Centro de Informática – João Pessoa
Disciplina: Computação Gráfica
Klismann de Oliveira Barros - 20200085284
Thaís Gabrielly Marques - 20180135293

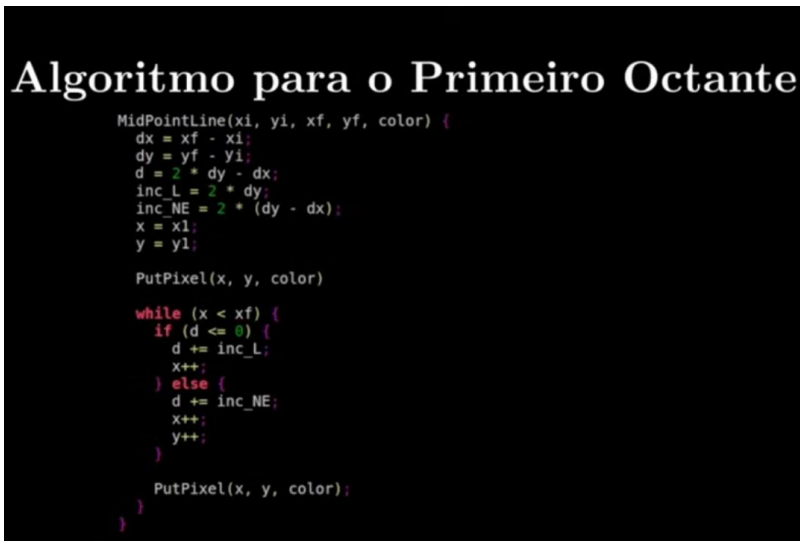
ATIVIDADE PRÁTICA 1 - RASTERIZAÇÃO DE LINHAS

Descrição:

O algoritmo do Ponto Médio, também conhecido por algoritmo de Bresenham, em homenagem a Jack Elton Bresenham. É um algoritmo criado para o desenho de linhas, em dispositivos matriciais (como por exemplo, um monitor), que permite determinar quais os pontos numa matriz de base quadriculada que devem ser destacados para atender o grau de inclinação de um ângulo. Foi pedido que nós desenvolvêssemos duas funções para rasterizar na tela, sendo que a primeira MidPointLineAlgorithm implementa o algoritmo de ponto médio e a segunda DrawTriangle que desenha um triângulo na tela após chamar a primeira função (MidPointLineAlgorithm), por fim teríamos que interpolar as cores ao longo da linha. Parece uma tarefa trivial, mas através de pesquisas descobrimos que, a versão básica fornecida em aula não funcionava para todos os octantes, seria preciso fornecer uma lógica de condições extras para chegar ao resultado final.

Estratégias adotadas:

Primeiro estudamos e implementamos o algoritmo fornecido na aula pelo professor:



Fonte: Aula 06 - Rasterização de Linhas: Algoritmo do Ponto Médio

Chegando a esses resultados:

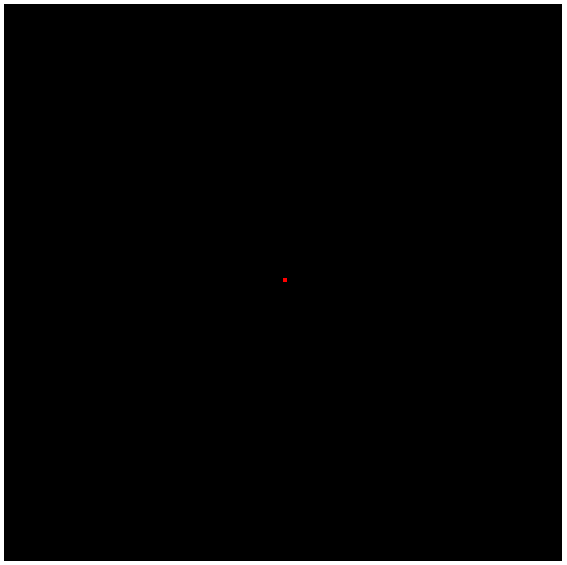


Imagem 2- Rasterizando um pixel

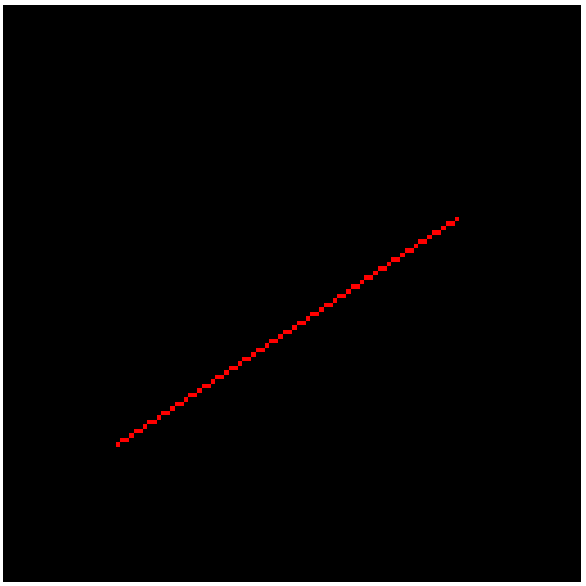


Imagem 3 - Rasterizando uma linha

Chegamos em um problema, como desenhar nos outros octantes? No algoritmo dado pelo professor tínhamos o primeiro e o quinto octante. Embora o algoritmo dado seja simples, a implementação para todos os octantes é mais complicada logicamente. Precisa de uma série de condições extras, para obedecer o seguinte:

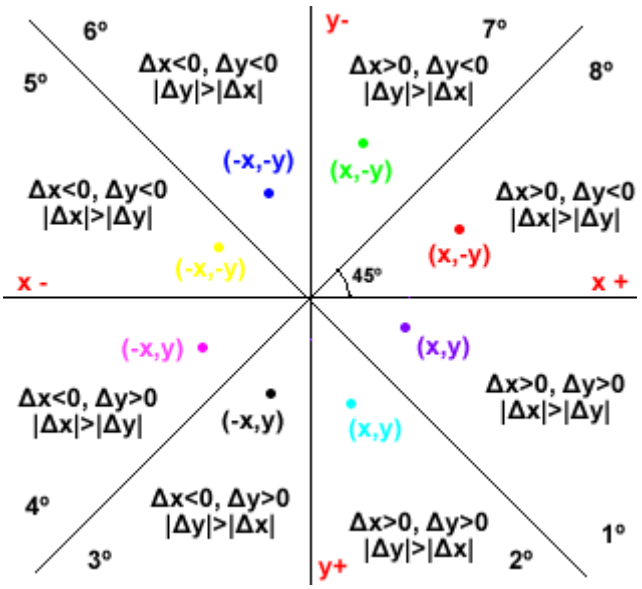
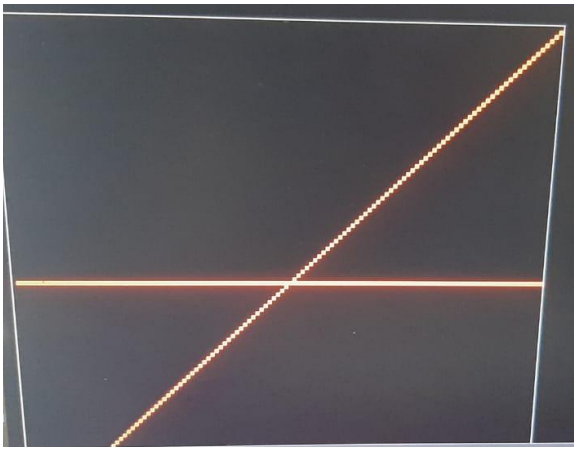
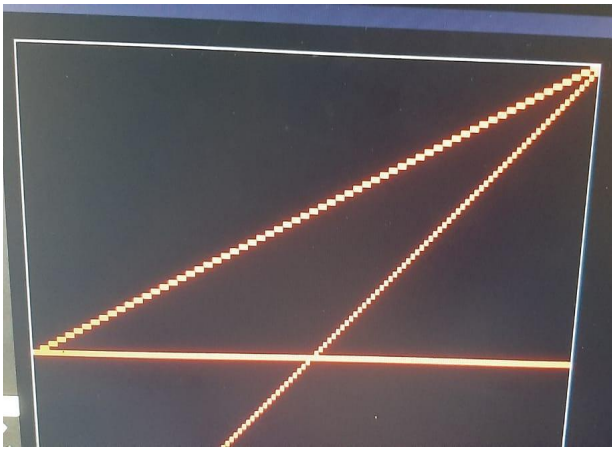


Imagem 4 - Observando dx e dy nos octantes
Fonte: http://henriquepontes.blogspot.com/

O resto dos casos são espelhados dos primeiros octantes já implementados. Isso significa que tudo o que precisa fazer é trocar algumas variáveis ou virar o sinal aqui e lá. De forma resumida, pegamos dois deltas e calculamos a inclinação da linha com base nos dois pontos finais.

