Universidade Federal de Jataí
Curso de Ciências da Computação

Computação gráfica – Uma abordagem prática

(Notas de aulas)
Módulo II

Marcos Wagner de Souza Ribeiro

2021

4- Geração de Primitivas Gráficas

Uma imagem pode ser descrita de várias maneiras. Uma delas é através de um dispositivo *raster* que a imagem é formada pela coloração das posições do conjunto de pixel dos dispositivos. Em outro extremo, a imagem pode ser formada por um conjunto de objetos complexos, como por exemplo a árvore em uma floresta ou uma mobília em uma sala, os quais são posicionados em coordenadas específicas dentro da cena.

As formas e cores dos objetos podem ser definidas internamente por *arrays* de pixel ou por um conjunto de estruturas geométricas básicas, tais como segmentos de reta e áreas poligonais coloridas. A cena é apresentada, então, tanto pela formação de *arrays* de pixel em uma estrutura de buffer quanto pela convenção em estruturas geométricas básicas descritas em um molde (padrão) de pixel.

Tipicamente, pacotes de programação gráfica oferecem funções para descrever uma cena em termos dessas estruturas geométricas básicas, referenciadas como primitivas de saída, e agrupam o conjunto de primitivas de saída em estruturas mais complexas. Cada primitiva de saída é especificada por coordenadas de entrada e por outros dados informando à maneira que os objetos deverão ser mostrados. Pontos e segmentos de reta são as estruturas geométricas mais simples de uma imagem. Primitivas de saída adicionais, que também são usadas para construir uma imagem, incluem círculos e outras sessões cônicas, superfícies quadráticas, curvas e superfícies irregulares, áreas poligonais coloridas e palavras formadas por caracteres.

Nas próximas seções, serão vistos procedimentos de geração de primitivas gráficas, descrevendo algoritmos ao nível dos dispositivos para exibir as primitivas de saída, com ênfase nos métodos de conversão *scan* para sistemas gráficos *raster*.

Rasterização

Rasterização é o processo de conversão da representação vetorial para a matricial. Ela permite realizar a conversão de um desenho tridimensional qualquer em uma representação inteira, possível de ser armazenada na memória (de vídeo ou impressão) de um dispositivo raster. A figura abaixo ilustra a rasterização de uma reta.

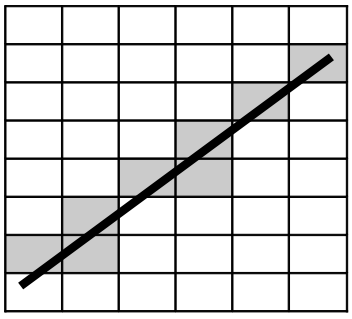


Figura 4.1 – Conversão da representação de uma reta na forma vetorial para a matricial.

Grande parte dos dispositivos de entrada e saída, tal como filmadoras digitais, scanners, vídeos e impressoras, usam uma tecnologia matricial, também denominada tecnologia ***raster***. Esses dispositivos possuem uma memória na qual é composta a imagem a ser posteriormente exibida no dispositivo.

Um vídeo *raster* é composto de uma memória em que estão armazenadas as informações que descrevem a imagem. Essa memória de vídeo é uma área de armazenamento onde cada posição indica quando um determinado pixel na tela deve estar apagado ou aceso e em qual cor.

4.1 Geração de Linhas

Ao tentar desenhar uma reta no vídeo, devemos nos lembrar de que essa reta não poderá ser desenhada da mesma forma que a desenharmos em uma folha de papel, ou seja, nem sempre será uma reta perfeita. Ela será desenhada pelos pixels que puderem ser acessados no dispositivo de visualização utilizado, através de uma aproximação a ser obtida com a utilização do quadriculado formado pelos pixels (Figura 4.1).

Dependendo da inclinação traçada, podemos obter uma linha com uma aparência serrilhada. Isso é denominado *aliasing* e é devido às quebras de continuidade impostas pela malha de pontos. Essa quebra das linhas tende a ser muito mais aparente à medida que os pontos apresentados na tela forem de tamanho maior (*dot pitch*) ou que o dispositivo possuir menor resolução (números possíveis de pixels nas duas direções). Esses serrilhados podem ser melhorados através da aplicação de algoritmos de anti-serilhamento conhecidos como algoritmos de *anti-aliasing*.

