[CG] [T1] Parte II: Rasterização de Retas

Publicado por CFLAVS em 9 DE MARÇO DE 2016

Esta é uma transcrição do conteúdo de título [CG] [T1] Parte II: Rasterização de Retas encontrado na página web: https://cflavs.wordpress.com/2016/03/09/cg-trabalho-individual-i-rasterizacao-de-retas/

A transcrição **é apenas para manter um registro pessoal caso o link deixe de existir** (e seu conteúdo seja perdido), **não há nenhuma intenção de plágio ou qualquer coisa do tipo**. Frisando: **eu não sou o autor deste material**.

Dando continuidade ao trabalho, após a rasterização de pontos, desenhar um segmento de tela na tela é um procedimento feito por meio de vários algoritmos tais como o Algoritmo Imediato [1] e Algoritmo Incremental Básico [2]. Entretanto, estes algoritmos possuem uma série de inconvenientes como a precisão no cálculo de determinadas retas (como as diagonais) e problemas de desempenho devido a aritmética realizada por eles.

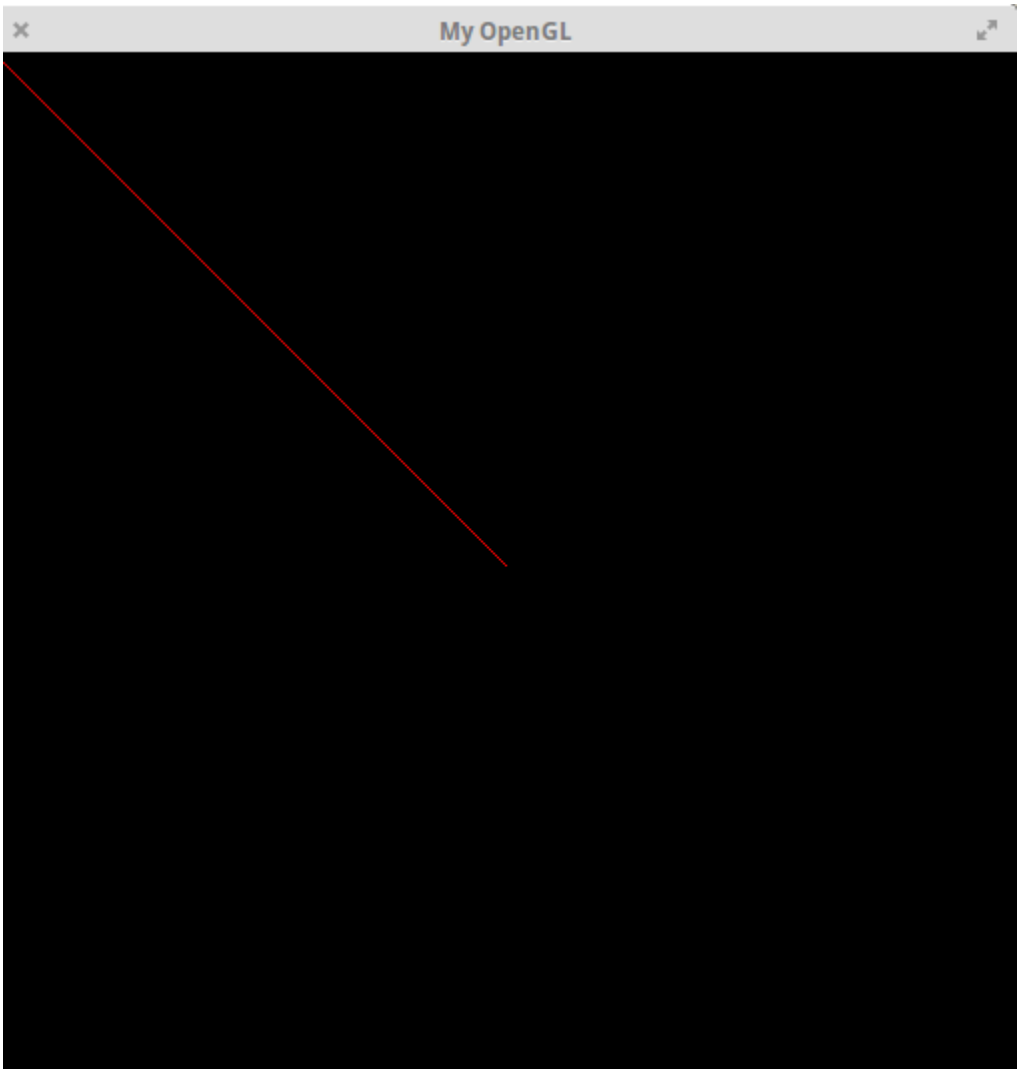
Sendo assim, será descrito aqui o algoritmo de Bresenham que reduz esses problemas por meio do ponto médio entre os dois vértices do segmento de reta desejado. Basicamente a ideia principal do algoritmo faz é calcular o ponto médio da reta desejada a partir da equação de reta $F(x,y) = ax + by + c$ onde, caso este retorne um valor positivo, significa que o ponto encontra-se abaixo da reta e deve ser selecionado o pixel logo acima da reta, caso contrário o pixel selecionado será aquele imediatamente abaixo da reta.

