Relatório Trabalho 1 - Introdução a Computação Gráfica
2017.2

Clístenes Onassis Chaves Araujo Mat: ***2019077969***
Lucas Correia Barbosa dos Santos Mat: ***2016000560***
Rebeca de Macêdo Ferreira Mat: ***2016000524***

A atividade proposta pelo professor Christian é a de implementarmos algoritmos de rasterização de pontos e linhas, formando também triângulos. O professor implementou e nos disponibilizou um framework com vários arquivos essenciais para implementação do trabalho.

De início uma breve definição de Rasterização:

Rasterização*, é a tarefa de converter uma imagem raster em uma imagem vetorial (pixels ou pontos) para a possível leitura do documento. -* ***Wikipedia***

- **PutPixel()**

Primeiramente iremos implementar a função **PutPixel()** que irá inserir um pixel na tela com uma cor e coordenada determinada. Essa função irá receber como parâmetros as coordenadas **x** e **y** do pixel, e a **cor**.

No arquivo **mygl.h,** no espaço reservado para modificações foi implementado o seguinte código:

```
10  typedef struct{
11      int r;
12      int g;
13      int b;
14      int a;
15  }color;
16
```

Este trecho de código cria uma estrutura chamada color que contém 4 variáveis inteiras cada uma representando um byte do pixel, tendo em vista que o pixel que utilizaremos ocupa 4 bytes, um para cada canal R, G, B, A. Esta estrutura tem a intenção de reduzir a quantidade de parâmetros na função a ser criada.

Ainda no mesmo arquivo, iremos implementar a função em si,

```
18  // Definicao da funcao PutPixel
19  void PutPixel(int x, int y, color cor){
20      FBptr[x*4 + IMAGE_WIDTH*y*4 + 0] = cor.r;
21      FBptr[x*4 + IMAGE_WIDTH*y*4 + 1] = cor.g;
22      FBptr[x*4 + IMAGE_WIDTH*y*4 + 2] = cor.b;
23      FBptr[x*4 + IMAGE_WIDTH*y*4 + 3] = cor.a;
24  }
```

como mostrado na imagem acima, a função recebe como parâmetros as coordenadas e a cor.

Tomando como material base o slide de Rasterização disponibilizado pelo professor, podemos entender que para um determinado Pixel(x,y) o cálculo a ser utilizado para ter acesso a determinada linha e coluna na tela é generalizado pelo Offset = (x*4 + y*w*4), onde w é a largura da tela, no caso do framework definido pela constante IMAGE_WIDTH.

Substituindo na fórmula, temos o resultado da imagem acima, cada linha representa um byte do pixel que recebe o valor do canal correspondente na variável cor.

No arquivo **main.cpp** no espaço reservado para modificações foi implementadas as seguintes instruções:

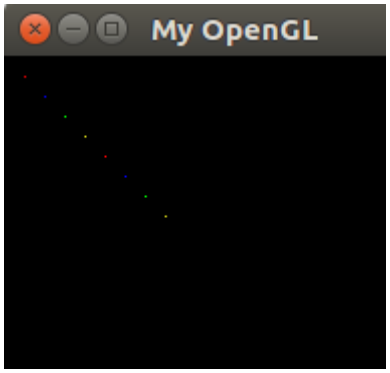
```
11  // Definindo algumas cores
12  color red;
13  red.r = 255;
14  red.g = 0;
15  red.b = 0;
16  red.a = 0;
17
18  color blue;
19  blue.r = 0;
20  blue.g = 0;
21  blue.b = 255;
22  blue.a = 0;
```

Criamos 4 variáveis para 4 cores distintas, red, blue, green, e yellow. As linhas de código acima modificam os canais da variável com o valor padrão de cada cor.(Segue análogo para green e yellow).

A seguir chamamos a função PutPixel(), passando como parâmetros a localização do ponto em pixels e a cor. (respeitando os limites de Largura e Altura da Janela 512x512 pixels).