Algumas falhas durante o processo:





Imagens 5 e 6 - Falha nos cálculos, bug nos octantes e rotação.

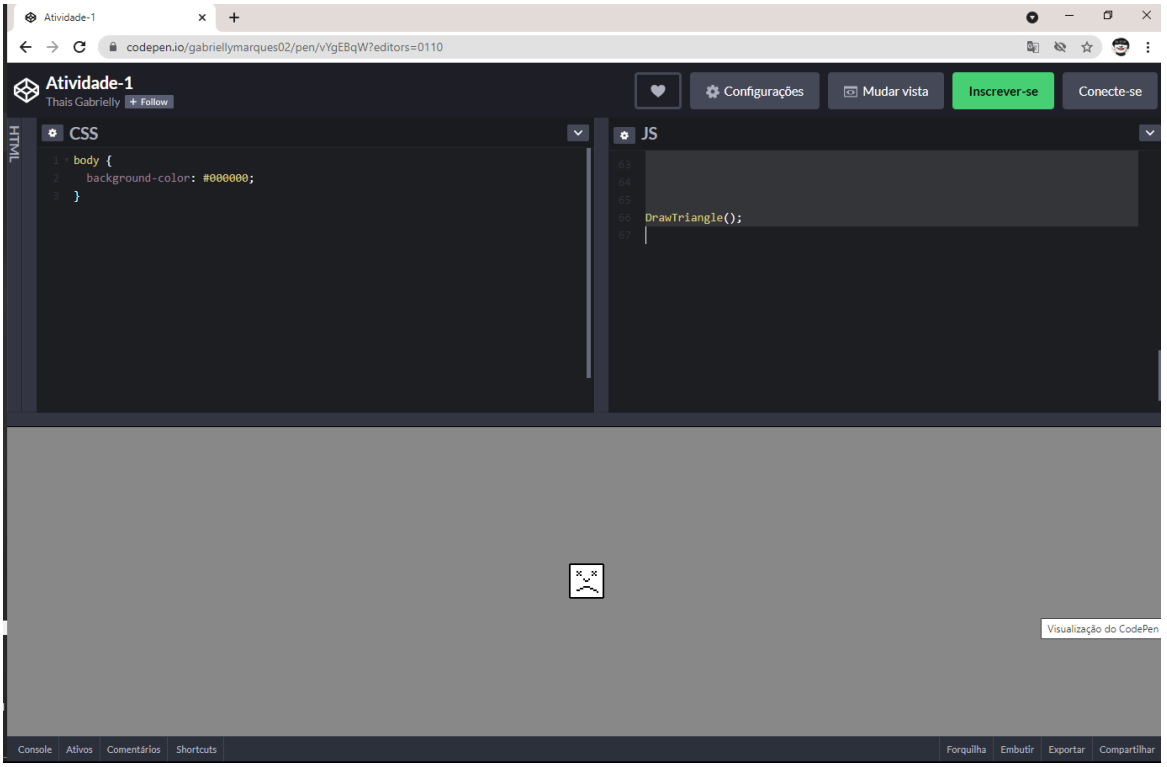


Imagem 7 - bug produzido por um else if digitado sem querer durante um teste de condição de um octante, quase perdemos o código, pois o link não abria nem no celular.