Os algoritmos de *anti-aliasing* buscam tentar "enganar" o olho humano. Eles geralmente conseguem isso fazendo as bordas de um desenho ficarem um pouco "borradas", ou melhor, menos nítidas. Geralmente usam uma cor intermediária entre a cor da linha e a cor do fundo, obtendo assim suavização do contraste entre as duas cores. Dessa maneira, tem-se uma linha que terá uma aparência mais perfeita para uma pessoa que a observe a certa distância.

Sendo a tela gráfica uma matriz de pontos, é impossível traçar uma linha direta de um ponto a outro, principalmente as linhas inclinadas (ou seja, com angulação diferente de 45º). Sendo assim, alguns pontos da tela deverão ser selecionados para representar a reta que se deseja desenhar. Esta seleção é feita pelos algoritmos de rasterização.

Linhas retas

Do ponto de vista matemático, uma reta no plano pode ser descrita por: $y = mx + b$.

Essa expressão pode ser interpretada da seguinte maneira: para qualquer valor de x, existe um valor de y, dado por $mx + b$. O parâmetro m tem um interpretação especial, sendo chamado de **coeficiente angular**, pois está ligado ao ângulo que a reta faz com o eixo x. Para $m <= 1$, a reta faz um ângulo entre 0º e 45º com o eixo x. Para $m > 1$ o ângulo encontra-se entre 45º e 90º.

O **coeficiente linear** b dá o valor do eixo y cruzado pela reta.

Dados dois pontos no plano, P1 e P2, pode-se obter m e b, ou seja, a equação da reta que passa pelos pontos: $m = (y2 – y1) / (x2 – x1)$ e $b = y1 – m*x1$

4.1.1 Método Analítico

O método analítico é o método mais simples que utiliza a equação da reta para obter todos os pontos de uma linha reta. A partir dos pontos extremos da linha, P1(x1,y1) e

P2(x2,y2), obtém o valor do coeficiente de inclinação da reta (*m*) e o valor da variável *b*, caso esta linha intercepte o eixo Y. Veja os cálculos abaixo:

y = m*x + b (Equação da reta)

m = DY / DX => m = (y2-y1)/(x2-x1)

b = y1 – m*x1

Iniciando a variável *x* com valor da coordenada mais à esquerda da linha, obteremos os valores da variável *y* a medida que incrementamos a variável *x* por 1. Os pontos da reta poderão ser ligados por meio da seguinte estrutura repetitiva:

Na linguagem C, poderia ser escrito o código da seguinte forma:

```
putpixel(x1,y1);
for(int x = x1; x <= x2; x++)  {
     y = m . x + b;
     set_pixel(x,y);
}
```

Porém, este método possui desvantagens:

- operações com ponto flutuante, no entanto, os pixels são inteiros;
- são realizados muitos cálculos no processo, o que gera ineficiência, pois a capacidade de processamento dos computadores da época (dec. 60) era fraca;
- escolha do pixel não é um fator considerado na elaboração da solução, o que pode ser qualquer um das redondezas do número obtido nas contas efetuadas. Isto altera a continuidade dos pontos da reta (favorece o aparecimento de "buracos").

4.1.2. Analisador Diferencial Digital (DDA)

A técnica utilizada no algoritmo DDA é similar a técnica do Algoritmo Analítico, porém estabelece uma relação primária comparativa entre o Dy e o Dx.

$Dy = m \cdot Dx \qquad Dx = Dy/m$

Se $Dx > Dy$, então incrementamos o intervalo Dx em uma unidade e calculamos os sucessivos valores para a coordenada y, pela seguinte equação:

$Dy = m * Dx \Rightarrow y_f - y_{f-1} = m \Rightarrow y_f = m + y_{f-1}$

Se $Dx < Dy$, então incrementamos o intervalo Dy em uma unidade e calculamos os sucessivos valores para a coordenada x, pela seguinte equação:

$Dx = Dy/m \Rightarrow x_k - x_{k-1} = 1 \Rightarrow x_k = 1/m + x_{k-1}$

Em termos práticos é definir um valor de incremento para o eixo com a menor distância entre o ponto inicial e o final.