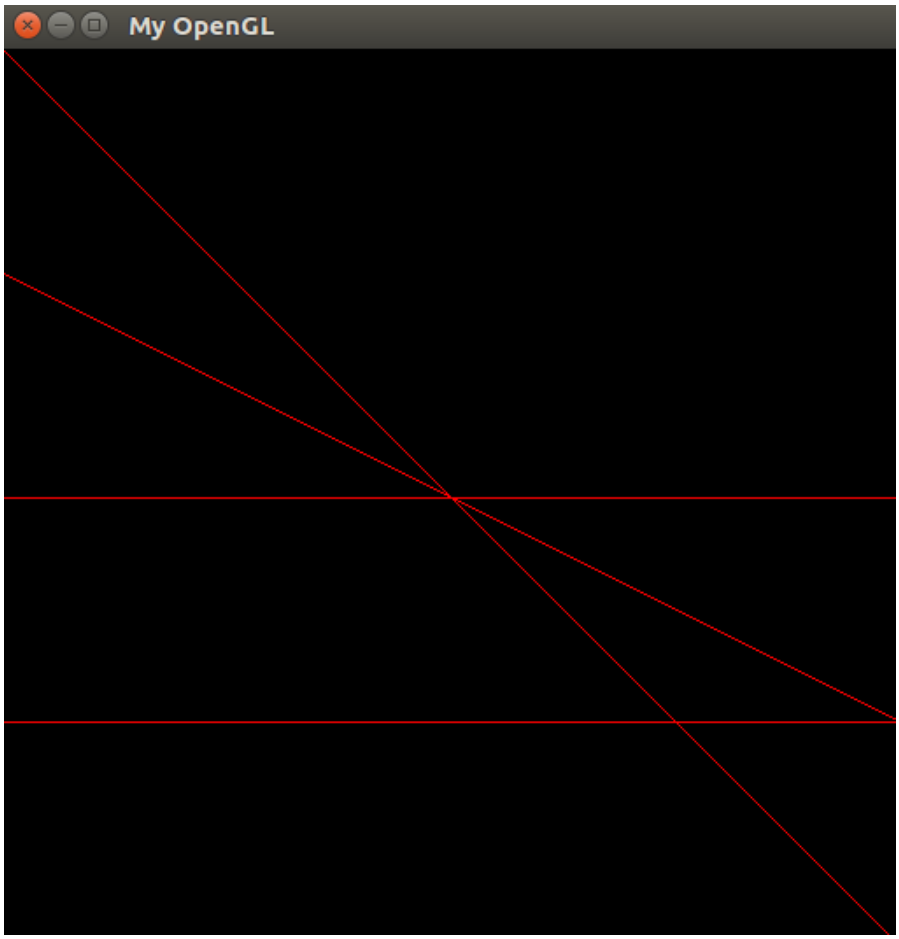
```
36  // Chamada do metodo PutPixel
37
38  PutPixel(10,10, red);
39  PutPixel(20,20, blue);
40  PutPixel(30,30, green);
41  PutPixel(40,40, yellow);
42  PutPixel(50,50, red);
43  PutPixel(60,60, blue);
44  PutPixel(70,70, green);
45  PutPixel(80,80, yellow);
46
```

A seguir o resultado obtido:



- **DrawLine()**

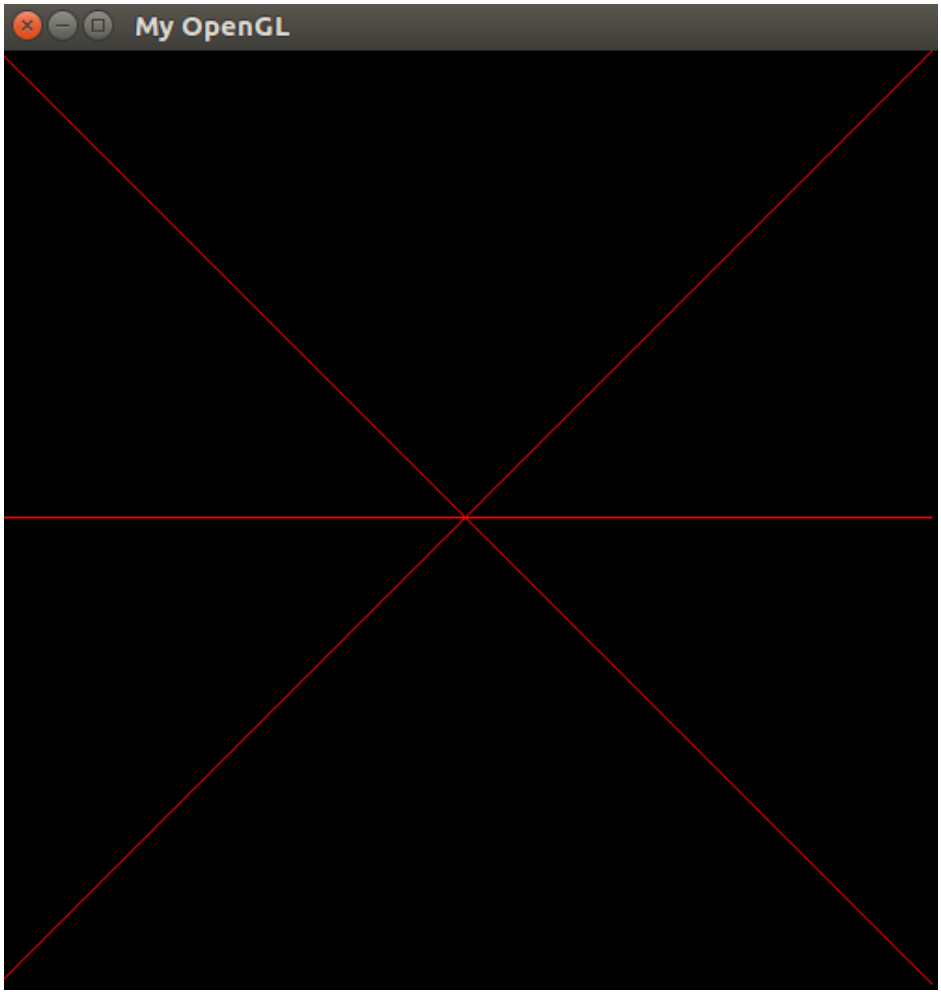
Para a implementação da função de rasterizar linhas nós usamos o algoritmo de Bresenham. Esse algoritmo otimiza os custos de operações para rasterizar linhas pois é baseado apenas na incrementação dos pontos.



Percebemos, então que este algoritmo apenas permite a rasterização de linhas apenas de 0º a 45º, no primeiro quadrante, então foi preciso fazer modificações no código para poder rasterizar linhas com outras inclinações.

```
if (dy < 0){
    inclinacao = -1;
    dy = -dy;
}
else{
   inclinacao = 1;
}
```

