

Leandro Lucas de Oliveira Bandeira
Rasterização de linhas

LEANDRO BANDEIRA

July 2024

1 Introdução

A rasterização de linhas é o método de conversão entre representações matriciais e vetoriais. Sabemos que cada pixel está localizada em uma posição (x, y) na tela, ou seja, dado um pixel inicial (x_i, y_i) e o pixel final por (x_f, y_f) , qual o conjunto de pixels entre eles que compõe todo o segmento? Para descobrir esse conjunto de pixels, é utilizado algoritmos de rasterização de linhas, para esse projeto foi utilizado o algoritmo de Bresenham que será explicado na seção 3.

2 Rasterização de Ponto

Antes de começar a rasterização de linha, nosso algoritmo deve ser capaz de imprimir pontos na tela. Para isso devemos ter o conceito de frame buffer enraizado. Todo pixel ocupa 4 bytes na tela, sendo cada byte respectivamente alocado a Red, Green, Blue e ao canal alfa. Um exemplo de frame buffer está representado na imagem 1.

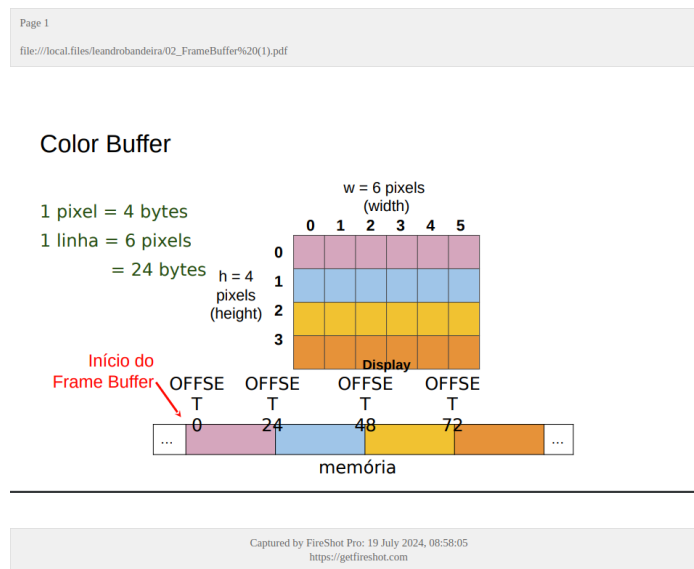


Figura 1: Exemplo de Frame Buffer

Como então devemos identificar o pixel na posição $(4,2)$? Cada valor horizontalmente são separados por 4 bytes, então temos $4 * 4 = 16$, porém,

cada linha é composta por 24 bytes, logo $24 * 2 = 48$, $48 + 16 = 64$, essa é posição no frame buffer daquele pixel. Na nossa tela, os pixels são representados conforme a imagem 2, sendo assim para realização do cálculo da posição inicial do pixel no frame buffer, será dado por: $x * 4 + y * 4 * \text{IMG-WIDTH}$. A implementação da função *PutPixel* está esquematizado na figura 3.

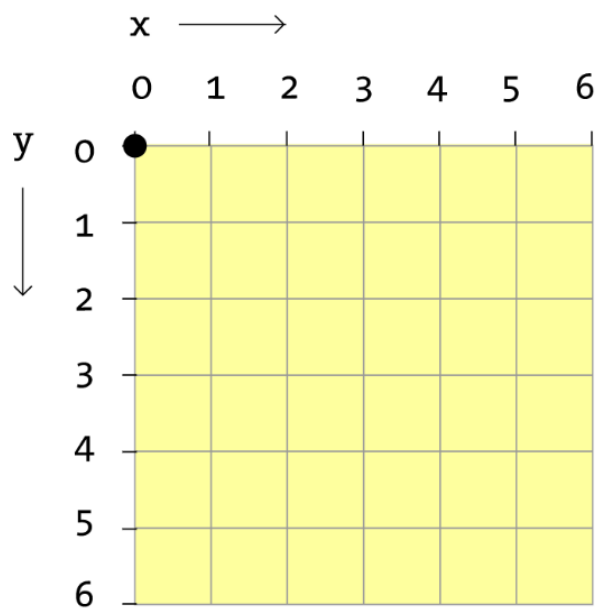
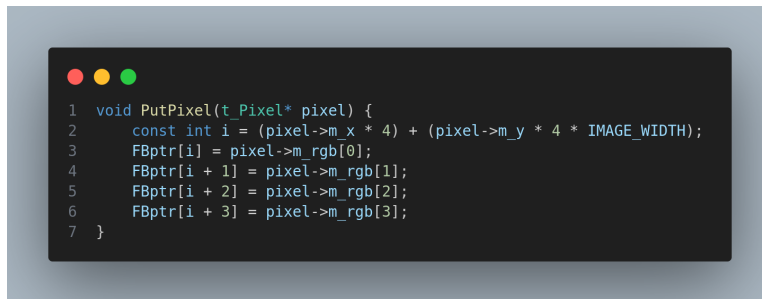


