Universidade Federal da Paraíba
Introdução à Computação Gráfica
Trabalho 1 - 2018.2

Gabriel Marques Barbosa
gabrielbarbosa@mat.ci.ufpb.br
CV Lattes: http://lattes.cnpq.br/4804534049787814

13 de Fevereiro de 2019

Conteúdo

1 Introdução

Este trabalho consiste na implementação de técnicas básicas da Rasterização requeridas na disciplina de Introdução à Computação Gráfica. São estas: Rasterização de um ponto, de uma linha e de um triângulo. Para tais, serão abordados seus aspectos, a ideia de sua implementação, a interpolação de cores, eventuais problemas encontrados e soluções dos mesmos. Além disso, duas funcionalidades extras foram implementadas e serão discutidas, são elas: a rasterização de um triângulo preenchido com cor única especificada pelo usuário e a de um arco circular, bem como a circunferência completa, esta por sua vez, sem preenchimento de cores.

O trabalho foi desenvolvido utilizando uma framework, disponibilizada pelo docente responsável pela disciplina, que simula o acesso à escrita direta na memória de vídeo, através de um ponteiro que referencia a posição de início da mesma. A linguagem adotada foi C++.

2 Desenvolvimento

O processo de rasterizar uma figura na tela do computador consiste em acender um determinado conjunto de pixels na tela com determinada cor. O conjunto de pixels acesos neste processo, combinados, cada um com sua cor, caracterizará e dará forma à figura desejada.

2.1 Rasterização de pontos e implementação do método PutPixel()

Matematicamente, uma forma de representar um ponto em $\mathbb{R}^2$ é através de suas coordenadas x e y, isto é, $P = (x_P, y_P)$, onde x_P e y_P correspondem às coordenadas x e y do ponto P.

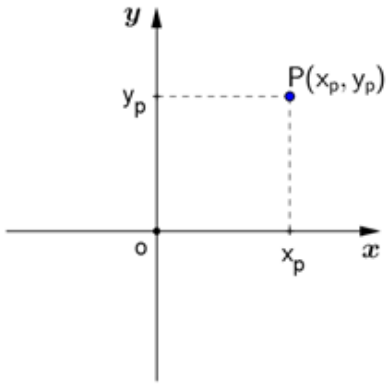


Figura 1: Representação de um ponto no plano

Agora, imaginemos a tela do computador como nosso plano $\mathbb{R}^2$, desta forma, o pixel seria o nosso ponto, representado pelas suas coordenadas x e y da tela, entretanto, o eixo y agora é visto "apontando" para baixo.

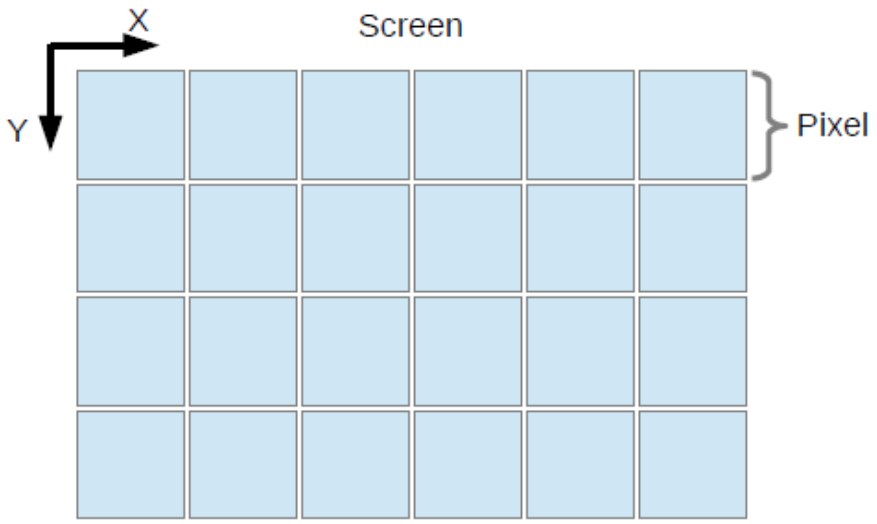


Figura 2: Ilustração dos pixels na tela de um computador

Então, para desenharmos um ponto na tela, basta que informemos suas coordenadas e a cor do mesmo. Esta tarefa será realizada pelo método **PutPixel()**, que receberá como parâmetro tais coordenadas e a cor que o pixel deve ter. Para cor, o formato **RGBA** (Red, Green, Blue, Alpha) será utilizado, assim, para cada pixel, 4 bytes serão usados para definir sua cor. Matematicamente, pode-se obter uma fórmula para o endereço (variável que foi chamada de adr, do inglês *"address"*) da memória referente a cor de um pixel qualquer:

$$adr = x \cdot 4 + y \cdot w \cdot 4 \tag{1}$$

onde w corresponde à largura (*width*) da tela e esta memória tem característica sequencial na framework utilizada, isto é, sabendo o endereço da cor R, o de G será esse endereço acrescido de 1, o de B será acrescido de 2 e o de Alpha acrescido de 3. O parâmetro Alpha foi considerado sempre o mesmo, 255 (256 bits, 0 à 255).

