Roteiro Revisão Prova I de CG

Rafael Amauri Diniz Augusto - 651047

1 - A transformação de translação tem o problema de não conseguir ser representada como uma multiplicação de matrizes sem coordenadas homogêneas usando uma matriz 2x2. A operação ainda pode ser feita, mas vai fazer com que a matriz de transformação seja dependente dos pontos que estão sendo movidos. A vantagens de usar coordenadas homogêneas está no fato de se conseguir fazer com que a matriz de operação se torne independente dos pontos movidos, fazendo com que ela não tenha que ser recalculada várias vezes antes de ser utilizada e que possa só ser repetida várias vezes.

2 - A rotação em 180 graus é equivalente à reflexão.

Rotação 180 =

$$\begin{bmatrix} \cos 180 & -\text{sen } 180 & 0 \\ \text{sen } 180 & \cos 180 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Traduzindo de graus para radianos, temos que sen pi = 0 e que cos pi = -1.
Isso nos dá a seguinte matriz:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Que é igual à matriz da reflexão.

3 - Movendo a figura para a origem antes da operação. Se fizer a translação de um dos pontos da figura e fixar ele em (0,0), é necessário apenas se lembrar de quantos pixels a figura andou nos eixos X e Y, aplicar a transformação, e depois mover a figura a mesma quantidade de pixels nos respectivos eixos.

4 - Pois a ordem das transformações afeta o resultado final. Fazer uma escala antes de uma rotação é diferente de fazer uma rotação e depois uma escala. Isso nos mostra que a operação de multiplicação de matrizes não é comutativa (significando que a ordem importa). Ela precisa ser inversa por causa da ordem que as contas são feitas, com o vetor de pontos. Por exemplo: se a gente fizer uma rotação e depois uma escala, vamos ter a seguinte fórmula:

Rotação = [R]
Escala = [S]
Ponto = [P]

O que queremos é fazer a rotação e depois a escala. Para otimizar essas operações, podemos combinar R e S em uma terceira matriz M, que vai fazer a operação para a gente.

Mas, se definirmos M como R * S, quando formos fazer M * P, o que acontece é que P vai ser multiplicado não por R e depois S, mas por uma matriz M que é um agregado de R e S, e na multiplicação o resultado final acabaria sendo R * (S * P), o inverso de o que queremos. Não é muito intuitivo pois estamos acostumados com a multiplicação ser uma operação comutativa, mas aqui não é assim que funciona. O ponto principal aqui é que a ordem das operações afeta o resultado final, e se não invertermos, primeiro P vai ser escalado e depois rotacionado.

5 -

A = (-1, -3)
B = (-2, 8)
C = (9,2)

a - T(-1,5)

[[1 0 -1], [0 1 5], [0 0 1]] x [-1 -3 1] =

6 - Porque o maior valor de delta indica a coordenada na qual a figura mais vai andar. Se por exemplo, a figura deve andar 5 pixels para a direita e 2 para cima, o delta X vai ser muito maior que o delta Y, e como delta X dita a maior quantidade de movimentos a serem feitos, a movimentação dos pontos será feita de 1 em 1 ao longo do eixo X, com os pontos em Y sendo calculados como os pontos que devem acompanhar essa linha. Os pontos em Y nunca vão aumentar de 1 em 1 nesse exemplo e serão entendidos como pontos flutuantes, que depois serão convertidos.

7 - Qual o maior delta. No primeiro caso, como o delta X é maior, os pontos da reta progridem no eixo X de 1 em 1. No segundo caso, como o delta Y é maior, os pontos progridem no eixo Y de 1 em 1.

8 - Porque o DDA trabalha com pontos flutuantes, e uma tela não aceita pontos flutuantes como coordenadas. A minha tela por exemplo tem resolução 3440x1440, sendo 3440 pixels no eixo X e 1440 pixels no eixo Y. Todos esses pixels são individuais, atomizados e representados por valores inteiros. Essa configuração não permite um ponto (2, 1.5) como valor para colorir, pois o ponto 1.5 não existe. Como esse 1.5 é um tipo de valor que o DDA pode retornar, é preciso converter para inteiro, e só assim ele pode ser colorido.