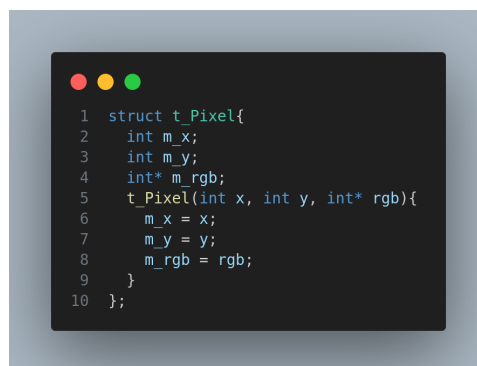
Figura 2: Exemplo de plano cartesiano

A função *PutPixel* foi implementada utilizando um ponteiro para a posição inicial do Frame Buffer. Para isso, foi necessário apenas criar uma variável que indique a posição inicial do buffer, utilizando o cálculo explicado anteriormente. A estrutura de dado nomeada pixel indica exatamente como ela se chama, nela armazenamos a posição (x, y) do pixel na tela e seu canal RGB e alfa. A estrutura do pixel está representado na imagem 4. Desse modo, podemos gerar o resultado da função *PutPixel*, printando vários pontos na tela, como visto na imagem 5.



```
1 void PutPixel(t_Pixel* pixel) {
2     const int i = (pixel->m_x * 4) + (pixel->m_y * 4 * IMAGE_WIDTH);
3     FBptr[i] = pixel->m_rgb[0];
4     FBptr[i + 1] = pixel->m_rgb[1];
5     FBptr[i + 2] = pixel->m_rgb[2];
6     FBptr[i + 3] = pixel->m_rgb[3];
7 }
```

Figura 3: Implementação PutPixel



```
1 struct t_Pixel{
2     int m_x;
3     int m_y;
4     int* m_rgb;
5     t_Pixel(int x, int y, int* rgb){
6         m_x = x;
7         m_y = y;
8         m_rgb = rgb;
9     }
10 };
```

