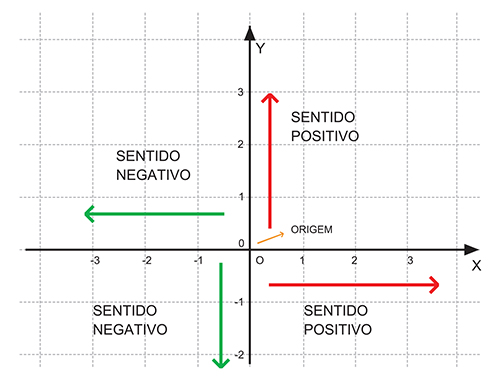
RASTERIZAÇÃO DE PRIMITIVAS

**Introdução**

O objetivo aqui é demonstrar como fazer rasterização de primitivas sem o auxílio dos comandos openGL, com objetivo didático de mostrar como funcionam as coisas "por trás". Toda via, o "não uso" de comandos openGL se dá apenas quando me refiro aos algoritmos de rasterização, a linguagem de programação utilizada será a linguagem C.

O usuário comum não tem acesso à memória de vídeo de um computador, sendo assim, será utilizado um Framework que simula o acesso à essa memória para que possa ser feito a rasterização sem o uso de código openGL.

**Conceitos prévios**

**** Para que não haja confusão, é necessário saber antes de tudo, que as coordenadas de um monitor não são as mesmas que as do plano cartesiano.

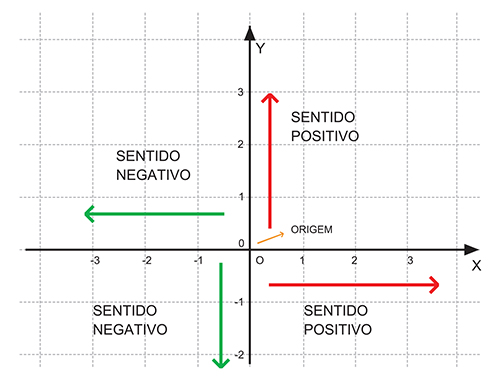
No plano cartesiano as coordenadas

começam no (0,0) e crescem para

cima e para direita, decrescem para

esquerda e para baixo.

Em coordenadas de um monitor é um pouco diferente, ainda temos as coordenadas (x,y), mas é como se olhássemos para o plano cartesiano e ele tivesse girado 90° para a direita.



De forma que as coordenadas

em um monitor começam no

ponto (0,0) que se encontram

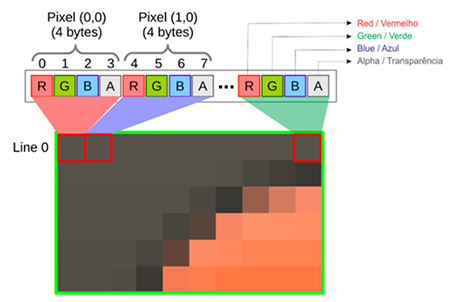
no canto superior esquerdo e

crescem para baixo e para

direita.

Outra coisa que deve ser ter em mente, é que para se obter cores diferenciadas nós usamos três cores distintas, que são vermelho, verde e azul (RGB - em inglês). Em openGL estas cores variam entre 0 e 1, porém como iremos simular o acesso a memória de vídeo, no nosso Framework as cores variam entre 0 e 255 (o que corresponde a sua intensidade).

**Pixel -** Um pixel é um ponto luminoso na tela e é composto por três canais de cores - Vermelho, Verde e Azul, e mais um canal para tratar a transparência destas cores, chamamos este canal de alpha, compondo um sistema chamado RGBA.



Os píxeis (sim, a escrita está correta) são representados por uma estrutura, de mesmo modo as cores.

typedef struct Point{

int x; // coordenada x do pixel

int y; // coordenada y do pixel

struct Color color; // o pixel possui uma cor

} Point;

typedef struct Color{

unsigned int r=255; // intensidade da cor vermelha

unsigned int g=255; // intensidade da cor verde

unsigned int b=255; // intensidade da cor azul

unsigned int a=255; // intensidade da transparência

} Color;

Para encontrar um ponto específico na tela precisamos de suas coordenadas. Você poderia pensar "óbvio, coordenada (x,y) diz onde está o ponto", porém não. Como já sabemos que vamos "acessar" a memória de vídeo para escrever os píxeis, vimos que o pixel possui 4 bytes de memória, então não se pode pensar que ao escrever na posição inicial da memória e logo após escrever na posição inicial mais um, dois pontos na tela estarão sendo escritos.