As medidas adotadas para implementar (sem a interpolação) foram:

```
function MidPointLineAlgorithm(x0, y0, x1, y1, color_0,color_1) {
    // Escreva seu código aqui!
        var x, y, dx1, dy1, dx0, dy0, ix, iy, x_aux, y_aux, i;

        //Calculando o delta das linhas
        dx1 = x1 - x0;
        dy1 = y1 - y0;

       //Ajustando para sempre inteiro positivo.
        dx0 = Math.abs(dx1);
        dy0 = Math.abs(dy1);

        //Calculando intervalos para ambos os eixos.
        ix = 2 * dy0 - dx0;
        iy = 2 * dx0 - dy0;


  // Se a linha for  do domínio de X, passar pelas próximas condições,
```

```
        if (dy0 <= dx0) {  // para saber por onde começar a desenhar.
            if (dx1 >= 0) { // trocando as coordenadas
                x = x0;
                y = y0;
                x_aux = x1;
            } else {
                x = x1;
                y = y1; x_aux = x0;
            }
            //desenhando o primeiro pixel
            color_buffer.putPixel(x, y, color_0);

            //rasterizando nos octantes
            for (i = 0; x < x_aux; i++) {
                x = x + 1;

                if (ix < 0) {
                    ix = ix + 2 * dy0;
                } else {
                    if ((dx1 < 0 && dy1 < 0) || (dx1 > 0 && dy1 > 0)) {
                        y = y + 1;
                    } else {
                        y = y - 1;
                    }
                    ix = ix + 2 * (dy0 - dx0);
                }

                color_buffer.putPixel(x, y, color_0);


            }

        } else { //aqui é o else do primeiro if, se a linha estiver
no domínio de Y,
            if (dy1 >= 0){//ocorre o mesmo processo agora tratando pelo
Y.
                x = x0;
                y = y0;
                y_aux = y1;
            } else {
                x = x1;
                y = y1;
                y_aux = y0;
```

```
            }

            color_buffer.putPixel(x, y, color_0, color_1);

            // Rasterizando
            for (i = 0; y < y_aux; i++) {
              y = y + 1;
              if (iy <= 0) {
                  iy = iy + 2 * dx0;
              } else {
                  if ((dx1 < 0 && dy1<0) || (dx1 > 0 && dy1 > 0)) {
                      x = x + 1;
                  } else {
                      x = x - 1;
                  }
                  iy = iy + 2 * (dx0 - dy0);
              }
               color_buffer.putPixel(x, y, color_0, color_1); // saída
de renderização.
            }
        }
  }
function  DrawTriangle(x0,  y0,  x1,  y1,  x2,  y2,  color_0,  color_1,
color_2) {
  // O que foi feito aqui? Apenas chamei a MidPointLineAlgorithm
  //e organizei as coordenadas para desenhar o triângulo.
  MidPointLineAlgorithm(x0, y0, x1, y1, color_0, color_1);
  MidPointLineAlgorithm(x1,y1, x2,y2, color_1, color_2);
  MidPointLineAlgorithm(x2,y2, x0,y0, color_2, color_0);

}
// desenhando um pixel
//MidPointLineAlgorithm(64,64,64,64,[255,0,0,255],[255,255,0,255]);

// linha do exemplo
//MidPointLineAlgorithm(25,30,100,80,[255,0,0,255],[255,255,0,255]);
```

Após os testes obtemos o seguinte resultado (Sem interpolação):

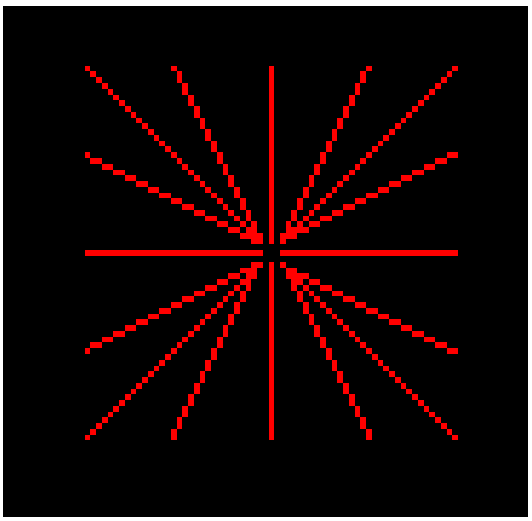


Imagem 8 - resultado final do **MidPointLineAlgorithm**

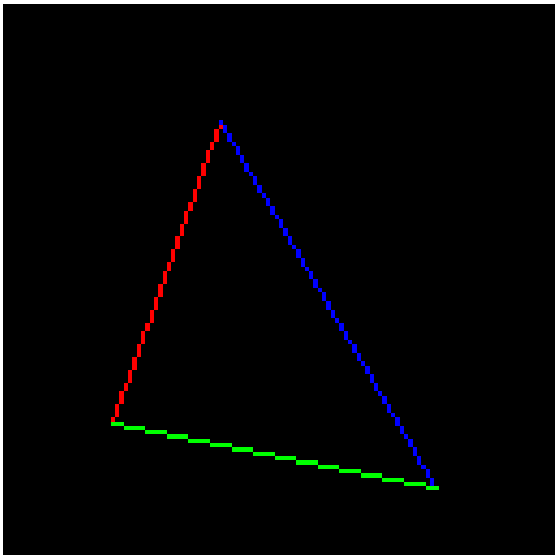


Imagem 9 - resultado final do **DrawTriangle**

Interpolação linear das cores:

A interpolação das cores funcionava com a divisão da cor do pixel respectivo, pela distância entre os pontos, por causa da incrementação, garantimos que o pixel terá uma cor diferente. A maior dificuldade encontrada foi por uma "seleção" que é feita pelo algoritmo na hora da pintura de uma das arestas do triângulo, porém totalmente funcional para todos os outros casos teste. Uma das maiores dificuldades foi entender que precisava-se direcionar qual seria a distribuição e como o algoritmo iria lidar com a pintura dos pixels.