A seguir, descrevemos o método DDA através da linguagem de programação em C e por meio de um diagrama para auxiliar no entendimento deste algoritmo.

```
if (abs(dx) > abs(dy))
      steps = abs(dx);
else steps = abs(dy);
xIncrement = dx / (float) steps;
yIncrement = dy / (float) steps;
setPixel(ROUND(x), ROUND(y));
for(k=0; k<steps; k++) {
      x += xIncrement;
      y += yIncrement;
      setpixel(ROUND(x), ROUND(y));
```

O algoritmo DDA é um método mais rápido para os cálculos de posição de pixels do que o algoritmo do método analítico. No entanto, este algoritmos ainda possui vários problemas, como:

- os vários arredondamentos das variáveis do tipo ponto-flutuante, levavam ao erro ao calcular a posição dos pixels;
- a utilização da aritmética de ponto-flutuante e a multiplicação, ainda exigiam muito tempo do processamento;

4.1.3. Algoritmo de Bresenham

O algoritmo de BRESENHAM propõe um trabalho somente com inteiros, ou seja, não há uso de variáveis reais. Para simplificar o algoritmo, supomos o incremento como sendo uma unidade e a inclinação da linha entre 0 e 1.

O algoritmo aproveita a coerência espacial, escolhendo entre dois valores de pixel vizinho. Isto é, em vez de computar o valor do próximo ***y*** em ponto flutuante, decidir se o próximo pixel irá ter as coordenadas *(x+1,y)* ou *(x+1, y+1) tendo como referência o incremento em X*. Para isso, requer que se avalie se a linha passa acima ou abaixo do ponto médio *(x+1,y+ 1/2)*.

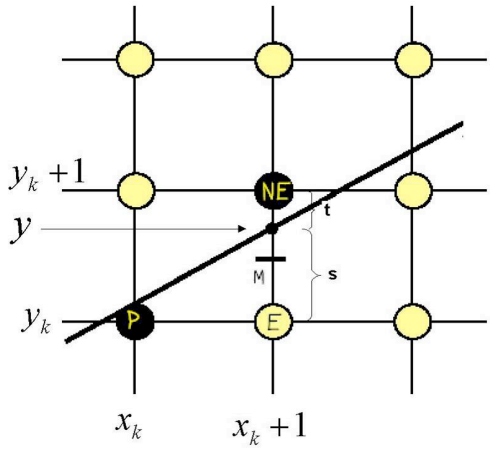
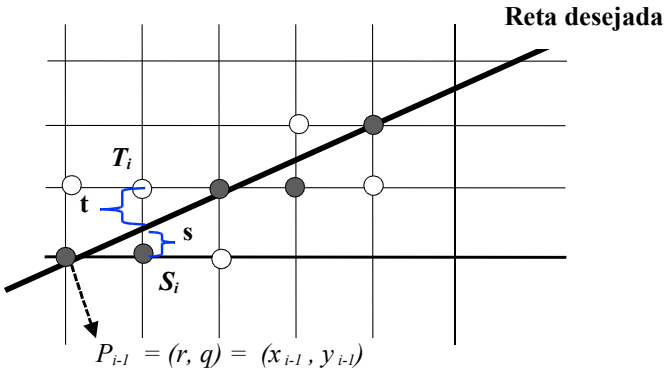


Figura 4.2 – Representação do algoritmo de Bresenham para geração de uma reta.

Se s < t então o pixel *E*(ou $S_i = (x_k+1,\ y)$) está mais perto da linha desejada e será o escolhido; senão será o pixel *NE*(ou $T_i(x_k+1,\ y_k+1)$) que estará mais perto da linha e será o escolhido.

Diante da escolha entre dois valores de pixel vizinhos, o algoritmo Bresenham usa uma variável de decisão *d i* que é proporcional à diferença entre os parâmetros "s" e "t", como mostra a figura 4.3 e os cálculos descritos a seguir.



Obs: Os pontos negros são os pixels selecionados pelo algoritmo de Bresenham.

Figura 4.3 – Representação dos parâmetros *s* e *t* utilizados no algoritmo de Bresenham.

