

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.

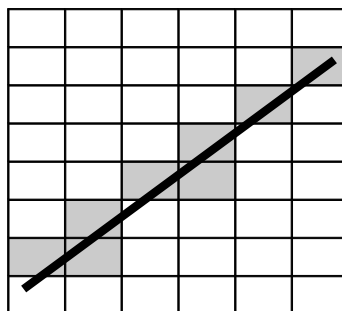


Figura 4.1 – Conversão da representação de uma reta na forma vetorial para a matricial.

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 45°). 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 $m \leq 1$, a reta faz um ângulo entre 0° e 45° com o eixo x . Para $m > 1$ o ângulo encontra-se entre 45° e 90°.

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:

$$y = m \cdot x + b \text{ (Equação da reta)}$$

$$m = DY / DX \Rightarrow m = (y_2 - y_1) / (x_2 - x_1)$$

$$b = y_1 - m \cdot x_1$$

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:

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

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 algoritmo 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 $(x+1, y+1/2)$.

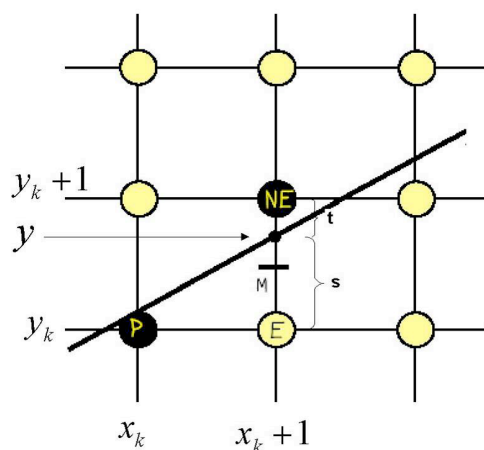
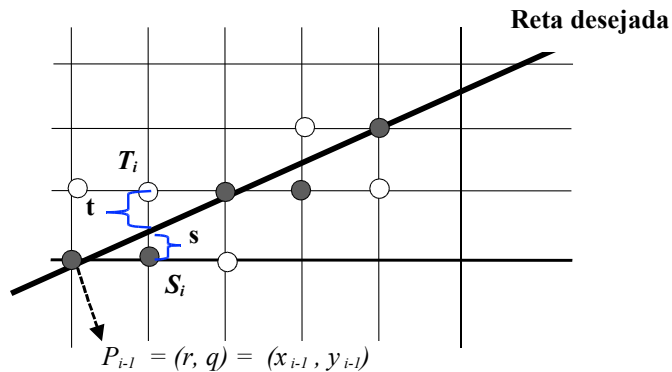


Figura 4.2 – Representação do algoritmo de Bresenham para geração de uma reta.

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 di 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.

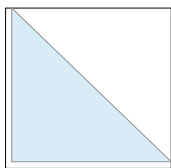
Existem três métodos básicos para preenchimento de uma área em sistemas *raster*. O método de **Varredura**, como o próprio nome diz, varre uma determinada área e tenta prever em que momento tem-se o início da borda do polígono. Um outro método inicia com um ponto do interior do polígono e começa a pintar 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 (x_{min} , x_{max} , y_{min} e y_{max}). 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).

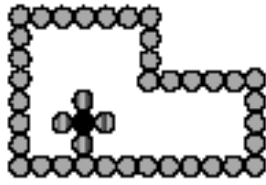


5. 2 Algoritmo de Boundary-Fill

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 *pixels* entre cada par de interseções são "setados" com uma cor especificada.

Podemos descrever este algoritmo seguindo os seguintes passos:

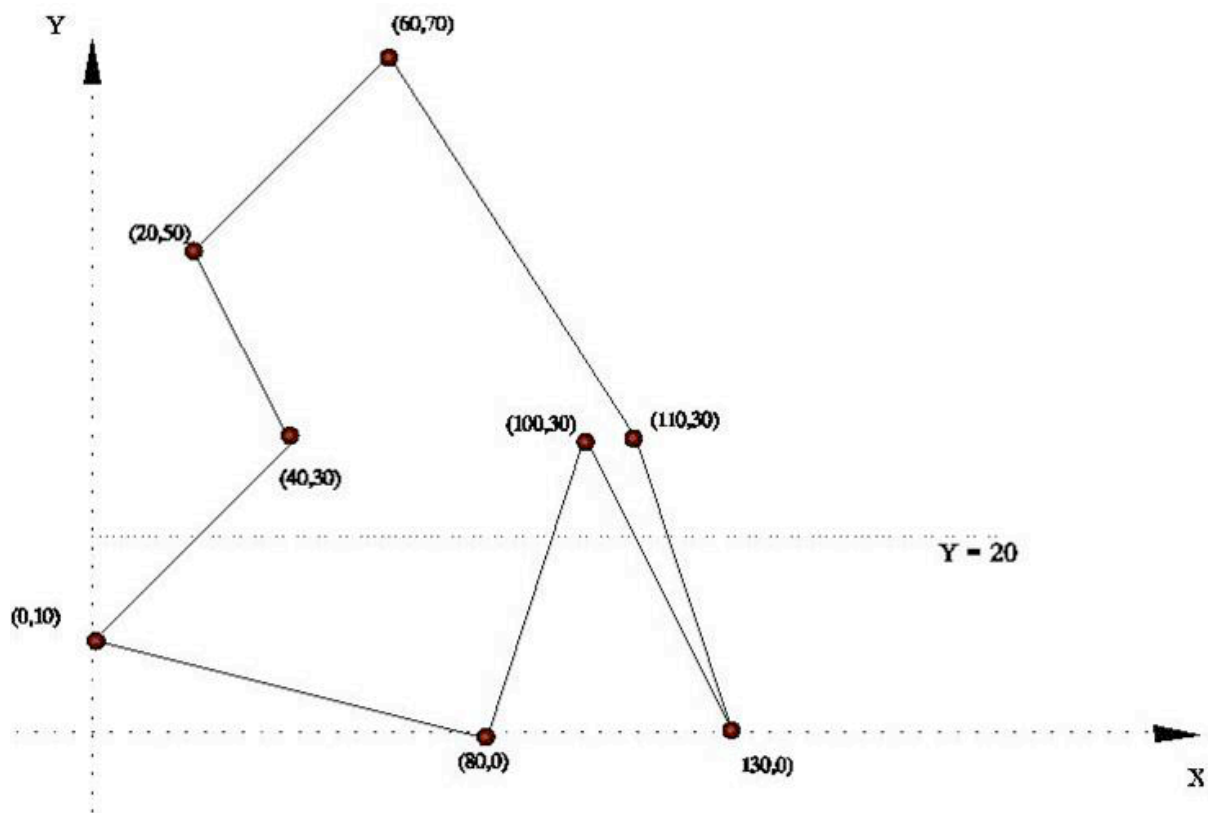


Figura 5.1 – Exemplo de um polígono com uma linha de varredura em $y = 20$.

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

LADO	Y_{\min}	Y_{\max}	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: $m = DY/DX \Rightarrow 1/m = DX/DY$

2º 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:

☐ $Y_{\text{varredura}} > Y_{\max}$

☐ $Y_{\text{varredura}} < Y_{\min}$

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

$$X = (1/M * (Y_{\text{varredura}} - Y_{\min}) + X_{\Rightarrow Y_{\min}});$$

3º 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, através das equações paramétricas de um círculo, desenhemos 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 **coseno**:

$$x = x_c + \text{raio} * \cos(\text{ang})$$

$$y = y_c + \text{raio} * \sin(\text{ang})$$

em que $\text{ang} = t * (2 \pi / n) \Rightarrow t = 0, 1, 2, \dots e n-1$

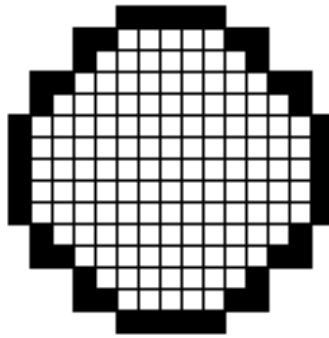
Diagrama:

$x = \text{raio};$ $y = 0$
Para t de 1 até 360 faça pixel (x , y , cor) $x = r \cdot \cos(\frac{p \cdot t}{180})$ $y = r \cdot \sin(\frac{p \cdot t}{180})$

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 a seguir mostra um possível resultado ao se criar uma circunferência pelas equações paramétricas.

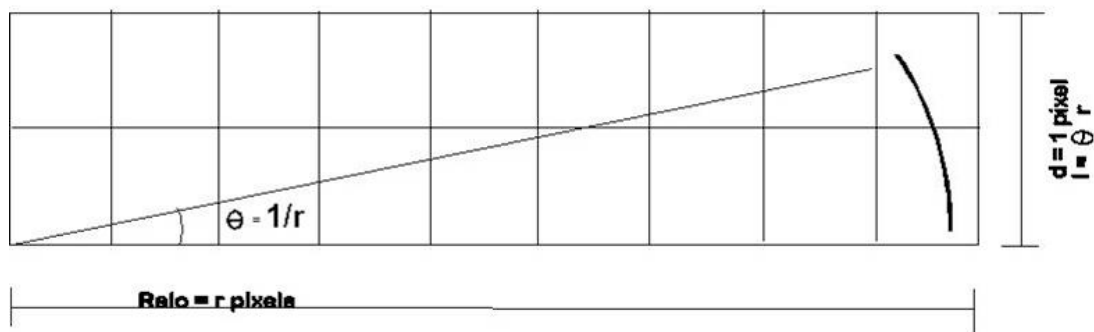


Criação de uma circunferência utilizando as equações paramétricas.

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:

Deslocamento em radianos = $1/r$ (r = raio)



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 45°. 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:

$$\begin{aligned}x_{n+1} &= x_n \cdot \cos q + y_n \cdot \sin q \\ y_{n+1} &= y_n \cdot \cos q - x_n \cdot \sin q\end{aligned}$$

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.

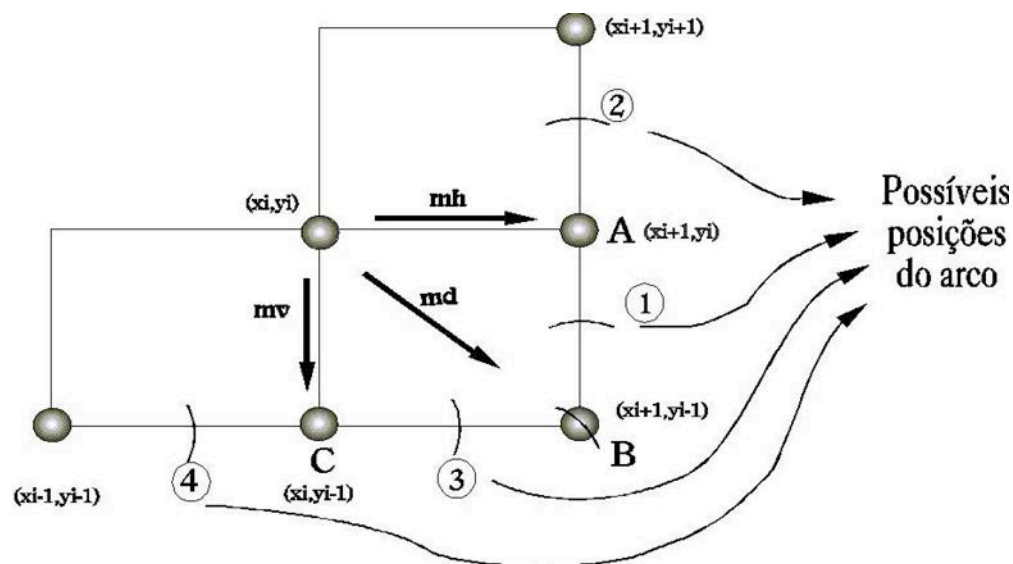
Para aplicar o algoritmo de Bresenham, utilizaremos a função de circunferência:

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

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 y_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 a seguir). 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.



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:

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

Dependendo do valor de i , temos os seguintes casos:

1º caso) Se $(i = 0) \Rightarrow$ o ponto B deve ser escolhido.

2º caso) Se $(i < 0) \Rightarrow$ 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 $d \leq 0$ o ponto escolhido é o A;

b) se $d > 0$ o ponto escolhido é o B;

3º 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 $t \leq 0$ o ponto escolhido é o B;

b) se $t > 0$ o ponto escolhido é o C;

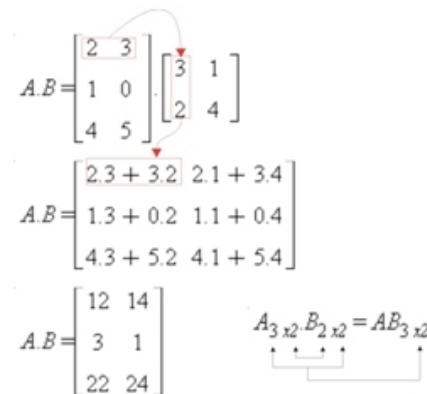
Por fim, o algoritmo de Bresenham é classificado como o melhor método para gerar uma circunferência, devido a sua eficiência, por isso é o mais utilizado nos pacotes de softwares gráficos.

7 - Transformações Geométricas

Inicialmente, para falarmos de transformações geométricas é necessário falarmos de matrizes. E, também falarmos das operações entre matrizes, que de forma geral resumirá-se a operação de multiplicação e adição de matrizes. Uma matriz é um arranjo de números dentro de colchetes ou parênteses, como os elementos 2, 0 da primeira linha e 1, 1 da segunda linha.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix}$$

Para que o produto exista, o número de colunas da primeira **matriz** tem que ser igual ao número de linhas da segunda **matriz**. Além disso, o resultado da **multiplicação** é uma **matriz** que possui o mesmo número de linhas da primeira **matriz** e o mesmo número de colunas da segunda **matriz**.



$$A.B = \begin{bmatrix} 2 & 3 \\ 1 & 0 \\ 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix}$$

$$A.B = \begin{bmatrix} 2.3 + 3.2 & 2.1 + 3.4 \\ 1.3 + 0.2 & 1.1 + 0.4 \\ 4.3 + 5.2 & 4.1 + 5.4 \end{bmatrix}$$

$$A.B = \begin{bmatrix} 12 & 14 \\ 3 & 1 \\ 22 & 24 \end{bmatrix}$$

$$A_{3 \times 2} \cdot B_{2 \times 2} = AB_{3 \times 2}$$

Portanto, as transformações geométricas são (alterações) operações matemáticas que permitem alterar uniformemente o aspecto de um desenho já armazenado no computador. Tais transformações permitem alterações uniformes de uma imagem definida sobre um sistema de coordenadas. Não há comprometimento da estrutura do desenho mas do aspecto que o mesmo assumirá. (mudança de orientação / escala).

Também é importante o conceito de matriz identidade, que transforma uma matriz em um elemento neutro, como mostra a notação a seguir.

$$A.I = \begin{bmatrix} 10 & 7 \\ -3 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 10 & 7 \\ -3 & 8 \end{bmatrix}$$

Observação: é interessante notar que as operações de transformação de visualização (WC/NDC) São combinações de transformações de escala (altera valores das coordenadas de modo proporcional) e de translação.

As transformações geométricas possuem três tipos fundamentais:

- ESCALA
- TRANSLAÇÃO
- ROTAÇÃO

7.1 Escala

Multiplicação de todas as coordenadas que definem o desenho por fatores de escala não nulos. Quando se aplica uma transformação de escala a um objeto, o resultado é um novo objeto semelhante ao original, porém “esticado” ou “encolhido”.

Desta forma, para definir um vetor de escala, dado por (S_x, S_y) , em que S_x é escala aplicada na direção x , e S_y é a escala aplicada na direção y . O “s” vem do inglês *scale*. Formalizando:

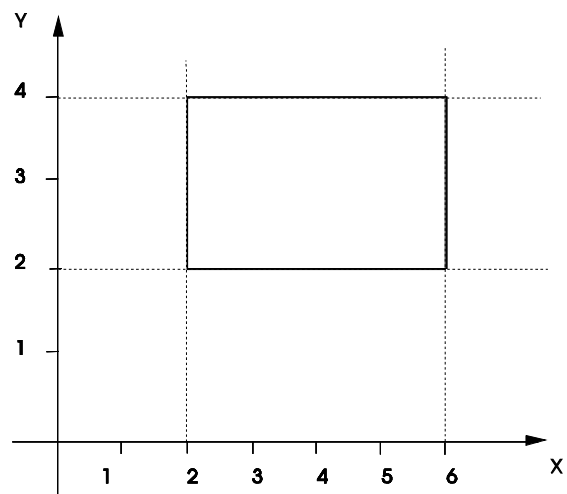
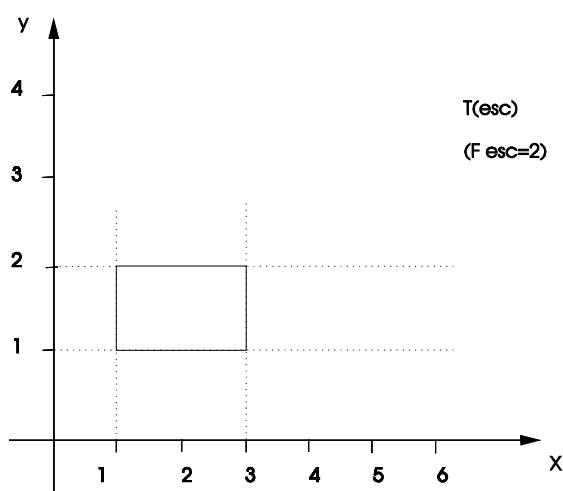
$$\begin{cases} x' = x * s_x \\ y' = y * s_y \end{cases}$$

Passando para a forma vetorial:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix}$$

Desta forma, temos como regra:

$$\text{obs: } \begin{cases} E > 1 \Rightarrow \text{Ampliação da imagem} \\ 0 < E < 1 \Rightarrow \text{redução da imagem} \\ E < 0 \Rightarrow \text{Espelhamento} \end{cases}$$



Obs: fatores E_x e E_y iguais \Rightarrow semelhança com o original

7.2 Translação

A translação é a movimentação do objeto para outra posição no sistema de coordenadas. Todos os pontos da imagem são deslocados de uma mesma distância em relação a posição anterior.

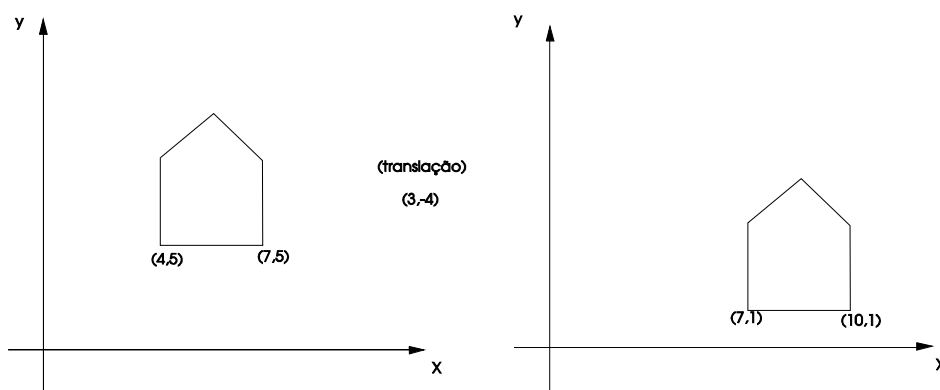
Formalizando:

$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$

Passando para a forma vetorial:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$

obs.: suponha uma linha constituída por um número muito grande de pontos, este processo pode consumir muito tempo. A frente, definiremos translação com somente pontos iniciais e finais de linha.



7.3 Rotação

Rotação é o ato de girar um objeto de um ângulo, num sistema de referência. Uma rotação em duas dimensões é aplicada pelo reposicionamento de cada ponto original do objeto ao longo de um círculo imaginário no plano xy. Para tanto devem ser especificados o ângulo de rotação Θ , que especifica o arco que um dado ponto deve ser deslocado e um centro de rotação (x_r, y_r) . Este ponto é considerado como *pivot* de rotação.

Movimentação de um objeto para uma outra posição, de forma que todos os pontos da imagem mantenham a mesma distância da origem que possuíam antes da transformação. Rotação em torno de um ponto qualquer: transformações de translação e de rotação em torno da origem.

Formalizando:

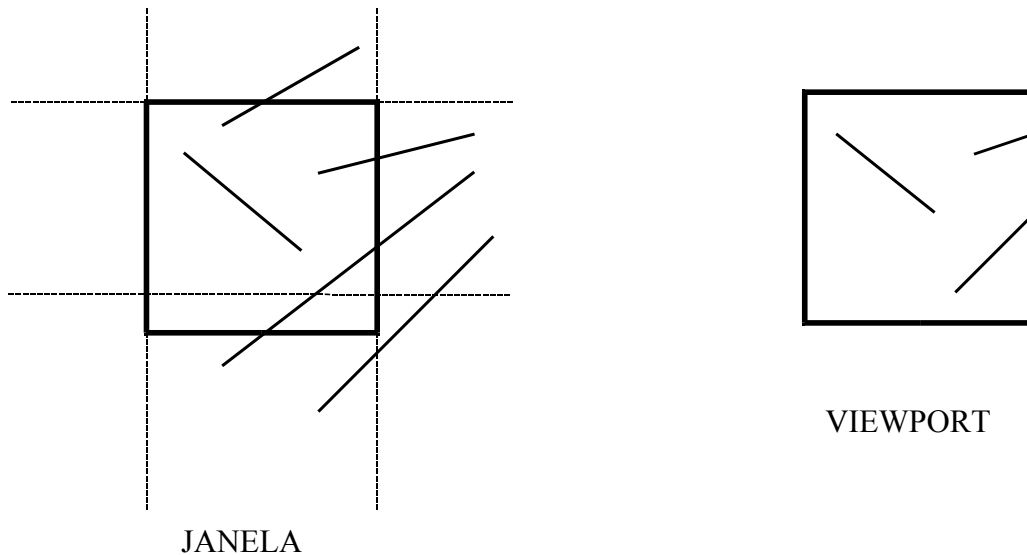
$$\begin{cases} x' = x * \cos \Theta - y * \sin \Theta \\ y' = y * \sin \Theta + x * \cos \Theta \end{cases}$$

