UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE INFORMÁTICA

CIÊNCIA DA COMPUTAÇÃO

Nome: Gabriel Leite Santana **- Matrícula:** 2016000284

Nome: Drayton Corrêa Filho **- Matrícula:** 2016058088

Relatório – Trabalho Preparatório I

João Pessoa

2018

INTRODUÇÃO

Esse relatório faz parte do primeiro trabalho preparatório da disciplina Introdução à Computação Gráfica, ministrada pelo professor Christian Azambuja Pagot, no período 2017.2. Sendo o objetivo do trabalho a rasterização de pontos e linhas na linguagem C++ através das bibliotecas GLUT e OpenGL utilizando o framework disponibilizado pelo próprio professor.

RASTERIZAÇÃO DE PONTOS

Para efetuar tal etapa foi necessário fornecer as coordenadas x e y do ponto e, através disso e do ponteiro *FBptr*, definir os componentes de cor R, G, B e A (Red, Blue, Green e Alpha) de cada pixel utilizado. Sendo esse procedimento feito na função *putPixel* cujo código é exibido logo abaixo.

```
void putPixel(Pixel pixel){

        FBptr[(4*(pixel.x + (pixel.y*IMAGE_WIDTH))) + 0] = pixel.R;
        FBptr[(4*(pixel.x + (pixel.y*IMAGE_WIDTH))) + 1] = pixel.G;
        FBptr[(4*(pixel.x + (pixel.y*IMAGE_WIDTH))) + 2] = pixel.B;
        FBptr[(4*(pixel.x + (pixel.y*IMAGE_WIDTH))) + 3] = pixel.A;
}
```

RASTERIZAÇÃO DE LINHAS

O algoritmo utilizado para rasterização foi o algoritmo de *Bresenham*, que consiste em um algoritmo incremental. O ponto chave do algoritmo de *Bresenham* consiste na decisão de qual pixel deve ser pintado na tela (se é o pixel imediatamente à frente do atual ou o pixel à “diagonal” do atual), de forma que cada caso possuí um tratamento. A maior dificuldade se encontra em generalizar esse algoritmo para todas as octantes do plano da tela, pois cada octante possui um cálculo diferente. Felizmente, podemos trabalhar com simetria entre os octantes, e o resultado é o obtido abaixo:

```
Pixel DrawLine(Pixel pi, Pixel pf){

        if(abs(pf.x - pi.x) > abs(pf.y - pi.y)){
                if(pf.x > pi.x){
                        return DrawLineDown(pi, pf);
                }else{
                        return DrawLineDown(pf, pi);
                }
        }else{
                if(pf.y > pi.y){
                        return DrawLineUp(pi, pf);
                }else{
                        return DrawLineUp(pf, pi);
                }
        }
```

```
Pixel DrawLineDown(Pixel pi, Pixel pf){

        Pixel p = pi;
        int flag_y = 1;
        int dx = pf.x - pi.x;
        int dy = pf.y - pi.y;
        int d = 2 * dy - dx;

        if(dy < 0){
                flag_y = -1;
                dy = -dy;
        }

        for(p.x; p.x <= pf.x; p.x+=1){
                putPixel(p);
                if(d > 0){
                        d -= 2 * dx;
                        p.y = p.y + flag_y;
                }
                d += + 2 * dy;
                ColorInterpolate(pi, &p, pf);
        }
        p.x--;
        p.y--;
        return p;
}
```

```
Pixel DrawLineUp(Pixel pi, Pixel pf){

        Pixel p = pi;
        int flag_x = 1;
        int dx = pf.x - pi.x;
        int dy = pf.y - pi.y;
        int d = 2 * dx - dy;

        if(dx < 0){
                flag_x = -1;
                dx = -dx;
        }

        for(p.y; p.y <= pf.y; p.y+=1){
                putPixel(p);
                if(d > 0){
                        d -= 2 * dy;
                        p.x = p.x + flag_x;
                }
                d += 2 * dx;
                ColorInterpolate(pi, &p, pf);
        }
        p.x--;
        p.y--;
        return p;
}
```