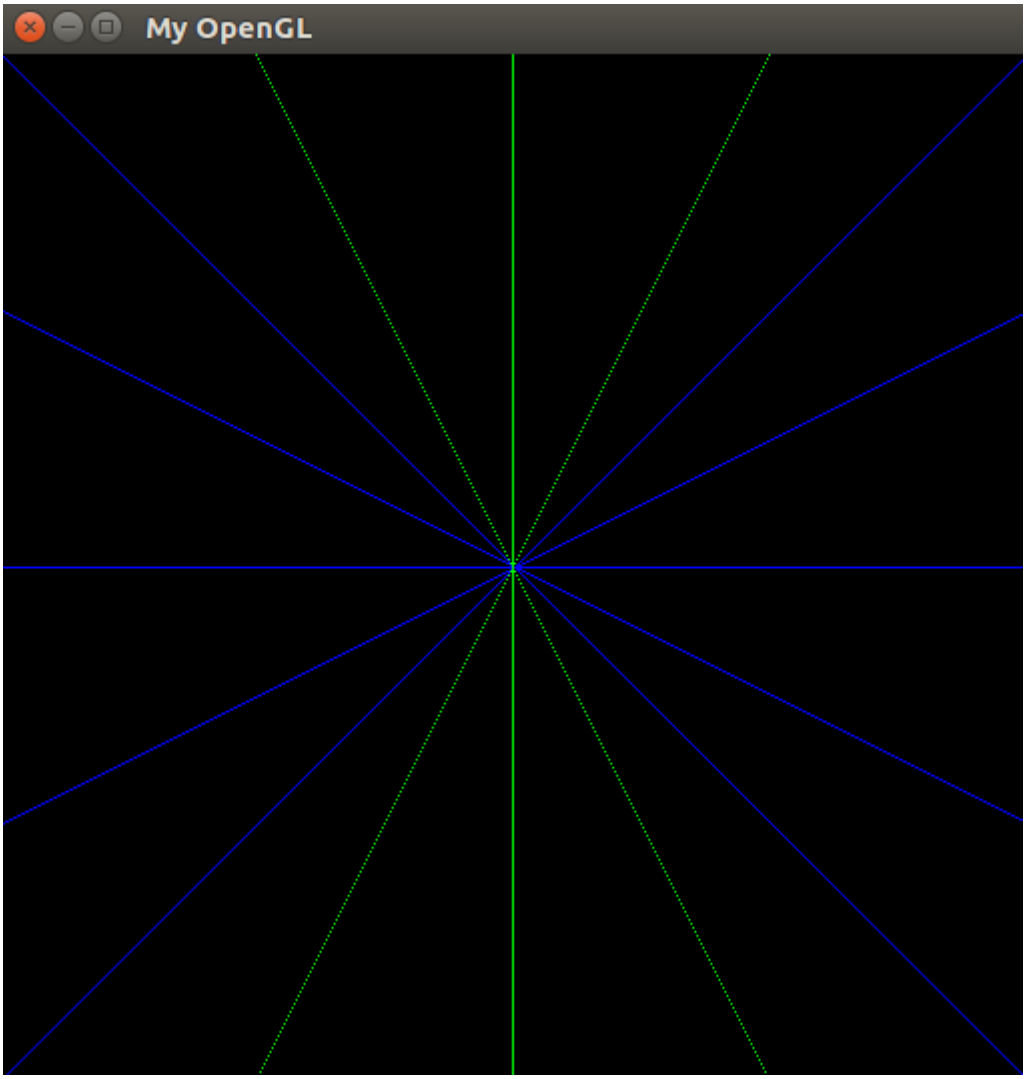
O resultado pra essas implementações foi a seguinte:



Contudo, ainda não conseguíamos rasterizar linhas com que tinham coeficiente angular maior que 1 (dx < dy), então invertemos os eixos de coordenadas e fizemos uma reta dx < dy ter seu coeficiente angular menor que 1, o que nos permitiria desenhar as retas e suas inclinações, fizemos uma função Inverte() , para fazer tal procedimento, que foi usada na hora de ‘escrever’ os pixels da linha.

```
void inverte(int a, int b){
    int aux2 = a;
    a = b;
    b = aux2;
}
```

Com isso obtivemos o seguinte resultado:



linhas azuis: dx > dy;
linhas verdes: dx < dy.