A seguir, descrevemos o método Bresenham por meio da linguagem de programação em C e por meio de um diagrama para auxiliar no entendimento deste algoritmo.

```
D = 0;
if (abs(dx) > abs(dy)) {
      c = 2 * dx; m = 2 * dy;
      for(;;) {
            putPixel(x,y);
            if (x == xf) break;
            x += incX;
            d += m;
            if (d >= dx) {y += incY; d -= c;}
      }
} else {
      c = 2 * dy; m = 2 * dx;
      for(;;) {
            putPixel(x,y);
            if (y == yf) break;
            y += incY;
            d += m;
            if (d >= dy) {x += incX; d -= c;}
      }
}
```

5 – Preenchimento de Áreas

Na maioria dos pacotes gráficos, uma primitiva é um sólido colorido ou uma área padrão de um polígono. Algumas vezes, outros tipos de áreas são analisados, mas as áreas poligonais são as mais fáceis de serem projetadas e processadas, desde que tenham limites.

Os algoritmos responsáveis pelo preenchimento de polígonos procuram as bordas de uma dada área, de forma a definir quando o pixel deve mudar de cor e quando deve parar o algoritmo.

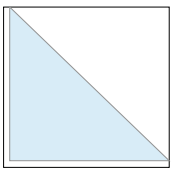
Existem três métodos básicos para preenchimento de uma área em sistemas *raster*. O método de **Varredura**, como o próprio nome diz, varre uma determinada área e tenta prever em que momento tem-se o início da borda do polígono. Um outro método inicia com um ponto do interior do polígono e começa a pintar ao redor deste ponto até encontrar as bordas do polígono, este método é chamado de **Boundary-Fill.** Por último o Análise Geométrica determina os intervalos nos quais uma linha de varredura (linha *scan line*) atravessa a área do polígono.

Nas próximas seções descreveremos os algoritmos: de Varredura, de Análise Geométrica e Boundary-Fill.

5. 1 Algoritmo de Varredura

No algoritmo de varredura o contorno do polígono já está desenhado na tela com uma determinada cor, diferente daquela escolhida para o fundo. Inicialmente define-se os pontos extremos que este polígono alcança (xmin, xmax, ymin e ymax). A partir daí busca-se encontrar um ponto preenchido e assim que ele é encontrado o método ativa o preenchimento que irá até encontrar um outro ponto preenchido. Este algoritmo possui alguns pontos negativos, como o tempo de processamento, e também depende diretamente do código construído para que seja eficiente.

No desenho abaixo tem-se a faixa de varredura em estrutura aramada e a área pintada em azul (ou preenchido).

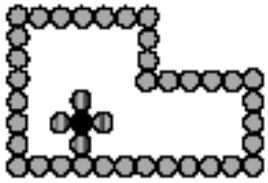


5. 2 Algoritmo de Boundary-Fill

Inicialmente, escolhe um ponto interno deste polígono. O processamento do algoritmo Boundary-Fill começa a partir desta escolha, que é então preenchido (ou pintado). Os pixels arredor do ponto escolhido, vão sendo pintados até que a borda do polígono seja encontrada.

Parâmetros de entrada:

- Um ponto (x,y) do interior do polígono
- Cor da sua borda
- Posições “vizinhas” do ponto (x,y) são testadas:
- Se não for a cor da borda o ponto é pintado com a cor de preenchimento, até que todos os *pixels* do polígono tenham sido testados.



Uma lista ligada armazena pontos que servem para continuar o algoritmo; tais pontos fazem o papel de ponto inicial, na iteração seguinte.

Este algoritmo se presta à preenchimento de qualquer área fechada.

5.3 Algoritmo de Análise Geométrica

O algoritmo de Análise Geométrica é o mais extenso em termos de fases para alcançar o resultado e baseia-se na descrição geométrica (como, por exemplo, uma lista de vértices que formam o polígono). Ele utiliza as linhas de varredura para identificar os pontos internos do polígono e as interseções com as arestas do mesmo. Os pontos de interseção que são identificados são ordenados da esquerda para a direita. Os p*ixels* entre cada par de interseções são "setados" com uma cor especificada.