INTERPOLAÇÃO DE CORES

Quando começamos a trabalhar com linhas, esbarramos num problema: de que cor pintar os pixels ao longo da linha? Como solução para isso, podemos usar a Interpolação Linear das cores, que é baseada no princípio da equação vetorial de uma reta, onde tendo dois pontos da reta, podemos obter um parâmetro t, onde 0 <= t <= 1 para obtermos todos os pontos da reta. Assim, podemos usar esse parâmetro para variar as cores do pixel ao longo da reta, conforme a distância dos pontos iniciais. Implementando, temos o algoritmo abaixo:

```
void ColorInterpolate(Pixel pi, Pixel *p, Pixel pf){
        int dx = pf.x - pi.x;
        int dy = pf.y - pi.y;
        int dx_aux = pf.x - p->x;
        int dy_aux = pf.y - p->y;
        double t =  sqrt(pow(dx_aux, 2) + pow(dy_aux, 2)) / sqrt(pow(dx, 2) + pow(dy, 2));
        p->R = pf.R + (pi.R - pf.R) * t;
        p->G = pf.G + (pi.G - pf.G) * t;
        p->B = pf.B + (pi.B - pf.B) * t;
        p->A = pf.A + (pi.A - pf.A) * t;
}
```

DESENHO DE TRIÂNGULOS

A parte final constituiu-se de criar uma função *DrawTriangle* que utilizou da função *DrawLine* de tal forma que fosse repetida três vezes recebendo três pontos os quais se conectavam para construir a figura geométrica.

```
void DrawTriangle(Pixel v1, Pixel v2, Pixel v3){
        DrawLine(v1, v2);
        DrawLine(v2, v3);
        DrawLine(v3, v1);
}
```

Por fim, para exibir a figura foram escolhidos os pontos (50, 50), (500, 25), (300, 500) e suas respectivas cores foram a vermelha, branca e azul. Sua geração é feita na *main* através da função *MyGLDraw* cujo resultado efetivo pode ser visto abaixo.

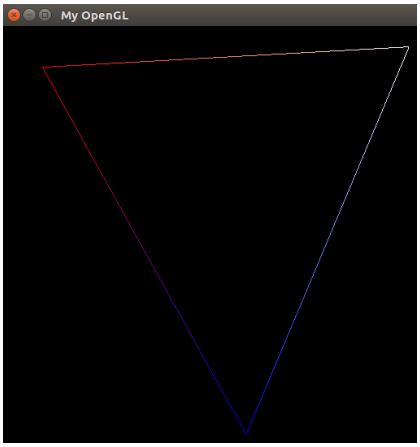
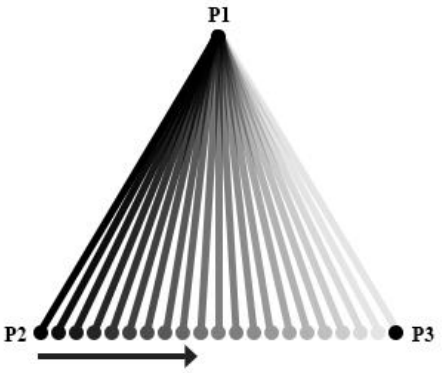


Figura 1 – Exibição do triângulo com DrawTriangle

PREENCHIMENTO DE TRIÂNGULOS:

Após o término daquilo que foi especificado no trabalho, partimos para uma etapa extra que consistiu de preencher os triângulos com uma interpolação de cores. Para isso, partimos do princípio de que para preencher qualquer triângulo seria necessário dividi-lo em duas partes mais simples de colorir, sendo essa simplificação feita ao separarmos o triângulo maior em dois triângulos que possuam uma de suas arestas paralela à um dos eixos.

Ao ter esse conceito em mente, buscamos fazer uma função que preenchesse um triângulo cuja aresta inferior fosse paralela ao eixo x. Para isso pegamos os três vértices da figura e formamos o desenho a partir de apenas uma aresta, mantendo um de seus pontos fixos, nesse caso o superior, e variando a coordenada x do outro ponto ao mesmo tempo que era construída uma nova reta. Como ilustrado pela imagem ao lado. A interpolação de cores muda a cada nova reta gerada no algoritmo, o que produz o resultado exibido abaixo.