Para implementar esta funcionalidade, criou-se uma ***struct*** *Point* para definir um ponto. Tal possui dois atributos do tipo inteiro, sendo estes suas coordenadas x e y. Também foi criada uma ***struct*** *Color*, com seus atributos sendo os valores R, G, B e A que formam a cor. O trecho de código pode ser visto na figura abaixo:

```
///****Estrutura Point, parametros pos_x (posição em x) e pos_y (posição em y)****
struct Point{
    int pos_x, pos_y;
    Point(int pos_x, int pos_y){
        this->pos_x = pos_x;
        this->pos_y = pos_y;
    };
    Point(){
    }
    bool operator <(Point & p){
        return pos_y<p.pos_y;
    }
};
///*************************************************************************************

///****Estrutura Color****
struct Color{
    unsigned char R,G,B,A;
};
///***********************
```

Figura 3: Trecho de código relacionado as estruturas point e color

O **bool operator** será discutido mais a frente, seu uso será na parte da implementação de uma das funcionalidades extras. Agora temos o que precisamos para poder rasterizar um ponto na tela, já que o **PutPixel()** receberá o ponto e a cor. O endereço da memória é obtido em função do ponto passado. O trecho de código é ilustrado abaixo:

```
///****Rasterizar Ponto na Tela****
void PutPixel(Point p, Color c){
    int address = p.pos_x*4 + p.pos_y*4*IMAGE_WIDTH;
    FBptr[address] = c.R;
    FBptr[address+1] = c.G;
    FBptr[address+2] = c.B;
    FBptr[address+3] = 255;
}
///*********************************
```

Figura 4: Trecho de código do método PutPixel()

A seguir um exemplo será mostrado, usando como parâmetro os pontos v_1, v_2, v_3 e v_4 (à esquerda na figura 5) e as cores c_1, c_2, c_3 e c_4 (à direita na figura 5). Cada c_i corresponde a cor do ponto v_i. A chamada da função e a definição dos parâmetros são feitas em uma função presente na framework, chamada de **MyGlDraw()**. A notação v para os pontos foi pensada no sentido de vértice, o que se fará claro o uso nos métodos que serão apresentados mais a frente neste trabalho.

```
Point v1;
v1.pos_x = 255;
v1.pos_y = 10;

Point v2;
v2.pos_x = 501;
v2.pos_y = 501;

Point v3;
v3.pos_x = 10;
v3.pos_y = 255;

Point v4;
v4.pos_x = 10;
v4.pos_y = 501;
```

```
Color c1;
c1.R = 255;
c1.G = 0;
c1.B = 0;
c1.A = 255;

Color c2;
c2.R = 0;
c2.G = 255;
c2.B = 0;
c2.A = 255;

Color c3;
c3.R = 0;
c3.G = 0;
c3.B = 255;
c3.A = 255;

Color c4;
c4.R = 255;
c4.G = 255;
c4.B = 0;
c4.A = 255;
```

Figura 5: Parâmetros usados no exemplo de funcionamento do PutPixel()

Um laço **for** foi usado neste exemplo apenas para ele prosseguir desenhando os pontos até um certo valor, apenas a caráter ilustrativo. A cada iteração do laço ele varia a posição dos pontos em uma quantidade fixa.

```
for(int j=0; j<100;j++){
    PutPixel(v1,c1);
    PutPixel(v2,c2);
    PutPixel(v3,c3);
    PutPixel(v4,c4);
    v1.pos_y += 5;
    v3.pos_x += 5;
    v2.pos_y -= 5;
    v2.pos_x -= 5;
    v4.pos_x += 5;
    v4.pos_y -= 5;
}
```

Figura 6: Chamada da função PutPixel()

E o resultado deste exemplo é mostrado na figura 7, logo abaixo.

