# Computação Gráfica 2017.2

## Atividade I – Rasteirizando pontos ,linhas e triângulos em C.

Nossa primeira atividade da disciplina de Computação Gráfica consistiu na implementação de funções para rasteirizar pontos , linhas e triângulos usando a linguagem C na construção do código de 3 funções: putPixel() , drawLine() e a drawTriangle(),as quais serão explicadas mais adiante. Usamos uma framework disponibilizada pelo professor Christian Pagot que nos permite simular o acesso direto á memória de vídeo.

Primeiro precisamos entender que nossa tela está dividida em pequenos quadrados chamados Pixels e estes podem assumir as cores red ,green ,blue e o alfa (funciona como uma transparência na tela),cada pixel é representado por 4 bytes e cada cor pode variar de 0 á 255.

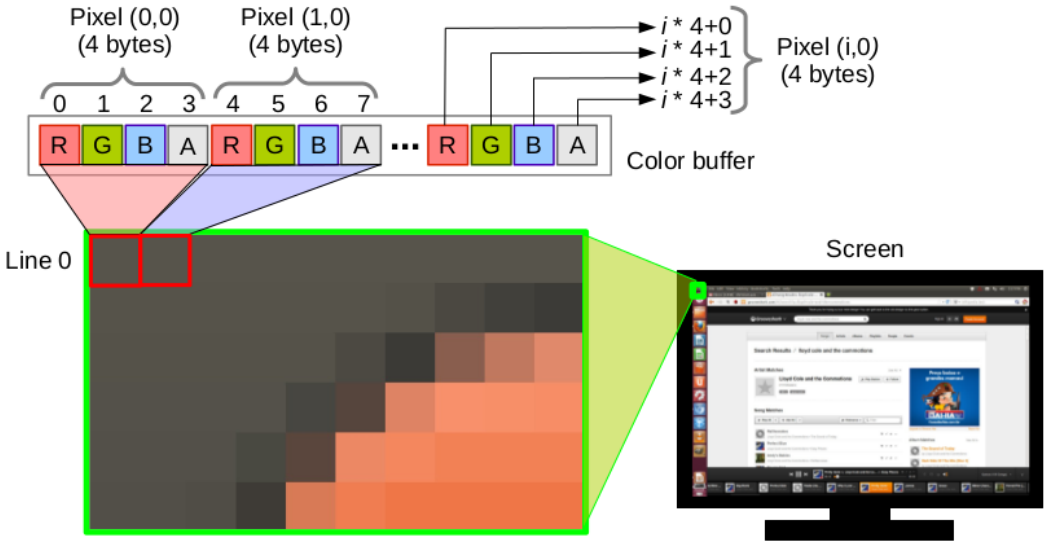



Figura 1: Representação dos pixels na tela.

Vamos começar definindo duas estruturas que irão armazenar os valores para cada coordenada do pixel e os valores dos componentes da cor desejada, todos serão do tipo inteiro. A estrutura tPixel terá as coordenadas x e y ,que correspondem a o ponto onde estará localizado o pixel na tela,já na estrutura tColor teremos as cores R,G,B e A, todas do tipo inteiro também.

```
typedef struct {
    int R;
    int G;
    int B;
    int A;
}tColor;
typedef struct{
    int posx;
    int posy;
}tPixel;
```

Figura 2 : Definição das estruturas em C.

## Função putPixel()

Esta função recebe como parâmetros as coordenadas (x,y) do pixel na tela e uma cor (RGBA),usamos uma relação matemática que nos permite encontrar a posição do pixel desejado na memória e por fim acendê-lo.

**offset = x*4+y*W*4;**

| | | |
|---|---|---|
| R | = | 4*X+ 4*Y*IMAGE_WIDTH |
| G | = | 4*X+ 4*Y*IMAGE_WIDTH **+1** |
| B | = | 4*X+ 4*Y*IMAGE_WIDTH **+2** |
| A | = | 4*X+ 4*Y*IMAGE_WIDTH **+3** |

Figura 3 : Calculo para acender o pixel na tela.

Obs: IMAGE_WIDTH é a largura da imagem.

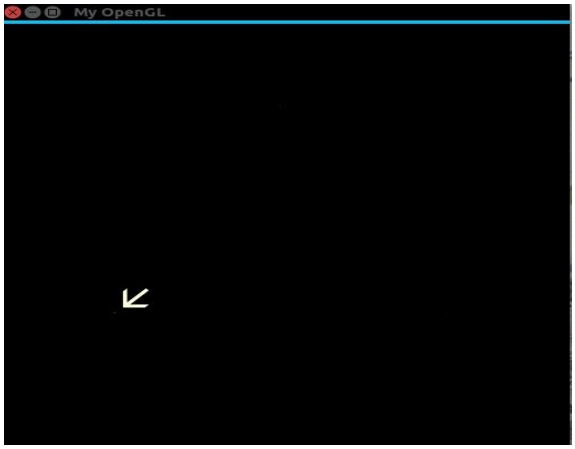



Figura 4: Pixel aceso na tela.

