Primeiro Trabalho de Introdução à Computação Gráfica

Grupo: Pedro de Abrantes Fernandes - 20160142370
Luciano Silva de Santana – 11408111

Link para a postagem no blog: https://cgpedroabrantes.blogspot.com.br/2017/09/primeiro-trabalho-de-introducao.html

Introdução:

Este trabalho tem como objetivo colocar em prática alguns conceitos e algorítimos que aprendemos em aula. Temos o objetivo de implementar as seguintes funções: ***PutPixel***, que *printa* um pixel na tela. A função ***DrawLine***, que desenha uma linha utilizando o algorítimo de Brasenham para rasterização de linhas. A função ***DrawTriangle***, que desenha um triangulo utilizando também a rasterização de linhas.

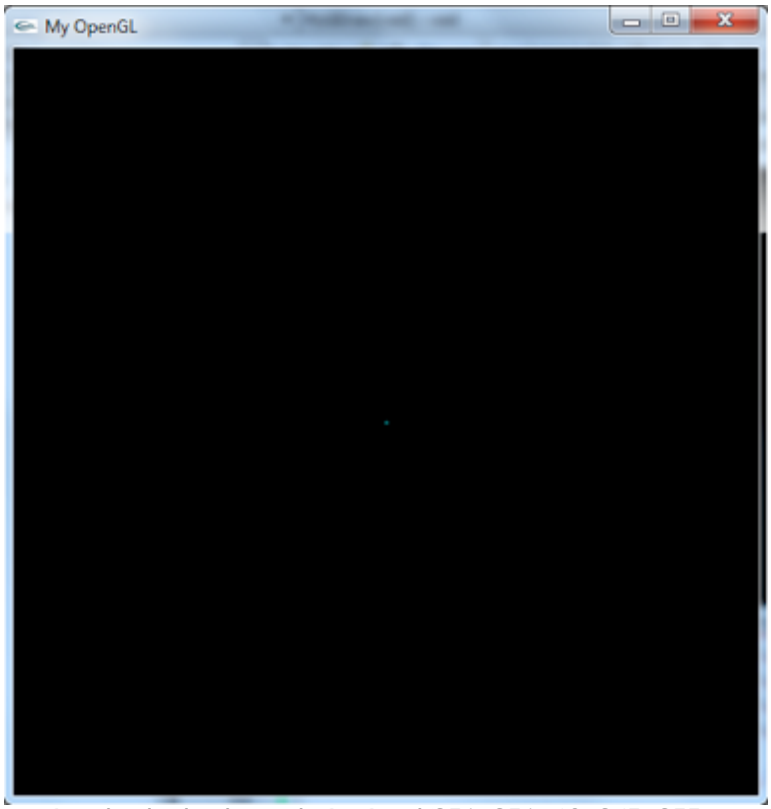
Função PutPixel()

A função PutPixel() rasteriza um ponto na memória de vídeo, recebendo como parâmetro sua posição (x, y) na tela e sua cor RGBA, como podemos ver no cabeçalho da função:

```
void PutPixel(int x, int y, int r, int g, int b)
{
	FBptr[(x*4) + (y*IMAGE_WIDTH*4) + 0] = r;
	FBptr[(x*4) + (y*IMAGE_WIDTH*4) + 1] = g;
	FBptr[(x*4) + (y*IMAGE_WIDTH*4) + 2] = b;
	FBptr[(x*4) + (y*IMAGE_WIDTH*4) + 3] = 255;


}
```

essa função faz os offsets na memória de vídeo para rasterizar o ponto na posição indicada e com as cores fornecidas na função.



Resultado da chamada PutPixel(256, 256, 10, 245, 255);

Função DrawLine()

A função DrawTriangle() deve rasterizar uma linha na tela, recebendo como parâmetro dois vértices v1 = (x0, y0) e v2 = (x1, y1) e suas cores RGBA e deve fazer isso utilizando o algorítimo de Brasenham. A função deve, além de rasterizar a linha, fazer com que as cores dos pixels ao longo da linha sejam obtidos por interpolação linear das cores dos vértices. Para ter "menos" parâmetros chamados na função, foi criado a struct abaixo para conter os valores de RGB de cada vértice

```
typedef struct {
    int r;
    int g;
    int b;
} tipoCor;
```

