outra versão:

```
Algoritmo DDA

int x, y, x1, x2, y1, y2, valor;

float a;

a = (y2 – y1)/(x2 – x1);

for ( x = x1; x <= x2; x++) {

      /* arredonda y */

      y = (y1 + a * ( x - x1 ));

      write_pixel (x, y, valor);

}
```

9.1.2 Algoritmo do ponto médio

O algoritmo do ponto médio utiliza apenas aritmética inteira e produz o mesmo resultado, ou seja, gera os mesmos *pixels*, que o algoritmo DDA.

Considere inicialmente uma reta de inclinação $0 \leq m \leq 1$, sendo (x_0,y_0) o ponto extremo esquerdo inferior e (x_1,y_1) a extremidade direita superior (Figura 9.3).

Assumindo que o *pixel* $P(x_p,y_p)$ tenha sido selecionado, o próximo passo consiste em escolher entre os *pixels* E (um incremento à direita de P) e NE (um incremento à direita e um incremento acima de P), como mostra a Figura 9.4.

Seja Q o ponto de interseção entre a linha que se deseja traçar e a vertical do *grid* $x = x_p + 1$. A técnica do ponto médio consiste em verificar em que lado da reta o ponto médio M se situa. Se o ponto M se encontra acima da linha, então o *pixel* E é o mais próximo da linha. Se M está abaixo da reta, então o *pixel* NE é o mais próximo da reta.

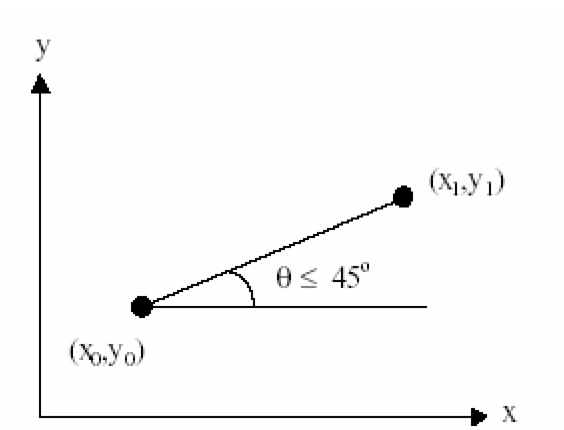

Figura 9.3 - Reta no 10 octante.

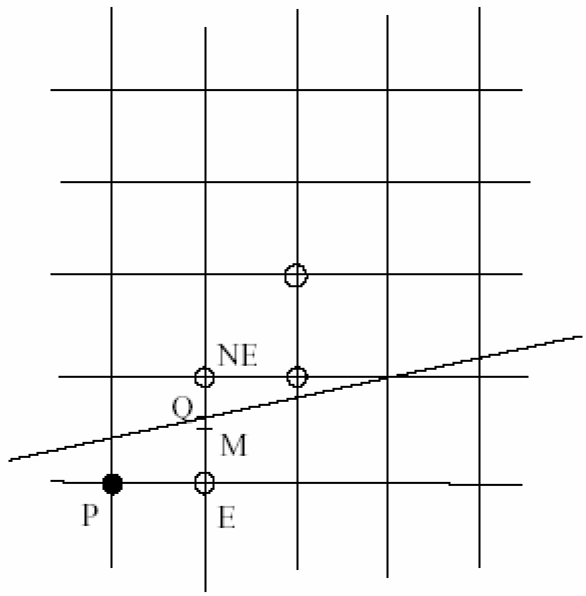

Figura 9.4 - Critério de seleção de um *pixel*.