Baseado nisso, o algoritmo primeiramente calcula o valor de d, variável responsável por verificar se a diferença entre as distâncias Δx e Δy é positiva ou negativa, uma vez que o coeficiente angular de F(x,y) pode também ser representado por estas duas distâncias. Caso negativo, o pixel selecionado terá apenas incremento em x, caso contrário o incremento será nas duas coordenadas, selecionando o pixel acima do pixel atual. Esta verificação continua até que os dois vértices se encontrem no eixo x, isto é, a verificação ocorre enquanto $x1<x2$.

```
void Bresenham::simpleBresenham(bool declive, bool simetrico, int rgba[],int rgba1[]){
     int d = 2 * dy - dx;
    int incr_e = 2 * dy;
    int incr_ne = 2 * (dy - dx);
    int x = p0.getX();
    int y = p0.getY();
    PutPixel(x,y,rgba);
    while (x < p1.getX()) {
           if (d <= 0) {
               d += incr_e;
               x++;
           } else {
               d += incr_ne;
               x++;
               y++;
           }PutPixel(x,y,rgba);
    }
}
```

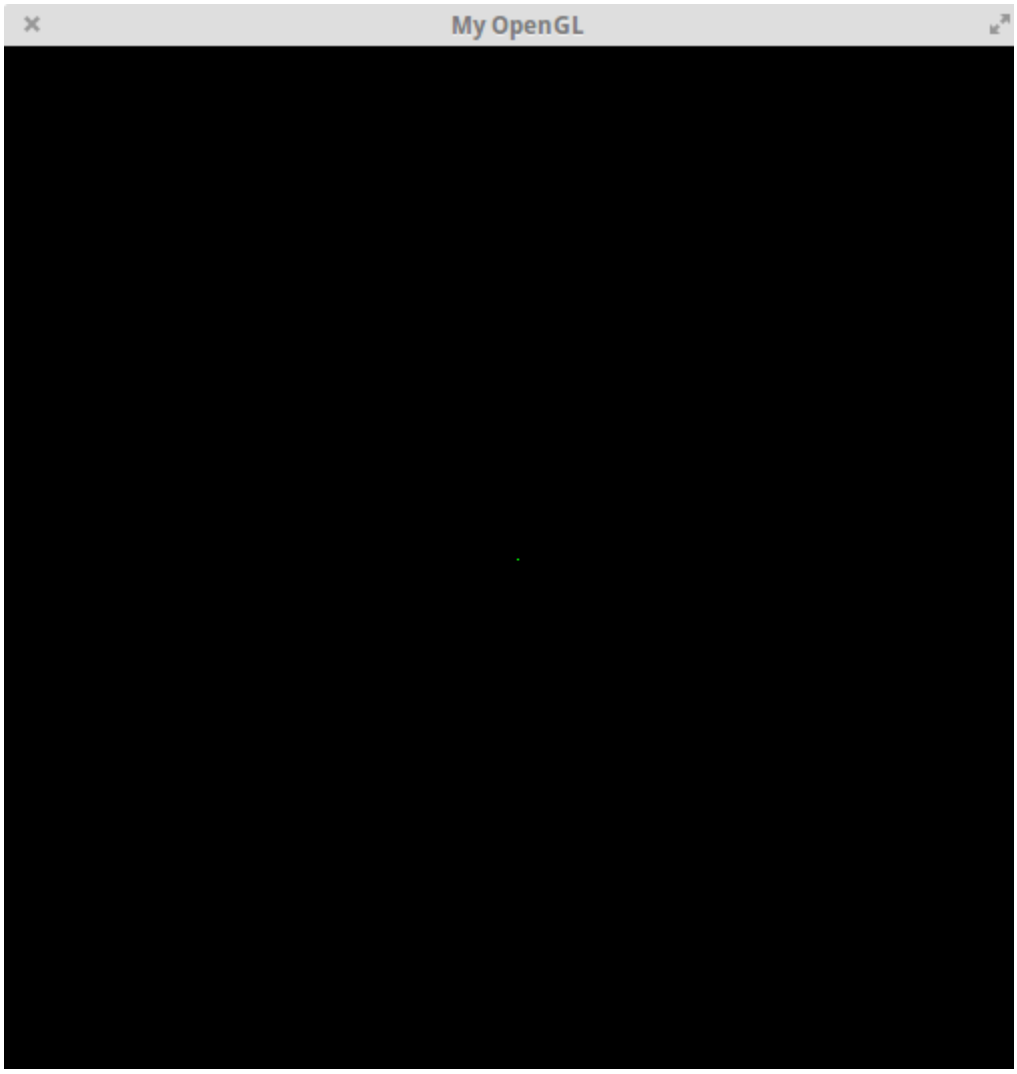
Algoritmo básico de Bresenham

Assim, se quisermos não apenas plotar um ponto nas coordenadas (0,0), como também traçar uma reta deste ponto a origem (256,256), utilizamos Bresenham gerando o seguinte resultado:



Rasterização de Reta no Oitavo Octante

Entretanto, percebe-se que embora o algoritmo funcione, este Bresenham torna-se muito básico e suscetível a falhas dependendo dos vértices colocados. Por exemplo, se quisermos rasterizar ao contrário, isto é, dos pontos (256,256) a (0,0) teremos não uma reta e sim um ponto.



Rasterização Incorreta dos Pontos (256,256) e (0,0)

Isto ocorre porque ao iniciarmos o Bresenham, como dito anteriormente, o algoritmo preenche os pixels enquanto x1<x2, dessa forma como x1 tem seu valor em 256 e x2 0, essa condição nunca é atingida de forma que o único ponto na tela é o ponto (256,256) do pixel inicial.

Para consertarmos isso, basta fazer uma pequena adaptação no algoritmo para que, se x1>x2, as coordenadas dos vértices sejam trocadas, fazendo assim x1<x2 novamente. No algoritmo proposto, isso pode ser feito por meio do método inverteVertice, onde primeiramente armazenamos as coordenadas x e y do primeiro vértice em uma variável temporária temp, para não perdermos o valor armazenado previamente, e em seguida setados os pixels novamente com suas coordenadas invertidas. Da mesma forma, o

ponteiro tempCor inverte os vetores com as componentes rgba. Por fim, os valores dx e dy são recalculados.