Utilizando a estrutura, foram criados:
tipoCor cor1 = {255, 0, 0};
tipoCor cor2 = {0, 0, 255};
tipoCor cor3 = {12, 245, 255};
tipoCor cor4 = {234, 103, 5};

Função: void DrawLine(int x0, int y0, int x1, int y1, tipoCor c1[], tipoCor c2[])

Para fazer a interpolação linear das cores dos vértices, devemos ver a variação das cores de um vértice em relação ao outro, ou seja, RGBV2 - RGBA V1. Por exemplo: se o vértice v1 tiver cor RGB (255, 0, 0)

e o vértice v2 (0, 0, 255), o R tem variação -255, o G tem variação 0 e o B tem variação de +255. Depois de calcular a variação, devemos calcular o tamanho da linha utilizando o teorema de Pitágoras. Sendo assim, temos que: Tamanho da Linha = sqrt((x1 - x0)² + (y1 - y0)²). Tendo agora a variação das cores de um vértice para o outro e o tamanho da linha, dividimos a variação pelo tamanho para obter o quanto de R,G e B será incrementado à cada ponto rasterizado. Para exemplificar, com as variações de RGB sendo respectivamente -255, 0, 255 e o tamanho da linha sendo 2550, dividimos cara variação pelo tamanho para obter o quanto será incrementado ou decrementado de cada cor a cada ponto. Temos então: addR = -255/2550 = -0.1; addG = 0/2550 = 0; addB = 255/2550 = +0.1. Assim, a cada ponto rasterizado será decrementado -0.1 de R, 0 de G e incrementado +0.1 de B.

```
int variacaoR = (c2->r) - (c1->r); //se a variacao for 0, vai incrementar 0
int variacaoG = (c2->g) - (c1->g);
int variacaoB = (c2->b) - (c1->b);

unsigned int vdx = dx; //variacao de x e y a cada incremento na forma de inteiro sem sinal
unsigned int vdy = dy;

float tamLinha = sqrt((vdx*vdx) + (vdy*vdy)); //tamanho da linha a ser desenhada

float addR = (float) variacaoR / tamLinha;
float addG = (float) variacaoG / tamLinha;
float addB = (float) variacaoB / tamLinha;

float r0 = c1->r; //valores iniciais de RGB
float g0 = c1->g;
float b0 = c1->b;
```

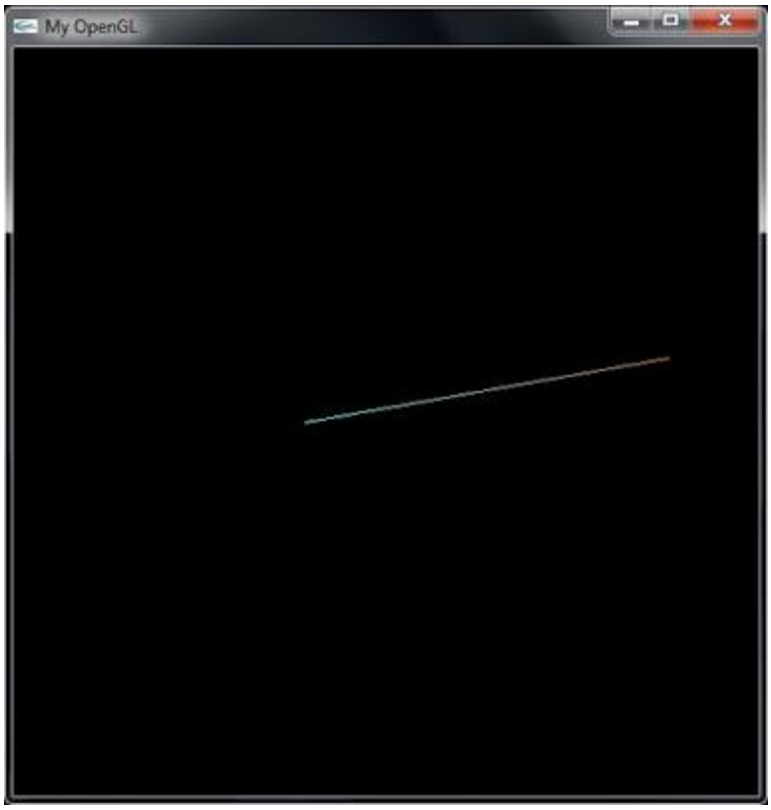
Resolvendo a interpolação linear, sobra o algorítimo de Brasenham para rasterização de linhas. Em aula, vimos o algorítimo de Brasenham que funciona para valores de 0 < m < 1, sendo m = dy/dx. Para começar, precisamos calcular o dx e dy, que são a variação de x e y de um vértice em relação ao outro. Assim: dx = x1 - x0 ; dy = y1 - y0. Temos também a variável de decisão d, que no caso que vimos em aula de 0 < m < 1, d = 2 * dy – dx. A variável d irá decidir se o próximo ponto será rasterizado à leste ou à nordeste do ponto anterior. Utilizando o algorítimo visto em aula, foi obtido a linha abaixo:

- **The entire algoritm for 0 < m < 1:**

```
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);
    }
}
```

The **computation** of ***d***, now, **involves** only **addition**!

Slopes **outside** the range **[0,1]** can be **handled** by **symmetry**!



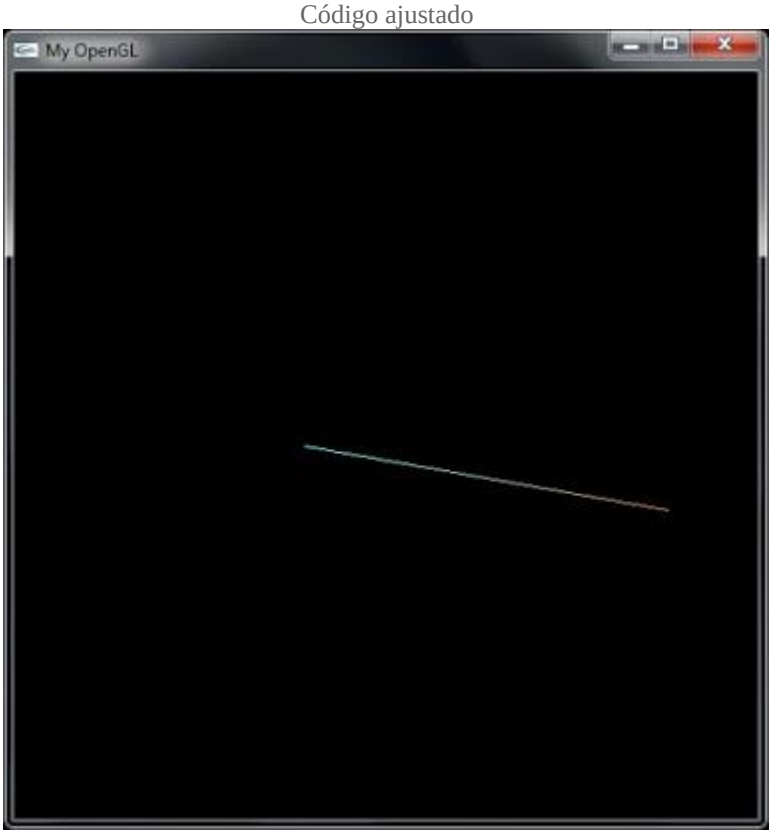
Chamada: DrawLine(200, 256, 450, 212, &cor3, &cor4);
tipoCor cor3 = {12, 245, 255}; tipoCor cor4 = {234, 103, 5};

Como m < 1 para ângulos de -45º à 45º com o eixo x, para generalizar o algorítimo, utilizei uma variável chamada inclinacao para identificar os casos em que m <= 1. A variável inclinacao recebe 1 se dy < 0 e recebe -1 se dy > 0. Assim os casos em que m <= 1 são aqueles onde dx >= inclinacao * dy. Sabendo que m <= 1 e que podemos utilizar o algorítimo visto em sala, foi preciso fazer um ajuste para os casos onde a reta é decrescente, ou seja, y1 < y0 (tendo em vista que o eixo y cresce de cima para baixo). Para fazer a reta decrescente, temos que fazer os ajustes: d = 2 * dy + dx, incr_ne = 2 * (dy + dx) e, se cair no caso d < 0, y deve ser decrementado ao invés de incrementado.

```
if(dx >= inclinacao * dy) // caso m <= 1
{
    if(dy < 0) // caso y1 < y0
    {
        d = 2 * dy + dx;
        incr_e = 2 * dy;
        incr_ne = 2 * (dy + dx);
        while(x < x1)
        {
            if(d < 0)
            {
                d += incr_ne;
                x++;
                y--;
            }
            else{
                d += incr_e;
                x++;
            }
            PutPixel(x, y, round(r0 += addR), round(g0 += addG), round(b0 += addB));
        }
    }
    else // caso y0 < y1
    {
        d = 2 * dy - dx;
        incr_e = 2 * dy;
        incr_ne = 2 * (dy - dx);

        if(dy == 0)
```