Podemos descrever este algoritmo seguindo os seguintes passos:

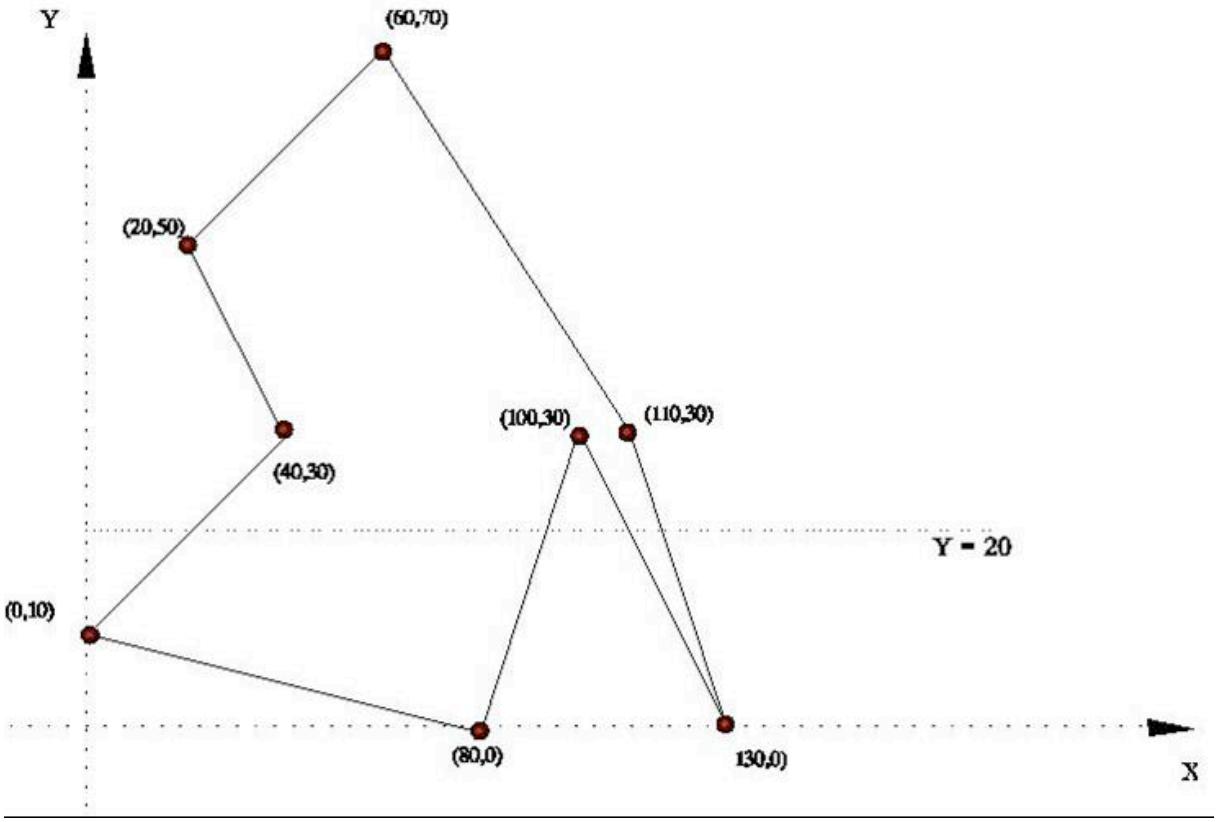


Figura 5.1 – Exemplo de um polígono com uma linha de varredura em y = 20.

1° passo) Montar a tabela de lados: neste passo será descritos todos os lados do polígono

LADO	Y_{min}	$Y_{máx}$	X para Y_{min}	1/m
1	0	10	80	-8,0
2	10	30	0	+2,0
3	30	50	40	-1,0
4	50	70	20	+2,0
5	30	70	110	-1,25
6	0	30	130	-0,67
7	0	30	130	-1,0
8	10	30	80	+0,67

É importante lembrar que: m = DY/DX => 1/m = DX/DY

2° passo) Interseção com a linha de varredura: identifica as diversas interseções que a linha de varredura possui com os lados do polígono.

Para eliminar os lados do polígono, os quais a linha de varredura não intercepta, são definidas as seguintes condições:

- Yvarredura > Ymax
- Yvarredura < Y_{min}

Se uma das condições acima for verdadeira para qualquer um dos lados do polígono, esse lado será descartado.

X = (1/M * (Yvarredura - Ymin) +X=>Ymin);

3° passo) Ordenam-se os pontos de interseção em ordem crescente e traçam-se as linhas, tomando os pontos de dois em dois. Os valores de x serão inseridos em uma lista.