A linha pode passar entre NE e E, ou ambos os *pixels* podem estar de um mesmo lado da linha, mas, em qualquer das hipóteses, o teste do ponto médio seleciona o *pixel* mais próximo. O erro, ou seja, a distância vertical entre o *pixel* e a linha, será sempre $\leq$ 1/2. Representando a reta na forma,

$$F(x,y) = ax + by + c = 0 \qquad (1.1)$$

e comparando com a equação:

$$y = (dy/dx)x + B \qquad (1.2)$$

pode-se verificar que os coeficientes de (1.1), a, b e c são iguais a:

$$a = dy$$
$$b = - dx$$
$$c = B\ dx$$

sendo $dx = x_1 - x_0$ e $dy = y_1 - y_0$. A função F(x,y) é igual a zero para os pontos da reta, positiva para os pontos abaixo da reta, e negativa para os pontos acima da reta. Para aplicar o critério do ponto médio deve-se, portanto, utilizar uma variável de teste d igual a:

$$d = F(\ M\ ) = F(\ x_p + 1,\ y_p + 1/2\ ) \qquad (1.3)$$

Utilizando a expressão (1.1) obtém-se :

$$d = a.(x_p + 1) + b.(y_p + 1/2) + c \qquad (1.4)$$

Se d > 0, seleciona-se o *pixel* NE, e se d ≤ 0 seleciona-se o *pixel* E. O ponto médio M e o valor de d para a linha vertical seguinte do *grid*, x = xp + 2, dependem ambos da escolha entre E e NE. Se E foi escolhido, o ponto M é incrementado em 1 na direção x:

$$dnew = F(xp+2, yp+1/2) = a.(xp+2) + b.(yp+1/2) + c \qquad (1.5)$$

Fazendo,

$$d_{old} = a.(x_p+1) + b.(y_p+1/2) + c \qquad (1.6)$$

e subtraindo d_{old} de d_{new} obtém-se a diferença incremental ΔE:

$$\Delta E = d_{new} - d_{old} = a.[(x_p+2)-(x_p+1)] = a = dy \qquad (1.7)$$

Portanto, o próximo valor de d pode ser calculado de uma forma incremental fazendo-se:

$$d_{new} = d_{old} + \Delta E = d_{old} + a \qquad (1.8)$$

Se o pixel NE foi o escolhido, o ponto M é incrementado em uma unidade em ambas as direções x e y, e então:

$$d_{new} = F(x_p + 2,y_p + 3/2) = a.(x_p + 2) + b.(y_p + 3/2) + c \qquad (1.9)$$

Neste caso, a diferença incremental ΔNE é dada por:

$$\Delta NE = d_{new} - d_{old} = a + b = dy - dx \qquad (1.10)$$

e, em conseqüência,

$$d_{new} = d_{old} + \Delta NE = d_{old} + a + b \qquad (1.11)$$

Em resumo, o algoritmo do ponto médio seleciona, em cada passo, um entre dois *pixels*, dependendo do sinal da variável de teste. A variável de teste é então atualizada somando-se ΔE ou

ΔNE, conforme o *pixel* escolhido no passo anterior. O primeiro *pixel* é simplesmente a extremidade (x_0, y_0), e o primeiro valor de d é calculado através da expressão:

$$d_{start} = F(x_0+1, y_0+1/2) = a(x_0+1) + b(y_0+1/2) + c =$$
$$= F(x_0, y_0) + a + b/2 \qquad (1.12)$$

Como (x_0, y_0) é um ponto da reta, então a expressão acima se reduz a:

$$d_{start} = a + b/2 = dy - dx/2 \qquad (1.13)$$

Para eliminar a fração em (1.13), pode-se multiplicar a função F(x, y) por 2, observando-se que a multiplicação de F por uma constante positiva não altera o sinal da variável de teste:

$$F(x, y) = 2(ax + by + c) \qquad (1.14)$$

Pode-se então escrever:

$$d_{start} = 2a + b = 2dy - dx \qquad (1.15)$$
$$\Delta E = 2a = 2dy \qquad (1.16)$$
$$\Delta NE = 2(a + b) = 2(dy - dx) \qquad (1.17)$$