Código ajustado

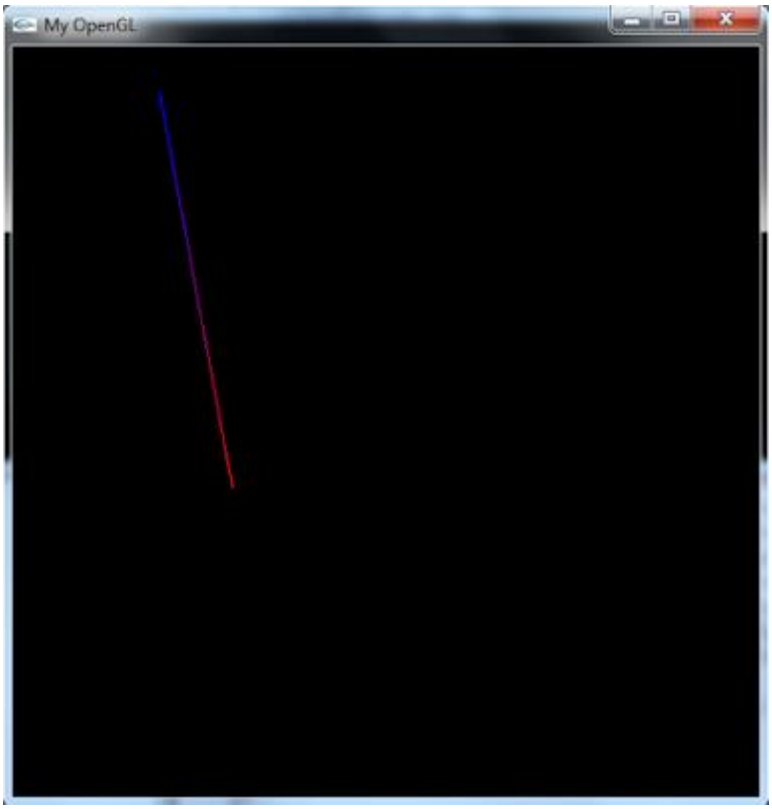


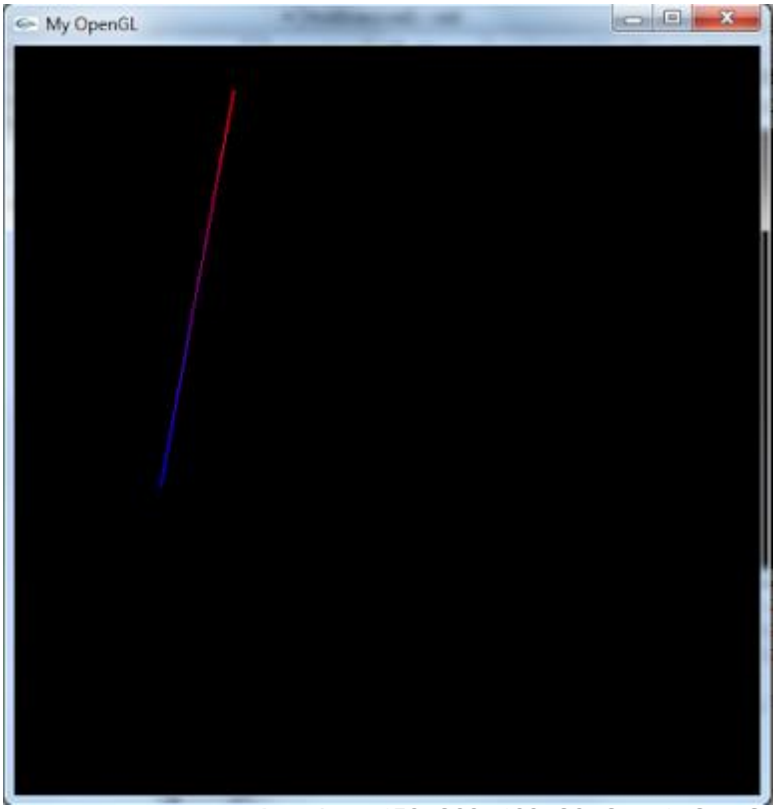
Resultado da chamada: DrawLine(200, 212, 450, 256, &cor3, &cor4);
tipoCor cor3 = {12, 245, 255}; tipoCor cor4 = {234, 103, 5};

