

**UNIVERSIDADE FEDERAL DA PARAÍBA**

**DEPARTAMENTO DE INFORMÁTICA**

**INTRODUÇÃO A COMPUTAÇÃO GRÁFICA**

**CIÊNCIA DA COMPUTAÇÃO**

**Aluno**: Vinícius Medeiros Wanderley **Matrícula:** 20160133667

**Aluno**: José Ítalo Alves de Oliveira Vitorino **Matrícula:** 2016005182

**TRABALHO 2 – RASTERIZAÇÃO EM C/C++ (ALGORITMO DE BRESENHAM)**

**Introdução:**

Devemos implementar um algoritmo para a rasterização de pontos e linhas. Além de fazer triângulos que serão ser desenhados através da rasterização das linhas que compõem suas arestas.

A rasterização destas primitivas será feita através da escrita direta na memória de vídeo. Como os sistemas operacionais atuais protegem a memória quanto ao acesso direto, utilizaremos um framework que simula o acesso à memória de vídeo.

Usaremos um framework desenvolvido pelo prof. Christian Azambuja Pagot pois os sistemas da atualidade não permitem acesso direto à memória de vídeo. Ele tem como objetivo simular a memória de vídeo, o frame buffer e o color buffer.

**O Algoritmo:**

Antes de tudo, vamos criar duas estruturas para o algoritmo, que que vão nos auxiliar durante o projeto, Color (Cor) e Coordenadas:

typedef struct Color{

float red;

float green;

float blue;

float alpha;

}tCor;

typedef struct Coordenadas{

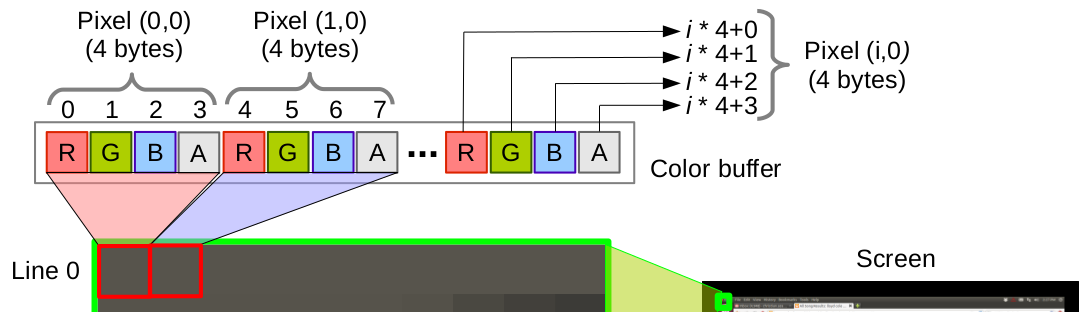
int coordenadaX;

int coordenadaY;

}tCoordenadas ;

**Pixel:**

Começaremos com o básico, um Pixel geralmente é representado por 4 bytes, cada byte representa respectivamente Red(Vermelho) , Green(Verde) , Blue(Azul) e Alpha(Transparência ou Brilho), cada um pode assumir de 0 até 255 , o alpha varia de 0 a 1. Os 4 componentes misturados produzem uma gama de cores suficiente para os projetos realizados nos computadores.



**Colorindo um Pixel:**

Na memória do computador a tela é representada de forma contínua, ela é uma linha que tem 4\*largura\*altura bytes de tamanho, como queremos pintar um pixel de coordenadas (x,y) é necessário fazer 4\*x + 4\*y\*largura , acessamos o primeiro elemento do pixel , ou seja , o vermelho.

void PutPixel(int coordenadaX, int coordenadaY, tCor \*cor){

int byte = (coordenadaX\* 4) + (coordenadaY \* IMAGE\_WIDTH \* 4);

FBptr[byte] = cor->red;

FBptr[byte + 1] = cor->green;

FBptr[byte + 2] = cor->blue;

FBptr[byte + 3] = cor->alpha;

}

Colocando os conceitos em algoritmo, temos o exemplo anterior onde Fbptr é um ponteiro para o início da memória e Color um dos struct que criamos e contem 4 floats e cada um é um componente RGBA

Chamando a função PutPixel no main.cpp:

tCor cor1 = {255,50,50,255};//definimos a cor que queremos

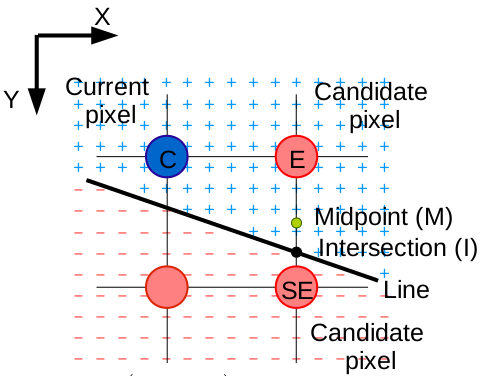
PutPixel (250,250, &cor1) ;//definimos as coordenadas e usamos a cor

O resultado ampliado é o seguinte:



**Algoritmo de Bresenham:**

O algoritmo de Jack Elton Bresenham, é baseado na ideia do ponto médio. E tem como objetivo selecionar o número mínimo de pixel para a linha ficar fina, buscando seguir o modelo matemático.



Cada círculo da imagem representa o ponto central de um pixel, entre os círculos existe um ponto médio (Midpoint M) exatamente no meio entre cada círculo, baseado nesse ponto o algoritmo decide qual pixel será escolhido para ser aceso. Caso a reta passe por baixo do ponto médio, a próxima posição com relação ao pixel atual (x, y) será (x+1, y), por outro lado, se a reta passar por cima, o próximo pixel será o (x+1, y+1), garantindo que o próximo pixel a ser aceso será o mais próximo da reta.