# Função drawLine()

Esta função desenha  uma linha na tela, recebendo como parâmetros dois pontos, um inicial e um final, mais a cor que ela terá. Para podermos rasterizar esta linha, usamos o algoritmo de Braseham's ,que determina quais pontos serão destacados, ou seja, quais pixels devem ser acesos na tela formando a reta, respeitando o ângulo de 0° á 45°,.Usamos o algoritmo para decidir a partir do pixel inicial qual será o próximo a ser acendido, se será o mais á sua direita(leste) ou á sua diagonal direita mais á cima (Nordeste),veja na figura abaixo:

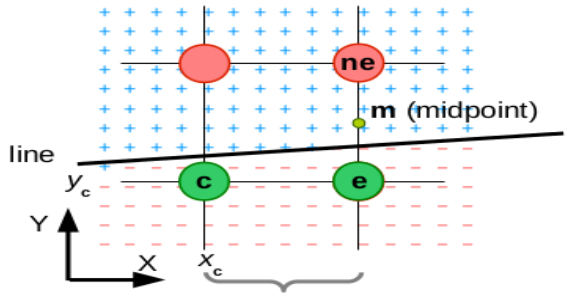



Figura 5 : Localização dos pixels que irão ser acesos.

Para a implementação do algoritmo de Braseham's tomei como base a derivação do mesmo ,conhecida por algoritmo do Ponto Médio, baseia-se na verificação da variável de decisão, se d=> 0 o pixel acendido será o abaixo da reta, caso contrário será o de cima, essa verificação será feita até que a reta esteja completamente desenhada.

```
MidPointLine() {
     int dx = x1 - x0;
     int dy = y1 - x0;
     int d = 2 * dy - dx;
     int incr_e = 2 * dy;
     int incr_ne = 2 * (dy - dx);
     int x = x0;
     int y = y0;
     PutPixel(x, y, color)
     while (x < x1) {
          if (d <= 0) {
               d += incr_e;
               x++;
          } else {
               d += incr_ne;
               x++;
               y++;
          }
          PutPixel(x, y, color);
     }
}
```

Figura 6 : Algoritmo do ponto médio.

Apesar de eliminar multiplicações e divisões este algoritmo não funciona para todos os 8 octantes, este é o nosso principal problema nesta função. Nos 1º,4º,5º e 8º octantes, temos que |dx|>|dy| e no restante |dy| >|dx|,assim basta fazermos as modificações de sinais nas variáveis x e y  para que funcione em todos eles, olhando a figura abaixo podemos ver quando será necessário fazer essas modificações:

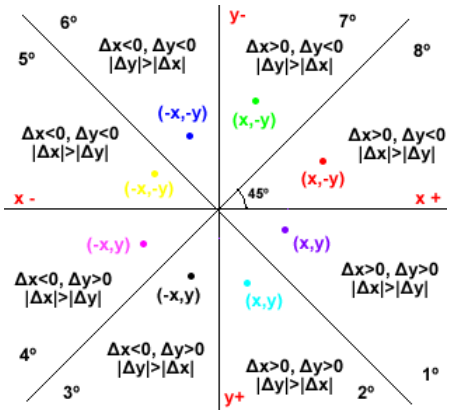



Figura 7 : Representação dos octantes.

Por fim vamos interpolar as cores, mas o que quer dizer isso? é mudar a cor gradualmente do ponto inicial até o ponto final, pegamos a diferença entre os dois pontos e dividimos pela distância, o resultado será incrementado na cor do  pixel anterior ,possibilitando saber a cor do próximo pixel, porem devemos verificar se os pixels estão no primeiro octante ,caso sim ,devemos pintar os pixels da quantidade  dx, já no segundo e sétimo a quantidade será igual ao dy.