Com os casos em que 0 < m < 1 resolvidos, precisamos resolver onde m > 1. Para esses casos, devemos inverter os eixos x e y. Os casos ainda são os mesmos do anterior (dy < 0 e dy >0), só que agora, aonde tinha x ou dx, devemos substituir por y ou dy para inverter os eixos e calcular corretamente. As inversões foram feitas diretamente nas fórmulas. Sendo assim, d = 2*dy ± dx virou d = dy ± 2*dx, por exemplo.

```
else // caso |m|>1
{
    if(dy < 0) // caso y1 < y0
    {
        d = dy + 2 * dx;
        incr_e = 2 * dx;
        incr_ne = 2 * (dy + dx);
        while(y > y1)
        {
            if(d < 0)
            {
                d += incr_e;
                y--;
            }
            else
            {
                d += incr_ne;
                x++;
                y--;
            }
            PutPixel(x, y, round(r0 += addR), round(g0 += addG), round(b0 += addB));
        }
    }
    else // caso y0 < y1
    {
        d = dy - 2 * dx;
        incr_e = -2 * dx;
        incr_ne = 2 * (dy - dx);
        while(y < y1)
```

Código para inverter os eixos

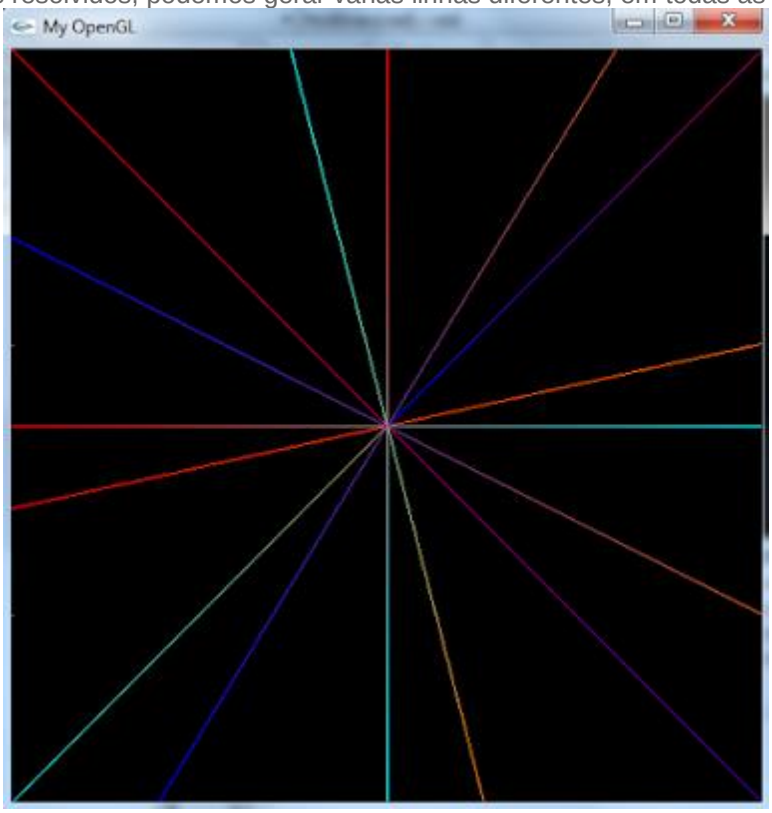




Resultados das chamadas: DrawLine(150, 300, 100, 30, &cor1, &cor2) e DrawLine(150, 30, 100, 300, &cor1, &cor2) respecivamente.

tipoCor cor1 = {255, 0, 0}; tipoCor cor2 = {0, 0, 255};

Com todos os casos resolvidos, podemos gerar várias linhas diferentes, em todas as direções.