Passando para a forma vetorial:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \Theta & -\operatorname{sen} \Theta \\ \operatorname{sen} \Theta & \cos \Theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

8 – Algoritmos de Recorte

Nesta seção discutiremos como desenhar segmentos de retas apenas quando estes segmentos estiverem dentro de um determinado “espaço”. Como mostra a figura a seguir em que um retângulo define os segmentos que devem aparecer, baseado no conceito de janela (window) e viewport.



Geralmente, qualquer procedimento que elimina aquelas partes da figura que estão tanto dentro quanto fora de uma região específica do espaço, é denominado como algoritmo de recorte.

Os algoritmos de recorte são aplicados em procedimentos de visualização 2D para identificar as partes de uma figura que estão dentro da janela de recorte. Tudo que está fora da janela de recorte é, então, eliminado da descrição da cena que será transferida para o dispositivo de saída, como o monitor. Um algoritmo eficiente redefinir os objetos que aparecem parcialmente na Janela. Isto reduz os cálculos, porque todas as matrizes de transformações geométricas podem ser concatenadas e aplicadas a descrição da cena antes do algoritmo de recorte ser carregado.

Nas próximas seções, serão descritos os seguintes algoritmos 2D:

- ☐ Recorte de pontos
- ☐ Recorte de linhas
 - Algoritmo de *Cohen-Sutherland*
- ☐ Recorte de Polígonos
 - Algoritmo de *Sutherland-Hodgman*

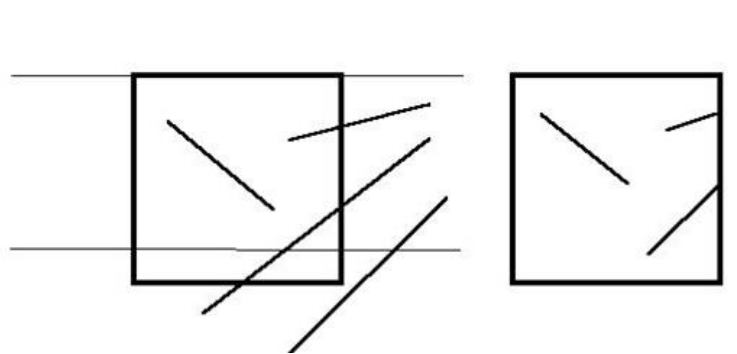
8.1 Pontos

Este algoritmo é um processo rápido e muito simples. O ponto $P = (x,y)$ que deve ser apresentado na *viewport* é aquele para o qual as inequações abaixo são satisfeitas.

$$\begin{cases} x_{min} \leq x \leq x_{max} \\ y_{min} \leq y \leq y_{max} \end{cases}$$

Se qualquer uma das quatro inequações não forem satisfeitas, o ponto P é recortado da viewport.

8.2 Linhas

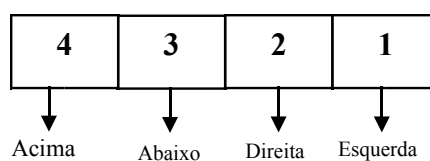


Este algoritmo exige mais cálculos e testes do que o processo anterior, embora seja necessário considerar apenas as partes finais da linha e não uma infinita quantidade de pontos. Um algoritmo de recorte de linhas processa cada linha em uma cena por meio de cálculos de interseções para determinar se toda linha ou parte dela será salva. A parte custosa deste algoritmo está no cálculo da interseção da linha com os limites da janela. Contudo, a principal finalidade de qualquer algoritmo de recorte de linha é minimizar os cálculos de interseção. Para isto, é realizado teste inicial na linha de forma a determinar se cálculos de interseção são realmente necessários. Inicialmente, o par de pontos finais pode ser checado para observar se ambos pertencem à janela, fazendo a linha ser aceita trivialmente. Ao mesmo tempo, a linha pode ser trivialmente rejeitada, por testes simples também, como no caso da linha CD, que apresenta $y > y_{max}$ (da janela). Podem ser, por exemplo, rejeitados pontos que estão abaixo de y_{min} e à esquerda de x_{min} ou à direita de x_{max} .

8.2.1 Algoritmo de Cohen-Sutherland

Este é um algoritmo projetado para identificar eficientemente que linhas podem ser trivialmente aceitas ou rejeitadas, dispensando os cálculos de interseções. Estes cálculos serão necessários apenas para linhas na qual os casos acima falharem.

Inicialmente, os pontos das extremidades da linha serão associados a um valor binário composto por 4 dígitos, chamado de código da região. Cada posição de bit é usado para indicar se o ponto está dentro ou fora de uma borda da janela de recorte. A figura a seguir abaixo ilustra uma possível ordenação para as posições dos bits, numerando de 1 à 4 da direita para a esquerda.



Então para esta ordem, os bits significam:

- Bit 1 - o ponto está à esquerda da janela - P(4)
- Bit 2 - o ponto está à direita da janela - P(3)
- Bit 3 - o ponto está à abaixo da janela - P(2)
- Bit 4 - o ponto está à acima da janela - P(1)

Desta forma, o primeiro passo deste algoritmo é codificar os extremos da linha a ser recortada. Para a obtenção deste código o plano XY é dividido em 9 partes por meio das bordas da Janela de Seleção (JS) (figura a seguir).