O algoritmo da Figura 9.5 sintetiza o desenvolvimento acima. Na generalização do algoritmo do ponto médio para qualquer inclinação deve-se assegurar que o segmento de reta P_0P_1 contém os mesmos *pixels* que o segmento P_1P_0. A única situação em que a escolha do *pixel* depende da direção da reta é quando a linha passa exatamente pelo ponto médio M, ou seja, quando d = 0. Portanto, quando se caminha da direita para a esquerda, deve-se selecionar o *pixel* SW para d = 0, como mostra a Figura 9.6. A Figura 9.7 ilustra as oito possibilidades a serem consideradas na generalização do algoritmo.

```
/*  Rotina para conversão de linhas pelo método do ponto médio :     */

midpoint_line ( int x0, int y0, int x1, int y1, int color )
{
        int dx, dy, incrE, incrNE, d, x, y ;

        dx  =  x1 - x0 ;
        dy  =  y1 - y0 ;
        d   =  2 * dy - dx ;
        incrE    =  2 * dy ;
        incrNE   =  2 * ( dy - dx ) ;
        y  =  y0 ;

        for ( x = x0 ;  x < x1 ;  x++ )  {
                writepixel ( x, y, color );
                if ( d > 0 )  {
                        d  =  d + incrNE ;
                        y++;  }
                else
                        d  =  d + incrE ;
        }
        writepixel ( x, y, color ) ;
}
```

Figura 9.5 - Conversão de linhas pelo método do ponto médio.

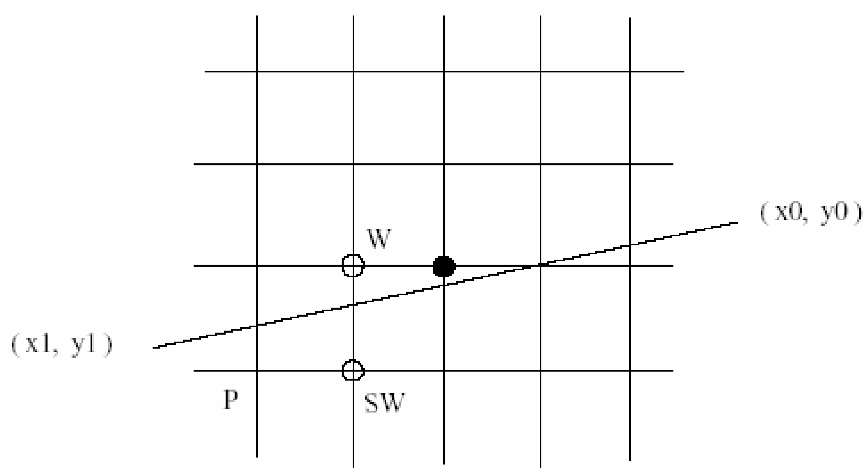

Figura 9.6 - Sentido direita-esquerda na conversão de linhas.