Figura 4: Implementação struct Pixel

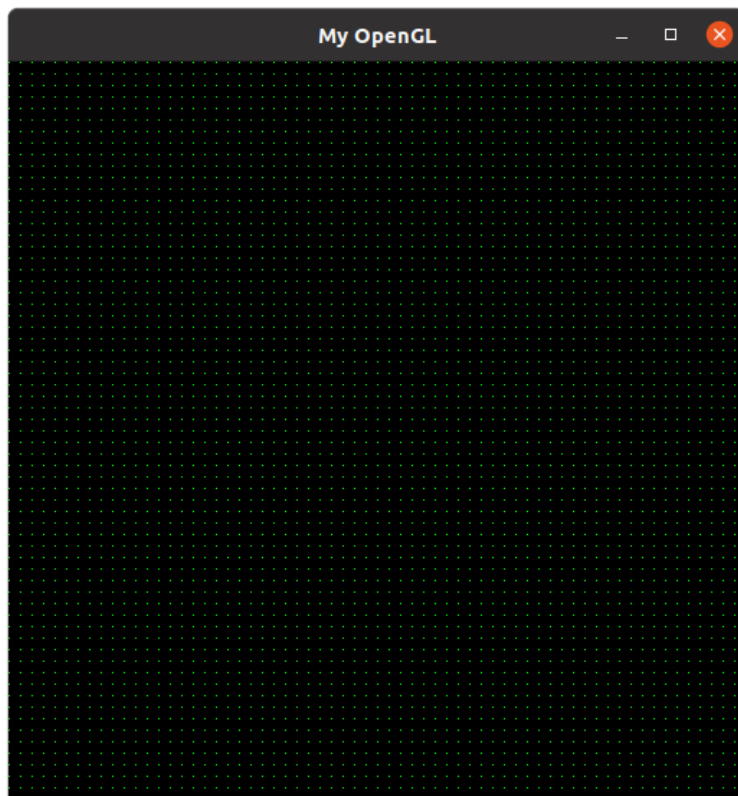


Figura 5: Resultados da função PutPixel

3 Rasterização de linha

Como dito anteriormente, para a implementação da rasterização de linha foi utilizado o algoritmo de Brsenham. O algoritmo pode ser visto na figura 6. Como a ideia é generalizar o algoritmo para o octeto, devemos alterar o algoritmo como se segue:

Algorithm 1 Algoritmo de Brsenham

```
 $dy \leftarrow -|yf - yi|$ 
 $dx \leftarrow |xf - xi|$ 
 $x \leftarrow xi$ 
 $e \leftarrow dx - dy$ 
while True do
     $Pixel \leftarrow x, y, rgb$ 
    if  $x == xf \wedge y == yf$  then Break
    end if
     $e2 \leftarrow 2 * e$ 
    if  $e2 \geq dy$  then
        if  $x == xf$  then Break
        end if
         $e \leftarrow e + dy$ 
         $x \leftarrow xi \geq xf ? x + 1 : x - 1$ 
    end if
    if  $e2 \leq dx$  then
        if  $y == yf$  then Break
        end if
         $e \leftarrow e + dx$ 
         $y \leftarrow yi \geq yf ? y + 1 : y - 1$ 
    end if
end while
```

Para análise do algoritmo acima, perceba que devemos somar ou subtrair as posições de x e y dependendo da relação entre os pontos, caso o ponto inicial seja menor que o final, vamos incrementá-lo em uma unidade, caso não, vamos decrementá-lo na mesma quantia. Além disso, é importante notar a relação do erro com a diferença entre os pontos x e y. Só alteramos o valor de x, caso o erro seja maior ou igual a variação de y, e apenas alteramos o valor de y, se o valor do erro for menor ou igual a variação de x.

Por fim, podemos ver o resultado da função DrawLine na imagem 7.

```

1 void DrawLine(t_Pixel* pixel_i, t_Pixel* pixel_f)
2 {
3     /* Configurações da cor da reta */
4     int* rgb = pixel_i->m_rgb;
5
6     /* Como o ponto está generalizado para qualquer posição do plano
7      * devemos pegar a diferença absoluta tanto em dx como em dy,
8      * porem em dy vamos pegar sempre o negativo */
9     int dy = -abs(pixel_f->m_y - pixel_i->m_y);
10    int dx = abs(pixel_f->m_x - pixel_i->m_x);
11    int x = pixel_i->m_x;
12    int y = pixel_i->m_y;
13    int e = dx + dy;
14
15    /* O loop abaixo está generalizado para quer ponto x e y no plano cartesiano
16     * só vamos parar o loop, caso as posições de x e y estejam exatamente iguais
17     * a suas posições finais.*/
18    while (1){
19        t_Pixel pixel(x, y, rgb);
20        PutPixel(&pixel);
21
22        if(x == pixel_f->m_x and y == pixel_f->m_y){
23            break;
24        }
25        /* Dobramos o errado para uma variável e2 */
26        int e2 = 2 * e;
27        /* Caso a variação de y, seja menor ou igual a e2, iremos mudar a posição de x
28         * caso o final seja maior, iremos somar caso não iremos subtrair */
29        if (e2 >= dy){
30            if (pixel_f->m_x == x){
31                break;
32            }
33            e += dy;
34
35            x = (pixel_i->m_x < pixel_f->m_x ? x+ 1 :x-1);
36        }
37        /* Nesse caso, fazemos o equivalente porem para y */
38        if (e2 <= dx){
39            if(pixel_f->m_y == y){
40                break;
41            }
42            e += dx;
43
44            y = (pixel_i->m_y < pixel_f->m_y ? y+ 1 :y-1);
45        }
46    }
47 }
48
49 }