Resultados obtidos:

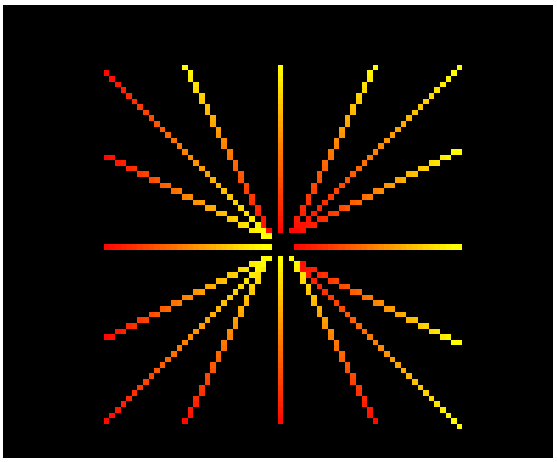


Imagem 10 - resultado final do **MidPointLineAlgorithm.**

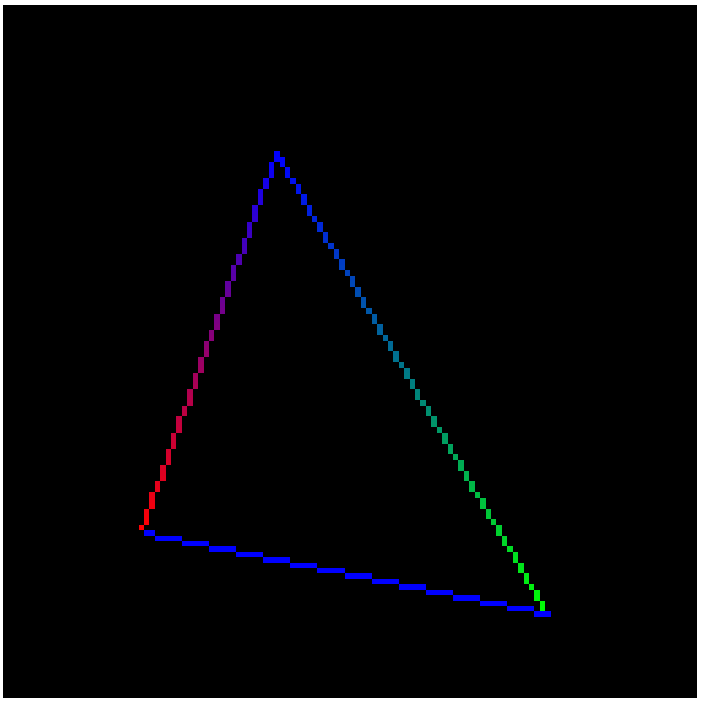


Imagem 11 - resultado final do **DrawTriangle.**

Dificuldades e possíveis melhorias:

A maior dificuldade na parte de implementação da função MidPointLineAlgorithm foi entender como generalizar para todos os octantes e aplicar a lógica de funcionamento. Precisou de muita pesquisa e rever bastante o material fornecido pelo professor. Uma possível melhoria seria criar um laço para ir desenhando as linhas em todos os octantes a partir de uma chamada da função MidPointLineAlgorithm (Parecido com esses exemplos: https://www.youtube.com/watch?v=WV1Jk9OKecA e https://youtu.be/Om6QIaTDcAs). Por

fim, uma possível melhoria para a interpolação seria colocar especificidades para pintura de pixels em cada octante de forma mais específica.

Referências bibliográficas:

- Vídeos aulas e notas do professor.
- https://www.tutorialspoint.com/computer_graphics/line_generation_algorithm.htm
- http://luanareiscg.blogspot.com/2016/03/rasterizacao-de-primitivas.html
- http://www.univasf.edu.br/~jorge.cavalcanti/comput_graf04_prim_graficas2.pdf
- https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
- http://www.roguebasin.com/index.php?title=Bresenham%27s_Line_Algorithm
- http://fleigfleig.blogspot.com/2016/08/interpolacao-de-cores-e-triangulos.html

Links dos repositórios online:

I) https://codepen.io/Klismann_Barros/pen/GRrZEBW?editors=0011 (Como interpolação linear).
II) https://codepen.io/gabriellymarques02/pen/vYgEBqW?editors=0010 (Sem interpolação).