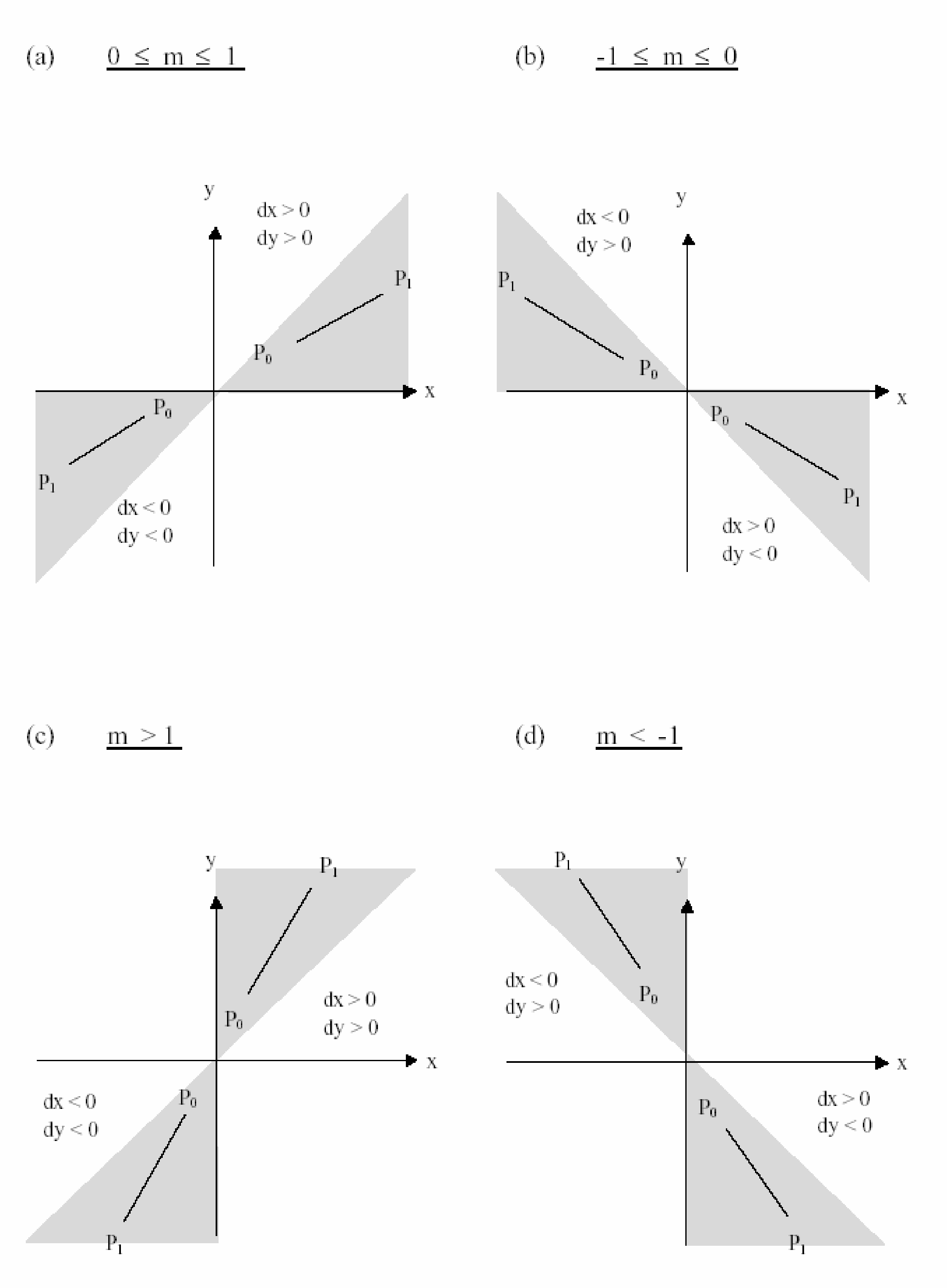

Figura 9.7 - Inclinação de retas nos oito octantes.

9.2 Conversão Matricial de Circunferências

A equação de uma circunferência com centro na origem e raio R, em coordenadas cartesianas, é dada por: $x^2 + y^2 = R^2$

Circunferências não centradas na origem podem ser transladadas para a origem, e os pixels reais podem ser traçados aplicando-se um offset aos pixels gerados pelo algoritmo de conversão matricial.

Existem muitas abordagens simples, porém ineficientes, para o traçado de círculos: Em algoritmos não incrementais, um polígono regular de n lados é usado como aproximação para a circunferência. Para que a aproximação seja razoável, deve-se escolher um valor suficientemente alto para n. Entretanto, quanto maior o valor de n, mais lento será o algoritmo, e várias estratégias de aceleração precisam ser usadas. Em geral os algoritmos incrementais de conversão matricial são mais rápidos.

Outra abordagem seria usar a equação explícita da circunferência, y = f(x): $y = \pm\sqrt{R^2 - x^2}$

Para desenhar 1/4 de circunferência (os outros 3/4 são desenhados por simetria), poderíamos variar x de 0 a R, em incrementos de uma unidade, calculando +y a cada passo através da equação acima. Essa estratégia funciona, mas é ineficiente porque requer operações de multiplicação e raiz quadrada. Além disso, haverá grandes gaps nas regiões onde a tangente à circunferência é infinita (valores de x próximos a R, Figura 5.1). Uma maneira de resolver o problema dos gaps é plotar R cos e R sin, para variando de 0 a 90 graus, mas essa solução também é ineficiente, pois precisa de funções caras - sin e cos.

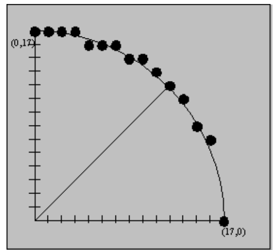

Figura 5.1 - Um arco de 1/4 de circunferência, obtido variando-se x em incrementos unitários, e calculando e arredondando y.