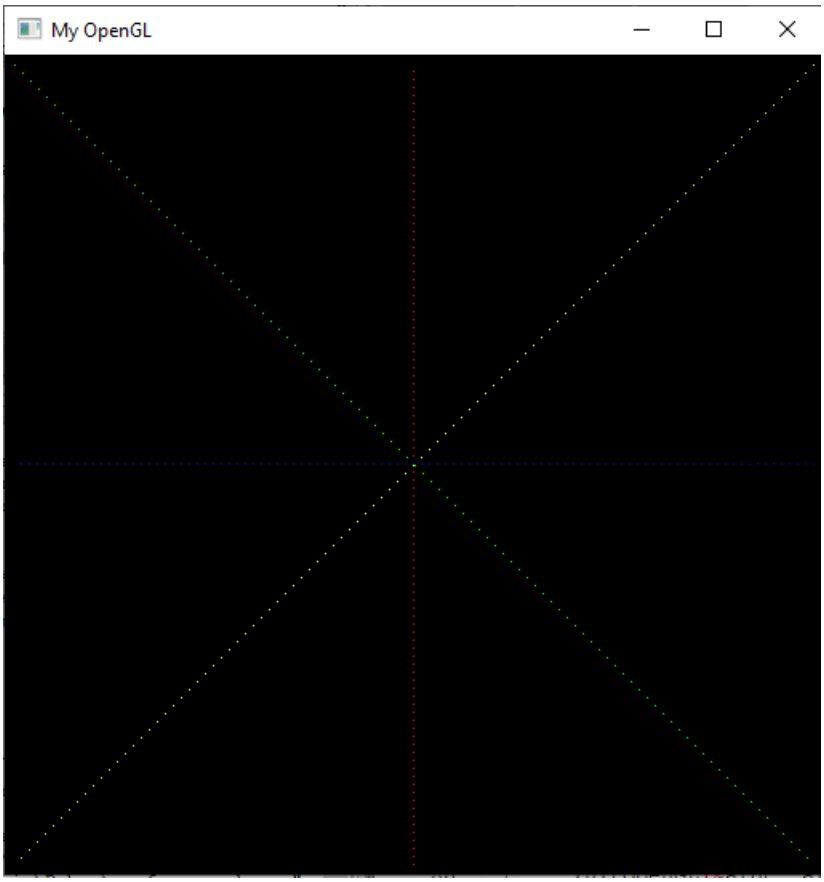


Figura 7: Resultados do exemplo do uso do PutPixel() sob os parâmetros previamente especificados

2.2 Rasterização de linhas e implementação do método DrawLine()

A seguir será discutido um método para rasterização de linhas retas na tela. Para tal, foi implementada a função **DrawLine()** baseada no algoritmo de Bresenham. Esta função terá como argumento os vértice inicial e final, do tipo ***struct*** *Point* e as cores respectivas. Exemplo: DrawLine(Point v1, Color c1, Point v2, Color c2) desenhará uma reta de v1 até v2, com c1 sendo a cor de v1 e c2 a de v2.

2.2.1 Algoritmo de Bresenham

A ideia por trás do algoritmo de Bresenham é utilizar uma variável de decisão que é recalculada a cada passo. Esta variável determina onde será desenhado

o próximo ponto. Se o valor desta variável, chamemos de d, for positivo, as coordenadas x e y serão incrementadas, isto é, o ponto será desenhado no pixel de coordenadas $P = (x + 1, y + 1)$. Se não, apenas o x será acrescido. Isso é ilustrado na figura abaixo:

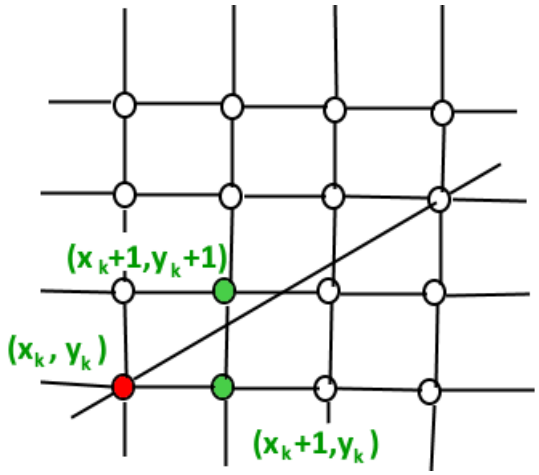


Figura 8: Representação do algoritmo de Bresenham

Podemos ver que o algoritmo consiste em variar o x de forma crescente e aumentar ou não o y, mas isso faz sentido apenas para linhas cuja inclinação sejam menores que a função identidade ($y = x$), ou seja, apenas para inclinações entre 0 e 45 graus.