9 - Como vai ser calculado o próximo X e o próximo Y. No primeiro caso, o próximo X é igual ao X anterior + 1, e Y vai ser igual a (deltaY / número de passos). No segundo caso, como delta Y é maior, a situação se inverte e o próximo X vai ser igual a (deltaX / número de passos), e o próximo Y vai ser igual ao Y anterior + 1.

10 - A: [-1, 4] [0, 5] [1, 5] [2, 6] [3, 6] [4, 7] [5, 7]
B: [5, 7] [4, 7] [3, 6] [2, 6] [1, 5] [0, 5] [-1, 4]

C: [-1, 4] [0, 5] [1, 6] [2, 7] [3, 8]
D: [2, 0] [3, 0] [4, 0] [5, 0] [6, 0]
E: [1, 3] [1, 4] [1, 5] [1, 6]

11 - O algoritmo de Bresenham não trabalha com pontos flutuantes e se mantém usando apenas inteiros para as operações. Isso dá uma vantagem enorme para ele, pois a operação em pontos flutuantes normalmente vão usar Float32 para armazenar valores. Com a adição da representação da mantissa e da precisão do tipo float, isso faz não só esse ser um tipo de dado mais delicado de armazenar para uma CPU, temos a adição do problema de ser mais custoso realizar operações nesse tipo de dado. Ao eliminar completamente o uso de floats para realizar as contas e usar apenas inteiros, o algoritmo fica mais rápido e mais eficiente.

12 - Pois o cálculo do p no algoritmo de Bresenham depende de uma subtração deltaY - deltaX (1 caso, no 2 caso é deltaX - deltaY), e se os valores não forem os valores absolutos, pode ser que o valor de P fique positivo antes da primeira iteração, resultando em valores calculados errados.

13 - No primeiro caso em X, ela pode ser feita antes sem alterar o funcionamento do algoritmo. No primeiro caso em Y isso já não pode acontecer, pois a atualização do valor da variável Y depende do valor da variável de decisão P, logo essa atualização depende inerentemente do valor de P, e a ordem não pode ser invertida.

14 - Enquanto P for negativo, apenas X é atualizado. Mas, quando P é negativo, tanto X quanto Y são atualizados com seus respectivos incrementos.

15 - A: [-1, 4] [0, 5] [1, 5] [2, 6] [3, 6] [4, 7] [5, 7]
B: [5, 7] [4, 6] [3, 6] [2, 5] [1, 5] [0, 4] [-1, 4]
C: [-1, 4] [0, 5] [1, 6] [2, 7] [3, 8]
D: [2, 0] [3, 0] [4, 0] [5, 0] [6, 0]
E: [1, 3] [1, 4] [1, 5] [1, 6]

16 - Tecnicamente todos são calculados, mas pela propriedade “espelho” de um círculo, na realidade só um octante precisa ser calculado. Todos os outros octantes são reflexões no eixo X, Y ou ambos.

17 - Fazer depois

18 - O X precisa ser atualizado antes de P e Y porque o cálculo de P precisa do valor que está guardado em X. O valor de Y só é atualizado se P for maior que zero, então apenas X pode ser atualizado antes das outras.

19 - Nos outros pontos que não estão na origem. O algoritmo não precisa saber se os pontos estão ou não estão na origem, pois na hora que a função for chamada serão usadas as posições de X e Y que não estão na origem.
20 - Fazer depois

21 - No recorte de polígonos sim, pois o recorte precisa ser em relação ao sentido da normal. No Liang-Barsky e no Cohen-Sutherland, não altera o resultado final. Um jeito fácil de pensar é pensar que temos dois recortes em caixa: um maior e um muito menor. Independentemente da ordem que eles forem aplicados, o recorte menor é quem vai ditar o que será renderizado, independentemente se ele vier antes ou depois na pipeline.