A posição do ponto é calculada da seguinte forma:

x \* 4 + y \* largura da imagem \* 4

A função offset faz este calculo:

unsigned int offset(Point point){

return point.x \* 4 + point.y \* IMAGE\_WIDTH \* 4;

}

**Rasterizando Pontos**

FBptr é o ponteiro que aponta para o Framebuffer, que é a simulação da memória de vídeo.

unsigned char \* FBptr;

void putPixel(Point point){

FBptr[ offset(point) + 0 ] = point.color.r;

FBptr[ offset(point) + 1 ] = point.color.g;

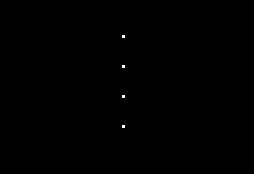
FBptr[ offset(point) + 2 ] = point.color.b;

FBptr[ offset(point) + 3 ] = point.color.a;

}

Um exemplo de 4 pontos desenhados na tela

Obs: foi usado a lupa do Windows para poder ver os pontos mais de perto e depois tirei print e cortei a imagem.



COORDENADAS DOS PONTOS

P1( 256, 240)

P2( 256, 250)

P3( 256, 260)

P4( 256, 270)

**Rasterização de Linha**

Sabemos que para trançar uma reta é necessário ao menos dois pontos. Usando a equação da reta conseguimos plotar todos os pontos desta reta( ou linha). Dados dois pontos, o próximos pontos serão obtidos apenas acrescentando um ao valor de x.

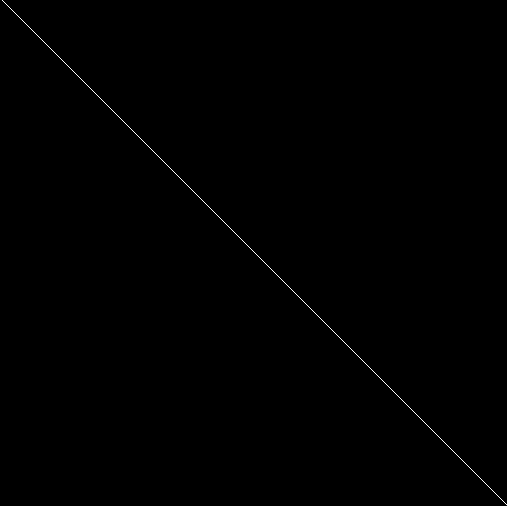
Equação da reta:

y = a \* x + b

a = (y1 - y0) / (x1 - x0)

b = y1 - a \* x1

EXEMPLO:



COORDENADAS DOS PONTOS INICIAIS

P1(0,0)

P2(512,512)

**Rasterização de Linha Usando o Algoritmo de Bresenham**

O algoritmo de Bresenham também é conhecido como algoritmo do Ponto Médio, é utilizado para traçar retas de forma incremental, trabalhando apenas com números inteiros. A ideia do algoritmo é bastante simples, ele utiliza a equação implícita da reta como uma função de decisão, para identificar qual o próximo pixel a ser ativado. Esta função é utilizada de forma incremental em cada pixel.

Desenvolvimento da **equação implícita da reta:**

y = a\*x + b //Partimos da Equação da Reta

y = (dy/dx)\* x + b

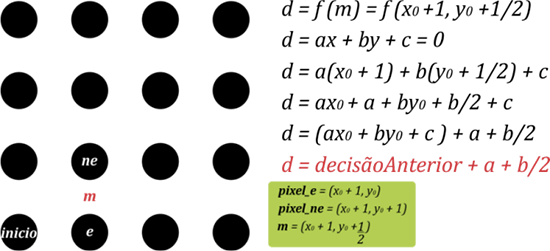
y \* dx = dy \* x + b \* dx //Multiplicando a equação por dx

dy \* x + (-y \* dx) + b \* dx = 0 //Igualando a equação a 0

decisão = αx + ßy + c , onde α = dy, ß = -dx, c = b \* dx

Se aplicarmos um ponto na equação implícita e obtivermos zero como resultado, significa que o ponto está na reta; se obtivermos valores negativos, o ponto está abaixo da reta; se obtivermos valores positivos, o ponto está acima da reta;

Seja m = (x0 + 1, y0 + 1/2) o ponto médio entre os pixels (x0 + 1, y0 + 1) e (x0 + 1, y0), iremos utilizar a função de decisão para avaliar qual pixel acender.