Simetria de ordem 8

Note que o traçado de uma circunferência pode tirar proveito de sua simetria. Considere uma circunferência centrada na origem. Se o ponto (x,y) pertence à circunferência, pode-se calcular de maneira trivial sete outros pontos da circunferência (Fig. 5.2). Consequentemente, basta computar um arco de circunferência de 45 para obter a circunferência toda. Para uma circunferência com centro na origem, os oito pontos simétricos podem ser traçados usando o procedimento CirclePoints indicado a seguir.

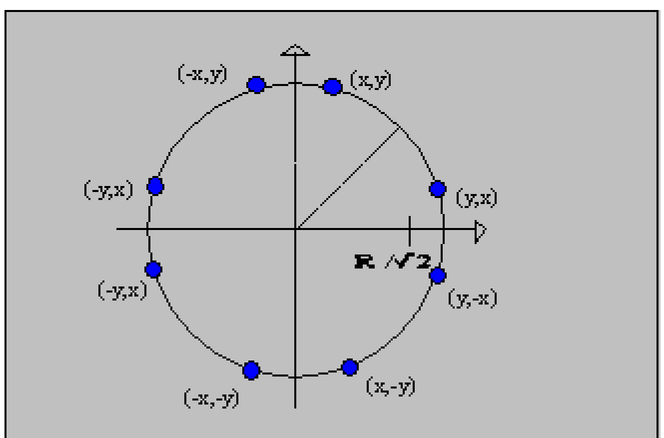

Figura 5.2 - Oito pontos simétricos em uma circunferência.

Vamos estudar especificamente um algoritmo derivado do algoritmo de Bresenham para geração de circunferências, chamado "Midpoint Circle Algorithm" em [Foley et. al].

```
void CirclePoints (int x, int y, int
value)
{
        write_pixel (x, y, value);
        write_pixel (y, x, value);
        write_pixel (y, -x, value);
        write_pixel (x, -y, value);
        write_pixel (-x, -y, value);
        write_pixel (-y, -x, value);
        write_pixel (-y, x, value);
        write_pixel (-x, y, value);
}
```

Procedimento *CirclePoints*.

Algoritmo do "Ponto-Médio" para Circunferências

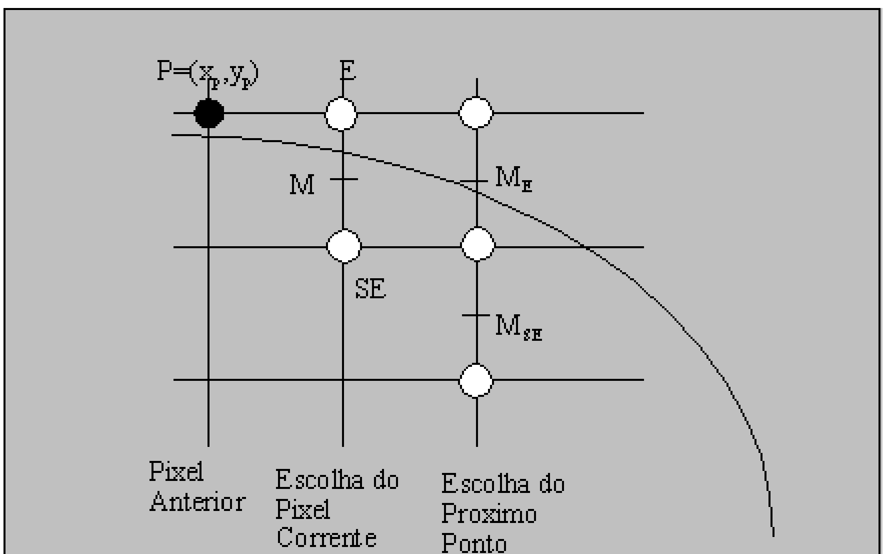

Figura 5.3 - Malha de pixels para o Algoritmo do Meio Ponto para Circunferências, ilustrando a escolha entre os pixels E e SE.

Consideraremos apenas um arco de 45 da circunferência, o 2º octante, de x=0, y=R a x=y= R/(2)1/2, e usaremos o procedimento CirclePoints para traçar todos os pontos da circunferência. Assim como o algoritmo gerador de linhas, a estratégia é selecionar entre 2 pixels na malha aquele que está mais próximo da circunferência, avaliando-se uma função no ponto intermediário entre os dois pixels. No segundo octante, se o pixel P em (xp,yp) foi previamente escolhido como o mais próximo da circunferência, a escolha do próximo pixel será entre os pixels E e SE (Fig. 5.3).

