1-Introdução:

O último estágio do pipeline gráfico moderno é a renderização, nela estruturas matemáticas continuas são projetadas no espaço discreto do monitor através de algoritimos bem conhecidos. Um desses foi desenvolvido em 1962 por um funcionário da IBM chamado Bresenham; seu método implementa interpolação discreta de pontos utlizando apenas de operações inteiras, permitindo eficiente rasterização de primitivas em baixo nível.

Foi aqui implementado o algorítimo mencionado com o objetivo de rasterizar triângulos a partir de três vertices bi-dimencionais e suas respectivas cores.

2-O algoritmo:

```
float dJ = j2-j1;
float dI = i2-i1;
float m = dJ/dI;        // Slope
writePixel(i1, j1);
float y = j1;
int i, j;
for ( i=i1+1; i<=i2; i++ ) {
    y = y+m;
    j = round(y);        // Round to nearest integer
    writePixel( i, j );
}
```

A base do algoritimo de Bresenham, ilustrada acima, consiste em interpolar os valores da segunda variavel através de um arredondamento de **f(x)** – função da reta real – para cada valor de **x,** que é incrementado a cada iteração. Os problemas mais evidentes são dois: a presença de operações de ponto flutuante em uma função iterativa implica uma baixa performance – em especial historicamente -, além disso, essa implementação funciona apenas para vetores do primeiro octante. Segue a re-implementação utilizando apenas de operações inteiras.

```
int deltaX = i2-i1;
int thresh = deltaX/2;          // Integer division rounds down
int ry = 0;
int deltaY = j2 - j1;
writePixel( i1, j1 );
int i;
int j = j1;
for ( i=i1+1; i<=i2; i++ ) {
    ry = ry + deltaY;
    if ( ry > thresh ) {
        j = j + 1;
        ry = ry - deltaX;
    }
    writePixel( i, j );
}
```

2.1- Implementação da generalização:

O algorítimo original de Bresenham funciona para casos em que o vetor diferença entre o ponto final e inicial faz parte do primeiro octante, ou seja, faz parte do seguinte conjunto:

$$\Omega = \{(x,y) \mid x > y,\ y/x <= 1,\ x >= 0,\ y >= 0\}$$

Uma possível implementação da versão generalizada do mesmo, portanto, consiste em desenvolver uma transformação $F:R^2 \rightarrow \Omega$ que, quando aplicada nos pontos de interesse **(x0,y0), (x1,y1)**, retorna dois pontos **(x0',y0'),(x1',y1')** que conformam os requisitos do algoritimo original.

A rasterização é então calculada nesse novo espaço e, imediatamente antes da escrita no Color Buffer, é aplicada uma transformação inversa F^{-1}, permitindo assim a aplicação do algoritimo de Bresenham em linhas retas de qualquer natureza.Segue a implementação das diversas partes de F:

```
namespace Transformations{

   //Transformations to be applied according to input vector's octant
   glm::mat2 transformations[8] = {{ 1, 0, 0, 1},
                                   { 0, 1, 1, 0},
                                   { 0,-1, 1, 0},
                                   {-1, 0, 0, 1},
                                   {-1, 0, 0,-1},
                                   { 0,-1,-1, 0},
                                   { 0, 1,-1, 0},
                                   { 1, 0, 0,-1}};

   //Inverses
   glm::mat2 inverse_transform[8] ={{ 1, 0, 0, 1},
                                    { 0, 1, 1, 0},
                                    { 0, 1,-1, 0},
                                    {-1, 0, 0, 1},
                                    {-1, 0, 0,-1},
                                    { 0,-1,-1, 0},
                                    { 0, -1,1, 0},
                                    { 1, 0, 0,-1}};
};
```

Cada elemento de *Transformations::transformations[n-1]* é uma matriz 2x2 que implementa a transformação necessaria para um vetor que está no n-ésimo octante.