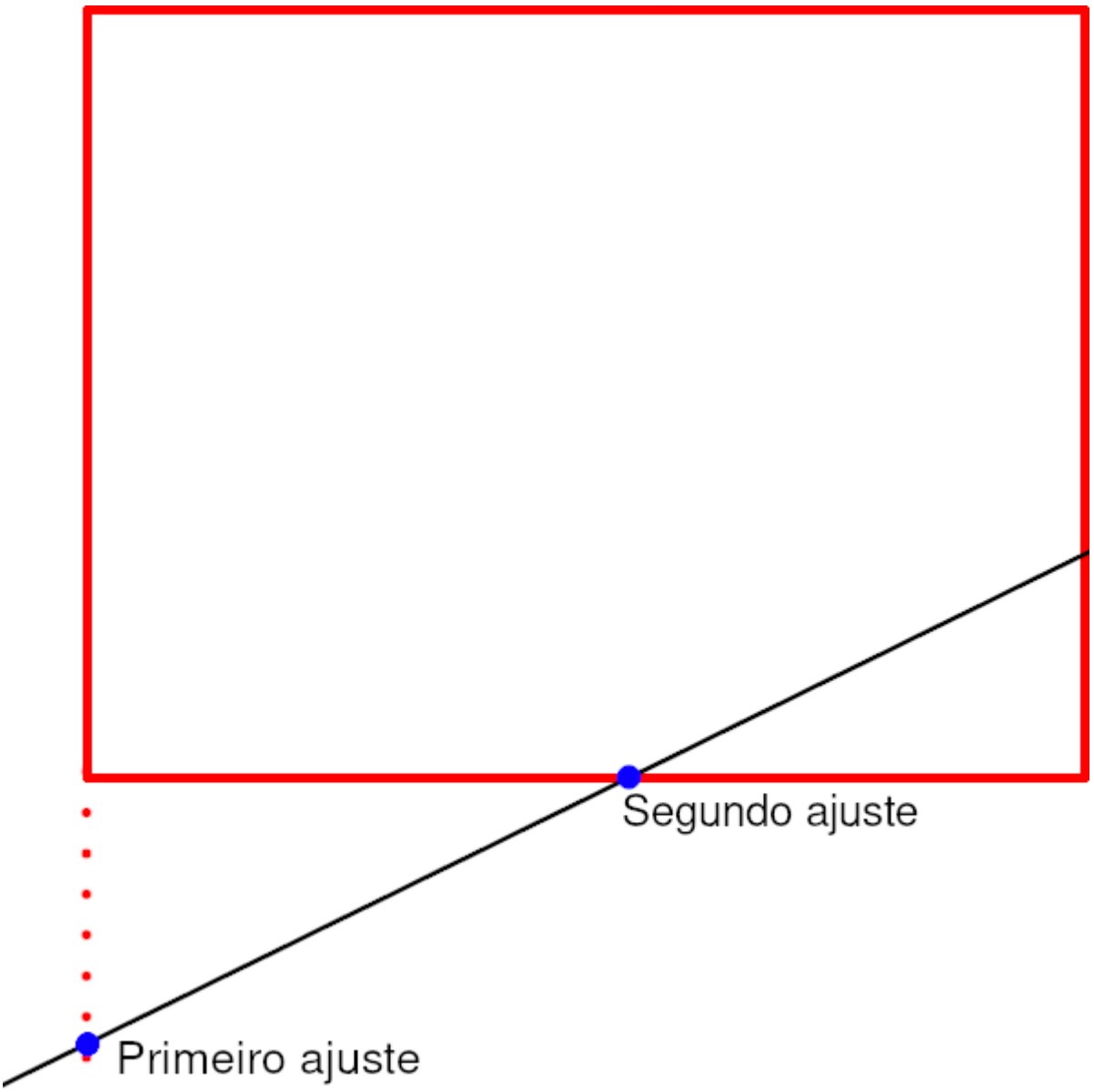
O que é influenciado de fato é a performance.

22 - Porque é impossível existir um código 3 e um código 7. Os código 3 e 7 indicam que um ponto estaria tanto à esquerda quanto à direita da área de recorte ao mesmo tempo, o que é geometricamente impossível.

23 - Se os dois códigos eventualmente virarem zero, indicando que já estavam dentro da área de recorte ou eles já serem zero na primeira iteração, indicando a mesma coisa. Alternativamente, outra condição de parada é a linha pode já estar fora da área de recorte e nem passar por ela, indicando que ela não precisa ser recortada.

24 - Dependendo de qual for a posição do ponto inicial e como for o recorte, ele pode ser reajustado mais de uma vez, principalmente se o primeiro ajuste levar para uma região de fronteira em apenas um eixo, mas que ainda é diferente da fronteira do recorte.

Exemplo:



25 - Porque nenhum dos dois pontos está dentro da janela e estão na mesma lateral. Se eles estão fora da área de visualização e em um mesmo canto, não precisam ser renderizados nunca.