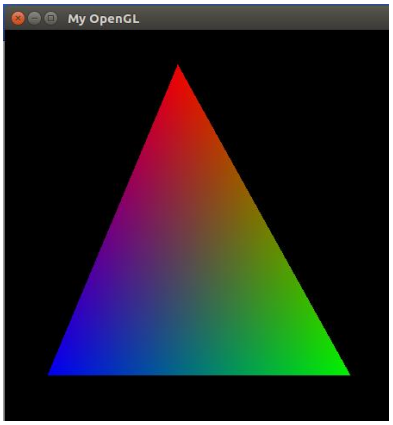


Figura 2 – Exibição do triângulo preenchido com DrawFilledTriangleUpper

Em seguida, tivemos de criar outra função que, dessa vez, a aresta superior fosse paralela ao eixo x. O conceito de sua execução é, em essência, o mesmo da anterior, a única mudança é que nessa foi fixado o ponto inferior e aquele a variar foi o x do ponto superior a cada nova geração da reta.

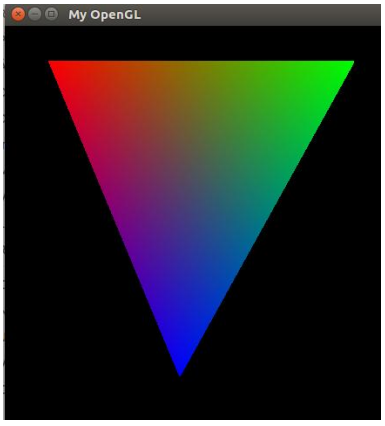
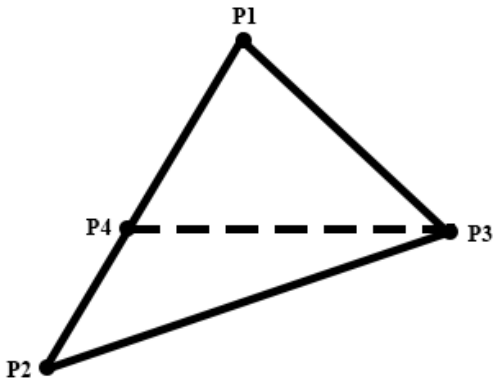
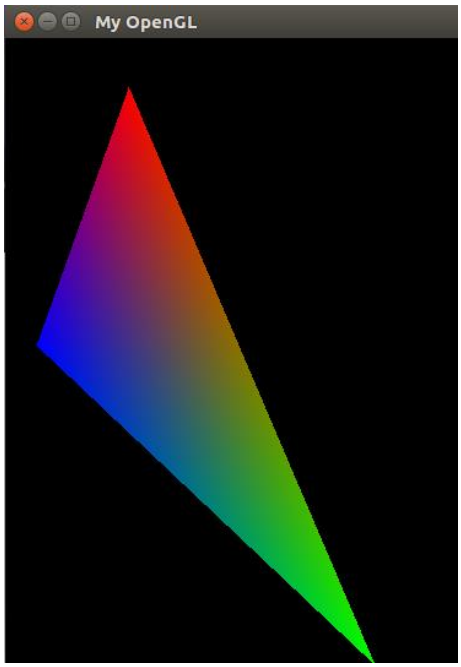


Figura 3 – Exibição do triângulo preenchido com DrawFilledTriangleLower

Feito isso, bastou unirmos ambos os casos criando um algoritmo que pudesse preencher qualquer triângulo. Em tal função, checamos se o y não varia na aresta superior ou na inferior (já que é possível usar diretamente uma das outras duas funções caso o y de uma dessas duas retas não varie), se ele variar em ambas, é criado um novo ponto que recebe a posição do ponto médio de uma das arestas laterais para produzir uma nova aresta paralela ao eixo x fazendo, assim, a divisão do triângulo em dois mais simples para que possam ser preenchidos por ambas as funções criadas anteriormente, *DrawFilledTriangleUpper* e *DrawFilledTriangleLower*.



Para que a interpolação das cores ocorra corretamente ela é apenas feita uma vez tomando como base P4 cuja coordenada x variará em ambos os triângulos, inferior e superior. Isso faz com que a transição fique correta e seja exibido o resultado abaixo.



REFERÊNCIAS BIBLIOGRÁFICAS

https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html

https://pt.wikipedia.org/wiki/Interpola%C3%A7%C3%A3o_linear