É necessario também descobrir qual octante original do vetor, o que pode ser feito com um fluxo logico basico:

```
inline int getOctant(Position p){
   if(abs(p.y) > abs(p.x)){
      if(p.x<0)
         return (p.y<0)? 6 : 3;
      else
         return (p.y<0)? 7 : 2;
   }
   else{
      if(p.x<0)
         return (p.y<0)? 5 : 4;
      else
         return (p.y<0)? 8 : 1;
   }
}
```

2.1: Interpolação de cores:

Como a generalização do algoritmo foi feita de tal forma que todo pixel a ser desenhado sempre está no primeiro octante – exceto no momento de desenha-lo,podemos, no loop do algoritimo base, operar supondo que **(x1 - x0)** pixels vão ser desenhados, pois esse é o comportamento natural do algoritimo de Bresenham. Dado isso, para calcularmos o largura do passo da interpolação de cada cor basta fazer **k=(corFinal – corInicial)/(x1-x0).** A interpolação é feita então adicionando **c * k * (corInicial)** à cor inicial, onde **c** é o indice da iteração;

```
//Color interpolation
stepR = (b_.color.r - a_.color.r)/(float)dX;
stepG = (b_.color.g - a_.color.g)/(float)dX;
stepB = (b_.color.b - a_.color.b)/(float)dX;
colorI.r = color0.r + c*stepR;
colorI.g = color0.g + c*stepG;
colorI.b = color0.b + c*stepB;
////////////////////
```

2.3: O algoritimo completo:

O codigo fonte completo está disponível no repositório onde este arquivo foi originalmente encontrado.E-mail para contato: *vinicius_gbapb@hotmail.com*

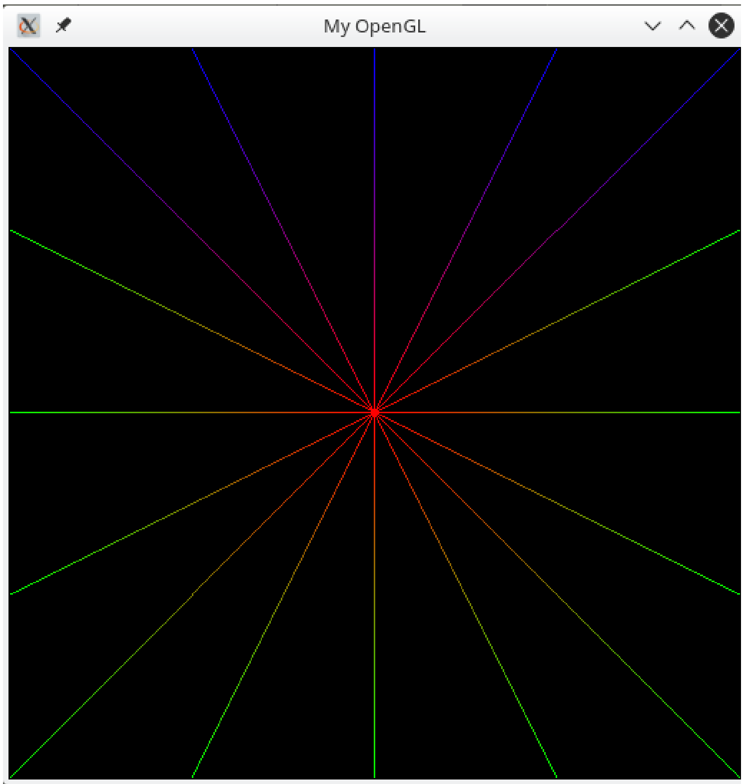
3: Desenhando triangulos:

Agora que sabemos rasterizar linhas e interpolar cores, o desenho de um triangulo é trivial:

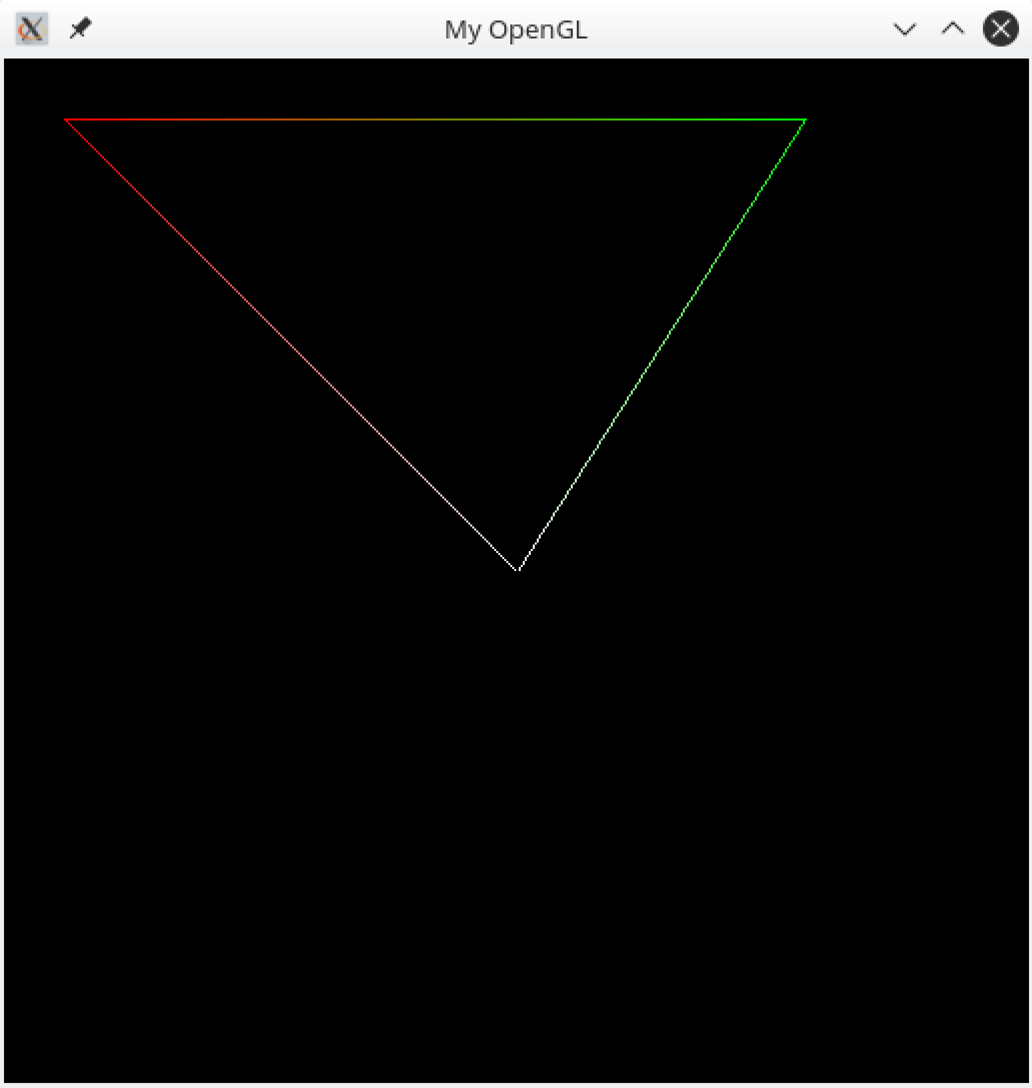
```
void Triangle::draw(){
   drawLine(vertices[0], vertices[1]);
   drawLine(vertices[1], vertices[2]);
   drawLine(vertices[2], vertices[2]);
}
```

4: Resultados:

Para a chamada ./cgprog 4:



Para a chamada ./cgprog 30 30 255 0 0 400 30 0 255 0 256 256 255 255 255



5- Referencias:

Bungartz, H., Griebel, M., & Zenger, C. W. (2004). *Introduction to computer graphics*. Hingham, MA: Charles River Media.