**Triângulo e Quadrado Usando Bresenham:**

O triangulo é bem mais fácil de fazer, pois ele é baseado no algoritmo da linha, vamos fazer 3 chamadas para desenhar 3 linhas , a partir dos vértices passados nos parâmetros.

void drawTriangle(Coordenadas \* pontoInicial, Color \* cor1, Coordenadas \* pontoIntermediario, Color \* cor2, Coordenadas \* pontoFinal, Color \* cor3) {

DrawLineNoBresenham(pontoInicial,pontoIntermediario,cor1,cor2);

DrawLineNoBresenham(pontoIntermediario,pontoFinal,cor2,cor3);

DrawLineNoBresenham(pontoFinal,pontoInicial,cor3,cor1);

}

Para chamar a função no main precisamos definir previamente as cores e coordenadas.

Coordenadas \* p0 = (struct Coordenadas \*) malloc(sizeof(Coordenadas));

Coordenadas \* p1 = (struct Coordenadas \*) malloc(sizeof(Coordenadas));

Coordenadas \* p2 = (struct Coordenadas \*) malloc(sizeof(Coordenadas));

p0->coordenadaX = 50;//definimos as coordenadas dos pixels

p0->coordenadaY = 350;

p1->coordenadaX = 200;

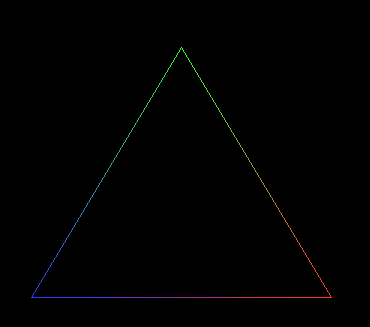
p1->coordenadaY = 100;

p2->coordenadaX = 350;

p2->coordenadaY = 350;

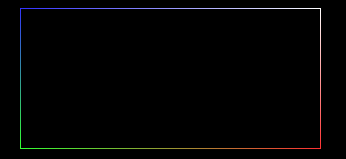
drawTriangle(p0,&cor3,p1,&cor2,p2,&cor1);//finalmente chamamos a função

**Resultado do drawTriangle:**



Como podemos perceber quando damos zoom normalmente as linhas diagonais começam a ficar serrilhadas, pois ao trazer do plano matemático para o computacional, ou seja, no caso do infinito para o finito, ocorrem algumas perdas.

O quadrado é feito de forma semelhante ao triangulo, apenas adicionando mais uma coordenada e cor.

****

**Interpolação de Cores:**

O método de Interpolação de cores foi bem simples de fazer, inicialmente pedimos duas cores, uma inicial e outra final, assim que recebemos nós usamos uma variável auxiliar para receber a cor inicial.

Color \* corAuxiliar = (struct Color \*) malloc(sizeof(Color));

corAuxiliar->red = corInicial->red;

corAuxiliar->green = corInicial->green;

corAuxiliar->blue = corInicial->blue;

corAuxiliar->aplha = corInicial->alpha;

Em seguida, determinamos o deltaR, deltaG, deltaB e deltaA, que é a variação de cor do ponto inicial ao ponto final, esses deltas são obtidos pela subtração dos valores finais pelos iniciais.

deltaR = (corFinal->red - corAuxiliar->red);

deltaG = (corFinal->green - corAuxiliar->green);

deltaB = (corFinal->blue - corAuxiliar->blue);

deltaA = (corFinal->alpha - corAuxiliar->alpha);

Após isso dividimos todos esses deltas obtidos pelo deltaX que é a variação da linha, e obtemos a quantidade de cor que varia de pixel para pixel de acordo com o desenho da linha, e podemos ter uma mudança de cor uniforme chegando do totalmente vermelho até o totalmente verde.



**Desempenho:**

Não há dúvidas de que Bresenham foi brilhante na construção desse algoritmo , definitivamente desempenha seu papel , mesmo com alguns problemas esperados como linhas serrilhadas que podem ser contornados com técnicas como Anti-Aliasing.

**Problemas Encontrados:**

Um dos principais problemas foi a implementação do algoritmo de Bresenham, que requereu estudo sobre o assunto para conseguir implementar corretamente. Além desse houveram outros como erro de segmentação, causado por um detalhe na hora de alocar a memória das estruturas em C, mas resolvemos usando a função malloc ().

Outro foi a dificuldade de encontrar o algoritmo de Bresenham generalizado para se basear. Felizmente achamos um trabalho que citava o livro Computer Graphics: Principles and Practice em C, o qual usaremos como referência para trabalhos futuros, pois é bastante completo para a cadeira de Introdução à Computação Gráfica. O algoritmo que trazemos é uma mistura de fragmentos que encontrei nesse livro e na internet.

**Referencias:**

- Slides do professor Cristian

- Foley, J. (2010). Computer graphics. 2nd ed. Boston [u.a.]: Addison-

Wesley.

- [https://medium.com/@filhojoseildo/implementa%C3%A7%C3%A3o-de-algoritmos-de-rasteriza%C3%A7%C3%A3o-ef413aaccf3d](https://medium.com/@filhojoseildo/implementação-de-algoritmos-de-rasterização-ef413aaccf3d)

- <https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html>

- <https://bitunico.wordpress.com/2012/12/16/rasterizacao-em-cc-algoritmo-de-bresenham/>