Seja a função F(x,y) = x2 + y2 - R2; cujo valor é 0 sobre a circunferência, positivo fora dela e negativo dentro. Se o ponto intermediário (o "ponto-médio") entre os pixels E e SE está fora da circunferência, o pixel SE é escolhido, porque está mais próximo dela. Por outro lado, se o pixel intermediário está dentro da circunferência, então o pixel E é escolhido.

Assim como no caso das linhas, a escolha é feita com base na variável de decisão d, que dá o valor da função no "ponto-médio":

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

Se dold < 0, E é escolhido, e o próximo ponto-médio será incrementado de 1 na direção x. Assim,

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

e dnew = dold + (2xp + 3); consequentemente E = 2xp + 3.

Se dold >= 0, SE é escolhido, e o próximo ponto-médio será incrementado de 1 na direção x e decrementado de 1 na direção y. Portanto:

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

Como dnew = dold + (2xp - 2yp + 5), SE = 2xp - 2yp + 5.

Note que no caso da reta (equação linear), E e NE eram constantes. No caso da circunferência (equação quadrática), E e SE variam a cada passo, sendo funções do valor específico de (xp,yp), o pixel escolhido na iteração anterior (P é chamado "ponto de avaliação"). As funções podem ser avaliadas diretamente, a cada passo, dados os valores de x e y do pixel escolhido na iteração anterior. Essa avaliação não é computacionalmente cara, uma vez que as funções são lineares.

Resumindo, os mesmos dois passos executados para o algoritmo do traçado de linhas são executados para o algoritmo de circunferências:

(1) escolher o pixel com base no sinal da variável d, calculada na iteração anterior;

(2) atualizar a variável d com o valor correspondente ao pixel escolhido. A diferença é que, na atualização de d, calculamos uma função linear do ponto de avaliação.

Falta ainda calcular a condição inicial (o 1º valor de d). Limitando a utilização do algoritmo a raios inteiros no segundo octante, o pixel inicial é dado por (0,R). O próximo "ponto-médio" está em $(1, R - 1/2)$, e portanto $F(1, R - 1/2) = 1 + (R2 - R + 1/4) - R2 = 5/4 - R$. O algoritmo, bastante parecido com o algoritmo para traçado de retas, é mostrado a seguir.

```
        void MidpointCircle (int raio, int valor)
/* Assume que o centro da circunferencia e' a origem */
        { int x,y;
          float d;

          /* Inicialização das variaveis */
          x=0;
          y=raio;
          d=5/4 - raio;
          CirclePoints (x,y,valor)

          while (y > x) {
                    if (d < 0) {  /* Seleciona E */
                       d=d + 2*x + 3;
                       x++;
                       }
                  else {   /* Seleciona SE */
                            d=d + 2*(x-y) + 5;
                            x++;  y--;
                            }
                    CirclePoints (x,y,valor)
          } /* Fim while */
        }  /* Fim da rotina MidpointCircle */
```

Algoritmo do **Ponto-Médio** para conversão Matricial de Circunferências.

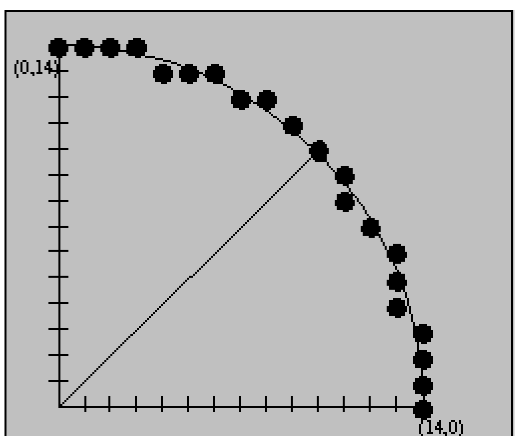

Figura 5.4 - Segundo octante da circunferência gerado com o algoritmo do Meio-Ponto, e primeiro octante gerado por simetria.