Note que a reta parte do ponto (x0,y0) logo, não existe decisão tomada anteriormente, podemos identificar o nosso valor de decisão aplicando f(x0 + 1, y0 + 1/2) - f (x0, y0).

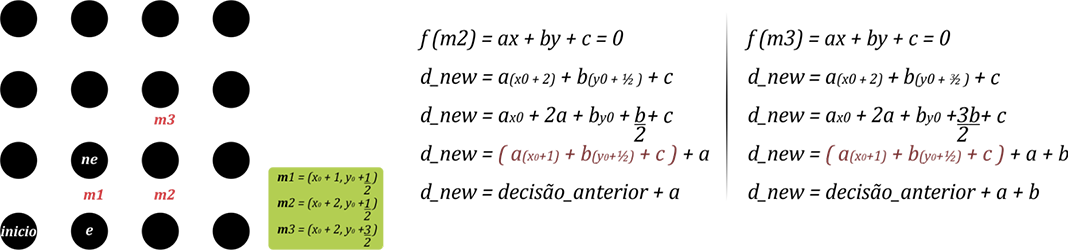
f (m) - f (inicio) = ax0 + by0 + c + a + b/2 - (ax0 + by0 + c )

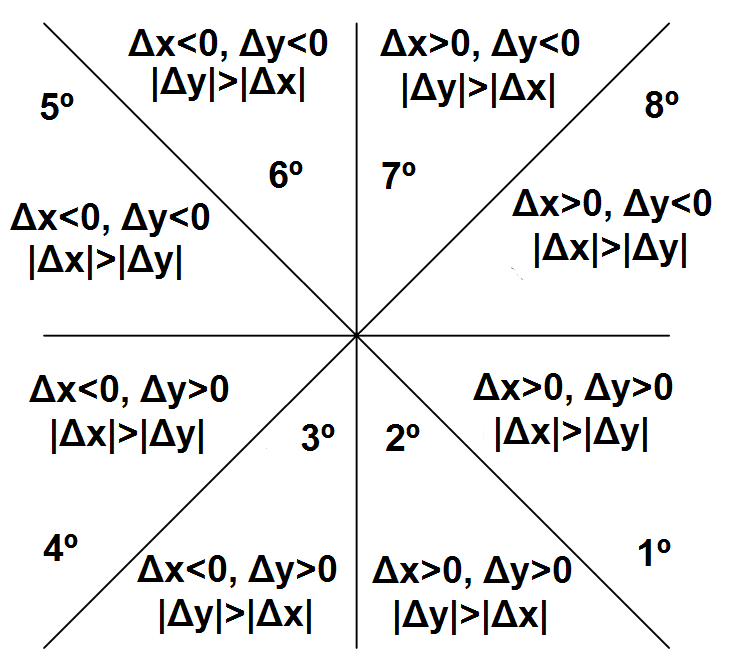
f (m) - f (inicio) = a + b/2

Logo, nosso valor de decisão inicial é:

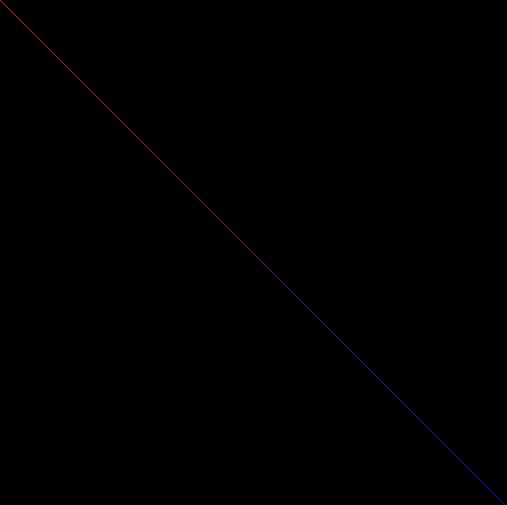
d = a + b/2

Após identificar qual pixel ativar através da função de decisão, é necessário verificar qual será o nosso próximo ponto médio, (x0 + 2, y0 + 1/2) ou (x0 + 2, y0 + 3/2).



