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.

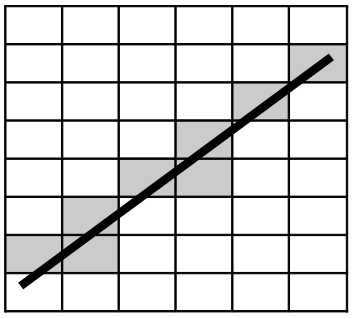


Figura 4.1 – Conversão da representação de uma reta na forma vetorial para a matricial.

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)*.

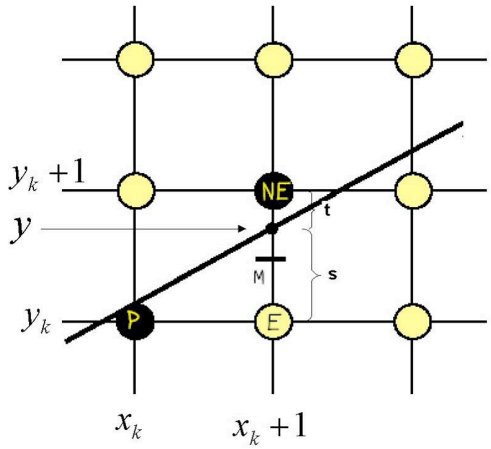
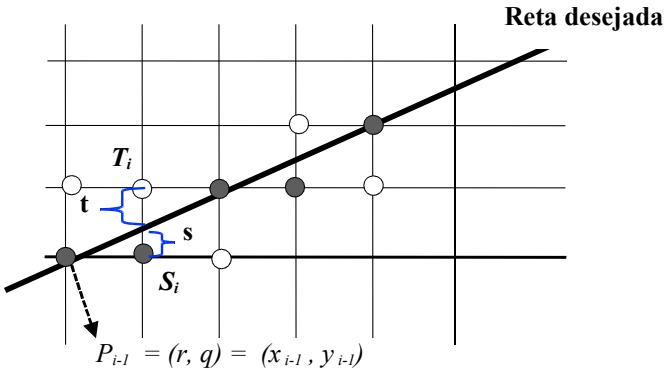


Figura 4.2 – Representação do algoritmo de Bresenham para geração de uma reta.

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;}
      }
}
```