Chamemos de dx a variação em x, isto é, $dx = x_{k+1} - x_k$, e de dy a variação em y, ou seja, $dy = y_{k+1} - y_k$. Temos 3 casos em que o algoritmo encontra problemas para desenhar as linhas:

- quando $dx < 0$, ou seja, $x_{k+1} < x_k$
- quando $dy < 0$, ou seja, $y_{k+1} < y_k$
- quando $dx < dy$

No primeiro caso, suponha que a reta será traçada partindo de um vértice v_1 até um vértice v_2 e que $dx < 0$. Para resolver isso, basta inverter os argumentos, ou seja, traçar a reta partindo de v_2 até v_1. A reta será a mesma, mas agora não teremos mais $dx < 0$. O trecho de código é mostrado abaixo:

```
if (v2.pos_x<v1.pos_x){
    DrawLine(v2,c2,v1,c1);
    return;
}
```

Figura 9: Solução para o caso $dx < 0$

No segundo caso, precisamos inverter o sinal de dy e decrementar o y, em vez de incrementá-lo. Para isso, uma variável inteira, chamemos de **signal** será usada. Inicializamos ela com valor 1, e quando ocorrer $dy < 0$, mudamos esta variável para -1 e acrescentamos o trecho $dy = \textbf{signal} * dy$ e, no laço que for pintando cada pixel, y será somado com **signal**. Assim o problema está resolvido.

```
if(v1.pos_y>v2.pos_y){
    signal = -1;
    dy = signal*dy;
}
```

Figura 10: Mudança de sinal de dy no caso em que $dy < 0$

No terceiro caso, o algoritmo é basicamente o mesmo, porém, os valores de dx serão substituídos pelos de dy e vice-versa.

2.2.2 Interpolação linear de cores

Para a interpolação linear de cores, foi feita uma função chamada **ColorInterpolation()**. Esta função recebe como parâmetros um valor que, digamos, representaria a "distância da cor de um vértice para a do outro", intuitivamente falando a caráter "didático", e as cores inicial e final (cor do vértice inicial e do final, respectivamente).

A cor atual em um determinado pixel presente na reta desenhada pela **DrawLine()** pode ser obtida como:

```
Color ColorInterpolation(float colorDist, Color ci, Color cf){

    Color actualColor;
    actualColor.R = ci.R - colorDist*(ci.R - cf.R);
    actualColor.G = ci.G - colorDist*(ci.G - cf.G);
    actualColor.B = ci.B - colorDist*(ci.B - cf.B);
    actualColor.A = 255;

    return actualColor;
}
```

Figura 11: Interpolação de cores

Onde *colorDist* é calculado a cada iteração no laço onde a reta será desenhada pixel a pixel.

```
count++;
colorDist = ((float)count/(float)dx);
actualColor = ColorInterpolation(colorDist, c1, c2);
```

Figura 12: Fator de variação de cor

Um exemplo, com os vértices $v_1 = (420, 50)$, $v_2 = (255, 450)$, $v_3 = (90, 50)$, $v_4 = (450, 280)$ e $v_5 = (60, 280)$ é ilustrado abaixo:

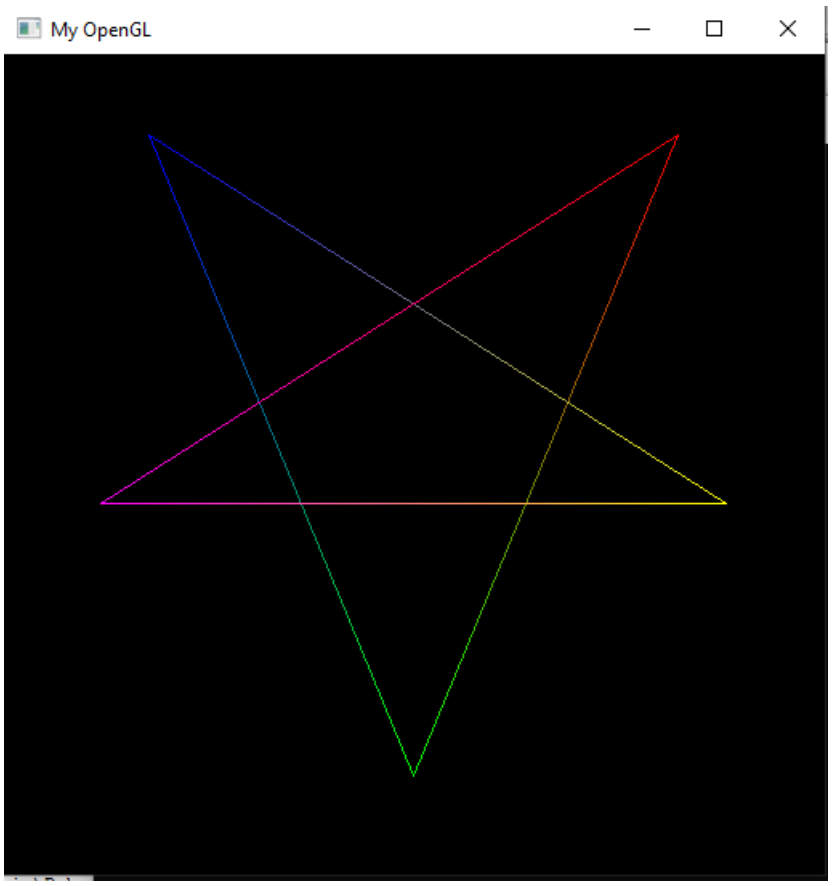


Figura 13: Resultado do DrawLine()

2.3 Rasterização de triângulos e implementação do método DrawTriangle()

Agora que tem-se capacidade de rasterizar uma linha, para rasterizar um triângulo, sem preenchimento de cores, o método **DrawTriangle()** apenas consiste na chamada do método **DrawLine()** para rasterizar 3 linhas que, juntas, formam a figura de um triângulo.

Mantendo os vértices do exemplo mostrado na seção anterior, vamos rasterizar um triângulo composto pelos vértices (v_1, v_2, v_5) e outro com vértices (v_1, v_2, v_3). O resultado é mostrado na figura abaixo:

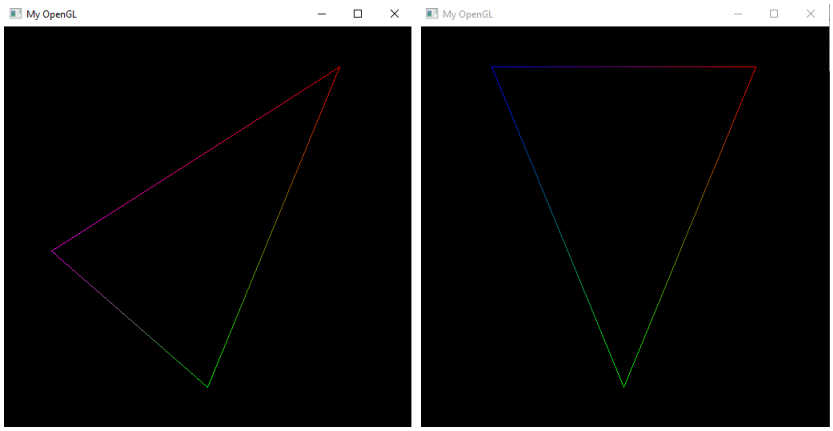


Figura 14: Resultado do DrawTriangle() para os parâmetros citados

3 Funcionalidades extras desenvolvidas

3.1 Rasterização de um triângulo preenchido com cor

Como um extra, o interesse agora é preencher internamente um trinângulo com uma cor. Uma breve pesquisa nos leva a 2 métodos para preenchimento de polígonos: o método *Bounding Box* e o *Scanline*. Aqui será abordado o *Scanline*, por ser considerado mais eficiente.

A ideia por trás do *Scanline* é, como o nome sugere, "escanear" a figura e desenhar linhas internas a cada nível, chamando a função **DrawLine()** com os parâmetros sendo dois pontos auxiliares.

Temos 3 casos a serem considerados:

1. **base plana, ou seja, vértices alinhados na base (*flat bottom triangle*).**

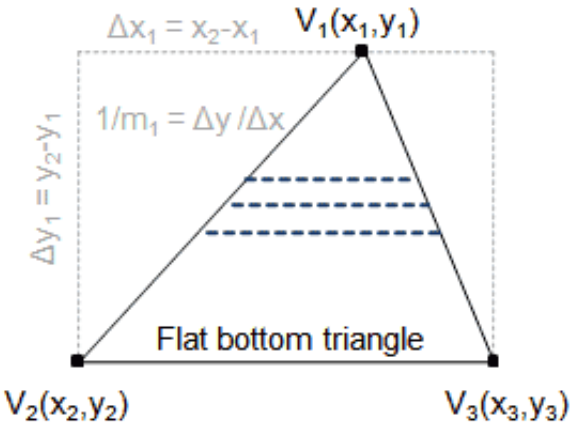


Figura 15: *Flat bottom triangle*

Neste caso, defina 2 variáveis auxiliares, x_{begin} e x_{end} que serão as coordenadas x a cada nível do *Scanline*. Estas coordenadas de x serão inicializadas com as coordenadas x do vértice mais alto, v_1 no caso da figura acima.