Entradas:
DrawLine(512, 0, 256, 256, &cor1 , &cor2);
DrawLine(256, 256 , 0, 0, &cor3, &cor4);
DrawLine(256, 0, 256, 512, &cor1, &cor3);
DrawLine(0, 256, 512, 256, &cor1, &cor3);
DrawLine(0, 512, 256, 256, &cor3, &cor4);
DrawLine(0, 0, 512, 512, &cor1, &cor2);
DrawLine(100, 512, 412, 0, &cor2, &cor4);
DrawLine(190, 0, 322, 512, &cor3, &cor4);
DrawLine(0, 312, 512, 200, &cor1, &cor4);
DrawLine(0, 128, 512, 384, &cor2, &cor4);

Função DrawTriangle()

A função DrawTriangle() deve receber 3 vértices, v1, v2 e v3 e, com a função DrawLine() implementada, apenas precisamos chamar ela 3 vezes para desenhar linhas de v1 para v2, de v2 para v3 e de v3 para v1, fechando assim o triângulo. Para diminuir a quantidade de parâmetros chamados na função, implementei uma struct chamada tipoPixel que contém a posição e as cores de cada vértice:

```
typedef struct {
    tipoCor c;
    int x;
    int y;
} tipoPixel;
```

Função:

```
void DrawTriangle(tipoPixel p1[], tipoPixel p2[], tipoPixel p3[])
{
	DrawLine(p1->x, p1->y, p2->x, p2->y, &p1->c, &p2->c);
	DrawLine(p2->x, p2->y, p3->x, p3->y, &p2->c, &p3->c);
	DrawLine(p3->x, p3->y, p1->x, p1->y, &p3->c, &p1->c);
}
```

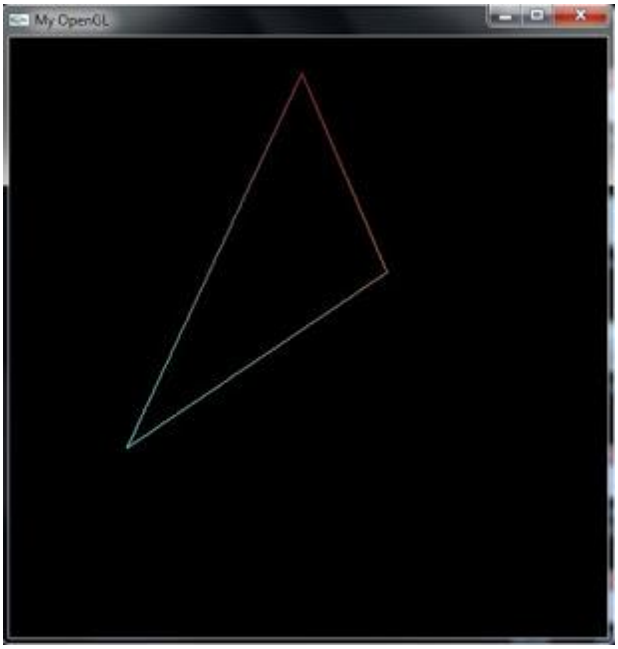
Com as linhas de código:

```
tipoPixel p1 = {cor1, 250, 30};
tipoPixel p2 = {cor3, 100, 350};
tipoPixel p3 = {cor4, 323, 200};

DrawTriangle(&p1, &p2, &p3);
```

tipoCor cor1 = {255, 0, 0}; tipoCor cor3 = {12, 245, 255}; tipoCor cor4 = {234, 103, 5};

Obtemos o triângulo abaixo:



Extra:

Função PaintTriangle()

A função PaintTriangle() deve receber 3 vértices, v1, v2 e v3. Para pintar o triangulo, primeiro é usado a função DrawLine() para desenhar uma linha de v1 para v2.

```
void PaintTriangle(tipoPixel p1[], tipoPixel p2[], tipoPixel p3[])
{
    tipoCor aux;
    int x1 = p3->x;
    int x0 = p2->x;
    int y0 = p2->y;
    int y1 = p3->y;
    tipoCor* c1 = &p2->c;
    tipoCor* c2 = &p3->c;


    DrawLine(p1->x, p1->y, p2->x, p2->y, &p1->c, &p2->c);
```