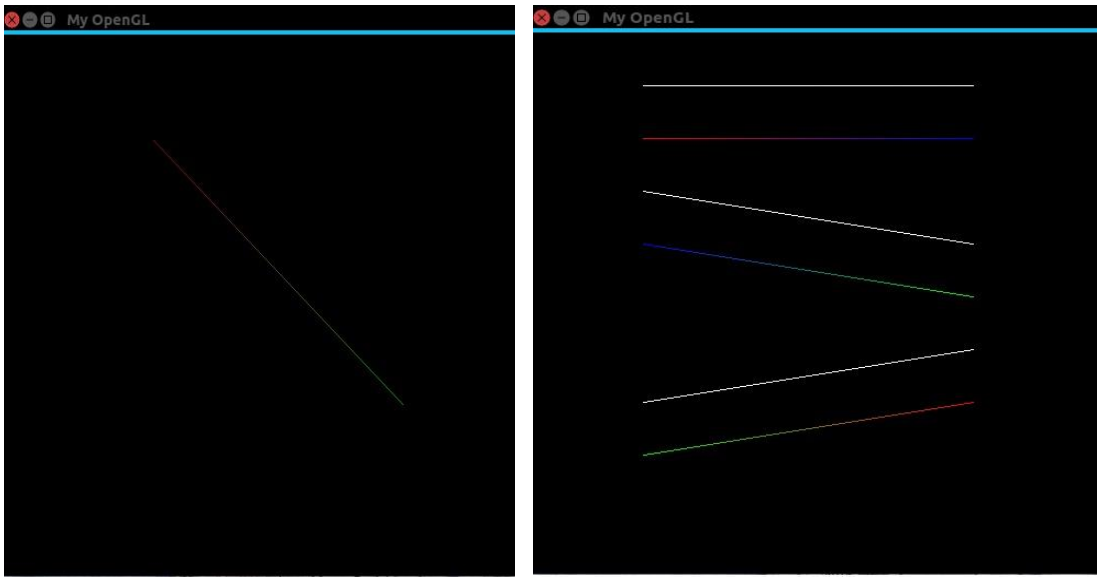



Figura 8-9: Linha com interpolação.

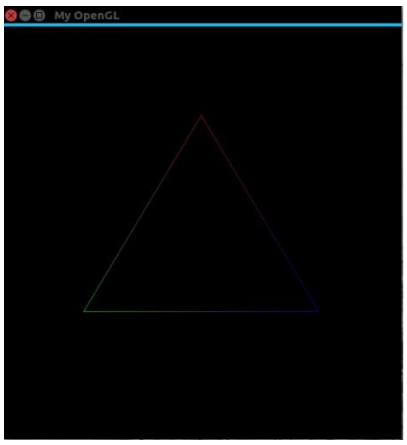
# Função drawTriangle()

Esta função não tem mistério ,passaremos os parâmetros da seguinte forma drawTriangle(Ponto 1,Ponto2,Ponto3,Cor1,Cor2,Cor3),dentro desta função iremos chamar três vezes a drawLine da seguinte forma:

drawLine(ponto1, ponto2, c1, c2);

drawLine(ponto2, ponto3, c2, c3);

drawLine(ponto3, ponto1, c3, c1);



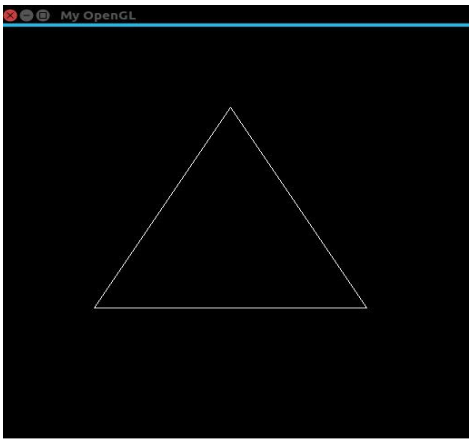




Figura 10 : Triângulo com interpolação. Figura 11 : Cores(255,255,255,255).

## Dificuldades Encontradas:

A maior dificuldade foi generalizar o algoritmo de Braseham's para os oito octantes ,porém foi facilmente solucionada quando passamos a entender como funciona a mudança de sinais nas variáveis x e y em cada octante. A parte da interpolação deu um pouco de trabalho ,pois precisávamos avaliar os valores dos dx e dy para fazer a distância da reta.

## Melhorias:

Com certeza uma melhoria significativa na rasteirização de triângulos seria conseguir preenche-lo, deixaria mais visível. Utilizar apenas uma struct para os vértices nos permitiriam usar menos parâmetros nas funções.

## Referências Bibliográficas:


- Slides disponibilizados pelo professor.
- http://letslearnbits.blogspot.com.br/2014/10/icgt1-interpolacao-de-cores.html
- http://icgdadepressao.blogspot.com.br/2016/03/rasterizacao-triangulos-com-o-algoritmo.html
- https://pt.wikipedia.org/wiki/Algoritmo_de_Bresenham