Rasterização de Retas para Formação de um Triângulo

Ana Gabriela, Bertrand Lira, Emanuel Souza

Março de 2023

1 Introdução

O trabalho tem como objetivo aplicar os conhecimentos de rasterização para a implementação das funções: PutPixel(acende um pixel na tela), DrawLine(desenha uma linha na tela utilizando a função PutPixel) e DrawTriangle(desenha um triângulo utilizando a função DrawLine para desenhar suas arestas). O algoritmo apresentado baseia-se em uma técnica de rasterização de primitivas, especificamente retas, cuja definição está relacionada ao processo de conversão de objetos gráficos - vetoriais - em uma imagem digital formada por elementos finitos de formato essencialmente retangular, os pixels, o que implica em uma conversão de elementos analógicos da reta com coordenadas não necessariamente inteiras para elementos digitais e inteiros e bem definidos dentro de uma escala de valores associados um a um. A rasterização decide quais pixels vão ser cobertos pelo elemento, aplicando ilumincação e textura.

2 Metodologia

O processo de tratamento especificamente de retas foi altamente considerado ao implementar o algoritmo, visto que a rasterização foi feita visando à modelagem desse tipo de elementos gráfico. Sendo assim, foi definido o eixo das abcissas de um plano cartesiano como o eixo **x**, horizontal, enquanto o das ordenadas foi associado como eixo **y**, vertical. O coeficiente linear da reta, que indica, além de seu ângulo de inclinação com a horizontal, a variação dos valores de y para cada x iterado. É importante ressaltar que houve a iteração desses valores no *frame buffer*, uma região de memória sequencial utilizada para armazenar temporariamente as informações que serão exibidas na tela, cujos valores vão aumentando(de forma gráfica) da esquerda para a direta e de cima para baixo, caracterizando operações análogas ao quarto

quadrante de um plano cartesiano, porém, funcionando, a título de abstração e código, como o primeiro quadrante espelhado para baixo.

2.1 PutPixel()

```
21  void PutPixel(int x, int y, int r, int g, int b){
22      int offset = x * 4 + y * 2048; //Calcula o offset
23      FBptr[offset] = r; //Valor da cor vermelha
24      FBptr[offset+1] = g; //Valor da cor verde
25      FBptr[offset+2] = b; //Valor da cor azul
26      FBptr[offset+3] = 255; //Valor da alfa, sempre estabelecido como máximo
27  }
```

A função PutPixel() é responsável por acender um pixel na tela e recebe cinco valores inteiros como parâmetros. Os dois primeiros são os valores de x e y, correspondentes ao posicionamento do pixel na tela, enquanto os três últimos são os valores das cores vermelha, verde e azul que combinadas formam a cor do pixel.

Para tal função, foi necessário o conhecimento sobre o funcionamento do *frame buffer*, que armazena as informações de cor de cada pixel da imagem que está sendo exibida na tela. Geralmente, o *frame buffer*, é organizado em uma grade de pixels, onde cada pixel é representado por um conjunto de valores que representam a cor e a transparência do pixel. A localização de um pixel na grade é determinada por sua posição x e y na imagem. Assumindo que a imagem é armazenada no frame buffer como uma matriz bidimensional, o offset de um pixel pode ser calculado da seguinte maneira:

$$\text{offset} = \text{x} * \text{bytesPorPixel} + \text{y} * \text{bytesPorPixel} * \text{larguraDaImagem}$$

Nos nossos exemplos, a imagem possui 512 pixels de largura e 512 pixels de altura, e cada pixel é representado por 4 bytes (RGBA). Logo, para descobrir o offset do pixel numa posição (x, y) seria calculado da seguinte maneira:

$$\text{offset} = \text{x} * 4 + \text{y} * 4 * 512$$
$$\text{offset} = \text{x} * 4 + \text{y} * 2048$$

Encontrado o offset, primeiro byte do pixel e valor do vermelho, basta setar os valores das cores em sequência, incrementando o valor do offset de um em um. Para fins de visibilidade, o valor de alfa foi definido sempre no máximo, como mostrado na linha 26.

2.2 DrawLine()

A função DrawLine() foi implementada através da utilização do algoritmo de Bresenham, cujas operações são baseadas em incrementação de valores inteiros, permitindo, assim, uma melhor otimização. O algoritmo implementado foi baseado no incremento do erro para o cálculo do pixel a ser aceso.