Após isso, quando a linha vai ser desenhada de v2 para v3, a cada chamada do PutPixel() no laço de *while*, é desenhada uma linha com o DrawLine() do vértice v1 para o pixel que acabou de ser rasterizado.

```
if(dx == 0) //caso retas verticais (onde x não varia)
{
    if(y0 < y1) //varia o y inicial ate o final
    {
        PutPixel(x0, y0, r0, g0, b0);
        while(y0++ < y1)
        {
            PutPixel(x0, y0, round(r0 += addR), round(g0 += addG), round(b0 += addB));
            aux = {round(r0), round(g0), round(b0)};
            DrawLine(p1->x, p1->y, x0, y0, &p1->c, &aux);
        }
    }
    else //varia o y final ate o inicial
    {
        PutPixel(x0, y0, r0, g0, b0);
        while(y1++ < y0)
        {
            PutPixel(x0, y1, round(r0 += addR), round(g0 += addG), round(b0 += addB));
            aux = {round(r0), round(g0), round(b0)};
            DrawLine(p1->x, p1->y, x0, y0, &p1->c, &aux);
        }
    }

    return;
```

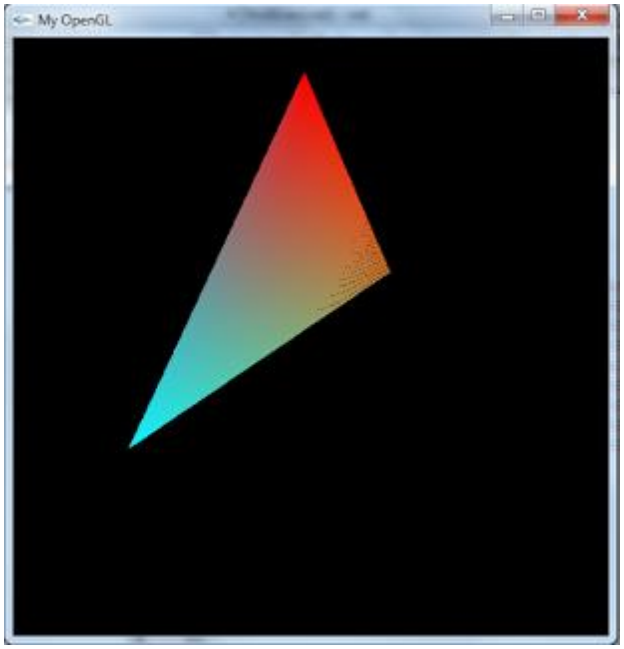
Utilizando as linhas de código abaixo:

```
tipoPixel p1 = {cor1, 250, 30};
tipoPixel p2 = {cor3, 100, 350};
tipoPixel p3 = {cor4, 323, 200};

PaintTriangle(&p1, &p2, &p3);
```

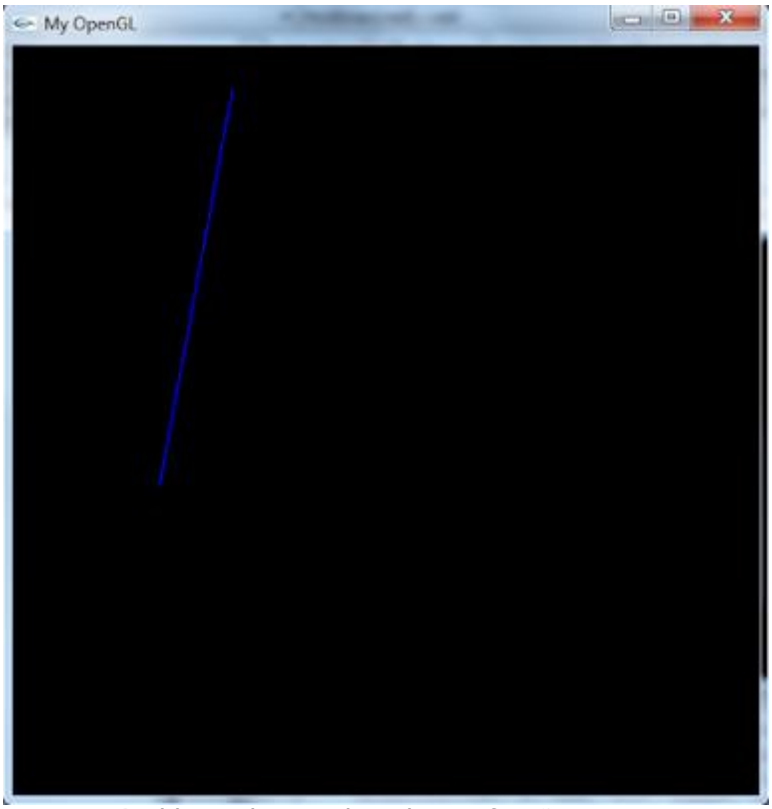
tipoCor cor1 = {255, 0, 0}; tipoCor cor3 = {12, 245, 255}; tipoCor cor4 = {234, 103, 5};