O algoritmo de análise geométrica tem um tratamento especial quando há interseção com os vértices do polígono. A linha de varredura que passa pelo vértice atravessa duas arestas do polígono na mesma posição, o que faz com que dois pontos sejam adicionados à lista de interseções da *scan-line*.

6. Geração de Circunferência

Os algoritmos para geração de circunferência (cônicas, de forma geral) enfrentam problemas similares ao da geração de retas, adicionando-se a dificuldade de cálculo de funções trigonométricas. A seguir, descrevem-se algumas propostas para geração desta figura geométrica.

6.1 Equação paramétrica

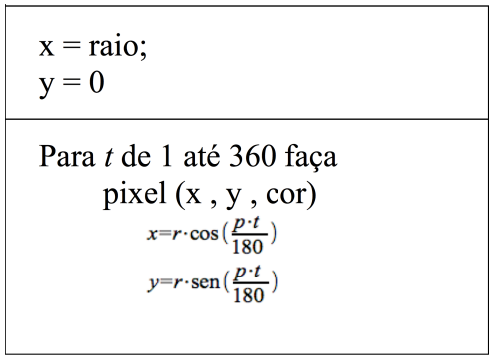
Um círculo é definido por um conjunto de pontos, e todos são constituídos pela distância entre o raio *r* e uma posição do centro (x_c,y_c).

Para se obter os pontos que definem os limites de um círculo, usamos as coordenadas polares r e [?]. Ou seja, através das equações paramétricas de um círculo, desenhamos um círculo com pontos igualmente espaçados ao longo da circunferência. As equações paramétricas são obtidas pelo cálculo de pontos num período, usando as funções **seno** e **co-seno**:

x = xc + raio * cos (ang)
y = yc + raio * sen (ang)

em que *ang* = t * (2 *pi / n) => t = 0,1,2... e n-1*

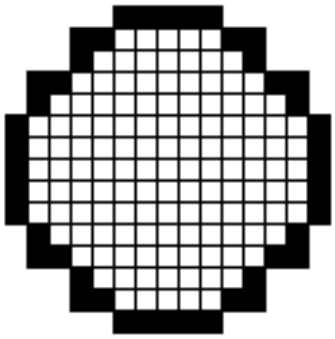
Diagrama:



Desvantagens deste método:

- uso de pontos flutuante;
- uso de funções trigonométricas, o que reduz muito a eficiência do algoritmo;
- densidade dos pontos varia com o raio, isto é, quanto maior o raio maior o número de espaços (buracos) entre os pontos.

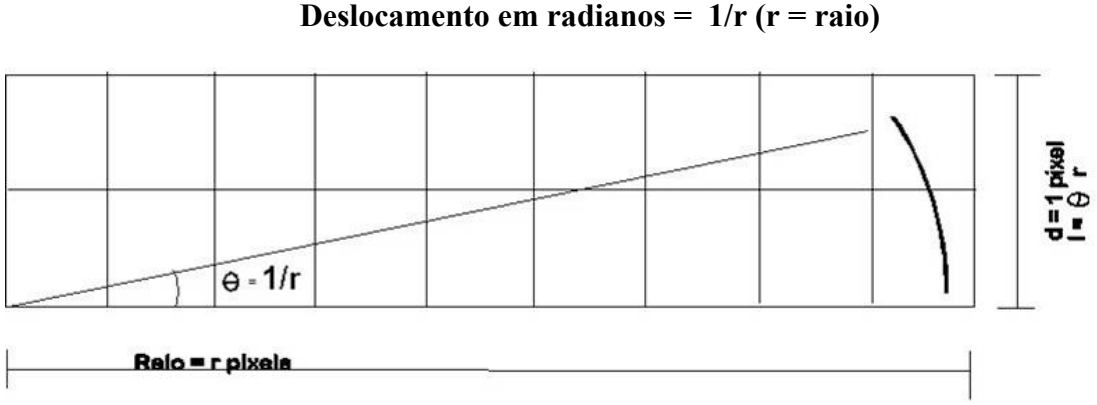
A figura as seguir mostra um possível resultado ao se criar uma circunferência pelas equações paramétricas.



Criação de uma circunferência utilizando as equações paramétricas.

6.2 Algoritmo Incremental Com Simetria

Para tentar resolver o problema dos "buracos", implementaremos o algoritmo incremental com simetria. Este algoritmo representa o deslocamento angular pelo incremento de uma unidade de pixel, como mostra a figura:



Representação do algoritmo Incremental com Simetria.