O algoritmo pode ser facilmente modificado para manipular circunferências centradas fora da origem, ou de raios não inteiros. Entretanto, um problema com esta versão é a necessidade de usar aritmética real, devido ao valor inicial de d.

Para eliminar as frações, vamos definir uma nova variável de decisão, h, onde h = d - 1/4, e substituir d por h + 1/4 no código. Agora, a inicialização é h = 1 - R, e a comparação d < 0 torna-se h < -1/4. Entretanto, como o valor inicial de h é inteiro, e a variável é incrementada de valores inteiros (E e SE), a comparação pode ser alterada simplesmente para h < 0. Temos agora um algoritmo que opera em termos de h, usando apenas aritmética inteira, que é mostrado a seguir. (Para manter a notação consistente com o algoritmo de linhas, substituimos novamente h por d). A Figura 5.4 mostra o segundo octante de uma circunferência de raio 14 gerada pelo algoritmo.

```
    void MidpointCircle (int raio, int valor)
/* Assume que o centro da circunferencia e' a origem
   Utiliza apenas aritmetica INTEIRA */
    { int x,y,d;

      /* Inicialização das variaveis */
      x=0;
      y=raio;
      d=1 - raio;
      CirclePoints (x,y,valor)

      while (y > x) {
            if (d < 0) {  /* Seleciona E */
               d=d + 2*x + 3;
               x++;
               }
          else {   /* Seleciona SE */
                  d=d + 2*(x-y) + 5;
                  x++;  y--;
                  }
            CirclePoints (x,y,valor)
      } /* Fim while */
    }  /* Fim da rotina MidpointCircle */
```

Algoritmo do **Ponto-Médio para conversão Matricial de Circunferências utilizando apenas aritmética Inteira**.

Diferenças de Segunda Ordem. É possível melhorar o desempenho do algoritmo usando ainda mais extensivamente a técnica da computação incremental. Vimos que as funções são equações lineares, e no algoritmo, elas estão sendo computadas diretamente. Entretanto, qualquer polinômio pode ser calculado incrementalmente - assim como foi feito com as variáveis de decisão. (Na verdade, estamos calculando diferenças parciais de 1ª e 2ª ordens). A estratégia consiste em avaliar as funções diretamente em dois pontos adjacentes, calcular a diferença (que no caso de polinômios é sempre um polinômio de menor grau), e aplicar esta diferença a cada iteração.

Se escolhemos E na iteração atual, o ponto de avaliação move de (xp,yp) para (xp + 1,yp). Como vimos, a diferença de 1ª ordem é Eold em (xp,yp) = 2xp + 3. Consequentemente,

$$\Delta E_{new} \; em \; (x_p + 1, y_p) = 2(x_p + 1) + 3$$

e a diferença de 2ª ordem é Enew - Eold = 2.

Analogamente, SEold em (xp,yp) = 2xp - 2yp + 5. Consequentemente,

$$\Delta SE_{new} \; em \; (x_p + 1, y_p) = 2(x_p + 1) - 2y_p + 5$$

e a diferença de 2ª ordem é SEnew - SEold = 2.

Se escolhemos SE na iteração atual, o ponto de avaliação move de (xp,yp) para (xp + 1,yp - 1). Consequentemente

$$\Delta E_{new} \; em \; (x_p + 1, y_p - 1) = 2(x_p + 1) + 3$$

e a diferença de 2ª ordem é Enew - Eold = 2. Além disso,

$$\Delta SE_{new} \; em \; (x_p + 1, y_p - 1) = 2(x_p + 1) - 2(y_p - 1) + 5$$

e a diferença de 2ª ordem é SEnew - SEold = 4.

A versão final do algoritmo dada a seguir, consiste dos seguintes passos:

(1) escolher o pixel com base no sinal da variável d calculada na iteração anterior;

(2) atualizar a variável d usando E ou SE, usando o valor de calculado na iteração anterior;

(3) atualizar os s para consisderar o movimento para o próximo pixel, utilizando as diferenças constantes previamente calculadas; e

(4) mover para o próximo pixel. E e SE são computados usando o pixel inicial (0,R).