A coordenada y nesse caso é incrementada por 1 a cada iteração, isto é, o *scan* é feito no sentido de descida e a cada nível variamos as duas variáveis auxiliares a partir das inclinações das arestas que incidem em v_1. Assim somos capazes de saber se "andamos" para esquerda ou para direita em cada uma dessas variáveis.

2. **topo plano, ou seja, vértices alinhados no topo (*flat top triangle*).**

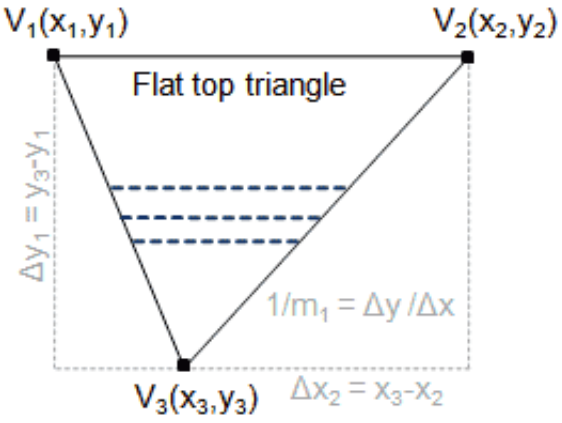


Figura 16: *Flat top triangle*

Neste caso, o procedimento é análogo ao caso anterior, entretanto, agora o vértice que será a referencia para as variáveis auxiliares será o vértice que está abaixo, no caso da figura acima, v_3. Repare que agora, para variar as linhas, y será decrementado, para que o *scan* seja no sentido de subida.

3. **Caso geral para um triângulo genérico.**

   Neste caso, precisamos de um vértice a mais v_4. Este vértice dividirá nosso triângulo em 2, um do tipo *flat bottom* e outro do tipo *flat top*, agora com v_4 como vértice comum de ambos. Agora basta aplicar os 2 casos acima para os novos triângulos correspondentes a estes casos.

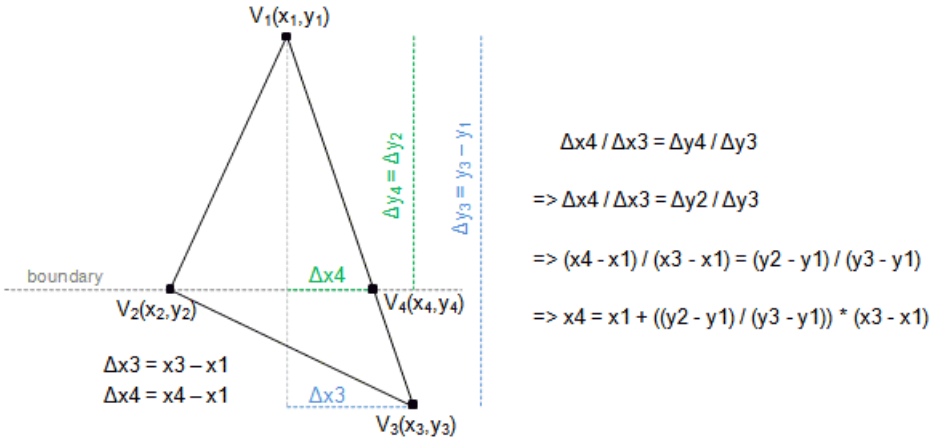


Figura 17: Triangulo genérico

Note que a coordenada y de v_4 coincide com a de v_2 (irá coincidir com algum dos vértices, neste caso foi o v_2), então só é necessário obter a coordenada x, como ilustrado acima. Uma observação é que o Scanline requer que os vértices sejam ordenados de acordo com a coordenada y, daí a necessidade de se ter o operador "menor que" (**bool operator** visto na figura 3).