26 - Eles são mantidos para que a transformação não seja destrutiva. Caso eles sejam substituídos, se nós precisarmos fazer outro recorte eventualmente, o recorte sairá errado. Por causa disso eles não podem ser substituídos.

27 -
x min = -2
x max = 5
y min = 1
y max = 6

Primeira linha = (-1, -3) -> (-2, 8)
C1 = 4
C2 = 8
Fora para fora! A linha será ignorada.

Segunda linha = (-2, 8) -> (9, 2)
C1 = 8
C2 = 2
Fora para fora! A linha será ignorada.

Terceira linha = (-1, -3) -> (9, 2)
C1 = 4
C2 = 8
Fora para fora! A linha também será ignorada!

28 - Para economizar processamento. O Liang-Barsky só vai atualizar os pontos da reta se ela estiver parcialmente dentro da janela. Caso ela esteja totalmente fora ou totalmente dentro, não é preciso ajustar nada

29 - Pois o primeiro cliptest garante que a reta está fora da área de recorte. Se ela já está fora, nada será calculado, e é possível otimizar o código ao aninhar eles.

30 - Pois o algoritmo Liang-Barsky entende a linha como uma representação “normalizada” entre 0 e 1, e é por isso que esses valores foram definidos assim. À medida que as interseções de linha forem calculadas e a linha for melhor entendida pelo algoritmo, esses pontos vão sendo atualizados. Outro motivo é a otimização, pois usar 0 e 1 torna os cálculos mais diretos e reduz a complexidade das contas envolvidas.

31 - Primeira linha = (-1, -3) -> (-2, 8)
deltaY = 11

Linha vai ser rejeitada!

Segunda linha = (-2, 8) -> (9, 2)
deltaY = -6; t2 = 0.72; t1 = 0.33

Linha que vai ser aceita:
x1 = 1
x2 = 6
y1 = 6
y2 = 3

Terceira linha = (-1, -3) -> (9, 2)
deltaY = 5; t2 = 0.7; t1 = 0.6

Linha que vai ser aceita:

x1 = 5
x2 = 6
y1 = 0
y2 = 0

32 - O problema do algoritmo Cohen-Sutherland e Liang-Barsky é que eles pegam polígonos fechados e após o recorte vão deixar esse polígono aberto dependendo do recorte. O algoritmo Sutherland-Hodgeman garante um polígono fechado nesse caso. Logo, ele é um algoritmo cujo objetivo é garantir um polígono fechado a cada recorte e lista de vértices que o definem, ordenada segundo a normal do plano do polígono.

33 - São 4 possibilidades:

Dentro para dentro - (V1 -> V2) -> Insere vértice posterior
Fora para dentro - (V1 -> V2) -> Insere interseção em V1 e insere V2
Dentro para fora - (V1 -> V2) -> Insere interseção em V2
Fora para fora - (V1 -> V2) -> Não salva nada

34 -
x min = -2
x max = 5
y min = 1
y max = 6

Primeira linha = (-1, -3) -> (-2, 8)
Fora para fora! A linha será ignorada.

Segunda linha = (-2, 8) -> (9, 2)
Fora para fora! A linha será ignorada.

Terceira linha = (-1, -3) -> (9, 2)
Fora para fora! A linha também será ignorada!

Como todos os vértices do triângulo estão completamente fora da janela de recorte, nada será feito nem adicionado à lista.