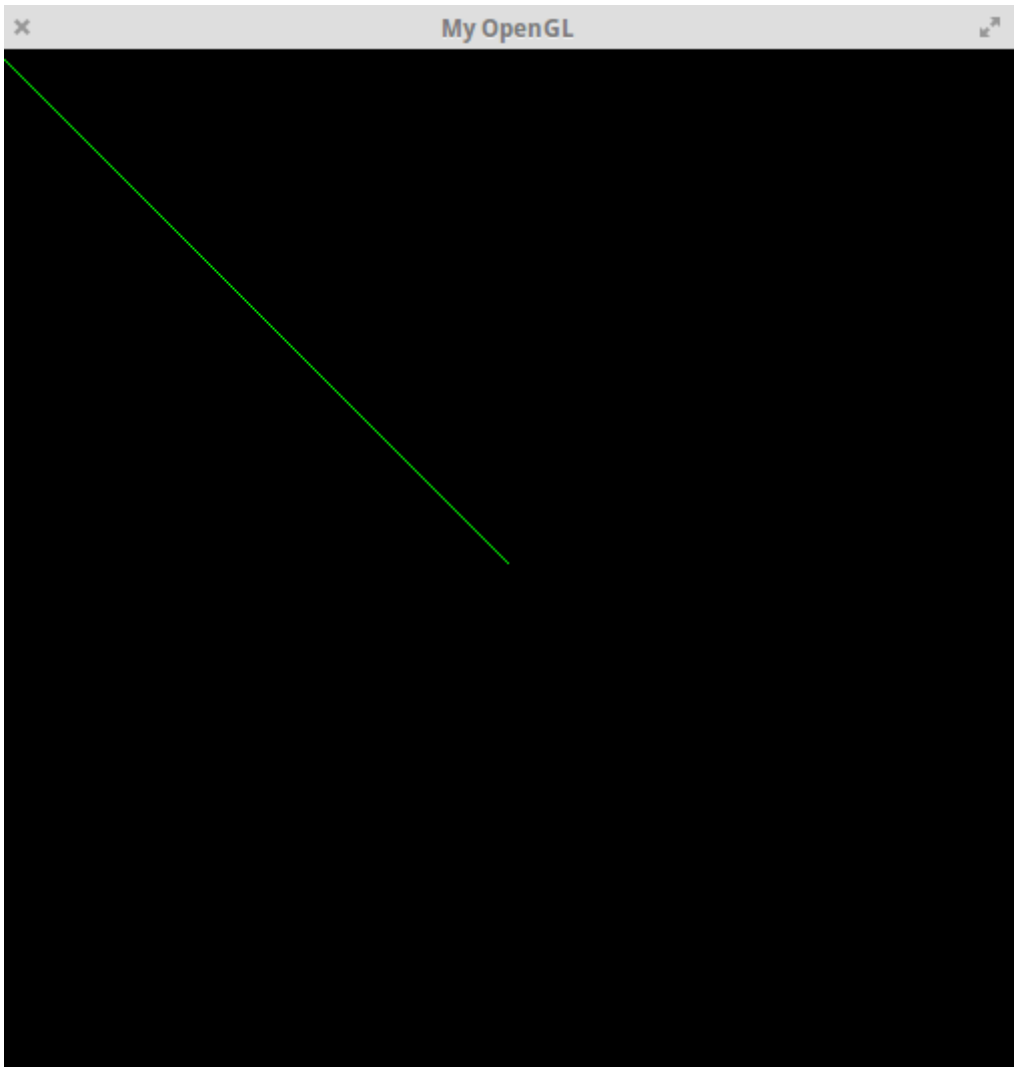
```
void Bresenham::inverteVertice(){
    //inverte os vertices
     int temp = p0.getX();
     p0.setX(p1.getX());
     p1.setX(temp);

     temp = p0.getY();
     p0.setY(p1.getY());
     p1.setY(temp);
     //inverte cores
     int *tempCor;
     tempCor = rgba;
     rgba = rgba1;
     rgba1 = tempCor;

     //recalcula dx e dy
     dx = p1.getX() - p0.getX();
     dy = p1.getY() - p0.getY();
}
```

Método inverteVertice()

Com esta alteração feita, podemos chamar o Bresenham normalmente e obteremos a reta agora rasterizada corretamente:



Rasterização Correta do Quarto Octante

Além dos valores de x e y invertidos, precisamos considerar que no sistema de coordenadas existem octantes. Entretanto, seguindo o algoritmo básico de Bresenham, plotaremos retas apenas nos octantes 4 e 8. Para fazermos a rasterização correta dos demais octantes, existem duas alterações a serem feitas.

Primeiramente, para transformar a reta da Figura anterior do octante 4 para o 5, as coordenadas mantem-se constantes em x e alteram-se apenas as coordenadas y. Assim, se quisermos ir das coordenadas (0,512) a (256,0) teremos a situação inversa do caso anterior, isto é, dx positivo enquanto dy negativo. Qual a consequência disso no algoritmo? O dy será negativo, fazendo com que os dois incrementos também sejam negativos. Isso faz com que ao iniciarmos o algoritmo, d será menor que 0 e, como os incrementos também são negativos, o valor do passo também será sempre negativo, fazendo com que o algoritmo fique sempre na condição d<=0 e assim, haverá incremento