1001 (9)	0001 (1)	0101 (5)
1000 (8)	0000 (0)	0100 (4)
1010 (10)	0010 (2)	0110 (6)

O código dos pontos de cada uma das 9 áreas é formado por um número de 4 bits, da seguinte maneira:

1º bit :

em 1 se o ponto está à esquerda da JS.

em 0 se o ponto não está à esquerda da JS.

2º bit :

em 1 se o ponto está à direita da JS.

em 0 se o ponto não está à direita da JS.

3º bit :

em 1 se o ponto está abaixo da JS.

em 0 se o ponto não está abaixo da JS.

4º bit :

em 1 se o ponto está acima da JS.

em 0 se o ponto não está acima da JS.

A vantagem do uso deste tipo de codificação para os extremos da linha a ser recortada reside no fato de que é possível, mesmo antes de iniciar o cálculo das intersecções, tomar algumas decisões sobre a linha:

a) Decidir se a linha está **TODA DENTRO** da Janela de Seleção e desta forma pode ser exibida sem recorte. Para obter esta informação basta verificar se ambos os códigos tem valor 0000. Ou então fazer um OR dos dois códigos, se der 0, a linha está dentro.

b) Decidir se a linha está **TODA FORA** da Janela de Seleção e desta forma não ser exibida. Para obter esta informação basta realizar um AND dos códigos, se o resultado for diferente de 0 a linha está toda fora.

Nos casos em que nenhuma das providências acima puder ser adotada, o algoritmo prossegue nos seguintes passos (assuma que a linha é definida pelos pontos extremos P1 e P2):

1) Calcule os códigos de P1 e de P2;

```
Se P1 estiver fora da JS
    então siga para o passo 2
senão troque P1 com P2;
```

2) Verifique, pelo código de P1, se este encontra-se à esquerda da JS. (1º bit em 1).

```
Se não estiver siga para o passo 3;
Se estiver
    então calcule Pi, o ponto de intersecção da reta P1-P2 com o lado esquerdo da JS.
    coloque Pi em P1 (P1 := Pi)recalcule o código de P1
    siga para o passo 6;
```

3) Verifique, pelo código de P1, se este encontra-se à direita da JS. (2º bit em 1).

```
Se não estiver siga para o passo 4;
Se estiver
    então calcule Pi, o ponto de intersecção da reta P1-P2 com o lado direito da J.S.
    coloque Pi em P1 (P1 := Pi), recalcule o código de P1
    siga para o passo 6;
```

4) Verifique, pelo código de P1, se este encontra-se acima da JS. (3º bit em 1).

```
Se não estiver siga para o passo 5;
Se estiver
    então calcule Pi, o ponto de intersecção da reta P1-P2 com o lado de cima da J.S.
    coloque Pi em P1 (P1 := Pi), recalcule o código de P1
    siga para o passo 6;
```

5) Verifique, pelo código de P1, se este encontra-se abaixo da JS. (4º bit em 1).

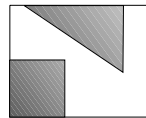
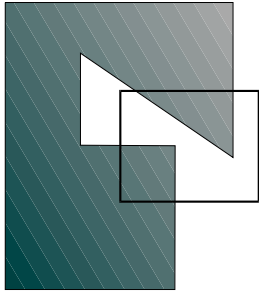
```
Se não estiver siga para o passo 6;
Se estiver
    então calcule Pi, o ponto de intersecção da reta P1-P2 com o lado de baixo da JS.
    coloque Pi em P1 (P1 := Pi), recalcule o código de P1
    siga para o passo 6;
```

6) Verifique se a nova linha P1-P2 está toda dentro ou toda fora da J.S.

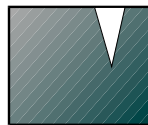
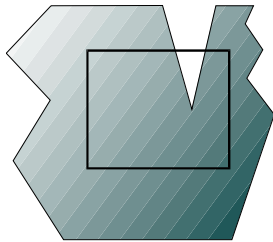
```
Recalcule o código de P1;
Se (P1 OR P2) = 0
    então linha recortada está em P1-P2
    encerra o algoritmo.
senão Se (P1 AND P2) <> 0
    então linha está toda fora da J.S;
    encerra o algoritmo.
senão volta ao passo 1.
```

8.3 Polígonos

Em algumas aplicações, há necessidade de promover o recorte de polígonos, cujos vértices estão armazenados numa estrutura de dados qualquer. Para serem exibidos, os polígonos devem primeiro passar por uma operação de transformação de visualização e depois por um processo de recorte até serem convertidos nas coordenadas do equipamento. O algoritmo que recorta polígonos deve prever diferentes casos, tais como:



Polígono côncavo é recortado em 2 polígonos separados e distintos.