Obtemos o triangulo:

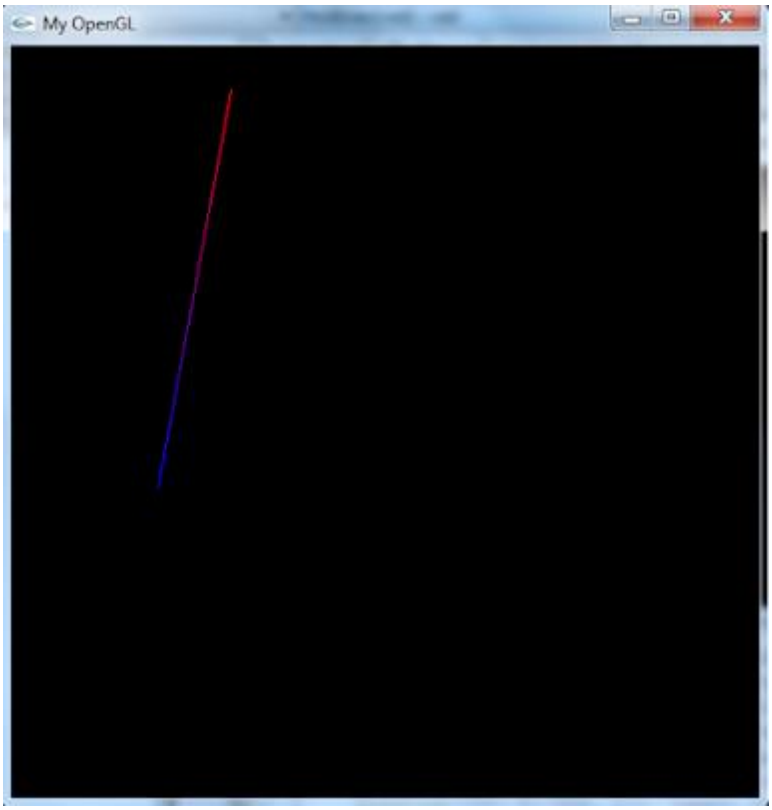


Dificuldades:

Inicialmente tive problemas para fazer o GLUT e o framework funcionarem no Windows. No código em si, encontrei minha primeira pequena dificuldade no algorítimo de interpolação linear das cores.
Também tive problemas tentando colorir linhas onde o x0 > x1 e y0 < y1, pois o algorítimo estava pegando a cor do segundo vértice e aplicando em toda a linha. Depois de alguns debugs descobri que o erro estava no cálculo do tamanho da linha, que estava dando negativo porque eu usei a função pow() da biblioteca math.h para elevar dx e dy ao quadrado. Consertei multiplicando um pelo outro diretamente.
A inversão de eixos, que pensei que a princípio seria um problema, foi mais tranquila do que tinha imaginado. Para fazer retas decrescentes também entendi como deveria fazer relativamente rápido.



Problema de quando utilizei a função pow().
Chamada: DrawLine(150, 30, 100, 300, &cor1, &cor2);
tipoCor cor1 = {255, 0, 0}; tipoCor cor2 = {0, 0, 255};



Mesma linha da figura acima quando ajeitei o problema trocando a função pow(dx, 2) por dx*dx. A mesma troca foi utilizada com dy.

Fontes de consulta:

https://pt.slideshare.net/saikrishnatanguturu/computer-graphics-ver10
http://www.im.ufal.br/professor/thales/icg/Aula1.2.pdf
https://pt.wikipedia.org/wiki/Algoritmo_de_Bresenham
Slides da aula