Importante: Esta versão do algoritmo de Bresenham funciona apenas para 0°<= ângulo <=1°, porém podemos obter retas com outros coeficientes angulares por reflexão, como demonstrado na imagem abaixo:

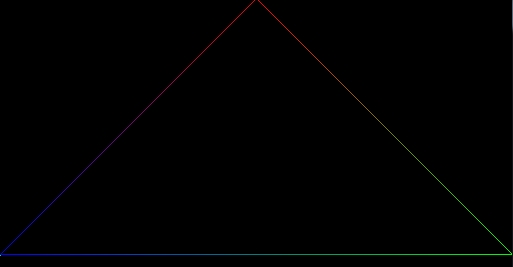
Exemplo abaixo da função DrawLine que utiliza o algoritmo de Bresenham com interpolação de cores:



P1(0,0) Cor(255,0,0,0)

P2(512,512) Cor(0,0,255,0)

Exemplo abaixo da função DrawTriangle que utiliza o algoritmo de Bresenham com interpolação de cores:



P1(256, 0 ) Cor(255,0,0,0)

P2(512,256) Cor(0,255,0,0)

P3(256, 0 ) Cor(0,0,255,0)

**Interpolação de cores**

A interpolação trata-se da mudança de cor em um seguimento de reta, que parte da cor do ponto inicial até a cor do ponto final.

A interpolação pode ser calculada através da seguinte equação:

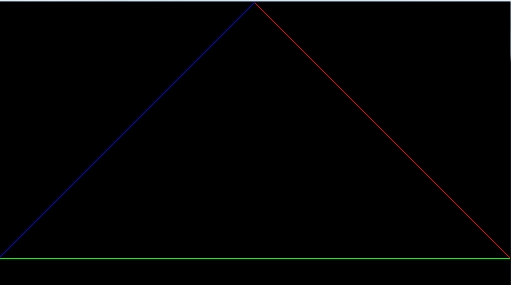
equacao.jpg

onde u = (ponto atual) / (quantidade de pontos - 1)

DIFICULDADES NO PROCESSO:

- A primeira tentativa de fazer interpolação não deu certo e as cores ao longo da reta não se alteravam, esse problema ocorria pois a função drawLine apenas desenha linhas entre uma coordenada e outra e coloca cada ponto na tela, dessa forma não interessava qual era a cor da segunda coordenada e a função de interpolação não funcionava, isso foi corrigido criando um array de pontos ( arrayPoint[512] ) e antes de chamar a função de interpolação determinei que a cor do último ponto a ser desenhado deveria ser igual a cor do ponto final, o que pode parecer óbvio, mas o computador não entende como nós, é preciso ensinar a ele o que fazer.

Imagem de como era a primeira versão da função de interpolação de cor abaixo:



**Informações sobre os arquivos:**

Os arquivos que compõem o Frameworksão:

• definitions.h

• main.cpp

• main.h

• Makefile

• mygl.h

O arquivo *definitions.h* contém a declaração das constantes que determinam a dimensão da tela(resolução) e o ponteiro (*FBptr*) para o início da memória de vídeo (mais especificamente, o início da região do *framebuffer*).

A primeira posição de memória, apontada por *FBptr*, corresponde ao pixel da posição (0,0), canto superior esquerdo da tela. Cada pixel possui 4 componentes de cor (RGBA), cada uma representada por 1 byte (*unsigned char*).

O arquivo *main.cpp* contém o programa principal e a declaração da função *MyGlDraw(),* de onde as funções de rasterização criadas foram chamadas.

O arquivo *main.h* contém a definição de algumas funções de auxílio na simulação do acesso à memória de vídeo e ao seu *scan*.

O arquivo *Makefile* que acompanha o *framework* é um script para compilação do sistema no ambiente Unix.

Importante: Os únicos arquivos modificados por mim foram o arquivos mygl.h e main.cpp.

Referências:

http://www.univasf.edu.br/~jorge.cavalcanti/comput\_graf04\_prim\_graficas2.pdf

https://www.codeproject.com/Articles/82091/OpenGL-Color-Interpolation

Material de aula do Professor Christian