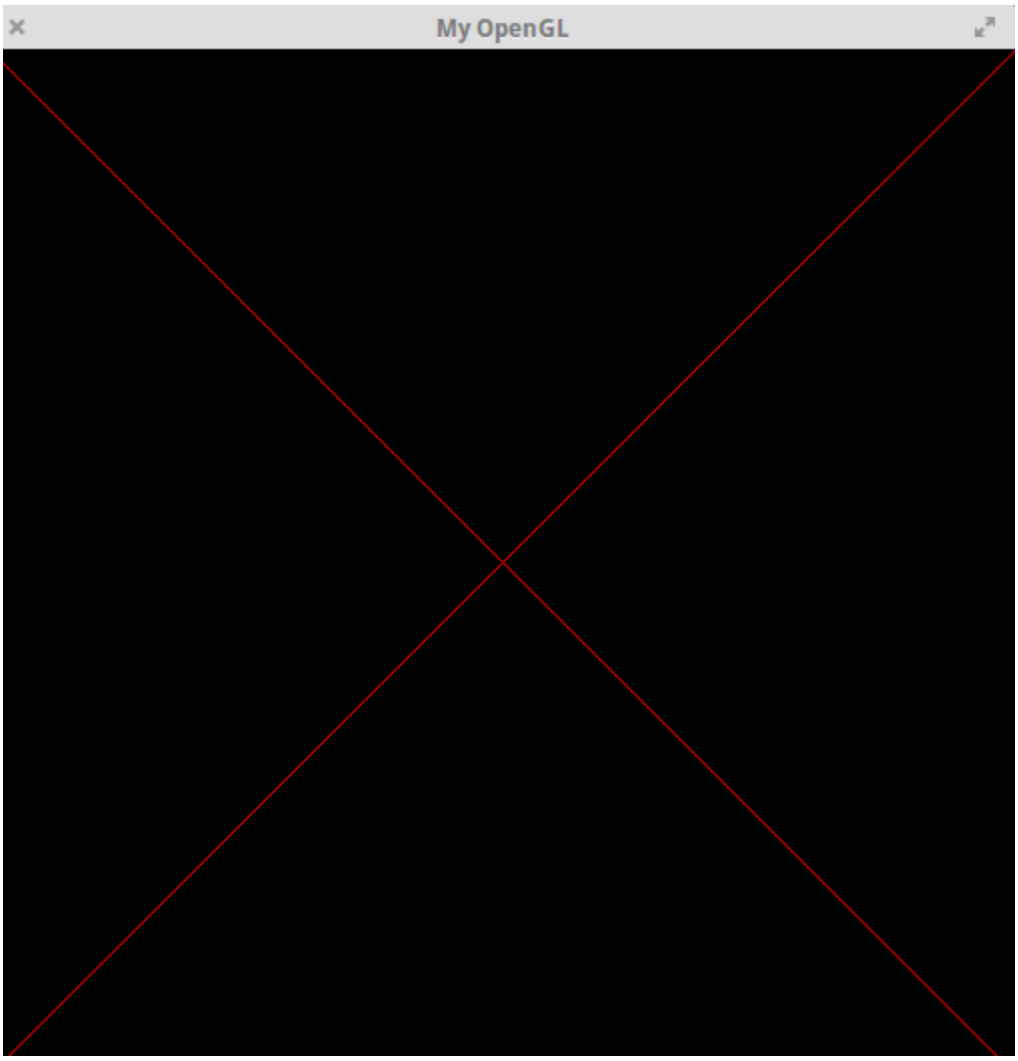
apenas em x e não em y, fazendo com que a reta não se desloque nesse eixo, resultando em uma reta constante.

Dessa forma, para revertemos o quadro basta apenas inicialmente atualizar o valor de dy pelo seu simétrico caso seja negativo, resultando em incrementos não necessariamente negativos. Além disso, como estaremos rebatendo do octante 4 para o 5 e o 8 para o 1, rebater um quadrante é apenas alterar o sinal em y, ou seja, ao invés de fazermos um incremento, faremos um decremento. Adaptando o algoritmo a esta condição, teremos:

```
void Bresenham::simpleBresenham(bool declive, bool simetrico, int rgba[],int rgba1[]){
    dy = -dy;
    int d = 2 * dy - dx;
    int incr_e = 2 * dy;
    int incr_ne = 2 * (dy - dx);
    int x = p0.getX();
    int y = p0.getY();
    PutPixel(x,y,rgba);
    while (x < p1.getX()) {
           if (d <= 0) {
               d += incr_e;
               x++;
           } else {
               d += incr_ne;
               x++;
               y--;
           }PutPixel(x,y,rgba);
    }
```

Bresenham quando dy menor que 0

Dessa forma, teremos agora retas em 4 octantes diferentes tal como demonstrado na Figura abaixo:



Rasterização de 4 Octantes

Nos quatro octantes restantes, além das situações tratadas previamente, temos que analisar também o coeficiente angular dado pela razão entre Δx e Δy por um motivo muito simples. Nestes octantes, o valor de Δy é superior ao de Δy, fazendo com que o coeficiente angular retorne um valor superior a 1 ou a -1. Qual a consequência deste coeficiente ter valor superior ao módulo de 1? Novamente retornando ao algoritmo de Bresenham, uma vez que o algoritmo se repete enquanto x1<x2, como Δy é maior, isso significa que quando tivermos x1>x2, ainda existirão pixels em y que não foram rasterizados, resultando em uma reta incompleta.