Inicialmente, tem-se a troca de variáveis e a utilização de uma *flag* booleana para uma ação equivalente.

```
48      bool steep = abs(y1 - y0) > abs(x1 - x0); //verifica se dy > dx
49
50      if (steep == true){ //se dy > dx, coeficiente angular > 1, ent troca o eixo
51          int aux = x0;
52          x0 = y0;
53          y0 = aux;
54          aux = x1;
55          x1 = y1;
56          y1 = aux;
57      }
```

Isso se deve ao fato da lógica de incremento de erro funcionar apenas para retas que se encaixam nas características das retas que se encontram no primeiro octante de um plano cartesiano, passando pela - ou tendo sua extensão indo de encontro a - origem, ou seja, linhas cujo coeficiente linear se encontra entre 0 e 1 e seu valor inicial no eixo das abcissas ser maior que o seu valor final no mesmo eixo.

Esse tratamento é feito tendo em vista as características de cada octante, como o valor inicial e final do eixo das ordenas, abcissas e os valores de seu coeficiente linear, que indica se a reta possui inclinação maior ou menor - ou igual - que 45 graus, e, a partir disso, adequando-os aos valores do primeiro octante. A variável booleana é responsável pela mudança de eixo de referência, de forma que a inclinação da reta seja sempre de 45 ou menos graus com relação a tal, apenas invertendo os valores de x ou y e acendendo os pixels correspondentes no *frame buffer*, sendo seus parâmetros trocados com base na necessidade de inversão de eixo. Já para o caso onde os valores iniciais do eixo das abcissas estão em ordem invertida, basta trocar os valores das variáveis, e para o mesmo caso no eixo das ordenas, o tratamento é feito pela variável **dy**, que vai indicar incremento ou decremento no valor de **y**, baseado em qual - incial ou final - é maior.

```
59      if(x0 > x1){
60          int aux = x0;
61          x0 = x1;
62          x1 = aux;
63          aux = y0;
64          y0 = y1;
65          y1 = aux;
66      }
67
68      if (y0 > y1){
69          dy = -1;
70      }else {
71          dy = 1;
72      }
```

Para o loop do cálculo dos pixels acendidos, há a iteração pela coordenada x, devidamente tratada anteriormente para apresentar as características do primeiro octante, que apresenta um valor y correspondente calculada pela incrementação do erro. O valor do erro é, inicialmente, 0, devido ao fato que o primeiro pixel sempre se encontrará na reta e, a partir do valor incrementado a ele, o algoritmo decide pelo incremento - ou decremento - associado ao y. O funcionamento da variável erro se baseia no quanto uma variável muda em relação a outra, o que pode fazer a reta cruzar um ponto que não pode ser associado a um pixel, pois este valor não é inteiro, logo ao acender um pixel(o qual possui, necessariamente, um valor inteiro associado a ele pela lógica do código), um erro é gerado e vai se acumulando. No tocante a essa situação, o algoritmo faz o cálculo correto para que esse erro não venha a indicar - consequentemente, acendendo - um pixel que não deveria estar associado àquela coordenada especifica, funcionando como uma variável de decisão para o acendimento de um pixel a nordeste ou a leste do atual a partir da modificação da variável y para o x iterado. Todos esses valores são baseado na modelagem matemática de uma reta, levando em consideração seu coeficiente linear, seus valores associados de x e y, e sua equação fundamental, que indica se o pixel aceso estará acima ou abaixo de onde a reta passa baseado no seu ponto médio.

```
80      for(int x = x0; x < x1; x++){
81          if (steep){
82              PutPixel(y, x, r, g ,b);
83          }else{
84              PutPixel(x, y, r, g ,b);
85          }
86
87          erro = erro + delta_y;
88          if (erro*2 >= delta_x){
89              y = y + dy;
90              erro = erro - delta_x;
91          }
92      }
```

2.3 DrawTriangle()

```
151  void DrawTriangle(int xa, int ya, int xb, int yb, int xc, int yc, int r, int g, int b)
152  {
153      DrawLine(xa,ya,xb,yb,r,g,b); //Desenha a primeira aresta através do DrawLine()
154      DrawLine(xa,ya,xc,yc,r,g,b); //Desenha a segunda aresta através do DrawLine()
155      DrawLine(xb,yb,xc,yc,r,g,b); //Desenha a terceira aresta através do DrawLine()
156  }
```

Já o DrawTriangle() recebe os valores de x e y de cada um dos três vértices do triângulo, além dos valores das cores vermelha, verde e azul que formarão a cor dos lados. Os valores dos vértices são passados em duplas para o DrawLine() para formar as arestas que se conectam e formam o triângulo.