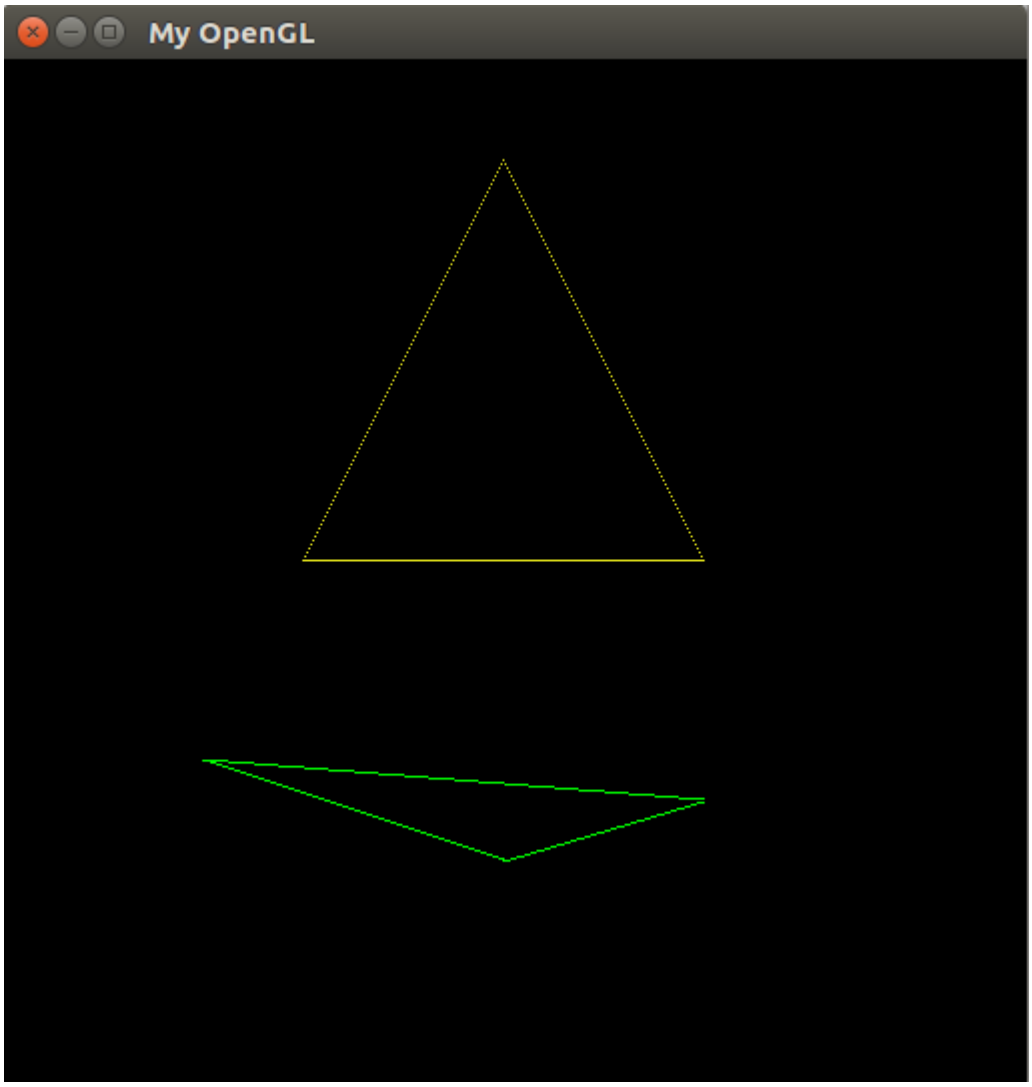
- **DrawTriangle()**

Esta função é bem simples, como já temos uma função que desenha retas, ela simplesmente recebe como parâmetro três vértices e três cores. Em seguida fizemos a ligação entre os vértices utilizando a função DrawLine().

```
137
138  void DrawTriangle(int x1, int y1, int x2, int y2, int x3, int y3,
139   color cor1, color cor2, color cor3){
140      DrawLine(x1, y1, x2, y2, cor1, cor2);
141      DrawLine(x2, y2, x3, y3, cor2, cor3);
142      DrawLine(x3, y3, x1, y1, cor3, cor1);
143
144  }
145
```

```
59
60   DrawTriangle(150,250,350,250,250,50, yellow, yellow, yellow);
61   DrawTriangle(100,350,350,370,250,400, green, green, green);
62
63
```