Para fazermos esta última adaptação, será feito algo bem semelhante ao cálculo quando Δx é negativo. Como precisamos agora satisfazer a razão entre Δx e Δy de forma a termos um coeficiente angular limitado superiormente e inferiormente em 1 e -1, se invertermos Δx e Δy atingiremos esta condição novamente. Dessa forma o que iremos fazer agora no

algoritmo por meio do método calcDeclive() é simplesmente fazer com que x1 seja igual a y1 e x2 seja igual a y2. Além de invertermos as coordenadas de cada ponto, para evitar valores negativos de Δx, multiplicamos o valor final das coordenadas x por (-1) conforme Figura abaixo.

```
void Bresenham::calcDeclive(){
    //inverte x com -y
     int temp = p0.getX();
     p0.setX(-p0.getY());
     p0.setY(temp);

     temp = p1.getX();
     p1.setX(-p1.getY());
     p1.setY(temp);

    //recalcula dx e dy
     dx = p1.getX() - p0.getX();
     dy = p1.getY() - p0.getY();
}
```

Método CalcDeclive()

Com isso acabamos o problema do coeficiente e, uma vez que as coordenadas foram alteradas, na hora de chamar o método PutPixel pós incrementos, precisamos inverter novamente as coordenadas, isto é feito por meio do método inverteCoordenadas.

```
void Bresenham::inverteCoordenadas(int x, int y, int rgba[]){
    PutPixel(y,-x,rgba);
}
```

Método inverteCoordenadas

Por fim, pode-se generalizar o algoritmo inicial para atender todos os casos aqui representados sem a necessidade de repetir código. Esta generalização foi feita aqui por meio de duas flags: simetrico e declive. Estas flags declaradas inicialmente falsas são responsáveis por atender as restrições de Δy e do coeficiente angular respectivamente.

No algoritmo final, o método calcBresenham, inicialmente faz a verificação se Δy é maior que Δx, em caso positivo a flag declive torna-se true e chama o método calcDeclive() previamente descrita, depois disso verifica se Δx é negativo e chama o método inverteVertice(). Por fim, se Δy for negativo multiplicamos o mesmo pelo seu simétrico e definimos simetrico também como true. Baseado nisso, o método finaliza chamando simpleBresenham que possui as duas flags como parâmetro para fazer o tratamento do algoritmo básico.

```
void Bresenham::calcBresenham(int x0, int y0, int x1, int y1, int rgba[],int rgba1[]) {
  //  Pixel p0,p1, pixelFinal;
    p0.setX(x0), p0.setY(y0);
    p1.setX(x1), p1.setY(y1);
    bool declive=false, simetrico=false;
    dx = p1.getX() - p0.getX();
    dy = p1.getY() - p0.getY();

    if (abs(dy) > abs(dx)) {
        declive=true;
       calcDeclive();}
    if (dx<0)
        inverteVertice();
    if (dy<0){
            dy = dy*(-1);
            simetrico = true;
            simpleBresenham(declive,simetrico,rgba,rgba1);
    }else
        simpleBresenham(declive,simetrico, rgba,rgba1);
}
```

