Universidade Federal da Paraíba
Centro de Informática

Introdução à Computação Gráfica
Atividade Prática 1 - Rasterização de Linhas

Francisco Siqueira Carneiro da Cunha Neto[1]
20190029015
[1]francisconeto@eng.ci.ufpb.br

10 de setembro de 2021

ATIVIDADE DESENVOLVIDA

Nesta atividade foi implementado o algoritmo de rasterização de linhas de Bresenham [1] na linguagem de programação JavaScript, usando o elemento HTML Canvas e a classe disponibilizada em https://codepen.io/ICG-UFPB/pen/dyWgQoj. A implementação foi feita em uma função `MidPointLineAlgorithm()` que rasteriza uma linha entre duas coordenadas dadas e define as cores de cada pixel pela interpolação de duas cores dadas. Esta função será o foco principal deste relatório. Também foi implementada uma função `DrawTriangle()` que rasteriza as arestas de um triângulo.

O código fonte também pode ser encontrado em seus repositórios no GitHub e CodePen.

PRIMEIRO OCTANTE

O primeiro passo no desenvolvimento foi escrever um algoritmo capaz de rasterizar apenas linhas que se encontram no primeiro octante do plano cartesiano. Esse algoritmo foi baseado na Aula 06 [2], e pode ser consultado em Alg. 1.

Algorithm 1 Rasterização de linhas no primeiro octante

$\alpha \leftarrow y_1 - y_0$
$\beta \leftarrow -(x_1 - x_0)$
$d \leftarrow 2 \times \alpha + \beta$
$x \leftarrow x_0$
$y \leftarrow y_0$
while $x \neq x_1$ **do**
  putPixel(x, y)
  if $d \geq 0$ **then**
    $y \leftarrow y + 1$
    $d \leftarrow d + 2 \times (\alpha + \beta)$
  else
    $d \leftarrow d + 2 \times \alpha$
  end if
  $x \leftarrow x + 1$
end while

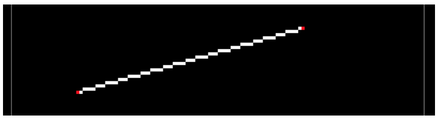

Figura 1: *Linha rasterizada no 1o octante do plano cartesiano.*

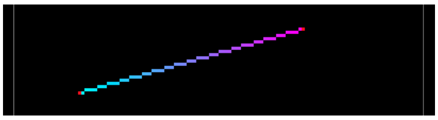

Figura 2: *Linha rasterizada no 1o octante com interpolação linear de cores.*

Nessa etapa também foi desenvolvida uma função `DebugLine()` que pinta dois pixels de vermelho, com o propósito de mostrar visualmente as coordenadas inicial e final desejadas para as linhas. Dessa forma é possível facilmente saber se a linha foi rasterizada corretamente.

Essa primeira versão do algoritmo rasterizou corretamente as linhas. A Fig. 1 mostra uma linha rasterizada por ele.

INTERPOLAÇÃO DE CORES

A definição de cores para cada pixel veio logo após, utilizando uma simples função de interpolação linear [3]. Imediatamente antes de preencher um pixel, o valor de vermelho, verde e azul da cor que ele deve assumir são calculados pela Eq. 1, onde c_0 e c_1 são os valores de vermelho, verde ou azul das cores informadas para os vértices da linha, x_0 e x_1 são as coordenadas X dos vértices da linha e x é a coordenada X do pixel a ser preenchido. Cada um desses valores é então multiplicado pelo valor normalizado do canal alfa para aquele pixel, calculado da mesma maneira.

$$c = c_0 + (x - x_0) \times (c_1 - c_0) \div (x_1 - x_0) \tag{1}$$

A Fig. 2 mostra uma linha rasterizada no primeiro octante com interpolação de cores, onde o vértice mais a esquerda possuí a cor, em RGBA, (0, 255, 255, 255), e o vértice mais a direita possuí a cor (255, 0, 255, 255). A Fig. 3 mostra a mesma

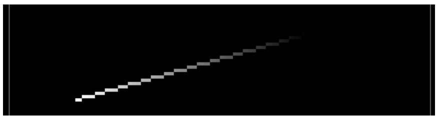

Figura 3: *Linha rasterizada no 10 octante com interpolação linear do canal alfa.*

linha com as cores (255, 255, 255, 255) e (255, 255, 255, 0) nos vértices da esquerda e da direita, respectivamente.

OUTROS OCTANTES

A próxima etapa foi a adaptação do algoritmo para que funcionasse em todos os octantes do plano cartesiano. Isso foi feito a partir da análise individual de cada octante, englobando comparações às retas do primeiro octante e análises matemáticas da equação da retas e afins.