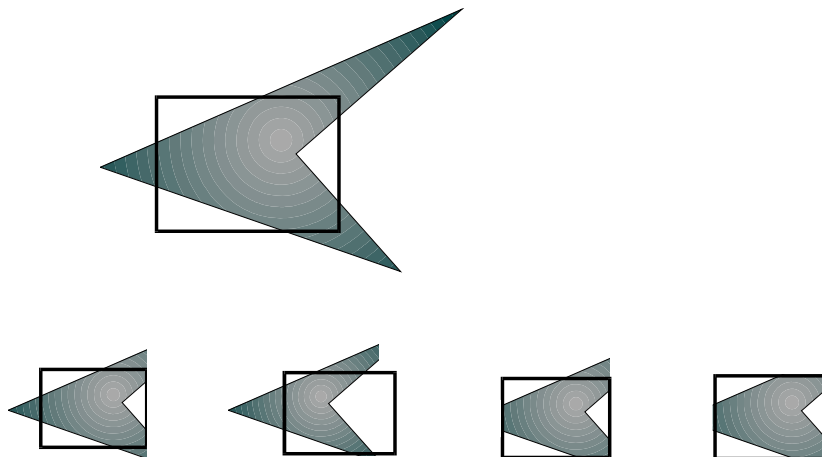


Recorte de polígono (côncavo) que está quase integralmente na janela

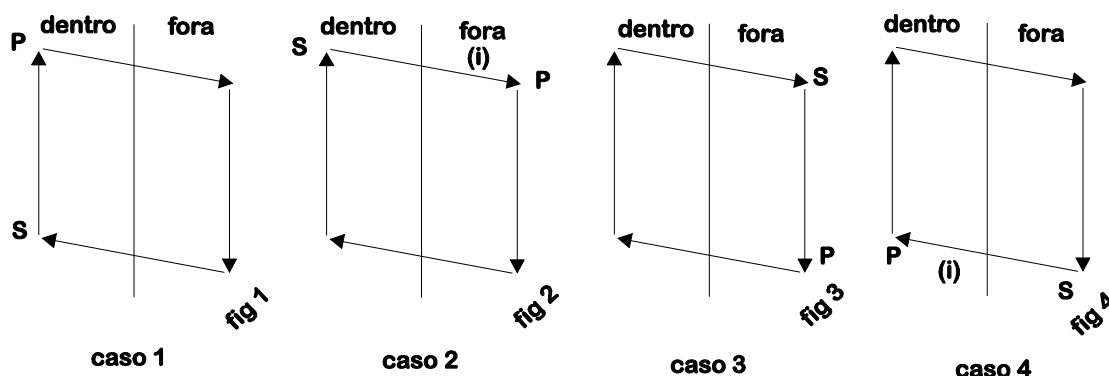
8.3.1. Algoritmo de Sutherland-Hodgman

Estratégia de solução de uma série de problemas simples e idênticos que resolvem o problema quando combinados: Recortar um polígono por meio do recorte de suas laterais (áreas que tocam os limites da janela)

Serão feitos 4 recortes:



Com o polígono de lados definidos pelos vértices: $U_1, U_2, U_3, \dots U_n$
 Para cada lado observa-se a relação entre vértices sucessivos e as janelas (limites)
 lados definidos pelos vértices da lista de saídas serão apresentados na tela. Casos possíveis:



CASO 1: dois vértices é adicionado à lista de saídas (p, no caso)

CASO 2: o ponto “i” de interseção é tratado como um vértice de saída (a ser traçado)

CASO 3: os dois vértices são descartados.

CASO 4: os dois pontos “i” e “p” são colocados na lista de vértices de saída.

Algoritmo destinado a obter as interseções

- O primeiro ponto não é colocado na lista de saídas, já que o mesmo é o vértice inicial e já se encontra na mesma, pois o processo é sequencial.
- Para linhas do polígono totalmente invisível, nenhum ponto é adicionado à lista de saídas (caso 3).
- para casos 2 e 4, é necessário calcular as interseções
- para um dado vértice, se necessário calcular se o mesmo está dentro ou fora de uma janela, temos uma função que aplica um teste baseado em produto vetorial, como:

$$\vec{P}_1 \cdot \vec{P}_2 \times \vec{P}_1 \cdot \vec{P}_3 = X$$

$$\vec{P}_1 \cdot \vec{P}_2 \times \vec{P}_1 \cdot \vec{P}_4 = .$$

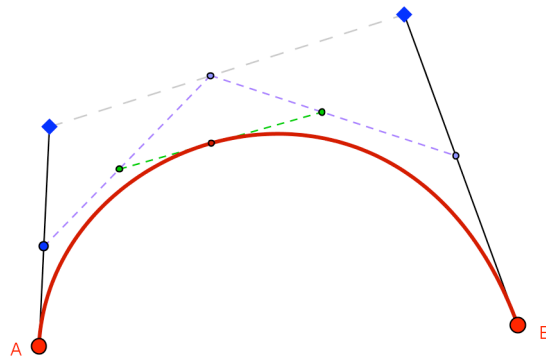
Faz-se o produto vetorial de $\vec{P}_1 \cdot \vec{P}_2 \times \vec{P}_1 \cdot \vec{P}_3$, caso o mesmo resultado em vetor entrando (módulo negativo) o ponto está do lado de dentro da janela, se o mesmo for positivo, o ponto está do lado de fora da janela.

O módulo do produto vetorial de dois vetores $V(V_x, V_y)$ e $W(W_x, W_y)$ é um vetor cuja magnitude é dada por $V_x W_y - V_y W_x$. Se este n° for positivo \Rightarrow ponto fora. Se este n° for negativo \Rightarrow ponto dentro.

9 – Curvas

9.1 Curvas de Bézier

Entre os algoritmos existente para geração de curvas, um é considerado prático e elegante e tem como base a especificação de quatro pontos básicos que determinam totalmente um segmento de curva: dois pontos extremos e dois pontos de controle. Na figura a seguir temos os pontos extremos A e B, os pontos de controles (em azul) e a curva em vermelho.