O gasto computacional também pode ser reduzido por este algoritmo, considerando a simetria da circunferência. Uma parte do círculo possui um similar nos outros quadrantes. Em outras palavras, pode-se gerar o 2º quadrante da circunferência no plano XY, pela simetria com o 1º quadrante em relação ao eixo y. Os 3º e 4º quadrantes também podem ser obtidos por simetria, pois são simétricos aos 1º e 2º quadrantes em relação ao eixo x.

A seção do círculo em octantes adjacentes é simétrica em relação à linha do ângulo de 45º. Então, basta apenas que um octante seja avaliado para obter os demais. Para cada pixel computado em um octante, oitos são pintados.

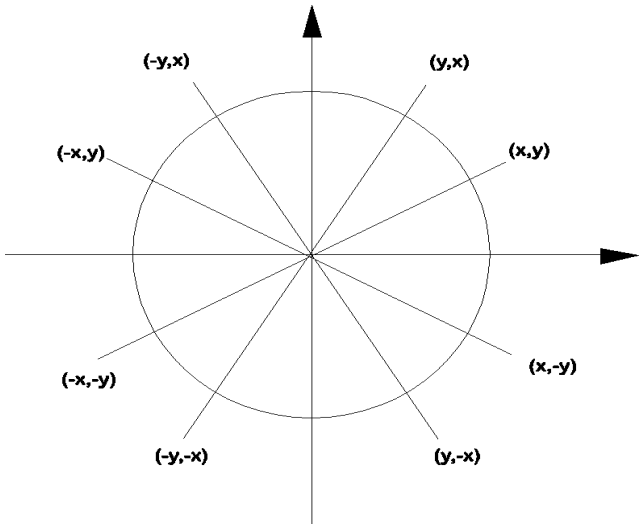


Figura 4.6 – Propriedade importante da circunferência: **simetria**.

O uso de funções trigonométricas também é solucionado neste método. Pelas equações:

$$x_{n+1} = x_n \cdot \cos q + y_n \cdot \operatorname{sen} q$$
$$y_{n+1} = y_n \cdot \cos q - x_n \operatorname{sen} q$$

O algoritmo incremental constrói uma circunferência com deslocamento angular constante e pequeno e com a rotação a partir de um ponto inicial. Nessas equações os valores do seno e do cosseno são fixos.

Descrição resumida do algoritmo Incremental com simetria:

```
        x <= r      y = 0;
% Gera pontos sobre o eixo:
        pixel (x , y , cor);
        pixel (-x , y , cor);
        pixel (x , -y , cor);
        pixel (-x , -y , cor);

% Demais pontos:
        x = r cos t;
        y = r sen t;
        pixel (x , y , cor);
        pixel (x , -y , cor);
        pixel (-x , y , cor);
        pixel (-x , -y , cor);
        pixel (y , x , cor);
        pixel (y , -x , cor);
        pixel (-y , -x , cor);
```

Desvantagens deste método:

- uso de x_n e y_n nas próximas iterações causam erros cumulativos;

- uso de números reais com necessidade de arredondamento, no cálculo de cada pixel.

6.2 Algoritmo de Bresenham

A solução dada por BRESENHAM também utiliza a noção de simetria, gerando o primeiro quadrante e os demais por simetria. Evita a utilização de raízes, potências e funções trigonométricas para não exigir esforço computacional e nem reduzir a eficiência do algoritmo.

Para aplicar o algoritmo de Bresenham, utilizaremos a função de circunferência:

$$f_{circle}(x,y) = x^2 + y^2 - r^2$$

Qualquer ponto (x,y) no limite do círculo com raio *r* satisfaz a equação do círculo, ou seja, $f_{circle}(x,y) = 0$. Se o ponto está no interior da circunferência: $f_{circle}(x,y) < 0$. E se o ponto está fora da circunferência: $f_{circle}(x,y) > 0$.

Baseando-se nesta função, será definido o parâmetro de decisão ?i que auxiliará na escolha do pixel que está mais próximo da curva ideal. A escolha deste pixel recai sobre três possíveis pixels (A, B ou C, como mostra a figura 4.7). O critério de seleção entre tais pontos leva em conta a distância relativa entre os mesmos e a circunferência ideal.