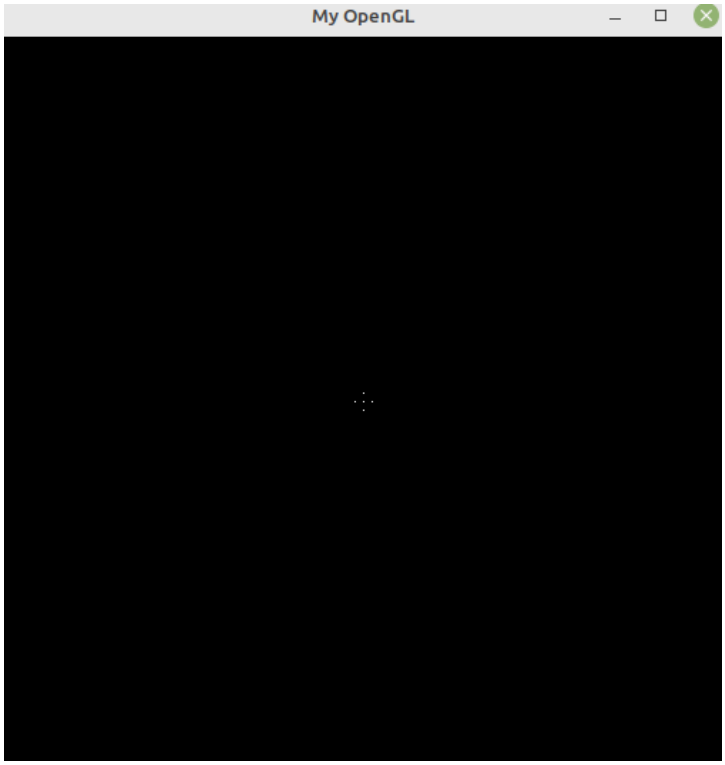
3 Resultados obtidos

Analisando as funções geradas, foram realizadas algumas chamadas dessas funções a fim de se obter seus resultados(suas saídas).

Primeiro, foi analisada a função PutPixel() com as seguintes chamadas:

```
 9   void MyGlDraw(void)
10   {
11       PutPixel(256,256,255,255,255);
12       PutPixel(250,256,255,255,255);
13       PutPixel(262,256,255,255,255);
14       PutPixel(256,250,255,255,255);
15       PutPixel(256,262,255,255,255);
16   }
```

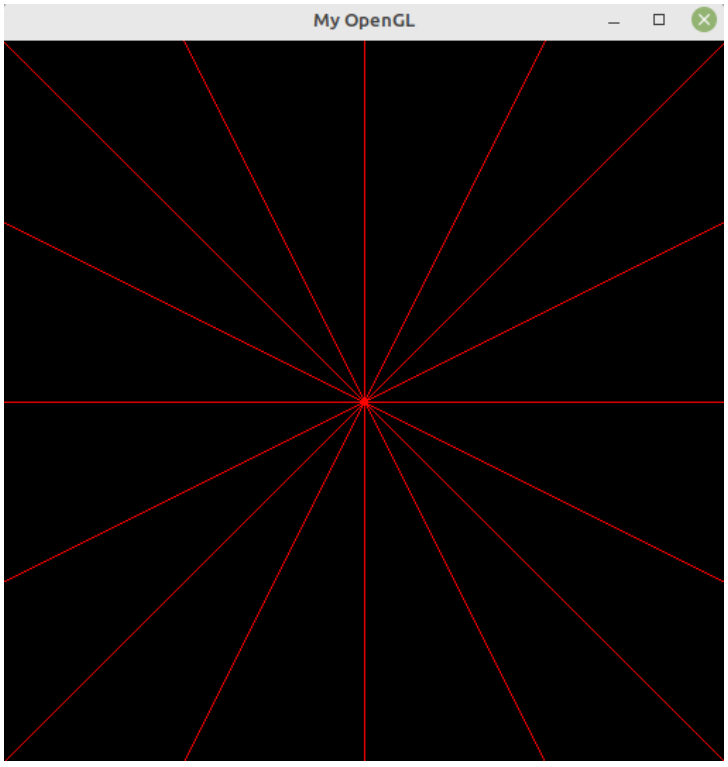
No código, é criado um pixel de "referência", o pixel(256,256) no meio da tela, apenas para fins de análises. Em seguida, foi criado mais 4 pixels em volta desse "pixel de referência". O resultado encontrado foi este:



Em seguida, foram feitas chamadas do DrawLine():

```
9   void MyGlDraw(void)
10  {
11      DrawLine(256,0,256,512,255,0,0);
12      DrawLine(0,256,512,256,255,0,0);
13      DrawLine(0,0,512,512,255,0,0);
14      DrawLine(0,512,512,0,255,0,0);
15      DrawLine(128,0,384,512,255,0,0);
16      DrawLine(384,0,128,512,255,0,0);
17      DrawLine(0,128,512,384,255,0,0);
18      DrawLine(0,384,512,128,255,0,0);
19  }
```

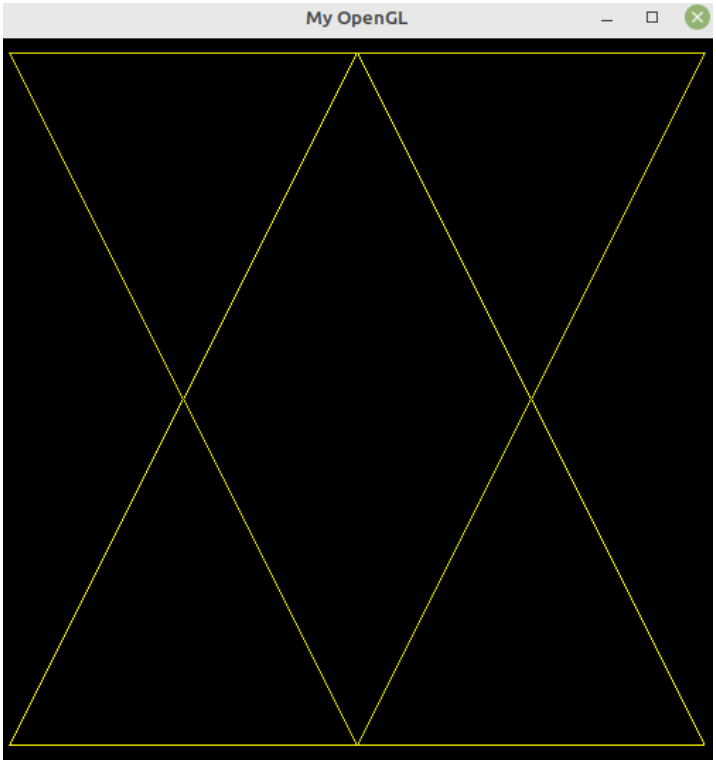
As chamadas das linhas passam por todos os quadrantes. Ao todo foram 8 chamadas, passando como parâmetros da função as coordenadas do ponto inicial e final, além das cores das linhas. O resultado encontrado foi este:



Por fim, foram feitas duas chamadas da função DrawTriangle():

```
 9  void MyGlDraw(void)
10  {
11      DrawTriangle(10,500,256,10,502,500,255,255,0);
12      DrawTriangle(10,10,502,10,256,500,225,225,0);
13  }
```

A primeira chamada cria o triângulo com vértices nas coordenadas (10,500), (256,10) e (502,500), exibindo na tela um triângulo normal. Já a segunda chamada, cria o triângulo com vértices (10,10), (502,10) e (256,500), exibindo um triângulo com medidas iguais à primeira chamada, mas espelhado para baixo. O resultado encontrado foi este:



Inicialmente, o algoritmo feito passa individualmente as coordenadas dos pontos e os valores RGB de cada pixel como parâmetros para as funções. Para possíveis melhorias do algoritmo, poderia ser implementada alguma estrura de dados, como por exemplo um array, para a passagem desses parâmetros. Além disso, poderia ser feito um maior tratamento das cores e da intensidade dos pixels.

Referências

Gainsbury, S.M., 2015. **Online gambling addiction: the relationship between internet gambling and disordered gambling**. Current addiction reports, 2(2), pp.185-193.

SWI Prolog: Getting tarted quickly. https://www.swi-prolog.org/pldoc/man?section=quickstart. Acesso em: 7 de jun de 2022.