```

Figura 6: Implementação da função DrawLine

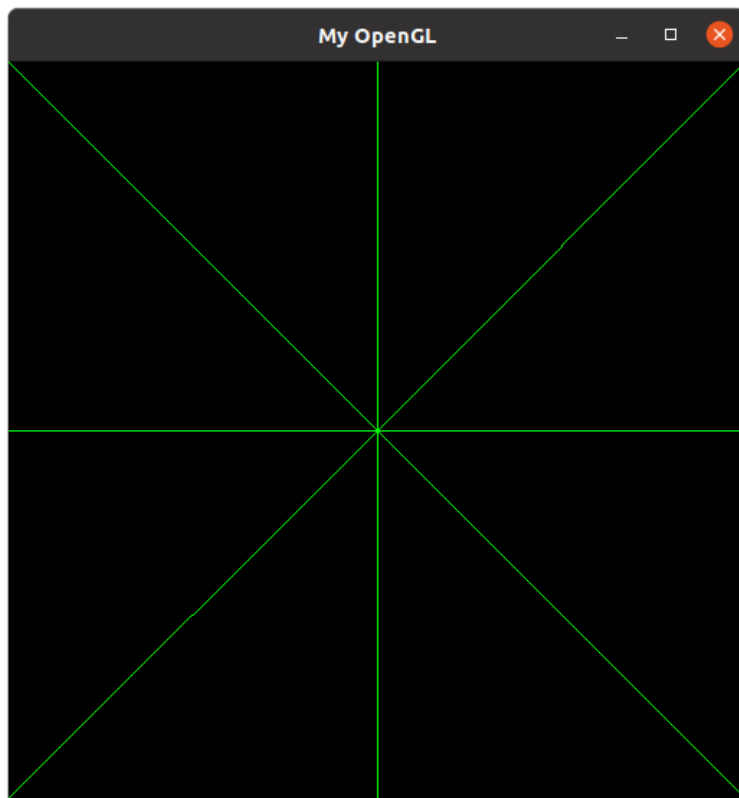


Figura 7: Resultados da função DrawLine

A screenshot of a code editor window with a dark background. It displays the implementation of a C++ function named DrawTriangle. The function takes three pointers to t_Pixel as arguments: pixeli, pixelm, and pixelj. The function body consists of three calls to DrawLine, each connecting one of the three vertices to form a triangle. The code is as follows:

```
1 void DrawTriangle(t_Pixel* pixeli, t_Pixel* pixelm, t_Pixel* pixelj){  
2  
3     DrawLine(pixeli, pixelm);  
4     DrawLine(pixelm, pixelj);  
5     DrawLine(pixeli, pixelj);  
6  
7 }
```

Figura 8: Implementação da função DrawTriangle

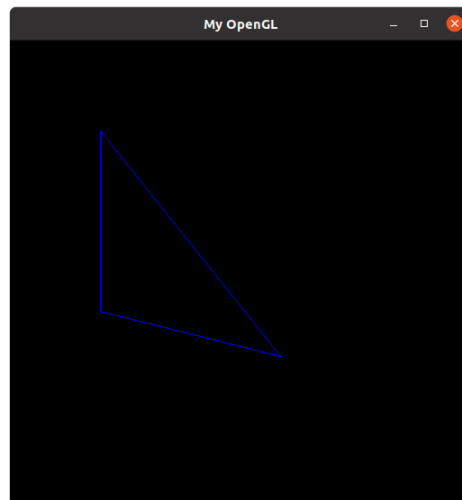


Figura 9: Resultado da função DrawTriangle

4 Função DrawTriangle

Como última implementação, temos a função DrawTriangle. A função é composta por 3 parâmetros, os quais são os três vértices de um triângulo. Supondo que A, B e C sejam os três vértices, a função cria uma linha de A para B, B para C e C para A, formando assim um triângulo. A implementação da função está representada na imagem 8 e os resultados estão na imagem 9.

5 Resultados

As funções PutPixel, DrawLine e DrawTriangle foram implementadas conforme seus algoritmos, sendo assim, os resultados obtidos foram satisfatórios. Para futuros trabalhos, é importante alterar o RGB das linhas

traçadas para facilitar a visualização do resultado.