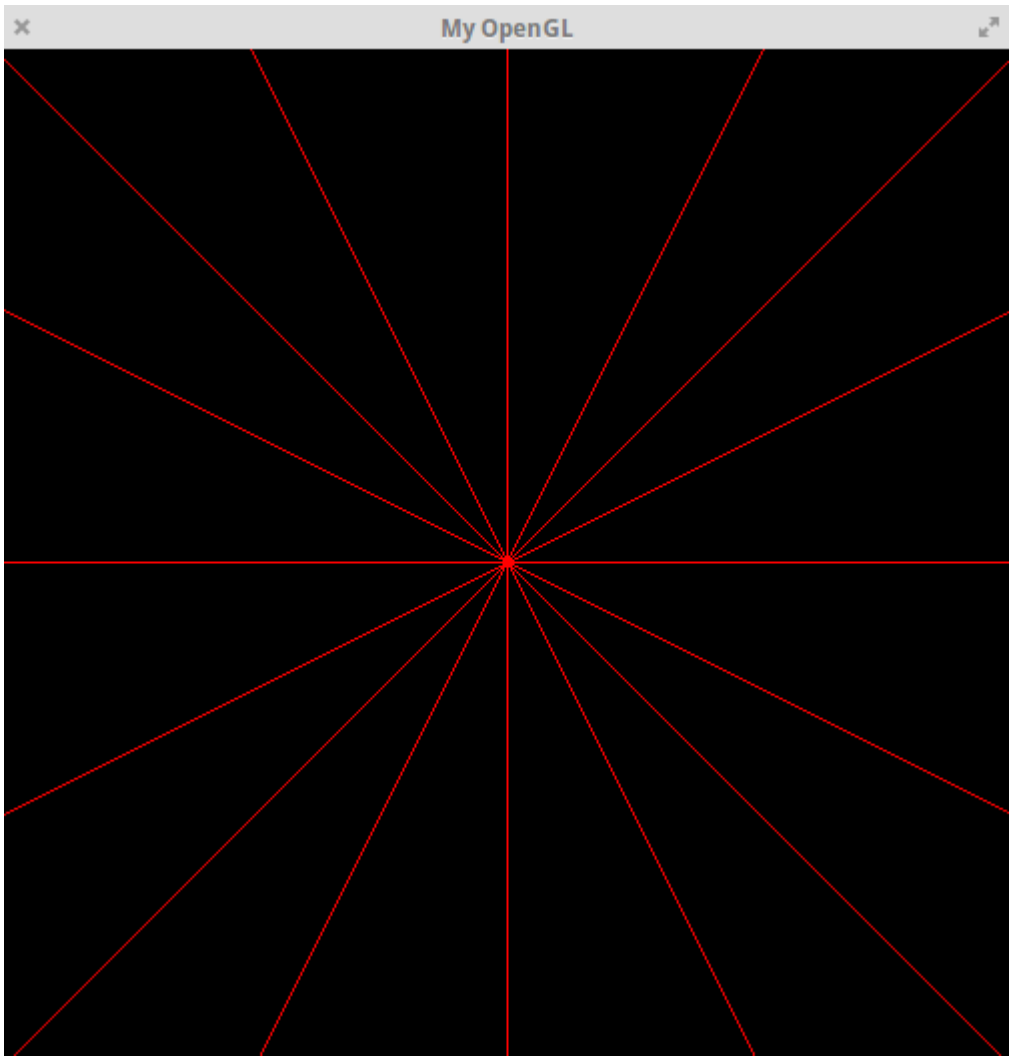
Para finalizar, o algoritmo genérico que atende todas as condições aqui discutidas foi implementado no método simpleBresenham. Como para o caso do coeficiente angular maior que módulo de 1 os pixels são invertidos, então o método PutPixel é tratado sempre que chamado. A outra alteração foi a inclusão de uma verificação da flag simetrico que, caso seja positiva, Δy é negativo e o valor de y deve ser decrementado para rebater a reta para o octante oposto.

```
void Bresenham::simpleBresenham(bool declive, bool simetrico, int rgba[],int rgba1[]){
    int d = 2 * dy - dx;
    int incr_e = 2 * dy;
    int incr_ne = 2 * (dy - dx);
    int x = p0.getX();
    int y = p0.getY();

    if(declive)
    inverteCoordenadas(x,y, rgbaFinal);
    else PutPixel(x,y,rgbaFinal);

     while (x < p1.getX()) {
           if (d <= 0) {
               d += incr_e;
               x++;
           } else {
               d += incr_ne;
               x++;
               if(simetrico)
               y--;
               else y++;
           }
           if(declive)
                inverteCoordenadas(x,y, rgbaFinal);
           else
                PutPixel(x,y,rgbaFinal);
    }
}
```

Com todas as alterações, podemos representar retas em todas as coordenadas possíveis na tela, conforme Figura abaixo:



Retas Desenhadas em Todos os Octantes

Dificuldades Encontradas

Como dificuldades para esta seção pode-se citar o compreendimento total de todas as condições dos octantes e o motivo de acontecer cada uma delas para correta implementação do algoritmo.

Referência

[CG] [T1] Parte II: Rasterização de Retas - Publicado por CFLAVS em 9 DE MARÇO DE 2016. https://cflavs.wordpress.com/2016/03/09/cg-trabalho-individual-i-rasterizacao-de-retas/

Acesso em: 21/05/2022