O método para desenhar esta curva é relativamente simples, pois baseia-se em recursão. Como mostra a figura a seguir, basta calcular seis pontos médios a saber:

$b_{1,1}$ – ponto médio entre P_1, P_2

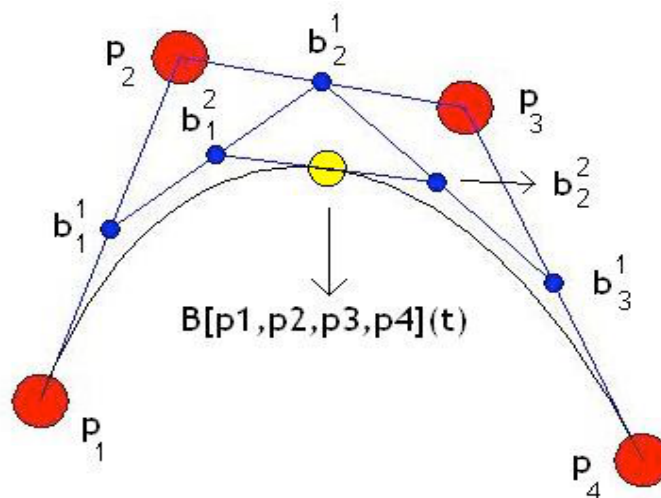
$b_{1,2}$ – ponto médio entre P_2, P_3

$b_{1,3}$ – ponto médio entre P_3, P_4

$b_{2,1}$ – ponto médio entre $b_{1,1}$ e $b_{1,2}$

$b_{2,2}$ – ponto médio entre $b_{1,2}$ e $b_{1,3}$

B – ponto médio entre $b_{2,1}$ e $b_{2,2}$ (em amarelo – $B[p_1, p_2, p_3, p_4]$)



Após a definição destes pontos médios, podemos separar as tarefas em duas mais simples:

- desenhar a curva P1-B, com pontos de controle b1,1 b2,1
- desenhar a curva B-P4, com pontos de controle b2,2 b1,3

Uma notação que pode ser usada é o produto de um vetor linha mais simples, $[t^3 \ t^2 \ t \ 1]$, por uma matriz 4x4, obtendo o seguinte resultado para a curva de Bézier:

$$B(t) = [t^3 \ t^2 \ t \ 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & 3 & -3 & 1 \\ -3 & 3 & -3 & 1 \\ 1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}.$$

Portanto, se executarmos a multiplicação A (primeira matriz – vetor linha) por BC (segunda e terceira) teremos o seguinte resultado:

$$B(t) = (-P_0 + 3P_1 + 3P_2 + P_3)t^3 + 3(P_0 - 2P_1 + P_3)t^2 - 3(P_1 - P_0)t + P_0$$

Esta será a base para construção do algoritmo de geração de curvas de Bézier.

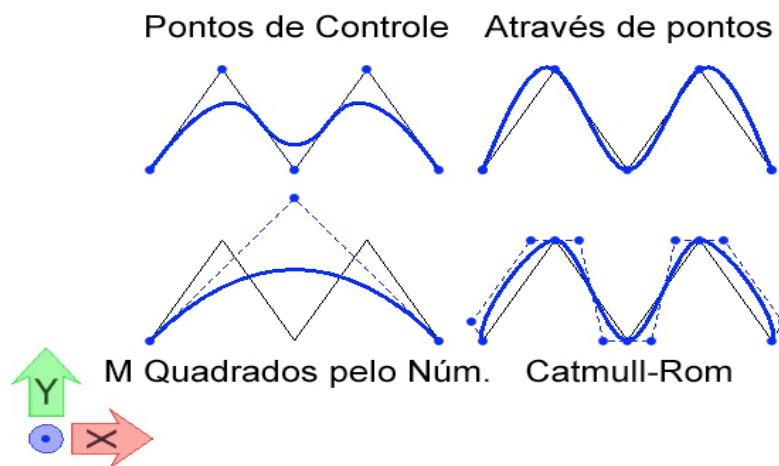
9.1 Curvas B-Spline

Além da técnica de Bézier, existem outras formas de curvas $x = f(t)$, $y = g(t)$, em que f e g são polinômios de terceiro grau em t . Uma técnica intitulada como B-spline, tem uma característica de que a curva geralmente normalmente não passa pelos pontos dados, os pontos de controle.

Ao contrário de curvas de ponto, há uma série de métodos, que podem ser escolhidos no menu opção de método, para calcular a curva final que resulta.

Método	Definem pontos de dados ou vértices do elemento
Pontos de controle	Vértices do polígono controle.
Através de pontos	Pontos na curva.
Mínimos quadrados por tolerância e mínimos quadrados pelo número	Um conjunto de pontos ao qual a curva se aproxima ou ao qual está "apto".
Catmull-Rom	Um conjunto de pontos que é bem aproximado.

Estas ilustrações mostram os diferentes tipos de curvas B-spline, construídas a partir da mesma cadeia de caracteres de linha.



A técnica B-splines facilita o desenho de curvas suaves que consistem em muitos segmentos de curva. Para evitar confusão, cada segmento de curva consiste em muitos segmentos de reta.

Assim, como na técnica de Bézier, uma notação que pode ser usada tendo a variável t (de 0 a 1) para cada segmento de curva é a seguinte, porém tendo como referência a quantidade de pontos definida para o exemplo (6).

$$B(t) = \frac{1}{6} \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & 3 & -3 & 1 \\ -3 & 3 & -3 & 1 \\ 1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}.$$

Portanto, se executarmos a multiplicação A (primeira matriz – vetor linha) por BC (segunda e terceira) teremos o seguinte resultado:

$$B(t) = \frac{1}{6} (-P_0 + 3P_1 - 3P_2 + P_3)t^3 + \frac{1}{2} (P_0 - 2P_1 + P_2)t^2 - \frac{1}{6} (P_0 + 4P_1)t + P_2$$

Esta será a base para construção do algoritmo de geração de curvas B-splines.