3°, 4°, 5° e 6° Octantes

Primeiro, notou-se que linhas nos octantes do lado esquerdo do plano (isto é, os que possuem $x < 0$) podem ser tratadas como idênticas àquelas nos octantes do lado direito, apenas sendo necessário inverter os vértices inicial e final.

No algoritmo, foi adicionado uma condição: caso $x_1 < x_0$ seja verdadeiro, a inversão é realizada. Dessa forma, só era necessário analisar os 2°, 7° e 8° octantes.

2° Octante

Analisando as linhas do 2° octante, percebeu-se que seu comportamento é similar às do primeiro octante, apenas com as coordenadas X e Y invertidas: enquanto, nas linhas do 1° octante, cada pixel subsequente sempre incrementa a coordenada X e pode ou não incrementar a coordenada Y; nas linhas do 2° octante cada pixel subsequente incrementa a coordenada Y e pode ou não incrementar a coordenada X.

Para realizar essa inversão, adotou-se um novo par de coordenadas P e Q, onde P é a coordenada que sempre é incrementada e Q a coordenada que só é incrementada ocasionalmente. P substituiu as ocorrências de X no algoritmo, e Q as ocorrências de Y. Para linhas no 1° octante, $p_0 \leftarrow x_0$, $q_0 \leftarrow y_0$, $p_1 \leftarrow x_1$ e $q_1 \leftarrow y_1$; para linhas no 2° octante, $p_0 \leftarrow y_0$, $q_0 \leftarrow x_0$, $p_1 \leftarrow y_1$ e $q_1 \leftarrow x_1$. O cálculo de α e β também é afetado por essa substituição, corrigindo modificações na equação geral da reta.

Além disso, uma váriavel `tall` indica se o valor de P se trata de uma coordenada X ou Y. Antes de preencher um pixel, se é checado se `tall` é verdadeiro; caso seja, a função chamada é `putPixel(q, p)`, caso contrário, a função chamada é `putPixel(p, q)`.

7° e 8° Octantes

A análise das linhas dos 7° e 8° octantes revelou que elas são construídas da mesma maneira que no 2° e 1° octantes, respectivamente, exceto que a coordenada Y não é incrementada, e sim decrementada.

Contudo, como em geral não se sabe se a coordenada Y é representada por P ou Q, é preciso decrementar um ou outro de acordo com o octante em especifico. Para esse fim, foram criadas duas variáveis `p_mod` e `q_mod`, inicialmente guardando o valor 1. Para o 7° octante, `p_mod` $\leftarrow -1$; para o 8° octante, `q_mod` $\leftarrow -1$. Então, ao invés de incrementar p e q por 1, p é incrementado pelo valor de `p_mod` e q pelo valor de `q_mod`.

Para considerar as modificações na equação da reta, o valor de α passa a ser ele mesmo multiplicado por `q_mod` e o valor de β passa a ser ele mesmo multiplicado por `p_mod`.

Reconhecimento do Octante

O último passo é reconhecer a qual octante a linha pertence. Essa operação é realizada após a inversão das coordenadas, portanto só é necessário diferenciar entre os 1°, 2°, 7° e 8° octantes.

O reconhecimento é feito a partir de duas comparações: entre δx e δy e entre y_1 e y_0. Se $\delta x > \delta y$, a linha está ou no 1° ou no 8° octante; dentre estes, se $y_1 > y_0$, a linha está no 1° octante e caso contrário, está no 8°. Caso $\delta x < \delta y$, a linha está ou no 2° ou no 7° octante; dentre estes, se $y_1 > y_0$, a linha está no 2° octante e caso contrário, está no 7°.

Resultados

Após alguns erros de implementação que distorceram os pontos iniciais e finais das linhas, o algoritmo foi capaz de rasterizar corretamente linhas em todos os octantes, como mostra a Fig. 4.

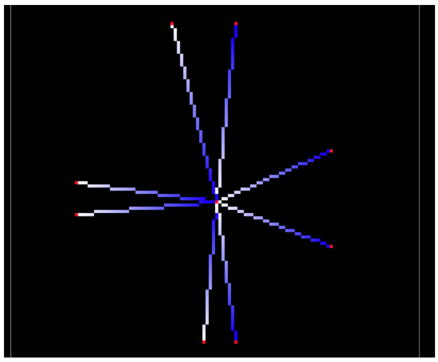

Figura 4: *Linhas rasterizadas em todos os octantes do plano cartesiano.*

Contudo, a interpolação das cores parecia inverter os vértices iniciais e finais nos octantes do lado esquerdo: como as coordenadas destes octantes são invertidas, as cores se invertiam junto. A solução foi (re)inverter as cores no momento em que se inverte as coordenadas dos vértices, levando-as para sua posição correta. A Fig. 5 mostra a imagem gerada após essa mudança.