35 -

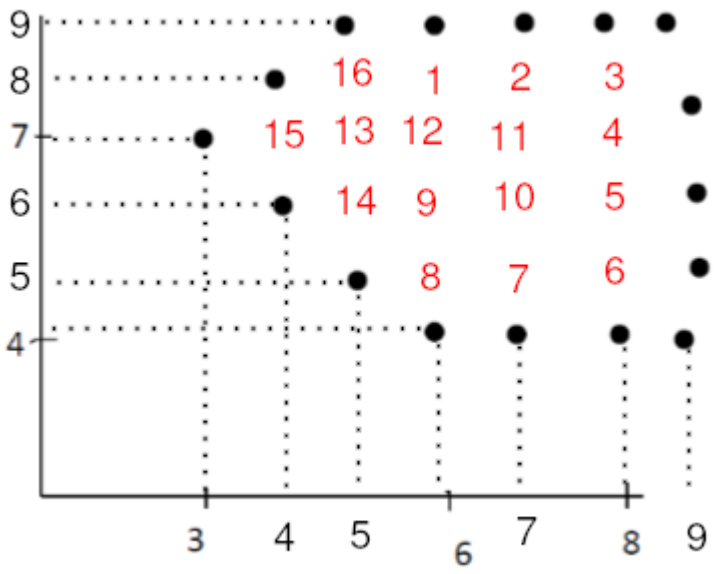
a) O algoritmo Boundary Fill tem a vantagem de ser muito simples de usar, porque ele preenche todas as cores que não são a cor de uma borda. Assim, é fácil definir uma cor que é a cor do limite (ou uma lista de cores, se for mais de uma) e colorir tudo menos ela. Esse algoritmo tem o problema de que se o recorte tiver um vazamento, o algoritmo vai vazar para fora da fronteira e colorir todo o resto, além de ele destruir tudo que está dentro do polígono que não seja de uma cor da borda. Outra desvantagem é que, dependendo da figura e do tipo de conectividade, alguns pixels podem ser ignorados e não ser coloridos.

b) O algoritmo Flood Fill tem a vantagem de também ser muito fácil de implementar também, e ele resolve o problema do Flood Fill de destruir tudo que não seja uma cor de borda. Nesse caso, é definida uma cor de preenchimento, e todos os pixels a partir de um pixel de origem que sejam da mesma cor que essa cor de preenchimento são coloridos. Mas o problema de vazamento de cor se mantém. Se existir um buraco na área de fronteira, o resto todo da figura que for dessa cor de preenchimento e for conectado ao pixel de origem vai continuar sendo colorido. Aqui também temos a desvantagem de que, dependendo da figura e do tipo de conectividade, alguns pixels podem ser ignorados e não ser coloridos.

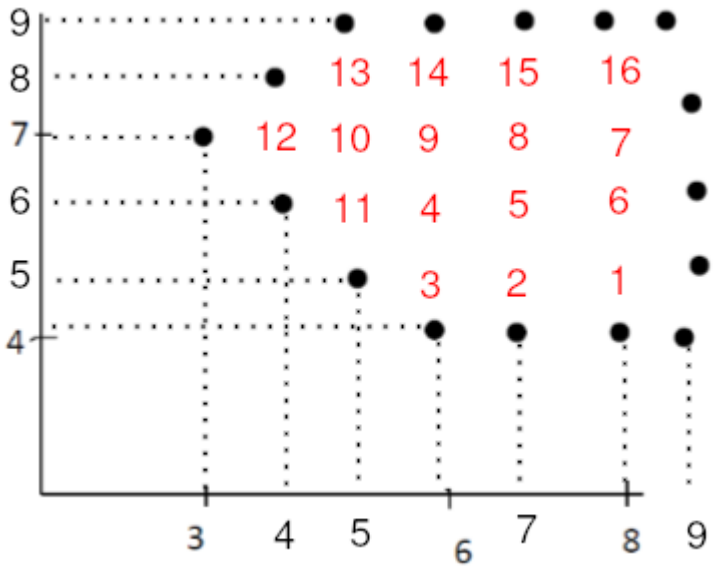
c) O algoritmo scanline é mais inteligente no sentido de que ele processa uma linha por vez. Sua grande vantagem é que ele consegue evitar recursões - poupando performance e recursos - e cada vértice é lido apenas uma vez. Os pontos são coloridos a partir da soma dos valores de vértices - simples e duplos - que são designados para cada vértice, e durante a varredura vão dizer se um pixel deve ser colorido ou não. A grande desvantagem do scanline é que ele é pesado no sentido de requerer que todos os vértices e polígonos estejam desenhados antes de começar a colorir os pixels.

36 - A conectividade 4 pode ter problemas em preencher pixels que se encontram em um encontro de duas linhas formando um ângulo. Já a conectividade 8 pode ter problemas com vazamento para fora da figura, onde uma área externa à desejada é preenchida.

37 - a)



b)



c)

Linha 9: 5, 6, 7, 8, 9

Linha 8: 4, 9

Linha 7: 3, 9

Linha 6, 4, 9

Linha 5: 5, 9

Linha 4: 6, 7, 8, 9