Podemos ver o resultado na figura abaixo:

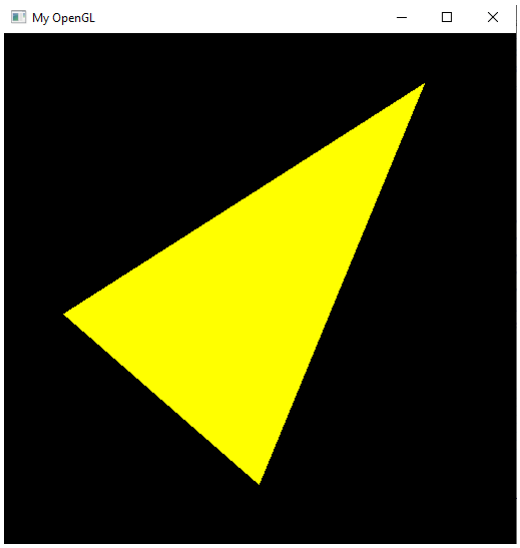
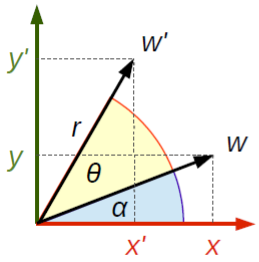


Figura 18: Triangulo preenchido com cor via *Scanline*

3.2 Rasterização de Arcos Circulares

Um outro adicional de interesse é uma forma de rasterizar uma circunferência e arcos circulares quaisquer. Para isso, usa-se uma outra forma de representar um ponto no plano: as coordenadas polares. Este sistema de coordenadas é capaz de descrever a posição de um ponto, definido um centro, um raio e um ângulo.



$$x = r\cos(\alpha)$$
$$y = r\sin(\alpha)$$

$$x' = r\cos(\alpha+\theta)$$
$$y' = r\sin(\alpha+\theta)$$

Figura 19: Coordenadas polares

Assim, para desenhar uma circunferência, ou arcos de circunferência, basta que apliquemos coordenadas polares sobre um ponto inicial. Uma observação é que as funções seno e cosseno em C/C++ precisam do ângulo em radianos.

A implementação seria, a partir do ponto que representa o centro da circunferência, obter o ponto inicial a partir do raio e todos os outros pontos serão o ponto atual acrescido da variação das coordenadas polares. O trecho de código pode ser visto abaixo:

```
float toRadians(float degrees) {
  return (degrees * 3.1415926535) / 180.0f;
}

void DrawCircleArc(Point center, float radius, Color cl, float angle, float angle_var){


    PutPixel(center, cl);

    float angle_aux = 0.0f;

    while(angle_aux <= angle) {

        float theta = toRadians(angle_aux);

        int x = center.pos_x + round(radius * cos((theta+angle_var)));
        int y = center.pos_y - round(radius * sin((theta+angle_var)));

        PutPixel(Point(x,y), cl);
        angle_aux += 0.5f;

    }

}
```

Figura 20: Código para rasterização de uma circunferência via coordenadas polares

Note que como seno e cosseno são funções limitadas no intervalo $[-1, 1] \in \mathbb{R}$, o "jogo de sinais" feito nas 2 linhas do código acima em que essas funções são usadas foi arranjado para que o desenho fosse feito no sentido anti-horário, começando a direita do centro, apenas por conveniência matemática.

A seguir, alguns exemplos serão ilustrados, onde para todos eles foi fixado o raio com valor 60, entretanto, apenas a título ilustrativo, um laço foi feito para, a cada arco desenhado, incrementar o raio por um valor fixo e desenhar um novo arco com uma outra cor, até um certo valor limitante.

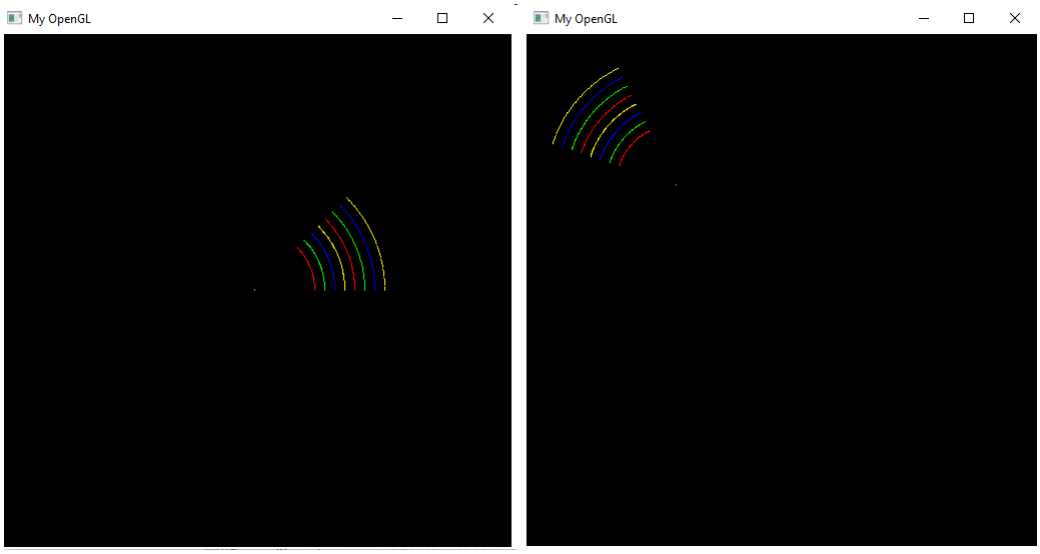


Figura 21: Arcos desenhados com rasterização usando coordenadas polares

Neste exemplo, os centros foram $(255, 255)$ na figura da esquerda e $(150, 150)$ na figura da direita. Os ângulos iniciais foram 0 e 45. Os ângulos dos arcos foram, respectivamente, 60 e 45.