Utiliza-se ainda um círculo centrado na origem. Inicia o algoritmo no ponto (0,R) e vai aumentando *x* exaustivamente, ou começa no ponto (R,0) e vai aumentando *y* gradativamente.

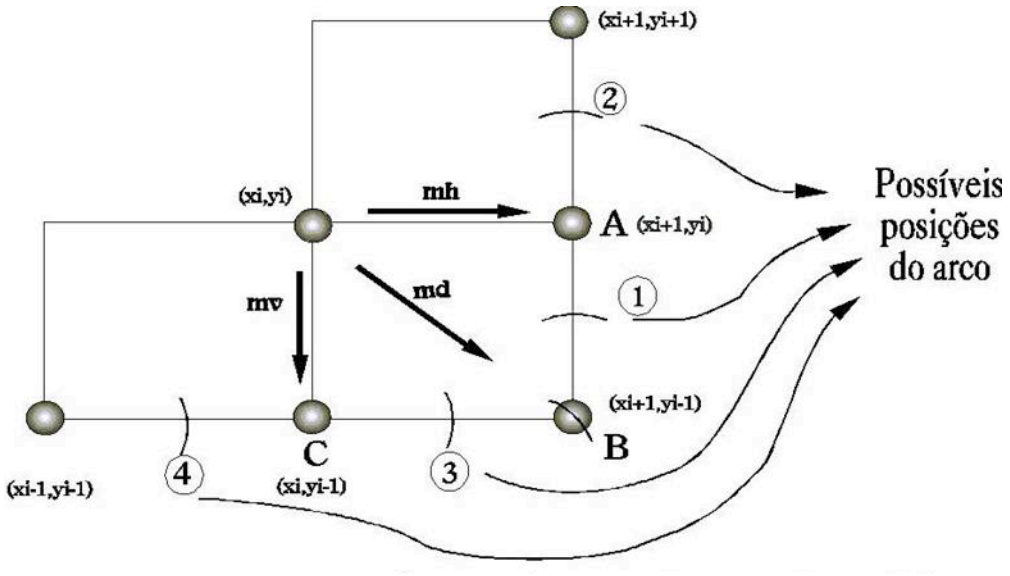


Figura 4.7 – Representação do algoritmo de Bresenham para geração de uma circunferência.

Se escolhermos como ponto inicial (0,R), a circunferência será gerada no sentido horário, e o algoritmo pode escolher entre 3 pontos diferentes. O critério de seleção entre os

pontos será dado pela distância relativa entre eles, sendo representados pelas seguintes variáveis:

- horizontalmente para direita (mh);
- diagonalmente para baixo à direita (md);
- verticalmente para baixo (mv);

$$mh = |(x_{i+1})^2 + (y_i)^2 - R^2|$$
$$md = |(x_{i+1})^2 + (y_{i-1})^2 - R^2|$$
$$mv = |(x_i)^2 + (y_{i-1})^2 - R^2|$$

O algoritmo escolhe o pixel que minimize o quadrado da distância entre um destes pixel e o círculo verdadeiro (mv, md, mh). Então, definindo o valor de *i* pela equação que calcula a diferença entre o quadrado da distância do pixel ao centro e o raio da circunferência, temos:

$i = (x_{i+1})^2 + (y_{i-1})^2 + R^2$ (pixel diagonal)

Dependendo do valor de i, temos os seguintes casos:

1° caso) Se (i = 0) => o ponto B deve ser escolhido.

2º caso) Se (i < 0)=> O ponto B está no interior do círculo, então deve-se escolher o melhor ponto pelo valor da variável d, que é calculado da seguinte forma:

d = mh - md

a) se d <= 0 o ponto escolhido é o A;
b) se d > 0 o ponto escolhido é o B;

3° caso) Se (i > 0) O ponto B está fora da circunferência então deve-se escolher o melhor ponto pelo valor da variável t, que é calculado da seguinte forma:

t = md - mv

a) se t <= 0 o ponto escolhido é o B;
b) se t > 0 o ponto escolhido é o C;

Por fim, o algoritmo de Bresenham é classificado como o melhor método para gerar uma circunferência, devido a sua eficiência, por isso é o mais utilizado nos pacotes de softwares gráficos.