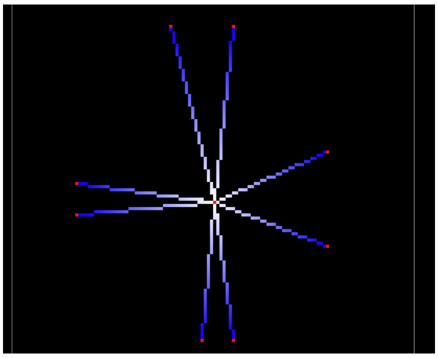

Figura 5: *Linhas rasterizadas com interpolação de cores corrigida.*

Ao fim, foi notado que, devido a estrutura do algoritmo, o pixel final acabava não sendo preenchido. Devido à função `DebugLine()`, que coloria de vermelho os pixels final e inicial, isso não foi notado até o final da implementação. A solução, porém, foi simples: basta preencher o pixel nas coordenadas finais após sair do *while loop*.

TRIÂNGULO

Após a rasterização de linhas e interpolação de cores correta em todos os octantes, a implementação da função `DrawTriangle()` foi trivial. O Alg. 2 descreve esta implementação.

Algorithm 2 Rasterização das arestas de um triângulo

function DRAWTRIANGLE(x_0, y_0, c_0, x_1, y_1, c_1, x_2, y_2, c_2)
  MidPointLineAlgorithm(x_0, y_0, c_0, x_1, y_1, c_1)
  MidPointLineAlgorithm(x_1, y_1, c_1, x_2, y_2, c_2)
  MidPointLineAlgorithm(x_2, y_2, c_2, x_0, y_0, c_0)
end function

A Fig. 6 mostra o resultado da chamada desta função.

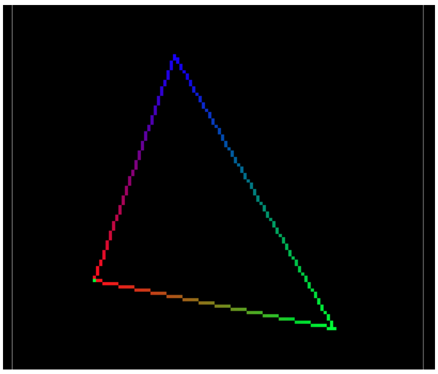

Figura 6: *Rasterização das arestas de um triângulo.*

IMAGEM LIVRE

Utilizando somente chamadas sucessivas das funções `MidPointLineAlgorithm()` e `DrawTriangle()`, houve uma tentativa de desenhar a personagem Totoro, do filme *Meu Amigo Totoro (1988)*. A Fig. 7 mostra a personagem em uma cena do filme, que serviu como referência para a imagem criada, por sua vez exposta na Fig. 8.

Figura 7: *Cena do filme Meu Amigo Totoro (1988).*

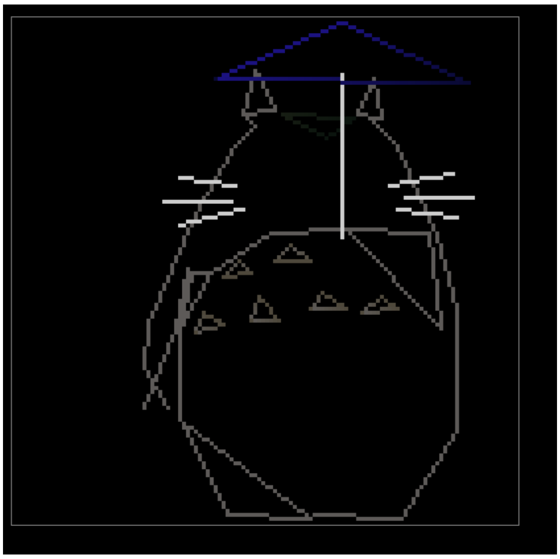

Figura 8: *Imagem da personagem Totoro criada com as funções implementadas.*

REFERÊNCIAS

[1] W. contributors. (2021). “Bresenham’s line algorithm — Wikipedia, The Free Encyclopedia,” endereço: https://en.wikipedia.org/w/index.php?title=Bresenham%5C%27s_line_algorithm&oldid=1026473544 (acesso em 10/09/2021).

[2] C. A. Pagot. (2021). “Aula 06 - Rasterização de Linhas: Algoritmo do Ponto Médio,” endereço: https://www.youtube.com/watch?v=tygja6rr62M (acesso em 03/09/2021).

[3] Wikipedia contributors. (2021). “Linear interpolation — Wikipedia, The Free Encyclopedia,” endereço: https://en.wikipedia.org/w/index.php?title=Linear_interpolation&oldid=1030419099 (acesso em 03/09/2021).