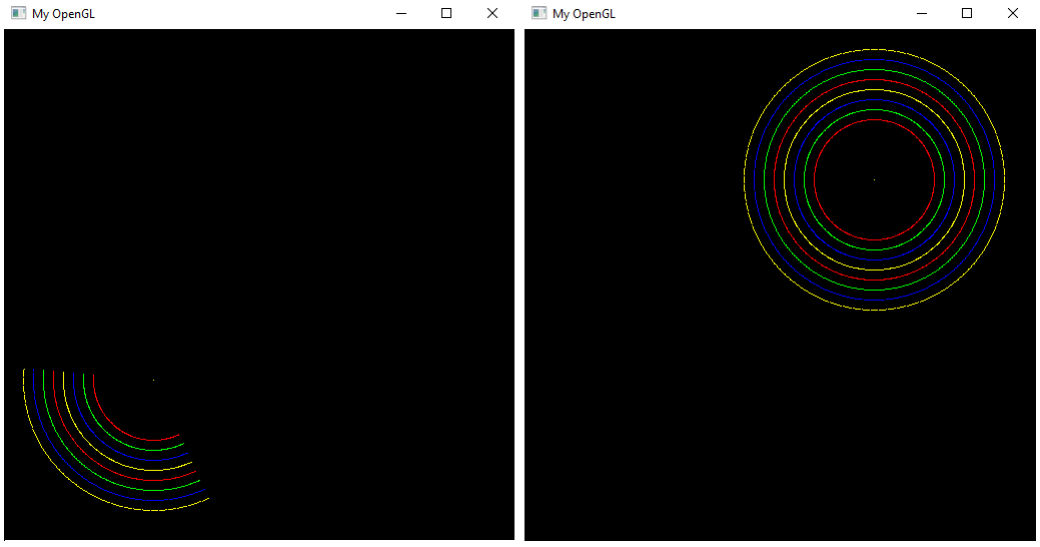


Figura 22: Arcos desenhados com rasterização usando coordenadas polares

Neste exemplo, os centros foram $(150, 350)$ na figura da esquerda e $(350, 150)$

na figura da direita. Os ângulos iniciais foram 170 e 0. Os ângulos dos arcos foram, respectivamente, 120 e 360.

4 Discussões

Pôde-se ver que o algoritmo de Bresenham tem muita utilidade, entretanto em sua forma mais básica ele tem certas limitações, como abordado ao longo desse trabalho. Entretanto ele é bastante eficaz e diversas variações dele são desenvolvidas para rasterizar figuras mais gerais, devido a sua eficiência e gama de aplicações.

Neste trabalho, o maior desafio consistiu em observar estas limitações e fazer as devidas adaptações no código para lidar com elas. Isto acaba se tornando mais claro quando se olha um pouco mais matematicamente para a ideia do algoritmo.

Nas atividades extras, a ideia da rasterização da circunferência utilizando coordenadas polares, embora simples, não parece ser a alternativa mais eficaz, visto que há cálculos de funções trigonométricas várias vezes. Uma versão sem uso desse cálculo "pesado" seria bem mais viável, principalmente no contexto de trabalhos que demandam muito processamento, o que não era o nosso caso.

5 Referências

1. PAGOT, C. A. Rasterization. Slides de aula de Introdução à Computação Gráfica - Lecture 2. 2018. 30 slides.

2. PAGOT, C. A. Geometric Transformations. Slides de aula de Introdução à Computação Gráfica - Lecture 6. 2018. 37 slides.

3. `https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/`. Último acesso: 14/01/2019.

4. `https://www.cs.drexel.edu/~david/Classes/CS536/Lectures/L-16_ScanlineRendering.pdf`. Último acesso: 10/02/2019.

5. `http://www-users.mat.uni.torun.pl/~wrona/3d_tutor/tri_fillers.html`. Último acesso: 08/02/2019.

6. `http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html`. Último acesso: 08/02/2019.