A seguir obtivemos os seguintes resultados:



Interpolação de Cores

A interpolação de cores consiste em variar as cores de uma reta, calculando a porcentagem que ainda falta para percorrer até chegar ao ponto final. Esta porcentagem se calcula pela razão entre a distância parcial e total do ponto multiplicada pelas cores, como determina a fórmula:

$$P_{cor} = p^*(p_I^{cor}) + (1-p)(p_F^{cor})$$

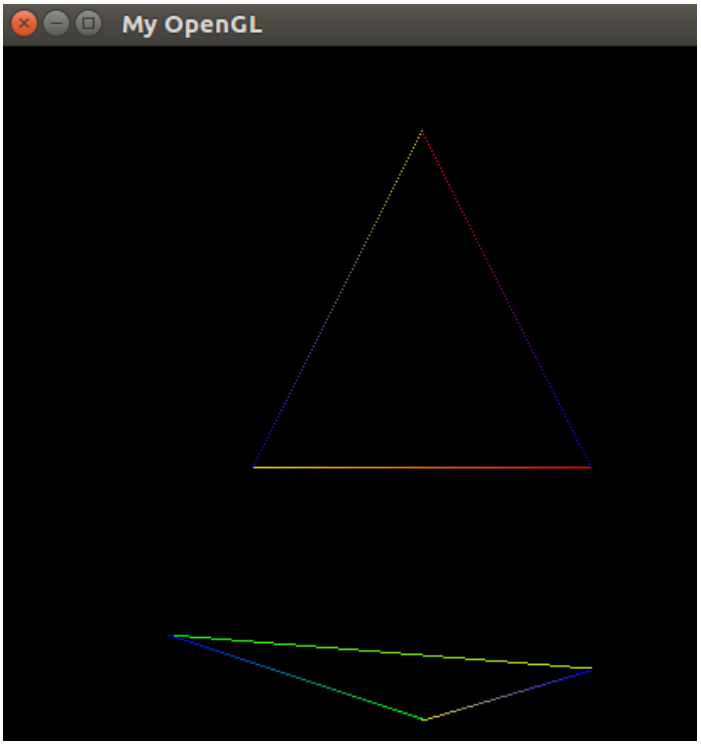
Pcor = nova cor do pixel
p = porcentagem calculada entre distância parcial e total (Varia entre 0 e 1)
corI = cor do pixel inicial
corF = cor do pixel final

A seguir seguem algumas linhas de códigos referentes a interpolação:

```
dx = x2 - x1;
dy = y2 - y1;

float distT = sqrt((dx*dx) + (dy*dy));
```

```
int distP = sqrt(((x2 - x)*(x2 - x)) + ((y2 - y)*(y2 - y)));

float p = (distP/distT);

color pCor;
pCor.r = ((p*cor1.r) + ((1-p)*cor2.r));

pCor.g = ((p*cor1.g) + ((1-p)*cor2.g));

pCor.b = ((p*cor1.b) + ((1-p)*cor2.b));
```

A seguir os resultados obtidos:



- ## Dificuldades encontradas

A primeira dificuldade encontrada foi a de rasterizar linhas em todos os quadrantes, já que o algoritmo de Bresenham só nos permite rasterizar no primeiro. Passamos muitas horas para entender como funcionava de fato todo o algoritmo de Bresenham para então começar a adaptá-lo para todas as direções. Enxergar as simetrias existentes foi um dos desafios, além de conseguir achar uma forma de rasterizar as linhas que tinha seu dx < dy.

- ## Referências

1. Slides de aula do professor Christian.

2. http://dlguilherme.blogspot.com.br/2013/06/rasterizacao-de-primitivas-computacao.html
3. https://bitunico.wordpress.com/2012/12/16/rasterizacao-em-cc-algoritmo-de-bresenham/